Fast N-Body Simulation

6.111 Final Project

Kade Phillips & Scott McCuen

15 December 2017

Table of Contents

Project Abstract

Technical Description
- Simulation (Kade)
  - Mathematical Basis of n-Body Simulation
  - Fixed-Point Arithmetic
  - Resource Minimization and Pipelining
- Display (Scott)
  - BRAM Framebuffers
  - Module 1: sim_data_receive
  - Module 2: sim_to_frame_pos
  - Module 3: frame_pos_to_frame
  - Module 4: frame1_to_frame2
  - Module 5: frame2_to_screen
  - Module 6: vga

Notable Problems and Solutions
- Limitations in FPGA Resources
- Difficulties with Memory
- Timing Issues

Lessons Learned and Future Work

Appendices
- Appendix A: Distributed Processing Unit (Verilog)
- Appendix B: Core Instantiation (Ruby)
- Appendix C: Display Module (Verilog)
- Appendix D: Reference Implementation (C)
- Appendix E: Calculation Verification (Ruby)

Project Abstract

Our project is a demonstration of an FPGA's strengths as parallelized, application-specific hardware, by performing a computation that showcases these properties: direct *n*-body simulation. On a general purpose processor, direct *n*-body simulation has a time complexity of $O(n^2)$, but on an FPGA this can be done with a time complexity of $O(n)$ for reasonable *n* (about 100 on a Virtex-7).

FPGAs seem to be used frequently as accelerators for N-body simulation, but we are not aware of an FPGA being used for the computation in its entirety. We expect that our FPGA implementation would outperform an equivalent program running on a mid-range processor by one to two orders of magnitude for $n > 1000$, if it were possible to synthesize at such scales.

The project is divided into two logically sequential parts: calculation and display. The simulation takes place in a box one million light-years across, with 64 particles. There is one register+arithmetic unit per particle. Wiring each unit to every other unit would reduce the problem to constant time but this requires impossibly dense routing. Our solution is to use a bus that communicates the position of each particle every timestep. To reduce the footprint of each unit (and headache for us), we have used fixed-point operations instead of floating-point. With a 64 MHz clock, the simulation runs at ~16 000 timesteps per second.

Technical Description

Simulation (Kade)

Mathematical Basis of *n*-Body Simulation

The motion of mass observed at a galactic scale does not agree with predictions based on classical mechanics. The currently accepted explanation for the discrepancy is the existence of additional, dark matter. One alternative proposal, however, instead modifies Newtonian dynamics. The following equations describe the rules of such systems:

Gravitation	$F = GMm \div r^2$
Modified Inertia	$F = ma^2 \div a_0$
⇒ Acceleration	$a = (GMa_0)^{½} \div r$

*where **F** is force, **G** is the gravitational constant, **M** is the mass of an interacting particle, **r** is the distance between the particles, **a** is acceleration, and **a_0** is a constant*

The problem of *n*-body simulation requires computing, for every particle, the total force acting on the particle, and then calculating its position **p** over time according to the basic differential equation

Motion	$\ddot{p} = a$

This has no analytic solution for *n* above small values, and so numerical methods are required. One especially straightforward stable method is called Verlet integration:

$$p_{t+1} = 2p_t - p_{t-1} + a\Delta t^2$$

*where **Δt** is the length of a simulation timestep*

Here, **a** is actually the result of summing all forces acting on the particle, and so when we make the assumption that all particles in the simulation are of equal mass, it is equal to

$$a = \Sigma\, (GMa_0)^{½} \div r_j$$

*for all other particles **j***

Conveniently, the constant under the sum may be taken out and combined with the $\mathbf{\Delta t^2}$ term, and we obtain

$$p_{t+1} = 2p_t - p_{t-1} + k \times \Sigma(1 \div r_j)$$

where k is a constant equal to $(GMa_0)^{1/2}\Delta t^2$

Chosen carefully, this constant **k** can be a power of two, and so multiplication simply becomes a left or right shift of binary representation; the same is true of the multiplication of $\mathbf{p_t}$ by two. This reduces Verlet integration to simple addition and subtraction, which is fast and requires comparatively little hardware, and is trivial to implement in Verilog.

For a particle whose position coordinates are $\mathbf{p} = \langle \mathbf{x}, \mathbf{y}, \mathbf{z} \rangle$, the value of $\mathbf{1 \div r_j}$ is calculated as

$$dx = x_j - x$$
$$dy = y_j - y$$
$$dz = z_j - z$$

$$|r_j| = (dx^2 + dy^2 + dz^2)^{1/2}$$

$$1 \div r_j = \langle\, dx \div |r_j|^2 ,\, dy \div |r_j|^2 ,\, dz \div |r_j|^2 \,\rangle$$

where $p_j = \langle\, x_j, y_j, z_j \,\rangle$ *is the position of an interacting particle*

so that the x coordinate of $\mathbf{1 \div r_j}$, for instance, is equal to $dx \div (dx^2+dy^2+dz^2)$, since the square cancels the root.

Fixed-Point Arithmetic

In order to conserve hardware resources, fixed-point arithmetic is used instead of floating-point. This requires meticulous and careful planning to minimize the loss of information. If two operands are *u* and *v* bits wide respectively, their sum may be up to $1+\max\{u, v\}$ bits wide and their product may be up to $u+v$ bits wide. If this exceeds width constraints, the results will have to be truncated by selecting a range of bits to keep; the rest are discarded. This also may necessitate relocating the fixed point, which has to be considered in later calculations.

For example, if two operands are 8-bit numbers typically in the ranges 0–127 and 32–63, their product will typically be in the range 0–8001:

$$_{7}0uuu\ uuuu_{0} \times {}_{7}00vv\ vvvv_{0} = {}_{15}000p\ pppp\ pppp\ pppp_{0}$$

If the result needs to be coerced into 8-bits, $_{12}p\ pppp\ ppp_{5}$ should be kept. 5 bits of information are lost in the process and the fixed point has been relocated from the 0's place to the 32's ($=2^5$) place, as a result of $_{12}0\ 0000\ 001_{5}$ would represent the number 32.

In order to make these sorts of decisions, the floating-point reference C implementation was used to print the minimum and maximum values encountered in practice of several intermediate calculations:

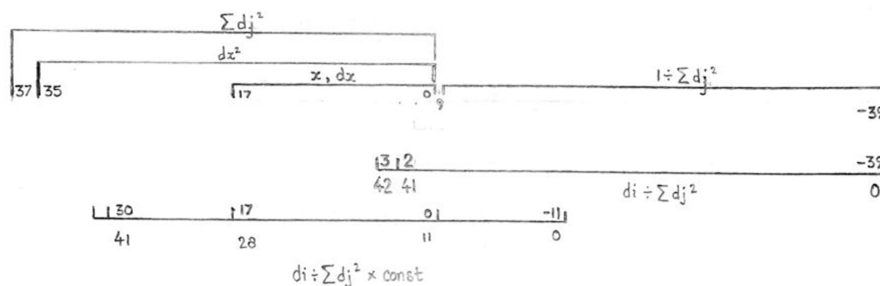
	min $\log_2(min)$	***max*** $\log_2(max)$	$\log_2(max - min)$	$\log_2(max \div min)$
dx, dy, *or* dz	3.6 E −2 −4.8	4.3 E +9 32.0	32.0	36.8
$\Sigma_j\ dj^2 = dx^2+dy^2+dz^2$	4.6 E +9 32.1	2.2 E +19 64.3	64.3	32.2
$di \div \Sigma_j\ dj^2$ *where i is x, y, or z*	1.4 E −20 −66.0	1.2 E −5 −16.3	−16.3	49.6
$\Sigma\,(di \div \Sigma_j\ dj^2)$	2.2 E −15 −48.7	1.2 E −5 −16.3	−16.3	32.3
Δx, Δy, *or* Δz	5.8 E −5 −14.1	3.2 E +5 18.3	18.3	32.4
x, y, *or* z	1.3 E +7 23.6	4.9 E +9 32.2	32.2	8.6

These observations were used to make the right-hand columns of the following table, which were the guidelines used during the implementation in Verilog. They can be compared with theoretical minimum and maximum values, shown in the left-hand columns.

	theoretical		observed in practice, $k = 1280$		
	range of values	bits in representation	range	bits	
$x, y,$ or z	$0..n$	N	$0..n$	N	
$dx, dy,$ or dz	$1..n^{*}$ ±	N	$1..n$ ±	N	
$\sum_j dj^2$	$1..3n^2$	$\sim 2N+1$	$n..n^2$	N	
$di \div \sum_j dj^2$	$\frac{1..n}{1..3n^2} = \frac{1}{3n^2}..n$ ±	$\sim 3N+1$	$\frac{1..n}{n..n^2} = \frac{1}{n^2}..1$ ±	$2N$	calculated
			$\frac{1}{n^2}..\frac{1}{\sqrt{n}}$ ±	$\frac{3}{2}N$	measured
$\sum_k di \div \sum_j dj^2$	$0..kn$ ±	$N \log k$	$\frac{1}{n^{3/2}}..\frac{1}{\sqrt{n}}$ ±	N	when $k = 1280$
$\sqrt{GMa_0}\,\Delta t^2 \sum_k \cdots$	$0..kn \times const$ ±	$N \log k \times const$	-	N	

