

Fixed Point Arithmetic and LU Decomposition in ALA

Scott Greenwald

May 13, 2009

1 Introduction

What

2 Number Representation

At the time of the mid-project report, I predicted that using a mixed-number representation would make the overall implementation easier. After implementing much of the machinery required for such an approach, I decided against it in favor of a binary fixed point representation.

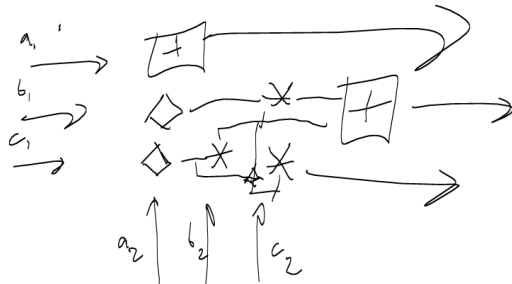
2.1 Mixed-Number Representation

The concept of mixed number representation was to use triples (a, b, c) to represent mixed numbers $a + b/c$, and operate on them with addition, multiplication, and division as follows

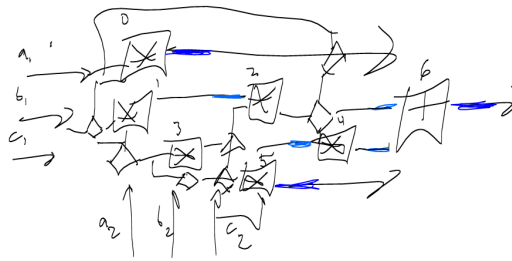
$$\begin{aligned}(a_1, b_1, c_1) + (a_2, b_2, c_2) &= (a_1 + a_2, b_1 c_2 + b_2 c_1, c_1 c_2) \\(a_1, b_1, c_1) * (a_2, b_2, c_2) &= (a_1 a_2, a_1 b_2 c_1 + a_2 b_1 c_2, c_1 c_2) \\(a_1, b_1, c_1) / (a_2, b_2, c_2) &= (0, c_2(a_1 c_1 + b_1), c_1(a_2 c_2 + b_2))\end{aligned}\tag{2.1}$$

Observe that computing the triple sum requires 2 additions and 3 multiplications, multiplication requires 6 multiplications and 1 addition, and division requires 4 multiplications and 2 additions. In the ALA implementation, each of the operators becomes a geometric object with a location on the grid. The following schematics illustrate the connectivity required.

