APPLICATION OF NUMERICAL ANALYSIS METHODS (CRAMER'S RULE)

Abstract:

Cramer's rule is an explicit formula for the solution of a system of linear equations with as many equations as unknowns, valid whenever the system has a unique solution. It expresses the solutions in terms of the determinant of the (square) coefficient matrix, and of the determinants of some other matrices obtained from it (discussed in the introduction section).

In this work, I propose a design for solving a system of linear equations with as many equations as unknowns based on **Cramer's rule**, using Verilog HDL.
The implementation poses no restrictions on the number of equations to be solved, one just has to increase the size of a certain register to accommodate more numbers (discussed in detail in the implementation section).

The design is verified with a test plan implemented in SystemVerilog, which asserts that the design produces the correct vales (discussed in the Evaluation section).

Introduction:

Cramer's rule uses determinants instead of the inverse to solve linear systems of equations. Which is a major disadvantage, as when the number of equations get bigger, the algorithm becomes computationally inefficient.
In the case of n equations, it is required to compute n+1 determinants, while there are other methods like Gaussian Elimination which produces the result with the same computational complexity as computing a single determinant.
Another disadvantage is that one can only solve for one variable at a time.

Considering these disadvantages, most computer programs do not use Cramer's rule to solve systems of equations. For people, however, it is generally the easiest way to solve systems of equations with a pen and paper.

- **The method**

Given a set of n equations with n unknowns, these unknowns are can be obtained by calculating the determinant of the coefficient matrix (matrix on the LHS of the equation), and the determinants of the matrices obtained from it by replacing one column by the vector on the RHS of the equation.

The method is better understood with an example.

  - **2 equations with 2 unknowns.**

Given the following system of $\begin{matrix} x - y = 4 \\ 2x + y = 2 \end{matrix}$ equations:

We first calculate the determinant of the coefficient $D = \begin{vmatrix} 1 & -1 \\ 2 & 1 \end{vmatrix} = 3$ matrix

Then we swap the vector on the RHS with the first column of the coefficient matrix, to obtain a different matrix, lets call it D_x

$$D_x = \begin{vmatrix} 4 & -1 \\ 2 & 1 \end{vmatrix} = 6$$

Then we do the same but with the other column of the the coefficient matrix, to obtain a different matrix D_y

$$D_y = \begin{vmatrix} 1 & 4 \\ 2 & 2 \end{vmatrix} = -6$$

Finally, to obtain the values of the unknowns x and y, we simply divide D_x and D_y by D.

$$x = \frac{D_x}{D} = \frac{6}{3} = 2$$

$$y = \frac{D_y}{D} = \frac{-6}{3} = -2$$

- **3 equations with 3 unknowns.**

The same method is used for any n equations with n unknowns.

$$\begin{aligned} 2x + 3y - 5z &= 1 \\ x + y - z &= 2 \\ 2y + z &= 8 \end{aligned}$$ Given:

We calculate the $D = \begin{vmatrix} 2 & 3 & -5 \\ 1 & 1 & -1 \\ 0 & 2 & 1 \end{vmatrix} = -7$ determinants:

$$D_x = \begin{vmatrix} 1 & 3 & -5 \\ 2 & 1 & -1 \\ 8 & 2 & 1 \end{vmatrix} = -7 \qquad D_y = \begin{vmatrix} 2 & 1 & -5 \\ 1 & 2 & -1 \\ 0 & 8 & 1 \end{vmatrix} = -21$$

$$D_z = \begin{vmatrix} 2 & 3 & 1 \\ 1 & 1 & 2 \\ 0 & 2 & 8 \end{vmatrix} = 14$$

Then calculate the $x = \frac{D_x}{D}$ $y = \frac{D_y}{D}$ $z = \frac{D_z}{D}$ unknowns:

So far, I've talked about the method without stating how these determinants are calculated. So, let's discuss that.

