CircuitSim Project Report

Andrew Yang (asy2130)
Case Schemmer (chs2164)
Faustina Cheng (fc2694)
Jary Tolentino (jt3577)
Ming Gong (mg4264)

May 2025

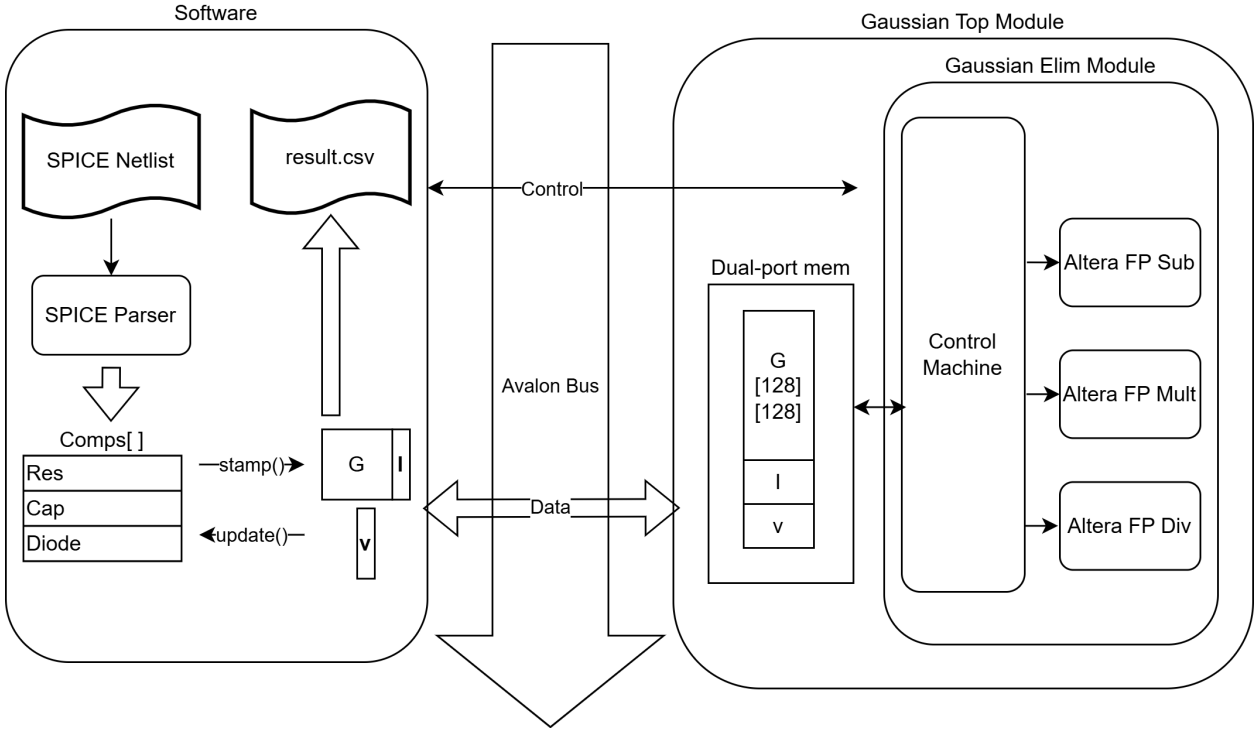
Contents

1 Introduction

For this project, we created an analog circuit simulator on the FPGA, inspired by professional tools like SPICE. Our simulator parses SPICE netlists into node and component representations. It then uses linear models to approximate the components, stamping their contributions into matrix equations. These equations are then offloaded to a hardware module, which solves them using Gaussian elimination. The resulting node voltages are returned to the software for further processing or visualization.

2 System Block Diagram

Below is our overall system diagram:



Details of the Circuit Simulator software algorithm are included in Section 4, and details/block diagrams of the Gaussian Elimination accelerator are included in Section 5.

3 Algorithms

The user provides input in the form of a SPICE-like netlist, specifying each component's type, node connections, and parameters (e.g., resistance, capacitance, etc.). The software parses the components and nodes into matrix equations using linear approximations and node voltage analysis. The FPGA performs Gaussian elimination to solve the resulting linear system, which operates in $O(n^3)$ time complexity. The simulation outputs the computed node voltages as a CSV file.

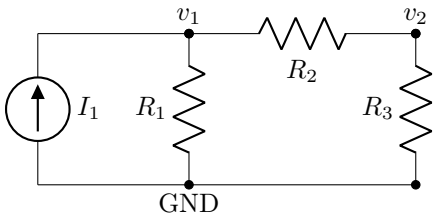
The software maintains two data structures:

- A component array that registers every circuit component, their connections, values, and memories
- The node matrix entries $G, \mathbf{v}, \mathbf{I}$.

3.1 Node Voltage Analysis

Node Voltage Analysis (NVA) is a method used to determine the voltage at various nodes in an electrical circuit. In the case of a purely resistor network, we can apply Kirchhoff's Current Law (KCL) at each node

and use Ohm's Law to express the currents. Let us consider a simple resistor network with current sources.



In this circuit, we can use Node Voltage Analysis to find the voltages v_1 and v_2 at nodes 1 and 2, respectively. The steps are as follows:

1. Apply Kirchhoff's Current Law (KCL) to each node, which states that the sum of currents leaving a node is zero.

   - At node v_1, the sum of the currents **leaving** v_1 (through the resistors and the current source I_1) should be zero:

   $$\frac{v_1}{R_1} + \frac{v_1 - v_2}{R_2} - I_1 = 0$$

   - At node v_2, again, the sum should be zero

   $$\frac{v_2}{R_2} + \frac{v_2 - v_1}{R_3} = 0$$

2. Construct a system. Define $G_n = \frac{1}{R_n}$. Move all the G (resistance) term to the LHS, and all I (current) terms to the RHS, We now have a system of linear equations:

   $$\begin{cases} G_1 v_1 + G_2(v_1 - v_2) = I_1 \\ G_3 v_2 + G_2(v_2 - v_1) = 0 \end{cases}$$

   Turning into a matrix, we have:

   $$\begin{bmatrix} G_1 + G_2 & -G_2 \\ -G_2 & G_2 + G_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} I_1 \\ 0 \end{bmatrix}$$

   $$G\mathbf{v} = \mathbf{I}$$

   We can then solve for v_1 and v_2.

   Since the components are linear, we can iterate through the component array and add their contributions to G and $\mathbf{I}$ one-by-one. This process is known as **stamping**.

Through nodal analysis, we can use Kirchhoff's Current Law to calculate the voltage at each node, when given n nodes with unknown voltage values. However, voltage sources are different because we cannot determine the current flowing through it purely by looking at its voltage value.

For ideal voltage soruces, we use modified nodal analysis (MVA), which adds an additional unknown: the current through the voltage source $\mathbf{v}$ vector. As a result, the size of the $\mathbf{v}$ vector is now n plus the number of unknown current values, which also means that the G and $\mathbf{I}$ vectors must increase in size accordingly so that the equations can be solved. This is solvable, since we also know that difference in voltage between the two nodes coinciding with the voltage source is equal to the voltage source's voltage. The matrix now has enough equations to solve for the unknowns, and is ready to be solved using any method that can solve simultaneous equations.

3.2 Software: Input Parsing

3.2.1 SPICE Parser

Our design supports a subset of circuits described using a SPICE netlist, specifically those that only have the components supported for this project. These components are:

- Ideal current and voltage sources
- Voltage controlled current and voltage sources
- Resistors
- Capacitors
- Inductors
- Diodes
- NMOS + PMOS

We only support a time-domain transient analysis. The timestep is set and a number of steps is calculated from the `.TRAN` line in the SPICE netlist.

3.2.2 Circuit Interpretation

In order to convert the netlist into a mathematically solvable form, we reconfigure the netlist as components and nodes, which are then "stamped" one by one onto the G, **v**and **I** matrices.

- Resistors and current sources: this is simple using node voltage analysis.
- Voltage sources: require Modified Nodal Analysis (subsubsection 4.1.1) to derive the current flowing through the current source.
- Time-varying and non-linear components: NVA cannot directly solve nonlinear terms. The software will translate these components into linear companion models (subsection 4.3 and subsection 4.2), which closely approximate the original components.

Some components may use more than one unknown in the system. From the netlist, the SPICE parser computes the total number of unknowns.

3.3 Component Array

We represent each circuit element using a `struct Component`, which contains

- **Type**: The component type: static, linear, or nonlinear
- **Function pointers** to add/update the component to/from the matrix
- **Parameters**: a union holding device-specific data (e.g. node indices, resistance, previous memory)

More details on the function implementation will be in section 4.

```
typedef struct {
    int n1, n2;                 // nodes
    float C, dt;                // device constants
    float i_prev, v_prev;       // previous values
} Cap;

typedef enum { STA_T, LIN_T, NL_T } CompType;
```

```
typedef struct Component {
    CompType type;
    void (*stamp_lin)(struct Component*);
    void (*stamp_nl) (struct Component*);
    void (*update_lin)   (struct Component*);
    void (*update_nl)   (struct Component*);
    union {
        VSrc   vsrc;
        Res    res;
        Cap    cap;
        Diode  dio;
        Nmos   nmos;
        // . . . other device types . . .
    } u;
} Component;
```

The netlist entries are parsed, and its corresponding `add_comp()` function is called to:

1. Assign the next free `Component` slot.
2. Set `type`, `stamp_lin`, `stamp_nl`, and `update`.
3. Initialize the union `u` with the device parameters.

The components are generally added in the order they are parsed, with the exception of voltage sources, which are added at the end so we can assign their internal nodes to available nodes.

Below is an example for adding a resistor. A resistor does not have memory or `update()` function.

```
Component comps[];
int ncomps = 0;

typedef struct {
    int n1, n2;
    float R;
    // memory variables will also be here
} Res;

void add_res(int n1, int n2, float R) {
    Component *c  = &comps[ncomps++];
    c->type       = STA_T;
    c->stamp_lin  = res_stamp;      // stamp function
    c->stamp_nl   = NULL;
    c->update_lin = NULL;           // no update function
    c->update_nl  = NULL;
    c->u.res      = (Res){
        .n1       = n1,
        .n2       = n2,
        .R        = R
    };
}
```

To stamp a resistor to the matrix, we increment the corresponding entries:

```
extern float G[MAT_SIZE][MAT_SIZE];
extern float Ivec[MAT_SIZE];

void res_stamp(Component *c) {
    Res *r = &c->u.res;
```

```
    int  n1 = r->n1, n2 = r->n2;
    float g = 1.0f / r->R;

    // ground node is -1 here
    if (n1 != -1) Gm[n1][n1] += g;
    if (n2 != -1) Gm[n2][n2] += g;
    if (n1 != -1 && n2 != -1) {
        Gm[n1][n2] -= g;
        Gm[n2][n1] -= g;
    }
}
```

3.4 Matrix Solver

To solve the system of equations, our design uses Gaussian elimination as the algorithm of choice. This is done on the FPGA, which takes the matrices and returns the solved unknowns.

For each timestep, the static components are stamped first and not updated since their values are consistent. Then, for each iteration, the nonlinear and time-varying components are stamped, and the G and $\mathbf{I}$ matrices are passed to the FPGA via memory mapping. The matrix solver on the FPGA is invoked via `ioctl()`, and the resulting $\mathbf{v}$ vector is copied back. The iterations continue until convergence.

3.5 Data Visualization

The simulation outputs the node voltages at each timestep (the time duration is specified within the SPICE netlist) to a CSV file, which can be easily visualized using any plotting tool. The output can be limited to certain nodes, which can be passed in as optional command-line arguments.

4 Models

4.1 Modeling Static Components

4.1.1 Independent Voltage Source

Node analysis favors current sources. We accommodate for ideal voltage sources using **modified** nodal analysis (MNA). For independent ideal voltage sources, we just need to add an extra row to our current matrix.

Suppose we add a voltage source V_s parallel to R_2 (between v_1 and v_2). This addition fixes $v_1 - v_2$, but adds a new branch of current I_{V_s} through the voltage source. Now our equation becomes

$$\begin{cases} G_1 v_1 + G_2(v_1 - v_2) + I_{V_s} = I_1 \\ G_2 v_2 + G_3(v_2 - v_1) - I_{V_s} = 0 \\ v_1 - v_2 = V_s \end{cases}$$

For matrix notation, we'll make I_{V_s} as a new unknown. In the simulator, I call it `ni`, the "internal" node

$$\begin{bmatrix} G_1 + G_2 & -G_2 & 1 \\ -G_2 & G_2 + G_3 & -1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ I_{V_s} \end{bmatrix} = \begin{bmatrix} I_1 \\ 0 \\ V_s \end{bmatrix}$$

Let's put in some real values: $I_1 = 1$, $V_s = 5$, $R_1 = R_2 = R_3 = 2$ ($G = 0.5$ for all G)

$$\begin{bmatrix} 1 & -0.5 & 1 \\ -0.5 & 1 & -1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ I_{V_s} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix}$$

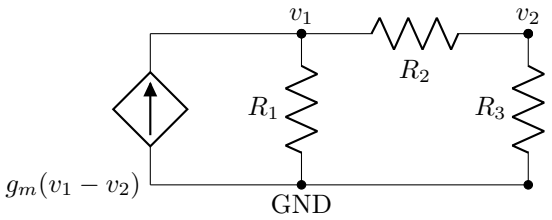
Solving the matrix,

$$\begin{bmatrix} v_1 \\ v_2 \\ I_{V_s} \end{bmatrix} = \begin{bmatrix} 3.5 \\ -1.5 \\ -3.25 \end{bmatrix}$$

The result is consistent with the SPICE simulation: https://tinyurl.com/22zch9en

4.1.2 Voltage Controlled Current Source

Independent current sources are simple: just add the RHS vector by the current entering the node.



The change is simple, also adding on the RHS by the current amount, which will be moved to the LHS

$$\begin{cases} G_1 v_1 + G_2(v_1 - v_2) = g_m(v_1 - v_2) \\ G_3 v_2 + G_2(v_2 - v_1) = 0 \end{cases}$$

$$\begin{bmatrix} G_1 + G_2 - g_m & -G_2 + g_m \\ -G_2 & G_2 + G_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that solution $\mathbf{v}$ will be 0, since the RHS is 0. There are no independent sources to power it.

4.1.3 Voltage Controlled Voltage Source

Voltage controlled voltage source (VCVS) is similar to independent voltage sources. Use an internal node to set up the matrix, and substitute the source equation to the RHS, replacing V_s.

With VCVS, we can model an *operational amplifier*: a VCVS with a very high gain (E. 1 million) between the two input terminals.

There are also two types of controlled current sources. Our model did not include them for simplicity.

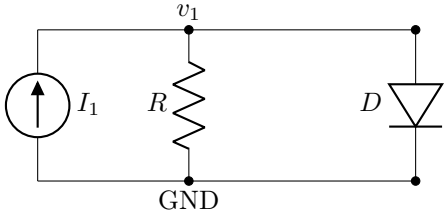
4.2 Modeling Nonlinear Static Components

4.2.1 Diodes: Newton-Raphson Method

Diodes has a nonlinear current-voltage relation:

$$i_d = I_s(e^{v_d/V_t} - 1)$$

where I_s, V_t are constants.



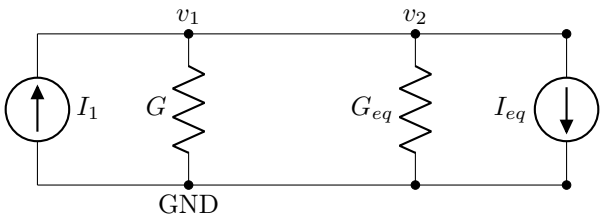
At step 0, let G_{eq} be derivative $\frac{di_d}{dv_d}$, where $i_d = i_{d0}$, $v_d = v_{d0}$:

$$G_{eq} \equiv \frac{di_d}{dv_d} = \frac{I_s}{V_t} e^{v_{d0}/V_t}$$

We apply the Newton–Raphson algorithm to build a linear estimate model around an initial guess.

$$\begin{aligned} i_d &\approx i_{d0} + G_{eq}(v_d - v_{d0}) \\ &= (i_{d0} - G_{eq} v_{d0}) + G_{eq} v_d \end{aligned}$$

Fortunately, any linear relation can be modelled by circuit elements. The constant term can be represented by a current source $I_{eq} = (i_{d0} - G_{eq} v_{d0})$, and the linear term can be represented by a resistor $G_{eq} = \frac{1}{R_{eq}}$. We add them to the matrix. This process is called **stamping**.



We can **solve** this linear circuit and get the node voltages.

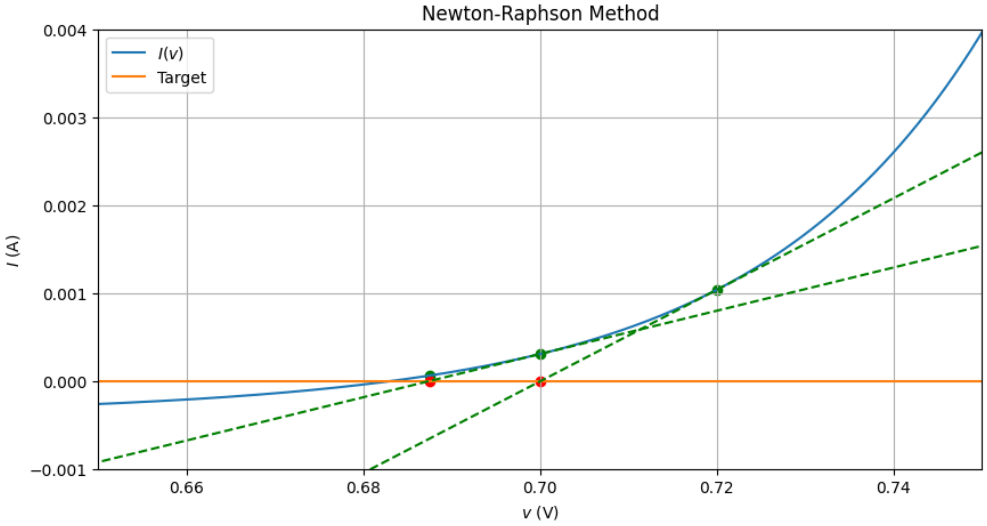
From the node voltages, we **update** our initial guess.

The above procedure of stamp-solve-update is repeated until the solution converges, which is detected when the deviation of the solution $\mathbf{v}$ is below a threshold.

In this example diode circuit, we have the relation

$$\frac{v_1}{R} + I_s(e^{v_1/v_t} - 1) - I_1 = 0$$

Choosing $I_1 = 1\,\text{mA}$, $I_s = 1 \times 10^{-15}\,\text{A}$, $R = 1\,\text{k}\Omega$, $v_t = 25.8\,\text{mV}$, if we choose the initial guess of $v_1 = 0.72\,\text{V}$, we will get the following iterations:



In summary, at a particular time (operating point), we perform the following loop:

1. `stamp()` all components. For nonlinear components,

(a) Make an initial guess on v_{d0}.

(b) Using v_{d0}, calculate I_{d0}, G_{eq}, I_{eq}

(c) Add G_{eq} and I_{eq} onto G and $\mathbf{I}$

2. Solve the matrix $G\mathbf{v} = \mathbf{I}$
3. `update()` the new v_{t0} from the $\mathbf{v}$ of the next iteration
4. Check for convergence. Finish if it's less than a threshold.

4.2.2 MOSFETs: Multivariable Newton-Raphson

A MOSFET is a 3-terminal nonlinear device. There are 3 terminals: gate, drain, and source. For an NMOS, v_{GS} and v_{DS} controls i_{DS}. We used a piecewise model:

- If $v_{GS} < V_T$, a threshold voltage, the MOS is off. $i_{DS} = 0$
- If $v_{GS} \geq V_T$, but $v_{DS} < v_{GS} - V_T$, the MOS is in the linear region. $i_{DS} = \beta(v_{GS} - V_T)v_{DS}$
- If $v_{GS} \geq V_T$, and $v_{DS} \geq v_{GS} - V_T$, the MOS is in saturation. $i_{DS} = \frac{1}{2}\beta(v_{GS} - V_T)^2$

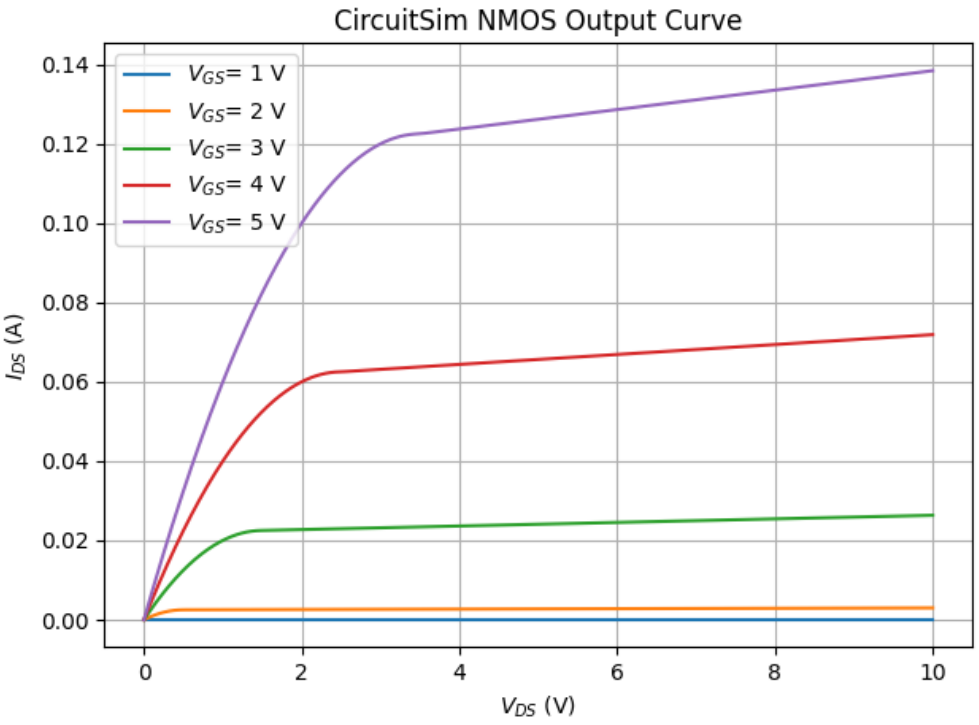
Above are just to demonstrate what "cutoff", "linear", and "saturation" mean in relation to the voltages. In reality, the device must be continuous.

Also, some part of the equation is not using all variables, which may under-define the matrix, making it singular (subsection 9.3). To account for physical hassles and avoid divergence or singularity as much as possible, the saturation region needs a "channel width modulation" term $(1 + \lambda v_{DS})$. To ensure continuity between the regions, we made the terms $(1 + \lambda(v_{DS} - (v_{GS} - V_T)))$. Since λ is a small value, this shouldn't affect anything too significant. The equation becomes:

- **Cutoff:** $i_{DS} = kv_{DS}$
- **Linear:** $i_{DS} = \beta((v_{GS} - V_T)v_{DS} - \frac{1}{2}v_{DS}^2)$
- **Saturation:** $i_{DS} = \frac{1}{2}\beta(v_{GS} - V_T)^2(1 + \lambda(v_{DS} - (v_{GS} - V_T))$

where β is the current coefficient specific to the mosfet, k is a small constant for off current, and λ is a small constant for channel width modulation

Below are the I-V curves of our NMOS model, with $\beta = 0.02, V_T = 1.5\,\text{V}, \lambda = 0.01, k = 1 \times 10^{-8}$: The choice of k makes the cutoff discontinuity negligible compared our floating point accuracy. The error would be fine for low-voltage digital simulation.



We use the same approach: Newton's method. This time, we are making a tangent *plane* approximation for each region, taking the partial derivative $g_m \equiv \frac{\partial i_{DS}}{\partial v_{GS}}$ and $g_d \equiv \frac{\partial i_{DS}}{\partial v_{DS}}$

Here's what electrical engineers call a "small-signal" model:

$$\begin{aligned} i'_{DS} &= i_{DS} + g_m(v'_{GS} - v_{GS}) + g_d(v'_{DS} - v_{DS}) \\ &= I_{eq} + g_m v'_{GS} + g_d v'_{DS} \end{aligned}$$

where $I_{eq} = i_{DS} - g_m v_{GS} - g_d v_{DS}$

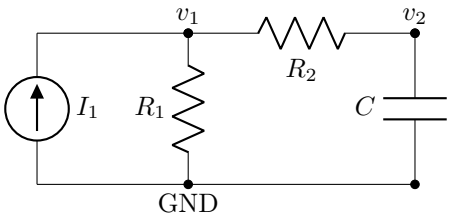
Below is our linear model. The sum of the branches is equal to i'_{DS}.

