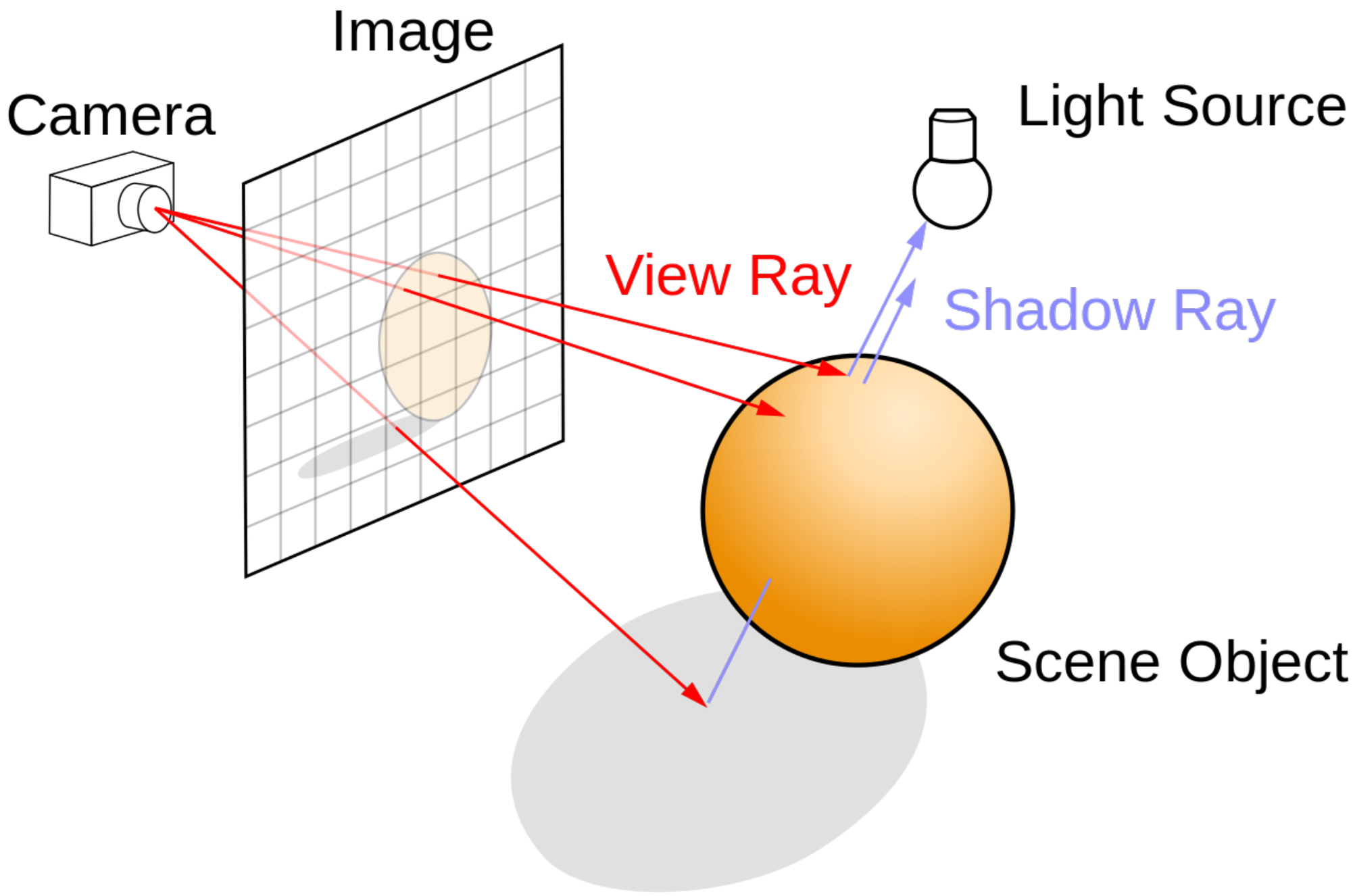
Ray-Quadric Intersection Solver

Matthew Whiteside

70% Finished Ray-Quadric Intersection Solver

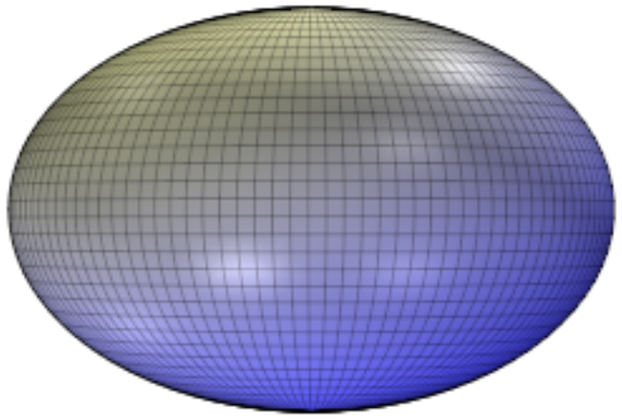
Matthew Whiteside

Background

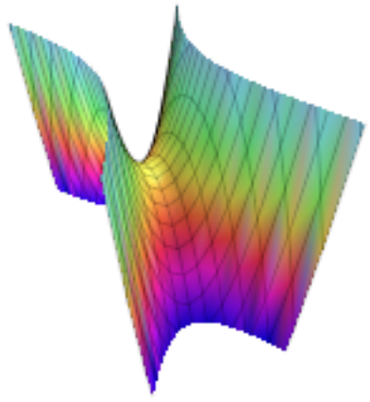


This slide is from wikipedia: http://upload.wikimedia.org/wikipedia/commons/thumb/8/83/Ray_trace_diagram.svg/1280px-Ray_trace_diagram.svg.png

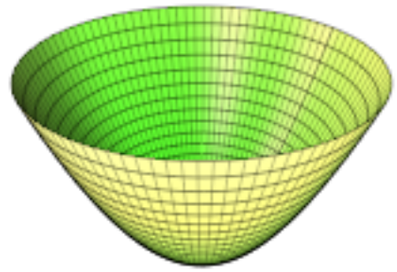