- **Calculating determinants**

  - ***2x2 matrix determinant***

Given a matrix **A** = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

Det(A) = a*d – b*c

  - ***3x3 matrix determinant***

Given a matrix **A** = $\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}$

Det(A) = a1 $\begin{bmatrix} b2 & c2 \\ b3 & c3 \end{bmatrix}$ – b1 $\begin{bmatrix} a2 & c2 \\ a3 & c3 \end{bmatrix}$ + c1 $\begin{bmatrix} a2 & b2 \\ a3 & b3 \end{bmatrix}$

= a1(b2*c3 – c2*b3) – b1(a2*c3 – c2*a3) + c1(a2*b3 – b2*a3)

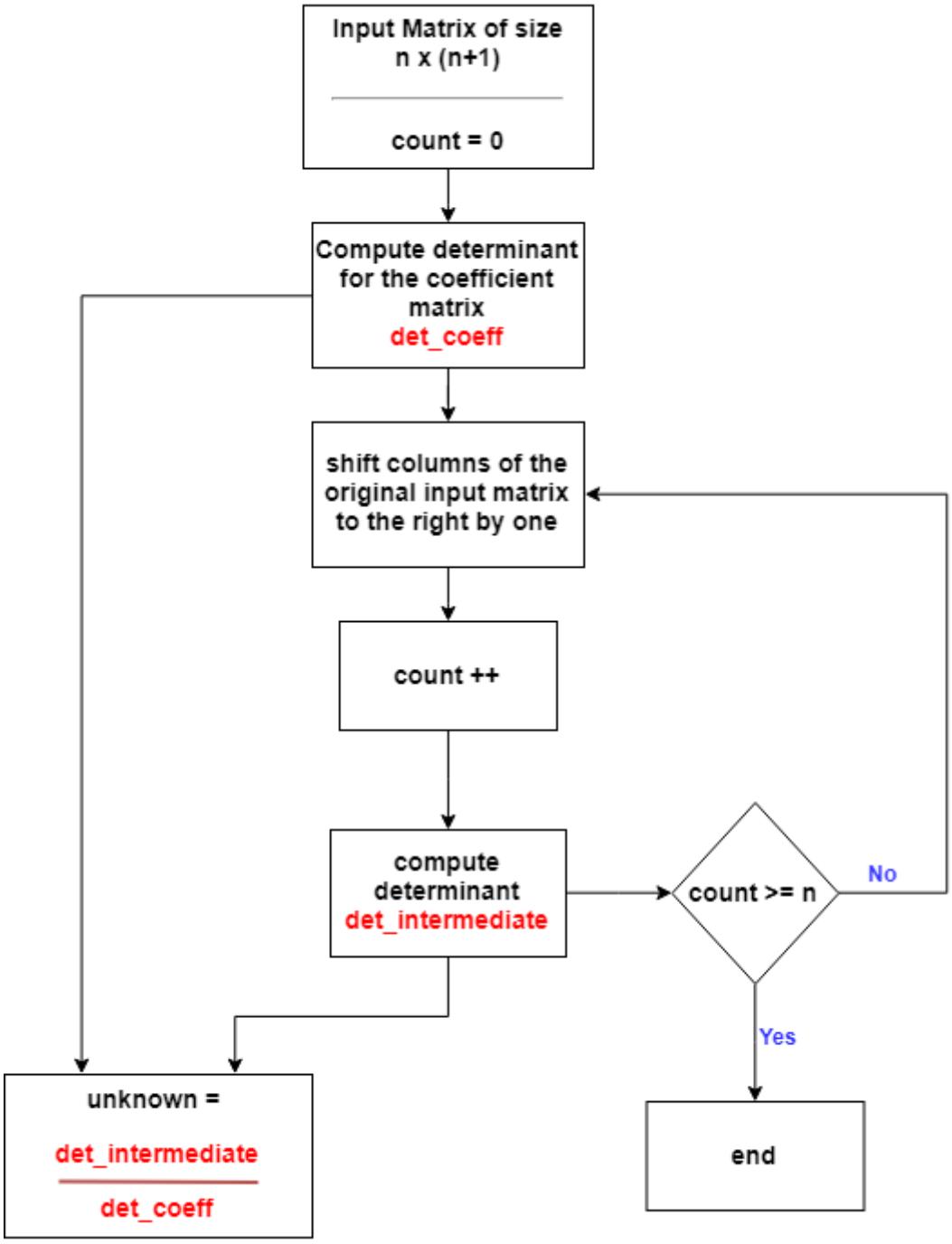
  - ***4x4 matrix determinant***

$$|A| = \begin{vmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{vmatrix} = A_{11} \begin{vmatrix} A_{22} & A_{23} & A_{24} \\ A_{32} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} \end{vmatrix} - A_{21} \begin{vmatrix} A_{12} & A_{13} & A_{14} \\ A_{32} & A_{33} & A_{34} \\ A_{42} & A_{43} & A_{44} \end{vmatrix}$$

