Space Efficient Division on FPGAs

Donald G. Bailey

Institute of Information Sciences and Technology, Massey University, Palmerston North
Email: D.G.Bailey@massey.ac.nz

Abstract

Division algorithms are discussed in the context of implementing image processing algorithms on FPGAs. The speed requirements of image processing are modest, but throughput is important, with one division having to be completed per clock cycle. Several division algorithms are compared, with the non-restoring algorithms being both the smallest and fastest of the basic methods. If necessary, the algorithms may be easily pipelined to meet the throughput requirements.

Keywords: Division, FGPA, image processing, restoring division, non-restoring division, SRT division

1 Introduction

Of the four basic arithmetic operations: addition, subtraction, multiplication, and division, division is the hardest to implement in hardware. One of the main reasons for this is that while addition, subtraction and multiplication are well defined and give exact answers, division is less so. The result of a division between integers (or even between floating point numbers with finite precision) will in general be a rational number, which in many cases cannot be represented exactly in binary with a fixed number of bits. This leads either to an approximate answer, or to a second definition of division: integer division.

Integer division considers both the dividend and divisor to be integers, and expresses the result uniquely as a quotient and remainder:

$$Dividend = Quotient \times Divisor + Remainder, \quad (1)$$

where the quotient is also an integer, and the remainder satisfies

$$0 \le Remainder < Divisor. \quad (2)$$

Within an image processing context, division occurs within several common image processing operations: contrast expansion, intensity normalisation, contrast ratio calculation, colour conversion (calculating the hue and saturation) are a few.

For contrast expansion, pixels within a limited range are expanded to fill the available range from 0 to 255, i.e.

$$Out(x,y) = 255 * \frac{Input(x,y) - P_{\min}}{P_{\max} - P_{\min}}. \quad (3)$$

If the limits, P_{min} and P_{max}, are known in advance, then the contrast expansion may be implemented using a multiplication, or even a lookup table. However, if the limits are dynamic (they are derived at runtime) then division is required to calculate the output, or to determine the multiplication factor or lookup table values.

Intensity normalisation scales an input image according to a reference image (for example of a plain white background):

$$Output(x,y) = \frac{Input(x,y)}{Reference(x,y)} * Scale. \quad (4)$$

This cannot be efficiently implemented using a multiplication or lookup table because the factor is different for each pixel processed.

The contrast ratio between two pixels is given by:

$$Contrast = \frac{|p_1 - p_2|}{p_1 + p_2}. \quad (5)$$

If the contrast ratio between two images is calculated on a pixel by pixel basis, then a division must be performed for each pixel.

To obtain the hue [1] from an RGB image the colour wheel is split into 6 sectors depending on the relative magnitudes of the red, green, and blue components. The hue value is then given from a ratio of components within each sector. For example, in sector 1, $B \le G \le R$ and the hue is given by

$$Hue = 60 * \frac{R - G}{R - B}. \quad (6)$$

There are similar equations for each of the 5 other sectors. The saturation value also requires a division:

$$Saturation = \frac{\max(R,G,B) - \min(R,G,B)}{\max(R,G,B)}. \quad (7)$$

When implemented on a PC, microcomputer, or DSP, modern devices usually have some support for division, even if it is at the compiler level. However, when implementing on hardware, for example on an FPGA, there is usually little or no support for division. While basic division algorithms are straight

forward, their hardware implementation is expensive in terms of both propagation delay and hardware resources.

This paper considers a number of algorithms for implementing division on an FPGA. The primary focus will be on a single quadrant divider (positive dividend and divisor) although extensions will be discussed at the end of this paper.

In the context of real-time image processing, we must be able to process 1 pixel each clock cycle. This means that the whole division operation must complete within a single clock cycle, or the division algorithm must be able to be pipelined (see Figure 1) so that a new division can be started each clock cycle. As long as an algorithm can be readily pipelined, the actual propagation delay (or latency) is not important, so algorithms that require fewer resources are preferred. Note that this is in contrast with the usual goal in selecting a division algorithm of trying to minimise the total propagation delay.

