Accelerated N-Body

CSEE 4840 - Spring 2025

Isaac Trost (wit2102), Kristian Nikolov (kdn2117), Robert Pendergrast (rlp2153),

Moises Mata (mm6155), Adib Khondoker (aak2250)

Simulating the Universe

Introduction

When two celestial objects interact through their gravitational forces, their motions are stable: it can be predicted. However, when more than three bodies are interacting, the system becomes chaotic - it is unpredictable and as far as is known, impossible to solve outright. This is known as the N-Body problem. Historically, n-body calculations have been determined through simulations, with initial simulations in the 1940's taking advantage of inverse-squared nature light propagation to compute the gravitational forces on various objects in a system. Over the course of the following decades, powerful n-body simulations have been developed by taking advantage of the increasing computational power of modern computer systems. This project makes use of the leapfrog integration algorithm to numerically determine the interaction between bodies. Custom hardware was developed on the Intel DE1-SOC FPGA to perform n-body calculations given a set of initial parameters detailing the objects' initial positions, masses and velocities. In the end, a functioning hardware accelerator was developed that could accurately calculate the motions of $N < 512$ celestial objects. Additionally, a VGA display hardware interface was developed to visually show the accelerator's calculated body positions for each returned timestep.

System Block Diagram

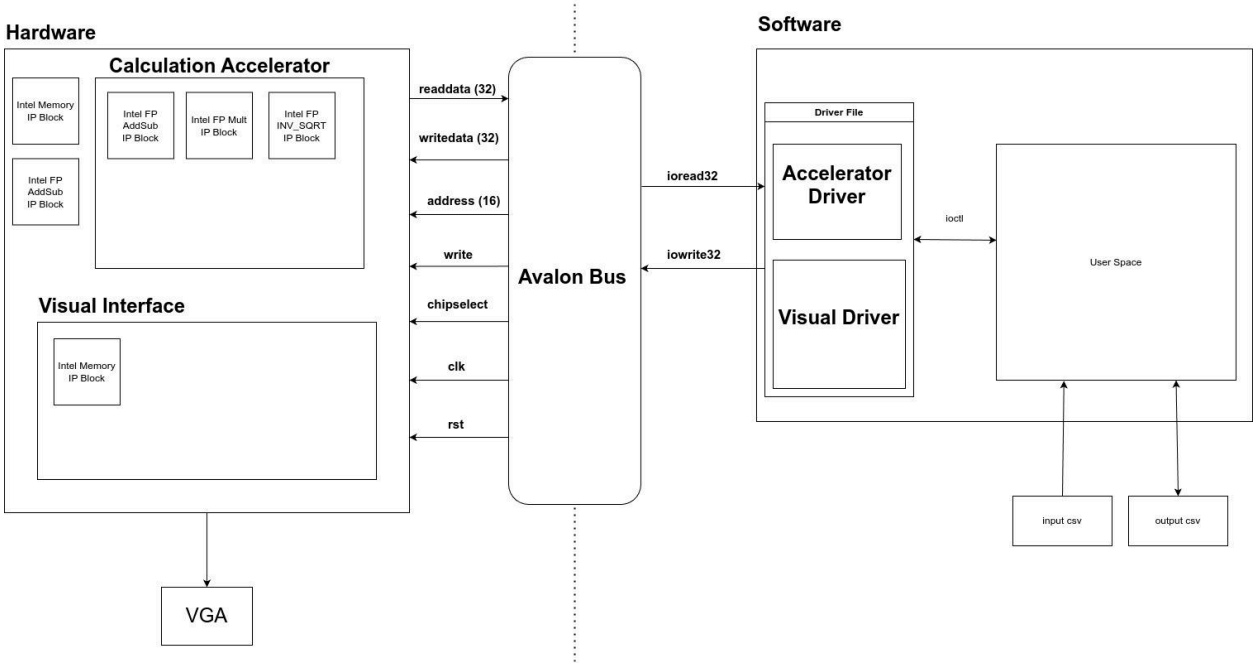


Figure 1: High-level system block diagram for the entire hardware accelerator.

On a high-level, the n-body accelerator will read in a pre-generated CSV detailing the x, y, vx, vy, and m parameters for a set number of bodies. The software interacts with the hardware through the Avalon Bus' 32 bit read/write interface. The hardware is broken up into a computational accelerator module and visual interface module. The accelerator module runs a modified leapfrog integration algorithm, while the display module controls a VGA display. The accelerator will return an output csv file, which can then be read into the display for visualization.

Algorithms

Leapfrog Integration Algorithm

The basis of this project revolves around accelerating the leapfrog integration algorithm, which is a second-order numerical integration method that is typically used when approximating physics equations, such as Newton's laws. The trajectories for each celestial body in the n-body problem are defined solely by their respective accelerations due to gravity, which is induced on a given body by each other. The gravitational force between two bodies is given by the following newtonian equation:

$$F = \frac{GMM}{R^2}$$

Therefore, the acceleration on any one single body in the N-Body system is given by:

$$A_i = \Sigma_{j=0}^{N} \frac{GM_j}{(R_i - R_j)^2}$$

Note that this equation can be further simplified when considering that the gravitational constant, G, is simply a scaling factor that is multiplied to the masses of the bodies. Therefore, to further simplify the accelerator's calculations, we assumed G to be equal to 1.0, and adjusted the inputted masses accordingly. The rough algorithm used to calculate the acceleration of a given body is defined by the following python code:

```python
def update_velocities(positions, masses, N, velecoties, half_step = False):
	for i in range(N):
		for j in range(N):
			if i != j:
				dx = positions[i,0] - positions[j,0]
				dy = positions[i,1] = positions[j,1]
				dx2 = dx**2
				dy2 = dy**2
				r2 = dx2 + dy2
```

```
r = (1/np.sqrt(r2))**3

if first:
       f = r*masses[i]/2
else:
       f = r*M[i]
vx = velocities[j,0] + dx*f
vy = velocities[j,1] + dy*f

velocities[j,0] = vx
velocities[j,1] = vy
```

In addition to determining acceleration on a given body, it is equally essential to update each body's position after each timestep. This step is done through the leapfrog integration algorithm, which is defined by the following three step process:

1. Update the velocity for half a time step using previous acceleration:
$$v_{i+1/2} = v_i + \frac{1}{2}a_i\Delta t$$
2. Update the position with the new velocity:
$$x_{i+1} = x_i + v_{i+1/2}\Delta t$$
3. Update the velocity for the second time step with updated acceleration:
$$v_{i+1} = v_{i+1/2} + \frac{1}{2}a_{i+1}\Delta t$$

However, in order to implement a pipelined hardware architecture that runs the above computations, we modified it such that for the first timestep, the accelerator computes a half-timestep update. Each preceding iteration contains a full time step update, which increases the efficiency of the process. This is equivalent to above, but we never explicitly calculate the velocities of the bodies on whole timesteps. Since we are not outputting those, this does not matter. Therefore, the implemented algorithm is as follows:

```python
#Initial half step
update_velocities(positions, masses, N, velocities, True)

t = 0

while t < iterations:
	#Full time steps
	positions += velocities * time_step
	update_velocities(positions, masses, N, velocities, False)
```

Prior to hardware implementation, the updated algorithm was tested to ensure that it produced the same results as the original leapfrog integration method.

VGA Display Algorithm

The design of the VGA Display Kernel Module facilitates the display of our N-body simulation on VGA hardware by implementing algorithms specific to the display of the simulation on VGA hardware. It revolves around the ioctl `VGA_BALL_WRITE PROPERTIES.` This ioctl copies one full simulation step worth of N-Body states from user space into kernel space. The data is copied through the `vga_ball_arg_t` struct which is defined in our kernel module header. It allows for easy access of each bodies' center, x and y, at that point in time.

One `vga_ball_arg_t` is populated in kernel space, the `draw_bodies()` function iterates through `vga_ball_arg_t`'s num bodies field, to draw a circle with the radius specified in the struct centered at the correct x and y coordinates. Drawing a circle is done by calculating a bounding box that has sides as long as the circle's diameter. Then calculating the L2 norm of each pixel with respect to the center coordinates. If the norm is less than the radius squared, the pixel is set to on.

```C/C++
int radius_squared = radius * radius;
    int x_min = x0 - radius;
    int y_min = y0 - radius;
    int x_max = x0 + radius;
    int y_max = y0 + radius;

    int x;
    int y;
    for (y = y_min; y <= y_max; y++){
        for (x = x_min; x <= x_max; x++){
            int dx = x - x0;
            int dy = y - y0;
            int d2 = dx*dx + dy*dy;
            if (d2 <= radius_squared){
           // set_pixel already has boundary checking, so we can just call it
                set_pixel(x, y, 1);
```

The last component in this structure is the set_pixel function. This function takes in coordinates, checks if those pixels are valid within the framebuffer for 640x480 VGA Displays, and will then set a specific bit within a word in our virtual framebuffer to the specified value if it is valid.

Hardware Design

N-Body Accelerator Design:

The main accelerator was implemented as a state machine with 3 states.

SW_READ_WRITE State:

The first state, SW_READ_WRITE, is when data is either being fed into the accelerator, or being read out. This is the state at the very beginning, when the GO signal is low, or when a set of calculations has been completed by the accelerator and we are waiting for handshake to finish with software. Since we only have access to 32 bit read/write operations from the Avalon bus, we must split each 64 bit read/write operation into two segments. Reading out of hardware is quite simple, as we either pass the upper or lower 32 bits of a given 64 bit value, depending on which address in memory is requested. For writing however, whenever we receive the lower bit address, we store whatever value is being written into a register, and when we receive an upper bit address, we combine the two and feed them into memory. This means that you always must pass the lower 64 bits, then the upper 64 bits for the number you are writing.

The software can only update memory positions in this state, but technically parameter registers can be updated whenever, and values can be read out whenever. That said, changing registers outside of this state will lead to undefined behavior, and values read outside of this state will not represent the values on actual timesteps.

CALC_ACCEL State:

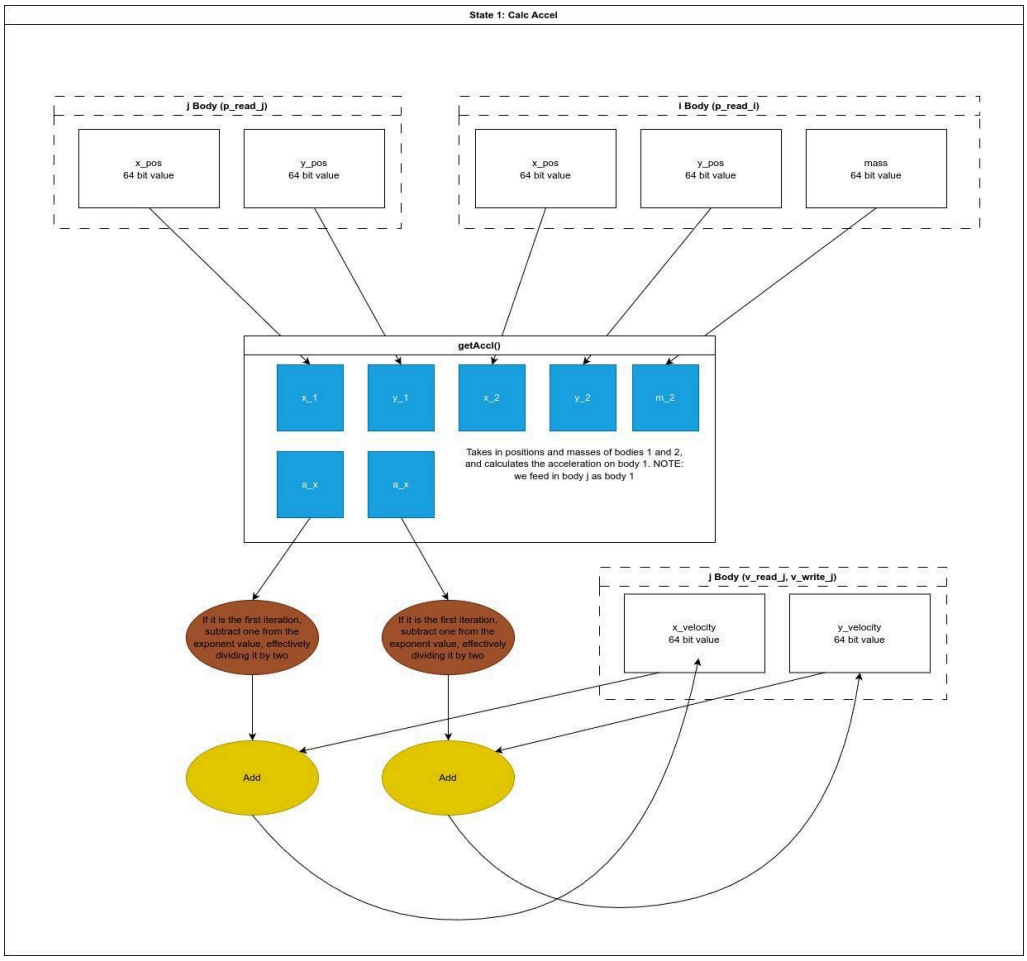


Figure 2: Calc_accel state

The second state, CALC_ACCEL, begins by looping over an outer body i, and an inner body j. We pass in the positions of both, as well as the mass of the outer body, into our acceleration calculator module, and increment the inner body j every timestep, setting it to 0 and incrementing i when it gets to the number of bodies - 1. The first block of inputs is from the jth body, and the second, which includes the mass, is from the ith body.

These get passed into the getAccel block, a module we made to calculate the acceleration on one body from another body. The operation of this block is detailed more below, but it operates with a fairly high latency.

Once valid outputs start being received from this module, after that set latency, we start pulling velocity values out of memory, and adding the acceleration and velocities to get updated velocity. Floating point accumulation is very challenging, so instead we are calculating the acceleration from one body at a time, and updating the velocity with that partial acceleration. This means, for example, if you are running a sim with 200 bodies, then during one iteration of this state, we will change the velocity 200 times. This introduces a lower limit on the number of bodies we can simulate, which is the latency of the adder. If we have fewer bodies than that, then the updated velocity wont be written back into the memory by the time we need to grab the next body. This can be simply fixed, however, by just having the driver pass in 0 mass bodies to pad us out to that required minimum.

To handle leap frogging, we have a flag set to check whether or not this is the first time acceleration values are being calculated. If so, the values are immediately divided by two (this is done by subtracting one from the exponent of the double).

Once this is done, and we have written fully updated the new velocities for all of the bodies, we move to the position update (POS_UPDATE) state.