$$+ A_{31} \begin{vmatrix} A_{12} & A_{13} & A_{14} \\ A_{22} & A_{23} & A_{24} \\ A_{42} & A_{43} & A_{44} \end{vmatrix} - A_{41} \begin{vmatrix} A_{12} & A_{13} & A_{14} \\ A_{22} & A_{23} & A_{24} \\ A_{32} & A_{33} & A_{34} \end{vmatrix}$$

So, we can see the pattern here, a 4x4 matrix is equivalent to 4 3x3 matrices, and a single 3x3 matrix is equivalent to 3 2x2 matrices. In general, an NxN matrix is equivalent to N (N-1xN-1) matrices.

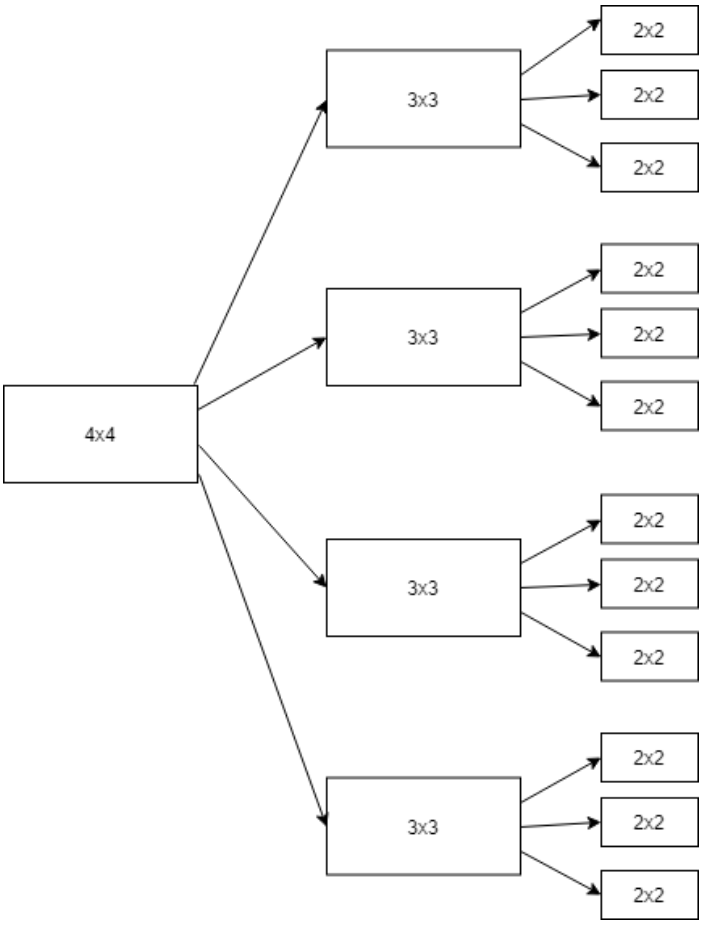
Proposed method:



The proposed method is a direct implementation of how one would use Cramer to solve a system of equations with a pen and paper.

First, I have a 2d array of integers as input (which is read from a text file), I then call a function to calculate the determinant of the coefficient matrix and store it in a register. **(Determinant calculation will be discussed in the following subpoint)**. Then, I loop for n times, where n is the coefficient matrix size, to form an intermediate matrix with every iteration, and compute its determinant, and divide it by the determinant of the coefficient matrix to get the result of the first unknown in the equation and output it in a file. Also, the result in the file is a decimal number with precision of 4 numbers after the decimal point.

