Final Report: Large-Scale N-Body Accelerator

Lucy He (lh3365), Xiyuan Peng (xp2236), Jingzeng Xie (jx2668),
Pengpeng Wang (pw2660), Charlotte Chen (hc3558)

Spring 2026

Contents

1 Introduction

The goal of this project is to design and implement an FPGA-based hardware accelerator for gravitational N-body simulation, a classic $O(N^2)$ problem in which each body interacts with every other body through pairwise forces (1). Our system targets the force-computation and time-update loop of a 2D gravity simulation. The hardware consists of four parallel two-body compute cores with pipelined datapath. Each core computes pairwise acceleration contributions using a fast inverse square root unit, followed by accumulation. A leapfrog integration step then updates particle velocities and positions over time. The host processor handles initialization, user interaction, and display coordination via an Avalon memory-mapped interface. The resulting simulation is visualized on a VGA display as an interactive 2D gravity simulator.

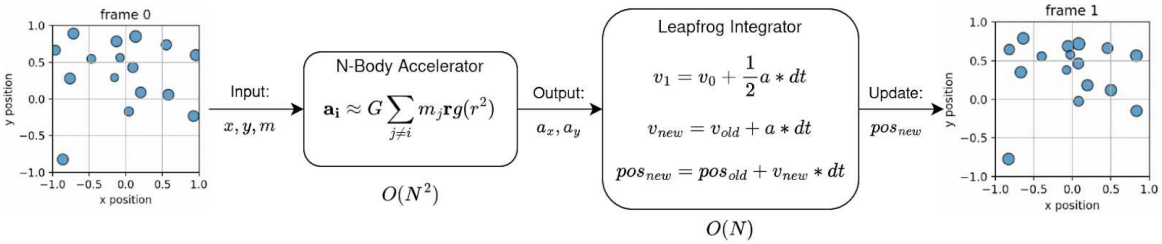

Figure 1: System Demo

2 System Block Diagram

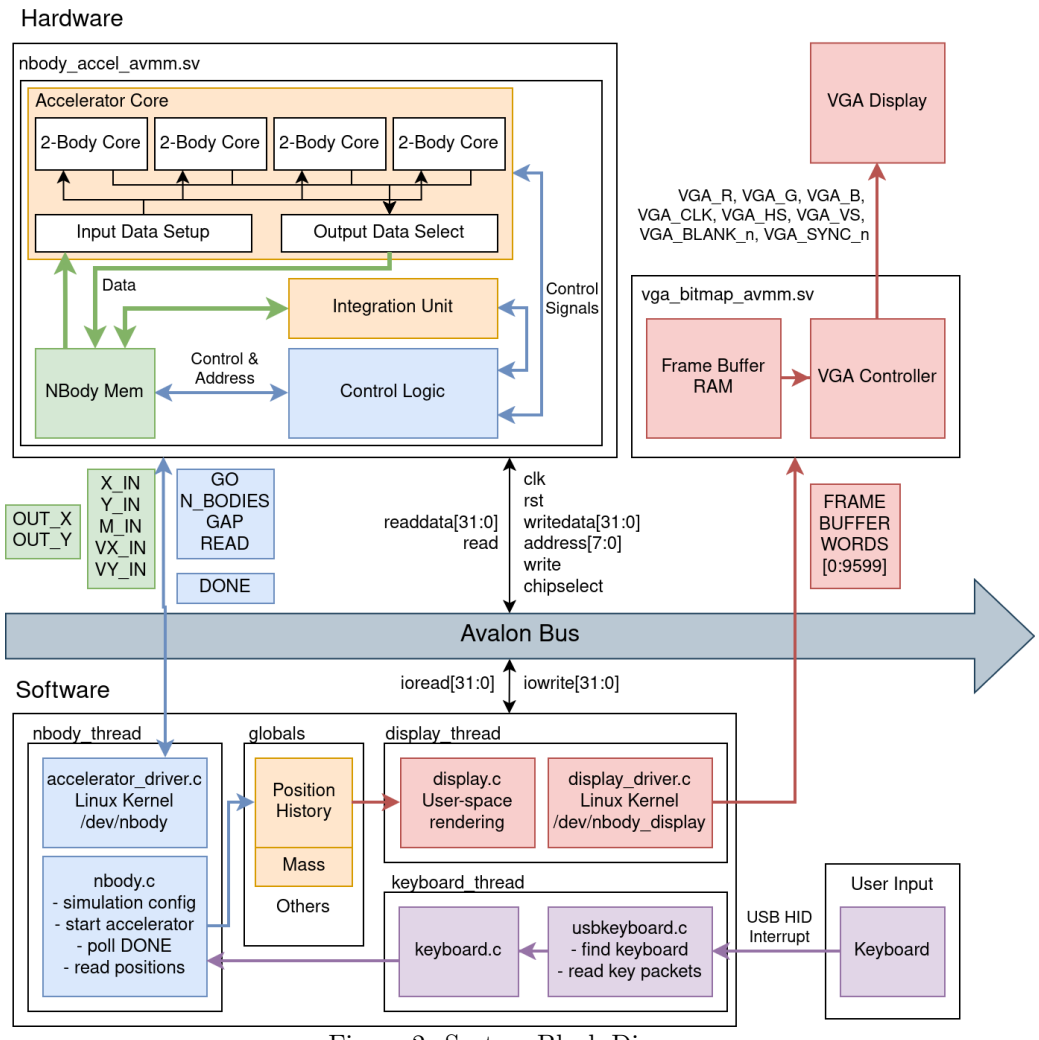

Figure 2: System Block Diagram

Figure 2 shows the full hardware/software organization of the N-body accelerator system. The design is split into two Avalon-MM peripherals: the N-body accelerator module, `nbody_accel_avmm.sv`, and the VGA display module, `vga_bitmap_avmm.sv`. Both peripherals are connected to the HPS software through the Avalon bus and are accessed from software using memory-mapped I/O through Linux kernel drivers.

The accelerator peripheral contains local N-body memory, four parallel 2-body computation cores, an integration unit, and control logic. Software writes body data and configuration registers through the Avalon bus, starts computation using the `GO` signal, and reads back updated positions when the hardware raises `DONE`.

The VGA peripheral contains a frame buffer RAM and VGA controller. Software converts body positions into pixels and writes the frame buffer through the Avalon bus. The VGA controller continuously reads this buffer and generates the VGA timing and RGB output signals for the monitor.

On the software side, the application uses separate threads for acceleration, display, and keyboard input. The accelerator thread controls the hardware simulation step, the display thread renders the current frame, and the keyboard thread handles user input. Shared global data stores the current body states and position history.