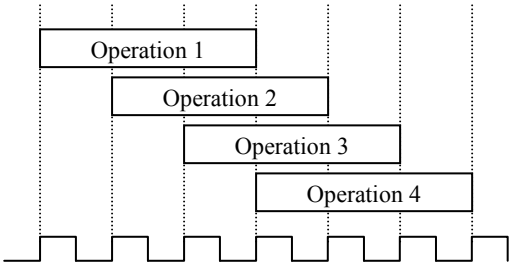

Figure 1: While the latency of a complete operation may be several clock cycles (3 here), the operation is pipelined so that a new operation begins each clock.

2 Division Algorithms

In all of the examples in the previous section, both the dividend and divisor are 8 bit integers. A second key characteristic is that the dividend is always less than (or equal to) the divisor, so the result is always a fraction. This fraction may then be scaled by a constant to obtain an integer result. Alternatively, the scaling may be performed prior to the division, enabling a simple integer division to be performed. The final 8 bit result is an approximation because any remainder is discarded.

2.1 Standard Binary Long Division

The standard long division algorithm is represented diagrammatically in Figure 2. It is an iterative algorithm, where at each iteration one bit of the quotient is determined. If the divisor is less than or equal to the shifted partial remainder from the previous iteration, the quotient bit is 1, otherwise it is 0. If it is 1, then divisor is subtracted from the partial remainder to get the new partial remainder.

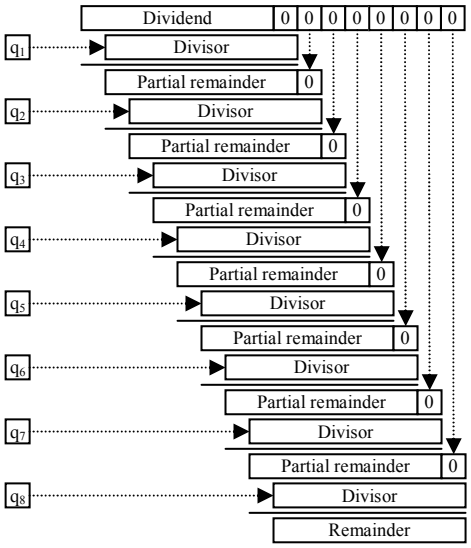

Figure 2: Standard long division.

Let V be the dividend, D be the divisor, q_i be the quotient bit from the ith iteration, and R_i be the ith partial remainder. Then

$$q_i = \begin{cases} 0 & \text{if } 2R_{i-1} < D \\ 1 & \text{if } 2R_{i-1} \geq D \end{cases} \qquad (8)$$
$$R_i = 2R_{i-1} - q_i \times D,$$

where the initial partial remainder is $R_0 = V$.

Note that equation (8) assumes that the initial dividend is less than the divisor. In the case that they are equal, the partial remainder will also equal the divisor after each iteration, and all of the quotient bits will be 1s. This maximum output value is the desired behaviour, so no special tests need to be made of this case. If, however, the dividend is greater than the divisor, significant bits may be lost as a result of the shifting, and an invalid output obtained. In the examples given, this should not happen, but in general there should be a further test to determine such an overflow condition. This may be implemented as a comparison between the divisor and dividend in parallel with the first iteration.

In the application described, the remainder is discarded, so the final subtraction is not actually required.

In performing the iteration of equation (8), the multiplication may be performed by ANDing q_i with D. The comparison to determine q_i must also be completed before the subtraction to determine the new remainder. Thus the critical path includes both the comparison and the subtraction. A speedup may be obtained by performing these in parallel, and multiplexing the previous remainder or the subtraction:

$$q_i = \begin{cases} 0 & \text{if} \quad 2R_{i-1} < D \\ 1 & \text{if} \quad 2R_{i-1} \geq D \end{cases}$$
$$R_i = \begin{cases} 2R_{i-1} & \text{if} \quad q_i = 0 \\ 2R_{i-1} - D & \text{if} \quad q_i = 1. \end{cases} \tag{9}$$

A further optimisation may be obtained by observing that the comparison to determine q_i may be determined by performing the same subtraction that is used to determine the next partial remainder, and testing the sign bit:

$$R_i' = 2R_{i-1} - D$$
$$q_i = \begin{cases} 0 & \text{if} \quad R_i' < 0 \\ 1 & \text{if} \quad R_i' \geq 0 \end{cases} \tag{10}$$
$$R_i = \begin{cases} 2R_{i-1} & \text{if} \quad q_i = 0 \\ R_i' & \text{if} \quad q_i = 1. \end{cases}$$

This algorithm is sometimes called "restoring division" because a test subtraction is made, and if the result goes negative, the original partial remainder is restored.