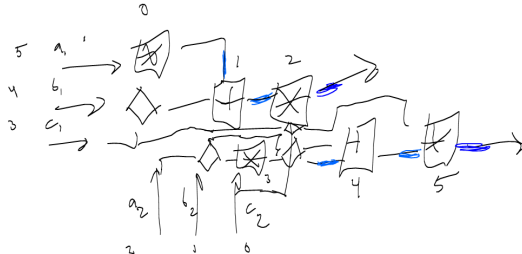
Addition



Multiplication

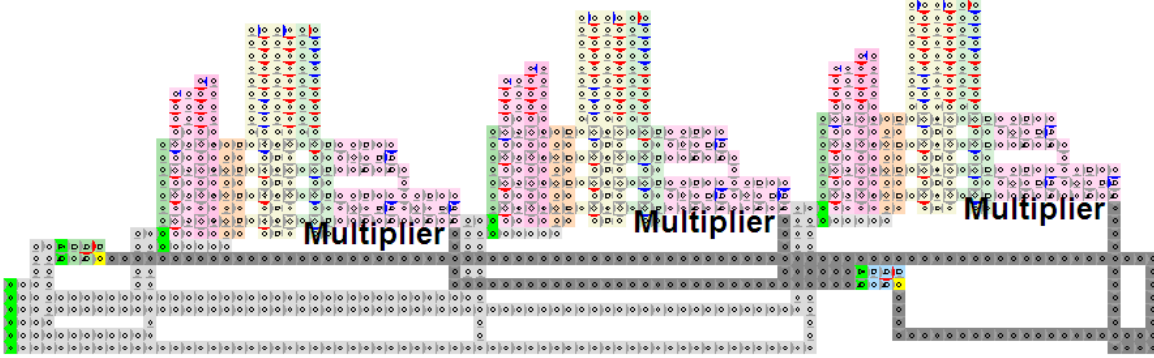


Division

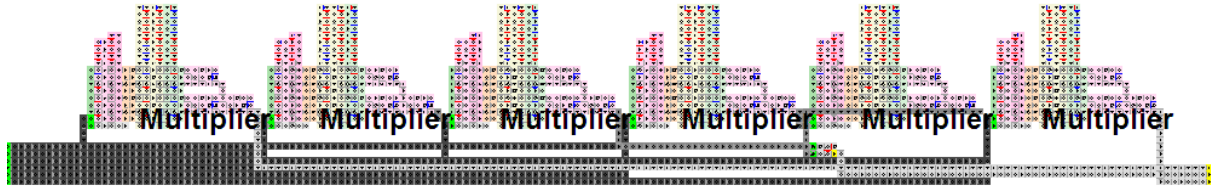


The corresponding implementations in ALA are pictured below

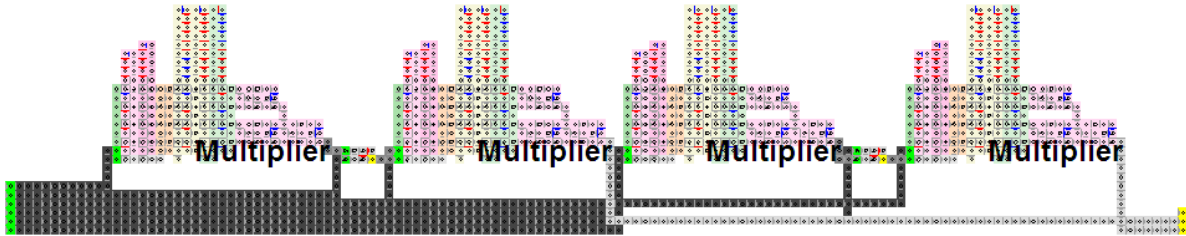
Addition



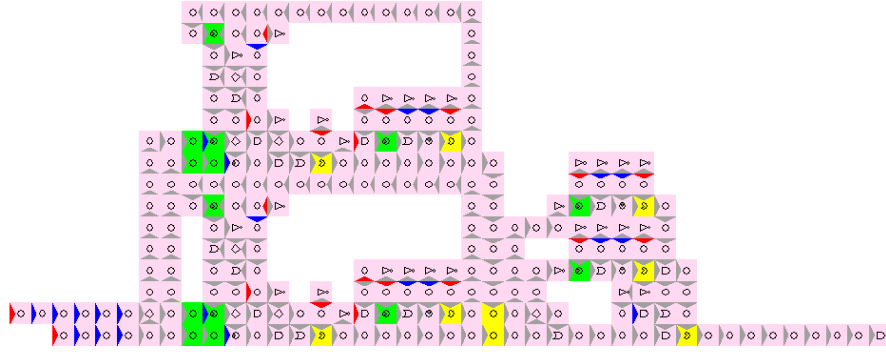
Multiplication



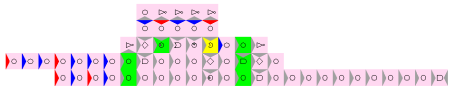
Division



By using varying numbers of multiplications and additions in computing the mixed number components a , b , and c , we end up with operands of varying length. The adder is a streaming operator of fixed size which requires that inputs arguments be of equal length, while the multiplier is hard-coded to multiply bit strings of specified lengths. In order to deal with these requirements, I devised a scheme based on “toothed tape,” which is a control sequence of zero’s terminated by a 1 which indicates the length of an argument. This can be used, for example, to pad a shorter argument to an addition to match a longer one. Pictured below is a module `maxtt` that takes two toothed tape arguments and returns a toothed tape of a length which is the maximum of the two input tapes. This operation can handle arbitrary-length tapes. See the demo file “demo-acc-add-cpt2.py”.



Using the output of `maxtt` (in fact, a version `maxtts` which leaves a “shadow” of the terminating 1 of the shorter input on the output), the following `pad` module pads data of the shorter length to match the longer one.



The above modules solve the problem of matching arguments to the addition module, but the problem of multiplication is more difficult because the length of arguments is hard-coded, we could either (a) use multipliers of fixed large length, and use `pad` to pad inputs appropriately, or (b) pain-stakingly compute the expected length of outputs at each stage and hard-code every multiplier in every row and column to be ready for the right number of bits. Both of these options are exceedingly unattractive, and therefore I turned my attention to fixed-point representation.

