Cairo University
Electronics and Communications Engineering

Computer Arithmetic, Lecture 9: Division

Hossam A. H. Fahmy

What is division and why do we care about it?

The division is defined as

$$a = bq + R$$

where a is the *dividend*, b is the *divisor*, q is the *quotient*, and R is the *remainder*. It is more complicated than addition, subtraction, or multiplication. Hence,

1. How much hardware do we devote to division?
2. Can we share that hardware between the division and other functions such as the multiplication or the square root? Should we share it?
3. How long is the time taken by division?

Some answers and other issues

- Depending on the algorithm used a division can have between 3 and 20 times the delay of a multiplication.
- Although the divide instruction is not very frequent, if the division unit is too slow it can cause a large performance degradation for the whole system.

1. How frequently do we encounter a division?
2. What is "slow"?

Frequency of operations

Here are the frequency of floating point instructions on the MIPS architecture for five programs of the benchmark SPECfp2000.

FP instruction	applu	art	equake	lucas	swim	average	% to FP instructions
Load	11.4	12.0	19.7	16.2	16.8	15.22	0.35
Store	4.2	4.5	2.7	18.2	5.0	6.92	0.16
add	2.3	4.5	9.8	8.2	9.0	6.76	0.16
subtract	2.9	0	1.3	7.6	4.7	3.30	0.08
multiply	8.6	4.1	12.9	9.4	6.9	8.38	0.19
divide	0.3	0.6	0.5	0	0.3	0.34	0.01
other	0.7	2.4	1.8	5.0	0.9	2.16	0.05

In an older study, the add and subtract instructions are about 40% of the floating point instructions. The multiply is 37%, the divide is 3% and the square root is 0.33%. The moving instructions in that older study are about 10%.

Why is there a difference?

The most important factor seems to be the compiler used and the kind of optimizations it made.

1. Optimizations decrease the number of Load and Store and increase the frequency of arithmetic operations (including divide).
2. Optimizations move the code and handle the dependencies better to minimize the *stall time* of the instructions waiting for their inputs.

What can we do then?

It is clear that:

- Add/Sub unit is the most important,
- Mul is second,
- we cannot slack too much on Div and Sqrt, most importantly in multiple issue processors.
  - The higher the issue rate, the higher CPI delay due to Div.

$\Rightarrow$ We should attempt to equalize the *weighted delay* of the different instructions.

Hardware sharing between the different instructions is possible as long as we can manage the conflict for resources.

How do we divide?

Three basic approaches are in use:

1. Table lookup.
2. Subtractive methods: (digit recurrence, converge linearly)
   (a) Restoring
   (b) Non-restoring
   (c) Shift over 0's
   (d) Brute force (multiple subtractors)
   (e) SRT
   (f) High radix
3. Multiplicative methods: (converge quadratically)
   (a) Newton-Raphson
   (b) Series expansion
   (c) Higher order series

Restoring division

Similar to our manual division. For a divisor b with bits up to the 2^{n-1} position, we get the quotient bits one at a time.

1. The partial remainder at the start is $R_n = a$ and we start our counter $i = n - 1$.
2. Then, $R_i = R_{i+1} - 2^i \times b$. If $R_i \geq 0$ (i.e. $R_{i+1} \geq 2^i b$) then $q_i = 1$. Otherwise, $q_i = 0$.
3. The correct remainder is $R_i = R_{i+1} - q_i \times 2^i \times b$. For $q_i = 0$, we must restore $R_i = R_{i+1}$.
4. Decrement i and continue the loop till all the bits are checked.

Notes:

- For the restoration, we save the previous R or do an addition.
- We need to shift the divisor and have an adder as wide as the dividend.