2.2 Non-restoring Division

Non-restoring arithmetic does not restore the result if the subtraction goes negative. Instead, it performs an addition in the next iteration. In this way the partial remainder will be kept between –D and D. The addition is equivalent to a bit weighting of $q_i = -1$ in the previous algorithm. Since $q_i \in \{-1, 1\}$, q_i should be referred to as a quotient digit rather than a quotient bit.

The test whether we add or subtract is also simpler, since we only need to test the sign of the partial remainder so far:

$$q_i = \begin{cases} -1 & \text{if} \quad R_{i-1} < 0 \\ 1 & \text{if} \quad R_{i-1} \geq 0 \end{cases}$$
$$R_i = \begin{cases} 2R_{i-1} + D & \text{if} \quad q_i = -1 \\ 2R_{i-1} - D & \text{if} \quad q_i = 1. \end{cases} \tag{11}$$

It is also necessary to convert the -1 and 1 weightings to conventional binary digits at the end. If a 0 bit is used to represent the -1 digit, then the obtained quotient is $Q_O = q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8$. To convert this to binary, we need to calculate

$$\begin{aligned} Q &= Q_O - \overline{Q_O} \\ &= Q_O + (Q_O + 1) \\ &= 2Q_O + 1, \end{aligned} \tag{12}$$

where the subtraction is replaced by the addition of a two's complement. Therefore a 1 is simply appended to the bits we already have.

Also observe that for positive dividends the first iteration will always be a subtraction, therefore the quotient bit for this is not actually needed (unless the dividend is negative). Therefore, the iteration may be initialised by setting $R_0 = 2V - D$. The next 8 iterations will give the 8 output bits, and again the addition or subtraction for the final iteration is not needed (since the remainder is discarded).

The advantage of using non-restoring arithmetic over the standard restoring division is that a test subtraction is not required; the sign bit determines whether an addition or subtraction is used. The disadvantage, though, is that an extra bit must be maintained in the partial remainder to keep track of the sign.

One limitation of the iteration expressed in equation (11) is that separate hardware is used to perform the addition and subtraction, and the results are multiplexed to give the remainder. This may be simplified to a single addition, and multiplexing whether *D* or *–D* is added:

$$q_i = \begin{cases} -1 & \text{if} \quad R_{i-1} < 0 \\ 1 & \text{if} \quad R_{i-1} \geq 0 \end{cases}$$
$$R_i = 2R_{i-1} + \begin{cases} D & \text{if} \quad q_i = -1 \\ -D & \text{if} \quad q_i = 1. \end{cases} \tag{13}$$

A further optimisation is to replace the –D with the two's complement of D:

$$q_i = \begin{cases} -1 & \text{if} \quad R_{i-1} < 0 \\ 1 & \text{if} \quad R_{i-1} \geq 0 \end{cases}$$
$$R_i = 2R_{i-1} + \begin{cases} D & \text{if} \quad q_i = -1 \\ \overline{D} + 1 & \text{if} \quad q_i = 1. \end{cases} \tag{14}$$

The addition of the 1 as part of the 2's complement does not actually require additional logic because the $2R_{i-1}$ will leave the least significant bit as 0. The 1 can be inserted instead if $q_i = 1$.

2.3 Higher Radix and SRT Division

The major limitation to the speed of the previous calculations is the propagation delay of the carry rippling through the whole calculation with each iteration.

The most obvious solution to this is to reduce the number of iterations, by determining more than 1 quotient bit per iteration. This effectively means that rather than performing binary division, base 4 or 8 are used (powers of two for computational simplicity). In general, the iteration for a radix-α algorithm is:

$$R_i = \alpha R_{i-1} - q_i D. \tag{15}$$

A radix-4 division, for example, only requires half the number of iterations, and therefore has a significantly shorter propagation delay. For a radix-4 division, the quotient digits $q_i \in \{0, 1, 2, 3\}$ would be used with a restoring division algorithm, or $q_i \in \{-3, -1, 1, 3\}$ with a non-restoring division. Extra comparisons are required with high-radix (>2) algorithms to determine

the digit, but these can be implemented in parallel and therefore would take no additional time.