Get Accel submodule:

Inside the acceleration calculator, values are pushed through a pipeline of calculations lasting 2 AddSub block delays + 5 Multiplier block delays + 1 Inverse Square Root block delay + 1 extra clock cycle to deal with comparing a body with itself (perturb one of the x or y so the difference is not 0, and set

the mass to 0). The operation of the module is detailed below:

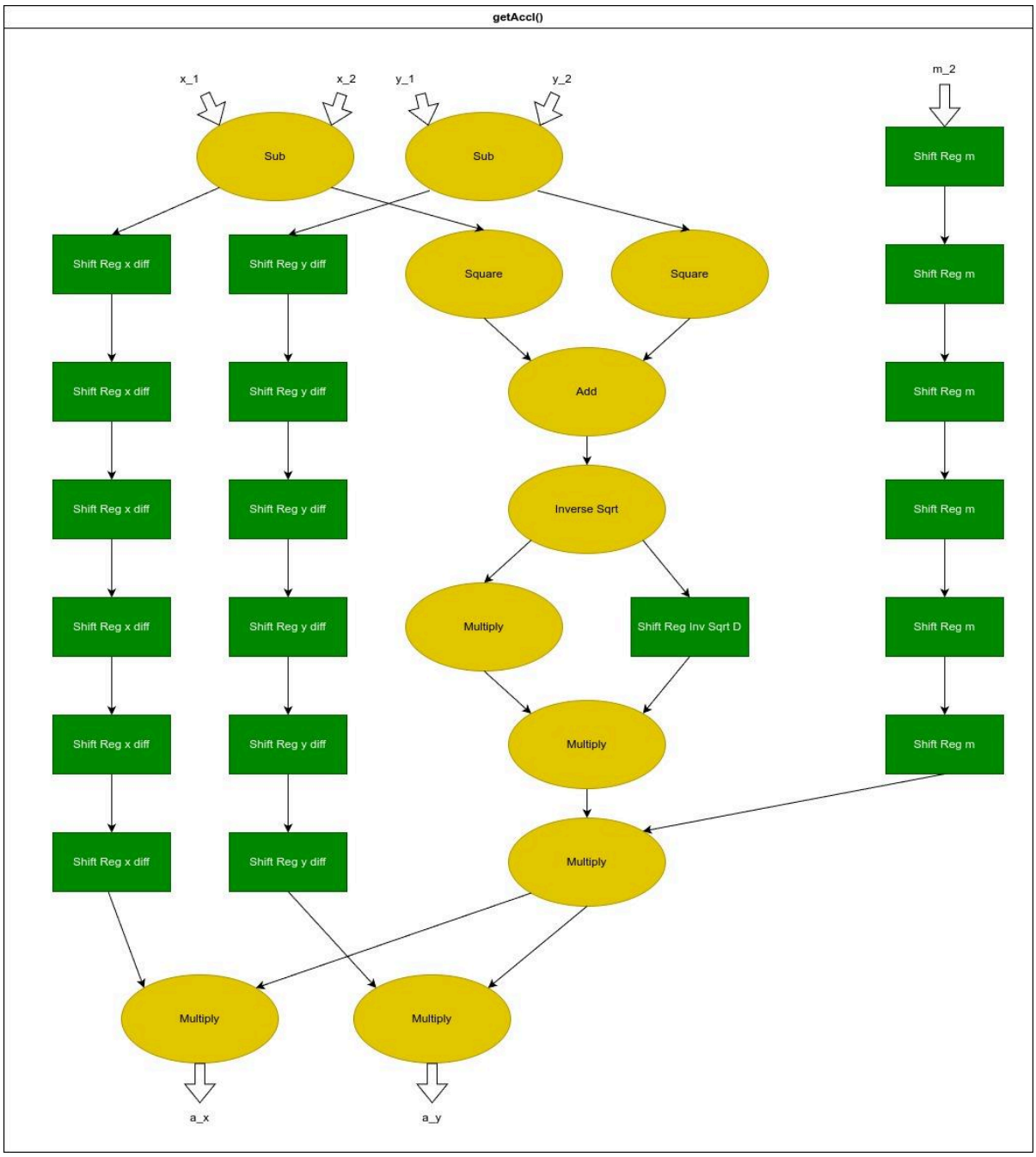


Figure 3: Acceleration calculation pipeline

We use fairly long shift registers to hold values that we will need later in the calculation, which are represented by the green boxes. The yellow circles show floating point IP cores to perform the math, and these have latencies ranging from 5 to 17.

This is calculating the length of the distance vector r = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Then calculating ax = $\frac{(x_1 - x_2)m_1}{r^3}$ and ay = $\frac{(y_1 - y_2)m_1}{r^3}$

POS_UPDATE State:

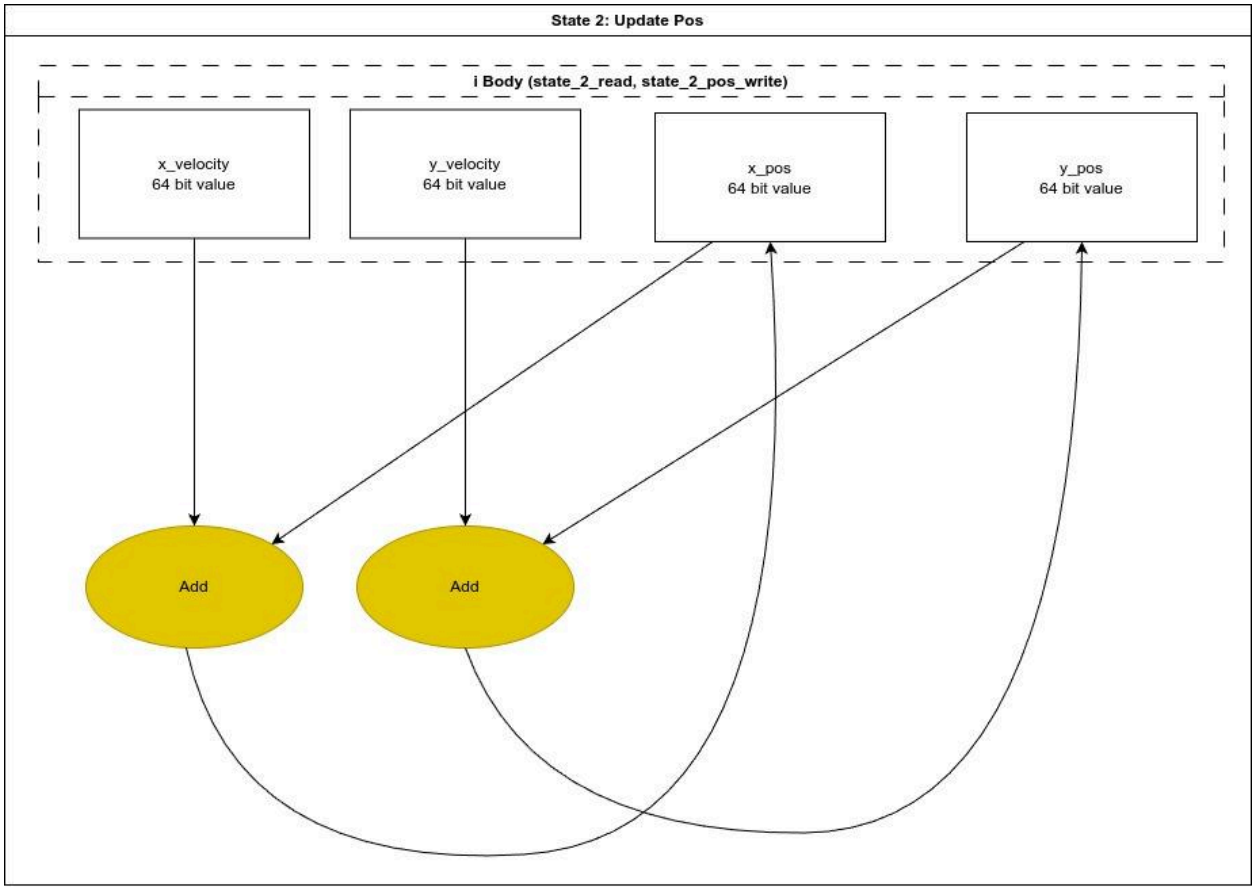


Figure 4: Position Update State

When we enter the position update state, we must use the velocities and positions to calculate the new positions. This is a fairly simple, and much shorter loop than above. We just feed the current position and velocity for the current timestep into an adder, and write the result back to the position memory. Once we have done this for everybody, then we have a choice. The handshake with software is a fairly slow part of this design, so we give the user of the accelerator the choice to not perform the handshake every timestep. The pass in a GAP parameter, which defines how many timesteps the accelerator will go before

communicating the results back. For example, if the gap is set to 100, the accelerator will loop between CALC_ACCEL and POS_UPDATE 100 times before going to SW_READ_WRITE and returning the results.

VGA Display Design:

The VGA display utilizes a memory-mapped framebuffer (38400 bytes) responsible for holding the data we read to the display. It renders the N-body simulation results by doing the equivalent of `v_count * H_WIDTH * 2 + hcount + 2`. This is effectively storing what the address of the pixel we will be on 2 cycles in the future is, so that by the time that pixel actually is being displayed, the memory will be already outputting the correct value. Bit 20 through 6 of this is used as our read memory address (`rdaddress`) which corresponds to the pixel index/ 32 **.** These 15 bit memory addresses are used to access 32 bit words in memory. We set our proxy for hcount to 0 if hcount above 1280, so that we get correct values for the first pixel on every row.

```
C/C++
logic [14:0]    rdaddress;
logic [31:0]    placecounter;
logic [31:0]   readdata;
logic [31:0]    hcount_32;
logic [31:0]    vcount_32;
logic [31:0]     vcount_x_512;
logic [31:0]     vcount_x_128;
logic [31:0]    vcountx20;
logic [10:0]    hcount;
logic [9:0]     vcount;

 assign vcount_32 = {22'b0, vcount};
 assign vcount_x_512 = vcount_32 << 8;
 assign vcount_x_128 = vcount_32 << 10;
 assign hcount_32 = (hcount > 11'd1300) ? 32'd1300 : {21'b0, hcount};
 assign vcountx20 = vcount_x_128 + vcount_x_512;
 assign placecounter = vcountx20 + hcount_32 + 32'd2;
 assign rdaddress = placecounter[20:6];
```

To ease development of this system, we created a comprehensive testbench and simulated the device in ModelSim. It simulates a 50MHz clock frequency and writes a checkerboard pattern to the framebuffer. This pattern was easy to see in the ModelSim wave form, as a correct output was a constant, periodic switch between `VGA_R`, `VGA_G`, and `VGA_B` being set to 8'hff and 8'h00.

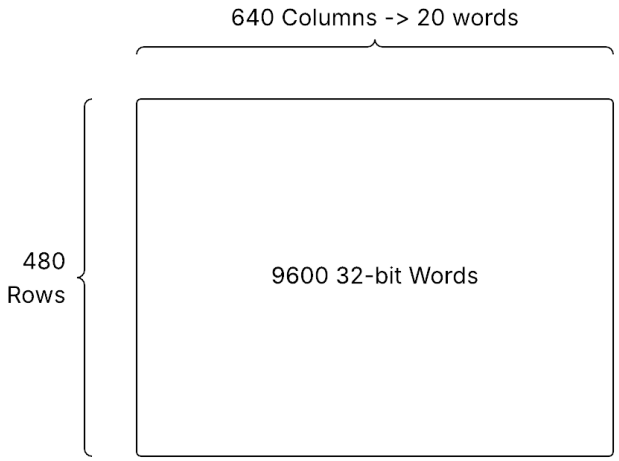


Figure 5: VGA display frame buffer

Simulator Design:

We simulated all of the major components of the system. This was more challenging than in other labs because of the IP cores. The rams were very easy, but each one of the FP cores required a tcl script to be run with certain global variables set first, and the documentation from intel was hard to find, and did not detail how do do that for more than one core at the same time, as they mostly seemed to be about testing the functionality of the core itself. Once we were able to solve that problem, however, we could begin writing testbenches for different parts of the design.

We had a simple testbench for the GetAccel block, which confirmed that it was doing the math correctly by comparing with known correct values from software we wrote.

The testbench for the full accelerator was a bit more tricky, as we had to simulate what some reasonable software input may be from the axi-lite bus. We could define all of the addresses for different parts of the interface, and write values, and confirm that everything is working. By checking internal signals, we were able to see exactly when the process was going wrong and quickly fix and rerun the sim, as the compile times for modelsim are much much lower than synthesis.

Finally, the testbench for the display was similar to that for the accelerator, but simpler since the display module does not write anything back out to the user.

Resource Budgets

Our main concerns with resource budgets were the usage of memory for data storage, as well as DSP block usage. Our initial plan to implement acceleration via parallelization required

significantly more DSP blocks. The decision to implement pipelining reduced our 64 bit Floating Point IP Block usage to 7 Multipliers , 5 AddSub, 1 Inverse Square Root. The estimated resource usage of these were 2479 LUTs for each AddSub block, 8 Multiplies and 521 LUTs for each Multiplier block, and finally 16 Multiplies, 1023 LUTs and 111616 Memory bits for each Inverse Square Root block.

For memory, in the accelerator itself we used 5 Memory blocks, each of which had 9 bit addresses (for 512 bodies) with each address containing 64 bits of value. Additionally the frame buffer required a memory block with 15 bit addresses with 32 bits of data.

When compiling we saw that we used far less DSP blocks than expected, likely due to optimizations by Quartus.

Flow Status	Successful - Wed May 14 09:15:27 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)	7,449 / 32,070 (23 %)
Total registers	10872
Total pins	362 / 457 (79 %)
Total virtual pins	0
Total block memory bits	897,774 / 4,065,280 (22 %)
Total DSP Blocks	37 / 87 (43 %)
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 %)

Hardware-Software Interface

Accelerator Interface:

The accelerator interface makes use of the Avalon Bus' 32 bit-wide read and write data ports to implement a 32 bit memory-mapped interface using 16 bit address spaces. These addresses are used to access the simulation's initialization parameters, as well as to read and write from the body positions.

Simulation Parameters Register Map:

Address	15-9b	8-0b
N_ADDR	0x40	XXXXXXXXX
GAP_ADDR	0x42	XXXXXXXXX

The hardware accelerator uses 64-bit floating point values to perform the kinematic calculators for the N bodies. Since the Avalon Bus only supports a maximum read and write width of 32 bits, it is necessary that each body parameter (X, Y, VX, VY, M) be split up into its upper and low 32 bits when being written and read from hardware.

Write Body Parameters Memory Map:

Address	15-9b	8-0b
X_LOW	0x44	Body_number (0-510)
X_UPPER	0x45	Body_number (0-510)
Y_LOW	0x46	Body_number (0-510)
Y_UPPER	0x47	Body_number (0-510)
M_LOW	0x48	Body_number (0-510)

M_UPPER	0x49	Body_number (0-510)
VX_LOW	0x4a	Body_number (0-510)
VX_UPPER	0x4b	Body_number (0-510)
VY_LOW	0x4c	Body_number (0-510)
VY_UPPER	0x4d	Body_number (0-510)

A similar memory mapping pattern is prescribed to the read parameters, which are accessed as the software reads the output X and Y positions from the accelerator module:

Address	**15-9b**	**8-0b**
READ_X_LOW	0x51	body_number
READ_X_UPPER	0x52	body_number
READ_Y_LOW	0x53	body_number
READ_Y_UPPER	0x54	body_number

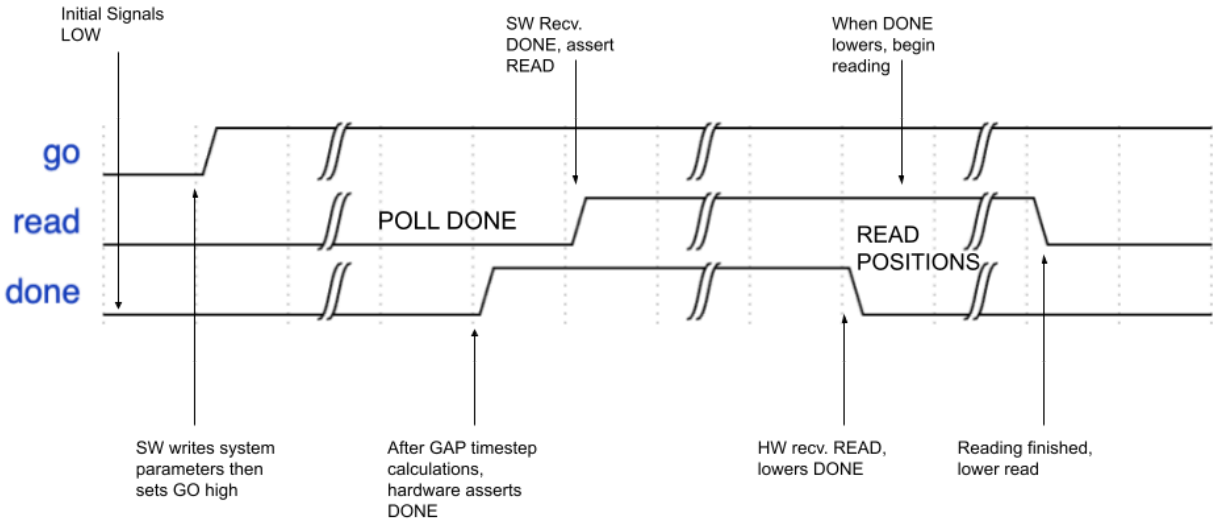
For both reading and writing body parameters, the lower 9 bits of the 16 bit memory address correspond to the body number, allowing for each of the 5 parameters for each body to be easily accessed. The upper 7 bits determine the parameter being stored at that address.