4.3 Modeling Time-varying Components

Now we have a general workflow for static components. For more time-varying components, we will need to employ linear companion models, which are linear approximations on the time axis.

We use Backward Euler's method to create companion models for capacitors and inductors

4.3.1 Capacitors



In the circuit above, using backward Euler's method, we compute the derivative of v_C for the next time step and use it to evaluate the node voltages.

$$
\begin{aligned}
i(t) &= C\frac{dv_C}{dt} = C\frac{d(v_2 - 0)}{dt} \\
i(t_0 + \Delta t) &\approx C\frac{v_2(t_0 + \Delta t) - v_2(t_0)}{\Delta t} \\
v_2(t_0 + \Delta t) &= v_2(t_0) + \frac{\Delta t}{C} i(t_0 + \Delta t) \\
v_2(t) &= v_2(t_0) + \frac{\Delta t}{C} i(t)
\end{aligned}
$$

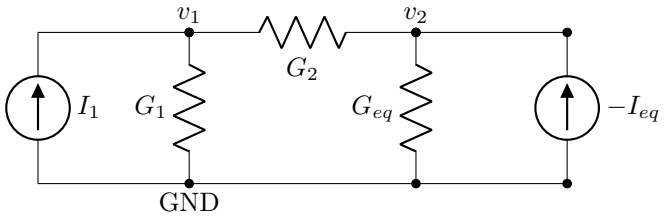
We now have a **linear** approximation of v_2 as a function of t.

Let $V_{eq} = v_2(t_0)$ and $R_{eq} = \frac{\Delta t}{C}$, the equation becomes

$$v_2(t) = V_{eq} + R_{eq} i(t)$$

This capacitor is converted into a voltage source and a resistor in series, at this particular timestamp, which is then converted to a current source in parallel with a resistor (Norton equivalent). Let $I_{eq} = -\frac{V_{eq}}{R_{eq}}$ and $G_{eq} = \frac{1}{R_{eq}}$

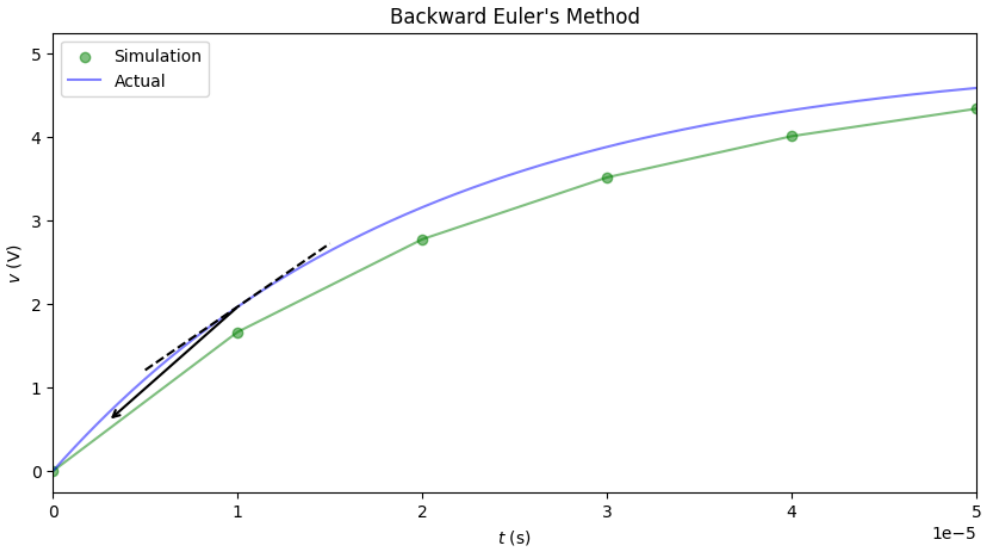
$$i(t) = I_{eq} + G_{eq} v_2(t)$$



Intuitively, as the capacitor's voltage v_C charges up, $|I_{eq}|$ approaches $G_{eq}V_{eq}$, cancelling the current through G_{eq}, so the steady-state current is zero.

Similar to Newton's method, we can add the contribution of I_{eq} and G_{eq} to the matrix, solve, and update the estimate in the next time step.

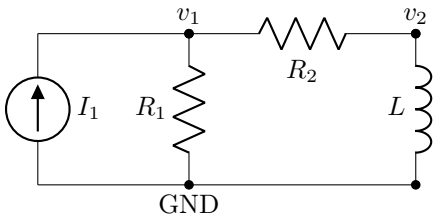
Below is a visualization of how CircuitSim solves the circuit above. We compute the derivative of the next time step, and we send it "back" to the current time step, and it becomes the slope of the linear approximation.



4.3.2 Inductors

Inductors are essentially very similar. An ideal inductor has $i - v$ relationship:

$$v(t) = L\frac{di}{dt}$$

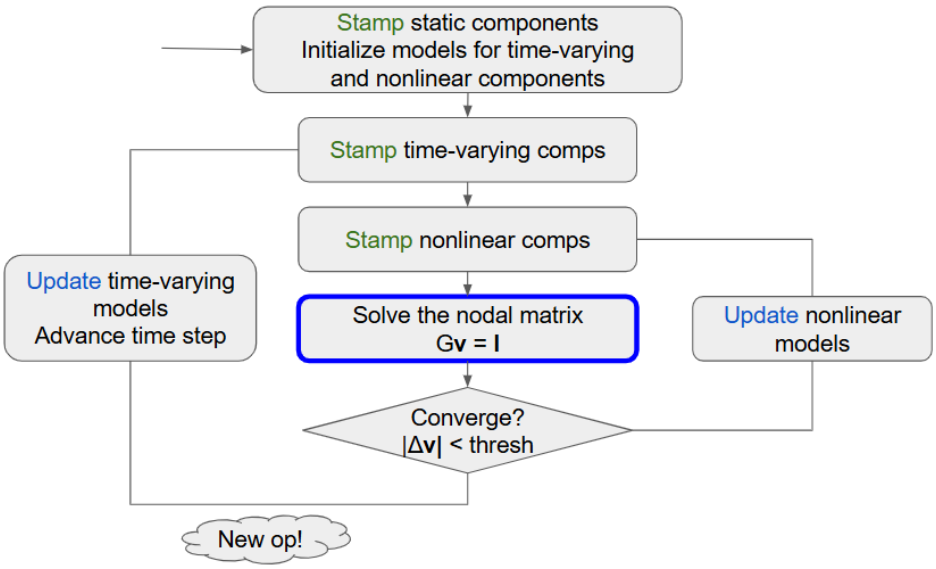


$$\begin{aligned} v(t) &= L\frac{di_L}{dt} \\ v_2(t+\Delta t) - 0 &\approx L\frac{i_L(t_0+\Delta t) - i_L(t_0)}{\Delta t} \\ i_L(t_0+\Delta t) &= i_L(t_0) + \frac{\Delta t}{L}v(t_0+\Delta t) \end{aligned}$$

Analogously, let $I_{eq} = i_L(t_0)$ and $G_eq = \frac{\Delta t}{L}$, we convert the inductor into a current source and resistor in parallel.

4.4 Simulation Summary

Below is the flow chart for the entire simulation loop For each time step, we run the Newton loop to ensure the nonlinear devices converge. Then, we advance the time step and run the Euler loop.



- `stamp()` iterate over the `Comps[]` array, loading every relevant component to $G, \mathbf{I}$
- Hardware solves the matrix
- `update()` iterate over the `Comps[]` array, updating their memory from the results of $\mathbf{v}$

5 Gaussian Elimination Hardware Accelerator

5.1 Theory

For demonstration, consider a voltage-source demo circuit, whose nodal equations yield the $3\times$ 3 system

Note that everything is 0-indexed here.

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 1 & -1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix}$$

5.1.1 Forward Elimination

Forward elimination is performed on each column k to produce an upper-triangular matrix.

Because the diagonal entries will be divided in this stage, we use **partial pivoting** to swap rows and place the largest available coefficient on the diagonal. This avoids zero or near-zero divisors (and the excessively large quotients they'd produce), thereby improving floating-point stability and overall numeric performance.

For each column k, scan the entries G_{ik} for $i \geq k$. The one with the largest magnitude will be swapped with G_{kk}, so that the diagonal entry G_{kk} holds the maximal pivot.

For column 0, $G_{10} = -2$ is the pivot, so we swap R_0 with R_1.

$$\begin{bmatrix} -2 & 1 & -1 \\ 1 & -2 & 1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 5 \\ 1 \end{bmatrix}$$

For each row j, we then subtract mR_0 from R_1 and R_2, where $m = \frac{G_{i0}}{G_{00}}$ to make column 0 upper triangular.

$$\begin{bmatrix} -2 & 1 & -1 \\ 0 & -1.5 & 0.5 \\ 0 & -0.5 & -0.5 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 5 \end{bmatrix}$$

The same operations are performed on the rest of the columns to eventually make the matrix upper-triangular.

$$\begin{bmatrix} -2 & 1 & -1 \\ 0 & -1.5 & 0.5 \\ 0 & 0 & -2/3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 14/3 \end{bmatrix}$$

5.1.2 Back substitution

From the upper triangular matrix, we can directly solve $c = -7$. Then, for each iteration, subtract each solved variable from the row, and solve a new unknown

Substitute $c = -7$ to R_1, we get $b = -3$; substitute $b = -3, c = -7$ to R_0, we get $a = 2$ Solution: $a = 2, b = -3, c = -7$

5.1.3 C prototype

Below is a software prototype of the algorithm:

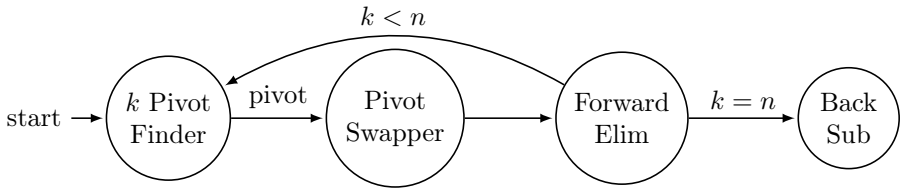
```c
int gaussian_elim(int n) {   // n: number of unknowns
    for (int k = 0; k < n; k++) {
        // 1. pivot finder
        int piv = k;
        for (int i = k+1; i < n; i++)
            if (fabsf(G[i][k]) > fabsf(G[piv][k]))
                piv = i;
        if (fabsf(G[piv][k]) < 1e-12f)
            return 1;       // singular matrix

        // 2. pivot swapper
        if (piv != k) {
            for (int j = k; j <= n; j++)
                { float t = G[k][j]; G[k][j] = G[piv][j]; G[piv][j] = t; }
            t = I[k]; I[k] = I[piv]; I[piv] = t;
        }

        // 3. elimination
        for (int i = k+1; i < n; i++) {
            float m = G[i][k] / G[k][k];
            for (int j = k; j <= n; j++)
                G[i][j] -= m * G[k][j];
        }
    }
    // 4. back substitution
    for (int i = n-1; i >= 0; i--) {
        float sum = I[i];
        for (int j = i+1; j < n; j++)
            sum -= G[i][j] * v[j];
        v[i] = sum / G[i][i];
    }
    return 0;
}
```

5.2 Hardware Block Design

The hardware does the Gaussian elimination, from the $G, \mathbf{I}$ prepared from the software, and returns $\mathbf{v}$ to the software. It mainly consists of three parts:

- A memory block storing the G, $\mathbf{v}$, $\mathbf{I}$ values
- Altera floating point IP to perform floating point subtraction, multiplication, and division
- A control machine that coordinates with the arithmetic IPs and reads and writes the results to the appropriate places in the memory.

5.3 Control State Machine



We designed a control state machine to manage interactions between the hardware, floating-point units, and memory blocks. At a high level, there are four main functional blocks, as discussed in previous sections. Each block contains multiple sub-states to iterate through the matrix.

Initially, we designed each block independently. Since the overall process is linear, we combined the blocks into a single unified state machine with of 44 states, encoded using 6 bits.

To validate the design, we first implemented a cycle-accurate simulation of the state machine in C, using dummy models for sequential memory access and pipelined floating-point operations. We tested this simulation with randomly generated matrices to verify correctness. Once validated, we translated the C model into SystemVerilog.

```
typedef enum logic [5:0] {
        GS_IDLE,
    // pivot finder
        PF_INIT, PF_READ_DIAG, PF_SCAN_CHECK, PF_READ_VAL, PF_EVALUATE,
    // pivot swapper
        PS_SWAP_G_A, PS_SWAP_G_B, PS_SWAP_G_WA, PS_SWAP_G_WB,
        PS_SWAP_I_A, PS_SWAP_I_B, PS_SWAP_I_WA, PS_SWAP_I_WB,
    // elimination
        EL_INIT_READ_PIVOT, EL_INIT_SETUP, EL_READ_AIK, EL_DIV_START, EL_DIV_WAIT,
        EL_COL_SETUP, EL_READ_COL, EL_READ_ROW, EL_MUL_START, EL_MUL_WAIT,
        EL_SUB_START, EL_SUB_WAIT, EL_WRITE_COL, EL_COL_INCREMENT, EL_ROW_INCREMENT,
    // back substitution
        BS_READ_I, BS_SETUP, BS_READ_A, BS_READ_V, BS_MUL_START, BS_MUL_WAIT,
        BS_NEXT_CHECK, BS_READ_DIAG, BS_DIV_START, BS_DIV_WAIT, BS_DIV_WRITE, BS_ROW_DEC,
    // control states
        GS_CHECK_K, GS_DONE, GS_FAILED
} state_t;
```

For simplicity, we will not show the entire state diagram here. The full states and transitions can be found in the appendix.

5.4 Protocol

The software instructs the Gaussian block to start computing, and polls the hardware `done` signal. We used the following protocol:

1. Before starting, the software sends the `reset` flag to clear all of the hardware's internal registers
2. The software turns off the `reset` flag and activates the `go` flag to kick start the hardware. The `go` flag may stay on for a few more cycles, which will be ignored by the hardware before it's done
3. After the hardware is done, it raises the `done` flag

- `Failed`: Something internal failed in the Gaussian module (E. `Nan` results, division by zero)
- `Singular`: The input matrix is singular.

The software keeps polling the result from the Avalon bus until the `done` flag is raised.

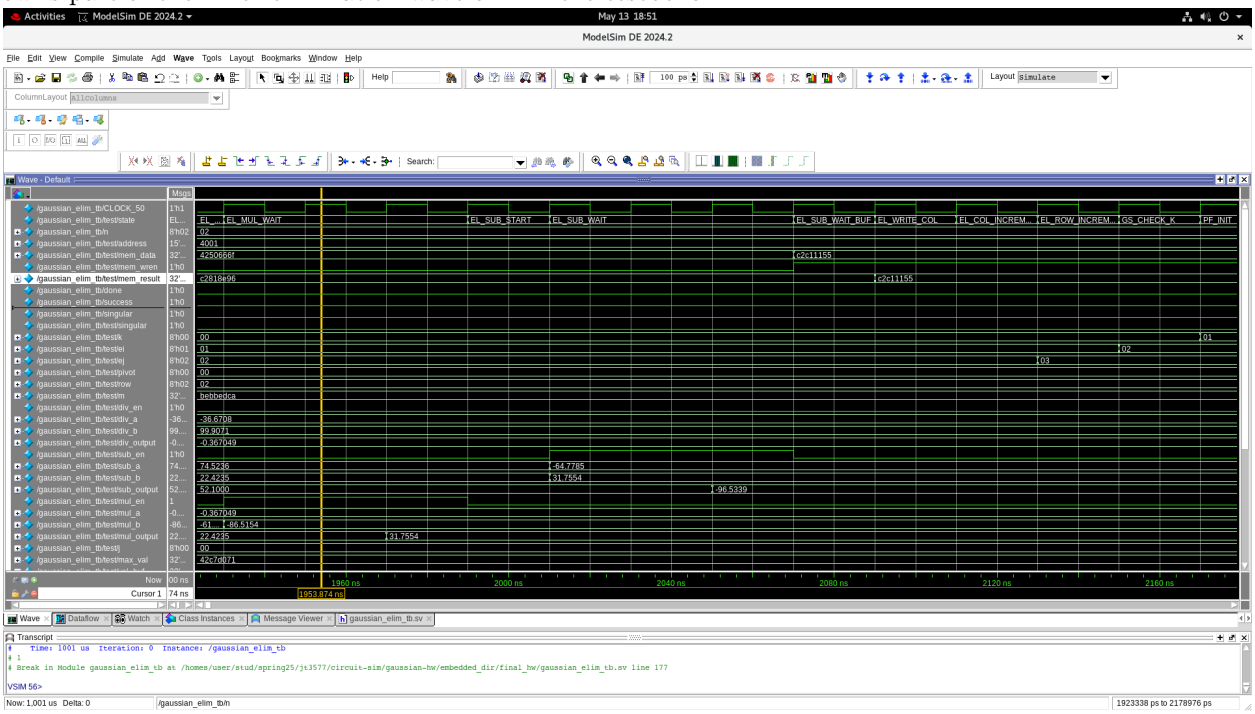
The detailed address map of the software-hardware interface are in section 7

5.5 Programmer's Process

- Fundamentally, converting such an extensive state machine with sequential steps into hardware is difficult. After taking the software prototype in C, and writing the cycle-accurate C simulation, we moved to SystemVerilog. We were able to translate pretty well our cycle-accurate C simulation into HDL, with one major exception. Our C simulation assumed instant memory access.
- When writing our SystemVerilog, we had the foresight to recognize that this wouldn't be the case, so we performed "pipelining", by preparing the address one cycle early. In this way, the address would then be ready in the cycle we needed. However, memory requires one cycle to access, from time address is asserted to output. This two-cycle memory access then needed to get pipelined as well. Occasionally, we would need to add in a stall state to prevent data hazards, but for the most part, intensive tracing of all the "grandparent" states of a state requiring memory was necessary.
- One additional key component of our work was the addition of custom floating-point SystemVerilog functions. These enabled us to perform absolute value, inequalities, and then yield output checks for all our floating-point IPs. Rather than stalling our whole process with more FP operations, we decided to perform these in combinatorial logic.
- The debugging process required ModelSim waveforms (see below), but one key process we did was checking the exact output of our cycle-accurate C simulation with the waveform values of the module. This proved very helpful.

5.6 Waveform

Below is part of the inner elimination waveform in the testbench.

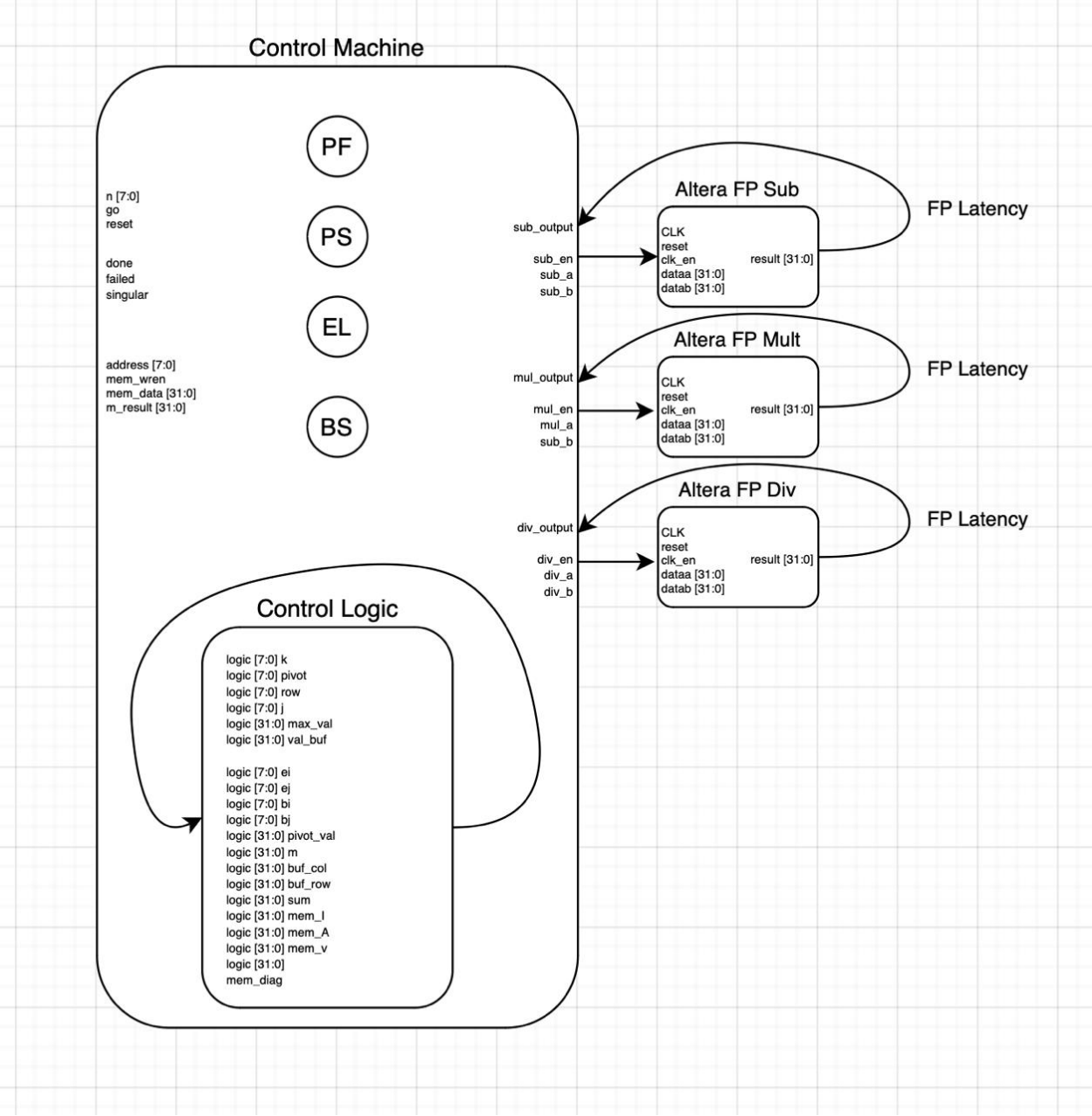


The Gaussian block prepares the inputs for the floating point modules and waits for the result. If the result is not immediately used, it is saved in temporary registers.

Based on the current indices, it calculates the memory address and reads/writes to the memory block.

5.7 Diagram

Below is a summary of our Gaussian elimination module, its pins, and internal registers



6 Floating Point Operations

6.1 Floating Point IP

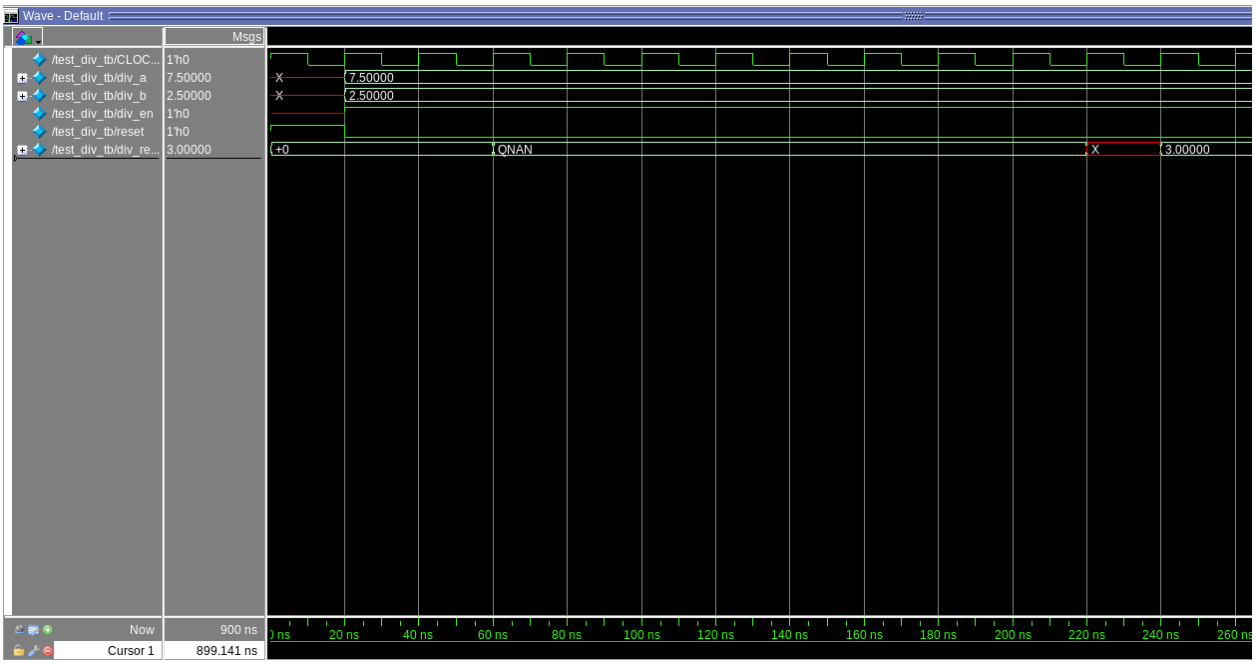
We intend to use the Floating Point IP provided by Altera. The general overview of all blocks is that they take in two inputs, a clock, and an enable switch. The floating point block will have significant latency,

which will take several clock cycles to return the result.