Another approach to obtain a faster division is to reduce the propagation delay by reducing the length of the critical path. The SRT division algorithm (named after Sweeney, Robertson [2], and Tocher [3] who all developed it independently at about the same time) achieves this by introducing redundancy into the digit set. For example, radix-2 SRT division uses the digits $q_i \in \{-1,0,1\}$, and radix-4 SRT uses digits $q_i \in \{-2,-1,0,1,2\}$. This gives an overlap in selecting between adjacent digits. This overlap means that the quotient digit may be determined from an approximation of the partial remainder, and does not need to wait for the carry to propagate fully. This may be readily achieved for example using a carry save adder (where the carry is not propagated, but is added in when the next sum is calculated). A disadvantage of the SRT method is that the divisor must be normalised (so that the MSB is 1) prior to starting the division. This normalisation requires additional hardware. The carry save adders used by SRT do not map onto the FPGA resources as efficiently. Extra time (and logic) is also required at the end to convert the quotient digits back to standard binary.

Once normalised, only the top 2 bits need to be examined to determine the quotient digit. Higher radix SRT algorithms are even faster because they calculate multiple quotient bits at a time, and require fewer iterations. This comes at the cost of increased hardware however, to determine the quotient digit. Lookup tables are commonly used, and these tables can get very large for high radix division. For this reason, most hardware implementations are restricted to radix-4 SRT, and are implemented serially so that a single lookup table can be used for all of the iterations.

Due to the complexity of division, there are many other advanced algorithms (for example [4,5]), and this is still an active research area.

3 Performance Measurements

We have been investigating mapping image processing algorithms onto FPGAs [6-10]. A lot of our basic algorithms have been developed on an RC100 board produced by Celoxica, which uses a Spartan-II FPGA. An RC300 board, using a Virtex-II FPGA is used for our more advanced work.

The divisions were implemented as macro expressions within the Handel-C language [11], compiled to an EDIF file within the Celoxica DK4.0 environment, and the EDIF file mapped onto the respective FPGAs using Xilinx ISE version 6.1.03.

The divisions defined by equations (8), (9), (10), (11), (13) and (14) were compared with Handel-C's built-in integer division. The Handel-C divide operator was more general, and did not assume that $V \le D$. The restrictions within Handel-C also meant that a 16 bit division was necessary to obtain the 8 fractional bits. For the other methods, the iterations were unrolled so that the complete division was performed in a single clock cycle.

Higher radix, and SRT based algorithms were not implemented, because they use significantly more resources. They would be fine implemented as iterative algorithms, but when the iterations are unrolled the hardware duplication becomes expensive.

Finally, the algorithm of equation (14) was also implemented using a 2-stage pipeline. This gives a 2 clock cycle latency, while allowing one division to be completed every clock cycle. The pipeline was implemented by registering the output of the 4th subtraction. The register was 20 bits wide: 9 bits of the partial remainder, the 3 quotient bits that have already been calculated, and the 8 divisor bits (because the divisor may be different for each pixel).

The resources required, and the maximum operating frequency are shown in Table 1 for each of the algorithms compared.

Table 1: Performance comparison of the different algorithms. The resource utilisation is indicated by the number of LUTs required, and the maximum frequency (F_{max}) is given in MHz.

	Equation	LUTs	F_{max} Spartan-II	F_{max} Virtex-II
Handel-C		747	7.210	10.965
Restoring algorithms	(8)	115	13.716	20.345
	(9)	108	19.243	29.719
	(10)	114	16.336	27.057
Non-restoring algorithms	(11)	144	24.175	40.073
	(13)	72	21.835	33.934
	(14)	65	21.792	32.756
	Pipelined	66	37.806	63.558

4 Discussion

The built-in Handel-C division was the slowest and used the most resources of all the methods. This is not surprising since a 16 bit division had to be used to obtain an 8 bit result. While the comparison is not really fair, this would be the default available without developing our own. The speed of the built-in division can be improved by pipelining. This may be achieved by registering the output with a series registers (depending on the number of pipeline stages), and using the "Enable retiming" compiler option to optimise the position of the registers within the operation [12]. However, pipelining was not considered because of the already significant resources required by the operation.