Accelerator Control Software Implementation

An additional three memory locations are used to handle the control sequencing and handshake procedures between the hardware and the software:

Address	**15-9b**	**8-0b**
GO_ADDR	0x40	XXXXXXXXX

READ_ADDR	0x41	XXXXXXXXX
DONE_ADDR	0x50	XXXXXXXXX



These three signals determine the handshake used to determine when the software should read the accelerator's updated position information. This processes is given by the following timing diagram:

VGA Display Interface:

Display Interface:

The display interface also uses the Avalon Bus' 32 bit-wide read and write ports to implement a 32 bit memory-mapped interface with 15 bit address spaces. These address spaces range from the `base address` to `base address + (9599 << 2)`. A 32 bit write to these addresses will write a 32 bit word to the word it points to.

Memory space with corresponding addresses — 9600 32 bit words with 15 bit addresses

Base	**Base + (1<<2)**	**…**	**Base + (9598 << 2)**	**Base + (9599 << 1)**

WORD 1	WORD 2	…	WORD 9599	WORD 9600

Display Software Control Interface:

The control software also writes to these memory addresses, which is directly displayed on the VGA monitor by the Display Module. As explained above in the VGA algorithm section, we do many calculations to determine which pixels need to be set to on versus which need to be set to off. Since it would be inefficient to modify single bits and write to memory for every single bit modified, we hold a virtual framebuffer within our kernel module device struct that is modified by our draw_bodies() function. We are able to calculate the word and bit offset through our x and y coordinates, and use bitwise operations to turn a single bit to one or zero, which represents turning a single pixel to white or black.

```C/C++
static inline void set_pixel(unsigned short x, unsigned short y, int value)
{
    if (x >= DISPLAY_WIDTH || y >= DISPLAY_HEIGHT || x < 0 || y < 0){
        return;
    }

    int index = (y*WORDS_PER_ROW) + x/32;
    if (value) {
        dev.framebuffer[index] |= 1 << (x % 32);
    } else{
        dev.framebuffer[index] &= ~(1 << (x % 32));
    }
}
```

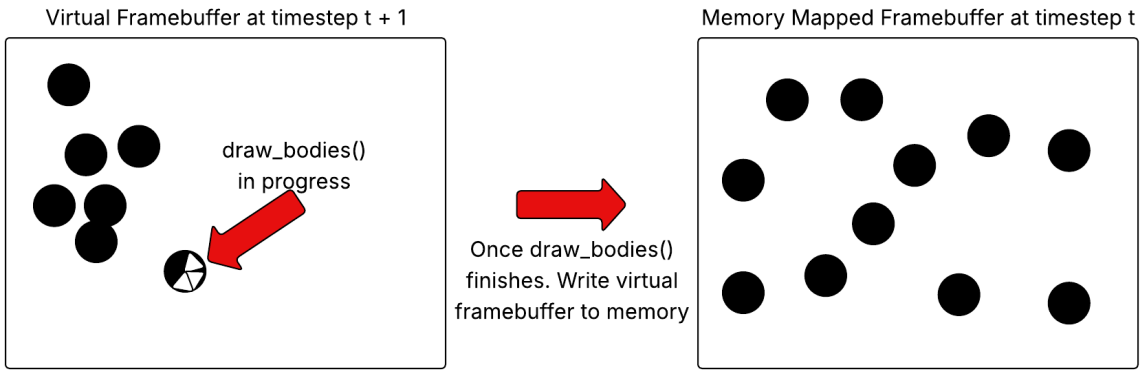


Figure 7: Frame buffer writing visualization

Contribution & Lessons Learned

Isaac Trost - I played a major role designing and implementing the accelerator itself, as well as helping with debugging on the display, accelerator, and software. I also, along with Kris, handled actually working with Quartus, Modelsim, and PlatformDesigner, with all the joy that that software provides. The most important lesson learned is to frequently backup the full state of the Quartus and its various dependencies, because sometimes things like to blow up, compilations start failing for no discernible reason, and even reverting changes to user defined files doesn't fix it. When this happens, it is much simpler to go from a known working backup than to try to fix the problem. A second tidbit would be to ensure you have a really good simulator and testbench. The testbench for the accelerator module absolutely saved us, allowing us to quickly see all the internal signals of the accelerator and figure out all the places our first iteration was failing. Using a similarly detailed testbench for the display would have made our lives much simpler.

Kristian Nikolov - The majority of my work was in designing the hardware in system verilog, getting compilations and synthesis working in Quartus and QuestaSim, and debugging the hardware using test benches. I did have the chance to pair program and debug some of the software portion, but that was mostly me helping others as they worked.

Working on large projects is always a great opportunity to learn new skills, and this was no exception. Although I had some experience with SystemVerilog, actually working on an FPGA solidified what I already knew, and pushed me to learn what I didn’t. Debugging is something I’m used to, but trying to pin-point exactly what’s going wrong in a somewhat complex state machine using the waveforms is simultaneously difficult but gratifying when you

finally hunt down whatever loop is just slightly off. Planning all your variables and going through example timestep progressions before implementing complicated loops in hardware is incredibly important.

I do wish I had more time to work on the software side of things, as the integration of everything seems interesting to me, but I was happy with the work I did.

Robert Pendergrast - I took ownership of implementing the software for userspace program and device driver, as well as implementing the software interface for the simulation module. I also developed simulations used for verifying the accelerator's output, and helped debug the accelerator hardware. The most important lesson that I learned was to take an iterative approach to develop the drivers by breaking everything up into verifiable steps before integrating an entire system. My advice for future projects follows suit: develop the software side and hardware together to really understand how they work together. It is also important to develop everything modularly, verifying each individual piece before building the next one. Also - save your files! We ran into a diabolical error where we didn't save the working binaries, overwrote them with malfunctioning ones, and then spent a good five hours trying to figure out how to fix it again.

Moises Mata - I was involved in the software interface development for both our Nbody Accelerator and VGA display hardware modules along with Robert, and was especially involved in the kernel and user space software that interacted with the VGA module. Some features I implemented for the VGA display are: the ability to read in simulations through previously generated CSV files, the logic to convert coordinates from our accelerator into acceptable VGA coordinates, and the implementation of relative sizing of each body according to their mass. I

also helped debug the VGA display verilog module (to my best abilities as a software oriented person), and wrote the checkerboard display testbench that helped verify that our display.sv implementation was indeed functional. Some advice I'd have for future projects is to have an extremely good idea of the Hardware Software split —- even before you write any code. I feel that a common pitfall with embedded system development is the difficulty in writing software without the associated hardware. Make sure that there is work to do while the hardware is in development if you are software oriented (or better yet, learn and help development!) — this can be great for team efficiency, especially in hardware heavy projects like the N-Body Sim.

Adib Khondoker - I was involved in the integration and debugging of the top module and the display.sv module. The majority of my work was helping debug code and quartus issues - researching solutions. I specifically aided in debugging the classic "off-by one error" we encountered in our VGA display implementation which required multiple code revisions. The most important lesson I learned is how critical low-level implementation details are to the correctness of the system. Even a minor misalignment or timing miscalculation causes significant visible errors in the timing diagram. My advice for future projects would be to get modelsim up and running as soon as possible so that you can simulate your verilog code. As well as having a thorough datapath diagram so that you understand exactly how everything is connected. It makes writing the verilog code a lot easier when you can visualize the signals and how data is flowing through your modules.

References

Camphuijsen et Al. “Visualizing the Few Body Problem.” 6 Nov. 2015
Gupta et Al. “FPGA Accelerated N-body Simulations.” 14 Oct. 2023
Quinn et Al. “Time stepping N-body simulations.” 3 Oct. 1997
“Leapfrog Integration” *Wikipedia, The Free Encyclopedia*, Wikimedia Foundation, 15 April 2025, https://en.wikipedia.org/wiki/Leapfrog_integration.
Intel, “Intel FP_FUCNTIONS”
https://cdrdv2-public.intel.com/666310/ug_fxp_mf-683339-666310.pdf
https://github.com/aryabhatta-dey/n-body

Code

nbody.c

C/C++

```
/*
 * Userspace program that communicates with the vga_ball device driver
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

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

#define MAXCHAR 1000

int nbody_fd;

int high = 0xFFFFFFFF;
int low = 0x00000000;

// ------------------------------------------------------
// Setting the Body for a Single Body
// ------------------------------------------------------
void set_body(double x, double y, double xv, double yv, double m, int n){
  body_t vla;
  vla.x = x;
  vla.y = y;
  vla.vx = xv;
  vla.vy = yv;
  vla.m = m;
  vla.n = n;
  //fprintf(stderr, "Setting Body %d: X: %f, Y: %f, M: %f\n",n,x,y,m);
```

```
  if(ioctl(nbody_fd, SET_BODY, &vla)){
       perror("ioctl(SET_BODY) failed");
       return;
  }
}

// -----------------------------------------------------
// Set the Simulation Paraameters - Called Once
// -----------------------------------------------------
void set_simulation_parameters(int N, int output_step){ //dt is always 2
  nbody_sim_config_t vla;
  vla.N = N;
  vla.gap = output_step;
  if(ioctl(nbody_fd, NBODY_SET_SIM_PARAMETERS, &vla)){
       perror("ioctl(NBODY_SET_SIM_PARAMETERS) failed");
       return;
  }
}

// -----------------------------------------------------
// Set the go signal high to start thes sim
// -----------------------------------------------------
void set_go(int go){
  if(ioctl(nbody_fd, WRITE_GO, &go)){
       perror("ioctl(WRITE_GO) failed");
       return;
  }
}

// -----------------------------------------------------
// Continuously poll the read signal
// -----------------------------------------------------
int poll_done(){

  int done;
  if (ioctl(nbody_fd, READ_DONE, &done)) {
       perror("ioctl(READ_DONE) failed");
       return -1;
  }
  //fprintf(stderr, "Polling: %d \n", done);
  if(done > 0){
       return 1;
  }
  return 0;
```

```
}

// -----------------------------------------------------
// Read all the positions into the struct
// -----------------------------------------------------
all_positions_t read_positions(int N){
  //ioctl goes here


  all_positions_t positions;

  for(int i = 0; i < N; i++){
       //fprintf(stderr, "Reading Body Number %d\n", i);
       body_pos_t vla;
       vla.n = i;
       if (ioctl(nbody_fd, READ_POSITIONS, &vla)){
       perror("ioctl(READ_POSITION) failed\n");
       }
       //fprintf(stderr, "Body %d Position Read: X:%f, Y:%f \n", N,vla.x,vla.y);
       positions.bodies[i] = vla;
  }
  return positions;
}

// -----------------------------------------------------
// Set the read signal
// -----------------------------------------------------
void set_read(int read){
  if(ioctl(nbody_fd, WRITE_READ, &read)){
       perror("ioctl(WRITE_READ) failed\n");
       return;
  }
}

// -----------------------------------------------------
// Function to read the input CSV
// -----------------------------------------------------
double* get_initial_state(char* filename, int N){
//Allocate Space for a the Body Paremeters a
  double* initial_state = (double*)calloc(N * 5 , sizeof(double));
  if (!initial_state) {
       fprintf(stderr, "Memory allocation failed\n");
       return NULL;
  }
```

```
  //fprintf(stderr, "Allocated %d bytes for initial state\n", N * 5 *
sizeof(double));

  //Open the file and read it
  FILE* file = fopen(filename, "r");
  if(!file){
       fprintf(stderr, "Could not open file %s\n", filename);
       return NULL;
  }
  char row[MAXCHAR];
  int i = 0;
  int flag = 0;
  //fprintf(stderr, "Reading file %s\n", filename);
  //Go through file and save all the data
  while(fgets(row, MAXCHAR, file)){
       char* token = strtok(row,",");
       while(token != NULL){
       initial_state[i] = atof(token);
       token = strtok(NULL,",");
       i++;
       if(i >= (N * 5)){
       flag = 1;
       break;
       }
       }
       if(flag){
       break;
       }
  }

  //fprintf(stderr, "Do we get here???? %lf %lf %lf\n", initial_state[0],
initial_state[1], initial_state[2]);

  //Close final and return pointer to the initial parameters
  fclose(file);
  //fprintf(stderr, "we closed the file\n");
  return initial_state;
}

// ------------------------------------------------------
// Main function - Runs Everything
// ------------------------------------------------------
int main(int argc, char** argv){
  //Check to make sure that the
```

```
  if (argc > 1) {
       fprintf(stderr, "Usage: ./nbody \n");
       return -1;
  }


  //Begin the userspace program
  static const char filename[] = "/dev/nbody_display";

  if ( (nbody_fd = open(filename, O_RDWR)) == -1) {
       fprintf(stderr, "could not open %s\n", filename);
       return -1;
  }

  set_go(low);
  set_read(low);

  //Check to make sure that the
  char one[] = " _    _     ____   ___  ______   __";
  char two[] = "| \\ | |   | __ ) / _ \\|  _ \\ \\ / /";
  char three[] = "|  \\| |_____|  _ \\| | | | | \\ V /";
  char four[] = "| |\\  |_____| |_) | |_| | |_| || | ";
  char five[] = "|_| \\_|  |____/ \\___/|____/ |_|  ";
  printf("%s\n",one);
  printf("%s\n",two);
  printf("%s\n",three);
  printf("%s\n",four);
  printf("%s\n",five);

  char thirty_two[] = "32body_input.csv";
  char sixty_four[] = "64body_input.csv";
  char one_two_eight[] = "128body_input.csv";
  char five_one_one[] = "511body_input.csv";
  char solar[] = "solar_system.csv";
  char initial_test[] = "input.csv";

  printf("\nWelcome to the N-Body Simulation Final Project!\n\n");
  printf("Please select a setting for how many bodies you would like to
simulate:\n 1 - 32 Bodies \n 2 - 64 Bodies \n 3 - 128 Bodies \n 4 - 511
Bodies\n 5 - Solar System\n");

  int num;
  char* selected_sim;
```

```
scanf("%d",&num);

int N;

switch (num){
     case 1:
     printf("You selected 32 Bodies!\n");
     selected_sim = thirty_two;
     N = 32;
     break;

     case 2:
     printf("You selected 64 Bodies!\n");
     selected_sim = sixty_four;
     N = 64;
     break;

     case 3:
     printf("You selected 128 Bodies!\n");
     selected_sim = one_two_eight;
     N = 128;
     break;

     case 4:
     printf("You selected 511 Bodies!\n");
     selected_sim = five_one_one;
     N = 511;
     break;

     case 5:
     printf("You selected the solar system!\n");
     selected_sim = solar;
     N = 14;
     break;

     case 6:
     printf("You selected secret test!\n");
     selected_sim = initial_test;
     N = 28;
     break;


     default:
     printf("Bad input! Please run the program again.\n");
```

```
     return -1;
}

printf("Set Gap Size:\n");

int output_step;
scanf("%d",&output_step);
if(output_step < 1){
     perror("Gap Size Must Be >= 1!!\n"); //For testing set to 6
     return -1;
}

printf("Number of Iterations:\n"); //Set to 12

int time_steps;
scanf("%d",&time_steps);
if(time_steps < 1){
     perror("Number of Iterations Must Be >= 1!!\n");
     return -1;
}

printf("N-Body Userspace program started\n");

// Read in Initial N-Body State FROM CSV File

struct timespec start, end, this;
clock_gettime(CLOCK_MONOTONIC, &start);

double* zeros = (double*)calloc(N * 5, sizeof(double));
for(int i = 0; i < N; i++){
     set_body(zeros[5*i + 0], //x
            zeros[5*i + 1], //y
            zeros[5*i + 2], //vx
            zeros[5*i + 3], //vy
            zeros[5*i + 4], //m
            i); //body number
}



double* initial_state = get_initial_state(selected_sim, N);

fprintf(stderr, "Initial Bodies Parameters Read In\n");
```

```
  //Create an array that saves all the timesteps
  all_positions_t* position_history = malloc(time_steps *
sizeof(all_positions_t));
  if (!position_history) {
       fprintf(stderr, "Failed to allocate memory for position history\n");
       return -1;
  }

  //Then set the initial parameters for the simulation
  set_simulation_parameters(N,output_step);

  // The initial parameters are read in - Send them to the driver

  for(int i = 0; i < N; i++){
       set_body(initial_state[5*i + 0], //x
              initial_state[5*i + 1], //y
              initial_state[5*i + 2], //vx
              initial_state[5*i + 3], //vy
              initial_state[5*i + 4], //m
              i); //body number
  }

  clock_gettime(CLOCK_MONOTONIC, &end);
  clock_gettime(CLOCK_MONOTONIC, &this);
  double elapsed_time = (end.tv_sec - start.tv_sec) +
                        (end.tv_nsec - start.tv_nsec) / 1e9;
  printf("Initial write time: %f seconds\n", elapsed_time);
  // Print initial positions to stdout


  //Send the go signal
  set_go(high);

  //fprintf(stderr, "Go signal set high!\n");

  //Do the looping - implemented as some sort of silly state machine
  int t = 0;
  while(t < time_steps){
       if(t % 10 == 0){
       //printf("Iteration %d complete!\n", t);
       }

       //fprintf(stderr, "Timestep %d Beginning:\n", t);
       //Do Polling
```

```
        int read = 0;

        /**/
        while(!read){
        //Wait for the poll signal
        //fprintf(stderr, "Polling...\n");
        if(poll_done()){
        //fprintf(stderr, "Received Done!\n");
        read = 1;
        set_read(high);
        break;
        }
        }
        //Read the positions from the driver

        //fprintf(stderr, "Read Set To High\n");
        //usleep(100000);

        while(1){
        if(poll_done() == 0){
        break;
        }
        }

        //fprintf(stderr, "Done is low again!\n");

        position_history[t] = read_positions(N);

        //fprintf(stderr, "How did we get here?\n");

        //Reading is finished, set read to low!
        set_read(low);
        // Calculate and print the elapsed time for the current timestep
        clock_gettime(CLOCK_MONOTONIC, &end);
        double timestep_elapsed_time = (end.tv_sec - start.tv_sec) +
                    (end.tv_nsec - start.tv_nsec) / 1e9;

        if ((end.tv_sec - this.tv_sec) + (end.tv_nsec - this.tv_nsec) / 1e9 > 1)
{
        printf("Total elapsed time: %f seconds, Iteration %d\n",
timestep_elapsed_time, t);
        clock_gettime(CLOCK_MONOTONIC, &this);
        }
```

```
     //Increment Time Thing
     t += 1;
}
set_go(low);
// Calculate and print the total simulation time

clock_gettime(CLOCK_MONOTONIC, &end);
double total_simulation_time = (end.tv_sec - start.tv_sec) +
                               (end.tv_nsec - start.tv_nsec) / 1e9;
printf("Total simulation time: %f seconds\n", total_simulation_time);

// Write all data to a CSV file
FILE* output = fopen("nbody_results.csv", "w");
if (output) {
     //Header with timestep, body0_x, body0_y, body0_m, body1_x, body1_y, ...
     fprintf(output, "timestep");
     for (int i = 0; i < N; i++) {
     fprintf(output, ",body%d_x,body%d_y,body%d_m", i, i, i);
     }

     fprintf(output, "\n");

     for (int t = 0; t < time_steps; t++) {
     fprintf(output, "%d", t);
     for (int i = 0; i < N; i++) {
            fprintf(output, ",%f,%f,%f",
                   position_history[t].bodies[i].x,
                   position_history[t].bodies[i].y,
            initial_state[5*i + 4]); // mass
     }
     fprintf(output, "\n");
     }
     fclose(output);
     fprintf(stderr, "Results saved to nbody_results.csv\n");
} else {
     fprintf(stderr, "Failed to open output file\n");
}

// Free the allocated memory to prevent memory leaks
free(zeros);
free(initial_state);
free(position_history);

printf("N-Body Userspace program terminating\n");
```

```
  printf("Displaying simulation!\n");
  return 0;
}
```