The entire module can be seen as a pipeline process. When `enable` is on, the inputs are fed into the pipeline, and the outputs will be available after the specified cycle. Note that the output may only hold for one cycle, until the next input overwrites it.

To ensure the result is read at the exact right clock cycle, we use a counter with a threshold signal to read the result.

Below is the waveform for the Altera FP division module



At 50 MHz, the division module's delay is 11 cycles. Note that intermediary values may be invalid or undefined. We need to ensure that the result is read exactly at 11 cycles after the enable signal is high. If the result is accessed after that, we will need registers to store the temporary results. Furthermore, the enable signal needs to be maintained high for this whole time to ensure the proper output. (See waveform above).

6.2 Getting ModelSim to Work

An important step in our project was to verify that it works in simulation. We created a test bench for our Verilog file and conducted it in Quartus and ModelSim. Furthermore, we used three instances of the `FP_FUNCTIONS` Intel FPGA IP to get our `fp_mult`, `fp_div`, and `fp_sub` blocks. However, this led to errors in ModelSim.

We followed the steps from Peiran's ModelSim Tutorial. To recap:

1. Have Design Under Test (DUT) and testbench SV file ready.
2. Configure Quartus with Testbench settings.
3. Start Analysis and Elaboration.
4. Run RTL simulation.

Once the ModelSim window pops up, we encountered the following error messages:

```
Error: xx.vhd(39): Library altera_mf not found.
Error: xx.vhd(40): (vcom-1136) Unknown identifier "altera_mf".
```

Not to worry! This happened because, despite setting the FP_FUNCTION to be built for SystemVerilog, Quartus still generated it in VHDL. To fix this, open your generated `.do` file (via `Tools > Tcl > Execute Macro...`) and add (and adjust) these lines:

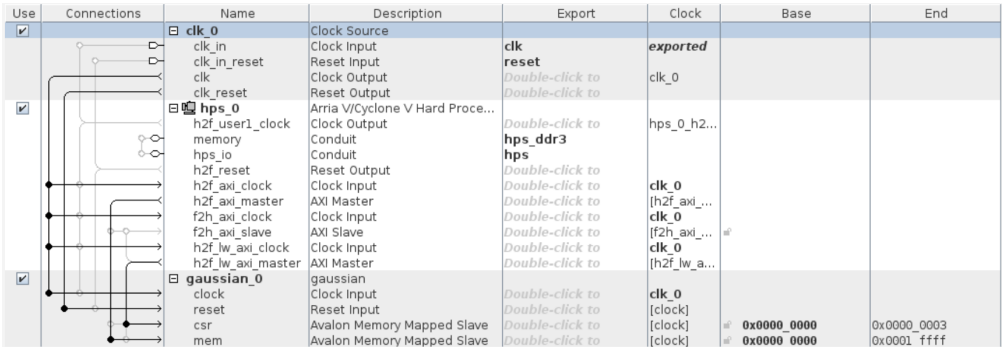
```
vlib verilog_libs/altera_mf_ver
vmap altera_mf_ver ./verilog_libs/altera_mf_ver
vlog -vlog01compat -work altera_mf_ver {/tools/intel/intelFPGA/21.1/quartus/eda/sim_lib/altera_mf.v}
```

Repeat similarly for `altera_lnsim_ver` and `lpm_ver`. Finally, locate the `vsim` invocation near the bottom of the file and add:

```
vsim <your_top_level> -L altera_mf_ver -L altera_lnsim_ver -L lpm_ver
```

(Don't forget to include the `-L` flags for your other libraries.)

7 Hardware/Software Interface



The main interactions between the hardware and software revolve around the transfer of the G matrix and V/I vectors to the FPGA and sending the result of the Gaussian elimination algorithm from the hardware to software.

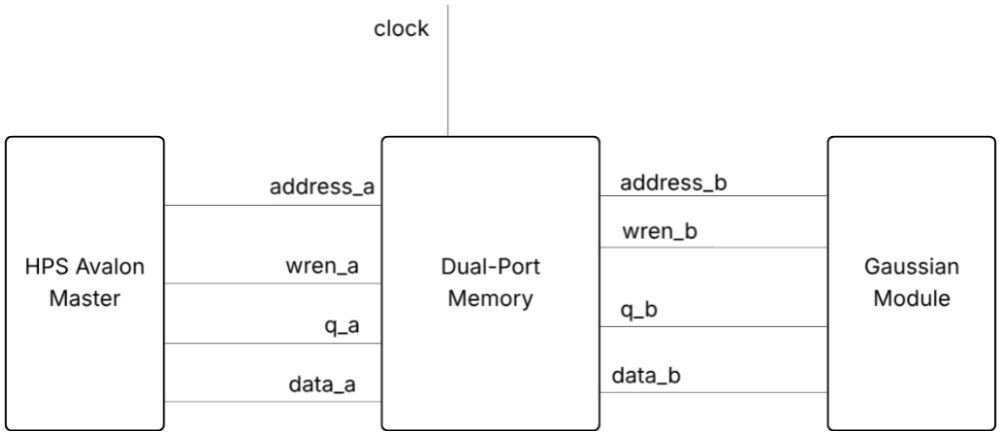
The software facilitates this by allocating space for the matrix and two vectors in software, and then "stamping" it via the aforementioned methods. Critically, this software representation is copied over to a 128 kilobyte M10K memory block via mmap and a modified memcpy implementation.

The memory is a dual port design, allowing for simultaneous read/write from both the HPS and the FPGA. Due to the sequential nature of the software and hardware components, these components are not used in tandem. However, the dual port design makes it significantly easier than our initial one port design to integrate between the HPS and FPGA.

The entire software suite for our hardware sits within the Gaussian module shown in the above diagram taken from Platform Designer.

The csr (Control Signal Registers) section of the Gaussian module connects to h2f_lw_axi_master, a lightweight AXI master that controls our module via the specifications outlined in the register map section below.

The mem (memory) section of the Gaussian module connects to the h2f_axi_master which under the hood, controls how the software writes to the FPGA's memory block, which sits within the Gaussian module. The memory block within the Gaussian module is connected to both the HPS via the aforementioned interface and to the Gaussian elimination module that actually runs the algorithm. This top level Gaussian module also decodes the control signals from the HPS, sending them to the Gaussian elimination module.



7.1 Control Register Map

Matrix dim (8 bits)			
G	R	Unused (6 bits)	
D	E	S	Unused (5 bits)

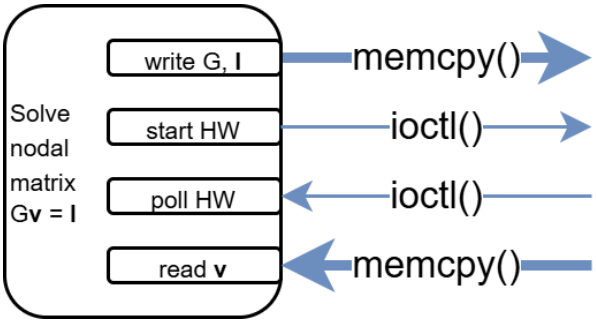
- The first 8 bits encode n, the size of the matrix the loop is going through
- The next 2 bits are the Gaussian module input flags: `go`, `reset` with 6 bits left unused
- The final 8 bits are the Gaussian module output flags: `done`, `error`, `singular` and 5 unused bits

Before writing to the FPGA memory, the software driver raises the reset bit via ioctl, which the FPGA reads, clearing the memory from the last iteration. It idles until the software driver writes to the memory, after which the go bit will be raised.

7.2 Software Call

To solve a system, the software would first write

1. Copy G, $\mathbf{I}$ to the dual-port memory
2. Send the control signals for the Gaussian module to start
3. Poll for the control signals from the Gaussian module
4. Copy the result $\mathbf{v}$ from the memory



7.3 Performance

Flow Summary	
Flow Status	Successful - Wed May 14 01:41:40 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	soc_system
Top-level Entity Name	soc_system_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	1,744 / 32,070 (5 %)
Total registers	2095
Total pins	362 / 457 (79 %)
Total virtual pins	0
Total block memory bits	1,082,368 / 4,065,280 (27 %)
Total DSP Blocks	5 / 87 (6 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	1 / 4 (25 %)

Our acclerator state machine only uses 5% of the total logic. It uses 27% of the memory bits to store the floating point entries.

8 Cool Applications

8.1 Linear Circuits

8.1.1 Passive LRC Circuit

Second-order LRC circuits are a nightmare for EE sophomores. Let us see if CircuitSim is able to handle them.

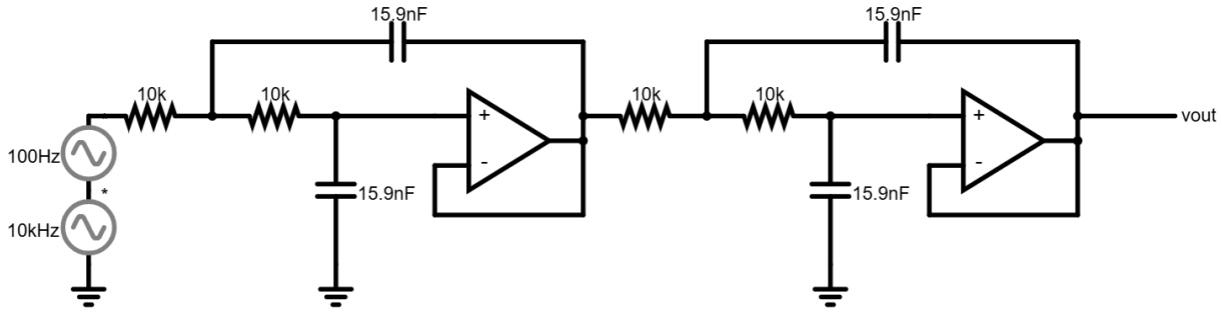
We plotted the voltage across the inductor and capacitor to observe the behavior

- The inductor "absorbs" all the sudden changes of the square wave
- The capacitor oscillates at a relatively low frequency.

The simulation result is exactly the same as expected.

8.1.2 4th-Order Filter

To fully extract the potential of CircuitSim to handle analog circuits, In this section, we build an active 4th-order low-pass filter by cascading two second-order filters of the Sallen-Key topology.



Here is the link to the circuit setup: https://tinyurl.com/22pxyowb

The op-amps are modelled with voltage-controlled voltage sources, with a gain of 1M, which is very close to an ideal op-amp.

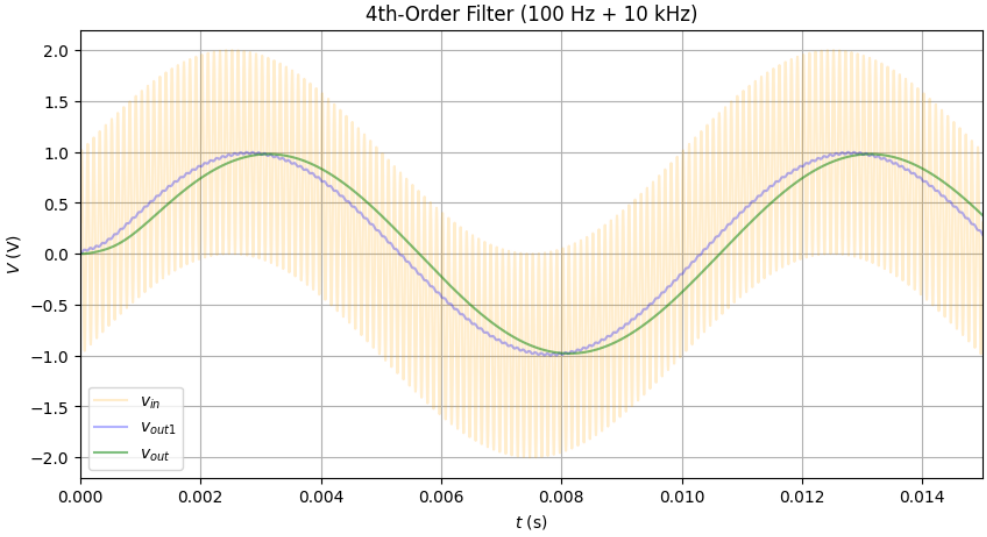
Assuming identical resistor/capacitor values, the transfer function of the 4th-order Sallen-Key topology is:

$$H(s) = (H_1(s))^2 = \frac{\omega_0^4}{\left(s^2 + 2\omega_0 s + \omega_0^2\right)^2}$$

where $\omega_0 = \frac{1}{RC}$.

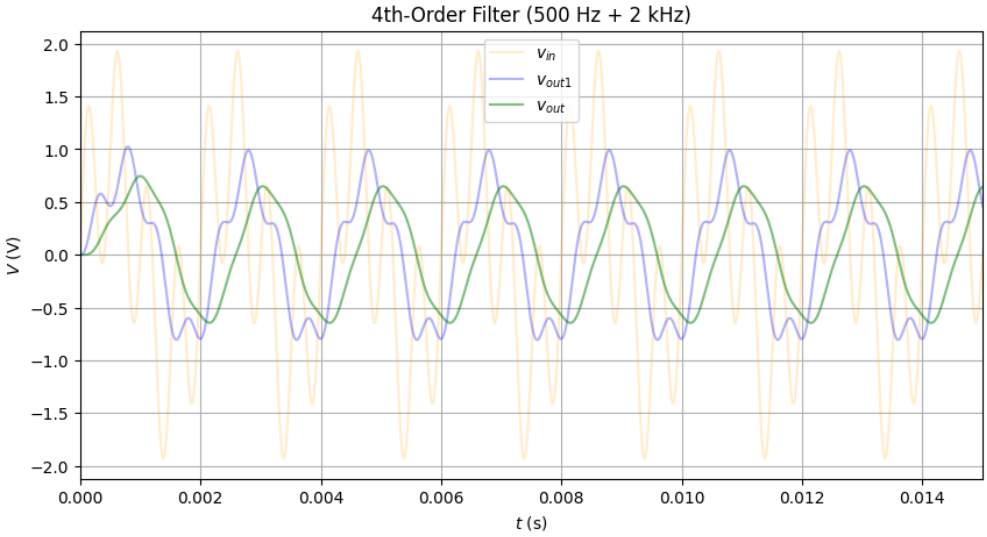
Suppose we want the cutoff frequency to be 1 kHz. Choose $R = 10\,\text{k}\Omega$ and $C = 15.9\,\text{nF}$.

CircuitSim does not yet support frequency-domain analysis. To test the filter, we input a mix of 100 Hz and 10 kHz frequencies and observe the time-domain response:



As you can see, the first stage (blue) filters out most of the 10 kHz component, with small sawtooth, still. The second stage (green) further smoothens the output.

For a stricter test, let's input a two-tone sine wave of frequency 500 Hz and 2 kHz.

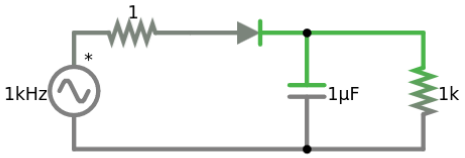


This time, the first stage did not filter well, but the second stage does the job and smoothes the output. The two tests above are nearly identical to the result from LTSpice.

8.2 Nonlinear Circuits

8.2.1 AC Rectifier

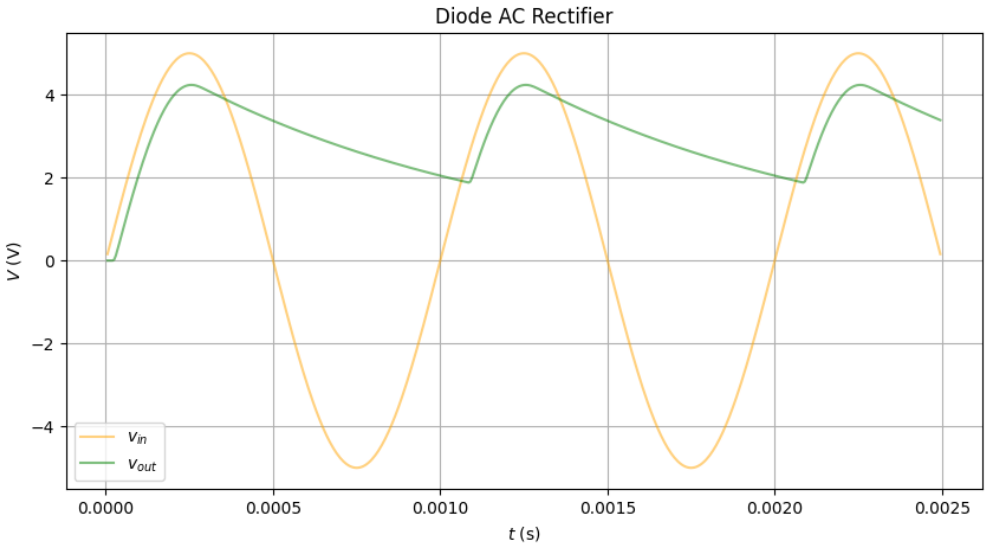
We can create a simple AC-DC rectifier with a diode and a capacitor A small resistance is placed in series with the source to prevent shorting the diode initially.



CircuitSim accurately simulates the following:

- At the positive half-cycle, the diode conducts with a small voltage drop, powering up the 1k load resistor and charging the capacitor
- At the negative half-cycle, the diode blocks the capacitor. The capacitor discharges exponentially to power the load resistor.

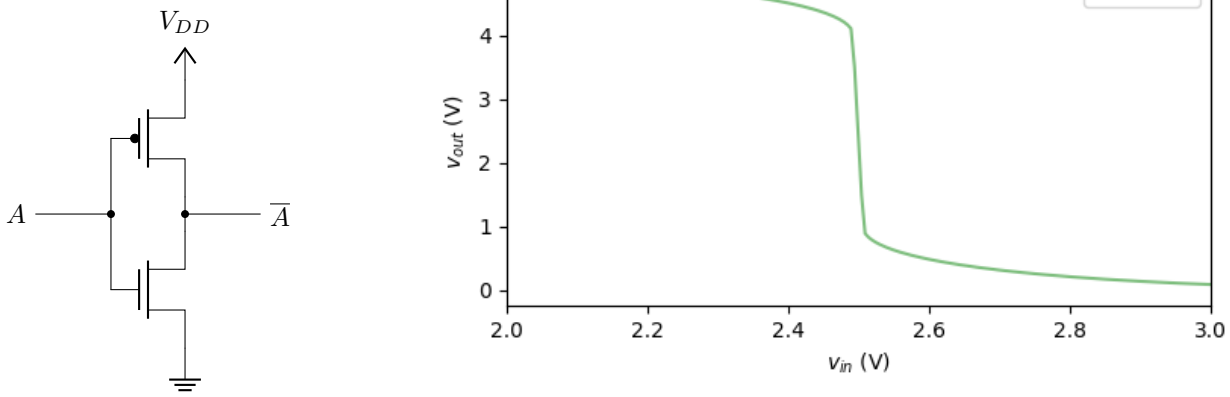
Below is the iconic regulator output curve:



8.2.2 CMOS: from Analog to Digital

With MOSFETs, we can step into the digital realm, using our simulator with an analog core.

A CMOS NOT gate is simply a PMOS and an NMOS. When the input voltage is high, the PMOS is off, and the NMOS is on, pulling v_{out} low. Similarly, when the input voltage is low, the PMOS is on; the NMOS is off, pulling v_{out} high.

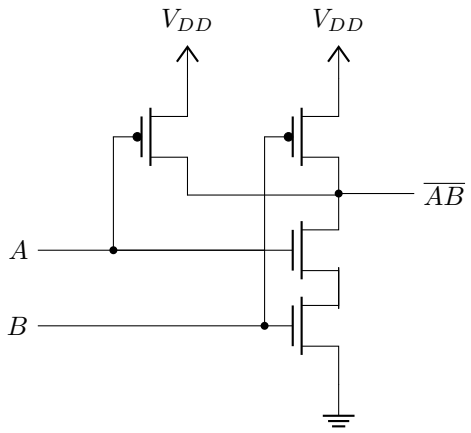


(a) CMOS inverter schematic

(b) Transfer characteristic

Figure 1: CMOS inverter and its voltage transfer curve

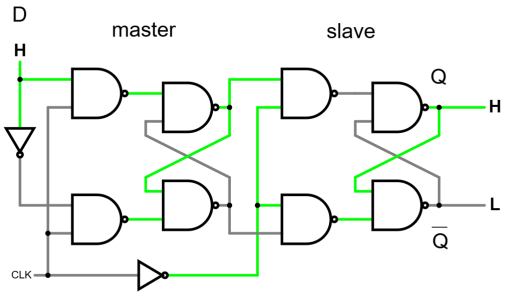
A NAND gate is similar. When *both* inputs are high, the output is pulled down; when *both* inputs are low, the output is pulled up.



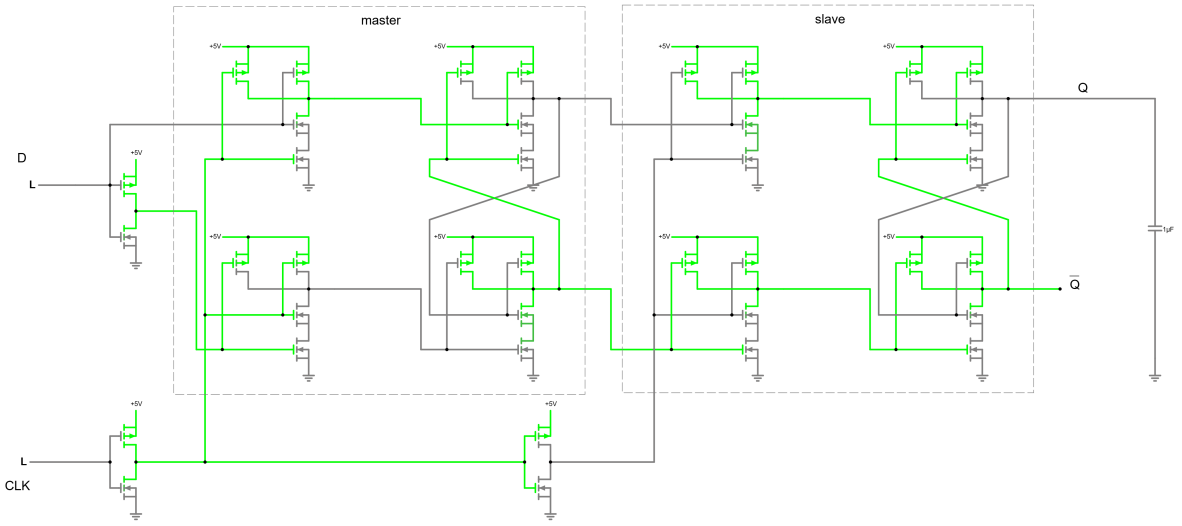
8.3 Integrated Circuits

8.3.1 Edge-triggered Flip-Flop

With digital inverters and NAND gates, we constructed a rising-edge-triggered master-slave D flip-flop, a fundamental state-holding element in digital design.