Overall, software manages simulation control and user interaction, while the FPGA hardware performs the expensive N-body computation and real-time VGA display. The Avalon bus connects the HPS software to both hardware peripherals.

3 The N-Body Problem

The N-body problem simulates particles whose motion is determined by pairwise interactions with all other bodies. In the direct formulation, each of the N bodies interacts with the other $N-1$ bodies, resulting in a computational complexity of $O(N^2)$.

The evolution of the system begins with the standard Newtonian gravity equation. For two bodies with masses m_i and m_j, the magnitude of the gravitational force is given by:

$$|F_{ij}| = G\frac{m_i m_j}{r_{ij}^2}, \tag{1}$$

where G is the gravitational constant and r_{ij} denotes the distance between the two bodies.

By applying Newton's second law, the acceleration induced by body j on body i can be written as:

$$a_{ij} = \frac{F_{ij}}{m_i} = G\frac{m_j}{r_{ij}^2}. \tag{2}$$

After summing the contributions from all other bodies, the simulation obtains the total acceleration vector of body i. To express both magnitude and direction in vector form, the relative displacement vector is introduced as $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$. Accordingly, the acceleration contribution from body j to body i is written as:

$$\mathbf{a}_{ij} = G\frac{m_j\,\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}. \tag{3}$$

In practical simulation, a softening factor ε^2 is added to avoid numerical instability when two bodies become too close. Therefore, the softened pairwise interaction is expressed as:

$$\mathbf{a}_{ij} \approx G\frac{m_j\,\mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \varepsilon^2\right)^{3/2}}. \tag{4}$$

For the two-dimensional implementation adopted in this work, the acceleration vector can be further decomposed. Let $\Delta x_{ij} = x_i - x_j$ and $\Delta y_{ij} = y_i - y_j$, then the squared distance between bodies i and j is:

$$r_{ij}^2 = (\Delta x_{ij})^2 + (\Delta y_{ij})^2. \tag{5}$$

Accordingly, the softened pairwise acceleration can be expressed component-wise as:

$$a_{ij,x} \approx G m_j \frac{\Delta x_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}}, \qquad a_{ij,y} \approx G m_j \frac{\Delta y_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}}. \tag{6}$$

The total acceleration of body i is then obtained by summing these contributions over all interacting bodies in the x and y directions separately. This decomposition is used directly in hardware, where the accelerator computes the two directional components before passing them to the integration stage.