nbody_display_driver.c

```
C/C++
/*
       * nbody_display_driver.c
       *
       * This file implements a Linux kernel driver for the N-body simulation
display.
       * It provides an interface for user-space applications to interact with
the
       * hardware and control the display.
       *
*/
#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 "nbody_display_driver.h"

#define DRIVER_NAME "nbody_display"

/* Device registers */

#define GO_ADDR(base) (base) + (0x40 << 11)
#define READ_ADDR(base) (base) + (0x41 << 11)
#define N_ADDR(base) (base) + (0x42 << 11)
```

```
#define GAP_ADDR(base) (base) + (0x43 << 11)

/* Memory */

#define X_ADDR_LOW(base, body) (base) + ((body<<2) + (0x44 << 11))
#define X_ADDR_HIGH(base, body) (base) + ((body<<2) + (0x45 << 11))

#define Y_ADDR_LOW(base, body) (base) + ((body<<2) + (0x46 << 11))
#define Y_ADDR_HIGH(base, body) (base) + ((body<<2) + (0x47 << 11))

#define M_ADDR_LOW(base, body) (base) + ((body<<2) + (0x48 << 11))
#define M_ADDR_HIGH(base, body) (base) + ((body<<2) + (0x49 << 11))

#define VX_ADDR_LOW(base, body) (base) + ((body<<2) + (0x4a << 11))
#define VX_ADDR_HIGH(base, body) (base) + ((body<<2) + (0x4b << 11))

#define VY_ADDR_LOW(base, body) (base) + ((body<<2) + (0x4c << 11))
#define VY_ADDR_HIGH(base, body) (base) + ((body<<2) + (0x4d << 11))

#define DONE_ADDR(base) (base) + (0x50 << 11)

#define READX_ADDR_LOW(base, body) (base) + ((0x51 << 11) + ((body)<<2))
#define READX_ADDR_HIGH(base, body) (base) + ((0x52 << 11) + ((body)<<2))

#define READY_ADDR_LOW(base, body) (base) + ((0x53 << 11) + ((body)<<2))
#define READY_ADDR_HIGH(base, body) (base) + ((0x54 << 11) + ((body)<<2))



/* Combined device information */
struct nbody_display_dev {
       // Hardware resources
       struct resource res;
       void __iomem *virtbase;

       // NBody simulation data
       body_t body_parameters;
       nbody_sim_config_t sim_config;
       int go;
       int done;
       int read;

       // Display data
       unsigned int framebuffer[FRAMEBUFFER_SIZE];
```

```
        vga_ball_arg_t vga_ball_arg;
} dev;

/* ========== NBody Simulation Functions ========== */

static void write_body(body_t * body_parameters) {
        int i = (int) body_parameters->n;
        int x_bits[2];
        int y_bits[2];
        int m_bits[2];
        int vx_bits[2];
        int vy_bits[2];

        memcpy(&x_bits, &body_parameters->x, sizeof(uint64_t));
        memcpy(&y_bits, &body_parameters->y, sizeof(uint64_t));
        memcpy(&m_bits, &body_parameters->m, sizeof(uint64_t));
        memcpy(&vx_bits, &body_parameters->vx, sizeof(uint64_t));
        memcpy(&vy_bits, &body_parameters->vy, sizeof(uint64_t));

        // Perform actual writes
        iowrite32(x_bits[0], X_ADDR_LOW(dev.virtbase, i));
        iowrite32(x_bits[1], X_ADDR_HIGH(dev.virtbase, i));

        iowrite32(y_bits[0], Y_ADDR_LOW(dev.virtbase, i));
        iowrite32(y_bits[1], Y_ADDR_HIGH(dev.virtbase, i));

        iowrite32(m_bits[0], M_ADDR_LOW(dev.virtbase, i));
        iowrite32(m_bits[1], M_ADDR_HIGH(dev.virtbase, i));

        iowrite32(vx_bits[0], VX_ADDR_LOW(dev.virtbase, i));
        iowrite32(vx_bits[1], VX_ADDR_HIGH(dev.virtbase, i));

        iowrite32(vy_bits[0], VY_ADDR_LOW(dev.virtbase, i));
        iowrite32(vy_bits[1], VY_ADDR_HIGH(dev.virtbase, i));
}

static void write_simulation_parameters(nbody_sim_config_t *parameters) {
        iowrite32(parameters->N, N_ADDR(dev.virtbase));
        iowrite32(parameters->gap, GAP_ADDR(dev.virtbase));
        dev.sim_config = *parameters;
}

static void read_position(body_pos_t *body) {
        int i = body->n;
```

```
        int x_bits[2];
        int y_bits[2];

        x_bits[0] = ioread32(READX_ADDR_LOW(dev.virtbase, i));
        x_bits[1] = ioread32(READX_ADDR_HIGH(dev.virtbase, i));

        y_bits[0] = ioread32(READY_ADDR_LOW(dev.virtbase, i));
        y_bits[1] = ioread32(READY_ADDR_HIGH(dev.virtbase, i));

        memcpy(&body->x, &x_bits, sizeof(uint64_t));
        memcpy(&body->y, &y_bits, sizeof(uint64_t));
}

static void write_go(int go) {
        iowrite32(go, GO_ADDR(dev.virtbase));
        dev.go = go;
}

static void write_read(int read) {
        iowrite32(read, READ_ADDR(dev.virtbase));
        dev.read = read;
}

static void read_done(int *status) {
        *status = ioread32(DONE_ADDR(dev.virtbase));
}

/* ========== Display/VGA Functions ========== */

/*
 * Set a pixel in the framebuffer
 * x, y: coordinates (0-639, 0-479)
 */
static inline void set_pixel(unsigned short x, unsigned short y, int value)
{
        if (x >= DISPLAY_WIDTH || y >= DISPLAY_HEIGHT || x < 0 || y < 0){
        return;
        }

        int index = (y*WORDS_PER_ROW) + x/32;
        if (value) {
        dev.framebuffer[index] |= 1 << (x % 32);
        } else{
        dev.framebuffer[index] &= ~(1 << (x % 32));
```

```
        }
}

/*
 * Clear the entire framebuffer
 */
static void clear_framebuffer(void)
{
        int i;
        for (i = 0; i < FRAMEBUFFER_SIZE; i++) {
        dev.framebuffer[i] = 0;
        iowrite32(0, dev.virtbase + (i << 2));
        }
}

/*
 * Fill the entire framebuffer (turn all pixels on)
 */
static void fill_framebuffer(void)
{
        int i;
        printk(KERN_INFO "vga_ball: Filling entire framebuffer (all pixels
on)\n");

        for (i = 0; i < FRAMEBUFFER_SIZE; i++) {
        dev.framebuffer[i] = 0xFFFFFFFF;  // All bits set to 1
        iowrite32(dev.framebuffer[i], dev.virtbase + (i << 2));
        }

        printk(KERN_INFO "vga_ball: Framebuffer filled with all pixels on\n");
}

//Right now the default radius is 5
static void draw_circle(unsigned short x0, unsigned short y0, unsigned short
radius) {
        // Validate inputs
        if (radius <= 0) {  // Set reasonable bounds
        radius = 0;
        }else if (radius > 20) {
        radius = 20;
        }

        int radius_squared = radius * radius;
        int x_min = x0 - radius;
```

```
        int y_min = y0 - radius;
        int x_max = x0 + radius;
        int y_max = y0 + radius;

        int x;
        int y;
        for (y = y_min; y <= y_max; y++){
        for (x = x_min; x <= x_max; x++){
              int dx = x - x0;
              int dy = y - y0;
              int d2 = dx*dx + dy*dy;
              if (d2 <= radius_squared){
              // set_pixel already has boundary checking, so we can just call it
              set_pixel(x, y, 1);
              }
        }
        }
}

static void draw_bodies(void)
{
        int i;
        printk(KERN_INFO "vga_ball: Drawing the Bodies\n");

        //clear virtual framebuffer
        for (i = 0; i < FRAMEBUFFER_SIZE; i++) {
        dev.framebuffer[i] = 0;
        }

        for (i = 0; i < dev.vga_ball_arg.num_bodies; i++) {
        draw_circle(
              dev.vga_ball_arg.bodies[i].x,
              dev.vga_ball_arg.bodies[i].y,
              dev.vga_ball_arg.bodies[i].radius);
        }

        for (i = 0; i < FRAMEBUFFER_SIZE; i++) {
        iowrite32(dev.framebuffer[i], dev.virtbase + (i << 2));
        }

        printk(KERN_INFO "vga_ball: Bodies Drawn\n");
}
```

```
static void draw_checkerboard(void)
{
	int i, j;
	for (i = 0; i < DISPLAY_HEIGHT; i++) {
	for (j = 0; j < DISPLAY_WIDTH; j++) {
		if ((i / 20) % 2 == (j / 20) % 2) {
		set_pixel(j, i, 1);
		} else {
		set_pixel(j, i, 0);
		}
	}
	}
}

/* ========== Main Driver Functions ========== */

static long nbody_display_ioctl(struct file *f, unsigned int cmd, unsigned long
arg) {
	nbody_sim_config_t sim_config;
	body_pos_t body_position;
	body_t body_parameters;
	vga_ball_arg_t vla;
	int go = 0;
	int status = 0;
	int i;

	printk(KERN_INFO "nbody_display: ioctl command %u received\n", cmd);

	switch (cmd) {
	/* NBody simulation commands */
	case WRITE_GO:
		if (copy_from_user(&go, (int *)arg, sizeof(int)))
		return -EFAULT;
		write_go(go);
		break;

	case NBODY_SET_SIM_PARAMETERS:
		if (copy_from_user(&sim_config, (nbody_sim_config_t *)arg,
sizeof(nbody_sim_config_t)))
		return -EFAULT;
		write_simulation_parameters(&sim_config);
		break;

	case WRITE_READ:
```

```
            if (copy_from_user(&go, (int *)arg, sizeof(int)))
            return -EFAULT;
            write_read(go);
            break;

      case READ_DONE:
            read_done(&status);
            if (copy_to_user((int *)arg, &status, sizeof(int)))
            return -EFAULT;
            break;

      case READ_POSITIONS:
            if (copy_from_user(&body_position, (body_pos_t *)arg,
sizeof(body_pos_t)))
            return -EFAULT;
            read_position(&body_position);
            if (copy_to_user((body_pos_t *)arg, &body_position,
sizeof(body_pos_t)))
            return -EFAULT;
            break;

      case SET_BODY:
            if (copy_from_user(&body_parameters, (body_t *)arg,
sizeof(body_t)))
            return -EFAULT;
            write_body(&body_parameters);
            break;

      /* Display commands */
      case VGA_BALL_WRITE_PROPERTIES:
            if (copy_from_user(&vla, (vga_ball_arg_t *)arg,
sizeof(vga_ball_arg_t))) {
            return -EACCES;
            }

            // Copy new body data
            memcpy(&dev.vga_ball_arg, &vla, sizeof(vga_ball_arg_t));

            // Draw the bodies to the display
            draw_bodies();
            break;

      case VGA_BALL_CLEAR_SCREEN:
            printk(KERN_INFO "nbody_display: Clearing screen\n");
```

```
                clear_framebuffer();
                dev.vga_ball_arg.num_bodies = 0;
                break;

        case VGA_BALL_FILL_SCREEN:
                printk(KERN_INFO "nbody_display: Filling screen\n");
                draw_checkerboard();
                //draw_bodies();
                break;

        default:
                printk(KERN_WARNING "nbody_display: Unknown ioctl command: %u\n",
cmd);
                return -EINVAL;
        }

        return 0;
}


/* The file operations structure */
static const struct file_operations nbody_display_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = nbody_display_ioctl,
};

/* Information for the "misc" framework */
static struct miscdevice nbody_display_misc_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name  = DRIVER_NAME,
        .fops  = &nbody_display_fops,
};


/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init nbody_display_probe(struct platform_device *pdev)
{
        int ret;

        /* Register ourselves as a misc device: creates /dev/vga_ball */
        ret = misc_register(&nbody_display_misc_device);
```

```
	/* 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) {
	ret = -EBUSY;
	goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
	ret = -ENOMEM;
	goto out_release_mem_region;
	}

	//LOOK INTO THIS
	dev.go = 0;
	dev.done = 0;
	dev.read = 0;
	dev.sim_config.N = 0;
	dev.sim_config.gap = 0;
	dev.body_parameters.n = 0;
	dev.body_parameters.x = 0;
	dev.body_parameters.y = 0;
	dev.body_parameters.m = 0;
	dev.body_parameters.vx = 0;
	dev.body_parameters.vy = 0;
	/* Initialize the device with empty screen */
	clear_framebuffer();
	dev.vga_ball_arg.num_bodies = 0;  // Changed from dev.vga_display_arg

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&nbody_display_misc_device);
```

```
	return ret;
}

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

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

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

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

/* Calball when the module is unloaded: release resources */
static void __exit nbody_display_exit(void)
{
```

```
        platform_driver_unregister(&nbody_display_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(nbody_display_init);
module_exit(nbody_display_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Robbie, Moises, Isaac, Adib, Kris");
MODULE_DESCRIPTION("nbody_display driver");
```