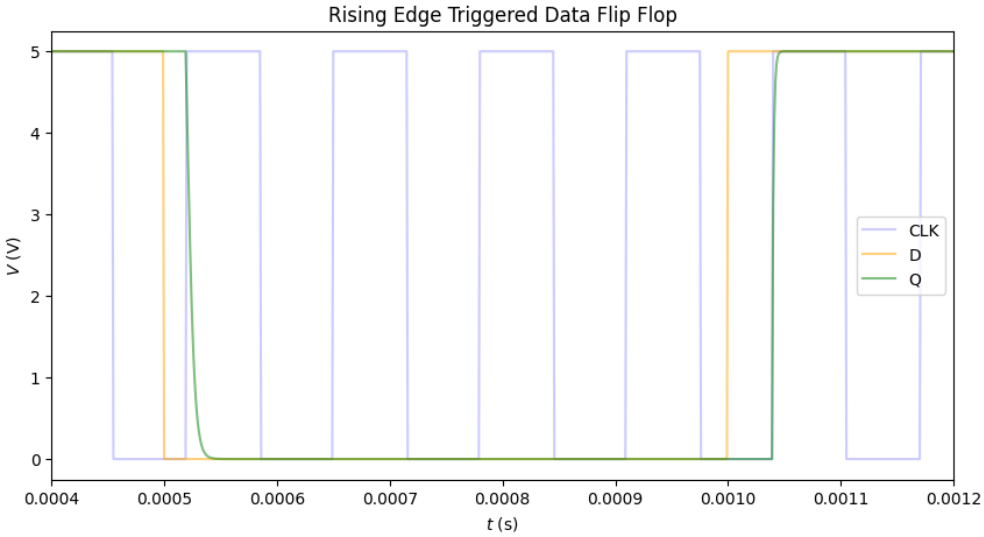


The master-slave configuration consisted of two stages: the master stage captures the input, and the slave stage releases it on the rising edge of the clock. Each stage uses four 2-input NAND gates, plus three inverters to buffer and steer the clock signal. The entire circuit has 38 transistors in total.



Here is the link to our circuit diagram: https://tinyurl.com/23ychjzp

Frankly, this circuit is no different than an ordinary gate-level test circuit. Our nonlinear matrix solver handles the inherent feedback loop robustly, converging to a solution that reflects both the new input and previous state. All of these are simulated in an analog manner.

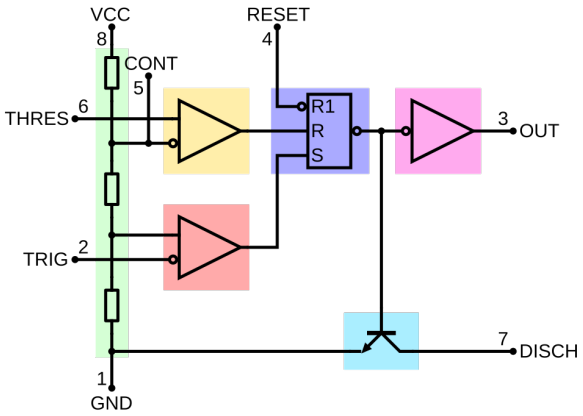


In the figure above, the flip-flop reliably updates its stored value to match `data` at the rising edge of `clk`.

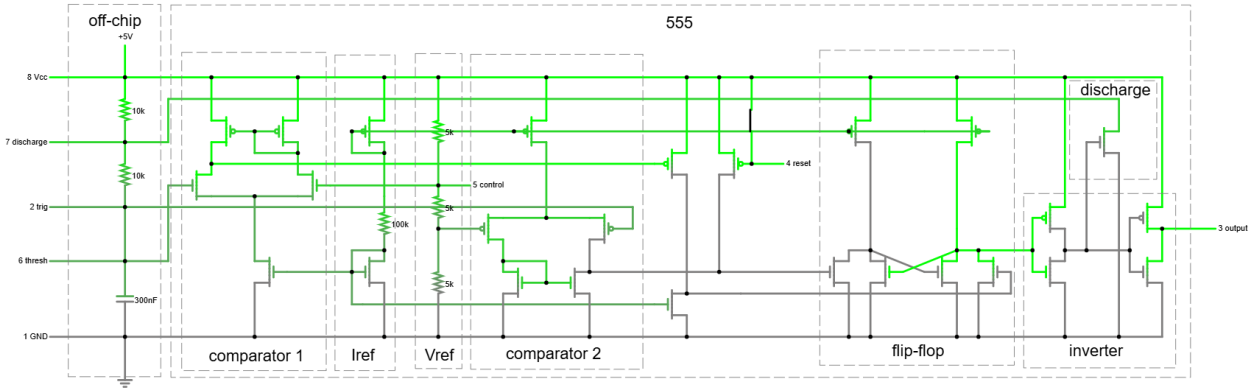
An optional capacitor can be appended to the output `Q` pin to simulate the gate capacitance of the next logic stage. Because the pull-up network employs two parallel PMOS transistors while the pull-down network uses two series NMOS transistors (all devices sharing the same area), the output rise and fall times are asymmetric. CircuitSim accurately simulates this behavior. In particular, the pull-down transition is noticeably slower, due to the higher equivalent resistance of the series NMOS path.

8.3.2 555 Hybrid Chip

Below is a block diagram of the 555 timer:

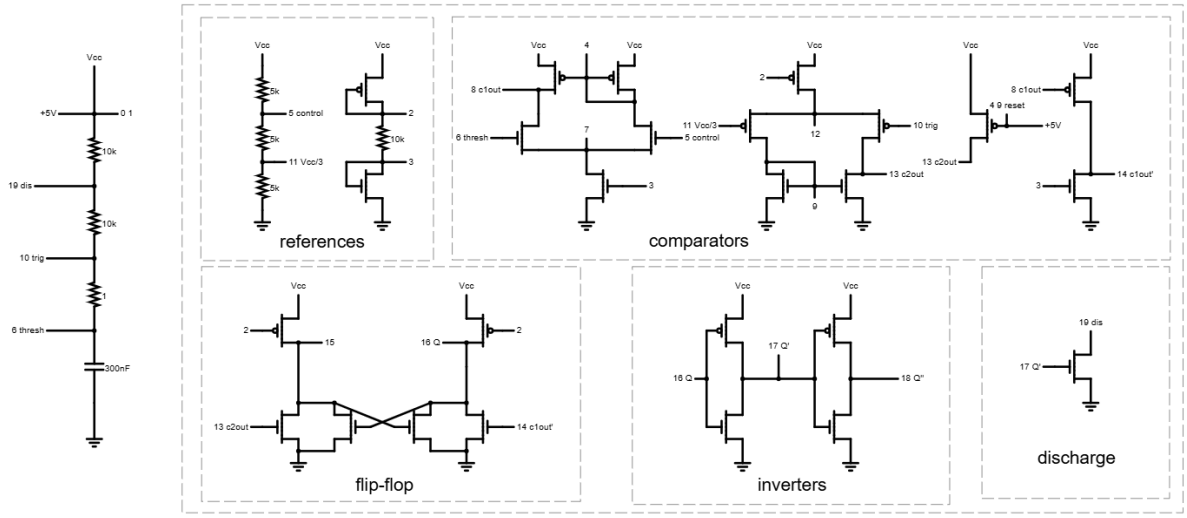


Here's its MOSFET implementation:



Link: https://tinyurl.com/27pevs7f

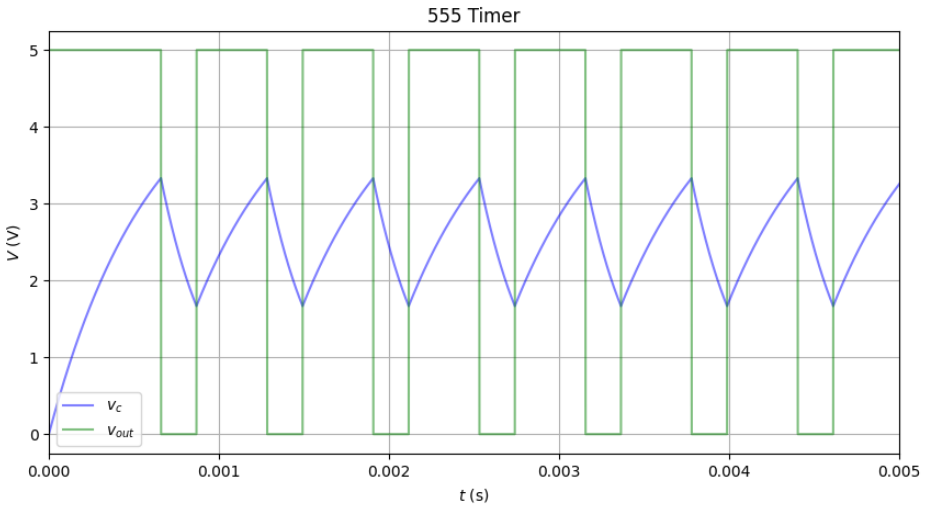
Looks insanely messy and complicated... Let's convert this circuit into a net list!



Link: https://tinyurl.com/2xvfqjgk

Now every block becomes clear. In the astable mode, for example,

- The references set up the reference voltage for the comparator, and the reference current (current mirror) for the MOS gates
- The comparators compare the capacitor voltage against the present values.
- The Flip-flop is an R-S latch that holds the previous state.
- If the capacitor voltage is higher than $^2/_3V_{CC}$, the digital logic turns on the discharge transistor, and the capacitor is discharged.
- When v_C discharges below $^1/_3V_{CC}$, the digital logic turns off the discharge transistor, and the capacitor is charging.
- With the help of the latch, the inverters output the charging/discharging state
- The charging and discharging period can be adjusted by the two external resistors, therefore changing the duty cycle and frequency of the oscillation.



Below are the simulation results with the astable configuration. $R_1 = R_2 = 10\,\text{k}\Omega$, and $C = 30\,\text{nF}$ Theoretically,

- Frequency $= \frac{1}{\ln(2)(R_1+2R_2)C} = 1.60\,\text{kHz}$
- Duty cycle $= \frac{R_2}{R_1+R_2} = 66.7\%$

The simulated frequency is 1.60 kHz and duty cycle 66.5 %

The simulation is very close to the theoretical values, despite there are a lot of non-idealities in the MOS models. It is rock solid!

9 Failure Modes

We've seen how CircuitSim works. Now it's time to see how it FAILS.

There are no perfect models. Under certain conditions, CircuitSim may return inaccurate, incorrect, invalid results, or even refuse to solve entirely.

Most of the problems can be attributed to the user input, circuit model, or solver logic.

9.1 Time Step Size

Large time steps degrade the accuracy of the discrete-time companion models for capacitors and inductors, potentially causing divergence or errors in the computed waveforms.

Typically, the companion resistance $R_{eq} = \frac{\Delta t}{C}$ and conductance $G_{eq} = \frac{\Delta t}{L}$ should be less than $\frac{1}{10}$ for an accurate simulation

9.2 Floating-point Limitations

Unreasonable initial guesses for nonlinear components—particularly diodes—can overflow the floating-point exponent range.

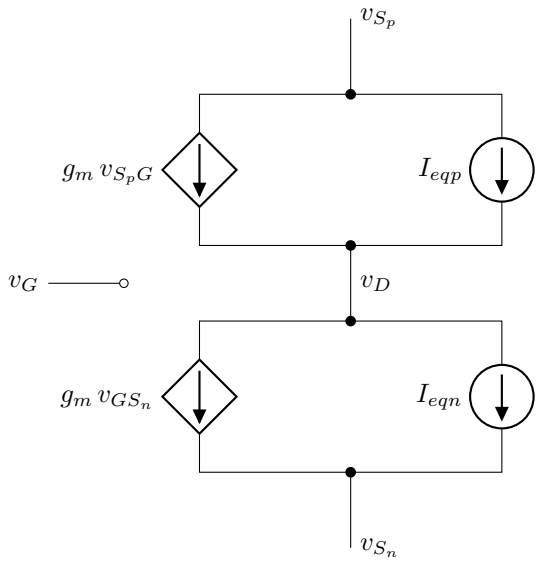
For example, an initial diode voltage of 5 V may internally escalate to around 1×10^{68} V! The output will be `NaN`.

Moreover, the Gaussian elimination solver incurs roundoff errors on the order of 10^{-6} relative to operand magnitudes; summing values of vastly different scales can magnify these errors.

9.3 Singular Matrices

Other than common cases that you may shoot yourself in your own foot, such as shorting a voltage source to ground, some hidden singularities in the models may happen if not configured correctly:

For instance, in a CMOS inverter model with $\lambda = 0$, both transistors in saturation give a drain conductance $g_d = 0$. The drain node v_D is decoupled, producing a singular matrix. Conceptually, we can assign any voltage to v_D, as long as both transistors stay in saturation mode.



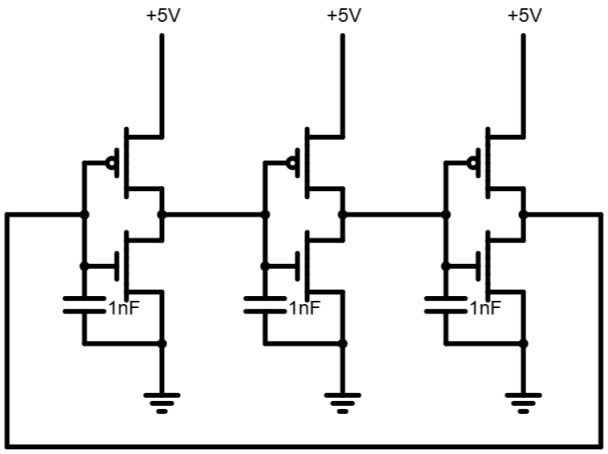
9.4 Convergence

Our MOSFET models are piecewise-defined and therefore lack continuous derivatives at region boundaries. Our model enforces a non-zero derivative, but discontinuities across the boundaries may prevent the solution from converging. The software limits the maximum number of iterations before throwing a warning and proceeding.

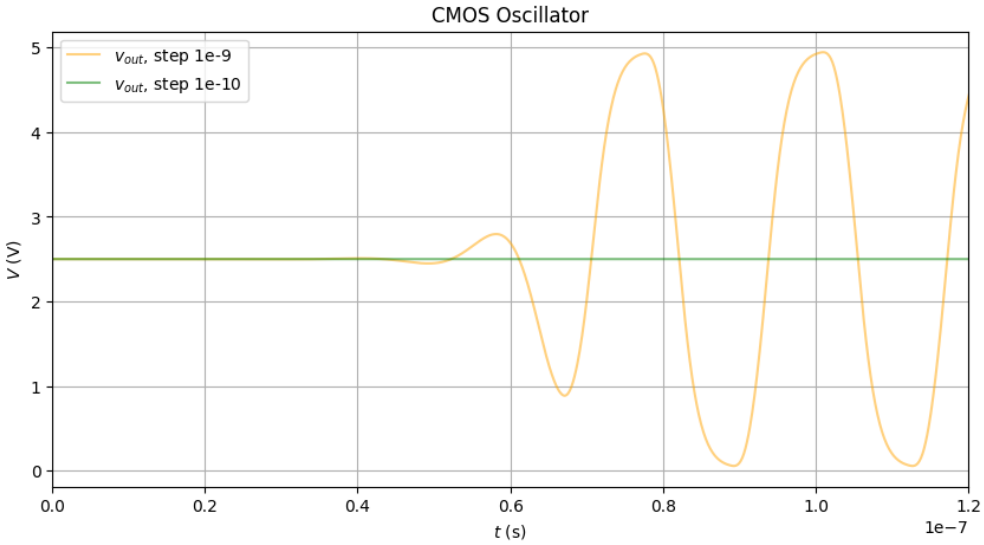
9.5 Unstable Equilibria

The Newton-Raphson loop will converge to a solution, but sometimes an undesired one.

Taking this CMOS oscillator for example. There are three solutions: 0 V, 5 V, and 2.5 V. Physically, it should oscillate between the 0 V, 5 V solutions. However, if all gate capacitor initial voltages are set to 2.5 V. When the timestep is 1 ns, all inverters stay at 2.5 V solution. This can't physically happen because any slight electron variations will tip the balance and force it to start oscillating. Similarly, at this accuracy, floating point approximation errors will tip the balance.



Using a finer time step of 0.1 ns, the oscillator gets perfectly stuck at the initial 2.5 V



For typical applications with random initial values, this typically wouldn't happen. However, if the initial condition is set perfectly to the unstable equilibrium, or if the circuit very accurately runs into such, the oscillator will get stuck on the unstable equilibrium forever.

10 Division of work

- Faustina and Andrew developed the SPICE parser
- Ming designed and implemented the entire circuit simulation logic
- Ming developed the circuits to test our simulation, Andrew and Faustina adapted it to SPICE netlists to test the parser
- Ming, Jary, and Case designed the Gaussian elimination accelerator
- Jary, Case, and Ming implemented the Gaussian elimination logic for the hardware
- Jary, Faustina, and Andrew worked on the integration between the software frontend and the FPGA Gaussian elimination backend

10.1 Lessons Learned

- First, start early. Starting ahead will give us a lot of time to design, test, debug, and refine this project.
- Clearly define and specify all signals, protocols, memory mappings, and I/O ports. We originally did not realize that memory read only takes two cycles, so we spent a lot of time fixing through the state machine to meet hardware requirements and protocols.

11 References

CircuitSim is an independent rewrite of SPICE. It is not affiliated with UC Berkeley or any other commercial versions of SPICE. Falstad is used for circuit diagrams and visualization, but all SPICE netlists and simulation results are obtained from CircuitSim

- SPICE algorithm overview: https://www.ecircuitcenter.com/SpiceTopics/Overview/Overview.htm
- Nodal analysis: https://www.ecircuitcenter.com/SpiceTopics/Nodal%20Analysis/Nodal%20Analysis.htm
- Modified nodal analysis: https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA2.html#Example_3
- Dependent sources: https://qucs.sourceforge.net/tech/node60.html
- Netwon-Raphson method: https://www.ecircuitcenter.com/SpiceTopics/Non-Linear%20Analysis/Non-Linear%20Analysis.htm
- Backward Euler method: https://electronics.stackexchange.com/questions/272012/companion-capacitor-model-in-circuit-simulation
- IEEE 754: https://www.h-schmidt.net/FloatConverter/IEEE754.html

12 Acknowledgments

Thank you to Professor Edwards and TA Peiran Wang for guiding us through this project, which would never have been completed without their guidance.

Also thank you to Newton, Euler, and Gauss for your amazing methods to turn this complicated problem into a matrix solver.

13 Code Segments

13.1 Circuit Simulator

%inputcode/Makefile

13.1.1 `circuit.h`

```
#ifndef CIRCUIT_H
#define CIRCUIT_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#ifdef __cplusplus
extern "C" {
#endif
```

```
#define MAT_SIZE     128    // maximum number of nodes
#define MAX_COMPS    100   // maximum number of components

#define MAX_NR_ITER 20
#define NR_TOL      1e-4f
#define DV_MAX      100.0f
#define VT_DEFAULT 0.025875f

#define PRINT_LIN   0
#define PRINT_NL    0
#define PRINT_MAT   0

#define FPGA_AXI_BASE 0xC0000000 // base address for RAM on FPGA
#define FPGA_AXI_SPAN 0x00020000 // 131072 bytes

// — Global nodal matrix and RHS —
extern float G[MAT_SIZE][MAT_SIZE];
extern float Ivec[MAT_SIZE];
extern float v[MAT_SIZE], v_prev[MAT_SIZE];
extern float t;

typedef struct {
    double v0;
    double v1;
    double td;
    double tr;
    double tf;
    double pw;
    double per;
} SpicePulseParams;

typedef struct {
    double vo;
    double va;
    double fo;
    double td;
    double a;
    double phase;
} SpiceSinParams;

typedef enum {
    SPICE_FUNC_PULSE,
    SPICE_FUNC_SIN,
    SPICE_FUNC_DC
} SpiceFuncType;

typedef struct {
    SpiceFuncType type;
    union {
        SpicePulseParams pulse;
        SpiceSinParams sin;
        double dc_value;
    } params;
} TransientSource;

// — Component data types —
// sources
```

```c
typedef struct {
    int n1, n2, ni;             // VSrc has an internal node
    TransientSource *src;
} VSrc;
typedef struct {
    int n1, n2;
    TransientSource *src;
} ISrc;
// dependent sources
typedef struct {
    int n1, n2;
    int np, nn, ni;
    float A;         // gain
} Vcvs;
typedef struct {
    int n1, n2;
    int np, nn;
    float A;
} Vccs;
typedef struct { int n1, n2, np, nn, ni; float A; } Ccvs;
typedef struct { int n1, n2, np, nn; float A; } Cccs;

// linear components
typedef struct {
    int n1, n2;
    float R;
} Res;

// time-varying components
typedef struct {
    int n1, n2;                 // nodes
    float C, dt;                // device constants
    float i_prev, v_prev;       // previous values
} Cap;
typedef struct {
    int n1, n2;
    float L, dt;
    float i_prev, v_prev;
} Ind;

// nonlinear components
typedef struct {
    int n1, n2;
    float Is, Vt;
    float v_prev;
} Diode;
typedef struct {
    int ng, nd, ns;
    float beta, Vt, lambda;
    float vgs_prev, vds_prev;
} Nmos;
typedef struct {
    int ng, nd, ns;
    float beta, Vt, lambda;
    float vsg_prev, vsd_prev;
} Pmos;

// generic component structure
```

```c
typedef enum { STA_T, LIN_T, NL_T } CompType;
typedef struct Component {
    CompType type;
    void (*stamp_lin)(struct Component*, float[][MAT_SIZE], float[]);
    void (*stamp_nl) (struct Component*, float[][MAT_SIZE], float[]);
    void (*update)   (struct Component*);
    union {
        VSrc   vsrc;
        ISrc   isrc;
        Vcvs   vcvs;
        Vccs   vccs;
        Ccvs   ccvs;
        Cccs   cccs;
        Res    res;
        Cap    cap;
        Ind    ind;
        Diode  dio;
        Nmos   nmos;
        Pmos   pmos;
    } u;
} Component;

// component registry
extern Component comps[MAX_COMPS];
extern int       ncomps; // number of components in the circuit
extern int       nnodes; // number of nodes in the circuit

// stamper and solver
void clear_system(void);
void clear_system_sta(void);
void stamp_static(void);
int solve_system(int n, int fd);
void update_all(float t);

void print_components(void);
void print_matrix(float m[][MAT_SIZE], float I[MAT_SIZE], int n);
void print_vector(float v[MAT_SIZE], int n);


void res_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);
void vsrc_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);
void isrc_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);
void vcvs_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);
void vccs_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);
void ccvs_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);
void cccs_stamp(Component *c, float Gm[][MAT_SIZE], float I[]);

void cap_stamp_lin(Component *c, float Gm[][MAT_SIZE], float I[]);
void cap_update(Component *c);

void ind_stamp_lin(Component *c, float Gm[][MAT_SIZE], float I[]);
void ind_update(Component *c);

void dio_stamp_nl(Component *c, float Gm[][MAT_SIZE], float I[]);
void dio_update(Component *c);

void nmos_stamp_nl(Component *c, float Gm[][MAT_SIZE], float I[]);
void nmos_update(Component *c);
```

```
void pmos_stamp_nl(Component *c, float Gm[][MAT_SIZE], float I[]);
void pmos_update(Component *c);


// registration & top-level
void add_vsrc(int n1, int n2, int ni, TransientSource *src);
void add_isrc(int n1, int n2, TransientSource *src);
void add_vcvs(int n1, int n2, int np, int nn, int ni, float A);
void add_vccs(int n1, int n2, int np, int nn, float A);
void add_ccvs(int n1, int n2, int np, int nn, int ni, float A);
void add_cccs(int n1, int n2, int np, int nn, float A);

void add_res (int n1, int n2, float R);
void add_cap (int n1, int n2, float C, float dt, float v0);
void add_ind (int n1, int n2, float L, float dt, float i0);
void add_diode(int n1, int n2, float Is, float Vt);
void add_nmos(int ng, int nd, int ns, float beta, float Vt, float lambda);
void add_pmos(int ng, int nd, int ns, float beta, float Vt, float lambda);

void add_not(int vin, int vout, int vDD);
void add_nand(int vina, int vinb, int vout, int vn, int vDD);

#ifdef __cplusplus
} // extern "C"
#endif

#endif // CIRCUIT_H
```

13.1.2 main.cpp

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include <sstream>
#include <memory>
#include "circuit.h"
#include <algorithm>
#include <functional>

// reference: https://web.stanford.edu/class/ee133/handouts/general/spice_ref.pdf

float G[MAT_SIZE][MAT_SIZE];
float Ivec[MAT_SIZE];
float v[MAT_SIZE], v_prev[MAT_SIZE];
float t = 0.0f;


std::vector<std::function<void(int)>> component_adders;
std::unordered_map<std::string, float> model_is;
std::unordered_map<std::string, std::tuple<float, float, float, std::string>> model_mosfet;

std::unordered_set<int> nodes;
```

```
float convertSPICENumToFloat(std::string input) {
    // dictionary corresponds between suffix and exponent
    std::unordered_map<char, int> suffixToExponent = {
        {'t', 12},
        {'g', 9},
        {'x', 6},
        {'k', 3},
        {'m', -3},
        {'u', -6},
        {'n', -9},
        {'p', -12},
        {'f', -15}
    };

    int exponent = 0;
    int num = 1;
    int splitPoint = -1;
    char curr;