Quadric Surfaces



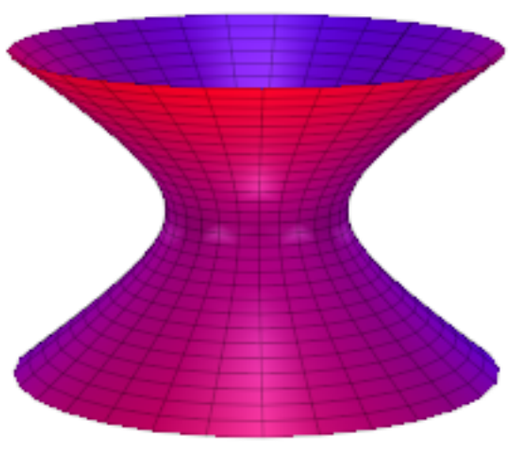
Ellipsoid



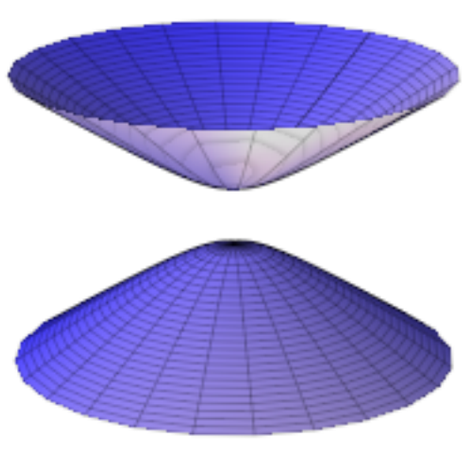
Hyperbolic paraboloid



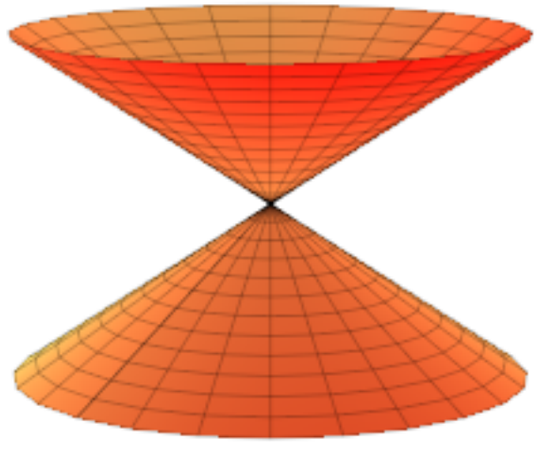
Elliptic paraboloid



Hyperboloid of one sheet



Hyperboloid of two sheets



Cone

http://hardycalculus.com/calcindex/IE_quadric.htm

http://users.wowway.com/~phkahler/quadrics.pdf

Defining the following set of 3-element vectors:

$V1 = (x_1^2 , y_1^2 , z_1^2)$
$V2 = 2(x_1y_1 , y_1z_1 , x_1z_1)$
$V3 = 2(x_0x_1 , y_0y_1 , z_0z_1)$
$V4 = 2(x_1y_0+x_0y_1 , y_0z_1+y_1z_0 , x_0z_1+x_1z_0)$
$V5 = 2(x_1 , y_1 , z_1) = 2P_1$
$V6 = (x_0^2 , y_0^2 , z_0^2)$
$V7 = 2(x_0y_0 , y_0z_0 , x_0z_0)$
$V8 = 2(x_0 , y_0 , z_0) = 2P_0$

$Q1 = (a , b , c)$
$Q2 = (d , e , f)$
$Q3 = (g , h , j)$

And substituting these into (5),(6),(7) results in the following (where • is the standard dot product):

$A = Q1 \bullet V1 + Q2 \bullet V2$
$B = Q1 \bullet V3 + Q2 \bullet V4 + Q3 \bullet V5$
$C = Q1 \bullet V6 + Q2 \bullet V7 + Q3 \bullet V8 + k$

Start with the general Quadric Equation in 3 variables:

1) $ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$

And the parametric equation for a ray:

2) $R = P_0 + P_1t$ where $P_0 = (x_0,y_0,z_0)$ $P_1 = (x_1,y_1,z_1)$

3) $R = (x_0+x_1t , y_0+y_1t , z_0+z_1t)$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Implementation

3rd Party IP Used

- Several floating point units from the UC Berkeley:
  - Fused Multiplier/Adder
  - Combined Division/Sqrt Unit
  - FP Comparator

GitHub This repository Search Explore Features Enterprise Blog Sign up Sign in

ucb-bar / berkeley-hardfloat Watch 64 Star 5 Fork 3

branch: master berkeley-hardfloat / src / main / scala / divSqrtRecodedFloat64.scala

yunsup on Mar 9 commit work-in-progress 64-bit div/sqrt unit

1 contributor

93 lines (78 sloc) 2.184 kb Raw Blame History

```
1  // See LICENSE for license details.
2
3  package hardfloat
4
5  import Chisel._
6  import Node._
7  import consts._
8
9  class divSqrtRecodedFloat64 extends Module
10 {
11   val io = new Bundle {
12     val inReady_div = Bool(OUTPUT)
13     val inReady_sqrt = Bool(OUTPUT)
14     val inValid = Bool(INPUT)
15     val sqrtOp = Bool(INPUT)
16     val a = Bits(INPUT, 65)
17     val b = Bits(INPUT, 65)
18     val roundingMode = Bits(INPUT, 2)
```

https://github.com/ucb-bar/berkeley-hardfloat

Author

John Hauser

Recoded Format

The floating-point units in this repository work on an internal recoded format (exponent has an additional bit) to handle subnormal numbers more efficiently in a microprocessor. A more detailed explanation will come soon, but in the mean time here are some example mappings for single-precision numbers.

```
IEEE format                          Recoded format
----------------------------------   ----------------------------------
s 00000000 00000000000000000000000   s 000------ 00000000000000000000000
s 00000000 00000000000000000000001   s 001101011 00000000000000000000000
s 00000000 00000000000000000000001f  s 001101100 f0000000000000000000000
s 00000000 00000000000000000000001ff s 001101101 ff0000000000000000000000
    ...              ...                 ...              ...
s 00000000 001fffffffffffffffffff    s 001111111 fffffffffffffffffff000
s 00000000 01fffffffffffffffffff     s 010000000 fffffffffffffffffff00
s 00000000 1fffffffffffffffffffff    s 010000001 fffffffffffffffffff0
s 00000001 ffffffffffffffffffffff    s 010000010 ffffffffffffffffffff
s 00000010 ffffffffffffffffffffff    s 010000011 ffffffffffffffffffff
    ...              ...                 ...              ...
s 11111101 ffffffffffffffffffffff    s 101111110 ffffffffffffffffffff
s 11111110 ffffffffffffffffffffff    s 101111111 ffffffffffffffffffff
s 11111111 00000000000000000000000   s 110------ ----------------------
s 11111111 ffffffffffffffffffffff    s 111------ ffffffffffffffffffff
```