- **Determinant Calculation**



The determinant is calculated in a recursive function, that reduces the input matrix to multiple smaller matrices and calls itself with each of these matrices, until the size of these smaller matrices becomes 2x2, then it computes its determinant and returns it to the previous function call, which uses these values to obtain the result of the original input matrix.

Implementation:

Note: Before I talk about the design, there's one thing I need to mention quickly about the testbench, because it affects something in the design. The way the testbench works is that it writes a matrix in the input file, then sets a signal for the main module to read this matrix and compute the results and stores it in an output file. Then the testbench reads that output file and make sure that the values are correct. So, I defined a parameter "test" which when set to 1, means that the testbench is the one that writes the input matrix in the file, so the design will wait for the start signal to become 1 before reading the file. If this parameter is set to 0, then we're not using the testbench and the user himself has already written the matrix in the input file, so the design will read it right away without waiting.

```
module cramer(start);

input start;

parameter size = 3;  // defines the size of the input square matrix
parameter test = 0;  // used by the testbench

reg signed [31:0] m [0:size-1][0:size-1];    // base square matrix read from the input file
reg signed [31:0] z [0:size-1][0:size-1];    // to store temporary matrices to compute their determinants
reg signed [31:0] r [0:size-1];     // stores the vector
reg signed [32*size*size-1:0] oneDarray;  // stores 1d array of bits to be able to use it as input to the function
reg [31:0] ans [0:3]; // stores the digits of the final results of the equations unknowns

integer a, b, q, remainder; // variables used in calculating floating point division
integer i, j, k, g, x, y, data_file, scan_file, out_file;   // variables used in for loops and file operations
integer signed determinant;   // stores the determinant of the base matrix
```

First, I defined the module "cramer" with one input "**start**", which is used by the testbench (discussed in the evaluation section).

- Parameter "**test**" is used to indicate if we're running the main design or the testbench

- Parameter "**size**" indicates the size of the input coefficient matrix.

- "**m**" is the 2d square coefficient matrix that'll be read from the file

- "**r**" is the vector on the right hand side of the equations

- "**z**" is a register to store the intermediate matrices

- "**oneDarray**" As functions don't accept multidimensional arrays as input, this register stores a flattened version of any 2d matrix of integers to be able to send it to the function.

- "**ans**" stores the individual digits of each of the final results (its use will become clear later)

- The rest are indicated in the comments in the screenshot above.

```
initial begin
   for (i=0; i<size; i=i+1) begin
      for (j=0; j<size; j=j+1) begin
         z[i][j] = 0;
      end
   end
end
```

- Then, I initialize "**z**" to zeros in the initial block.

```
always @* begin
   if(test == 1) wait(start == 1);

   if (test == 0 || start == 1) begin
      data_file = $fopen("input.txt", "r");
      for (i=0; i<size; i=i+1) begin
         for (j=0; j<size+1; j=j+1) begin
            if (j==size) scan_file = $fscanf(data_file, "%d\n", r[i]);
            else scan_file = $fscanf(data_file, "%d\n", m[i][j]);
         end
      end
   end
   $fclose(data_file);
   $display("Read Matrix %p", m);
   $display("Read RHS vector %p", r);
```

- In the always block, the first thing I do is check if the parameter "**test**" is 1, if so then I'll wait for the input "**start**" to be 1 to continue (the reason is discussed in the note written in the previous page).

- Then I have an if condition that will execute if either the start signal becomes 1 (by the testbench), or if we're not testing and the user has written the matrix in the input file and is running the module directly.

- Inside the if, I use 2 for loops to read the 2d matrix from the input file "**input.txt**" and store the coefficient matrix (square matrix) in the reg "**m**", and the RHS vector in the reg "**r**".

```
/*_____________Construct 1d array of bits from the input 2d array of 32-bit ints_____________*/
k = 0;
for (i=0; i<size; i=i+1) begin
   for (j=0; j<size; j=j+1) begin
      if (m[i][j] < 0) oneDarray[k*32+31 -: 32] = m[i][j];
      else oneDarray[k*32+31 -: 32] = m[i][j];
      k = k+1;
   end
end
```