nbody_display_driver.h

C/C++

```
#ifndef _NBODY_DISPLAY_DRIVER_H
#define _NBODY_DISPLAY_DRIVER_H

#include <linux/ioctl.h>

// Display parameters
#define DISPLAY_WIDTH 640
#define DISPLAY_HEIGHT 480
#define MAX_BODIES 512

// Framebuffer calculations
#define FRAMEBUFFER_SIZE (DISPLAY_HEIGHT * DISPLAY_WIDTH/32)
#define WORDS_PER_ROW (DISPLAY_WIDTH / 32)
#define BYTE_PER_ROW (DISPLAY_WIDTH / 8)

// ---------------------------------------
// NBody Simulation Structures
// ---------------------------------------

// Body parameters structure
typedef struct {
  double x, y, vx, vy, m;
  int n;
} body_t;

// Body position structure (for reading back)
typedef struct {
  double x, y;
  int n;
```

```
} body_pos_t;

// All positions structure
typedef struct {
  body_pos_t bodies[MAX_BODIES];
} all_positions_t;

// Simulation configuration structure
typedef struct {
  int N;
  int gap;
} nbody_sim_config_t;

// ---------------------------------------
// Display Structures
// ---------------------------------------

// Structure for circle properties
typedef struct {
  unsigned short x, y, radius, n, m;
} vga_ball_props_t;

// Structure for all circles
typedef struct {
  vga_ball_props_t bodies[MAX_BODIES];
  int num_bodies;
} vga_ball_arg_t;

// ---------------------------------------
// IOCTL Commands
// ---------------------------------------
#define UNIFIED_MAGIC 'q'

// NBody simulation commands
#define NBODY_SET_SIM_PARAMETERS _IOW(UNIFIED_MAGIC, 1, nbody_sim_config_t)
#define WRITE_GO                 _IOW(UNIFIED_MAGIC, 2, int)
#define READ_DONE                _IOR(UNIFIED_MAGIC, 3, int)
#define READ_POSITIONS           _IOR(UNIFIED_MAGIC, 4, body_pos_t)
#define WRITE_READ               _IOW(UNIFIED_MAGIC, 5, int)
#define SET_BODY                 _IOW(UNIFIED_MAGIC, 6, body_t)

// Display commands (offset to avoid conflict)
#define VGA_BALL_WRITE_PROPERTIES _IOW(UNIFIED_MAGIC, 7, vga_ball_arg_t)
#define VGA_BALL_CLEAR_SCREEN     _IO(UNIFIED_MAGIC, 8)
```

```
#define VGA_BALL_FILL_SCREEN      _IO(UNIFIED_MAGIC, 9)

#endif /* _NBODY_DISPLAY_DRIVER_H */
```

display.c

```
C/C++
/*
 * Userspace program that reads in the CSV and plays the simulation on the VGA
display
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "nbody_display_driver.h"

#define MAXCHAR 100000
#define CSV_FILENAME "nbody_results.csv"
#define PLAYBACK_DELAY_MS 100
#define MAX_TIMESTEPS 40000   //Fixed allocation (will fix later)
#define MIN_RADIUS 2
#define MAX_RADIUS 6

typedef struct {
       float x;
       float y;
       int n;
       float m;
} body_full;
// Function to convert nbody simulation coordinates to display coordinates
static void convert_coordinates(float nbody_x, float nbody_y,
                                short *display_x, short *display_y) {
       // Scale from -500,500 range to display coordinates
       *display_x = (short)((nbody_x + 500.0) / 1000.0 * (DISPLAY_WIDTH));
```

```
	*display_y = (short)((nbody_y + 500.0) / 1000.0 * (DISPLAY_HEIGHT));

}

// Function to parse a CSV line into a vga_ball_arg_t structure -- This
populates one timestep of data
static int parse_csv_line(char* line, body_full* arg, int max_bodies) {
	char* token;

	//Ensure num_bodies starts at 0
	int num_bodies = 0;
	// Get the timestep (first column)
	token = strtok(line, ",");
	if (!token) return -1;

	// Process each body's x and y coordinates
	for (int i = 0; i < max_bodies; i++) {
	// Passing in Null to strtok will continue tokenizing the same string
	token = strtok(NULL, ",");
	if (!token) break;  //Exit the loop, we've accounted for all bodies
	float x = atof(token);

	token = strtok(NULL, ",");
	if (!token) break;  //Exit the loop, we've accounted for all bodies
	float y = atof(token);

	token = strtok(NULL, ",");
	if (!token) break;  //Exit the loop, we've accounted for all bodies
	float m = atof(token);

	if(m == 0) {
		continue; // Skip bodies with mass 0
	}
	arg[i].x = x;
	arg[i].y = y;
	arg[i].n = i;
	arg[i].m = m;
	num_bodies++;
	}

	return num_bodies;
}

int main(int argc, char** argv) {
```

```
       if (argc != 2) {
       fprintf(stderr, "Usage: %s <playback_speed>\n", argv[0]);
       fprintf(stderr, "  playback_speed: speed multiplier (1.0 = normal
speed)\n");
       return -1;
       }

       double playback_speed = atof(argv[1]);
       if (playback_speed <= 0) {
       fprintf(stderr, "Playback speed must be positive\n");
       return -1;
       }

       int vga_fd;
       static const char vga_device[] = "/dev/nbody_display";
       if ((vga_fd = open(vga_device, O_RDWR)) == -1) {
       fprintf(stderr, "Could not open %s\n", vga_device);
       return -1;
       }

       FILE* csv_file = fopen(CSV_FILENAME, "r");
       if (!csv_file) {
       fprintf(stderr, "Could not open file %s\n", CSV_FILENAME);
       close(vga_fd);
       return -1;
       }

       printf("Reading simulation data from %s\n", CSV_FILENAME);

       // Skip header line
       char line[MAXCHAR];
       fgets(line, MAXCHAR, csv_file);

       // Allocate memory for all timesteps at once (fixed allocation)
       vga_ball_arg_t* simulation_data = calloc(MAX_TIMESTEPS,
sizeof(vga_ball_arg_t));
       body_full* row_data = calloc(MAX_BODIES, sizeof(body_full));

       if (!simulation_data) {
       fprintf(stderr, "Memory allocation failed\n");
       fclose(csv_file);
       close(vga_fd);
       return -1;
       }
```

```
	//Find the actual amt of timesteps
	int actual_timesteps = 0;

	char first_line = 1;
	float xzero, yzero, xrange, yrange;
	float mzero, mrange;
	//Read from CSV (one line per timestep)
	while (fgets(line, MAXCHAR, csv_file)) {
	//Use above function to parse the line
	int n_bodies = parse_csv_line(line, row_data, MAX_BODIES);

	if (first_line) {
		simulation_data[0].num_bodies = n_bodies;
		printf("Detected %d bodies in the simulation\n", n_bodies);
	} else {
		//Bodies stay the same for each timestep
		simulation_data[actual_timesteps].num_bodies =
simulation_data[0].num_bodies;
	}
	if(first_line) {
		first_line = 0;
		xzero = row_data[0].x;
		yzero = row_data[0].y;
		mzero = row_data[0].m;
		xrange = row_data[0].x;
		yrange = row_data[0].y;
		mrange = row_data[0].m;

		for (int i = 1; i < n_bodies; i++) {
		// printf("Body %d: x=%.2f, y=%.2f, m=%.2f\n", i, row_data[i].x,
row_data[i].y, row_data[i].m);
		if (row_data[i].x < xzero) xzero = row_data[i].x;
		if (row_data[i].y < yzero) yzero = row_data[i].y;
		if (row_data[i].m < mzero) mzero = row_data[i].m;

		if (row_data[i].x > xrange) xrange = row_data[i].x;
		if (row_data[i].y > yrange) yrange = row_data[i].y;
		if (row_data[i].m > mrange) mrange = row_data[i].m;
		}

		xrange -= xzero;
		yrange -= yzero;
		xzero -= .1 * xrange;
```

```
            yzero -= .1 * yrange;
            xrange *= 1.2;
            yrange *= 1.2;
            mrange -= mzero;
      }

      for (int i = 0; i < n_bodies; i++) {
            if(row_data[i].x > xzero && row_data[i].x < xzero + xrange &&
            row_data[i].y > yzero && row_data[i].y < yzero + yrange &&
row_data[i].m != 0) {
            simulation_data[actual_timesteps].bodies[i].x = ((row_data[i].x -
xzero) / xrange) * (float)DISPLAY_WIDTH;
            simulation_data[actual_timesteps].bodies[i].y = ((row_data[i].y -
yzero) / yrange) * (float)DISPLAY_HEIGHT;
            simulation_data[actual_timesteps].bodies[i].m = ((row_data[i].m -
mzero) / mrange) * (float) 255;
            simulation_data[actual_timesteps].bodies[i].n = i;
            simulation_data[actual_timesteps].bodies[i].radius = (unsigned
short)(((row_data[i].m - mzero) / mrange) * (float)(MAX_RADIUS - MIN_RADIUS) +
MIN_RADIUS);


            } else {
            simulation_data[actual_timesteps].bodies[i].radius = 0;
            }
      }

      // When reading the first line, theres a special case to populate
num_bodies


      actual_timesteps++;

      }

      fclose(csv_file);
      printf("Loaded %d timesteps with %d bodies\n", actual_timesteps,
simulation_data[0].num_bodies);

      // double min_mass = 1e100;  // Start with a very large number
      // double max_mass = 0; //Start with 0

      // for (int i = 0; i < simulation_data[0].num_bodies; i++) {
      //     if (simulation_data[0].bodies[i].m < min_mass) {
```

```
        //           min_mass = simulation_data[0].bodies[i].m;
        //     }
        //     if (simulation_data[0].bodies[i].m > max_mass) {
        //           max_mass = simulation_data[0].bodies[i].m;
        //     }
        // }

        // printf("Mass range: min=%.2f, max=%.2f\n", min_mass, max_mass);

        // for (int t = 0; t < actual_timesteps; t++) {
        //     for (int i = 0; i < simulation_data[t].num_bodies; i++) {
        //           double mass_normalized = 0.0;

        //           // Prevent division by zero if all masses are the same
        //           if (max_mass > min_mass) {
        //           //the percentage of max mass
        //           mass_normalized = (simulation_data[t].bodies[i].m -
min_mass) / (max_mass - min_mass);
        //           }

        //           simulation_data[t].bodies[i].radius = (unsigned
short)(mass_normalized * MAX_RADIUS + MIN_RADIUS);
        //     }
        // }


        // Clear screen
        if (ioctl(vga_fd, VGA_BALL_CLEAR_SCREEN, 0) < 0) {
        perror("ioctl(VGA_BALL_CLEAR_SCREEN) failed");
        free(simulation_data);
        close(vga_fd);
        return -1;
        }

        // Playback loop
        for (int t = 0; t < actual_timesteps; t++) {
        if (ioctl(vga_fd, VGA_BALL_WRITE_PROPERTIES, &simulation_data[t]) < 0) {
              perror("ioctl(VGA_BALL_WRITE_PROPERTIES) failed");
              break;
        }

        usleep((int)(PLAYBACK_DELAY_MS * 1000 / playback_speed));
```

```
        }

        printf("\nPlayback complete\n");

        free(simulation_data);
        close(vga_fd);

        return 0;
}
```