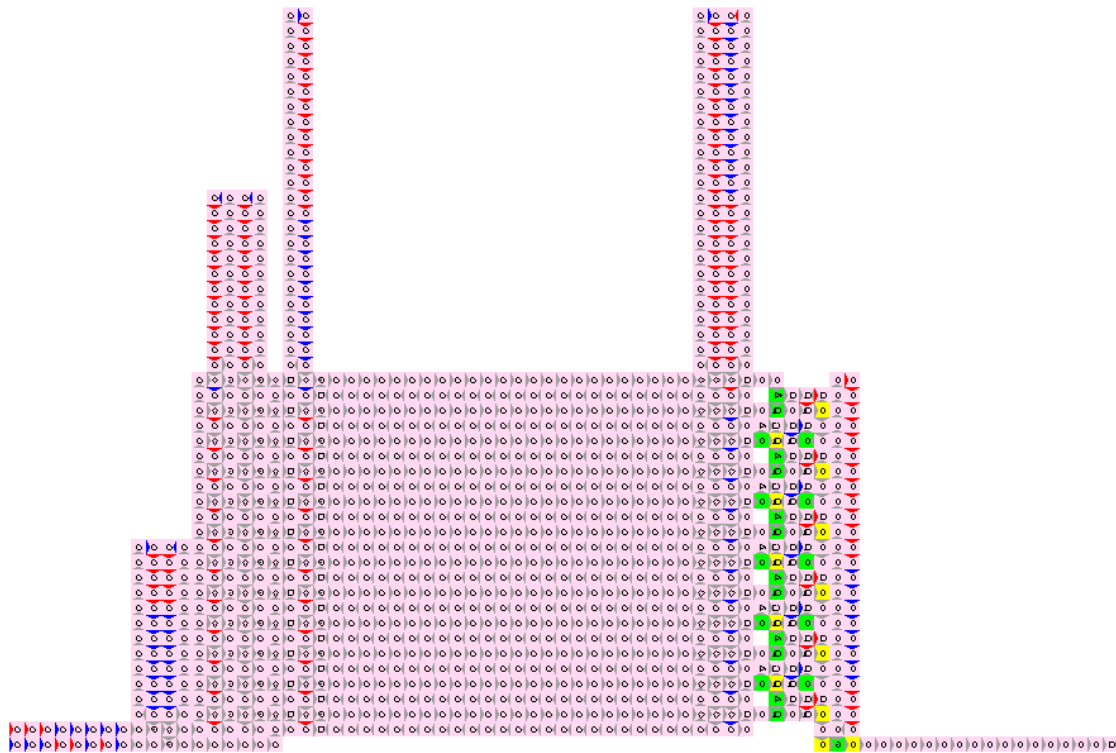
2.2 Fixed Point Representation

By using fixed-point, I pass around single strings rather than triples, and perform a single arithmetic operation for each computational (as opposed to representational) arithmetic operation. Albeit, the single operation has some additional overhead related to fixed-point-ness, but this turns out to be small compared to the overhead of mixed-number representation.

I chose to used two’s complement (as opposed to signed) representation to handle negative numbers.