Integer example

Example 1 Let us illustrate the restoring division process for a binary division of 29/3:

$$
\begin{array}{lll}
29-3\times 2^4=-19 & & q_4=1\\
-19+3\times 2^4=+29 & \text{restore} & \boxed{q_4=0}\\
29-3\times 2^3=+5 & & \boxed{q_3=1}\\
+5-3\times 2^2=-7 & & q_2=1\\
-7+3\times 2^2=+5 & \text{restore} & \boxed{q_2=0}\\
+5-3\times 2^1=-1 & & q_1=1\\
-1+3\times 2^1=+5 & \text{restore} & \boxed{q_1=0}\\
+5-3\times 2^0=+2 & & \boxed{q_0=1}
\end{array}
$$

FP algorithm

Usualy division is done on FP numbers and not on integers. Here, we use:

1. The partial remainder at the start is $R_1=\frac{a}{2}$ and we start our counter $i=0$.
2. Then, $R_i=2\times R_{i+1}-b$. If $R_i\geq 0$ (i.e. $2\times R_{i+1}\geq b$) then $q_i=1$. Otherwise, $q_i=0$.
3. The correct remainder is $R_i=2\times R_{i+1}-q_i\times b$. For $q_i=0$, we must restore $R_i=2\times R_{i+1}$.
4. Decrement i and continue the loop till all the bits are checked.

FP example, restoring

Example 2 For $a=01.1000110$ and $b=01.0010000$ then $C(b)=10.1110000$

$$
\begin{array}{lrll}
R_0= & a-b= & 01.1000110+10.1110000 & =00.0110110 \quad q_0=1\\
R_{\bar{1}}= & 2R_0-b= & 00.1101100+10.1110000 & =11.1011100 \quad q_{\bar{1}}=0\\
 & & \text{restore } R_{\bar{1}}=00.1101100 & \\
R_{\bar{2}}= & 2R_{\bar{1}}-b= & 01.1011000+10.1110000 & =00.1001000 \quad q_{\bar{2}}=1\\
R_{\bar{3}}= & 2R_{\bar{2}}-b= & 01.0010000+10.1110000 & =00.0000000 \quad q_{\bar{3}}=1
\end{array}
$$

We get $01.1000110=01.0010000\times 1.011+0$.

Can we skip the restoration step?

The issues of subtractive division

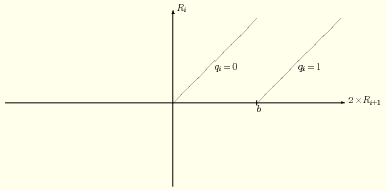
In the general case (base β), we are doing three steps. We

1. compare $\beta\times R_{i+1}$ with b and its multiples to find q_i,
2. generate the correct multiple $q_i\times b$, and
3. subtract to get $R_i=\beta\times R_{i+1}-q_i\times b$.

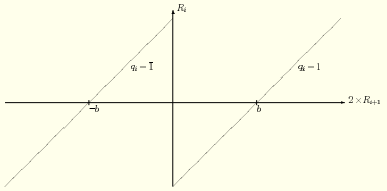
If we make each of these steps simpler we get a faster division.

From restoring to non-restoring division

In the case of binary restoring division, we have this diagram:



If we do not restore when R_i becomes negative, then in the following step we must set $q_{i-1} = \bar{1}$. In effect, $q_i q_{i-1} = 1\bar{1} = 01$ which is equivalent to restoring and then subtracting in the following lower significance. In the non-restoring case we get this diagram:



Non-restoring algorithm

1. The partial remainder at the start is $R_1 = \frac{a}{2}$ and we start our counter $i = 0$.
2. If $2 \times R_{i+1} \geq 0$ then $q_i = 1$. Otherwise, $q_i = -1$.
3. Then, $R_i = 2 \times R_{i+1} - q_i \times b$.
4. Decrement i and continue the loop till all the bits are checked.

FP example, non-restoring

Example 3 For $a = 01.1000110$ and $b = 01.0010000$ then $C(b) = 10.1110000$

$$
\begin{array}{llll}
R_0 = & a - b = & 01.1000110 + 10.1110000 & = 00.0110110 \quad q_0 = 1 \\
R_{\bar{1}} = & 2R_0 - b = & 00.1101100 + 10.1110000 & = 11.1011100 \quad q_{\bar{1}} = 1 \\
R_{\bar{2}} = & 2R_{\bar{1}} + b = & 11.0111000 + 01.0010000 & = 00.1001000 \quad q_{\bar{2}} = -1 \\
R_{\bar{3}} = & 2R_{\bar{2}} - b = & 01.0010000 + 10.1110000 & = 00.0000000 \quad q_{\bar{3}} = 1
\end{array}
$$

We get $q = 1.1\bar{1}1 = 1.011$ as before.