    // find the start of the suffix
    for (int i = 0; i < input.size(); i++) {
        curr = std::tolower(input[i], std::locale());
        if (suffixToExponent.count(curr) > 0) {
            exponent = suffixToExponent[curr];
            splitPoint = i;
            break;
        } else if (curr == 'e') {
            // find the exponent
            if (i + 1 < input.size()) {
                exponent = std::stoi(input.substr(i + 1));
            }
            splitPoint = i;
            break;
        }
    }

    // read the number up to the suffix
    if (splitPoint != -1) {
        num =  std::stof(input.substr(0, splitPoint));
        return num * pow(10, exponent);
    } else {
        return std::stof(input);
    }
}

void parseLine(const std::string& line) {
    std::string new_line = (std::string)line;
    std::replace(new_line.begin(), new_line.end(), '(', ' ');
    std::replace(new_line.begin(), new_line.end(), ')', ' ');
    std::transform(new_line.begin(), new_line.end(), new_line.begin(), ::toupper);

    std::stringstream ss(new_line);
    std::string token;

    std::vector<std::string> tokens;
    while (ss >> token) {
        tokens.push_back(token);
```

```
}

std::unique_ptr<Component> componentPtr = nullptr;

if (tokens[0] == ".MODEL") {
    // model: .MODEL MODName D (IS= N= Rs= CJO= Tt= BV= IBV=)
    std::string model_name = tokens[1];
    if (tokens[2] == "D") { // Diode model
        for (size_t i = 2; i < tokens.size(); i++) {
            if (tokens[i].find("IS=") == 0) {
                model_is[model_name] = convertSPICENumToFloat(tokens[i].substr(3));
                break;
            }
        }
    } else if (tokens[2] == "NMOS" || tokens[2] == "PMOS") { // Nmos model
        std::get<0>(model_mosfet[model_name]) = 0.00002;
        std::get<1>(model_mosfet[model_name]) = 0;
        std::get<2>(model_mosfet[model_name]) = 0;

        std::get<3>(model_mosfet[model_name]) = tokens[2];

        for (size_t i = 2; i < tokens.size(); i++) {
            if (tokens[i].find("KP=") == 0 || tokens[i].find("KN=") == 0) {
                std::get<0>(model_mosfet[model_name]) = convertSPICENumToFloat(tokens[i].substr(3));
            }
            else if (tokens[i].find("VTO=") == 0) {
                std::get<1>(model_mosfet[model_name]) = convertSPICENumToFloat(tokens[i].substr(4));
            }
            else if (tokens[i].find("LAMBDA=") == 0) {
                std::get<2>(model_mosfet[model_name]) = convertSPICENumToFloat(tokens[i].substr(7));
            }
        }
    } else {
        std::cout << "Unknown model type: " << tokens[2] << std::endl;
    }

    return;
}

if (tokens[0][0] == 'V' || tokens[0][0] == 'I') {
    // voltage source: Vname N+ N- DCValue
    // first case is DC
    SpiceFuncType source_type;
    std::string new_line = (std::string)line;
    std::replace(new_line.begin(), new_line.end(), '(', ' ');
    std::replace(new_line.begin(), new_line.end(), ')', ' ');

    std::stringstream ss(new_line);
    std::string token;
    bool is_vsrc = tokens[0][0] == 'V';

    std::string source_name;
    if (!(ss >> source_name) || source_name.empty()) {
        throw std::runtime_error("Source name is empty or invalid.");
    }

    std::string node1, node2;
    if (!(ss >> node1 >> node2)) {
```

```
        throw std::runtime_error("Could not read source nodes for " + source_name);
    }

    if (!(ss >> token)) {
        throw std::runtime_error("Could not read transient function or DC value for " + source_name);
    }

    std::string upper_token = token;
    std::transform(upper_token.begin(), upper_token.end(), upper_token.begin(), ::toupper);

    // Check for explicit DC keyword
    if (upper_token == "DC") {
        source_type = SPICE_FUNC_DC;
        if (!(ss >> token)) {
            throw std::runtime_error("Expected DC value after 'DC' keyword for " + source_name);
        }
        // The next token *is* the DC value
        try {
            auto dc_source_ptr = std::make_unique<TransientSource>();
            dc_source_ptr->type = SPICE_FUNC_DC;
            dc_source_ptr->params.dc_value = convertSPICENumToFloat(token);
            int n1 = std::stoi(node1) - 1;
            int n2 = std::stoi(node2) - 1;
            nodes.insert(n1);
            nodes.insert(n2);

            if (!is_vsrc) {
                add_isrc(n1, n2, dc_source_ptr.release());
            } else {
                component_adders.push_back([n1, n2, src = dc_source_ptr.release()](int ni) mutable {
                    add_vsrc(n1, n2, ni, src);
                });
            }
            nnodes = std::max({nnodes, n1, n2});
        } catch (const std::exception& e) {
            throw std::runtime_error("Expected DC value after 'DC' keyword for " + source_name + ": " + e.what());
        }
    }
    // Check for transient function keywords
    else if (upper_token == "PULSE") source_type = SPICE_FUNC_PULSE;
    else if (upper_token == "SIN") source_type = SPICE_FUNC_SIN;
    else {
        // Not a keyword, assume it's an implicit DC value
        source_type = SPICE_FUNC_DC;
        try {
            float value = convertSPICENumToFloat(token);
            int n1 = std::stoi(node1) - 1;
            int n2 = std::stoi(node2) - 1;

            nodes.insert(n1);
            nodes.insert(n2);

            auto dc_source = std::make_unique<TransientSource>();
            dc_source->type = SPICE_FUNC_DC;
            dc_source->params.dc_value = value;
            if (!is_vsrc) {
                add_isrc(n1, n2, dc_source.release());
            } else {
```

```
                component_adders.push_back([n1, n2, src = dc_source.release()](int ni) mutable {
                    add_vsrc(n1, n2, ni, src);
                });
            }
            nnodes = std::max({nnodes, n1, n2});
        } catch (const std::exception& e) {
            throw std::runtime_error("Expected function keyword or DC value, got '" + token + "' for " + source_name +
            ↪  ": " + e.what());
        }
    }

    if (source_type != SPICE_FUNC_DC) {
        // Get the rest of the line containing parameters
        std::string params_str;
        std::getline(ss, params_str);

        size_t first = params_str.find_first_not_of(" \t(");
        size_t last = params_str.find_last_not_of(" \t)");
        if (first == std::string::npos || last == std::string::npos) {
            params_str = "";
        } else {
            params_str = params_str.substr(first, (last - first + 1));
        }


        std::stringstream param_ss(params_str);
        std::string param_token;
        std::vector<float> values;

        try {
            // Read all parameter tokens into a temporary vector
            while (param_ss >> param_token) {
                values.push_back(convertSPICENumToFloat(param_token));
            }
            auto source_ptr = std::make_unique<TransientSource>();
            // Assign parameters based on type and expected count
            switch (source_type) {
                case SPICE_FUNC_DC:
                    // handled above
                    break;
                case SPICE_FUNC_PULSE:
                    if (values.size() == 7) {
                        source_ptr->params.pulse = SpicePulseParams{values[0], values[1], values[2], values[3],
                        ↪  values[4], values[5], values[6]};
                        source_ptr->type = SPICE_FUNC_PULSE;
                        int n1 = std::stoi(node1) - 1;
                        int n2 = std::stoi(node2) - 1;

                        nodes.insert(n1);
                        nodes.insert(n2);

                        if (!is_vsrc) {
                            add_isrc(n1, n2, source_ptr.release());
                        } else {
                            component_adders.push_back([n1, n2,src = source_ptr.release()](int ni) mutable {
                                add_vsrc(n1, n2, ni, src);
                            });
                        }
```

```
                        nnodes = std::max({nnodes, n1, n2});
                    } else { throw std::runtime_error("Expected 7 parameters for Pulse, got " +
                    ↪  std::to_string(values.size())); }
                    break;
                case SPICE_FUNC_SIN:
                    if (values.size() == 6) {
                       source_ptr->params.sin = SpiceSinParams{values[0], values[1], values[2], values[3], values[4],
                        ↪  values[5]};
                    } else if (values.size() == 5) { // Handle optional td
                       source_ptr->params.sin = SpiceSinParams{values[0], values[1], values[2], values[3], values[4],
                        ↪  0.0};
                    } else if (values.size() == 4) { // Handle optional td, a
                       source_ptr->params.sin = SpiceSinParams{values[0], values[1], values[2], values[3], 0.0, 0.0};
                    } else if (values.size() == 3) {// Handle optional td, a, phase
                        source_ptr->params.sin = SpiceSinParams{values[0], values[1], values[2], 0.0, 0.0, 0.0};
                    }
                    else { throw std::runtime_error("Expected 4, 5, or 6 parameters for Sin, got " +
                    ↪  std::to_string(values.size())); }
                    source_ptr->type = SPICE_FUNC_SIN;
                    int n1 = std::stoi(node1) - 1;
                    int n2 = std::stoi(node2) - 1;

                    nodes.insert(n1);
                    nodes.insert(n2);

                    if (!is_vsrc) {
                        add_isrc(n1, n2, source_ptr.release());
                    } else {
                        component_adders.push_back([n1, n2, src = source_ptr.release()](int ni) mutable {
                            add_vsrc(n1, n2, ni, src);
                        });
                    }
                    nnodes = std::max({nnodes, n1, n2});
                    break;
            }
        } catch (const std::exception& e) {
            throw std::runtime_error("Error parsing parameters for " + source_name + " (" + upper_token + "): " +
            ↪  e.what());
        }
    }

} else if (tokens[0][0] == 'R') {
    // resistor: Rname N+ N- Value
    if (tokens.size() != 4) {
        std::cout << "Error: " << line << std::endl;
        return;
    }
    int n1 = std::stoi(tokens[1]) - 1;
    int n2 = std::stoi(tokens[2]) - 1;

    nodes.insert(n1);
    nodes.insert(n2);

    add_res(n1, n2, convertSPICENumToFloat(tokens[3]));
    nnodes = std::max({nnodes, n1, n2});
} else if (tokens[0][0] == 'C') {
    // capacitor: Cname N+ N- Value <IC=Initial Condition>
```

```
        if (tokens.size() != 5 && tokens.size() != 4) {
            std::cout << "Error: " << line << std::endl;
            return;
        }
        float v0 = 0.0f;

        if (tokens.size() == 5) {
            v0 = convertSPICENumToFloat(tokens[4]);
        }

        int n1 = std::stoi(tokens[1]) - 1;
        int n2 = std::stoi(tokens[2]) - 1;

        nodes.insert(n1);
        nodes.insert(n2);

        nnodes = std::max({nnodes, n1, n2});
        add_cap(n1, n2, convertSPICENumToFloat(tokens[3]), 5e-6f, v0);
    } else if (tokens[0][0] == 'L') {
        // inductor: Lname N+ N- Value <IC=Initial Condition>
        if (tokens.size() != 5 && tokens.size() != 4) {
            std::cout << "Error: " << line << std::endl;
            return;
        }

        float i0 = 0.0f;

        if (tokens.size() == 5) {
            i0 = convertSPICENumToFloat(tokens[4]);
        }

        int n1 = std::stoi(tokens[1]) - 1;
        int n2 = std::stoi(tokens[2]) - 1;
        nnodes = std::max({nnodes, n1, n2});

        add_ind(n1, n2, convertSPICENumToFloat(tokens[3]), 5e-6f, i0);
    } else if (tokens[0][0] == 'E' || tokens[0][0] == 'G') {
        // voltage controlled voltage source: Ename N+ N- NC+ NC- Gain
        // voltage controlled current source: Gname N+ N- NC+ NC- Gain
        if (tokens.size() != 6) {
            std::cout << "Dependent source has incorrect number of arguments: " << line << std::endl;
            return;
        }
        std::string source_name = tokens[0];
        try {
            int n1 = std::stoi(tokens[1]) - 1;
            int n2 = std::stoi(tokens[2]) - 1;
            int n3 = std::stoi(tokens[3]) - 1;
            int n4 = std::stoi(tokens[4]) - 1;
            float gain = convertSPICENumToFloat(tokens[5]);

            nodes.insert(n1);
            nodes.insert(n2);
            nodes.insert(n3);
            nodes.insert(n4);

            nnodes = std::max({nnodes, n1, n2, n3, n4});
            if (tokens[0][0] == 'G') {
```

```
            // Voltage controlled current source
            add_vccs(n1, n2, n3, n4, gain);
        } else {
            // Voltage controlled voltage source
            component_adders.push_back([n1, n2, n3, n4, gain](int ni) {
                add_vcvs(n1, n2, n3, n4, ni, gain);
            });
        }
    } catch (const std::exception& e) {
        throw std::runtime_error("Unexpected function keyword for dependent source, got '" + token + "' for " +
        ↪  source_name + ": " + e.what());
    }
}
else if (tokens[0][0] == 'D') {
    // diode: Dname N+ N- ModelName <IS=ISValue> <N=NValue> <VJ=VJValue> <XTI=XTIValue> <BV=BvValue> <IBV=IBVValue>
    if (tokens.size() != 4) {
        std::cout << "Incorrect number of arguments for Diode: " << line << std::endl;
        return;
    }
    std::string diode_name = tokens[0];
    try {
        int n1 = std::stoi(tokens[1]) - 1;
        int n2 = std::stoi(tokens[2]) - 1;
        std::string model_name = tokens[3];
        float is = 1e-15f;
        if (model_is.find(model_name) != model_is.end()) {
            is = model_is[model_name];
        }

        nodes.insert(n1);
        nodes.insert(n2);
        nnodes = std::max({nnodes, n1, n2});

        add_diode(n1, n2, is, VT_DEFAULT);
    } catch (const std::exception& e) {
        throw std::runtime_error("Unexpected arguments for Diode " + diode_name + ": " + e.what());
    }
}
else if (tokens[0][0] == 'M') {
    // MOSFET: Mname D G S B ModelName <L=LValue> <W=WValue> <AD=ADValue> <AS=ASValue> <PS=PSValue> <PD=PDValue>
    if (tokens.size() < 6) {
        std::cout << "Insufficient arguments for MOSFET: " << line << std::endl;
        return;
    }
    std::string mosfet_name = tokens[0];
    try {
        int n1 = std::stoi(tokens[1]) - 1; // ND
        int n2 = std::stoi(tokens[2]) - 1; // NG
        int n3 = std::stoi(tokens[3]) - 1; // NS
        std::string model_name = tokens[4];
        float l;
        float w;
        for (size_t i = 5; i < tokens.size(); i++) {
            if (tokens[i].find("L=") == 0) {
                l = convertSPICENumToFloat(tokens[i].substr(2));
            } else if (tokens[i].find("W=") == 0) {
                w = convertSPICENumToFloat(tokens[i].substr(2));
            }
```

```
            }

            if (l == 0 || w == 0) {
                std::cout << "Error: L and W must be specified for MOSFET " << mosfet_name << std::endl;
                return;
            }

            float kp = std::get<0>(model_mosfet[model_name]);
            float vto = std::get<1>(model_mosfet[model_name]);
            if (vto < 0) {
                vto *= -1;
            }
            float lambda = std::get<2>(model_mosfet[model_name]);

            nodes.insert(n1);
            nodes.insert(n2);
            nodes.insert(n3);
            nnodes = std::max({nnodes, n1, n2, n3});

            if (std::get<3>(model_mosfet[model_name]) == "NMOS") {
                add_nmos(n2, n1, n3, kp * w / l, vto, lambda);
            } else if (std::get<3>(model_mosfet[model_name]) == "PMOS") {
                add_pmos(n2, n1, n3, kp * w / l, vto, lambda);
            } else {
                std::cout << "Unknown MOSFET model: " << model_name << std::endl;
                return;
            }
        } catch (const std::exception& e) {
            throw std::runtime_error("Unexpected arguments for MOSFET " + mosfet_name + ": " + e.what());
        }
    } else {
        // something went wrong!
        std::cout << "Not implemented yet or error: " << line << std::endl;
        return;
    }

    //TODO: analysis information

}

int main(int argc, char **argv) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <input_file> <output_file>" << std::endl;
        return 1;
    }

    std::string input_file = argv[1];
    std::ifstream file(input_file);
    if (!file.is_open()) {
        std::cerr << "File cannot be opened: " << input_file << std::endl;
        return 1;
    }

    std::string output_file = argv[2];
    std::ofstream output(output_file);
    if (!output.is_open()) {
        std::cerr << "File cannot be opened: " << output_file << std::endl;
        return 1;
```

```
}

float time_step = -1;
int nsteps = -1;
float start_time = 0;

std::string line;
// skip first line of input file
std::getline(file, line);

while (std::getline(file, line)) {
    if (line.empty()) {
        continue;
    } else if (line[0] == '*') {
        continue;
    }

    std::stringstream ss(line);
    std::string token;

    std::vector<std::string> tokens;
    while (ss >> token) {
        tokens.push_back(token);
    }

    if (tokens[0] == ".END") {
        break;
    }

    if (tokens[0] == ".TRAN") {
        if (time_step != -1) {
            std::cerr << "Error: Conflicting .TRAN statements are specified." << std::endl;
            return 1;
        }

        if (tokens.size() < 3 || tokens.size() > 4) {
            std::cerr << "Error: .TRAN has incorrect number of arguments." << std::endl;
            return 1;
        }

        time_step = convertSPICENumToFloat(tokens[1]);
        float stop_time = convertSPICENumToFloat(tokens[2]);

        if (tokens.size() == 3) {
            nsteps = static_cast<int>(stop_time / time_step);
        } else if (tokens.size() == 4) {
            start_time = convertSPICENumToFloat(tokens[3]);
        }

        continue;
    }

    // otherwise we have a valid line so we split the line by whitespace
    parseLine(line);
}

if (time_step < 0 || nsteps < 0) {
    std::cerr << "Error: .TRAN not found or invalid." << std::endl;
```

```
    return 1;
}

nnodes++;

if (nnodes + 1 != nodes.size()) {
    std::cerr << "Error: Number of nodes does not match the number of unique nodes found." << std::endl;
    return 1;
}

for (const auto& add_component : component_adders) {
    add_component(nnodes++);
}

std::vector<int> printed_nodes;
for (int i = 3; i < argc; i++) {
    try {
        int node = std::stoi(argv[i]) - 1;
        if (node >= 0 && node < nnodes) {
            printed_nodes.push_back(node);
        } else {
            std::cerr << "Invalid node number: " << argv[i] << std::endl;
        }
    } catch (const std::invalid_argument&) {
        std::cerr << "Invalid node number: " << argv[i] << std::endl;
    }
}
if (printed_nodes.empty()) {
    for (int i = 0; i < nnodes; i++) {
        printed_nodes.push_back(i);
    }
}

stamp_static();


output << "time";
for (int i : printed_nodes) {
    output << "," << "node" << i + 1;
}
output << "\n";

for (int n = 0; n < nsteps; n++) {
    t = n * time_step;
    update_all(t);

    std::stringstream s;

    if (n * time_step < start_time) {
        continue;
    }

    s << t;

    for (int i : printed_nodes) {
        s << "," << v[i];
    }
    s << "\n";
```

```
        output << s.str();
    }

    file.close();
    return 0;

}
```

13.1.3 `newton.c`

```
// newton.c
#include <stddef.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include "circuit.h"


float G_sta[MAT_SIZE][MAT_SIZE];
float I_sta[MAT_SIZE];
float G_lin[MAT_SIZE][MAT_SIZE];
float I_lin[MAT_SIZE];

int open_fpga(void) {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd == -1) {
        perror("open");
        exit(1);
    }
    return fd;
}

void* mmap_memory(int fd) {
    void* virtual_base;

        virtual_base = mmap(NULL, FPGA_AXI_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED, fd, FPGA_AXI_BASE);

        if (virtual_base == MAP_FAILED) {
                perror("mmap");
                exit(1);
        }
    return virtual_base;
}

int my_memcpy(volatile float* source, volatile float* dest, size_t size) {
        size_t num_floats = size / sizeof(float);
        for (size_t i = 0; i < num_floats; i++) {
                dest[i] = source[i];
        }
        return 0;
}

// main loop to do time and nonlinear simulation

/* zero G and Ivec */
```

```
void clear_system(void) {
    for (int i = 0; i < MAT_SIZE; i++) {
        Ivec[i] = 0.0f;
        for (int j = 0; j < MAT_SIZE; j++)
            G[i][j] = 0.0f;
    }
}

/* clear everything EXCEPT static components */
void clear_system_sta(void) {
    memcpy(G_lin, G_sta, sizeof G_sta);
    memcpy(G, G_sta, sizeof G_sta);
    memcpy(I_lin, I_sta, sizeof I_sta);
    memcpy(Ivec, I_sta, sizeof I_sta);
}

/* stamp all static components */
void stamp_static(void) {
    for (int i = 0; i < ncomps; i++) {
        if (comps[i].type == STA_T && comps[i].stamp_lin) {
            comps[i].stamp_lin(&comps[i], G_sta, I_sta);
        }
    }
}

/* Main loop: linear and nonlinear */
void update_all(float t) {
    (void)t;
    // 0. retrieve static component matrix
    clear_system_sta();

    // 1. stamp all linear components
    for (int i = 0; i < ncomps; i++) {
        if (comps[i].type == LIN_T && comps[i].stamp_lin) {
            comps[i].stamp_lin(&comps[i], G, Ivec);
        }
    }

    // 2. Snapshot G, I with only linear components, and previous v
    float prev_v[MAT_SIZE];
    memcpy(G_lin, G, sizeof G);
    memcpy(I_lin, Ivec, sizeof Ivec);
    memcpy(v_prev, v, sizeof v);

    int fd = open_fpga();
    void* virtual_base = mmap_memory(fd);
    volatile float* fpga_base = (volatile float*)virtual_base;

    int ret = 0;

    // 3. Newton-Raphson loop for all nonlinear components
    for (int iter = 0; iter < MAX_NR_ITER; iter++) {
        // 3.1 restore the linear stamps
        memcpy(G, G_lin, sizeof G);
        memcpy(Ivec, I_lin, sizeof Ivec);

        // 3.2 stamp all nonlinear components
        for (int i = 0; i < ncomps; i++) {
```

```
        if (comps[i].type == NL_T && comps[i].stamp_nl)
            comps[i].stamp_nl(&comps[i], G, Ivec);
    }

    // 3.3 solve the system
    if (PRINT_MAT)
        print_matrix(G, Ivec, nnodes);

    /* solve_system() solves in v, for Gv = I
    G, v, Ivec are all declared globally*/
    // 3.3.1 write G to FPGA
    my_memcpy(fpga_base, (volatile float*)G, sizeof(G));

    // 3.3.2 write Ivec to FPGA
    my_memcpy(fpga_base + sizeof(G) / sizeof(float), Ivec, sizeof(Ivec));

    ret = solve_system(nnodes);
    if (ret != 0) {
        fprintf(stderr, "Error: solve_system failed with code %d\n", ret);
    }

    // 3.3.3 read v from FPGA
    my_memcpy(v, fpga_base + sizeof(G) / sizeof(float) + sizeof(Ivec) / sizeof(float), sizeof(v));

    if (PRINT_MAT)
        print_vector(v, nnodes);

    // 3.4 compute dv, check for convergence
    float maxdv = 0.0f;
    for (int n = 0; n < nnodes; n++) {
        float dv = fabsf(v[n] - prev_v[n]);
        if (dv > maxdv) maxdv = dv;
    }
    if (PRINT_MAT)
        printf("iter %d: max_res=%.6g\n", iter, maxdv);

    if (maxdv < NR_TOL) {   // converged!
        break;
    }

    // 3.5 Update every device's internal operating point
    for (int i = 0; i < ncomps; i++) {
        if (comps[i].type == NL_T && comps[i].update) {
            comps[i].update(&comps[i]); // update diode only
        }
    }

    // 3.6 prepare for next iteration
    memcpy(prev_v, v, sizeof prev_v);
}

if (munmap(virtual_base, FPGA_AXI_SPAN) == -1) {
            perror("munmap");
            close(fd);
            exit(1);
    }

    close(fd);
```

```
    // 4) final update so device states match the converged voltages
    for (int i = 0; i < ncomps; i++) {
        if (comps[i].type == LIN_T && comps[i].update) {
            comps[i].update(&comps[i]);
        }
    }
}