- The code above flattens the read 2d array of 32-bit integers “m” and saves it as a stream of 32bits in a long vector “**oneDarray**”. This is to be able to send it to a function as functions don’t accept multidimensional inputs.

```
/*_____________________________Compute coefficient matrix determinant_____________________________*/
determinant = det(size, oneDarray);
b = determinant;
$display("Read Matrix determinant = %0d", determinant);

out_file = $fopen("output.txt", "w");
if (determinant == 0) $fwrite(out_file, "x");
else begin
```

- The code above calls a function “det” with the size of the coefficient matrix, and the actual coefficient matrix after flattening it. The function computes the determinant of the coefficient matrix and stores the result in “determinant”. I then save determinant in another integer variable “b” for a reason which will become clear later. If the coefficient matrix determinant is zero, then ill output “x” in the output file and stop.

Note: “x” means that either the system has no solution at all or have infinite number of solutions.

```
/*_____________________Create intermediate matrices and compute their determinants_____________________*/
out_file = $fopen("output.txt", "w");
for (g=0; g<size; g=g+1) begin // create n matrices

   for (i=0; i<size; i=i+1) begin
      for (j=0; j<size; j=j+1) begin
         if (j == g) z[i][j] = r[i];
         else z[i][j] = m[i][j];
      end
   end
   k = 0;
   for (x=0; x<size; x=x+1) begin
      for (y=0; y<size; y=y+1) begin
         if (z[x][y] < 0) oneDarray[k*32+31 -: 32] = z[x][y];
         else oneDarray[k*32+31 -: 32] = z[x][y];
         k = k+1;
      end
   end
   a = det(size, oneDarray);
   $display("Intermediate Matrix %0d = %p", g, z);
   $display("Intermediate Matrix %0d determinant = %0d", g, a);
```

- Then I open the output file which will store the results. The first for loop with variable “g” (Note that its end is yet to come) is responsible for creating the different intermediate matrices with each iteration, compute their determinants, divide it by the determinant of the coefficient matrix, and store the result in the output file. It will execute the same number of times as the size of the coefficient matrix.

Inside it are first 2 for loops which are responsible for creating a different intermediate matrix with each iteration of “g” and store it in “z”.

Then, I again flatten this 2d matrix to be a 1d array of bits and call the determinant function which will compute the determinant of this intermediate matrix.

```
/*_____________________________________Floating point division_____________________________________*/

if (a[31] && b[31]) begin
   a = -a;
   b = -b;
end
else if (a[31] == 1'b1) begin
   a = -a;
   $fwrite(out_file, "-");
end
else if (b[31] == 1'b1) begin
   b = -b;
   $fwrite(out_file, "-");
end
q = a/b;
remainder = a - (q*b);
for (i=0; i<4; i=i+1) begin
   remainder = remainder*10;
   ans[i] = remainder/b;
   remainder = remainder - (ans[i]*b);
end
b = determinant;
```

- The above code does the division of the intermediate matrix determinant by the coefficient matrix determinant, with a precision of 4 digits after the decimal point.

First, I check the most significant bit in each number to see if its negative or positive, if both are negative then I flip them to positive, if one of them is negative then I output a negative sign in the output file, then flip it to positive and continue the calculation.

I then compute the quotient and remainder of the division operation and loop, the quotient is outputted directly after the negative sign if present while the remainder is first converted to a decimal value then output this value afterwards.

I save the original coefficient matrix determinant “determinant” in “b” again because the value of “b” would be changed from negative to positive if it was positive in the first place.

```
      /*_____________________________________Write results in the output file_____________________________________*/
      $fwrite(out_file, "%0d . ", q);
      for (i=0; i<4; i=i+1) begin
         $fwrite(out_file, "%0d", ans[i]);
      end
      $fwrite(out_file, "\n");
   end
   $fclose(out_file);
end //end always
endmodule
```

- The above code outputs first the quotient followed by a decimal point in the output file, then loops on the values stored in “ans” and outputs them after the decimal point. So, the first line in the output file would be something like “2 . 1250”, which corresponds to the value of the first unknown in the system of equations.

- **Determinant Function (det)**

The function to compute the determinants is a recursive function which takes the matrix (as an array of bits) and the matrix size as input, and if the matrix size is bigger than 2x2, it keeps reducing it and recursively call itself with the reduced auxiliary matrices and their sizes, until it gets a 2x2 matrix, then it computes its determinant and returns the value to where it was called.