nbody.sv

```C/C++
//Nbody.sv
// This module is the wrapper for the entire system. It will take in values
from the bus
// write to its memories, and call getaccl and leapfrog
//
// Lower 9 (log2 BODIES) bits are body number
// Upper 7 (16 - log2 BODIES) bits other params
// REGISTERS:
// 0000000 = GO
// 0000001 = READ
// 0000010 = N_BODIES
// 0000011 = GAP
// MEMORY:
// 0000100 = write_x_data_LOWER
// 0000101 = write_x_data_UPPER
// 0000110 = write_y_data_LOWER
// 0000111 = write_y_data_UPPER
// 0001000 = write_m_data_LOWER
// 0001001 = write_m_data_UPPER
// 0001010 = write_vx_data_LOWER
// 0001011 = write_vx_data_UPPER
// 0001100 = write_vy_data_LOWER
// 0001101 = write_vy_data_UPPER

// 1000000 = DONE
// 1000001 = READ_X_LOWER
// 1000010 = READ_X_UPPER
// 1000011 = READ_Y_LOWER
```

```
// 1000100 = READ_Y_UPPER


`timescale 1 ps / 1 ps
module nbody #(
	parameter BODIES = 512,
	parameter EXT_DATA_WIDTH = 32, // for Reading and writing from the bus :)
	parameter DATA_WIDTH = 64,
	parameter ADDR_WIDTH = 16,
	parameter BODY_ADDR_WIDTH = $clog2(BODIES),
	parameter MultTime = 5, // Number of cycles for mult
	parameter AddTime = 9, // Number of cycles for add/sub
	parameter InvSqrtTime = 17, // Number of cycles for invsqert
	parameter AcclLatency = AddTime + MultTime + AddTime + InvSqrtTime +
MultTime * 4 + 1, // The one is for the startup thing we need to do to confirm
we dont devide by 0
	parameter MinBodies = 21
)(
	input logic clk,
	input logic rst,
	input logic [EXT_DATA_WIDTH-1:0] writedata,
	input logic read,
	input logic write,
	input logic [ADDR_WIDTH-1:0] addr,
	input logic chipselect,
	output logic [EXT_DATA_WIDTH-1:0] readdata,
	output logic [7:0] VGA_R, VGA_G, VGA_B,
	output logic 	   VGA_CLK, VGA_HS, VGA_VS,
			   VGA_BLANK_n,
	output logic 	   VGA_SYNC_n

);

	localparam SW_READ_WRITE = 2'b00;
	localparam CALC_ACCEL = 2'b01;
	localparam UPDATE_POS = 2'b10;

	// Registers:
	localparam GO             = 7'h40;
	localparam READ           = 7'h41;
	localparam N_BODIES       = 7'h42;
	localparam GAP            = 7'h43;
	// Memory:
	localparam X_SEL_LOWER    = 7'h44;
```

```
        localparam X_SEL_UPPER     = 7'h45;
        localparam Y_SEL_LOWER     = 7'h46;
        localparam Y_SEL_UPPER     = 7'h47;
        localparam M_SEL_LOWER     = 7'h48;
        localparam M_SEL_UPPER     = 7'h49;
        localparam VX_SEL_LOWER    = 7'h4a;
        localparam VX_SEL_UPPER    = 7'h4b;
        localparam VY_SEL_LOWER    = 7'h4c;
        localparam VY_SEL_UPPER    = 7'h4d;
        // Out:
        localparam DONE            = 7'h50;
        localparam READ_X_LOWER    = 7'h51;
        localparam READ_X_UPPER    = 7'h52;
        localparam READ_Y_LOWER    = 7'h53;
        localparam READ_Y_UPPER    = 7'h54;

        logic go;
        logic done;
        logic read_sw;
        logic [$clog2(BODIES)-1:0] num_bodies;
        logic [EXT_DATA_WIDTH-1:0] gap, gap_counter;
        logic [DATA_WIDTH-1:0] write_vy_data, write_vx_data, write_m_data,
write_x_data, write_y_data;
        logic wren_x, wren_y, wren_m, wren_vx, wren_vy;
        logic [DATA_WIDTH-1:0] out_x, out_y, out_m, out_vx, out_vy;
        logic [1:0] state;
        logic [BODY_ADDR_WIDTH-1:0] m_read_addr, v_read_addr;
        logic [BODY_ADDR_WIDTH-1:0] pos_input_1_addr;
        logic [BODY_ADDR_WIDTH-1:0] pos_input_2_addr;
        logic [BODY_ADDR_WIDTH-1:0] s1_counter;

        logic [DATA_WIDTH-1:0] ax, ay;
        logic [DATA_WIDTH-1:0] ax_shifted, ay_shifted;

        logic [BODY_ADDR_WIDTH-1:0] p_read_i, p_read_j, v_read_i, v_read_j,
v_write_i, v_write_j;
        logic valid_accl, valid_dv;

        logic [DATA_WIDTH-1:0] x_output_1, y_output_1, x_output_2, y_output_2;
        // Odd things will happen if you try to write when the hardware is not in
a state where it expects you too, as the addresses passed into the ram will be
wrong, but you will still be writing.
```

```
	logic state_2_write_enable; // This one is 0 when when we are not in state 2, if if we are, it tracks whether we are writing (based on latency of adders and such)
	logic [BODY_ADDR_WIDTH-1:0] v_write_addr;

	logic [BODY_ADDR_WIDTH-1:0] state_2_read, state_2_pos_write;

	logic [DATA_WIDTH-1:0] add_x_out, add_y_out, add_x_input_1, add_x_input_2, add_y_input_1, add_y_input_2;
	logic [EXT_DATA_WIDTH-1:0] write_register;
	logic [31:0] state_1_timer;
	logic endstate;

	logic first_time;


	// // Instantiating the display
	Display display(
	.clk(clk),
	.reset(rst),
	.writedata(writedata),
	.write(write&(~addr[15])), // Don't write to the display
	.chipselect(chipselect),
	.address(addr[14:0]),
	.VGA_R(VGA_R),
	.VGA_G(VGA_G),
	.VGA_B(VGA_B),
	.VGA_CLK(VGA_CLK),
	.VGA_HS(VGA_HS),
	.VGA_VS(VGA_VS),
	.VGA_BLANK_n(VGA_BLANK_n),
	.VGA_SYNC_n(VGA_SYNC_n)
	);
	// For testing: force all VGA outputs low


	// The main state machine
	always_ff @(posedge clk or posedge rst) begin
	//TODO: logic for letting software read and write values goes here
	if (rst) begin
		state <= SW_READ_WRITE;
		gap_counter <= 0;
		gap <= 0;
		num_bodies <= 0;
```

```
                done <= 0;
        end else begin
                if (write == 1 && chipselect == 1) begin
                if (addr[15:9] == GO) begin
                        go <= writedata[0];
                end else if (addr[15:9] == READ) begin
                        read_sw <= writedata[0];
                end else if (addr[15:9] == N_BODIES) begin
                        num_bodies <= writedata[BODY_ADDR_WIDTH-1:0];
                end else if (addr[15:9] == GAP) begin
                        gap <= writedata;
                end
                if (addr[9] == 0) begin
                        write_register <= writedata;
                end
                end

                case (state)
                SW_READ_WRITE: begin // Software reading/writing
                        //if go is not high, then we are not going to do anything
(except take in writes from software)
                        // if we raised done, we are waiting for read to go high
before dropping done
                        //once read and done are both low (and go is high
obvisously), we can start the next cycle

                        if(go == 1) begin //handshake logic
                                if(read_sw == 1) begin
                                done <= 0;
                                end else if (done == 0) begin
                                state <= CALC_ACCEL;
                                state_1_timer <= 0;
                                v_write_i <= 0;
                                v_write_j <= 0;
                                v_read_i  <= 0;
                                v_read_j  <= 0;
                                p_read_i  <= 0;
                                p_read_j  <= 0;
                                valid_dv  <= 0;
                                valid_dv  <= 0;
                                valid_accl <= 0;
                                gap_counter <= 0;
                                end
                        end
```

```
                    else begin
                          first_time <= 1;
                          done <= 0;
                    end
                    // zeroing out all the shit


             end
             CALC_ACCEL: begin // Compute accelerations, update velocities
                    if (go == 1'b0) begin
                          state <= SW_READ_WRITE;
                    end
                    else if (v_write_i == num_bodies - 1 && v_write_j ==
num_bodies - 1) begin
                          // Finished, start UPDATE_POS
                          state <= UPDATE_POS;
                          state_2_write_enable <= 1'b0;
                          state_2_pos_write <= 0;
                          state_2_read <= 0;
                          state_2_pos_write <= 0;
                          state_2_write_enable <= 1'b0;
                          state_1_timer <= 0;
                          v_write_i <= 0;
                          v_write_j <= 0;
                          v_read_i  <= 0;
                          v_read_j  <= 0;
                          p_read_i  <= 0;
                          p_read_j  <= 0;
                          valid_dv  <= 0;
                          valid_accl  <= 0;
                    end
                    else begin
                          state_1_timer <= state_1_timer + 1;
                          p_read_j <= p_read_j + 9'b1;
                          if (state_1_timer == AcclLatency - 1) begin
                          valid_accl <= 1'b1;
                          end
                          if (state_1_timer == AcclLatency + AddTime + 1) begin
                          valid_dv <= 1'b1;
                          end
                          if (valid_accl) begin
                          v_read_j <= v_read_j + 9'b1;
                          end
                          if (valid_dv) begin
```

```
                        v_write_j <= v_write_j + 9'b1;
                        end

                        if (p_read_j == num_bodies - 1) begin
                        p_read_j <= 9'b0;
                        p_read_i <= p_read_i + 9'b1;
                        end

                        if (v_read_j == num_bodies - 1) begin
                        v_read_j <= 0;
                        v_read_i <= v_read_i + 9'b1;
                        end

                        if (v_write_j == num_bodies - 1) begin
                        v_write_j <= 0;
                        v_write_i <= v_write_i + 9'b1;
                        end



                end

        end
        UPDATE_POS: begin // Update positions



                state_2_read <= state_2_read + 9'b1;

                if (go == 0) begin
                        state <= SW_READ_WRITE;
                end
                if (state_2_read == AddTime+1) begin
                        // finished the startup time, now we can start
writing things back
                        state_2_write_enable <= 1'b1;
                end else if (state_2_write_enable) begin
                        if (state_2_pos_write != num_bodies - 1) begin
                        state_2_pos_write <= state_2_pos_write + 9'b1; //
must be zeroed out at the start
                        end else begin
                        state_2_write_enable <= 1'b0;
                        state_2_pos_write <= 0;
                        first_time <= 0;
```

```
                        if (gap_counter == gap - 1) begin
                                state <= SW_READ_WRITE;
                                done <= 1;
                        end else begin
                                state <= CALC_ACCEL;
                                gap_counter <= gap_counter + 1;
                        end
                        end
                end
        end
        default: state <= SW_READ_WRITE; // Default to idle state
        endcase
end
end

always_comb begin : blockName
if (read == 1 && chipselect == 1) begin
        if (addr[15:9] == DONE) begin
        readdata = {{(EXT_DATA_WIDTH-1){1'b0}}, done};
        end else if (addr[15:9] == READ_X_LOWER) begin
        readdata =  x_output_1[EXT_DATA_WIDTH-1:0];
        end else if (addr[15:9] == READ_X_UPPER) begin
        readdata =  x_output_1[DATA_WIDTH-1:EXT_DATA_WIDTH];
        end else if (addr[15:9] == READ_Y_LOWER) begin
        readdata =  y_output_1[EXT_DATA_WIDTH-1:0];
        end else if (addr[15:9] == READ_Y_UPPER) begin
        readdata =  y_output_1[DATA_WIDTH-1:EXT_DATA_WIDTH];
        end else begin
        readdata = {EXT_DATA_WIDTH{1'b1}};
        end
end
else begin
        readdata <= {EXT_DATA_WIDTH{1'b0}};
end
case (state)
        SW_READ_WRITE: begin
        // All the memories get the addr.
        pos_input_1_addr = addr[BODY_ADDR_WIDTH-1:0];
        pos_input_2_addr = addr[BODY_ADDR_WIDTH-1:0];
        m_read_addr = addr[BODY_ADDR_WIDTH-1:0];
        v_read_addr = addr[BODY_ADDR_WIDTH-1:0];
        v_write_addr = addr[BODY_ADDR_WIDTH-1:0];

        write_x_data = {writedata,write_register};
```

```
            write_y_data = {writedata,write_register};
            write_m_data = {writedata,write_register};
            write_vx_data = {writedata,write_register};
            write_vy_data = {writedata,write_register};

            // Write enable for the different memories
            wren_x = (addr[15:9] == X_SEL_UPPER) ? (write & chipselect) :
1'b0;
            wren_y = (addr[15:9] == Y_SEL_UPPER) ? (write & chipselect) :
1'b0;
            wren_m = (addr[15:9] == M_SEL_UPPER) ? (write & chipselect) :
1'b0;
            wren_vx = (addr[15:9] == VX_SEL_UPPER) ? (write & chipselect) :
1'b0;
            wren_vy = (addr[15:9] == VY_SEL_UPPER) ? (write & chipselect) :
1'b0;


            ax_shifted = 0;
            ay_shifted = 0;
            add_x_input_1 = 0;
            add_x_input_2 = 0;
            add_y_input_1 = 0;
            add_y_input_2 = 0;

            end
            CALC_ACCEL: begin

            pos_input_1_addr = p_read_i;
            pos_input_2_addr = p_read_j;
            m_read_addr = p_read_i;
            v_read_addr = v_read_j;
            v_write_addr = v_write_j;

            // The 0 values don't matter
            write_x_data = 0;
            write_y_data = 0;
            write_m_data = 0;
            write_vx_data = add_x_out;
            write_vy_data = add_y_out;

            wren_x = 0;
            wren_y = 0;
            wren_m = 0;
            wren_vx = valid_dv;
```

```
                wren_vy = valid_dv;

                //TODO: We are assuming big indian, if not, deal with it
                if (first_time) begin
                        ax_shifted = {ax[63], ax[62:52] > 0? ax[62:52] - 11'b1 :
11'b0, ax[51:0]};
                        ay_shifted = {ay[63], ay[62:52] > 0? ay[62:52] - 11'b1 :
11'b0, ay[51:0]};
                end else begin
                        ax_shifted = ax;
                        ay_shifted = ay;
                end

                add_x_input_1 = ax_shifted;
                add_x_input_2 = out_vx;
                add_y_input_1 = ay_shifted;
                add_y_input_2 = out_vy;


                end
                UPDATE_POS: begin
                pos_input_1_addr = state_2_pos_write;
                pos_input_2_addr = state_2_read;
                v_read_addr = state_2_read; //also zero at the start, these should
always hold the same value (v_read and pos)
                m_read_addr = 0;
                v_write_addr = 0;

                write_x_data = add_x_out; //Make sure this is zeroed out before
hand
                write_y_data = add_y_out;
                write_m_data = 0;
                write_vx_data = 0;
                write_vy_data = 0;


                wren_x = state_2_write_enable;
                wren_y = state_2_write_enable;
                wren_m = 0;
                wren_vx = 0;
                wren_vy = 0;

                ax_shifted = 0;
                ay_shifted = 0;
```

```
        add_x_input_1 = x_output_2;
        add_x_input_2 = out_vx;
        add_y_input_1 = y_output_2;
        add_y_input_2 = out_vy;
        end
        default: begin
        // No action
        end
endcase
end
// Write enable for software

AddSub AddX(
.clk(clk),
.areset(rst),
.a(add_x_input_1),
.b(add_x_input_2),
.q(add_x_out)
);
AddSub AddY(
.clk(clk),
.areset(rst),
.a(add_y_input_1),
.b(add_y_input_2),
.q(add_y_out)
);
RAM2   x(
.clock ( clk ),
.address_a ( pos_input_1_addr ),
.address_b ( pos_input_2_addr ),
.data_a ( write_x_data ),
.data_b ( 64'b0 ),
.wren_a ( wren_x ),
.wren_b ( 1'b0 ),
.q_a ( x_output_1 ),
.q_b ( x_output_2 )
);
RAM2   y(
.clock ( clk ),
.address_a ( pos_input_1_addr ),
.address_b ( pos_input_2_addr ),
.data_a ( write_y_data ),
.data_b ( 64'b0 ),
```

```
.wren_a ( wren_y ),
.wren_b ( 1'b0 ),
.q_a ( y_output_1 ),
.q_b ( y_output_2 )
);
RAM    RAM_m (
.clock ( clk ),
.data ( write_m_data ),
.rdaddress ( m_read_addr ),
.wraddress ( addr[BODY_ADDR_WIDTH-1:0] ),
.wren ( wren_m ),
.q ( out_m )
);
RAM    RAM_vx (
.clock ( clk ),
.data ( write_vx_data ),
.rdaddress ( v_read_addr ),
.wraddress ( v_write_addr ),
.wren ( wren_vx ),
.q ( out_vx )
);
RAM    RAM_vy (
.clock ( clk ),
.data ( write_vy_data ),
.rdaddress ( v_read_addr ),
.wraddress ( v_write_addr ),
.wren ( wren_vy ),
.q ( out_vy )
);

getAccl #(
.MultTime(MultTime),
.AddTime(AddTime),
.InvSqrtTime(InvSqrtTime)
) accl (
.rst(rst), // NOT CORRECT TODO
.clk(clk),
.x1(x_output_1),
.y1(y_output_1),
.x2(x_output_2),
.y2(y_output_2),
.m2(out_m),
.ax(ax),
.ay(ay)
```

```
      );

endmodule
```