/* prints the type of every component*/
void print_components(void) {
    for (int i = 0; i < ncomps; i++) {
        printf("Component %d: ", i);
        switch (comps[i].type) {
            case LIN_T: printf("LIN_T "); break;
            case STA_T: printf("STA_T "); break;
            case NL_T:  printf("NL_T ");  break;
            default:    printf("??? ");   break;
        }
        printf("\n");
    }
}
```

13.1.4 `solver.c`

```
#include "circuit.h"

#include <stdio.h>
#include <stdlib.h>
#include "gaussian.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define POLL_INTERVAL_US 10000  // 10 ms



/* Read and print the input sent to the Gaussian */
void print_in(int gaussian_fd) {
  gaussian_arg_t vla;
  printf("%d\n", gaussian_fd);
  if (ioctl(gaussian_fd, GAUSSIAN_READ, &vla)) {
      perror("ioctl(GAUSCXXSIAN_READ) failed");
      return;
  }
    printf("INPUT n=%u  GO=%u  RST=%u\n",
           vla.input.n,
           vla.input.g,
           vla.input.r);
}
```

```c
/* Read and print the output obtained from Gaussian */
gaussian_out_t get_out(int gaussian_fd) {
    gaussian_arg_t vla;
    if (ioctl(gaussian_fd, GAUSSIAN_READ, &vla) < 0) {
        perror("ioctl(GAUSSIAN_READ) failed");
        exit(1);
    }
    printf("OUTPUT DONE=%u  E=%u  S=%u\n\n",
            vla.output.d,
            vla.output.e,
            vla.output.s);

    return vla.output;
}

/* Set martrix size and GO/RES flags */
void set_in(const gaussian_in_t *c, int gaussian_fd)
{
    gaussian_arg_t vla;
    memset(&vla, 0, sizeof(vla));
    vla.input = *c;
    if (ioctl(gaussian_fd, GAUSSIAN_WRITE, &vla)) {
        perror("ioctl(GAUSSIAN_WRITE) failed");
        return;
    }
}

/* This will become solve_system(n), the function called by newton.c */
int solve_system(int n, int gaussian_fd) {
    if (n <= 0 || n > 255) {
        fprintf(stderr, "Matrix size must be 1-255\n");
        return -1;
    }

    // kick off the hardware



    /* Kick off the computation: set size=n, GO=1, RST=0 */
    gaussian_in_t in = {
        .n = (uint8_t)n,
        .g = 1,    // GO
        .r = 0     // RESET
    };
    set_in(&in, gaussian_fd);

    print_in(gaussian_fd);

    /* Clear GO so we can detect future runs */
    in.g = 0;
    /* Warning: GO may hold high for multiple cycles. Make sure HW can handle that */
    set_in(&in, gaussian_fd);

    print_in(gaussian_fd);

    /* poll until DONE = 1 */
    printf("Waiting for DONE flag...\n");
    while (1) {
```

```
        gaussian_out_t out = get_out(gaussian_fd);
        printf("  DONE=%u  E=%u  S=%u\r",
               out.d, out.e, out.s);
        fflush(stdout);
        if (out.d)
            break;
        usleep(POLL_INTERVAL_US);
    }

    printf("Gaussian Userspace program terminating\n");

    return 0;
}



void print_matrix(float m[][MAT_SIZE], float I[MAT_SIZE], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%8.4g ", m[i][j]);
        printf(" | %8.4g\n", I[i]);
    }
    printf("\n");
}

void print_vector(float v[MAT_SIZE], int n) {
    for (int i = 0; i < n; i++)
        printf("%8.4g ", v[i]);
    printf("\n");
}
```

13.1.5 Circuit Models

13.1.6 `add_component.c`

```
#include "circuit.h"

// API to register components in the circuit

Component comps[MAX_COMPS];
int       ncomps = 0;
int       nnodes = 0;

// — Component registration —
void add_vsrc(int n1,int n2, int ni, TransientSource *src) {
    // ni for an internal node for an extra row in the equation.
    Component *c = &comps[ncomps++];
    c->type      = LIN_T;
    c->stamp_lin = vsrc_stamp;
    c->stamp_nl  = NULL;
    c->update    = NULL;
    c->u.vsrc     = (VSrc){n1,n2, ni, src};
}
void add_isrc(int n1,int n2, TransientSource *src) {
    Component *c = &comps[ncomps++];
    c->type      = LIN_T;
```

```
    c->stamp_lin = isrc_stamp;
    c->stamp_nl  = NULL;
    c->update    = NULL;
    c->u.isrc     = (ISrc){n1,n2, src};
}
void add_vcvs(int n1, int n2, int np, int nn, int ni, float A) {
    Component *c = &comps[ncomps++];
    c->type      = STA_T;
    c->stamp_lin = vcvs_stamp;
    c->stamp_nl  = NULL;
    c->update    = NULL;
    c->u.vcvs    = (Vcvs){n1,n2, np, nn, ni, A};
}
void add_vccs(int n1, int n2, int np, int nn, float A) {
    Component *c = &comps[ncomps++];
    c->type      = STA_T;
    c->stamp_lin = vccs_stamp;
    c->stamp_nl  = NULL;
    c->update    = NULL;
    c->u.vccs     = (Vccs){n1,n2, np, nn, A};
}
void add_res(int n1, int n2, float R) {
    Component *c = &comps[ncomps++];
    c->type      = STA_T;
    c->stamp_lin = res_stamp;
    c->stamp_nl  = NULL;
    c->update    = NULL;
    c->u.res     = (Res){n1,n2,R};
}

void add_cap(int n1,int n2,float C,float dt,float v0) {
    Component *c = &comps[ncomps++];
    c->type      = LIN_T;
    c->stamp_lin = cap_stamp_lin;
    c->stamp_nl  = NULL;
    c->update    = cap_update;
    c->u.cap     = (Cap){n1,n2,C,dt,v0,0};   // initial conditions are 0
}

void add_ind(int n1,int n2,float L,float dt, float i0) {
    Component *c = &comps[ncomps++];
    c->type      = LIN_T;
    c->stamp_lin = ind_stamp_lin;
    c->stamp_nl  = NULL;
    c->update    = ind_update;
    c->u.ind     = (Ind){n1,n2,L,dt,0,i0};
}
void add_diode(int n1, int n2, float Is, float Vt) {
    Component *c = &comps[ncomps++];
    c->type      = NL_T;
    c->stamp_lin = NULL;
    c->stamp_nl  = dio_stamp_nl;
    c->update    = dio_update;
    c->u.dio     = (Diode){n1,n2,Is,Vt,1.0f};   // initial guess: 1V
}
void add_nmos(int ng, int nd, int ns, float beta, float Vt, float lambda) {
    // Linear: beta * [(Vgs-Vt) * Vds - Vds^2 / 2]
    // Saturation: 0.5 beta * (Vgs-Vt)^2
```

```
    Component *c = &comps[ncomps++];
    c->type      = NL_T;
    c->stamp_lin = NULL;
    c->stamp_nl  = nmos_stamp_nl;
    c->update    = nmos_update;
    c->u.nmos    = (Nmos){ng,nd,ns,beta,Vt, lambda, 2.5f, 2.5f};
}
void add_pmos(int ng, int nd, int ns, float beta, float Vt, float lambda) {
    // everything's negative
    Component *c = &comps[ncomps++];
    c->type      = NL_T;
    c->stamp_lin = NULL;
    c->stamp_nl  = pmos_stamp_nl;
    c->update    = pmos_update;
    c->u.pmos    = (Pmos){ng,nd,ns,beta,Vt, lambda, 2.5f, 2.5f};
}

void add_not(int vin, int vout, int vDD) {
    add_nmos(vin, vout, -1, 0.02f, 1.5f, 0.01f);
    add_pmos(vin, vout, vDD, 0.02f, 1.5f, 0.01f);
}

void add_nand(int vina, int vinb, int vout, int vn, int vDD) {
    add_nmos(vina, vout, vn, 0.02f, 1.5f, 0.01f);
    add_nmos(vinb, vn, -1, 0.02f, 1.5f, 0.01f);
    add_pmos(vina, vout, vDD, 0.02f, 1.5f, 0.01f);
    add_pmos(vinb, vout, vDD, 0.02f, 1.5f, 0.01f);
}
```

13.1.7 `linear.c`

```
// linear.c
#include "circuit.h"

// all linear components

float pulse(float t, SpicePulseParams params) {
    printf("pulse: t=%.6g  v0=%.6g v1=%.6g td=%.6g  tr=%.6g  tf=%.6g  pw=%.6g  per=%.6g\n", t, params.v0, params.v1,
    ↪  params.td, params.tr, params.tf, params.pw, params.per);
    float delay_adj_t = t - params.td;

    if (delay_adj_t < 0) return 0.0f;

    float period = params.per;

    float location_in_period = fmod(delay_adj_t, period);

    printf("pulse: location_in_period=%.6g\n", location_in_period);

    // check if on rising edge
    if (location_in_period < params.tr) {
        return params.v0 + (params.v1 - params.v0) * (location_in_period / params.tr);
    }

    // check if on high level
    if (location_in_period < params.tr + params.pw) {
```

```
        return params.v1;
    }

    // check if on falling edge
    if (location_in_period < params.tr + params.pw + params.tf) {
        return params.v1 - (params.v1 - params.v0) * ((location_in_period - params.tr - params.pw) / params.tf);
    }

    // check if on low level
    return params.v0;
}

// — Resistor (and Norton-equiv Vsrc) stamp with -1 = ground —
void res_stamp(Component *c, float Gm[][MAT_SIZE], float I[]) {
    (void)I;
    Res *r = &c->u.res;
    float G = 1.0f / r->R;
    int   n1 = r->n1, n2 = r->n2;

    if (n1 != -1) Gm[n1][n1] += G;
    if (n2 != -1) Gm[n2][n2] += G;
    if (n1 != -1 && n2 != -1) {
        Gm[n1][n2] -= G;
        Gm[n2][n1] -= G;
    }
}

/* add an ideal voltage source; stamps a "virtual" node (ni) to the circuit.*/
void vsrc_stamp(Component *c, float Gm[][MAT_SIZE], float I[]) {
    VSrc *s = &c->u.vsrc;
    TransientSource *src = s->src;
    float E;
    if (src->type == SPICE_FUNC_SIN) {
        SpiceSinParams *params = &src->params.sin;
        E = params->vo + params->va * exp(-params->a * (t - params->td)) * sinf(2.0f * M_PI * params->fo * (t -
        ↪  params->td) + params->phase / 360.0f);
    } else if (src->type == SPICE_FUNC_PULSE) {
        // SpicePulseParams *params = &src->params.pulse;
        //TODO: implement pulse
        E = pulse(t, src->params.pulse);
        printf("vsrc_stamp: E=%.6g\n", E);
    } else if (src->type == SPICE_FUNC_DC) {
        E = src->params.dc_value;
    } else {
        printf("vsrc_stamp: unknown source type\n");
        E = 0.0f;
        // something went wrong if we're here
    }

    //printf("vsrc_stamp: E=%.6g\n", E);
    int n1 = s->n1, n2 = s->n2, ni = s->ni;

    if (n1 != -1) {
        Gm[n1][ni] += 1.0f;
        Gm[ni][n1] += 1.0f;  // symmetry for the voltage equation
    }
    if (n2 != -1) {
        Gm[n2][ni] -= 1.0f;1
```

```
        Gm[ni][n2] -= 1.0f;
    }

    // Voltage equation: v(n1) - v(n2) = E
    // stamped as  +1·v(n1) -1*v(n2) +0*I_vs  = E
    I[ni] += E;
}

void isrc_stamp(Component *c, float Gm[][MAT_SIZE], float I[]) {
    (void)Gm; (void)I;
    ISrc *s = &c->u.isrc;
    TransientSource *src = s->src;
    float Ieq;
    if (src->type == SPICE_FUNC_SIN) {
        SpiceSinParams *params = &src->params.sin;
        Ieq = params->vo + params->va * exp(-params->a * (t - params->td)) * sinf(2.0f * M_PI * params->fo * (t -
        ↪  params->td) + params->phase / 360.0f);
    } else if (src->type == SPICE_FUNC_PULSE) {
        Ieq = pulse(t, src->params.pulse);
        //TODO: implement pulse
    } else if (src->type == SPICE_FUNC_DC) {
        Ieq = src->params.dc_value;
    } else {
        printf("vsrc_stamp: unknown source type\n");
        Ieq = 0.0f;
        // something went wrong if we're here
    }

    int   n1 = s->n1, n2 = s->n2;

    if (n1 != -1) I[n1] += Ieq;
    if (n2 != -1) I[n2] -= Ieq;
}

/*
 * Voltage-Controlled Voltage Source (VCVS):
 *    v(n1)-v(n2) = mu * (v(np)-v(nn))
 * adds one extra current-unknown at index ni.
 */
void vcvs_stamp(Component *c, float Gm[][MAT_SIZE], float I[]) {
    (void)I;
    Vcvs *s = &c->u.vcvs;
    int n1   = s->n1,
        n2   = s->n2,
        np   = s->np,
        nn   = s->nn,
        ni   = s->ni;
    float mu = s->A;

    if (n1) Gm[n1][ni] +=  1.0f;
    if (n2) Gm[n2][ni] += -1.0f;

    if (n1) Gm[ni][n1] +=  1.0f;
    if (n2) Gm[ni][n2] += -1.0f;
    if (np) Gm[ni][np] += -mu;
    if (nn) Gm[ni][nn] +=  mu;

    //no direct RHS contribution for a pure dependent source
```

```
}

/*
 * Voltage-Controlled Current Source (VCCS):
 *    i = gm * (v(np)-v(nn))
 * injects from n2 to n1.
 */
void vccs_stamp(Component *c, float Gm[][MAT_SIZE], float I[]) {
    (void)I;
    Vccs *s = &c->u.vccs;
    int   n1 = s->n1,
          n2 = s->n2,
          np = s->np,
          nn = s->nn;
    float gm = s->A;

    if (n1 && np) Gm[n1][np] += -gm;
    if (n1 && nn) Gm[n1][nn] += gm;
    if (n2 && np) Gm[n2][np] += gm;
    if (n2 && nn) Gm[n2][nn] += -gm;
}


void cap_stamp_lin(Component *c, float Gm[][MAT_SIZE], float I[]) {
    Cap *p = &c->u.cap;
    float Gc  = p->C / p->dt;
    float i_prev = p->i_prev;

    int n1 = p->n1, n2 = p->n2;
    if (n1 != -1) Gm[n1][n1] += Gc;
    if (n2 != -1) Gm[n2][n2] += Gc;
    if (n1 != -1 && n2 != -1) {
        Gm[n1][n2] -= Gc;
        Gm[n2][n1] -= Gc;
    }
    if (n1 != -1) I[n1] -= i_prev; // Ieq is defined leaving n1, entering n2
    if (n2 != -1) I[n2] += i_prev;
}

void cap_update(Component *c) {
    Cap *p = &c->u.cap;
    float vc = (p->n1!=-1? v[p->n1]:0) - (p->n2!=-1? v[p->n2]:0);
    float Gc = p->C / p->dt;
    float Ieq = -Gc * vc;

    p->i_prev = Ieq;
    p->v_prev = vc;

    if (PRINT_LIN)
        printf("cap_update: i_prev=%.6g  v_prev=%.6g Ieq=%.6g\n", p->i_prev, p->v_prev, Ieq);
}


void ind_stamp_lin(Component *c, float Gm[][MAT_SIZE], float I[]) {
    Ind *p = &c->u.ind;
    float Gl  = p->dt / p->L;
    float i_prev = p->i_prev;
```

```
    int n1 = p->n1, n2 = p->n2;
    if (n1 != -1) Gm[n1][n1] += Gl;
    if (n2 != -1) Gm[n2][n2] += Gl;
    if (n1 != -1 && n2 != -1) {
        Gm[n1][n2] -= Gl;
        Gm[n2][n1] -= Gl;
    }
    if (n1 != -1) I[n1] -= i_prev;
    if (n2 != -1) I[n2] += i_prev;
}

void ind_update(Component *c) {
    Ind *p = &c->u.ind;
    float vl = (p->n1!=-1? v[p->n1]:0) - (p->n2!=-1? v[p->n2]:0);
    float Gl = p->dt / p->L;
    float Ieq = Gl * vl;

    p->i_prev = p->i_prev + Ieq;
    p->v_prev = vl;

    if (PRINT_LIN)
        printf("ind_update: i_prev=%.6g  v_prev=%.6g\n", p->i_prev, p->v_prev);
}
```

13.1.8 nonlinear.c

```
// nonlinear.c
#include "circuit.h"
#define CUTOFF_I 1e-8f

// nonlinear components: diode, nmos, pmos

void dio_stamp_nl(Component *c, float Gm[][MAT_SIZE], float I[]) {
    Diode *d = &c->u.dio;
    // 1. read last stage's voltage
    float Vd = d->v_prev;

    // 2. compute new iteration parameters
    float ExpV = expf(Vd / d->Vt);
    float Geq = (d->Is / d->Vt) * ExpV;             // Geq = Is/Vt * exp(Vd/Vt)
    float Ieq = d->Is * (ExpV - 1.0f) - Geq * Vd;   // Ieq = Id - Geq*Vd
    if (PRINT_NL)
        printf("diode_stamp_nl: Vd=%.6g  Geq=%.6g  Ieq=%.6g v_prev=%.6g\n", Vd, Geq, Ieq, Vd);

    // 3. stamp G and I
    int n1 = d->n1, n2 = d->n2;
    if (n1 != -1) Gm[n1][n1] += (Geq);
    if (n2 != -1) Gm[n2][n2] += (Geq);
    if (n1 != -1 && n2 != -1) {
        Gm[n1][n2] -= (Geq);
        Gm[n2][n1] -= (Geq);
    }
    if (n1 != -1) I[n1] -= (Ieq);
    if (n2 != -1) I[n2] += (Ieq);
}
```

```
void dio_update(Component *c) {
    Diode *d = &c->u.dio;
    d->v_prev = (d->n1!=-1? v[d->n1]:0) - (d->n2!=-1? v[d->n2]:0);
    if (PRINT_NL)
        printf("diode_update: v_prev=%.6g\n", d->v_prev);
}


void nmos_stamp_nl(Component *c, float Gm[][MAT_SIZE], float I[]) {
    Nmos *m = &c->u.nmos;
    int ng = m->ng, nd = m->nd, ns = m->ns;
    float beta = m->beta, VT = m->Vt, lambda = m->lambda;

    // 1. read last iteration's voltages
    float Vgs = m->vgs_prev;
    float Vds = m->vds_prev;

    // 2. compute Ids and small-signal gains g_d = dIds/dVds, g_m = dIds/dVgs
    float Ids, g_d, g_m;
    if (Vgs <= VT) {        // cutoff
        Ids = CUTOFF_I * Vds; // small cutoff current
        g_d = CUTOFF_I;
        g_m = 0.0f;
    } else {
        float Vov = Vgs - VT;
        if (Vds < Vov) {    // triode
            Ids = beta * (Vds*Vov - 0.5f*Vds*Vds);
            g_d = beta * (Vov - Vds);
            g_m = beta * Vds;
        } else {            // saturation
            Ids = 0.5f * beta * Vov*Vov * (1 + lambda*(Vds - Vov));
            g_d = 0.5f * beta * Vov*Vov * lambda;
            g_m = beta * Vov * (1 + lambda*(Vds- Vov)) - 0.5f * beta * Vov*Vov *lambda ;
        }
    }

  //    Ieq = Ids_prev - g_d*Vds_prev - g_m*Vgs_prev
    float Ieq = Ids - g_d * Vds - g_m * Vgs;

    // 3. stamp on G and I
    if (nd != -1) {
        Gm[nd][nd] +=  g_d;
        if (ns != -1) Gm[nd][ns] -=  g_d;
    }
    if (ns != -1) {
        Gm[ns][ns] +=  g_d;
        if (nd != -1) Gm[ns][nd] -=  g_d;
    }
    if (ng != -1) {
        // injection at drain
        if (nd != -1) {
            Gm[nd][ng] += g_m;
            if (ns != -1) Gm[nd][ns] += -g_m;
        }
        // injection at source
        if (ns != -1) {
            Gm[ns][ng] += -g_m;
            Gm[ns][ns] += +g_m;
```

```
        }
    }

    if (nd != -1) I[nd] -= Ieq;
    if (ns != -1) I[ns] += Ieq;

    if (PRINT_NL)
        printf("nmos_stamp_nl: vds=%.6g vgs=%.6g Ids=%.6g, g_m=%.6g g_d=%.6g Ieq=%.6g\n", m->vds_prev, m->vgs_prev,  Ids,
        ↪  g_m, g_d, Ieq);
}

void nmos_update(Component *c) {
    Nmos *m = &c->u.nmos;
    float vg = (m->ng != -1 ? v[m->ng] : 0.0f);
    float vd = (m->nd != -1 ? v[m->nd] : 0.0f);
    float vs = (m->ns != -1 ? v[m->ns] : 0.0f);

    float new_vgs = vg - vs;
    float new_vds = vd - vs;

    // clamp each delta to prevent abrupt changes
    float dgs = new_vgs - m->vgs_prev;
    if      (dgs >  DV_MAX) new_vgs = m->vgs_prev + DV_MAX;
    else if (dgs < -DV_MAX) new_vgs = m->vgs_prev - DV_MAX;
    float dds = new_vds - m->vds_prev;
    if      (dds >  DV_MAX) new_vds = m->vds_prev + DV_MAX;
    else if (dds < -DV_MAX) new_vds = m->vds_prev - DV_MAX;

    // store the clamped values
    m->vgs_prev = new_vgs;
    m->vds_prev = new_vds;

    if (PRINT_NL)
        printf("nmos_update (clamped): vgs=%.6g  vds=%.6g\n", m->vgs_prev, m->vds_prev);
}

// — PMOS nonlinear companion stamp —
void pmos_stamp_nl(Component *c, float Gm[][MAT_SIZE], float I[]) {
    Pmos *m = &c->u.pmos;
    int ng = m->ng, nd = m->nd, ns = m->ns;
    float beta = m->beta, VT = m->Vt, lambda = m->lambda;

    // 1. read last iteration's voltages
    float Vsg = m->vsg_prev;
    float Vsd = m->vsd_prev;

    // 2. compute Isd and small-signal gains g_d = dIsd/dVsd, g_m = dIsd/dVsg
    float Isd, g_d, g_m;
    if (Vsg <= VT) {        // cutoff
        Isd = CUTOFF_I * Vsd; // small cutoff current
        g_d = CUTOFF_I;
        g_m = 0.0f;
    } else {
        float Vov = Vsg - VT;
        if (Vsd < Vov) {    // triode
            Isd = beta * (Vsd*Vov - 0.5f*Vsd*Vsd);
            g_d = beta * (Vov - Vsd);
            g_m = beta * Vsd;
```

```
        } else {            // saturation
            Isd = 0.5f * beta * Vov*Vov * (1 + lambda*(Vsd - Vov));
            g_d = 0.5f * beta * Vov*Vov * lambda;
            g_m = beta * Vov * (1 + lambda*(Vsd- Vov)) - 0.5f * beta * Vov*Vov *lambda ;
        }
    }

    //    Ieq = Isd_prev - g_d*Vsd_prev - g_m*Vsg_prev
    float Ieq = Isd - g_d*Vsd - g_m*Vsg;

    // 3. stamp on G and I
    if (nd != -1) {
        Gm[nd][nd] +=  g_d;
        if (ns != -1) Gm[nd][ns] -=  g_d;
    }
    if (ns != -1) {
        Gm[ns][ns] +=  g_d;
        if (nd != -1) Gm[ns][nd] -=  g_d;
    }
    if (ng != -1) {
        // injection at drain
        if (nd != -1) {
            Gm[nd][ng] += g_m;
            if (ns != -1) Gm[nd][ns] += -g_m;
        }
        // injection at source
        if (ns != -1) {
            Gm[ns][ng] += -g_m;
            Gm[ns][ns] += +g_m;
        }
    }

    if (nd != -1) I[nd] += Ieq;
    if (ns != -1) I[ns] -= Ieq;

    if (PRINT_NL)
        printf("pmos_stamp_nl: Vsd=%.6g Vsg=%.6g Isd=%.6g, g_m=%.6g g_d=%.6g Ieq=%.6g\n",
            Vsd, Vsg, Isd, g_m, g_d, Ieq);
}

void pmos_update(Component *c) {
    Pmos *m = &c->u.pmos;
    float vg = (m->ng != -1 ? v[m->ng] : 0.0f);
    float vd = (m->nd != -1 ? v[m->nd] : 0.0f);
    float vs = (m->ns != -1 ? v[m->ns] : 0.0f);

    float new_vsg = vs - vg;
    float new_vsd = vs - vd;