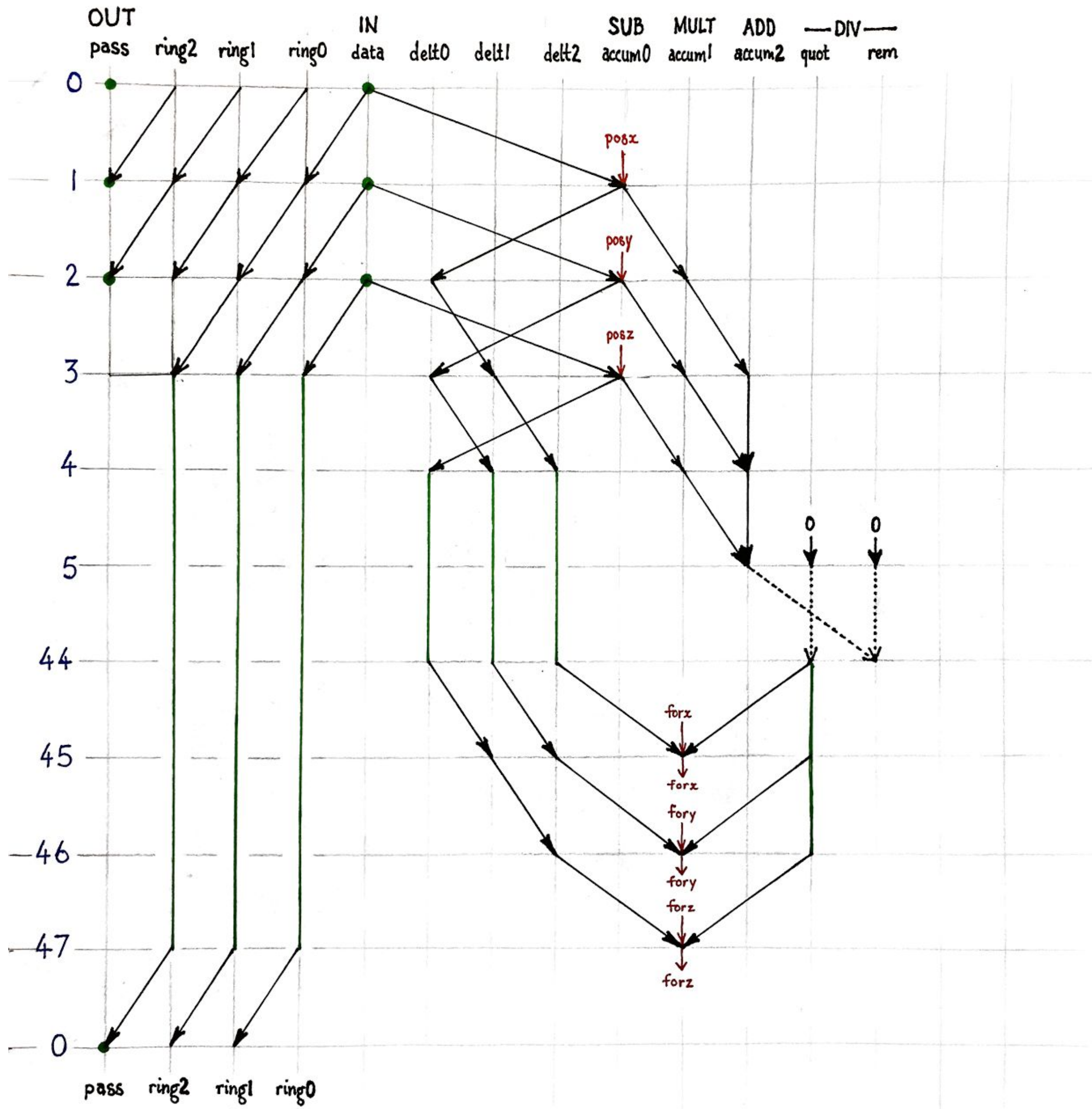
* Actually 0, but makes division intractable   ± result is signed k = N° of particles

To keep track of the fixed-point locations as well as the width of various intermediate calculations, the following diagram was also drawn:



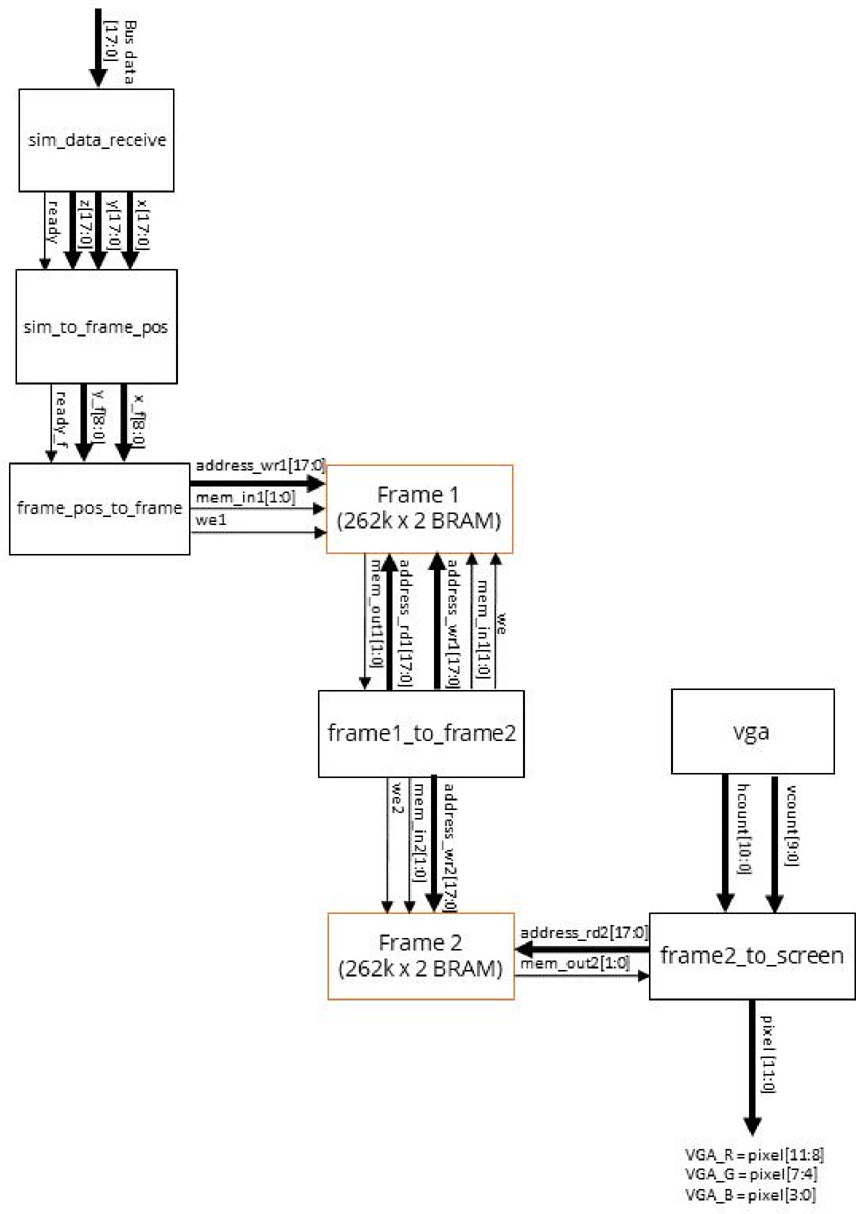
Resource Minimization and Pipelining

A distributed processing unit makes use of only one multiplier, one adder, and one subtractor, and pipelines it computations in order to do so. The diagram on the following page illustrates the flow and timing of data within the processing unit.


MAIN LOOP TIMING
6.111 DISTRIBUTED PROCESSING UNIT
OUT
pass
ring2
ring1
ring0
IN
data
delt0
delt1
delt2
SUB
accum0
MULT
accum1
ADD
accum2
DIV
quot
rem
posx
posy
posz
forx
fory
forz
0
1
2
3
4
5
44
45
46
47
0
pass
ring2
ring1
ring0

Display (Scott)

The display portion of the project is broken up into six submodules instantiated within one top level module. Verilog for these modules is provided in Appendix C. Essentially, the six modules read data from an 18-bit bus between the distributed processing units and output the data on a 1024 by 768 display through VGA. The function of each module is described in more detail below, and an overall block diagram is provided in the figure below.