Its hard to describe the operation of a recursive function like this one in words, so I'll use the example I wrote in the introduction.

Given a matrix **A =** $\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}$ sent to the function det.

The function will check the matrix size sent to it and find that its 3, so it'll reduce it to 3 matrices of size 2x2 and recursively call itself with each one of these matrices and its size (2) as input.

$$\text{a1} \begin{bmatrix} \boldsymbol{b2} & \boldsymbol{c2} \\ \boldsymbol{b3} & \boldsymbol{c3} \end{bmatrix} - \text{b1} \begin{bmatrix} a2 & c2 \\ a3 & c3 \end{bmatrix} + \text{c1} \begin{bmatrix} a2 & b2 \\ a3 & b3 \end{bmatrix}$$

With each call, the function will check the size and find that its 2, so it'll compute the determinant of this 2x2 matrix and returns its value to where it was called.

Determinant of $\begin{bmatrix} \boldsymbol{b2} & \boldsymbol{c2} \\ \boldsymbol{b3} & \boldsymbol{c3} \end{bmatrix}$ **= b2*c3 – c2*b3**

```
/*------------------------------------------------------------ Function Start -----
function automatic integer signed det;
   input integer n;
   input signed [32*size*size-1:0] input_array_1d;
   integer i, j, k, x, y, z, e, r;
   integer signed out, temp;
   reg signed [31:0] array_2d [0:size-1][0:size-1];
   reg signed [31:0] temp_2d [0:size-1][0:size-1];
   reg signed [32*size*size-1:0] temp_1d;
```

- The function “det” is an automatic function to be able to call it recursively without its variables overriding each other with each call.

- Its inputs are first the matrix size which I save in integer “**n**”, and the matrix itself (flattened version of it) which I save in “**input_array_1d**” of size the size of the original input matrix.

- “**array_2d**” stores the “**input_array_1d**” but in a 2d array of integers form, to be able to make the determinant matrix computations.

- “**temp_2d**” stores the reduced matrices in a 2d array of integers form.

- “**temp_1d**” stores a flattened version of “**temp_2d**” to be able to recursively call the “det” function with.

```
begin
   out = 0;
   /*__________Reconstruct the 2d array of ints for calculation_________*/
   z = 0;
   for (i=0; i<n; i=i+1) begin
      for (j=0; j<n; j=j+1) begin
         array_2d[i][j] = input_array_1d[z*32+31 -: 32];
         z = z+1;
      end
   end
```

- The above code reconstructs a 2d array of 32-bit integers from the flat input matrix to the function “**input_array_1d**”**,** and saves it in “**array_2d**”. notice it depends on the value of “n” which is the size of the input matrix.

```
/*_____________________Compute the Determinant_____________________*/
if (n == 2) begin
   out = (array_2d[0][0] * array_2d[1][1]) - (array_2d[0][1] * array_2d[1][0]);
   det = out;
end
else begin
   temp = 0;
   for (i=0; i<n; i=i+1) begin
      x=0;
      for (j=1; j<n; j=j+1) begin
         y=0;
         for (k=0; k<n; k=k+1) begin
            if (k != i) begin
               temp_2d[x][y] = array_2d[j][k];
               y=y+1;
            end
         end
      x=x+1;
      end
```

- The above code is the core of the determinant calculation. First, I check if the matrix size is 2, if so then I compute its determinant and return this value.

If the matrix is bigger than 2x2, then I construct n matrices of size (n-1 x n-1). The first for loop (with variable i) loops n times, creating the n matrices.

The next 2 for loops create one intermediate matrix and store it in "**temp_2d**".

```
/*_____________________Flatten the 2d array_____________________*/
z=0;
for (e=0; e<(n-1); e=e+1) begin
   for (r=0; r<(n-1); r=r+1) begin
      if (temp_2d[e][r] < 0) temp_1d[z*32+31 -: 32] = temp_2d[e][r];
      else temp_1d[z*32+31 -: 32] = temp_2d[e][r];
      z = z+1;
   end
end
```

- Next, I flatten this "**temp_2d**" array and save it as a one-dimensional array of bits in "**temp_1d**".

```
        /*__________________Recursively call the det function_________________*/
        temp = det(n-1, temp_1d);
        out = out + (array_2d[0][i] * (-1**(i)) * temp);
        end
      det = out;
      end
   end
endfunction
```

- Then, I recursively call the function “det” with a single flattened intermediate matrix and store its determinant in “**temp**”.