Scala/Chisel → SystemVerilog

```scala
// See LICENSE for license details.

package hardfloat

import Chisel._
import Node._

object recodedFloatNToFloatN
{
  def apply(in: UInt, sigWidth: Int, expWidth: Int) = {
    val sign = in(sigWidth+expWidth)
    val expIn = in(sigWidth+expWidth-1, sigWidth)
    val fractIn = in(sigWidth-1, 0)

    val isHighSubnormalIn = expIn(expWidth-3, 0) < UInt(2)
    val isSubnormal = expIn(expWidth-1, expWidth-3) === UInt(1) ||
    val isNormal = expIn(expWidth-1, expWidth-2) === UInt(1) && !is
    val isSpecial = expIn(expWidth-1, expWidth-2) === UInt(3)
    val isNaN = isSpecial && expIn(expWidth-3)

    val denormShiftDist = UInt(2) - expIn(log2Up(sigWidth)-1, 0)
    val subnormal_fractOut = (Cat(Bool(true), fractIn) >> denormShi
    val normal_expOut = expIn(expWidth-2, 0) - UInt((1 << (expWidth

    val expOut = Mux(isNormal, normal_expOut, Fill(expWidth-1, isSp
    val fractOut = Mux(isNormal || isNaN, fractIn, Mux(isSubnormal,

    Cat(sign, expOut, fractOut)
  }
}
```

```systemverilog
function automatic logic [IEEE_argWidth-1:0] decodeUCBFloat(logic [internalAr
  logic sign,isHighSubnormalIn,isSubnormal,isNormal,isSpecial,isNaN;

  logic [sigWidth-1:0] fractIn,stuff,subnormal_fractOut,fractOut;
  logic [expWidth-1:0] expIn;
  logic [expWidth-2:0] normal_expOut,expOut;

  integer denormShiftDist;

  sign = in[64];
  expIn = in[63:52];
  fractIn = in[51:0];

  isHighSubnormalIn = (expIn[expWidth-3: 0] < 2);
  isSubnormal = expIn[expWidth-1: expWidth-3] === 1 || expIn[expWidth-1:expWi
  isNormal = expIn[expWidth-1:expWidth-2] === 1 && !isHighSubnormalIn || expI
  isSpecial = expIn[expWidth-1:expWidth-2] === 3;
  isNaN = isSpecial && expIn[expWidth-3];

  denormShiftDist = 2 - expIn[5:0];//this line is hardcoded for 52 bit signif
  stuff = {1'b1,fractIn} >> denormShiftDist;
  subnormal_fractOut = stuff[sigWidth-1:0];
  normal_expOut = expIn[expWidth-2:0] - ((1 << (expWidth-2))+1);

  expOut = isNormal ? normal_expOut : {(expWidth-1){isSpecial}};
  fractOut = isNormal || isNaN ?  fractIn : isSubnormal ? subnormal_fractOut

  return {sign, expOut, fractOut};

endfunction
```