Overall block diagram of display modules

BRAM Framebuffers

To store the data that we intend to display on the screen, we created two 262k x 2 block RAMs. This gives each pixel in the 512 by 512 frame a unique address with two bits of information, allowing us to generate four different colors. In order to enable constant reading and writing from each memory while avoiding collisions, we implemented each framebuffer as a "simple dual-port" BRAM. Each BRAM has a Write port (port A) and a Read port (port B), and the two ports have separate address lines. See the connections that modules have to the BRAMs in the figure above.

Module 1: `sim_data_receive`

The purpose of `sim_data_receive` is to extract positional data from the 18 bit bus that connects distributed processing units. The module is a simple FSM (4 states) with a counter to ensure that its timing matches the timing of the DSPs. Data on the DSP bus arrives sequentially as X, Y, and Z. The counter of this module is initialized to line up with the DSP modules. The module starts at clock cycle 4, and at clock cycle 0 it receives X positional data and copies it to an 18 bit bus `x_pos`. On the next two clock cycles, it receives Y and Z and copies them to `y_pos` and `z_pos`. Every 48 clock cycles, the counter resets. After receiving all three axes, the module drives `pos_ready` high to signal that the data on its output buses are valid.

Module 2: `sim_to_frame_pos`

The purpose of `sim_to_frame_pos` is to convert the data on the three 18 bit positional buses from `sim_data_receive` to a position in the display frame. The frame is 512 by 512 pixels, so we require 9 bits to determine an x position and 9 bits to determine a y position. The initial implementation of `sim_to_frame_pos` took only the top 9 bits from the x and y buses and used them to set the frame position directly. The implementation of `sim_to_frame_pos` used in the final version of the project selected the top 9 bits of either x and y, x and z, or y and z based on the state of two switches on the Nexys board. After detecting that `pos_ready` is high (signal from `sim_data_receive`), `sim_to_frame_pos` copies the top 9 bits from the two appropriate positional buses and sets `pos_frame_ready` high to signal to `frame_pos_to_frame` that its output data are valid.

Module 3: `frame_pos_to_frame`

The purpose of `frame_pos_to_frame` is to take the frame position from `sim_to_frame_pos` and create a 3 by 3 pixel particle at that position in the first frame buffer. The module is implemented as a 10 state FSM, because only one particle can be written to the frame buffer at a time. After the module detects that `pos_frame_ready` goes high, it writes the value specified by `PARTICLE_COLOR` to all 9 pixels of the particle. In the frame buffer, pixels are stored sequentially by row. The beginning of the memory represents the pixels in the

top row of the frame from left to right, and each subsequent row follows. Consequently, the position of a pixel that is one row above another pixel is 512 (width of frame) positions earlier in the frame buffer. To create the 3 by 3 pixel particle, the module writes to 3 addresses in each of 3 memory locations that are 512 addresses apart. After the last pixel is written, the module returns to its wait state until it detects valid data from `sim_to_frame_pos`.

Module 4: `frame1_to_frame2`

The purpose of `frame1_to_frame2` is to copy the contents of Framebuffer 1 to Framebuffer 2 and clear the contents of Framebuffer 1 in the process. The module is triggered by the wire `ready` which goes high when `hcount` and `vcount` are both zero, meaning the “beginning” of the screen is being written to (top left). The module starts at this point to ensure that part of Framebuffer 2 has been updated before the module `frame2_to_screen` attempts to read the contents of Framebuffer 2. As long as this module writes to Framebuffer 2 before it is read, there are no conflicts. After it is triggered, the module continues copying the contents of Framebuffer 1 to Framebuffer 2, one address at a time, until it reaches the last address of Framebuffer 1, which is 0x3FFFF. As the module copies data from Framebuffer 1 to Framebuffer 2, it must also erase the contents of Framebuffer 1 to ensure that each frame shows only current data. It does this by writing `FRAME_BACK_COLOR` to Framebuffer 1. If we did not erase the contents, particles would appear to leave trails and the frame would quickly saturate with one color.

Module 5: `frame2_to_screen`

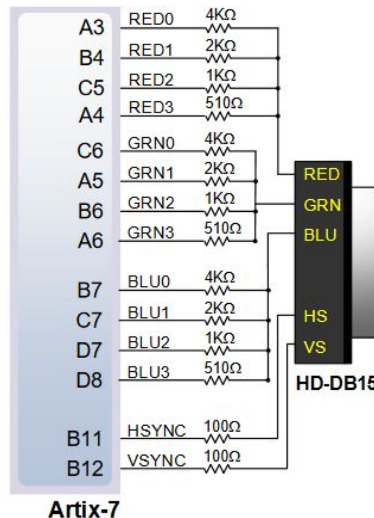
The purpose of `frame2_to_screen` is to translate the contents of Framebuffer 2 to pixel values for display through VGA. The module defines `fx` and `fy` which correspond to the x and y position of a pixel on the screen with the origin at the upper left corner of the frame (the first position in the framebuffers). These wires are used to calculate the appropriate address to read data from. The input wire `in_frame` is high when `hcount` and `vcount` specify a pixel within the boundaries of the frame to be displayed.

Both framebuffers store only two bits at each location in memory, so we can only create 4 colors. This module translates the two bit color specified by data from Framebuffer 2 into a 12 bit value that can be displayed through VGA.If `in_frame` is high, the module assigns a value to `pixel` based on the data read from Framebuffer 2. If `in_frame` is low and `in_border` is high, the module assigns the value specified by `BORDER_COLOR` to `pixel`. If neither wire is high, the module assigns the value specified by `BACK_COLOR` to `pixel`.

Module 6: `vga`

The purpose of `vga` is to generate the signals necessary to display to a 1024 by 768 screen. This is the same module provided in the code template for Lab 4, although it has been modified for a different size screen. In order to generate the 65 MHz clock necessary for

VGA at this resolution, we used the Clocking Wizard in Vivado. Note that VGA on the Nexys 4 uses only 12 bits, as opposed to 24 bits on the Labkit. Rather than using a 24 bit video DAC chip, the Nexys 4 simply mixes the values of the 4 bits used for each color (red, green, blue) using simple resistor ladder. This allows the Nexys to achieve 16 distinct values for each color without using any complicated DAC circuitry. See the figure below for a schematic of this circuitry.



VGA DAC circuitry on Nexys 4

Notable Problems and Solutions

Limitations in FPGA Resources

Throughout the project we encountered limitations imposed by the available resources on the FPGA. We designed our project to be easily scalable, and our goal from the beginning was to make use of a majority of the FPGA's resources. The limitations of the FPGA were most relevant in the design of the simulation portion of the project. Memory was not an issue in this part of the project; each distributed processing unit uses only 537 bits of storage. The primary limitation imposed by the FPGA was due to the number of DSP slices available. The FPGA that we used provided only 240 DSP slices, and our intent was to use one slice for each DPU. Consequently, we had to limit our simulation to 240 particles due to resource constraints alone.

When we synthesized the simulation, we found that Vivado used 2 DSP slices for each DPU rather than 1, so our simulation was limited even further. Additionally, because we were using such a large portion of the FPGA resources, the time to upload our project to the FPGA exceeded 20 minutes, which limited our ability to test out new iterations. Further, we found that Vivado used an unexpectedly large number of LUTs to implement the

distributed processing ring. We dealt with each of these limitations as we encountered them, but future work might involve exploring the true boundaries of the FPGA by changing the way Vivado synthesizes the design.

Difficulties with Memory

Before settling on using Vivado's Block Memory Generator to configure the two BRAM framebuffers, we experimented with a few other strategies. We originally intended to use a larger framebuffer, so we considered using DDR2 rather than BRAM to store each framebuffer. However, after deciding to use only a 512 by 512 pixel framebuffer, we chose to use BRAM for easier configuration.

When we originally implemented BRAM, we attempted to instantiate a module that Vivado would interpret as a BRAM, much like in Lab 5. This worked in the case that we generated a simple, 1 port BRAM. However, we were unable to successfully create a two-port BRAM by simply using Verilog. Vivado would instead used distributed RAM resources to synthesize the two-port memory, consuming valuable resources that we needed to conserve for use by the Distributed Processing Units. Additionally, this use of resources tremendously increased the time required to synthesize our design.

To address these difficulties, we opted to use the Block Memory Generator. This tool allowed us to easily configure and instantiate two simple dual-port 262k x 2 BRAMs for our two framebuffers.

Timing Issues

As with any Verilog project, we ran into timing issues as we attempted to integrate all of our work. We concluded that the only way to minimize timing issues was to diligently divide the project into small, simple modules while very carefully considering the timing relationships between modules. To avoid clocking issues between the modules, we used the 65 MHz clock generated for VGA to clock all modules. Each module that depended on a previous module to complete a particular task had a "ready" wire to indicate a start time. Modules that ran cyclically with the DPU ring used counters to keep track of the 48 clock cycles of the simulation. Any module that executed a particular sequence of events was designed as a simple FSM.