Then I add to “**out**” (which was initialized to 0 in the beginning) the value of the determinant stored at “temp”, multiplied by the corresponding element in the first row, multiplied by an alternating sign, satisfying the following equation:

$$\det\begin{bmatrix} a1 & b1 & c1 & d1 \\ a2 & b2 & c2 & d2 \\ a3 & b3 & c3 & d3 \\ a4 & b4 & c4 & d4 \end{bmatrix} = a1\begin{bmatrix} b2 & c2 & d2 \\ b3 & c3 & d3 \\ b4 & c4 & d4 \end{bmatrix} - b1\begin{bmatrix} a2 & c2 & d2 \\ a3 & c3 & d3 \\ a4 & c4 & d4 \end{bmatrix}$$
$$+ c1\begin{bmatrix} a2 & b2 & d2 \\ a3 & b3 & d3 \\ a4 & b4 & d4 \end{bmatrix} - d1\begin{bmatrix} a2 & b2 & c2 \\ a3 & b3 & c3 \\ a4 & b4 & c4 \end{bmatrix}$$

$$\text{Where } \det\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix} = a1\begin{bmatrix} b2 & c2 \\ b3 & c3 \end{bmatrix} - b1\begin{bmatrix} a2 & c2 \\ a3 & c3 \end{bmatrix} + c1\begin{bmatrix} a2 & b2 \\ a3 & b3 \end{bmatrix}$$

Evaluation:

To test the design, I implemented a testbench using SystemVerilog covering the following test plan.

I had to choose a matrix size to be able to apply the test cases to. I chose matrices of size 3x3, as it'll internally compute 2x2 determinants so if a 3x3 works, then defiantly a 2x2 works.

I didn't choose 4x4 matrices as finding the numbers to satisfy the test plan I made was a bit hard, however, after I test the 3x3 test cases, I also test with 2x2, 4x4 and a 5x5 sized matrices.

Since I save the values in the output file in the format **"number1 . number2"**, therefore in the testbench I read 3 values and store them in a reg "**ans**", then for each line I assert twice, one on "number1" which is the quotient of the result, and one on "number2" which is the number after the decimal point.

To make sure the obtained results are correct, I wrote a C program (included in the deliverables) to calculate the determinants, also I used an online system of equations solver using Cramer https://matrix.reshish.com/cramer.php and asserted that the values obtained from the design are the correct ones.

Test case	Description
Test case 1	All calculated outputs are positive integer numbers (no fractions)
Test case 2	The coefficient matrix has a negative determinant (no fractions in output)
Test case 3	An intermediate determinant is a negative number (no fractions in output)
Test case 4	Both coefficient and intermediate determinants are negative numbers (no fractions in output)
Test case 5	Outputs contain decimal values, positive and negative
Test case 6	Intermediate matrix has a zero determinant
Test case 7	Coefficient matrix has a zero determinant
Test case 8	Testing a 2x2 matrix as input
Test case 9	Testing a 4x4 matrix as input

```
                                /*_________Test Case 1 _________*/

m[0][0] = 1; m[0][1] = 2; m[0][2] = -1; r[0] = 5;
m[1][0] = 2; m[1][1] = -1; m[1][2] = 1; r[1] = 1;
m[2][0] = 3; m[2][1] = 3; m[2][2] = -2; r[2] = 8;
st = 0;
data_file = $fopen("input.txt", "w");
for (i=0; i<3; i=i+1) begin
   for (j=0; j<3+1; j=j+1) begin
      if (j==3) $fwrite(data_file, "%0d\n", r[i]);
      else $fwrite(data_file, "%0d ", m[i][j]);
   end
end
$fclose(data_file);
#100; st = 1; #100; st = 0;

out_file = $fopen("output.txt", "r");
for (i=0; i<3; i=i+1) begin
   for (j=0; j<3; j=j+1) begin
      if (j==1) scan_file = $fscanf(out_file,"%c\n", garbage);
      else scan_file = $fscanf(out_file,"%d\n", ans[i][j]);
   end
end
$fclose(out_file);
$display("x1= %0d.%0d", ans[0][0], ans[0][2]);
$display("x2= %0d.%0d", ans[1][0], ans[1][2]);
$display("x3= %0d.%0d", ans[2][0], ans[2][2]);

                              /* __ASSERT__ */
assert(ans[0][0]==1 && ans[0][2]==0) else $error("Test Case 1 wrong value");
assert(ans[1][0]==3 && ans[1][2]==0) else $error("Test Case 1 wrong value");
assert(ans[2][0]==2 && ans[2][2]==0) else $error("Test Case 1 wrong value");
```