display.Sv

```
C/C++
/* DISPLAY.sv
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Register map:

 *  Memory-mapped framebuffer:
 * - 32K addressable locations (15-bit address)
 * - Each 32-bit word contains 32 pixels (1 bit per pixel)
 * - Addressing: 0x0000 to 0x7FFF
 * - Pixel data: 1 = white (0xFF), 0 = black (0x00)
 * - Pixels are read from framebuffer in sequence and displayed on screen
 */
// This module is built to run on 50 mhz, to get it on 100 you need to do some
things.

module Display(
       input logic         clk,
      input logic          reset,
       input logic [31:0]  writedata,
       input logic         write,
       input        chipselect,
       input logic [14:0]  address,

       output logic [7:0] VGA_R, VGA_G, VGA_B,
       output logic        VGA_CLK, VGA_HS, VGA_VS,
                           VGA_BLANK_n,
       output logic        VGA_SYNC_n);

       logic [10:0]        hcount;
       logic [9:0]  vcount;
       logic [31:0] vcount_32;
       logic [31:0] vcount_x_512;
```

```
        logic [31:0] vcount_x_128;
        logic [31:0] vcountx20;

        logic [14:0] rdaddress;
        logic [31:0] placecounter;
        logic [31:0] next_pix;
        logic [31:0]   readdata;
        logic [31:0] hcount_32;

    vga_counters counters(.clk50(clk), .*);

    RAM_DISP framebuffer(
        .clock(clk),
        .data(writedata),
        .wren(write),
        .wraddress(address),
        .rdaddress(rdaddress),
        .q(readdata)
        );
        assign vcount_32 = {22'b0, vcount};
        assign vcount_x_512 = vcount_32 << 8;
        assign vcount_x_128 = vcount_32 << 10;
        assign hcount_32 = (hcount > 11'd1300) ? 32'd1300 : {21'b0, hcount};
        assign vcountx20 = vcount_x_128 + vcount_x_512;
        assign placecounter = vcountx20 + hcount_32 + 32'd2;
        assign rdaddress = placecounter[20:6];


    always_comb begin
        if (readdata[hcount[5:1]] == 1'b1) begin
        VGA_R = 8'hff;
        VGA_G = 8'hff;
        VGA_B = 8'hff;
        end else begin
        VGA_R = 8'h00;
        VGA_G = 8'h00;
        VGA_B = 8'h00;
        end
    end

endmodule

module vga_counters(
 input logic        clk50, reset,
```

```
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0           1279         1599 0
 *            ---------------           --------
 * ___________|     Video         |_____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *     _______________________    _____________
 * |____|     VGA_HS        |____|
 */
   // Parameters for hcount
   parameter HACTIVE       = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC         = 11'd 192,
             HBACK_PORCH   = 11'd 96,
             HTOTAL        = HACTIVE + HFRONT_PORCH + HSYNC +
                             HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE       = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC         = 10'd 2,
             VBACK_PORCH   = 10'd 33,
             VTOTAL        = VACTIVE + VFRONT_PORCH + VSYNC +
                             VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
       if (reset)          hcount <= 0;
       else if (endOfLine) hcount <= 0;
       else                hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
```

```
        if (reset)          vcount <= 0;
        else if (endOfLine)
        if (endOfField)   vcount <= 0;
        else              vcount <= vcount + 10'd 1;

    assign endOfField = vcount == VTOTAL - 1;

    // Horizontal sync: from 0x520 to 0x5DF (0x57F)
    // 101 0010 0000 to 101 1101 1111
    assign VGA_HS = !( (hcount[10:8] == 3'b101) &
              !(hcount[7:5] == 3'b111));
    assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

    assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused

    // Horizontal active: 0 to 1279      Vertical active: 0 to 479
    // 101 0000 0000  1280        01 1110 0000  480
    // 110 0011 1111  1599        10 0000 1100  524
    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
              !( vcount[9] | (vcount[8:5] == 4'b1111) );

    /* VGA_CLK is 25 MHz
        *             __     __     __
        * clk50       __|  |__|  |__|
        *
        *             _____         __
        * hcount[0]__|      |_____|
        */
    assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule
```

getAccl.Sv

C/C++

```
// This module should take in the XYZ of one body, XYZ and M of the second,
// and output AX, AY, AZ experinced by the first due to the second
// The wrapping program has to ensure that valid data is being fed in, and that
it keeps track of what is coming out.
// Latency is <<INPUT THIS>> cycles
```

```
`timescale 1 ps / 1 ps

module getAccl #(
	parameter MultTime = 11, // Number of cycles for mult
	parameter AddTime = 20, // Number of cycles for add/sub
	parameter InvSqrtTime = 27 // Number of cycles for invsqert
)(
	input logic rst,		// Reset signal
	input logic clk,		// Clock signal
	input logic [63:0] x1,	// X coordinate of the first body
	input logic [63:0] y1,	// Y coordinate of the first body
	input logic [63:0] x2,	// X coordinate of the second body
	input logic [63:0] y2,	// Y coordinate of the second body
	input logic [63:0] m2,	// Mass of the second body (premultaplied by
G)
	output logic [63:0] ax,	// Acceleration in the X direction
	output logic [63:0] ay	// Acceleration in the Y di ); rection
);
	logic [63:0] m_late, dx, dy, dx2, dy2, dx_late, dy_late, d_inv, d_inv_2,
d_inv_3, f_val, d2;
	logic [63:0] d_inv_late;
	logic [63:0] x1_real, y1_real, x2_real, y2_real, m2_real;

	always_ff @(posedge clk or posedge rst) begin
	if (rst) begin
		x1_real <= 0;
		y1_real <= 0;
		x2_real <= 1;
		y2_real <= 1;
		m2_real <= 0;
	end else begin
		y1_real <= y1;
		y2_real <= y2;
		if (x1 == x2 && y1 == y2) begin
		m2_real <= 0; // Set m2 to 0 if comparing a body to itself
		x2_real <= 64'h405c000000000000;
		x1_real <= 64'heff0000000000000;
		end else begin
		m2_real <= m2;
		x2_real <= x2;
		x1_real <= x1;
		end
	end
	end
```

```
	shift_register #(.SHIFTS(AddTime + MultTime + AddTime + InvSqrtTime +
MultTime * 2)) shiftM // This is 20+49+37+11+11 for each of the objects this
has to pass around
	(
	.clk(clk),
	.rst(rst),
	.in(m2_real),
	.out(m_late)
	);
	shift_register #(.SHIFTS(MultTime + AddTime + InvSqrtTime + MultTime *
3)) shiftDX // This is 49+37+11+11 for each of the objects this has to pass
around
	(
	.clk(clk),
	.rst(rst),
	.in(dx),
	.out(dx_late)
	);
	shift_register #(.SHIFTS(MultTime + AddTime + InvSqrtTime + MultTime *
3)) shiftDY // This is 49+37+11+11 for each of the objects this has to pass
around
	(
	.clk(clk),
	.rst(rst),
	.in(dy),
	.out(dy_late)
	);

	AddSub subX(
	.clk(clk),
	.areset(rst),
	.a(x1_real),
	.b(x2_real),
	.s(dx)
	);

	AddSub subY(
	.clk(clk),
	.areset(rst),
	.a(y1_real),
	.b(y2_real),
	.s(dy)
	);
```

```
Mult multDX2 (
.clk(clk),
.areset(rst),
.a(dx),
.b(dx),
.q(dx2)
);

Mult multDY2 (
.clk(clk),
.areset(rst),
.a(dy),
.b(dy),
.q(dy2)
);
AddSub addD(
.clk(clk),
.areset(rst),
.a(dy2),
.b(dx2),
.q(d2)
);
InvSqrt invs(
.clk(clk),
.areset(rst),
.a(d2),
.q(d_inv)
);

shift_register #(.SHIFTS(MultTime)) shiftDInv (
.clk(clk),
.rst(rst),
.in(d_inv),
.out(d_inv_late)
);

Mult multinvD2 (
.clk(clk),
.areset(rst),
.a(d_inv),
.b(d_inv),
.q(d_inv_2)
);
```

```
        Mult multinvD3 (
        .clk(clk),
        .areset(rst),
        .a(d_inv_2),
        .b(d_inv_late),
        .q(d_inv_3)
        );

        Mult multF (
        .clk(clk),
        .areset(rst),
        .a(m_late),
        .b(d_inv_3),
        .q(f_val)
        );

        Mult multAX (
        .clk(clk),
        .areset(rst),
        .a(f_val),
        .b(dx_late),
        .q(ax)
        );

        Mult multAY (
        .clk(clk),
        .areset(rst),
        .a(f_val),
        .b(dy_late),
        .q(ay)
        );


endmodule
```

shift_register.sv

```
C/C++
module shift_register #(
	parameter SHIFTS = 4 // Number of shifts (delay cycles)
)(
	input wire clk,                  // Clock signal
	input wire rst,                  // Reset signal (active high)
	input wire [63:0] in,            // 64-bit input signal
	output wire [63:0] out           // 64-bit output signal
);

	// Internal shift register storage
	reg [63:0] shift_reg [SHIFTS-1:0];

	// Shift register logic
	always @(posedge clk or posedge rst) begin
	if (rst) begin
		integer i;
		for (i = 0; i < SHIFTS; i = i + 1) begin
		shift_reg[i] <= 64'b0; // Reset all bits to 0
		end
	end else begin
		integer i;
		for (i = SHIFTS-1; i > 0; i = i - 1) begin
		shift_reg[i] <= shift_reg[i-1]; // Shift values
		end
		shift_reg[0] <= in; // Insert new input
	end
	end

	// Output the value from SHIFTS cycles ago
	assign out = shift_reg[SHIFTS-1];

endmodule
```

getAcclTB.sv

```
C/C++
`timescale 1 ps / 1 ps
module getAcclTb;
```

```
    // Parameters
    parameter CLK_PERIOD = 10; // Clock period in ns (100 MHz)
    parameter MultTime = 11;   // Match DUT parameter
    parameter AddTime = 20;    // Match DUT parameter
    parameter InvSqrtTime = 27;// Match DUT parameter
    // Total latency = AddTime + MultTime + AddTime + InvSqrtTime + MultTime +
MultTime + MultTime = 122 cycles
    parameter LATENCY = 122;

    // Testbench Signals
    logic clk;
    logic rst;
    logic [63:0] x1, y1, z1;
    logic [63:0] x2, y2, m2;
    logic [63:0] ax, ay;

    // Instantiate the Design Under Test (DUT)
    getAccl #(
        .MultTime(MultTime),
        .AddTime(AddTime),
        .InvSqrtTime(InvSqrtTime)
    ) dut (
        .rst(rst),
        .clk(clk),
        .x1(x1),
        .y1(y1),
        .x2(x2),
        .y2(y2),
        .m2(m2),
        .ax(ax),
        .ay(ay)
        // az is not an output of the provided module
    );

    // Clock Generation
    initial begin
        clk = 0;
        forever #(CLK_PERIOD / 2) clk = ~clk;
    end

    // Reset and Stimulus Generation
    initial begin
        // For better time printing
        $timeformat(-9, 2, " ns", 10);
```

```
        // Initialize inputs (using floating point representation)
        rst = 1;
        x1 = $realtobits(0.0); y1 = $realtobits(0.0); z1 = $realtobits(0.0);
        x2 = $realtobits(0.0); y2 = $realtobits(0.0); m2 = $realtobits(0.0);

        // Apply reset
        # (CLK_PERIOD * 5);
        rst = 0;
        $display("Time=%t: Deasserting reset.", $time);

        // Wait for reset to propagate
        # (CLK_PERIOD);
       @(posedge clk);

        // --- Test Case 1 ---
        $display("Time=%t: Applying Test Case 1", $time);
        x1 = $realtobits(10.0); y1 = $realtobits(20.0); z1 = $realtobits(0.0);
        x2 = $realtobits(10.0); y2 = $realtobits(20.0); m2 =
$realtobits(500.0); // Example values
        # (CLK_PERIOD * (1)); // Wait for output + margin
       @(posedge clk);
        x1 = $realtobits(10.0); y1 = $realtobits(100.0); z1 = $realtobits(0.0);
        x2 = $realtobits(0.0); y2 = $realtobits(0.0); m2 = $realtobits(0.0); //
Example values
       # (CLK_PERIOD * (1)); // Wait for output + margin
       @(posedge clk);
        x1 = $realtobits(10.0); y1 = $realtobits(-10.0); z1 = $realtobits(0.0);
        x2 = $realtobits(10.0); y2 = $realtobits(-10.0); m2 = $realtobits(0.0);
// Example values
       # (CLK_PERIOD * (1)); // Wait for output + margin
       @(posedge clk);
        x1 = $realtobits(-100.0); y1 = $realtobits(10.0); z1 =
$realtobits(0.0);
        x2 = $realtobits(0.0); y2 = $realtobits(0.0); m2 = $realtobits(200.0);
// Example values
       # (CLK_PERIOD * (1)); // Wait for output + margin
       @(posedge clk);
        x1 = $realtobits(0.0); y1 = $realtobits(0.0); z1 = $realtobits(0.0);
        x2 = $realtobits(0.0); y2 = $realtobits(0.0); m2 = $realtobits(0.0); //
Example values
       # (CLK_PERIOD * (1)); // Wait for output + margin
       @(posedge clk);
```

```
        // --- Test Case 2 ---
        $display("Time=%t: Applying Test Case 2", $time);
        x1 = $realtobits(5.0); y1 = $realtobits(5.0); z1 = $realtobits(0.0);
        x2 = $realtobits(-5.0); y2 = $realtobits(-5.0); m2 =
$realtobits(200.0); // Example values
        # (CLK_PERIOD * (LATENCY + 10)); // Wait for output + margin
       @(posedge clk);

        // --- Test Case 3 ---
        $display("Time=%t: Applying Test Case 3", $time);
        x1 = $realtobits(1.0); y1 = $realtobits(2.0); z1 = $realtobits(3.0);
        x2 = $realtobits(4.0); y2 = $realtobits(5.0); m2 = $realtobits(50.0);
// Example values
        # (CLK_PERIOD * (LATENCY + 10)); // Wait for output + margin
       @(posedge clk);

        $display("Time=%t: Simulation finished.", $time);
        $finish;
    end

    // Monitoring (Optional: Can be verbose)
    // initial begin
    //     $monitor("Time=%t rst=%b x1=%d y1=%d x2=%d y2=%d m2=%d -> ax=%d
ay=%d",
    //              $time, rst, x1, y1, x2, y2, m2, ax, ay);
    // end

endmodule
```