From $\{\bar{1}, 1\} \rightarrow \{0, 1\}$

1. Shift left by one bit position and put 1 as the new *LSB*.
2. Convert every $\bar{1}$ to 0 and leave each 1 as is.
3. Invert the *MSB*(the sign bit of the two's complement).

For the previous example we have

Input		1.	1	$\bar{1}$	1
Shift	1	1.	$\bar{1}$	1	1
Convert	1	1.	0	1	1
Invert	0	1.	0	1	1 $\leftarrow$ Output

(Hint for the proof: use $p_i = \frac{q_i + 1}{2}$ or $q_i = 2p_i - 1$.)

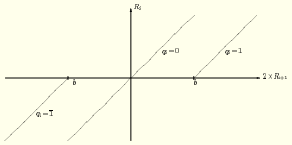
Skiping steps to improve the speed

Example 4 For $a = 1.0100101$ and $b = 1.010000$, we get:

$$R_0 = \quad a - b = \quad 01.0100101 + 10.110000 \quad = 00.0000101 \quad q_0 = 1$$

The following bits in the final quotient are all zeros till we shift enough to get $2R_{\bar{4}} = 1.01$ and $q_{\bar{5}} = 1$.

- Instead of an addition in every cycle, we shift over groups of zeros or groups of ones.
- This gives a variable latency algorithm that is fast on the average.



Getting faster by brute force

1. Form several multiples of the divisor, for example: $(1.00)b$, $(1.01)_2 b = (\frac{5}{4})_{10} b$, $(1.10)_2 b = (\frac{3}{2})_{10} b$, and $(1.11)_2 b = (\frac{7}{4})_{10} b$.
2. Choose the one that is the closest to the remainder.
3. Subtract to get the next remainder and record 3 bits for the quotient.

- Either use several multipliers by those fixed constants with several hardware comparators, or
- use a lookup table with the first few bits of the divisor and the partial remainder as the index to get the quotient bits.

The larger the hardware resources you put, the higher the radix you can use, and the faster the operation becomes.

Redundancy gives you more

In these steps:

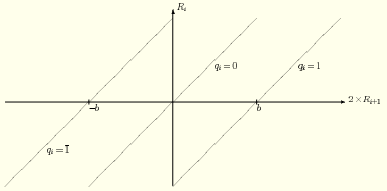
1. compare $\beta \times R_{i+1}$ with b and its multiples to find q_i,
2. generate the correct multiple $q_i \times b$, and
3. subtract to get $R_i = \beta \times R_{i+1} - q_i \times b$.

- use a larger radix to get more quotient bits each iteration,
- simplify the comparison,
- simplify the generation of $q_i b$, and
- use redundancy to make the subtraction faster.

Simple SRT

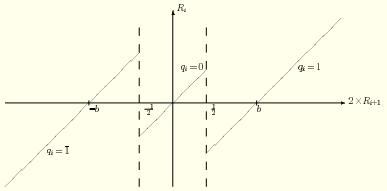
Sweeney of IBM, Robertson of University of Illinois, and Tocher of Imperial College independently proposed similar procedures that we now call SRT.

- The quotient is represented by a redundant set.
- A higher radix is used sometimes.
- Inspect only a few bits to decide on the next quotient digit.



Simplify the comparison

Since it is possible to use either digit in the overlapping regions, we choose the border to make the comparison easier. For example, compare $2R_{i+1} \geq 00.1$ by looking at the first three bits only.



Make the subtraction faster, use CSAs

- Form only the needed most significant bits in R_i for the comparison.
- Leave the rest as two bit vectors (sum and carry) to be added in the next iteration to $-q_i b$.

Go for a higher radix

- We want more than one bit per iteration.
- Radix-4 gives two bits.
- We can use $\{-3, -2, -1, 0, 1, 2, 3\}$ or $\{-2, -1, 0, 1, 2\}$ or something else.
- The trade-off is between the amount of overlap which simplifies the selection and the generation of $q_i b$.

Conclusion of subtractive division

- For high speed, SRT is the most widely used.
- SRT with radix-8 is also possible at the price of hard multiples.
- Subtractive algorithms provide both the quotient and the remainder as required by the IEEE standard.
- Subtractive algorithms converge linearly.