Fixed-Point Multiplier

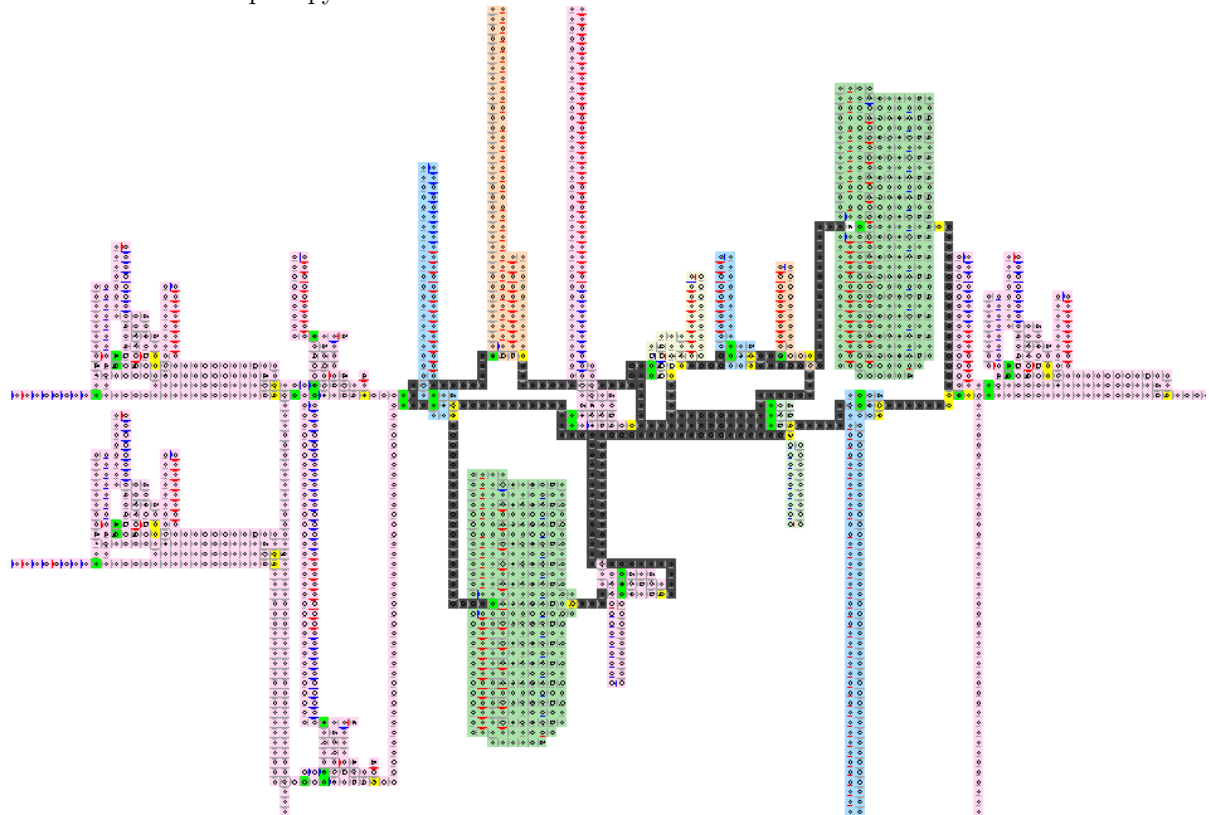
See demo file “demo-tcmult.py”



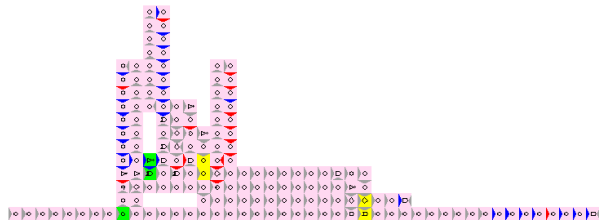
A few words about the multiplier. The uniform structure visible at the center serves the purpose of token-buffering. The basic concept of the integer multiplier (completed prior to this semester) is sum offset partial products, as one might do on paper. The extra apparatus for fixed-point two's complement multiplication is (i) padding, which is necessary to get all the bits right (I admit that there are some details I haven't worked out as to why the padding must be at least some specific amount) (ii) truncation, where the offset is tuned to correspond to the number of decimal places.

Fixed-Point Divider

See demo file “demo-fpdiv.py”



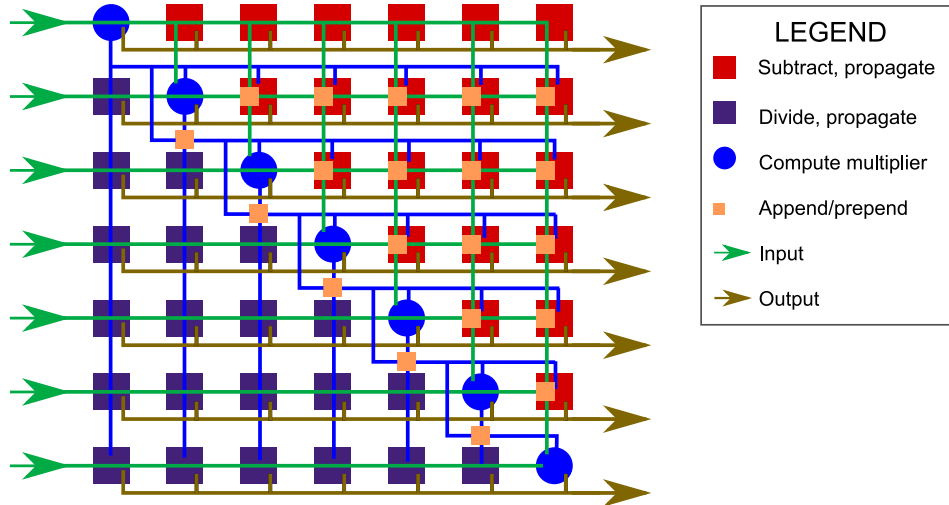
A few words about the divider. Integer division was completed prior to this semester - part of the extra apparatus, similarly as above, is devoted to padding and truncation. The other part is for two's complement conversion. I couldn't figure out how two's complement division is supposed to work with negative numbers, so I take the absolute value, taking note of sign bits, compute the positive quotient, and then invert if appropriate. Here is the module that takes the absolute value and computes the sign bit. See demo file “demo-tcai.py”



The module can be recognized from its shape on the left side of the divider. The similar module on the right also performs two's complement inversion, but instead of being predicated on the sign bit, it takes a one-bit input specifying whether or not to invert.