The standard restoring division algorithms were the next slowest. As expected, the speed improvement from equation (8) to equation (9) was significant. However the algorithm of equation (10) was a surprise. Using the subtraction to perform the comparison, and reusing the difference as a multiplexer input was expected to significantly reduce the hardware required. Instead, it used more hardware, and incurred a speed penalty as well. Without investigating the resultant netlist, it is surmised that separate LUTs are being used for performing the addition and for the multiplexing. While these could be combined if implemented in a lower level language such as VHDL, from within Handel-C such a low level optimisation can't be forced, and relies on Handel-C's optimisation. The similar resource requirements of equations (9) and (10) imply that Handel-C may already be using a subtraction to perform the comparison, and reusing the hardware for the difference as well.

The non-restoring division algorithms were all faster than the restoring algorithms. This is because the result of the previous iteration is used directly to select either the addition or subtraction operation, and there is no further overhead The algorithm of equation (11) is clearly building separate hardware for the addition and subtraction, because when the multiplexer is moved in equation (13) only half the resources are required. The implementation of equation (13) is slower because with equation (11) the addition and subtraction for the next stage can begin before the carry propagates to the most significant bit, where it is used to select either the addition or subtraction. So while equation (13) reduces the resources, to do so it lengthens the critical path.

From the number of LUTs required to implement equation (13), the multiplexer and addition are clearly using the same LUT. Replacing –D with an explicit 2's complement in equation (14) reduces the resource requirement further. This implies that Handel-C was building separate hardware to calculate –D. The slight speed hit from equation (14) results from feeding the sign bit back around to the LSB for the next iteration, whereas this was implicit in the calculation of –D in equation (13).

As expected, the 2 stage pipelined algorithm is almost twice as fast as the unpipelined version. It does not achieve exactly two times speedup because the registers may not be exactly in the middle of the critical path, and the registers also have an associated setup time. The pipelined algorithm only used 17 flip-flops, rather than the expected 20. This is because the 3 quotient bits that had already been calculated were delayed using shift registers rather than flip-flops. However this should result in 3 additional LUTs rather than the 1 observed. Handel-C and/or the ISE mapper must be making further optimisations as a result of pipelining.

The restoring algorithms only work in a single quadrant, that is with a positive dividend and divisor. When working with a negative dividend or divisor, a two's complement is required to make the respective number positive, and a further two's complement is required at the end if the result is negative. These would require additional hardware, and also lower the maximum operating speed. Another advantage of the non-restoring algorithms is that they operate implicitly in 2 quadrants. They work equally well with a signed dividend, with the first iteration giving the sign bit. A relatively trivial change is required to operate in 4 quadrants, that is with a signed divisor as well.

5 Conclusions

In a real-time image processing application, a divider operating on every pixel must have a throughput of a single clock cycle. This means that the iterations used to obtain each quotient bit, or digit, must be unrolled. If the propagation delay is too long, the division can be pipelined by inserting registers partway through the calculation, and spreading the operation over two or more clock cycles. In image processing applications, the actual latency is not particularly important, so the goal of this paper has been to minimise the logic resources required by the implementation.

In the application considered, the division operator built into Handel-C is both resource expensive and slow. It is better to use a custom division operation. Restoring algorithms are slower than non-restoring algorithms, and a properly implemented non-restoring algorithm uses the least resources. A further advantage of the non-restoring algorithm is that it works without change with signed dividends, and only a relatively trivial change is required for it to work with a signed divisor.

6 Acknowledgements

The author would like to gratefully acknowledge the Celoxica University Programme for generously providing the DK4 Design Suite, and the Xilinx University Programme for providing the ISE Foundation.

7 References

[1] J.D. Foley and A.V. Dam, *Fundamentals of Interactive Computer Graphics*. Reading, Massachusetts: Addison-Wesley (1982).

[2] J.E. Robertson, "A new class of digital division methods", *IRE Transactions on Electronic Computers*, EC-7: 218-222 (1958).

[3] K.D. Tocher, "Techniques of multiplication and division for automatic binary divider", *Quarterly Journal of Mechanics and Applied Mathematics*, 11:(3) 364-384 (1958).

[4] K. Tatas, D.J. Soudris, D. Siomos, M. Dasygenis, and A. Thanailakis, "A novel division algorithm for parallel and sequential processing", in *9th International Conference on Electronics, Circuits and Systems*, 2: 553- 556 (2002).