The above code Is a sample from the testbench, covering test case 1. I first define the matrix, write it the input file, set "**st**" to 1, then to 0, which is the input signal to Cramer module, so the module will read the matrix from the file when "**st**" becomes 1, and it'll output the results in the output file, then I read this output file and assert that the values are correct.

Note: I'll include the transcript of the testbench in the appendix section.

Test case 1:

$$\begin{bmatrix} 1 & 2 & -1 & 5 \\ 2 & -1 & 1 & 1 \\ 3 & 3 & -2 & 8 \end{bmatrix}$$

Output from file:

1 . 0000
3 . 0000
2 . 0000

Output from the online solver:

x1 = 1
x2 = 3
x3 = 2

Test case 2 & 4:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 4 \end{bmatrix}$$

Output from file:

2 . 0000
2 . 0000
-1 . 0000

Output from the online solver:

x1 = 2
x2 = 2
x3 = -1

Test case 3:

$$\begin{bmatrix} 3 & 1 & 1 & 3 \\ 2 & 2 & 5 & -1 \\ 1 & -3 & -4 & 2 \end{bmatrix}$$

Output from file:

1 . 0000
1 . 0000
-1 . 0000

Output from the online solver:

x1 = 1
x2 = 1
x3 = -1

Test case 5:

$$\begin{bmatrix} 3 & 2 & 1 & 20 \\ 4 & 0 & -10 & -10 \\ -1 & -2 & 2 & 1 \end{bmatrix}$$

Output from file:

5 . 6250
-0 . 0625
3 . 2500

Output from the online solver:

x1 = 45/8
x2 = -1/16
x3 = 13/4

Test case 6:

$$\begin{bmatrix} 4 & -1 & 3 & 2 \\ 1 & 5 & -2 & 3 \\ 3 & 2 & 4 & 6 \end{bmatrix}$$

Output from file:

0 . 0000
1 . 0000
1 . 0000

Output from the online solver:

x1 = 0
x2 = 1
x3 = 1

Test case 7:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 3 \end{bmatrix}$$

Output from file:

X

Output from the online solver:

Equations either inconsistent or have infinite solutions

Test case 8:

$$\begin{bmatrix} 2 & -3 & 3 \\ 4 & -2 & 10 \end{bmatrix}$$

Output from file:

3 . 0000
1 . 0000

Output from the online solver

x1 = 3
x2 = 1

Test case 9:

$$\begin{bmatrix} 10 & -1 & 2 & 0 & 6 \\ -1 & 11 & -1 & 3 & 25 \\ 2 & -1 & 10 & -1 & -11 \\ 0 & 3 & -1 & 8 & 15 \end{bmatrix}$$

Output from file:

1 . 0000
2 . 0000
-1 . 0000
1 . 0000

Output from the online solver:

x1 = 1
x2 = 2
x3 = -1
x4 = 1

References:

http://www.ilectureonline.com/lectures/subject/MATH/36/272/3905

http://www.ilectureonline.com/lectures/subject/MATH/36/272/3906

https://www.youtube.com/watch?v=jBsC34PxzoM

https://www.youtube.com/watch?v=Ip3X9LOh2dk&vl=en

https://matrix.reshish.com/cramer.php

https://en.wikipedia.org/wiki/Cramer%27s_rule

https://semath.info/src/determinant-four-by-four.html