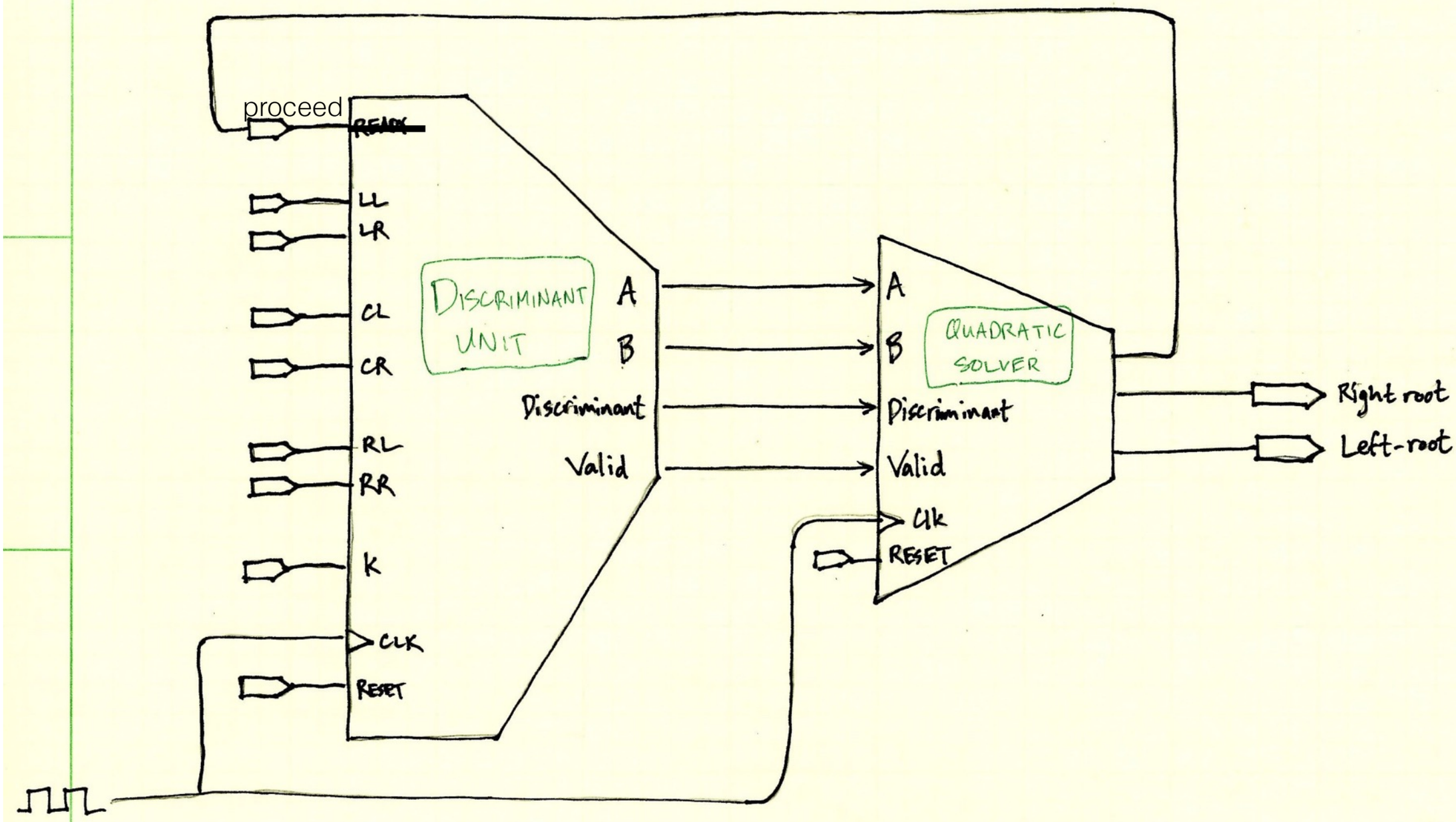
Why choose the UCB FP units given the problems mentioned?