[5] R.K.L. Trummer, *A High-Performance Data-Dependent Hardware Integer Divider*. Masters Thesis. Salzburg. (2005).

[6] C.T. Johnston, D.G. Bailey, and K.T. Gribbon, "Optimisation of a Colour Segmentation and Tracking Algorithm for Real-time FPGA Implementation", in *Image and Vision Computing New Zealand*, Dunedin, New Zealand, 422-427 (28-29 November, 2005).

[7] C.T. Johnston, K.T. Gribbon, and D.G. Bailey, "FPGA Based Remote Object Tracking for Real-time Control", in *International Conference on Sensing Technology*, Palmerston North, New Zealand, 66-71 (21-23 November, 2005).

[8] K. Gribbon, D.G. Bailey, and C.T. Johnston, "Colour Edge Enhancement", in *Image and Vision Computing New Zealand 2004*, Akaroa, New Zealand, 291-296 (21-23 November, 2004).

[9] C.T. Johnston, K.T. Gribbon, and D.G. Bailey, "Implementing Image Processing Algorithms on FPGAs", in *Eleventh Electronics New Zealand Conference (ENZCon '04)*, Palmerston North, New Zealand, 118-123 (15-16 November, 2004).

[10] K.T. Gribbon, C.T. Johnston, and D.G. Bailey, "A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation", in *Image and Vision Computing New Zealand 2003*, Palmerston North, New Zealand, 408-413 (26-28 November, 2003).

[11] Celoxica, *DK4 - Handel-C Language Reference Manual*: Celoxica Limited (2005).

[12] Celoxica, *DK4 - DK Design Suite User Guide*: Celoxica Limited (2005).

8 Appendix: Macro Expressions

The Handel-C macro expressions used to implement the different methods are given here. Compile time recursion using the select operator unrolls the iterations. The first two macro expressions provide definitions common to the different implementations.

```
macro expr u8_0 = (unsigned 8)0;
macro expr u1_0 = (unsigned 1)0;
```

Handel-C's division operator:

```
macro expr divide(V,D) =
   ((V @ u8_0)/(u8_0 @ D)) <- 8;
```

Equation (8):

```
macro expr mult(q,D) = D & (q@q@q@q@q@q@q@q@q);
macro expr div8b(R,D,i) =
   div8a( R, D, ((R@u1_0)>=(u1_0@D)), i );
macro expr div8a(R,D,q,i) =
   select( i == 8, q,
   q @ div8b( (R<<1)-mult(q,D), D, i+1 ));
macro expr divide8(V,D) = div8b( V, D, 1 );
```

Equation (9):

```
macro expr div9b(R,D,i) =
   div9a( R, D, ((R@u1_0)>=(u1_0@D)), i );
macro expr div9a(R,D,q,i) =
   select( i == 8, q,
   q @ div9b( q?(R<<1)-D:(R<<1), D, i+1 ));
macro expr divide9(V,D) = div9b( V, D, 1 );
```

Equation (10):

```
macro expr div10b(R,D,i) =
   div10a( R, D, (R@u1_0)-(u1_0@D), i );
macro expr div10a(R,D,s,i) =
   select( i==8, ~s[8],
   ~s[8] @ div10b( s[8]?(R<<1):(s<-8),
                   D, i+1 ));
macro expr divide10(V,D) = div10b( V, D, 1 );
```

Equation (11):

```
macro expr div11a(R,D,i) =
   select( i == 8, ~R[8], ~R[8] @ div11a(
        R[8]?((R<<1)+D):((R<<1)-D), D, i+1));
macro expr divide11(V,D) =
   div11a( (V@u1_0)-(u1_0@D), u1_0@D, 1 );
```

Equation (13):

```
macro expr div13a(R,D,i) =
   select( i == 8, ~R[8], ~R[8] @ div13a(
        (R<<1)+(R[8]?D:-D), D, i+1 ));
macro expr divide13(V,D) =
   div13a( (V@u1_0)-(u1_0@D), u1_0@D, 1 );
```

Equation (14):

```
macro expr div14a(R,D,i) =
   select( i == 8, ~R[8], ~R[8] @ div14a(
        (R[7:0]@~R[8])+(R[8]?D:~D), D, i+1));
macro expr divide14(V,D) =
   div14a( (V@u1_0)-(u1_0@D), u1_0@D, 1 );
```