The biggest timing issue we encountered involved reading and writing to the two framebuffers and the display. Because we only had 64 particles in our final simulation, writing to the first framebuffer was relatively fast compared to copying one framebuffer to the other. However, copying between framebuffers and writing to the screen both take substantial amounts of time. To avoid issues between the `frame1_to_frame2` and `frame2_to_screen` modules, we had to ensure that `frame1_to_frame2` started copying

contents over to Framebuffer 2 well before `frame2_to_screen` attempted to read those contents.

Lessons Learned and Future Work

We learned a number of important lessons during this project, some technical and some not so technical. One of the biggest realizations we had was in the value of simulation. As our project increased in size, synthesis times grew above 20 minutes and we were unable to quickly test changes to our modules. To deal with this, we ended up simulating the entire project in Vivado and stepping through each module over a time span of several seconds. This allowed us to effectively pinpoint bugs in our project and prevent them from propagating through to subsequent modules.

We also learned that it is extremely helpful to break up large, complex projects into many small, simple, easily testable modules. In the case of the simulation, this involved creating many small identical DPUs. Once one DPU was correct, the entire simulation was correct. Further, this allowed us to quickly change the nature of the entire system. In the case of the display code, breaking the project into modules allowed us to easily test if we were manipulating signals as we intended. It also allowed us to entirely change the behavior of the display without risking breaking the entire system. At a higher level, the project was broken into two logically sequential units with only one connection (an 18 bit bus). This greatly simplified integrating our code together and helped us to avoid a lot of unnecessary debugging.

Lastly, we found that writing reference implementations is tremendously helpful in debugging. Because we implemented the simulation in both C and Ruby, we had a specific behavior to aim for. Further, implementing the simulation in these languages helped us to think through details of the simulation that would otherwise lead to complications.

In the future, we would like to increase the size of our simulation beyond the limitations imposed by the FPGA’s DSP slices and implement 3D visualization capabilities as we originally intended.

We found this project to be a challenging and rewarding experience, and we would like to thank the 6.111 staff for a wonderful semester.

Appendices

Appendix A: Distributed Processing Unit (Verilog)

```
module dpu #(parameter LN, INITX, INITY, INITZ, INITPX, INITPY, INITPZ) (input clk, input update, input
[17:0] data, output reg [17:0] pass);

        reg unsigned [17:0] prevposx;
        reg unsigned [17:0] prevposy;
        reg unsigned [17:0] prevposz;
        initial prevposx = INITPX;
        initial prevposy = INITPY;
        initial prevposz = INITPZ;

        reg unsigned [17:0] posx;
        reg unsigned [17:0] posy;
        reg unsigned [17:0] posz;
        initial posx = INITX;
        initial posy = INITY;
        initial posz = INITZ;

        reg signed [42:0] forx;
        reg signed [42:0] fory;
        reg signed [42:0] forz;
        initial forx = 0;
        initial fory = 0;
        initial forz = 0;

        reg unsigned [17:0] ring0;
        reg unsigned [17:0] ring1;
        reg unsigned [17:0] ring2;
        initial ring0 = 0;
        initial ring1 = 0;
        initial ring2 = 0;

        reg negx;
        reg negy;
        reg negz;
        initial negx = 0;
        initial negy = 0;
        initial negz = 0;

        reg unsigned [17:0] accum0;  // SUB      18   bits
        reg unsigned [41:0] accum1;  // MULT     42   bits = 24+18
        reg unsigned [37:0] accum2;  // ADD      36+2 bits
        reg unsigned [38:0] quot;        //      36+3 bits
        reg unsigned [38:0] rem;         //      36+3 bits
        initial accum0 = 0;
        initial accum1 = 0;
        initial accum2 = 0;
        initial quot = 0;
        initial rem = 0;
```

```
        reg [17:0] delt0;
        reg [17:0] delt1;
        reg [17:0] delt2;
        initial delt0 = 0;
        initial delt1 = 0;
        initial delt2 = 0;

        reg [LN-1:0] count;
        initial count = 0;

        reg [5:0] cycle;
        initial cycle = 4;

        always @(posedge clk) begin
        if (count == 0) begin
        if (cycle == 0) begin
                accum0 <= {posx, 1'b0} - {1'b0, prevposx};       // be wary of potential clipping here -
result is (probably) 18 bits, but intermediate is 19 bits
        end
        else if (cycle == 1) begin
                if (update) begin
                        posx <= accum0 + forx[38:21];
                prevposx <= posx;
                end
                forx <= 0;
                accum0 <= {posy, 1'b0} - {1'b0, prevposy};
        end
        else if (cycle == 2) begin
                if (update) begin
                        posy <= accum0 + fory[38:21];
                prevposy <= posy;
                end
                fory <= 0;
                accum0 <= {posz, 1'b0} - {1'b0, prevposz};
        end
        else if (cycle == 3) begin
                if (update) begin
                        posz <= accum0 + forz[38:21];
                prevposz <= posz;
                end
                forz <= 0;
        end
        else if (cycle == 4) begin
                ring0 <= posx;          // x
        end
        else if (cycle == 5) begin
                ring0 <= posy;          // y
                ring1 <= ring0;         // x
        end
        else if (cycle == 6) begin
                ring0 <= posz;          // z
                ring1 <= ring0;         // y
                ring2 <= ring1;         // x
        end
        else if (cycle == 7) begin
                ring1 <= ring0;         // z
                ring2 <= ring1;         // y
```

```
        pass <= ring2;          // setup out x
end
end
else begin
if (cycle == 0) begin
        reg signed [18:0] diff;
        ring0 <= data;
        ring1 <= ring0;
        ring2 <= ring1;
        pass <= ring2;          // setup out y
        diff  = data - posx;    // dx
        accum0 <= (diff<0 ? -diff : diff);
        negx <= (diff<0);
end
else if (cycle == 1) begin
        reg signed [18:0] diff;
        ring0 <= data;
        ring1 <= ring0;
        ring2 <= ring1;
        pass <= ring2;          // setup out z
        delt0 <= accum0;
        diff  = data - posy;    // dy
        accum0 <= (diff<0 ? -diff : diff);
        negy <= (diff<0);
        accum1 <= accum0 * accum0;  // dx*dx
end
else if (cycle == 2) begin
        reg signed [18:0] diff;
        ring0 <= data;
        ring1 <= ring0;
        ring2 <= ring1;
        delt0 <= accum0;
        delt1 <= delt0;
        diff  = data - posz;    // dz
        accum0 <= (diff<0 ? -diff : diff);
        negz <= (diff<0);
        accum1 <= accum0 * accum0;  // dy*dy
        accum2 <= accum1;       // dx*dx
end
else if (cycle == 3) begin
        delt0 <= accum0;
        delt1 <= delt0;
        delt2 <= delt1;
        accum1 <= accum0 * accum0;  // dz*dz
        accum2 <= accum2 + accum1;  // dx*dx + dy*dy
end
else if (cycle == 4) begin
        accum2 <= accum2 + accum1;  // dx*dx + dy*dy + dz*dz;
        quot <= 0;
        rem <= 0;
end
else if (cycle < (5+39)) begin   // (1×2^39)÷Σ  = (1÷Σ)×2^39
        reg unsigned [38:0] nextR;
        nextR[38:0] = {rem[37:0], (cycle==5 ? 1'b1 : 1'b0)};
        if (nextR >= accum2) begin
                rem <= nextR - {1'b0, accum2};
                quot <= {quot[37:0], 1'b1};
```

```
                end
                else begin
                        rem <= nextR;
                        quot <= {quot[37:0], 1'b0};
                end
        end
        else if (cycle == 44) begin
                delt1 <= delt0;
                delt2 <= delt1;
                accum1 <= delt2 * quot[23:0];   // dx/Σ
        end
        else if (cycle == 45) begin
                reg signed [42:0] signed_accum1;
                signed_accum1 = accum1;
                delt2 <= delt1;
                accum1 <= delt2 * quot[23:0];   // dy/Σ
                forx <= (negx ? forx - signed_accum1 : forx + signed_accum1);
        end
        else if (cycle == 46) begin
                reg signed [42:0] signed_accum1;
                signed_accum1 = accum1;
                accum1 <= delt2 * quot[23:0];   // dz/Σ
                fory <= (negy ? fory - signed_accum1 : fory + signed_accum1);
        end
        else if (cycle == 47) begin
                reg signed [42:0] signed_accum1;
                signed_accum1 = accum1;
                forz <= (negz ? forz - signed_accum1 : forz + signed_accum1);
                ring1 <= ring0;
                ring2 <= ring1;
                pass <= ring2;          // setup out x
        end
        end
        cycle <= (cycle==47 ? 0 : cycle+1);
        count <= (cycle==47 ? count+1 : count);
        end

endmodule
```