3 LU Decomposition

This diagram was produced at the conclusion of the first half of the project, when the theoretical basis for the implementation was planned.



Filling out the details with modules that are implemented, here is a very messy diagram of how it will look (in a similar-but-distinct color scheme)



As a parallel computational circuit, this layout is special because products, sums, and quotients that are used multiple times are broadcast in parallel (arrival in rolling order of geographic proximity) to all owner-computes modules that require them.

4 Conclusion

My stated goal was to implement the LU Decomposition in asynchronous logic automata. I had also initially stated that the primary challenges associated with this project would be (i) number representation, and (ii) simulator size. Both of these were indeed obstacles to be reckoned with - although I was successful at not letting number representation subsume the entire project, it did subsume approximately half of the project, and all of the implementation time. In the end it was a combination of this and limitations imposed by simulator size that limited producing a working LU Decomposition.

As has been documented above, I was successful at (i) implementing fixed-point multiplication and division in two's complement representation, (ii) producing a detailed plan for using the modules I've completed to lay out an LU Decomposition circuit as soon as simulator size permits.

A Symbolic LU

This symbolic representation was instrumental in helping me formulate the algorithm presented above.

$$\begin{pmatrix} aa & bb & cc \\ dd & ff & gg \\ ww & xx & yy \end{pmatrix}$$

aa	bb	cc
$\frac{dd}{aa}$	$-\frac{bb \cdot dd}{aa} + ff$	$-\frac{cc \cdot dd}{aa} + gg$
$\frac{ww}{aa}$	$-\frac{bb \cdot ww}{aa} + xx$ $-\frac{bb \cdot dd}{aa} + ff$	$-\frac{cc \cdot ww}{aa} -$ $\frac{(-\frac{cc \cdot dd}{aa} + gg)(-\frac{bb \cdot ww}{aa} + xx)}{-\frac{bb \cdot dd}{aa} + ff} + yy$

B ALA Model

Asynchronous Logic Automata (ALA) architecture is based on the concept of a lattice of one-bit processors networked by nearest-neighbor connections. Each processor can be configured to act as one of seven primary gate types, each of which take either one or two inputs, and can provide output, in the 2-d case, to between one and four of its physical output channels.

In 2-d, any input or output can be specified as N,S,E, or W. A token is a 0 or a 1, and an input or output can contain at most one token at an given time. A gate fires as soon as its inputs are full and its outputs are clear.

In the tables below, an "x" denotes no token. This means, for example, that a crossover gate fires when either of its inputs arrives and the corresponding output is clear.

Gate Behavior

AND Gate


Glyph	D		
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	0
	0	1	0
	1	0	0
	1	1	1

WIRE

AND gate with equal inputs


Glyph	O	
Inputs	1	
Outputs	1,2,3,4	
Behavior	in	out
	0	0
	1	1

NAND Gate


Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	1
	0	1	1
	1	0	1
	1	1	0

INV


NAND gate with equal inputs

Glyph		
Inputs	1	
Outputs	1,2,3,4	
Behavior	in	out
	0	1
	1	0


OR Gate

Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	0
	0	1	1
	1	0	1
	1	1	1


XOR Gate

Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in1	in2	out
	0	0	0
	0	1	1
	1	0	1
	1	1	0


COPY Gate

Glyph				
Inputs	2			
Outputs	1,2,3,4			
Behavior	in	in _c	out	in'
	0	0	0	x
	0	1	0	0
	1	0	1	x
	1	1	1	1

DELETE Gate

Glyph			
Inputs	2		
Outputs	1,2,3,4		
Behavior	in	in _c	out
	0	0	0
	0	1	x
	1	0	1
	1	1	x

CROSSOVER Gate

Glyph				
Inputs	2			
Outputs	2			
Behavior	in1	in2	out1	out2
	0	x	0	x
	x	0	x	0
	1	x	1	x
	x	1	x	1
	0	0	0	0
	0	1	0	1
	1	0	1	0
	1	1	1	1

Example Circuit

The following example circuit performs multiplication of one 4-bit operand with a second operand whose word length depends on bit loop initialization.