    // clamp each delta to prevent abrupt changes
    float dsg = new_vsg - m->vsg_prev;
    if      (dsg >  DV_MAX) new_vsg = m->vsg_prev + DV_MAX;
    else if (dsg < -DV_MAX) new_vsg = m->vsg_prev - DV_MAX;
    float dsd = new_vsd - m->vsd_prev;
    if      (dsd >  DV_MAX) new_vsd = m->vsd_prev + DV_MAX;
    else if (dsd < -DV_MAX) new_vsd = m->vsd_prev - DV_MAX;

    // store the clamped values
```

```
    m->vsg_prev = new_vsg;
    m->vsd_prev = new_vsd;

    if (PRINT_NL)
        printf("pmos_update (clamped): Vsg=%.6g  Vsd=%.6g\n", m->vsg_prev, m->vsd_prev);
}
```

13.2 Gaussian elimination block

13.2.1 gaussian_elim_compat_tb.sv

```
module gaussian_elim_compat_tb
        #(parameter
          EPSILON = 32'b00110011111010110101111111010101,
          DIV_LATENCY = 11,
          SUB_LATENCY = 2,
          MUL_LATENCY = 2,
          MAT_SHIFT = 7,
          FLOAT_SHIFT = 2,
          MAT_SIZE = 15'd128)
        (input clock,

        input logic start_in,
        input logic reset,
        input logic [7:0] n,

        output logic [14:0] address,
        output logic [31:0] mem_data,
        output logic mem_wren,
        input logic [31:0] mem_result,

        output logic done,
        output logic success,
        output logic singular_out);
typedef enum logic [5:0] {
        GS_IDLE,
        PF_INIT, PF_INIT_MEM_WAIT, PF_READ_DIAG, PF_SCAN_CHECK, PF_READ_VAL, PF_EVALUATE,
        PS_SWAP_G_A_PRE_BUF, PS_SWAP_G_A, PS_SWAP_G_B, PS_SWAP_G_WA, PS_SWAP_G_WB,
        PS_SWAP_I_A, PS_SWAP_I_B, PS_SWAP_I_WA, PS_SWAP_I_WB,
        EL_INIT_READ_PIVOT, EL_INIT_SETUP, EL_READ_AIK, EL_DIV_START, EL_DIV_WAIT,
        EL_COL_SETUP, EL_READ_COL, EL_READ_ROW, EL_MUL_START, EL_MUL_WAIT,
        EL_SUB_START, EL_SUB_WAIT, EL_SUB_WAIT_BUF, EL_WRITE_COL, EL_COL_INCREMENT, EL_ROW_INCREMENT,
        BS_READ_I, BS_SETUP, BS_READ_A, BS_READ_V, BS_MUL_START, BS_MUL_WAIT, BS_SUB_START, BS_SUB_WAIT,
        BS_NEXT_CHECK, BS_READ_DIAG, BS_DIV_START, BS_DIV_WAIT, BS_DIV_WAIT_BUF, BS_DIV_WRITE, BS_ROW_DEC,
        GS_CHECK_K, GS_DONE, GS_FAILED} state_t;
    state_t state;
//    state_t next_state;

    logic singular;
    logic start;
    logic [4:0] fp_latency_count;
    logic [(MAT_SHIFT + 2'd2) << (MAT_SHIFT + FLOAT_SHIFT) - 1'b1:0] mem;

    logic [7:0] k;
    logic [7:0] pivot;
    logic [7:0] row;
```

```
    logic [7:0] j;
    logic [31:0] max_val;
    logic [31:0] val_buf;


    logic [7:0] ei;
    logic [7:0] ej;
    logic [7:0] bi;
    logic [7:0] bj;
    logic [31:0] pivot_val;
    logic [31:0] m;
    logic [31:0] buf_col;
    logic [31:0] buf_row;
    logic [31:0] sum;
    logic [31:0] mem_I;
    logic [31:0] mem_A;
    logic [31:0] mem_v;
    logic [31:0] mem_diag;

    logic sub_en;
    logic [31:0] sub_a;
    logic [31:0] sub_b;
    logic sub_nan;

    logic sub_overflow;
    logic [31:0] sub_output;
    logic sub_underflow;

    fp_sub sub_u(
            .areset(reset),
            .en(sub_en),
            .clk(clock),
            .a(sub_a),
            .b(sub_b),
                 .q(sub_output));

    logic areset;
    logic mul_en;
    logic [31:0] mul_a;
    logic [31:0] mul_b;
    logic [31:0] mul_output;
    logic [31:0] mul_result;

    fp_mult mul_u(
            .clk(clock),
            .areset(areset),
            .en(mul_en),
            .a(mul_a),
            .b(mul_b),
            .q(mul_output));

    logic div_en;
    logic [31:0] div_a;
    logic [31:0] div_b;
    logic [31:0] div_output;

    fp_div div_u(
            .clk(clock),
```

```
            .areset(reset),
            .en(div_en),
            .a(div_a),
            .b(div_b),
            .q(div_output));


function [31:0] fabsf(input [31:0] a);
begin
        fabsf = {1'b0,a[30:0]};
end
endfunction

function [0:0] fbig(input [31:0] a);
begin
        fbig = a[30:23] < 104;
end
endfunction

function [0:0] fgtf(input [31:0] a, input [31:0] b);
begin
        fgtf = (a[31] == 1'b0 && b[31] == 1'b1) ||
                    (a[31] == 1'b0 && b[31] == 1'b0 &&
                    (a[30:23] > b[30:23] ||
                    (a[30:23] == b[30:23] && a[22:0] > b[22:0]))) ||
                    (a[31] == 1'b1 && b[31] == 1'b1 &&
                    (a[30:23] < b[30:23] ||
                    (a[30:23] == b[30:23] && a[22:0] < b[22:0])));
end
endfunction

function [14:0] get_G(input [7:0] i, input [7:0] j);
begin
        get_G = ({7'b0,i} << (MAT_SHIFT)) + {7'b0,j};
end
endfunction

function [14:0] get_I(input [7:0] i);
begin
        get_I = (MAT_SIZE << (MAT_SHIFT)) + {7'b0,i};
end
endfunction

function [14:0] get_v(input [7:0] i);
begin
        get_v = ((MAT_SIZE + 15'b1) << (MAT_SHIFT)) + {7'b0,i};
end
endfunction

function [0:0] mul_exception_check(input[31:0] mul_a, input [31:0] mul_b, input [31:0] mul_output);
begin
        if(mul_output[30:23] == 8'b1111_1111 || mul_a[31] ^ mul_b[31] != mul_output[31]) begin
                mul_exception_check = 1'b1;
        end
        else begin
                mul_exception_check = 1'b0;
        end
end
```

```
endfunction

    function [0:0] sub_exception_check(input[31:0] sub_a, input [31:0] sub_b, input [31:0] sub_output);
    begin
            if (sub_a[30:0] == 31'b0 && sub_b[30:0] == 31'b0) begin
               sub_exception_check = 1'b0;
            end
       else if(sub_output[30:23] == 8'b1111_1111 || (sub_a == sub_b && sub_output[30:0] != 31'b0) ||
       ↪  (fgtf(sub_a,sub_b) ^ (sub_output[31] == 1'b0 && sub_output[30:0] != 31'b0))) begin
                sub_exception_check = 1'b1;
            end
            else begin
                    sub_exception_check = 1'b0;
            end
    end
    endfunction

    function [0:0] div_exception_check(input[31:0] div_a, input [31:0] div_b, input [31:0] div_output);
    begin
       if(mul_exception_check(div_a,div_b,div_output) || div_b[30:0] == 31'b0) begin
                div_exception_check = 1'b1;
            end
            else begin
                div_exception_check = 1'b0;
            end
    end
    endfunction
always_ff @(posedge clock or posedge reset) begin
    //    start <= start_in;
       if(reset) begin
               done <= 1'b0;
               success <= 1'b0;
               singular_out <= 1'b0;
               //start = start_in;
               state <= GS_IDLE;
               k <= 8'd0;
                   pivot <= 8'd0;
                   row <= 8'd0;
                   j <= 8'd0;
                   max_val <= 32'd0;
                   val_buf <= 32'd0;
                   ei <= 7'd0; ej <= 7'd0; bi <= 7'd0; bj <= 7'd0;
                   pivot_val <= 32'b0;
                   m <= 32'b0; buf_col <= 32'b0; buf_row <= 32'b0;
                   sum <= 32'b0; mem_I <= 32'b0; mem_v <= 32'b0; mem_diag <= 32'b0;

                   sub_en <= 1'b0; mul_en <= 1'b0; div_en <= 1'b0;
                   sub_a <= 32'b0; sub_b <= 32'b0; mul_a <= 32'b0; mul_b <= 32'b0; div_a <= 32'b0; div_b <= 32'b0;

            end
       else begin
               singular_out <= singular;
               case(state)
                       GS_IDLE: begin
                               done <= 1'b0;
                               success <= 1'b0;
                               singular <= 1'b0;
                               if(start_in) begin
```

```
                state <= PF_INIT;
        end


end
PF_INIT: begin
        row <= k;
        pivot <= k;
        state <= PF_INIT_MEM_WAIT;

        address <= get_G(k,k);
        mem_wren <= 1'b0;

end
PF_INIT_MEM_WAIT: begin
        state <= PF_READ_DIAG;
end
PF_READ_DIAG: begin
        max_val <= fabsf(mem_result);
        state <= PF_SCAN_CHECK;
end
PF_SCAN_CHECK: begin
        row <= row + 8'b1;
        if(row + 8'b1 < n) begin
                state <= PF_READ_VAL;
                address <= get_G(row + 8'b1,k);
                mem_wren <= 1'b0;
        end
        else begin
                j <= k;
                if(fbig(fabsf(max_val))) begin //checks exponent, sees if its less than 2e-7
                        singular <= 1'b1;
                        state <= GS_DONE;
                end
                else begin
                        state <= PS_SWAP_G_A_PRE_BUF;
                        address <= get_G(k,k); //j being updated this cycle to value of k, so
                        ↪  must set it to be k,k
                        mem_wren <= 1'b0;

                end
        end
end
PF_READ_VAL: begin
        state <= PF_EVALUATE;
end
PF_EVALUATE: begin
        if(fgtf(fabsf(mem_result),max_val)) begin
                max_val <= fabsf(mem_result);
                pivot <= row;
        end
        state <= PF_SCAN_CHECK;
end
PS_SWAP_G_A_PRE_BUF: begin
        state <= PS_SWAP_G_A;

        if(pivot != k) begin
                address <= get_G(pivot,j);
```

```
                mem_wren <= 1'b0;
        end
        else begin
                address <= get_G(k,k);
                mem_wren <= 1'b0;
        end
end
PS_SWAP_G_A: begin
        if(pivot == k) begin
                state <= EL_INIT_READ_PIVOT;
        end
        else begin
                state <= PS_SWAP_G_B;
                //address <= get_G(pivot,j);
                //mem_wren <= 1'b0;
                address <= get_G(pivot,j);
                mem_wren <= 1'b1;
                mem_data <= mem_result;
        end
end
PS_SWAP_G_B: begin
        state <= PS_SWAP_G_WA;
        //address <= get_G(k,j);
        //mem_wren <= 1'b1;
        //mem_data <= mem_result;
        address <= get_G(k,j);
        mem_wren <= 1'b1;
        mem_data <= mem_result;
end
PS_SWAP_G_WA: begin
        //write performed
        //mem_wren <= 1'b0;
        state <= PS_SWAP_G_WB;
        //address <= get_G(pivot,j);
        //mem_wren <= 1'b1;
        if(j + 8'b1 < n) begin
                address <= get_G(k,j + 8'b1);
                mem_wren <= 1'b0;

        end
        else begin
                address <= get_I(k);
                mem_wren <= 1'b0;
        end
end
PS_SWAP_G_WB: begin
        //write performed
        j <= j + 8'b1;
        if(j + 8'b1 < n) begin
                state <= PS_SWAP_G_A;
                //address <= get_G(k,j + 8'b1);
                //mem_wren <= 1'b0;
                address <= get_G(pivot,j + 8'b1);
                mem_wren <= 1'b0;
        end
        else begin
                state <= PS_SWAP_I_A;
                //address <= get_I(k);
```

```
                //mem_wren <= 1'b0;
                address <= get_I(pivot);
                mem_wren <= 1'b0;
        end
end
PS_SWAP_I_A: begin
        state <= PS_SWAP_I_B;
        //address <= get_I(pivot);
        //mem_wren <= 1'b0;
        address <= get_I(pivot);
        mem_wren <= 1'b1;
        mem_data <= mem_result;
end
PS_SWAP_I_B: begin
        mem_data <= mem_result;
        state <= PS_SWAP_I_WA;
        //address <= get_I(k);
        //mem_wren <= 1'b1;
        //mem_data <= mem_result; //temp_b gets set that cycle, must use mem_result
        address <= get_I(k);
        mem_wren <= 1'b1;
        mem_data <= mem_result;
end
PS_SWAP_I_WA: begin
        //write performed
        //mem_wren <= 1'b0;
        state <= PS_SWAP_I_WB;
        //address <= get_I(pivot);
        //mem_wren <= 1'b1;
        address <= get_G(k,k);
        mem_wren <= 1'b0;
end
PS_SWAP_I_WB: begin
        //write performed
        //mem_wren <= 1'b0;
        state <= EL_INIT_READ_PIVOT;
        //address <= get_G(k,k);
        //mem_wren <= 1'b0;
end


EL_INIT_READ_PIVOT: begin
        pivot_val <= mem_result;
        state <= EL_INIT_SETUP;
        if(!fbig(fabsf(mem_result))) begin
                address <= get_G(k + 8'b1, k);
                mem_wren <= 1'b0;
        end
end
EL_INIT_SETUP: begin
        if(fbig(fabsf(pivot_val))) begin
                singular <= 1'b1;
                state <= GS_DONE;
        end
        else begin
                ei <= k + 8'b1;
                state <= EL_READ_AIK;
                //address <= get_G(k + 8'b1, k); //ei changing to value of k + 1 in this cycle
```

```
                //mem_wren <= 1'b0;
        end
end
EL_READ_AIK: begin
        if(ei < n) begin
                buf_col <= mem_result;
                state <= EL_DIV_START;
        end
        else begin
                state <= GS_CHECK_K;
        end
end
EL_DIV_START: begin
        div_a <= buf_col;
        div_b <= pivot_val;
        div_en <= 1'b1;
        state <= EL_DIV_WAIT;
        fp_latency_count <= 5'b0;
        //FOR ALL STARTS OF FP CALCULATIONS MUST
        //0 OUT LATENCY COUNTER
end
EL_DIV_WAIT: begin
        //div_en <= 1'b0;
        fp_latency_count <= fp_latency_count + 5'b1;
        if(fp_latency_count == DIV_LATENCY) begin
               div_en <= 1'b0;
               if(div_exception_check(div_a,div_b,div_output)) begin
                       state <= GS_FAILED;
               end
               else begin
                       state <= EL_COL_SETUP;

                       if(k < n) begin
                                address <= get_G(k,k);
                                mem_wren <= 1'b0;
                       end
                           else begin
                                address <= get_I(k);
                                mem_wren <= 1'b0;
                       end
               end
        end
end
EL_COL_SETUP: begin
        m <= div_output;
        ej <= k;
        state <= EL_READ_COL;
        /*if(k < n) begin
                address <= get_G(k,k);
                mem_wren <= 1'b0;
        end
        else begin
                address <= get_I(k);
                mem_wren <= 1'b0;
        end*/
        if(k < n) begin
                address <= get_G(ei,k);
                mem_wren <= 1'b0;
```

```
                end
                else begin
                        address <= get_I(ei);
                        mem_wren <= 1'b0;
                end

        end
        EL_READ_COL: begin
                /*if(ej < n) begin
                        buf_col <= 1;//get_G(k,ej)
                end
                else begin
                        buf_col <= 1;//get_I(k)
                end*/
                buf_col <= mem_result;
                state <= EL_READ_ROW;
                /*if(ej < n) begin
                        address <= get_G(ei,ej);
                        mem_wren <= 1'b0;
                end
                else begin
                        address <= get_I(ei);
                        mem_wren <= 1'b0;
                end*/
        end
        EL_READ_ROW: begin
                /*if(ej < n) begin
                        buf_row <= 1;//get_G(ei,ej)
                end
                else begin
                        buf_row <= 1;//get_I(ei)
                end*/
                buf_row <= mem_result;
                state <= EL_MUL_START;
        end
        EL_MUL_START: begin
                mul_a <= m;
                mul_b <= buf_col;
                mul_en <= 1'b1;
                state <= EL_MUL_WAIT;
                fp_latency_count <= 5'b0;
        end
        EL_MUL_WAIT: begin
                //mul_en <= 1'b0;
                fp_latency_count <= fp_latency_count + 5'b1;
                if(fp_latency_count == MUL_LATENCY) begin
                        //insert basic exception check?,
                        mul_en <= 1'b0;
                        if(mul_exception_check(mul_a,mul_b,mul_output)) begin
                                state <= GS_FAILED;
                        end
                        else begin
                                state <= EL_SUB_START;
                        end
                end
        end
        EL_SUB_START: begin
                sub_a <= buf_row;
```

```
                                sub_b <= mul_output;
                                sub_en <= 1'b1;
                                state <= EL_SUB_WAIT;
                                fp_latency_count <= 5'b0;
                        end
                        EL_SUB_WAIT: begin
                                //sub_en <= 1'b0;
                                fp_latency_count <= fp_latency_count + 5'b1;
                                if(fp_latency_count == SUB_LATENCY) begin
                                        sub_en <= 1'b0;
                                        if(sub_exception_check(sub_a,sub_b,sub_output)) begin
                                                state <= GS_FAILED;
                                        end
                                        else begin
                                                state <= EL_SUB_WAIT_BUF;


                                                if(ej < n) begin
                                                        address <= get_G(ei,ej);
                                                        mem_wren <= 1'b1;
                                                        mem_data <= sub_output;
                                                end
                                                else begin
                                                        address <= get_I(ei);
                                                        mem_wren <= 1'b1;
                                                        mem_data <= sub_output;
                                                end
                                                        end
                                end
                        end
                        EL_SUB_WAIT_BUF: begin
/*                                        if(ej < n) begin
                                              address <= get_G(ei,ej);
                                              mem_wren <= 1'b1;
                                              mem_data <= sub_output;
                                        end
                                        else begin
                                                address <= get_I(ei);
                                                mem_wren <= 1'b1;
                                                mem_data <= sub_output;
                                        end*/
                                        state <= EL_WRITE_COL;
                        end
                        EL_WRITE_COL: begin
                                /*if(ej < n) begin
                                        //write(get_G(ei,ej), sub_result)
                                end
                                else begin
                                        //write(get_I(ei), sub_result
                                end*/
                                //mem_wren <= 1'b0;
                                state <= EL_COL_INCREMENT;

                                if(ej + 8'b1 <= n) begin
                                        if(ej + 8'b1 < n) begin
                                                address <= get_G(k,ej + 8'b1);
                                               // address <= get_G(ei, ej + 8'b1);
                                                mem_wren <= 1'b0;
```

```
                end
                else begin
                        address <= get_I(k);
                        mem_wren <= 1'b0;
                end
        end
end
EL_COL_INCREMENT: begin
        ej <= ej + 8'b1;
        if(ej + 8'b1 <= n) begin
                 state <= EL_READ_COL;
                if(ej + 8'b1 < n) begin
                        address <= get_G(ei,ej + 8'b1);
                        mem_wren <= 1'b0;
                end
                else begin
                        address <= get_I(ei);
                        mem_wren <= 1'b0;
                end
        end
        else begin
                state = EL_ROW_INCREMENT;
                if(ei + 8'b1 < n) begin
                        address <= get_G(ei + 8'b1, k);
                        mem_wren <= 1'b0;
                end
        end



     end
EL_ROW_INCREMENT: begin
        ei <= ei + 8'b1;
        if(ei + 8'b1 < n) begin
                state <= EL_READ_AIK;
                //address <= get_G(ei + 8'b1, k); //ei being incremented in this cycle
                //mem_wren <= 1'b0;
        end
        else begin
                state <= GS_CHECK_K;
        end
end
GS_CHECK_K: begin
        k <= k + 8'b1;
        if(k + 8'b1 < n) begin
                state <= PF_INIT;
        end
        else begin
                bi <= n - 8'b1;
                bj <= n;
                state <= BS_READ_I;
                address <= get_I(n - 8'b1);
                mem_wren <= 1'b0;
        end

end
BS_READ_I: begin
        //mem_I <= mem_result;
```

```
                state <= BS_SETUP;
                if(bj < n) begin
                        address <= get_G(bi, bj);
                        mem_wren <= 1'b0;
                end
                else begin
                        address <= get_G(bi, bi);
                        mem_wren <= 1'b0;
                end
        end
        BS_SETUP: begin
                sum <= mem_result;
                bj <= bi + 8'b1;
                if(bj < n) begin
                        state <= BS_READ_A;
                        //address <= get_G(bi + 8'b1, bj);
                        //mem_wren <= 1'b0;
                        address <= get_v(bi + 8'b1);
                        mem_wren <= 1'b0;
                end
                else begin
                        state <= BS_READ_DIAG;
                        //address <= get_G(bi, bi);
                        //mem_wren <= 1'b0;
                end
        end
        BS_READ_A: begin
                mem_A <= mem_result;
                state <= BS_READ_V;
                //address <= get_v(bj);
                //mem_wren <= 1'b0;
        end
        BS_READ_V: begin
                mem_v <= mem_result;
                state <= BS_MUL_START;
        end
        BS_MUL_START: begin
                mul_a <= mem_A;
                mul_b <= mem_v;
                mul_en <= 1'b1;
                state <= BS_MUL_WAIT;
                fp_latency_count <= 5'b0;
        end
        BS_MUL_WAIT: begin
                //mul_en <= 1'b0;
                fp_latency_count <= fp_latency_count + 5'b1;
                if(fp_latency_count == MUL_LATENCY) begin
                        mul_en <= 1'b0;
                        if(mul_exception_check(mul_a,mul_b,mul_output)) begin
                                state <= GS_FAILED;
                        end
                        else begin
                                //sum <= mul_outpusim:/gaussian_elim_tb/test/start

                                state <= BS_SUB_START;


                                /*if(bj + 8'b1 < n) begin
```

```
                                address <= get_G(bi,bj + 8'b1);
                                mem_wren <= 1'b0;
                        end
                        else begin
                                address <= get_G(bi,bi);
                                mem_wren <= 1'b0;
                        end*/
                end
        end
end
BS_SUB_START: begin
        sub_a <= sum;
        sub_b <= mul_output;
        sub_en <= 1'b1;
        state <= BS_SUB_WAIT;
        fp_latency_count <= 5'b0;
end
BS_SUB_WAIT: begin
        fp_latency_count <= fp_latency_count + 5'b1;
        if(fp_latency_count == SUB_LATENCY) begin
                sub_en <= 1'b0;
                if(sub_exception_check(sub_a,sub_b,sub_output)) begin
                        state <= GS_FAILED;
                end
                else begin
                        state <= BS_NEXT_CHECK;
                    sum <= sub_output;
                        if(bj + 8'b1 < n) begin
                                address <= get_G(bi,bj + 8'b1);
                                mem_wren <= 1'b0;
                        end
                        else begin
                                address <= get_G(bi,bi);
                                mem_wren <= 1'b0;
                        end
                end
        end
end

BS_NEXT_CHECK: begin
        bj <= bj + 8'b1;
        if(bj + 8'b1 < n) begin
                state <= BS_READ_A;
                address <= get_v(bj + 8'b1);
                mem_wren <= 1'b0;
        end
        else begin
                state <= BS_READ_DIAG;
                //address <= get_G(bi,bi);
                //mem_wren <= 1'b0;
        end
end
BS_READ_DIAG: begin
        mem_diag <= mem_result;
        state <= BS_DIV_START;
end
BS_DIV_START: begin
        if(fbig(fabsf(mem_diag))) begin
```