Appendix B: Core Instantiation (Ruby)

```
#!/usr/bin/ruby
LOG_NUM = 6
K = 256

class Numeric
        def fmt
        return self.round.to_s(16).upcase.rjust(5, '0')
        end
end
verilog = File.new("procs.v", "w")
verilog.puts "module network#{2**LOG_NUM} (input clk, input update, output wire [17:0] peek);"
(2**LOG_NUM).times do |core|
        verilog.puts "  wire [17:0] interconnect#{core};"
```

```
end
verilog.puts ""
verilog.puts "  assign peek = interconnect0;"
verilog.puts ""
(2**LOG_NUM).times do |core|
        x_ofs = (rand()-0.5)*0x00800
        y_ofs = (rand()-0.5)*0x00800
        z_ofs = (rand()-0.5)*0x00080
        xinit = 0x20000 + x_ofs
        yinit = 0x20000 + y_ofs
        zinit = 0x20000 + z_ofs
        xprev = xinit + (y_ofs/K)
        yprev = yinit - (x_ofs/K)
        zprev = zinit
        verilog.puts "  dpu #(#{LOG_NUM},"
        verilog.puts "          18'h#{xinit.fmt}, 18'h#{yinit.fmt}, 18'h#{zinit.fmt},"
        verilog.puts "          18'h#{xprev.fmt}, 18'h#{yprev.fmt}, 18'h#{zprev.fmt})"
        verilog.puts "          core#{core} (clk, update, interconnect#{core},
interconnect#{(core+1)%(2**LOG_NUM)});"
end
verilog.puts "endmodule"
verilog.close
```

Appendix C: Display Module (Verilog)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Create Date:
// Design Name:
// Module Name: labkit
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////////

module display(
   input CLK100MHZ,
   input[15:0] SW,
   input BTNC, BTNU, BTNL, BTNR, BTND, CPU_RESETN,
   output[3:0] VGA_R,
   output[3:0] VGA_B,
   output[3:0] VGA_G,
   output[7:0] JA,
```

```
   output VGA_HS,
   output VGA_VS,
   output LED16_B, LED16_G, LED16_R,
   output LED17_B, LED17_G, LED17_R,
   output[15:0] LED,
   output[7:0] SEG,  // segments A-G (0-6), DP (7)
   output[7:0] AN       // Display 0-7
   );
////////////////////////////////////////////////////////////////////////

   // CLOCK GENERATION (65 MHZ)
   wire clk_out1, clk_in1;
   // generate 65MHz clock for VGA
   clk_wiz_0 clk1
        (
        // Clock out ports
        .clk_out1(clk_out1),    // output clk_out1
        // Clock in ports
        .CLK100MHZ(CLK100MHZ));          // input clk_in1

////////////////////////////////////////////////////////////////////////

   // PROCESSING CORES
   reg [5:0] cyclecount;
   reg [5:0] shortcount;
   reg [13:0] longcount;
   initial cyclecount = 4;
   initial shortcount = 0;
   initial longcount = 0;
   always @(posedge clk_out1) begin
        cyclecount <= (cyclecount==47 ? 0 : cyclecount+1);
        shortcount <= (cyclecount==47 ? shortcount+1 : shortcount);
        longcount <= ((cyclecount==47 && shortcount==63) ? (longcount==SW[15:2] ? 0 : longcount+1) :
longcount);
   end

   wire update;
   assign update = (longcount==0);

   wire [17:0] sim_data;
   network64 dpus(clk_out1, update, sim_data);

   // FRAMEBUFFER WIRES AND REGISTERS
   wire we1;
   wire we2;
   wire [17:0] address_rd1, address_wr1, address, address_rd2, address_wr2;
   wire [1:0] mem_in1, mem_in2;
   wire [1:0] mem_out1, mem_out2;

   // FRAMEBUFFER INSTANTIATION
   blk_mem_gen_0 frame1 (
        .clka(clk_out1), // input wire clka
        .wea(we1),      // input wire [0 : 0] wea
        .addra(address_wr1),  // input wire [17 : 0] addra
        .dina(mem_in1),  // input wire [1 : 0] dina
        .clkb(clk_out1), // input wire clkb
        .addrb(address_rd1),  // input wire [17 : 0] addrb
```

```
        .doutb(mem_out1)  // output wire [1 : 0] doutb
   );

   blk_mem_gen_1 frame2 (
        .clka(clk_out1), // input wire clka
        .wea(we2),       // input wire [0 : 0] wea
        .addra(address_wr2),  // input wire [17 : 0] addra
        .dina(mem_in2),  // input wire [1 : 0] dina
        .clkb(clk_out1), // input wire clkb
        .addrb(address_rd2),  // input wire [17 : 0] addrb
        .doutb(mem_out2)  // output wire [1 : 0] doutb
   );


////  instantiate 7-segment display;
//      wire [31:0] data;
//      wire [6:0] segments;
//      display_8hex display(.clk(clock_25mhz),.data(data), .seg(segments), .strobe(AN));
//      assign SEG[6:0] = segments;
//      assign SEG[7] = 1'b1;

        // create wires to connect to board devices

        // link board devices to useful names

//      DEBOUNCE BUTTONS
//      parameter DB_DELAY = 250000;
//      wire hidden_db, brake_db, ignition_db, driver_db, pass_db, reprogram_db, reset;
//      debounce #(.DELAY(DB_DELAY)) db_up(.reset(reset), .clock(clock_25mhz), .noisy(hidden_sw),
.clean(hidden_db));

        // create wires for connecting modules

//
//////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////////
// Verilog to display contents of framebuffer

// VGA WIRES AND INSTANTIATION

        // DEFINE SCREEN AND FRAME SIZE AND LOCATION
        parameter SCREEN_WIDTH = 1024;
        parameter SCREEN_HEIGHT = 768;
        parameter FRAME_WIDTH = 512;
        parameter FRAME_HEIGHT = 512;
        parameter FRAME_X_POS = (SCREEN_WIDTH >> 1) - (FRAME_WIDTH >> 1);
        parameter FRAME_Y_POS = (SCREEN_HEIGHT >> 1) - (FRAME_HEIGHT >> 1);

        parameter BORDER_THICKNESS = 20;
        parameter BORDER_COLOR = 12'hFFF;

        parameter BACK_COLOR = 12'h47F;
        parameter FRAME_BACK_COLOR = 2'b00;

    // INSTANTIATE VGA MODULE
```

```
        wire [10:0] hcount;
        wire [9:0] vcount;
        wire hsync, vsync, at_display_area, in_frame, in_border;
        vga vga1(.vga_clock(clk_out1),.hcount(hcount),.vcount(vcount),
        .hsync(hsync),.vsync(vsync),.at_display_area(at_display_area));

        wire [11:0] rgb, pixel;

        // make red square
//   blob #(.WIDTH(640),.HEIGHT(480),.COLOR(12'hF_FF))
//       square2(.x(0),.y(0),.hcount(hcount),.vcount(vcount),.pixel(rgb));

        // SET BOUNDARIES FOR FRAME AND FRAME BORDER
        assign in_frame = ((hcount >= FRAME_X_POS) && (hcount < FRAME_X_POS + FRAME_WIDTH)
        && (vcount >= FRAME_Y_POS) && (vcount < FRAME_Y_POS + FRAME_HEIGHT));
        assign in_border = ((hcount >= FRAME_X_POS - BORDER_THICKNESS) && (hcount < FRAME_X_POS +
FRAME_WIDTH + BORDER_THICKNESS)
                && (vcount >= FRAME_Y_POS - BORDER_THICKNESS) && (vcount < FRAME_Y_POS + FRAME_HEIGHT +
BORDER_THICKNESS)
                && !in_frame);

        assign VGA_HS = ~hsync;
        assign VGA_VS = ~vsync;
        assign VGA_R = at_display_area ? pixel[11:8] : 0;
        assign VGA_G = at_display_area ? pixel[7:4] : 0;
        assign VGA_B = at_display_area ? pixel[3:0] : 0;

////////////////////////////////////////////////////////////
// INSTANTIATE ALL MODULES BELOW
////////////////////////////////////////////////////////////

// SIM_DATA_RECEIVE

    wire [17:0] x_pos, y_pos, z_pos;
    wire pos_ready;
    sim_data_receive receive(.clk(clk_out1),.data_bus(sim_data),.x_pos(x_pos),
         .y_pos(y_pos),.z_pos(z_pos),.pos_ready(pos_ready));

// SIM_TO_FRAME_POS

    wire [8:0] x_pos_frame, y_pos_frame;
    wire pos_frame_ready;
        assign orientation = SW[1:0];

    sim_to_frame_pos transfer1b(.clk(clk_out1),.x_pos_sim(x_pos),.y_pos_sim(y_pos),.z_pos_sim(z_pos),
         .pos_ready(pos_ready),.x_pos_frame(x_pos_frame),.y_pos_frame(y_pos_frame),
         .pos_frame_ready(pos_frame_ready),.orientation(orientation));

// FRAME_POS_TO_FRAME
        wire wr_permission;
        wire we1_pos, we1_frame;
        wire [1:0] mem_in1_pos, mem_in1_frame;
        wire [17:0] address_wr1_pos, address_wr1_frame;

        assign wr_permission = (vcount > ((SCREEN_HEIGHT >> 1) + (FRAME_HEIGHT >> 1)));
    assign we1 = wr_permission ? we1_pos : we1_frame;
    assign address_wr1 = wr_permission ? address_wr1_pos : address_wr1_frame;
```