nBodyTB.sv

```
C/C++
`timescale 1 ps / 1 ps
module nbodyTb;

    // Parameters
    parameter CLK_PERIOD = 10; // Clock period in ns (100 MHz)
    parameter MultTime = 11;   // Match DUT parameter
```

```
    parameter AddTime = 20;    // Match DUT parameter
    parameter InvSqrtTime = 27;// Match DUT parameter
    // Total latency = AddTime + MultTime + AddTime + InvSqrtTime + MultTime +
MultTime + MultTime = 122 cycles
    parameter LATENCY = 122;

    logic clk, rst;
    logic [63:0] writedata, readdata, tmpdata;
    logic read, write;
    logic [15:0] addr;
    logic chipselect;
    nbody dut  (
        .clk(clk),
        .rst(rst),
        .writedata(writedata),
        .readdata(readdata),
        .read(read),
        .write(write),
        .addr(addr),
        .chipselect(chipselect)
    );
    // --- Define arrays for body data (size 22) ---
    localparam NUM_BODIES_TO_INIT = 25;
    localparam DEFINED_BODIES = 3;
    logic [31:0] read_register;

    real x_coords[NUM_BODIES_TO_INIT];
    real y_coords[NUM_BODIES_TO_INIT];
    real vx_coords[NUM_BODIES_TO_INIT];
    real vy_coords[NUM_BODIES_TO_INIT];
    real mass_values[NUM_BODIES_TO_INIT];

    // Define select codes for the upper address bits (assuming ADDR_WIDTH=16,
BODY_ADDR_WIDTH=9)
    // These values should match what your nbody.sv expects for addr[15:9]
    // Registers:
    localparam GO             = 7'h00;
    localparam READ           = 7'h01;
    localparam N_BODIES       = 7'h02;
    localparam GAP            = 7'h03;
    // Memory:
    localparam X_SEL_LOWER    = 7'h04;
    localparam X_SEL_UPPER    = 7'h05;
    localparam Y_SEL_LOWER    = 7'h06;
```

```
    localparam Y_SEL_UPPER    = 7'h07;
    localparam M_SEL_LOWER    = 7'h08;
    localparam M_SEL_UPPER    = 7'h09;
    localparam VX_SEL_LOWER   = 7'h10;
    localparam VX_SEL_UPPER   = 7'h11;
    localparam VY_SEL_LOWER   = 7'h12;
    localparam VY_SEL_UPPER   = 7'h13;
    // Out:
    localparam DONE           = 7'b1000000;
    localparam READ_X_LOWER   = 7'b1000001;
    localparam READ_X_UPPER   = 7'b1000010;
    localparam READ_Y_LOWER   = 7'b1000011;
    localparam READ_Y_UPPER   = 7'b1000100;
    // Adjust BODY_ADDR_WIDTH if your nbody module supports a different number
of bodies than 512
    localparam BODY_ADDR_WIDTH = 9;

    initial begin
        clk = 0;
        forever #(CLK_PERIOD / 2) clk = ~clk;
    end

    initial begin
        // For better time printing
        rst = 1;
        # (CLK_PERIOD * 5);
        rst = 0;
        $display("Time=%t: Deasserting reset.", $time);
        // Initialize inputs
        // # CLK_PERIOD;
        // @posedge clk;
        // Writing N_BODIES
        chipselect = 1;
        write = 1;
        read = 0;
        addr = (N_BODIES << BODY_ADDR_WIDTH);
        writedata = 64'd25;

        # (CLK_PERIOD); // Wait a cycle before starting the new sequence



        // --- Populate the first 3 elements with specific values ---
        // Body 0
```

```
        x_coords[0] = 1.0; y_coords[0] = 10.0; vx_coords[0] = 0.1; vy_coords[0]
= 0.0; mass_values[0] = 1000.0;
        // Body 1
        x_coords[1] = -5.0; y_coords[1] = -15.0; vx_coords[1] = 0.0;
vy_coords[1] = -0.05; mass_values[1] = 500.0;
        // Body 2
        x_coords[2] = 20.0; y_coords[2] = 0.0; vx_coords[2] = -0.2;
vy_coords[2] = 0.2; mass_values[2] = 2000.0;

        // --- Populate the rest with 0s ---
        for (int i = DEFINED_BODIES; i < NUM_BODIES_TO_INIT; i++) begin
            x_coords[i] = 0.0;
            y_coords[i] = 0.0;
            vx_coords[i] = 0.0;
            vy_coords[i] = 0.0;
            mass_values[i] = 0.0; // Or a very small non-zero mass if 0 mass is
problematic
        end

        // --- Write these values into the device in a loop ---
        $display("Time=%t: Starting to write %0d body initial conditions.",
$time, NUM_BODIES_TO_INIT);
        for (int i = 0; i < NUM_BODIES_TO_INIT; i++) begin
            // Write Lower X coordinate
            @(posedge clk);
            chipselect = 1;
            write      = 1;
            read       = 0;
            addr       = (X_SEL_LOWER << BODY_ADDR_WIDTH) | i;
            tmpdata = $realtobits(x_coords[i]);
            writedata  = tmpdata[31:0];
            $display("Time=%t: Writing Body Lower %0d X: Addr=0x%h, Data=%f",
$time, i, addr, x_coords[i]);

            // Write Upper X coordinate
            @(posedge clk);
            addr       = (X_SEL_UPPER << BODY_ADDR_WIDTH) | i;
            writedata  = tmpdata[63:32];
            $display("Time=%t: Writing Body Upper %0d X: Addr=0x%h, Data=%f",
$time, i, addr, x_coords[i]);

            // Write Lower Y coordinate
            @(posedge clk);
            addr       = (Y_SEL_LOWER << BODY_ADDR_WIDTH) | i;
```

```
            tmpdata = $realtobits(y_coords[i]);
            writedata  = tmpdata[31:0];
            $display("Time=%t: Writing Body Lower %0d Y: Addr=0x%h, Data=%f",
$time, i, addr, y_coords[i]);

            // Write Upper Y coordinate
            @(posedge clk);
            addr       = (Y_SEL_UPPER << BODY_ADDR_WIDTH) | i;
            writedata  = tmpdata[63:32];
            $display("Time=%t: Writing Body Upper %0d Y: Addr=0x%h, Data=%f",
$time, i, addr, y_coords[i]);

            // Write Lower VX coordinate
            @(posedge clk);
            addr       = (VX_SEL_LOWER << BODY_ADDR_WIDTH) | i;
            tmpdata = $realtobits(vx_coords[i]);
            writedata  = tmpdata[31:0];
            $display("Time=%t: Writing Body Lower %0d VX: Addr=0x%h, Data=%f",
$time, i, addr, vx_coords[i]);

            // Write Upper VX coordinate
            @(posedge clk);
            addr       = (VX_SEL_UPPER << BODY_ADDR_WIDTH) | i;
            writedata  = tmpdata[63:32];
            $display("Time=%t: Writing Body Upper %0d VX: Addr=0x%h, Data=%f",
$time, i, addr, vx_coords[i]);

            // Write Lower VY coordinate
            @(posedge clk);
            addr       = (VY_SEL_LOWER << BODY_ADDR_WIDTH) | i;
            tmpdata = $realtobits(vy_coords[i]);
            writedata  = tmpdata[31:0];
            $display("Time=%t: Writing Body Lower %0d VY: Addr=0x%h, Data=%f",
$time, i, addr, vy_coords[i]);

            // Write Upper VY coordinate
            @(posedge clk);
            addr       = (VY_SEL_UPPER << BODY_ADDR_WIDTH) | i;
            writedata  = tmpdata[63:32];
            $display("Time=%t: Writing Body Upper %0d VY: Addr=0x%h, Data=%f",
$time, i, addr, vy_coords[i]);

            // Write Lower Mass
            @(posedge clk);
```

```
            addr       = (M_SEL_LOWER << BODY_ADDR_WIDTH) | i;
            tmpdata = $realtobits(mass_values[i]);
            writedata  = tmpdata[31:0];
            $display("Time=%t: Writing Body Lower %0d Mass: Addr=0x%h,
Data=%f", $time, i, addr, mass_values[i]);

            // Write Upper Mass
            @(posedge clk);
            addr       = (M_SEL_UPPER << BODY_ADDR_WIDTH) | i;
            writedata  = tmpdata[63:32];
            $display("Time=%t: Writing Body Upper %0d Mass: Addr=0x%h,
Data=%f", $time, i, addr, mass_values[i]);

            // Hold last write for a cycle for the DUT to see it
            #CLK_PERIOD;
        end

        /*
        // After writing all values, de-assert control signals
        @(posedge clk);
        chipselect = 0;
        write      = 0;
        addr       = 0;
        writedata  = 0;
        $display("Time=%t: Finished writing body initial conditions.", $time);
        */

        // Assign gap value
        @(posedge clk);
            chipselect = 1'b1;
            write      = 1'b1;
            addr       = (GAP << BODY_ADDR_WIDTH);
            writedata  = 64'd6;

        //  Turn go on
        @(posedge clk);
            chipselect = 1'b1;
            write      = 1'b1;
            addr       = (GO << BODY_ADDR_WIDTH);
            writedata  = 64'b1;



        # (CLK_PERIOD * 6000); // Wait for simulation to run for a while
```

```
        $display("Time=%t: Testbench finishing.", $time);

        @(posedge clk);
            chipselect = 1'b1;
            write      = 1'b1;
            writedata  = 1'b1;
            read       = 1'b0;
            addr       = (READ << BODY_ADDR_WIDTH);

        for (int i = 0; i < 3; i++) begin

            $display("For body %i", i);
            @(posedge clk);
                chipselect = 1'b1;
                write      = 1'b0;
                writedata  = 1'b0;
                read       = 1'b1;
                addr       = (READ_X_LOWER << BODY_ADDR_WIDTH) | i;

            @(posedge clk);
                read_register = readdata;

            @(posedge clk);
                addr       = (READ_X_UPPER << BODY_ADDR_WIDTH) | i;

            @(posedge clk);
            $display("X = %f", $bitstoreal({readdata,read_register}));

            @(posedge clk);
                addr       = (READ_Y_LOWER << BODY_ADDR_WIDTH) | i;

            @(posedge clk);
                read_register = readdata;

            @(posedge clk);
                addr       = (READ_Y_UPPER << BODY_ADDR_WIDTH) | i;

            @(posedge clk);
            $display("Y = %f", $bitstoreal({readdata,read_register}));
        end
        @(posedge clk);
        @(posedge clk);
            chipselect = 1'b1;
            write      = 1'b1;
```

```
    addr       = (READ << BODY_ADDR_WIDTH);
    writedata  = 64'b0;
# (CLK_PERIOD * 6000); // Wait for simulation to run for a while
$display("Time=%t: Testbench finishing for Gap 2.", $time);

@(posedge clk);
    chipselect = 1'b1;
    write      = 1'b1;
    writedata  = 1'b1;
    read       = 1'b0;
    addr       = (READ << BODY_ADDR_WIDTH);

for (int i = 0; i < 3; i++) begin
    $display("For body %i", i);
    @(posedge clk);
        chipselect = 1'b1;
        write      = 1'b0;
        writedata  = 1'b0;
        read       = 1'b1;
        addr       = (READ_X_LOWER << BODY_ADDR_WIDTH) | i;

    @(posedge clk);
        read_register = readdata;

    @(posedge clk);
        addr       = (READ_X_UPPER << BODY_ADDR_WIDTH) | i;

    @(posedge clk);
        $display("X = %f", $bitstoreal({readdata,read_register}));

    @(posedge clk);
        addr       = (READ_Y_LOWER << BODY_ADDR_WIDTH) | i;

    @(posedge clk);
        read_register = readdata;

    @(posedge clk);
        addr       = (READ_Y_UPPER << BODY_ADDR_WIDTH) | i;

    @(posedge clk);
    $display("Y = %f", $bitstoreal({readdata,read_register}));


end
```

```
        @(posedge clk);
        @(posedge clk);
            chipselect = 1'b1;
            write      = 1'b1;
            addr       = (READ << BODY_ADDR_WIDTH);
            writedata  = 64'b0;
        $finish;

    end

endmodule
```

displayTb.sv

```
C/C++
`timescale 1 ps / 1 ps

module displayTb;

    // Parameters
    parameter CLK_PERIOD = 20; // Clock period in ns (50 MHz)
    parameter DISPLAY_WIDTH = 640;
    parameter DISPLAY_HEIGHT = 480;
    parameter CHECKER_SIZE = 20;

    // Testbench Signals
    logic clk;
    logic reset;
    logic [31:0] writedata;
    logic write;
    logic chipselect;
    logic [14:0] address;

    // VGA outputs
    logic [7:0] VGA_R, VGA_G, VGA_B;
    logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n;

    // Instantiate the DUT
    vga_ball dut (
        .clk(clk),
        .reset(reset),
```

```
        .writedata(writedata),
        .write(write),
        .chipselect(chipselect),
        .address(address),
        .VGA_R(VGA_R),
        .VGA_G(VGA_G),
        .VGA_B(VGA_B),
        .VGA_CLK(VGA_CLK),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .VGA_BLANK_n(VGA_BLANK_n),
        .VGA_SYNC_n(VGA_SYNC_n)
    );

    // Clock Generation
    initial begin
        clk = 0;
        forever #(CLK_PERIOD / 2) clk = ~clk;
    end

    // Calculate checkerboard pattern value for a given position
    function logic get_expected_pixel(int x, int y);
        return ((x / CHECKER_SIZE) % 2 == (y / CHECKER_SIZE) % 2) ? 1'b1 :
1'b0;
    endfunction

    // Main test sequence
    initial begin
        // Declare variables that will be used in the loops
        logic [31:0] pattern_word;
        int addr;
        int pass_count;
        int fail_count;
        logic verbose;
        logic expected;
        logic actual;
        int x, y, bit_idx, i;

        // Initialize signals
        reset = 1;
        write = 0;
        chipselect = 0;
        address = 0;
        writedata = 0;
```

```
        // Apply reset
        #(CLK_PERIOD * 5);
        reset = 0;
        $display("Time=%t: Deasserting reset", $time);

        // Wait for reset to propagate
        #(CLK_PERIOD * 10);

        // Test: Write checkerboard pattern to framebuffer
        $display("Time=%t: Starting to write checkerboard pattern to
framebuffer", $time);

        // Loop through pixel rows in blocks of 32 pixels (one word)
        for (y = 0; y < DISPLAY_HEIGHT; y++) begin
            // Loop through pixel columns in blocks of 32 pixels
            for (x = 0; y < DISPLAY_WIDTH; x += 32) begin
                // Construct a 32-bit word with the appropriate checkerboard
pattern
                pattern_word = 0;

                for (bit_idx = 0; bit_idx < 32; bit_idx++) begin
                    if (x + bit_idx < DISPLAY_WIDTH) begin
                        // Set the bit according to checkerboard pattern
                        if (get_expected_pixel(x + bit_idx, y))
                            pattern_word[bit_idx] = 1'b1;
                    end
                end

                // Calculate the address for this 32-bit word
                addr = (y * (DISPLAY_WIDTH / 32)) + (x / 32);

                // Write the word to the framebuffer
                @(posedge clk);
                chipselect = 1;
                write = 1;
                address = addr[14:0];
                writedata = pattern_word;

                // Print progress for certain positions
                if (y % 40 == 0 && x == 0) begin
                    $display("Time=%t: Writing row %0d, pattern word = %h",
$time, y, pattern_word);
                end
```

```
                @(posedge clk);
            end
        end

        // Release control signals
        chipselect = 0;
        write = 0;

        $display("Time=%t: Finished writing checkerboard pattern", $time);

        // Wait a moment for the pattern to stabilize in the display
        #(CLK_PERIOD * 100);

        // Start comprehensive verification
        $display("Time=%t: Starting comprehensive verification of checkerboard
pattern", $time);

        // Initialize tracking variables
        pass_count = 0;
        fail_count = 0;

        // Sample a subset of pixels for verification (every 20th pixel)
        // This reduces simulation time while still providing good coverage
        for (y = 0; y < DISPLAY_HEIGHT; y += 5) begin
            for (x = 0; x < DISPLAY_WIDTH; x += 5) begin
                // Use existing verification mechanism but with less logging
                expected = get_expected_pixel(x, y);

                // Force DUT to display this point
                force dut.hcount = {x[9:0], 1'b0}; // Properly formatting
11-bit hcount
                force dut.vcount = y;

                // Allow time to propagate
                #(CLK_PERIOD * 1);

                // Check result (VGA_B is what's being checked based on
display.sv)
                actual = (VGA_B == 8'hFF) ? 1'b1 : 1'b0;
                if (actual == expected) begin
                    pass_count++;
                    // Only print occasionally for progress indication
                    if (x % 100 == 0 && y % 100 == 0)
```

```
                    $display("PASS: Pixel (%0d,%0d)", x, y);
                end else begin
                    fail_count++;
                    // Always print failures
                    $display("FAIL: Pixel (%0d,%0d) - expected %0b, got %0b",
                            x, y, expected, actual);

                    // Detailed debug for failures - print the pattern word for
this region
                    addr = (y * (DISPLAY_WIDTH / 32)) + (x / 32);
                    bit_idx = x % 32;
                    $display("  Debug: addr=0x%h, bit_pos=%0d, pattern_word
used=%h",
                            addr, bit_idx, pattern_word);
                end

                // Release the forced values
                release dut.hcount;
                release dut.vcount;
            end

            // Print progress every few rows
            if (y % 50 == 0) begin
                $display("Verified up to row %0d, %0d/%0d pixels passed
(%0.2f%%)",
                        y, pass_count, pass_count + fail_count,
                        (pass_count * 100.0) / (pass_count + fail_count));
            end
        end

        $display("\n===== CHECKERBOARD VERIFICATION SUMMARY =====");
        $display("Total pixels verified: %0d", pass_count + fail_count);
        $display("Passed: %0d", pass_count);
        $display("Failed: %0d", fail_count);
        $display("Pass rate: %0.2f%%", (pass_count * 100.0) / (pass_count +
fail_count));

        if (fail_count == 0)
            $display("VERIFICATION PASSED: All checked pixels match expected
checkerboard pattern");
        else
            $display("VERIFICATION FAILED: %0d pixels did not match expected
pattern", fail_count);
```

```
        $display("Time=%t: Simulation complete", $time);
        $finish;
    end

endmodule
```