1. Wanted to use fused-multiplier/adder

2. UCB designs are free and open source

3. To become familiar with the RISC-V ecosystem

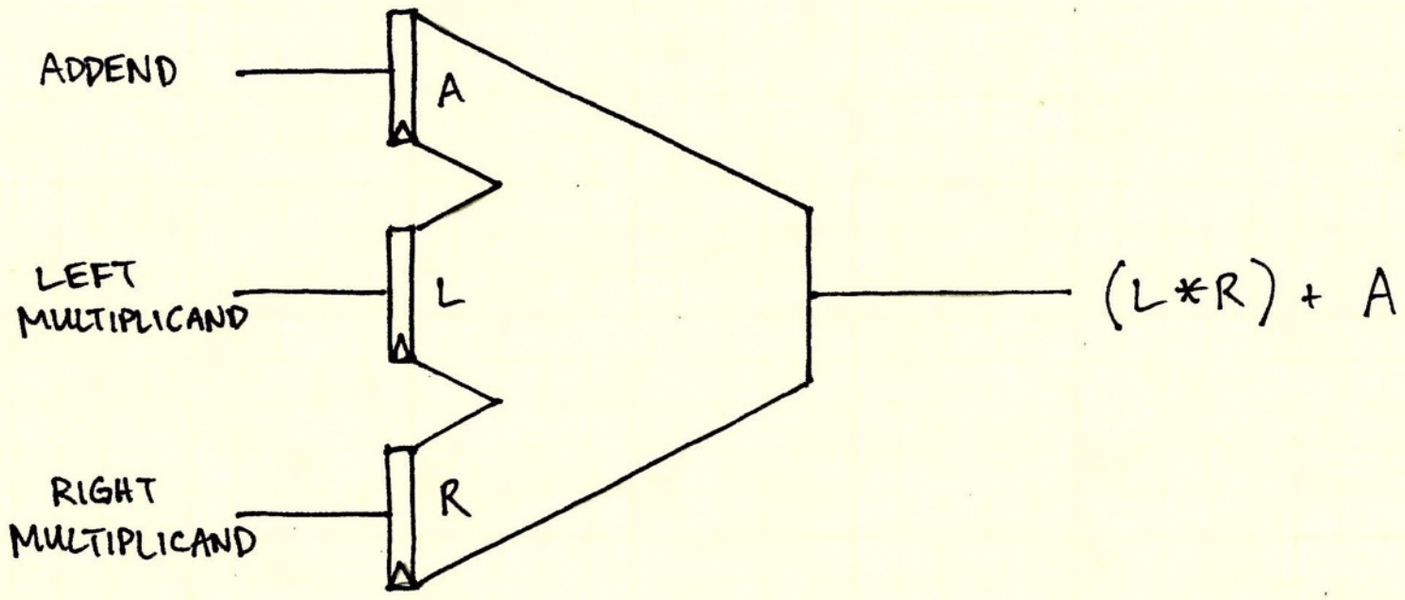
HIGH LEVEL OVERVIEW



inputs are 3 pairs of 3-vectors, that correspond to the 3 dot products at the bottom of the Paul Kahler paper

64-BIT FUSED MULTIPLIER-ADDER (&SUBTRACTOR)