```
    assign mem_in1 = wr_permission ? mem_in1_pos : mem_in1_frame;

    frame_pos_to_frame #(.FRAME_HEIGHT(FRAME_HEIGHT)) transfer2(.clk(clk_out1),.x_pos_frame(x_pos_frame),
         .y_pos_frame(y_pos_frame),.pos_frame_ready(pos_frame_ready),.address_wr(address_wr1_pos),
         .mem_in(mem_in1_pos),.we(we1_pos));

// FRAME1_TO_FRAME2

    wire begin_transfer;
        assign begin_transfer = (hcount==1 && vcount==1);

    frame1_to_frame2 #(.FRAME_BACK_COLOR(FRAME_BACK_COLOR))
transfer3(.clk(clk_out1),.ready(begin_transfer),

.mem_out1(mem_out1),.address_rd1(address_rd1),.mem_in1(mem_in1_frame),.address_wr1(address_wr1_frame),
         .we1(we1_frame),.mem_in2(mem_in2),.address_wr2(address_wr2),.we2(we2));



// FRAME2_TO_SCREEN

        frame2_to_screen
#(.BACK_COLOR(BACK_COLOR),.BORDER_COLOR(BORDER_COLOR),.SCREEN_WIDTH(SCREEN_WIDTH),
        .SCREEN_HEIGHT(SCREEN_HEIGHT),.FRAME_WIDTH(FRAME_WIDTH),.FRAME_HEIGHT(FRAME_HEIGHT))
        f2_to_sc(.clk(clk_out1),.mem_out(mem_out2),.hcount(hcount),.vcount(vcount),
        .in_frame(in_frame),.in_border(in_border),.address_rd(address_rd2), .pixel(pixel));

//      always @(posedge clk_out1) begin
//      if(in_frame) begin
//              pixel <= mem_out2[1] ? (mem_out2[0] ? 12'hF00 : 12'h0F0) : (mem_out2[0] ? 12'h00F :
12'h000);
//              address_rd2 <= address_rd2 + 1;
//      end
//      else if (in_border) pixel <= BORDER_COLOR;
//      else pixel <= BACK_COLOR;
//      address_wr2 <= address_wr2 + 1;
//      mem_in2 <= {hcount[9],vcount[9]};
//      end

//      mybram1 #(.LOGSIZE(18),.WIDTH(2)) memory1(.addr(address_rd),
//      .clk(clk_out1),.din(mem_in),.dout(mem_out),.we(we));

//      mybram2 #(.LOGSIZE(12),.WIDTH(2)) memory2(.addr_wr(address_rd2),.addr_rd(address_wr2),
//              .clk(clk_out1),.din(mem_in2),.dout(mem_out2),.we(we));

endmodule

module frame2_to_screen
        #(parameter BACK_COLOR = 12'h47F,
                BORDER_COLOR = 12'hFFF,
                SCREEN_WIDTH = 1024,
                SCREEN_HEIGHT = 768,
                FRAME_WIDTH = 512,
                FRAME_HEIGHT = 512)
        (input wire clk,
        input wire [1:0] mem_out,
        input wire [10:0] hcount,
```

```
        input wire [9:0] vcount,
        input wire in_frame,
        input wire in_border,
        output [17:0] address_rd,
        output [11:0] pixel);

        wire [17:0] fx = hcount - SCREEN_WIDTH/2 + FRAME_WIDTH/2;
        wire [17:0] fy = vcount - SCREEN_HEIGHT/2 + FRAME_HEIGHT/2;
        assign address_rd = fx + (fy<<9) + 1;

        assign pixel = in_frame ? (mem_out[1] ? (mem_out[0] ? 12'hFFF : 12'hF00) : (mem_out[0] ? 12'h00F :
12'h000))
                                : (in_border ? BORDER_COLOR : BACK_COLOR);

endmodule

module vga(input vga_clock,
        output reg [10:0] hcount = 0,    // pixel number on current line
        output reg [9:0] vcount = 0,     // line number
        output reg vsync, hsync,
        output at_display_area);

   // Comments applies to XVGA 1024x768, left in for reference
   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);   // active H  1023, 639
   assign hsyncon = (hcount == 1047);    // active H + FP 1047, 655
   assign hsyncoff = (hcount == 1183);   // active H + fp + sync  1183, 751
   assign hreset = (hcount == 1343);     // active H + fp + sync + bp 1343, 799

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 767);   // active V   767, 479
   assign vsyncon = hreset & (vcount == 776);    // active V + fp   776, 490
   assign vsyncoff = hreset & (vcount == 783);   // active V + fp + sync  783, 492
   assign vreset = hreset & (vcount == 805);     // active V + fp + sync + bp 805, 523

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vga_clock) begin
        hcount <= hreset ? 0 : hcount + 1;
        hblank <= next_hblank;
        hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
        vblank <= next_vblank;
        vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

   end

   assign at_display_area = ((hcount >= 0) && (hcount < 1024) && (vcount >= 0) && (vcount < 768));
```

```
endmodule

// ////////////////////////////////////////////////////////////////////
// //
// // blob: generate rectangle on screen
// //
// ////////////////////////////////////////////////////////////////////
// module blob
   // #(parameter WIDTH = 64,            // default width: 64 pixels
                // HEIGHT = 64,          // default height: 64 pixels
                // COLOR = 24'hFF_FF_FF)  // default color: white
   // (input [10:0] x,hcount,
        // input [9:0] y,vcount,
        // input vsync,
        // output reg [11:0] pixel);

   // always @ * begin
        // if ((hcount >= x && hcount < (x+WIDTH)) &&
      // (vcount >= y && vcount < (y+HEIGHT)))
    // pixel = COLOR;
        // else pixel = 0;
   // end
// endmodule

///////////////////////////////////////////////////////////////////////////////
//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
///////////////////////////////////////////////////////////////////////////////

// module mybram1 #(parameter LOGSIZE=14, WIDTH=1)
                // (input wire [LOGSIZE-1:0] addr,
                // input wire clk,
                // input wire [WIDTH-1:0] din,
                // output reg [WIDTH-1:0] dout,
                // input wire we);
   // // let the tools infer the right number of BRAMs
   // (* ram_style = "block" *)
   // reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
   // always @(posedge clk) begin
        // if (we) mem[addr] <= din;
        // dout <= mem[addr];
   // end
// endmodule

// module mybram2 #(parameter LOGSIZE=14, WIDTH=1)
                // (input wire [LOGSIZE-1:0] addr_wr,
                // input wire [LOGSIZE-1:0] addr_rd,
                // input wire clk,
                // input wire [WIDTH-1:0] din,
                // output reg [WIDTH-1:0] dout,
                // input wire we);
   // // let the tools infer the right number of BRAMs
   // (* ram_style = "block" *)
   // reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
```

```
   // always @(posedge clk) begin
        // if (we) mem[addr_wr] <= din;
        // dout <= (we && addr_rd==addr_wr ? din : mem[addr_rd]);
   // end
// endmodule


/////////////////////////////////////////////////////////////////
// Receives data from simulation bus, 18 bit bus X then Y then Z
/////////////////////////////////////////////////////////////////

module sim_data_receive
        (input wire clk,
        input wire [17:0] data_bus,
        output reg [17:0] x_pos,
        output reg [17:0] y_pos,
        output reg [17:0] z_pos,
        output reg pos_ready);

    parameter WAIT  = 2'b00;
    parameter REC_X = 2'b01;
    parameter REC_Y = 2'b10;
    parameter REC_Z = 2'b11;

    reg [1:0] receive_state = WAIT;
    // SET INITIAL COUNTER VALUE TO MATCH WITH SIMULATION OUTPUT
    reg [5:0] counter = 6'd4;

    always @(posedge clk) begin
         if (counter == 47)
                counter <= 0;
         else
                counter <= counter + 1;

         case (receive_state)
                WAIT: begin
                       pos_ready <= 1'b0;
                        if (counter == 47) receive_state <= REC_X;
                end
                REC_X: begin
                        x_pos <= data_bus;
                        receive_state <= REC_Y;
                end
                REC_Y: begin
                        y_pos <= data_bus;
                        receive_state <= REC_Z;
                end
                REC_Z: begin
                        z_pos <= data_bus;
                        pos_ready <= 1'b1;
                        receive_state <= WAIT;
                end
                default: begin
                        pos_ready <= 1'b0;
                end
         endcase
    end
```

```
endmodule

////////////////////////////////////////////////////////////////
// Takes data from data_receive_FSM in form of three 18 bit buses
// (X, Y, Z), and converts to X and Y within 512 by 512 frame
// can implement more interesting versions time permitting
////////////////////////////////////////////////////////////////

module sim_to_frame_pos
    (input wire clk,
     input wire [17:0] x_pos_sim,
     input wire [17:0] y_pos_sim,
     input wire [17:0] z_pos_sim,
     input wire pos_ready,
     output reg [8:0] x_pos_frame,
     output reg [8:0] y_pos_frame,
     output reg pos_frame_ready,
     input wire [1:0] orientation
     );