```
                        state <= GS_DONE;
                end
                else begin
                        div_a <= sum;
                        div_b <= mem_diag;
                        div_en <= 1'b1;
                        state <= BS_DIV_WAIT;
                        fp_latency_count <= 5'b0;
                end
        end
        BS_DIV_WAIT: begin
                //div_en <= 1'b0;
                fp_latency_count <= fp_latency_count + 5'b1;
                if(fp_latency_count == DIV_LATENCY) begin
                        div_en <= 1'b0;
                        if(div_exception_check(div_a,div_b,div_output)) begin
                                state <= GS_FAILED;
                        end
                        else begin
                                //write(get_v(bi),div_output) performed in next state
                                state <= BS_DIV_WAIT_BUF;
                                address <= get_v(bi);
                                mem_wren <= 1'b1;
                                mem_data <= div_output;
                        end
                end
        end
        BS_DIV_WAIT_BUF: begin
                //address <= get_v(bi);
                //mem_wren <= 1'b1;
                //mem_data <= div_output;

                state <= BS_DIV_WRITE;
        end
        BS_DIV_WRITE: begin
                mem_wren <= 1'b0;
                state <= BS_ROW_DEC;

                if(bi - 8'b1 >= 0) begin
                        address <= get_I(bi - 8'b1);
                        mem_wren <= 1'b0;
                end
        end
        BS_ROW_DEC: begin
                bi <= bi - 8'b1;
                bj <= bi;
                if(bi - 8'b1 >= 0) begin
                        state <= BS_READ_I;
                        //address <= get_I(bi - 8'b1);
                        //mem_wren <= 1'b0;
                end
                else begin
                        state <= GS_DONE;
                end
        end
        GS_DONE: begin
                done <= 1;
                success <= 1;
```

```
                                    // reset only when starting again
                                    if (start == 1'b1 || reset == 1'b1)
                                        state <= GS_IDLE;
                            end
                            GS_FAILED: begin
                                    done <= 1;
                                    success <= 0;
                                    if (start == 1'b1 || reset == 1'b1)
                                        state <= GS_IDLE;
                            end
                    endcase
            end
    end
endmodule
```

13.2.2 gaussian_top.sv

```
module gaussian_top
(
        input logic clk,
        input logic reset,

        input logic [7:0] csr_writedata,
        input logic csr_write,
        input logic csr_chipselect,
        input logic [1:0] csr_address,


        output logic [7:0] csr_readdata,

        input logic [31:0] mem_writedata,
        input logic mem_write,
        input logic [14:CXX0]  mem_address,
        input logic mem_chipselect,
        output logic [31:0] mem_readdata
);
// status and control register logic
logic [7:0] n;
logic go;
logic csr_reset;

logic done;
logic success;
logic singular_out;

always_comb begin
        csr_readdata = {done, ~success, singular_out, 5'b0};
end

always_ff @(posedge clk) begin
        if (csr_chipselect && csr_write)
                case (csr_address)
                        2'h0: n <= csr_writedata;
                        2'h1: begin
                                go <= csr_writedata[0:0];
                                csr_reset <= csr_writedata[1:1];
```

```
                        end
                        endcase
        end


// two port memory instance
logic [31:0] gaussian_writedata;
logic [14:0] gaussian_address;
logic gaussian_write;
logic [31:0] gaussian_readdata;

mem2p mem(
        .address_a(mem_address),
        .address_b(gaussian_address),
        .clock(clk),
        .data_a(mem_writedata),
        .data_b(gaussian_writedata),
        .wren_a(mem_write),
        .wren_b(gaussian_write),
        .q_a(mem_readdata),
        .q_b(gaussian_readdata)
);


// instance of gaussian core
gaussian_elim_compat_tb gaussian(
        .clock(clk),
        .start_in(go),
        .reset(csr_reset),
        .n(n),
        .address(gaussian_address),
        .mem_data(gaussian_writedata),
        .mem_wren(gaussian_write),
        .mem_result(gaussian_readdata),
        .done(done),
        .success(success),
        .singular_out(singular_out)
);


endmodule
```

13.3 Memory

13.3.1 `mem2p.c`

```
// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: mem2p.v
// Megafunction Name(s):
//                        altsyncram
//
```

```
// Simulation Library Files(s):
//                        altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 21.1.0 Build 842 10/21/2021 SJ Lite Edition
// ************************************************************


//Copyright (C) 2021  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and any partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel FPGA IP License Agreement, or other applicable license
//agreement, including, without limitation, that your use is for
//the sole purpose of programming logic devices manufactured by
//Intel and sold by Intel or its authorized distributors.  Please
//refer to the applicable agreement for further details, at
//https://fpgasoftware.intel.com/eula.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module mem2p (
        address_a,
        address_b,
        clock,
        data_a,
        data_b,
        wren_a,
        wren_b,
        q_a,
        q_b);

        input        [14:0]  address_a;
        input        [14:0]  address_b;
        input                clock;
        input        [31:0]  data_a;
        input        [31:0]  data_b;
        input                wren_a;
        input                wren_b;
        output       [31:0]  q_a;
        output       [31:0]  q_b;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
        tri1                 clock;
        tri0                 wren_a;
        tri0                 wren_b;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif
```

```
wire [31:0] sub_wire0;
wire [31:0] sub_wire1;
wire [31:0] q_a = sub_wire0[31:0];
wire [31:0] q_b = sub_wire1[31:0];

altsyncram        altsyncram_component (
                        .address_a (address_a),
                        .address_b (address_b),
                        .clock0 (clock),
                        .data_a (data_a),
                        .data_b (data_b),
                        .wren_a (wren_a),
                        .wren_b (wren_b),
                        .q_a (sub_wire0),
                        .q_b (sub_wire1),
                        .aclr0 (1'b0),
                        .aclr1 (1'b0),
                        .addressstall_a (1'b0),
                        .addressstall_b (1'b0),
                        .byteena_a (1'b1),
                        .byteena_b (1'b1),
                        .clock1 (1'b1),
                        .clocken0 (1'b1),
                        .clocken1 (1'b1),
                        .clocken2 (1'b1),
                        .clocken3 (1'b1),
                        .eccstatus (),
                        .rden_a (1'b1),
                        .rden_b (1'b1));
defparam
        altsyncram_component.address_reg_b = "CLOCK0",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_input_b = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.clock_enable_output_b = "BYPASS",
        altsyncram_component.indata_reg_b = "CLOCK0",
        altsyncram_component.init_file = "test_memory.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 32768,
        altsyncram_component.numwords_b = 32768,
        altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_aclr_b = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.outdata_reg_b = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M10K",
        altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.read_during_write_mode_port_b = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.widthad_a = 15,
        altsyncram_component.widthad_b = 15,
        altsyncram_component.width_a = 32,
        altsyncram_component.width_b = 32,
        altsyncram_component.width_byteena_a = 1,
        altsyncram_component.width_byteena_b = 1,
```

```
                altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0";


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC "0"
// Retrieval info: PRIVATE: CLRq NUMERIC "0"
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRrren NUMERIC "0"
// Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRwren NUMERIC "0"
// Retrieval info: PRIVATE: Clock NUMERIC "0"
// Retrieval info: PRIVATE: Clock_A NUMERIC "0"
// Retrieval info: PRIVATE: Clock_B NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MEMSIZE NUMERIC "1048576"
// Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "1"
// Retrieval info: PRIVATE: MIFfilename STRING "test_memory.mif"
// Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "3"
// Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "2"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "2"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "0"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: REGrren NUMERIC "0"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
// Retrieval info: PRIVATE: VarWidth NUMERIC "0"
```

```
// Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "32"
// Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "32"
// Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "32"
// Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "32"
// Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: INDATA_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: INIT_FILE STRING "test_memory.mif"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "32768"
// Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "32768"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "BIDIR_DUAL_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// Retrieval info: CONSTANT: OUTDATA_REG_B STRING "UNREGISTERED"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: RAM_BLOCK_TYPE STRING "M10K"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS STRING "DONT_CARE"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING "NEW_DATA_NO_NBE_READ"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_B STRING "NEW_DATA_NO_NBE_READ"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "15"
// Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "15"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
// Retrieval info: CONSTANT: WIDTH_B NUMERIC "32"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_B NUMERIC "1"
// Retrieval info: CONSTANT: WRCONTROL_WRADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: USED_PORT: address_a 0 0 15 0 INPUT NODEFVAL "address_a[14..0]"
// Retrieval info: USED_PORT: address_b 0 0 15 0 INPUT NODEFVAL "address_b[14..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data_a 0 0 32 0 INPUT NODEFVAL "data_a[31..0]"
// Retrieval info: USED_PORT: data_b 0 0 32 0 INPUT NODEFVAL "data_b[31..0]"
// Retrieval info: USED_PORT: q_a 0 0 32 0 OUTPUT NODEFVAL "q_a[31..0]"
// Retrieval info: USED_PORT: q_b 0 0 32 0 OUTPUT NODEFVAL "q_b[31..0]"
// Retrieval info: USED_PORT: wren_a 0 0 0 0 INPUT GND "wren_a"
// Retrieval info: USED_PORT: wren_b 0 0 0 0 INPUT GND "wren_b"
// Retrieval info: CONNECT: @address_a 0 0 15 0 address_a 0 0 15 0
// Retrieval info: CONNECT: @address_b 0 0 15 0 address_b 0 0 15 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 32 0 data_a 0 0 32 0
// Retrieval info: CONNECT: @data_b 0 0 32 0 data_b 0 0 32 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren_a 0 0 0 0
// Retrieval info: CONNECT: @wren_b 0 0 0 0 wren_b 0 0 0 0
// Retrieval info: CONNECT: q_a 0 0 32 0 @q_a 0 0 32 0
// Retrieval info: CONNECT: q_b 0 0 32 0 @q_b 0 0 32 0
// Retrieval info: GEN_FILE: TYPE_NORMAL mem2p.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem2p.inc FALSE
```

```
// Retrieval info: GEN_FILE: TYPE_NORMAL mem2p.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem2p.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem2p_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL mem2p_bb.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

13.4 I/O

13.4.1 gaussian.h

```
#ifndef _GAUSSIAN_H
#define _GAUSSIAN_H

#include <sys/ioctl.h>
#include <stdint.h>

/*
 * Register mapping (24b total)
 * --------------------------------
 * |    n (8b)                  |  |
 * ------------------------------  > gaussian_in_t
 * | G | R |    unused (6b)     |  |
 * --------------------------------
 * | D | E | S | unused (5b)    | > gaussian_out_t
 *
 * First 16b: gaussian_in_t
 * Last 8b: gaussian_out_t
 */

#pragma pack(push, 1)

typedef struct {
    uint8_t n;         // 8-bit input
    // next byte: 3 flags + 5 bits reserved
//    uint8_t g:1;
//    uint8_t r:1;
    uint8_t :6;        // unused
    uint8_t r:1;
    uint8_t g:1;

} gaussian_in_t;


typedef struct {
    // next byte: 2 flags + 6 bits reserved
    uint8_t d:1;
    uint8_t e:1;
    uint8_t s:1;
    uint8_t :5;        // unused
} gaussian_out_t;

typedef struct {
  gaussian_in_t input;
  gaussian_out_t output;
} gaussian_arg_t;
```

```
void print_in(int);

gaussian_out_t get_out(int);

void set_in(const gaussian_in_t*, int);

/* TODO: check the sizeof the structs */

#define GAUSSIAN_MAGIC 'q'

/* ioctls and their arguments */
#define GAUSSIAN_WRITE _IOW(GAUSSIAN_MAGIC, 1, gaussian_arg_t)
#define GAUSSIAN_READ  _IOR(GAUSSIAN_MAGIC, 2, gaussian_arg_t)
#pragma pack(pop)

#endif
```

13.4.2 gaussian.c

```
/* * Device driver for the matrix solver
 *
 * A Platform device implemented using the misc subsystem
 *
 * Ming Gong
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod gaussian.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree gaussian.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "gaussian.h"
```

```
#define DRIVER_NAME "gaussian"

/* Device registers */
#define N(x)           ((x)+0)
#define IFLAGS(x) ((x)+1)
#define OFLAGS(x) ((x)+2)

/*
 * Information about our device
 */
struct gaussian_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
        gaussian_in_t in;
                gaussian_out_t out;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_in(gaussian_in_t *in)
{
        iowrite8(in->n, N(dev.virtbase) );
        u8 iflags;
        iflags =  (in->g & 0x1)        /* bit 0 */
                                | ((in->r & 0x1) << 1); /* bit 1 */
        iowrite8(iflags, IFLAGS(dev.virtbase));
        dev.in = *in;
}

static void read_out(gaussian_out_t *out)
{
    uint8_t oflags = ioread8(OFLAGS(dev.virtbase));
    out->d = (oflags >> 7) & 1;
    out->e = (oflags >> 6) & 1;
    out->s = (oflags >> 5) & 1;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long gaussian_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        gaussian_arg_t vla;

        switch (cmd) {
        case GAUSSIAN_WRITE:
                if (copy_from_user(&vla, (gaussian_arg_t *) arg,
                                   sizeof(gaussian_arg_t)))
                        return -EACCES;
                //write_background(&vla.background);
                write_in(&vla.input);
                break;
```

```
        case GAUSSIAN_READ:
                read_out(&vla.output);
                if (copy_to_user((gaussian_arg_t *) arg, &vla,
                                                sizeof(gaussian_arg_t)))
                        return -EACCES;
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations gaussian_fops = {
        .owner                = THIS_MODULE,
        .unlocked_ioctl = gaussian_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice gaussian_misc_device = {
        .minor                = MISC_DYNAMIC_MINOR,
        .name                = DRIVER_NAME,
        .fops                = &gaussian_fops,
};

/*
 * Initialization code: get resources (registers)
 */
static int __init gaussian_probe(struct platform_device *pdev)
{
        int ret;

        /* Register ourselves as a misc device: creates /dev/gaussian */
        ret = misc_register(&gaussian_misc_device);
    if (ret) {
        pr_err("gaussian: misc_register failed (%d)\n", ret);
        return ret;
    }

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
        pr_err("gaussian: region busy @ %pa\n", &dev.res.start);
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
```

```
        if (dev.virtbase == NULL) {
            pr_err("gaussian: iomap failed\n");
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        /* Welcome banner */
    pr_info("gaussian: probed @ phys %pa, virt %p\n",
            &dev.res.start, dev.virtbase);
    pr_info("gaussian: ready!  Welcome to the Gaussian driver.\n");

        // clear all inputs
    memset(&dev.in, 0, sizeof(dev.in));
    write_in(&dev.in);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&gaussian_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int gaussian_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&gaussian_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id gaussian_of_match[] = {
        { .compatible = "csee4840,gaussian-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, gaussian_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver gaussian_driver = {
        .driver        = {
                .name        = DRIVER_NAME,
                .owner        = THIS_MODULE,
                .of_match_table = of_match_ptr(gaussian_of_match),
        },
        .remove        = __exit_p(gaussian_remove),
};

/* Called when the module is loaded: set things up */
static int __init gaussian_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&gaussian_driver, gaussian_probe);
}
```

```
/* Calball when the module is unloaded: release resources */
static void __exit gaussian_exit(void)
{
        platform_driver_unregister(&gaussian_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(gaussian_init);
module_exit(gaussian_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ming Gong, Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("Gaussian driver");
```

13.4.3 gaussian.mod.c

```
#include <linux/build-salt.h>
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

BUILD_SALT;

MODULE_INFO(vermagic, VERMAGIC_STRING);
MODULE_INFO(name, KBUILD_MODNAME);

__visible struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
        .name = KBUILD_MODNAME,
        .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
        .exit = cleanup_module,
#endif
        .arch = MODULE_ARCH_INIT,
};

#ifdef RETPOLINE
MODULE_INFO(retpoline, "Y");
#endif

static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";

MODULE_ALIAS("of:N*T*Ccsee4840,gaussian-1.0");
MODULE_ALIAS("of:N*T*Ccsee4840,gaussian-1.0C*");
```

13.5 Test Circuits

13.5.1 Simple Circuit

```
* Simple

* Components
I1 1 0 DC 5;
R1 1 2 0.5;
R2 2 0 0.5;

.TRAN 5e-6 3e-5

* End of Netlist
.END
```

13.5.2 Voltage Source

```
* Simple Voltage Source and Resistor Circuit

* Components
V1 1 0 DC 5;
R1 1 0 1k;

* End of Netlist
.END
```

13.5.3 LRC Circuit

```
* Simple LRC

* Components
V1 1 0 DC 5
R1 1 2 10;
L1 2 3 1e-3;
C1 3 0 1e-6;

* End of Netlist
.END
```

13.5.4 Opamp Integrator

```
// RC intergrator with opamp (modelled by a vcvs)
#include "circuit.h"
#include "driver.h"
#include "input_funcs.h"
#include <string.h>   // for memset

float time_step = 5e-6f;
int nsteps = 10000;
int opamp_gain = 1e6f;
```

```
void setup(void) {
    add_vsrc(1, -1, 0, dc5);
    add_res(2, 1, 1000.0f);
    add_cap(2, 3, 1e-5f, time_step);
    add_vcvs(3, -1, -1, 2, 4, opamp_gain);
    nnodes = 5;
}

// choose your own header
void print_header(FILE *f) {
    fprintf(f, "time,vin,vout\n");
}

// choose your own row format
void print_row(FILE *f) {
    // you have access to t, v[], etc.
    fprintf(f, "%g,%g,%g\n",
            t,      // time
            v[1],
            v[3]
    );
}
```

13.5.5 4th Order Filter

```
* Simple LRC

* Components
V1 1 0 SIN(0 1 500)
V2 2 1 SIN(0 1 2000)

R1 2 3 10000
C1 4 0 15.915e-9
R2 3 4 10000
C2 3 5 15.915e-9

E1 5 0 4 5 1e6

R3 5 6 10000
C3 7 0 15.915e-9
R4 6 7 10000
C4 6 8 15.915e-9

E2 8 0 7 8 1e6

.TRAN 5e-6 5e-2

.END
```

13.5.6 Half Wave Rectifier

```
* Diode regulator

* Model
```

```
.model diode_model D (RS=16 CJO=2PF TT=12N BV=100 IBV=0.1PA, IS=1E-14)

* Components
V1 1 0 SIN(0 5 1000)
R1 1 2 1.0
D1 2 3 diode_model
C1 3 0 1e-6 5e-6
R2 3 0 1000.0

.TRAN 5e-6 5e-4

.END
```

13.5.7 Data Flip-Flop

```
#include "circuit.h"
#include "driver.h"
#include "input_funcs.h"
#include <string.h>   // for memset

float time_step = 1e-6f;
int nsteps = 10000;

void setup(void) {
    // Node 0, 1: vsrc, vsrc'
    add_vsrc(0, -1, 1, dc5);
    // 2, 3, 4 vin, vin'
    add_vsrc(2, -1, 3, square5);
    add_not(2, 4, 0);
    // 5, 6: CLK
    add_vsrc(5, -1, 6, clk10k);
    // 7, 8: CLK', CLK''
    add_not(5, 7, 0);
    add_not(7, 8, 0);
    // 9-12: master NAND internals
    // 13, 14: master NAND out-
    add_nand(2, 7, 13, 9, 0);
    add_nand(4, 7, 14, 10, 0);
    // 15: 16: master out, out'
    add_nand(13, 16, 15, 11, 0);    // Q master
    add_nand(14, 15, 16, 12, 0);

    // 17-20: slave NAND internals
    // 21, 22: slave NAND out-in
    add_nand(15, 8, 21, 17, 0);
    add_nand(16, 8, 22, 18, 0);
    // 23: 24: slave out, out'
    add_nand(21, 24, 23, 19, 0);
    add_nand(22, 23, 24, 20, 0);

    add_cap(23, -1, 1e-7f, time_step);   // output cap
    /*The output cap's charging period must << CLK period. Otherwise it will fail to charge
    This behavior is expected. Falstad yields similar results.
    For our configuration, 1uF is fine, but beyond 2uF, you will see charging issues*/
    nnodes =25;
}
```

```
// choose your own header
void print_header(FILE *f) {
    fprintf(f, "time,clk,vin,vout\n");
}

// choose your own row format
void print_row(FILE *f) {
    // you have access to t, v[], etc.
    fprintf(f, "%g,%g,%g,%g\n",
            t,      // time
            v[5],   // clk node
            v[2],   // vin node
            v[23]   // vout node
    );
}
```

13.5.8 555 Timer

```
* 555

* add_vsrc(0, -1, 1, dc5);
V1 1 0 5

.model pmos_model pmos (kp=0.02 vt=1.5 lambda=0.001)
.model nmos_model nmos (kn=0.02 vt=1.5 lambda=0.001)

* add_pmos(2, 2, 0, 0.02f, 1.5f, 0.001f);
M1 2 2 1 pmos_model L=1 W=1

* add_nmos(3, 3, -1, 0.02f, 1.5f, 0.001f);
M2 3 3 0 nmos_model L=1 W=1

* add_res(2, 3, 1e4f);
R1 3 2 1e4

* add_pmos(4, 8, 0, 0.02f, 1.5f, 0.001f);
M3 8 4 1 pmos_model L=1 W=1

* add_pmos(4, 4, 0, 0.02f, 1.5f, 0.001f);
M4 4 4 1 pmos_model L=1 W=1

* add_nmos(6, 8, 7, 0.02f, 1.5f, 0.001f);
M5 8 6 7 nmos_model L=1 W=1

* add_nmos(5, 4, 7, 0.02f, 1.5f, 0.001f);
M5 4 5 7 nmos_model L=1 W=1

* add_nmos(3, 7,  -1, 0.02f, 1.5f, 0.001f);
M6 7 3 0 nmos_model L=1 W=1

* add_nmos(9, 9, -1, 0.02f, 1.5f, 0.001f);
M7 9 9 0 nmos_model L=1 W=1

* add_nmos(9, 13, -1, 0.02f, 1.5f, 0.001f);
```

```
M8 13 9 0 nmos_model L=1 W=1

* add_pmos(11, 9, 12, 0.02f, 1.5f, 0.001f);
M9 9 11 12 pmos_model L=1 W=1

* add_pmos(10, 13, 12, 0.02f, 1.5f, 0.001f);
M10 13 10 12 pmos_model L=1 W=1

* add_pmos(2, 12, 0, 0.02f, 1.5f, 0.001f);
M11 12 2 1 pmos_model L=1 W=1

* add_res(0, 5, 5e3f);
R2 1 5 5e3

* add_res(5, 11, 5e3f);
R3 5 11 5e3

* add_res(11, -1, 5e3f);
R4 11 0 5e3

* add_pmos(8, 14, 0, 0.02f, 1.5f, 0.001f);
M12 14 8 1 pmos_model L=1 W=1

* add_nmos(3, 14, -1, 0.02f, 1.5f, 0.001f);
M13 14 3 0 nmos_model L=1 W=1

* add_pmos(2, 15, 0, 0.02f, 1.5f, 0.001f);
M14 15 2 1 pmos_model L=1 W=1

* add_pmos(2, 16, 0, 0.02f, 1.5f, 0.001f);
M15 16 2 1 pmos_model L=1 W=1

* add_nmos(13, 15, -1, 0.02f, 1.5f, 0.001f);
M16 15 13 0 nmos_model L=1 W=1

* add_nmos(14, 16, -1, 0.02f, 1.5f, 0.001f);
M17 16 14 0 nmos_model L=1 W=1

* add_nmos(16, 15, -1, 0.02f, 1.5f, 0.001f);
M18 15 16 0 nmos_model L=1 W=1

* add_nmos(15, 16, -1, 0.02f, 1.5f, 0.001f);
M19 16 15 0 nmos_model L=1 W=1

* add_not(16, 17, 0);
M20 17 16 0 nmos_model L=1 W=1
M21 17 16 1 pmos_model L=1 W=1

* add_not(17, 18, 0);
M22 18 17 0 nmos_model L=1 W=1
M23 18 17 1 pmos_model L=1 W=1

* add_nmos(17, 19, -1, 0.02f, 1.5f, 0.001f);
M24 19 17 0 nmos_model L=1 W=1

* add_res(0, 19, 1e4f);
R5 1 19 1e4
```

```
* add_res(19, 10, 1e4f);
R6 19 10 1e4

* add_res(10, 6, 0.01f);
R7 10 6 0.01

* add_cap(6, -1, 3e-8f, time_step);
C1 6 0 3e-8

.TRAN 5e-6 5e-4

.END
```