AKA FMA64

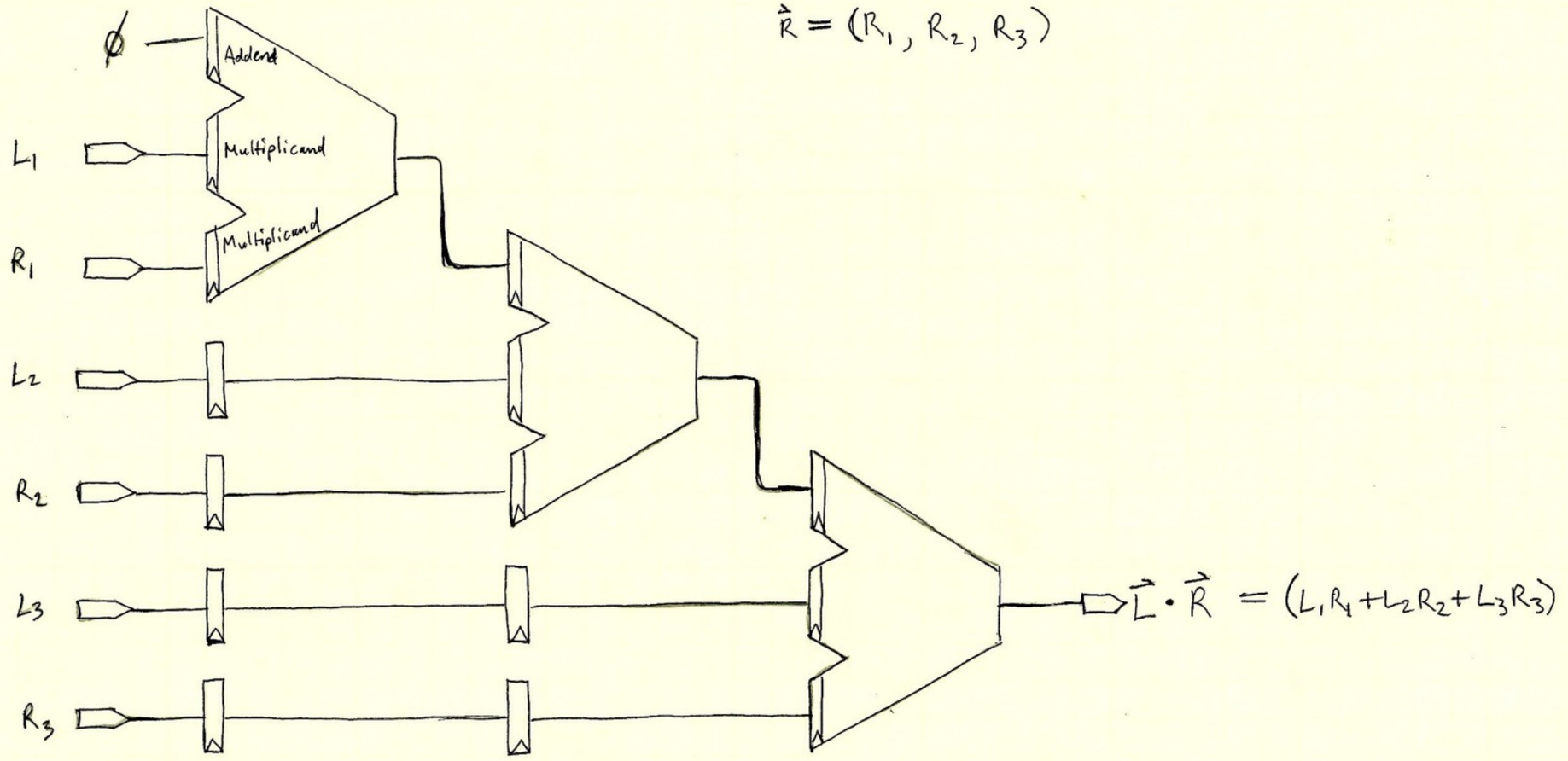


NOTES

- WRAPPED BERKELEY UNIT TO HIDE INPUTS I DIDN'T CARE ABOUT
- ADDED REGISTERS AT THE INPUTS

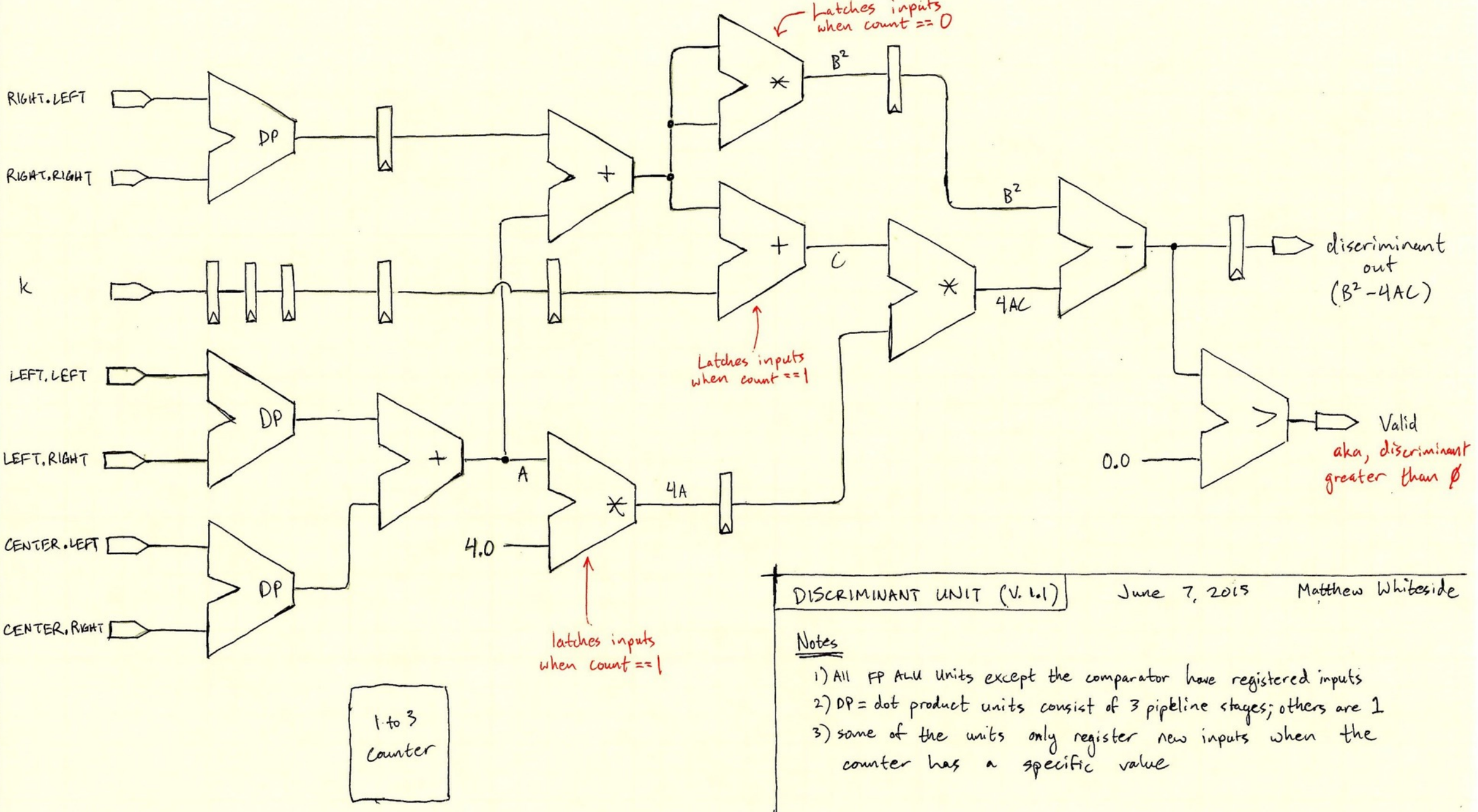
$\vec{L} = (L_1, L_2, L_3)$

$\vec{R} = (R_1, R_2, R_3)$



DOT PRODUCT UNIT

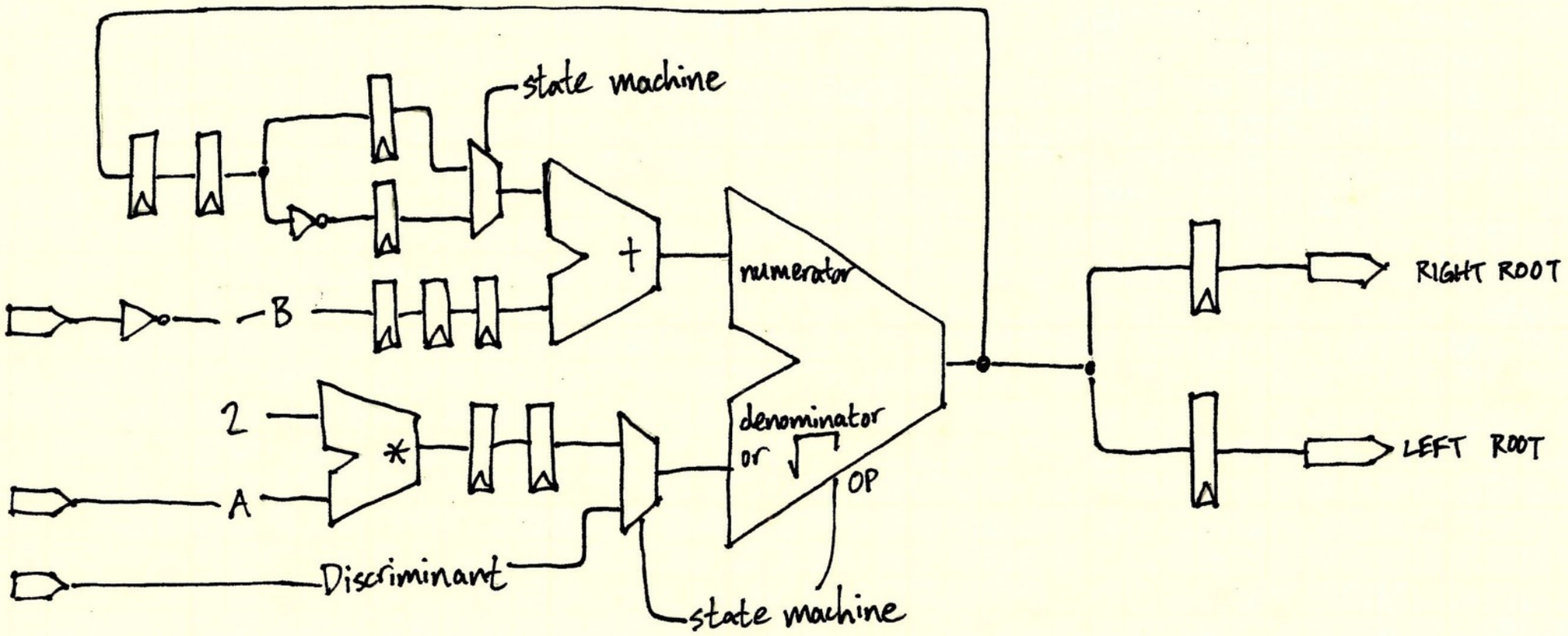
Pipeline Stage 1 2 3 4 5 6 7 8 9



DISCRIMINANT UNIT (V. 1.1) June 7, 2015 Matthew Whiteside

Notes

1) All FP ALU units except the comparator have registered inputs
2) DP = dot product units consist of 3 pipeline stages; others are 1
3) some of the units only register new inputs when the counter has a specific value



QUADRATIC SOLVER

Note: this is my guess as to roughly what Questa is synthesizing.

Verification

Basic Approach

- Store arguments in a queue of structs
    - queue is not synthesizable so implement with an array and maintain with a for-loop
- Push and pop items onto the queue when the ready and valid signals are raised respectively
- Assert statements to verify requirements for the unit, i.e.. discriminant > 0, number of ops in flight < 5

Verification Status

- Still in progress, due to pipelining not yet correct in quadratic solver unit.

- Will exercise the quadratic unit with randomized test values, with a mix of valid and invalid inputs

- Pipelining the div/sqrt unit is tricky due to the operations can be interleaved, and this will be the main focus of the tests

Veloce Status

- Compiles ✓
- Was able to run it on an earlier version of my test bench last Friday
  - haven’t run with most recent changes
- Notes: real number expressions won’t compile in an always block, but they will in a forever block.