    // CALCULATE FRAME POSITION FROM SIM POSITION
    // switch codes for orientation
    parameter XY = 2'b00;
    parameter XZ = 2'b01;
    parameter YZ = 2'b10;

    always @(posedge clk) begin
         // if full 54 bits of X,Y,Z data received by sim_data_receive,
         // shove X and Y into frame and tell frame_pos_to_frame that
         // frame position is ready
         if (pos_ready) begin
                 case(orientation)
                         XY: begin
                                 x_pos_frame <= x_pos_sim[17:9];
                                 y_pos_frame <= y_pos_sim[17:9];
                         end
                         XZ: begin
                                 x_pos_frame <= x_pos_sim[17:9];
                                 y_pos_frame <= z_pos_sim[17:9];
                         end
                         YZ: begin
                                 x_pos_frame <= y_pos_sim[17:9];
                                 y_pos_frame <= z_pos_sim[17:9];
                         end
                         default: begin
                                 x_pos_frame <= x_pos_sim[17:9];
                                 y_pos_frame <= y_pos_sim[17:9];
                         end
                 endcase
                 pos_frame_ready <= 1'b1;
         end
         else pos_frame_ready <= 1'b0;
    end
endmodule

////////////////////////////////////////////////////////////////
```

```
// Takes frame position of particle from sim_to_frame_pos and
// writes to first frame buffer , creating a 3x3 particle
/////////////////////////////////////////////////////////////////

module frame_pos_to_frame
        #(parameter FRAME_HEIGHT = 512)
    (input wire clk,
     input wire [8:0] x_pos_frame,
     input wire [8:0] y_pos_frame,
     input wire pos_frame_ready,
     output reg [17:0] address_wr,
     output reg [1:0] mem_in,
     output reg we
     );

    parameter BEGIN = 4'b1111;
    parameter PIX0 = 4'd0;
    parameter PIX1 = 4'd1;
    parameter PIX2 = 4'd2;
    parameter PIX3 = 4'd3;
    parameter PIX4 = 4'd4;
    parameter PIX5 = 4'd5;
    parameter PIX6 = 4'd6;
    parameter PIX7 = 4'd7;
    parameter PIX8 = 4'd8;
    parameter PARTICLE_COLOR = 2'b11;

    // write pixel color to frame buffer in 3x3 square
    reg [3:0] pix_state = BEGIN;
    always @(posedge clk) begin
         case (pix_state)
                 BEGIN: begin
                         if (pos_frame_ready) begin
                                 we <= 1;
                                 mem_in <= PARTICLE_COLOR;
                                 pix_state <= PIX0;
                         end
                         else we <= 0;
                 end
                 PIX0: begin
                         address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame - FRAME_HEIGHT - 1;
                         pix_state <= pix_state + 1;
                 end
                 PIX1: begin
                         address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame - FRAME_HEIGHT;
                         pix_state <= pix_state + 1;
                 end
                 PIX2: begin
                         address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame - FRAME_HEIGHT + 1;
                         pix_state <= pix_state + 1;
                 end
                 PIX3: begin
                         address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame - 1;
                         pix_state <= pix_state + 1;
                 end
                 PIX4: begin
                         address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame;
```

```
                    pix_state <= pix_state + 1;
                end
                PIX5: begin
                    address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame + 1;
                    pix_state <= pix_state + 1;
                end
                PIX6: begin
                    address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame + FRAME_HEIGHT - 1;
                    pix_state <= pix_state + 1;
                end
                PIX7: begin
                    address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame + FRAME_HEIGHT;
                    pix_state <= pix_state + 1;
                end
                PIX8: begin
                    address_wr <= x_pos_frame + FRAME_HEIGHT*y_pos_frame + FRAME_HEIGHT + 1;
                    pix_state <= BEGIN;
                end
                default: pix_state <= BEGIN;
        endcase
    end
endmodule

/////////////////////////////////////////////////////////////////
// Copies frame 1 into frame 2, clearing contents of frame 1 in the process
/////////////////////////////////////////////////////////////////

module frame1_to_frame2
        #(parameter FRAME_BACK_COLOR = 2'b00)  // default color: black
    (input wire clk,
    input wire ready,           // should be (hcount==0 && vcount==0)
    input wire [1:0] mem_out1,
    output wire [17:0] address_rd1,
    output reg [1:0] mem_in1,
    output reg [17:0] address_wr1,
    output reg we1,
    output reg [1:0] mem_in2,
    output reg [17:0] address_wr2,
    output reg we2);

    // NEED TO SET INITIAL VALUES AND START AT CORRECT TIME

    reg [17:0] addr;
    initial addr = 18'h3FFFF;
    assign address_rd1 = addr;

    always @(posedge clk) begin
        if (ready || (addr!=18'h3FFFF)) begin
        we1 <= 1'b1;
        we2 <= 1'b1;
        mem_in2 <= mem_out1;
        address_wr1 <= addr;
        address_wr2 <= addr;
        mem_in1 <= FRAME_BACK_COLOR;
        addr <= addr+1;
        end
        else begin
```

```
        we1 <= 1'b0;
        we2 <= 1'b0;
        end
    end

endmodule
```

Appendix D: Reference Implementation (C)

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

// 1 unit distance = 0.0596 ly
//              G = 1.559e-13 ly^3/(Ms*yr^2)
//              a0 = 1.261e-11 ly/yr^2

// If all particles have the same mass m, then the acceleration on a particle is
//
//      sqrt(G*m*a0) / r
//
// using modified Newtonian gravity

// Want total mass of simulation to be ~1e12 solar masses (10 000 particles => 100 million solar masses
per particle)
//
// G*a0 = 1.966e-24 ly^4 / (Ms * yr^4)
//
// G*a0 * 1e8 Ms = 1.966e-16 ly^4 / yr^4
//
// sqrt(...) = 1.402e-8 ly^2 / yr^2  =  3.947e-6 u^2 / yr^2  ~  2^-18 u^2/yr^2
//
// so acceleration (in distance units / years^2) = (1/r) >> 18

// Time constant for system is ~1/sqrt(G*totalMass/length^3) = 2.5e9 years
// System can run at ~100 steps per second
// Want time constant to correspond to ~100 seconds
// => timestep ~ 250 000 years ~ 2^18
// Then 14 billion years corresponds to about 9 minutes, which sounds right
// We'll just use Δt = 2^18 years


// The numerical method we will use is Verlet
//
//      x <- 2*x - prev + (Σ(1/r) << 18)
//      prev <- x
//
// where the left shift by 18 corresponds to multiplication by 2^-18 * (2^18)^2 = sqrt(G*a0*m) * Δt^2

// Used in previous version, may come in handy
//   memset(&particles[0][idx].velocity, 0, sizeof(struct vector));

#define N 1280
#define WIDTH 1376
```

```
#define MAX_X 1365
#define MAX_Y 511
#define TRAILS 1

struct vector {long double x, y, z;};
struct particle {struct vector posn, prev;};

int main(void) {
        FILE* disp = fopen("/dev/fb0", "w");
        uint32_t (*disp_buffer) = malloc(WIDTH*(MAX_Y+1) * sizeof(uint32_t));

        fputs("\e[2J", stdout);

        struct particle (*particles)[N] = malloc(2*N*sizeof(struct particle));
        srand(42);
        for(unsigned int idx=0; idx<N; idx++){
        long double x = (1L<<31) + (rand()>>1) - (1L<<29); // 1, 29
        long double y = (1L<<31) + (rand()>>1) - (1L<<29); // 1, 29
        long double z = (1L<<31) + (rand()>>5) - (1L<<25); // 5, 25
        particles[0][idx].posn.x = x;
        particles[0][idx].posn.y = y;
        particles[0][idx].posn.z = z;
        particles[0][idx].prev.x = (y>(1L<<31) ? x+0x200000 : x-0x200000);
        particles[0][idx].prev.y = (x<(1L<<31) ? y+0x200000 : y-0x200000);
        particles[0][idx].prev.z = z + (rand()>>7) - (1L<<23);
        //particles[0][idx].prev.x = x + (rand()>>7) - (1L<<23); //(y>(1L<<31) ? x+0x200000 : x-0x200000);
        //particles[0][idx].prev.y = y + (rand()>>7) - (1L<<23); //(x<(1L<<31) ? y+0x200000 : y-0x200000);
        //particles[0][idx].prev.z = z + (rand()>>7) - (1L<<23);
        }

        struct vector (*forces) = malloc(N*(N-1)/2 * sizeof(struct vector));
        // index = (r**2 - r)/2 + c
        //   for N > r > c >= 0

        long double max_d = 0;
        long double min_d = 7.777e100;
        long double max_s = 0;
        long double min_s = 7.777e100;
        long double max_m = 0;
        long double min_m = 7.777e100;
        long double max_f = 0;
        long double min_f = 7.777e100;
        long double max_p = 0;
        long double min_p = 7.777e100;

        unsigned int timestep = 0;
        unsigned int s = 0;
        while (1){
        for(unsigned int p1=1; p1<N; p1++){
        for(unsigned int p0=0; p0<p1; p0++){
                long double x0 = particles[s][p0].posn.x;
                long double y0 = particles[s][p0].posn.y;
                long double z0 = particles[s][p0].posn.z;

                long double x1 = particles[s][p1].posn.x;
                long double y1 = particles[s][p1].posn.y;
                long double z1 = particles[s][p1].posn.z;
```

```
                long double dx = x1-x0;
                long double dy = y1-y0;
                long double dz = z1-z0;

                max_d = fmaxl(max_d, fmaxl(fabsl(dx), fmaxl(fabsl(dy), fabsl(dz))));
                min_d = fminl(min_d, fminl(fabsl(dx), fminl(fabsl(dy), fabsl(dz))));

                // unit length vector with direction of r: (vec1 - vec2) * 1/r
                // vector with length 1/r: (vec1 - vec2) * 1/r^2
                long double sum_sq = (dx*dx + dy*dy + dz*dz);
                long double r_inv_squared = 1.0/sum_sq;

                max_s = fmaxl(max_s, sum_sq);
                min_s = fminl(min_s, sum_sq);

                unsigned int jdx = p1*(p1-1)/2 + p0;
                forces[jdx].x = dx*r_inv_squared;
                forces[jdx].y = dy*r_inv_squared;
                forces[jdx].z = dz*r_inv_squared;

                max_m = fmaxl(max_m, fmaxl(fabsl(dx*r_inv_squared), fmaxl(fabsl(dy*r_inv_squared),
fabsl(dz*r_inv_squared))));
                min_m = fminl(min_m, fminl(fabsl(dx*r_inv_squared), fminl(fabsl(dy*r_inv_squared),
fabsl(dz*r_inv_squared))));
        }
        }

        printf("\x1B[35;1H%21s %21s %21s %21s %21s %21s\n", "max di", "min di", "max S", "min S", "max q",
"min q");
        printf("%21Le %21Le %21Le %21Le %21Le %21Le\n\n", max_d, min_d, max_s, min_s, max_m, min_m);

        unsigned int px=0;
        for(unsigned int px_y=0; px_y<=MAX_Y; px_y++){
        for(unsigned int px_x=0; px_x<WIDTH; px_x++){
                if(((px_x==0 || px_x==511)&&(px_y<512)) || ((px_y==0 || px_y==511)&&(px_x<512))){
                disp_buffer[px] = 0x00FFFFFF;
                }
                else{
                disp_buffer[px] = TRAILS ? (disp_buffer[px]>>1) : 0;
                }
                px++;
        }
        }

        for(unsigned int p=0; p<N; p++){
        struct vector sum;
        sum.x = 0; sum.y = 0; sum.z = 0;
        for(unsigned int before=0; before<p; before++){
                unsigned int kdx = p*(p-1)/2 + before;
                sum.x -= forces[kdx].x;
                sum.y -= forces[kdx].y;
                sum.z -= forces[kdx].z;
        }
        for(unsigned int after=p+1; after<N; after++){
                unsigned int kdx = after*(after-1)/2 + p;
                sum.x += forces[kdx].x;
```

```
                sum.y += forces[kdx].y;
                sum.z += forces[kdx].z;
        }

        max_f = fmaxl(max_f, fmaxl(fabsl(sum.x), fmaxl(fabsl(sum.y), fabsl(sum.z))));
        min_f = fminl(min_f, fminl(fabsl(sum.x), fminl(fabsl(sum.y), fabsl(sum.z))));

        particles[!s][p].prev   = particles[s][p].posn;
        particles[!s][p].posn.x = particles[s][p].posn.x*2 - particles[s][p].prev.x +
sum.x*262144.0*100000.0;
        particles[!s][p].posn.y = particles[s][p].posn.y*2 - particles[s][p].prev.y +
sum.y*262144.0*100000.0;
        particles[!s][p].posn.z = particles[s][p].posn.z*2 - particles[s][p].prev.z +
sum.z*262144.0*100000.0;

        max_p = fmaxl(max_p, fmaxl(particles[!s][p].posn.x, fmaxl(particles[!s][p].posn.y,
particles[!s][p].posn.z)));
        min_p = fminl(min_p, fminl(particles[!s][p].posn.x, fminl(particles[!s][p].posn.y,
particles[!s][p].posn.z)));

        if(particles[!s][p].posn.x > 0 &&
        particles[!s][p].posn.x < (1L<<32) &&
        particles[!s][p].posn.y > 0 &&
        particles[!s][p].posn.y > 0 &&
        particles[!s][p].posn.z < (1L<<32) &&
        particles[!s][p].posn.z < (1L<<32)){
                unsigned int x_coord = (uint32_t)(particles[!s][p].posn.x) >> 23;
                unsigned int y_coord = 511 - ((uint32_t)(particles[!s][p].posn.y) >> 23);
                unsigned char intensity = ((uint32_t)(particles[!s][p].posn.z) >> 24);
                disp_buffer[y_coord*WIDTH + x_coord] = intensity + (intensity<<8) + (intensity<<16);
        }
        }

        printf("%21s %21s %21s %21s %21s %21s\n", "max f", "min f", "max v", "min v", "max p", "min p");
        printf("%21Le %21Le %21Le %21Le %21Le %21Le\n\n", max_f, min_f, max_f*262144.0*100000.0,
min_f*262144.0*100000.0, max_p, min_p);

        rewind(disp);
        fwrite(disp_buffer, sizeof(uint32_t), WIDTH*(MAX_Y+1), disp);

        printf("%4u steps", timestep);
        fflush(stdout);
        s = !s;
        timestep++;
        }
        fclose(disp);
}
```

Appendix E: Calculation Verification (Ruby)

```
xpos = [0x13030, 0x0CFD0, 0x0CFD0, 0x13030]
ypos = [0x13030, 0x13030, 0x0CFD0, 0x0CFD0]
zpos = [0x10100, 0x10300, 0x10500, 0x10700]

xprv = [0x13030, 0x0CFD0, 0x0CFD0, 0x13030]
yprv = [0x13030, 0x13030, 0x0CFD0, 0x0CFD0]
```

```
zprv = [0x10100, 0x10300, 0x10500, 0x10700]

xfs = [0, 0, 0, 0]
yfs = [0, 0, 0, 0]
zfs = [0, 0, 0, 0]

GALACTIC_CONST = 16

cycle = 0
while true
        gets
        puts "CYCLE #{cycle} ".ljust(70, '-')
        4.times do |core|
        puts "CORE #{core}"
        puts "   POSITION"
        puts "  x: " + xpos[core].round.to_s.rjust(6) + " " + xpos[core].round.to_s(16)
        puts "  y: " + ypos[core].round.to_s.rjust(6) + " " + ypos[core].round.to_s(16)
        puts "  z: " + zpos[core].round.to_s.rjust(6) + " " + zpos[core].round.to_s(16)
        xfs[core] = 0
        yfs[core] = 0
        zfs[core] = 0
        ([0,1,2,3]-[core]).each do |idx|
        dist = (xpos[idx]-xpos[core])**2 + (ypos[idx]-ypos[core])**2 + (zpos[idx]-zpos[core])**2
        inv_dist = 1.0/dist
        xforce = (xpos[idx]-xpos[core])*inv_dist
        yforce = (ypos[idx]-ypos[core])*inv_dist
        zforce = (zpos[idx]-zpos[core])*inv_dist
        xfs[core] = xfs[core] + xforce
        yfs[core] = yfs[core] + yforce
        zfs[core] = zfs[core] + zforce
=begin
        puts "   AGAINST #{idx}"
        puts "  inv dist: #{inv_dist}"
        puts "  x force: #{xforce}"
        puts "  y force: #{yforce}"
        puts "  z force: #{zforce}"
=end
        end
=begin
        puts "   TOTAL"
        puts "  x force: #{xfs[core]}"
        puts "  y force: #{yfs[core]}"
        puts "  z force: #{zfs[core]}"
=end
        end
        4.times do |core|
        newx = xpos[core]*2 - xprv[core] + xfs[core]*(2**GALACTIC_CONST)
        xprv[core] = xpos[core]
        xpos[core] = newx
        newy = ypos[core]*2 - yprv[core] + yfs[core]*(2**GALACTIC_CONST)
        yprv[core] = ypos[core]
        ypos[core] = newy
        newz = zpos[core]*2 - zprv[core] + zfs[core]*(2**GALACTIC_CONST)
        zprv[core] = zpos[core]
        zpos[core] = newz
        end
        cycle += 1
```

end