The goal of this project is to accelerate the $O(N^2)$ pairwise interaction computation, which is the main bottleneck in direct N-body simulation. To improve throughput, the hardware uses four parallel, fully pipelined 2-body computation cores. The design also adopts a custom 27-bit floating-point format so that each mantissa multiplication maps efficiently to one 18×18 DSP block. In addition, a fast inverse square root (FISR) unit is used to efficiently approximate the inverse-distance term, and a leapfrog integration unit is implemented in hardware to update body velocity and position after acceleration computation.

4 Hardware Implementation

4.1 Key Computation Building Blocks

The accelerator is composed of several core computation modules that together implement the N-body pipeline. These hardware building blocks are designed around the custom 27-bit floating-point datapath and are optimized for efficient FPGA execution.

The main computation modules include:

- **Floating Point Adder** Performs floating-point accumulation and arithmetic reduction operations required during force summation.
- **Floating Point Multiplier** Computes the multiplication operations used in distance calculation, force scaling, and acceleration updates.
- **Fast Inverse Square Root** Implements an optimized approximation of $1/\sqrt{x}$, which is the computational bottleneck of the gravitational interaction equation.
- **Two Body Core** Computes the pairwise gravitational interaction between two particles and generates the corresponding acceleration contribution.

Together, these modules form the computational foundation of the accelerator and enable large-scale parallel N-body simulation on the FPGA platform.

4.1.1 Floating Point Adder

The floating-point adder is implemented as a two-stage pipelined unit. In the first stage, the exponents of the two inputs are compared, and the mantissa with the smaller exponent is shifted to align both operands to the same exponent. In the second stage, the aligned mantissas are either added or

subtracted depending on the input signs, followed by normalization to produce the final floating-point result.

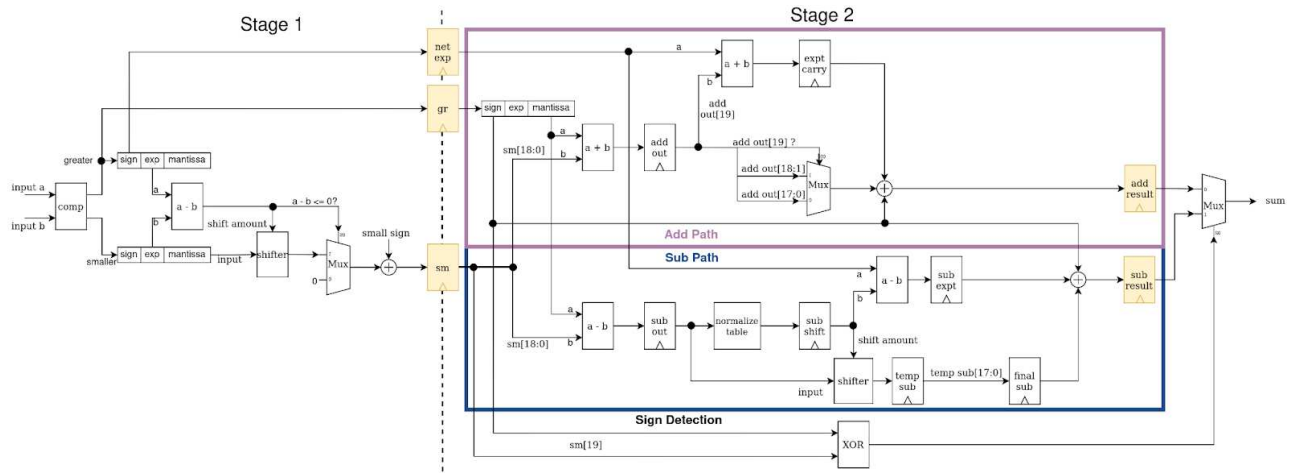

Figure 3: Fpadder Mudule

4.1.2 Floating Point Multiplier

The floating-point multiplier is implemented as a fully combinational unit. The output sign is computed by XORing the two input signs, the output exponent is computed by adding the input exponents with bias correction, and the mantissas are multiplied to form the product. The result is then normalized and packed back into the custom floating-point format.

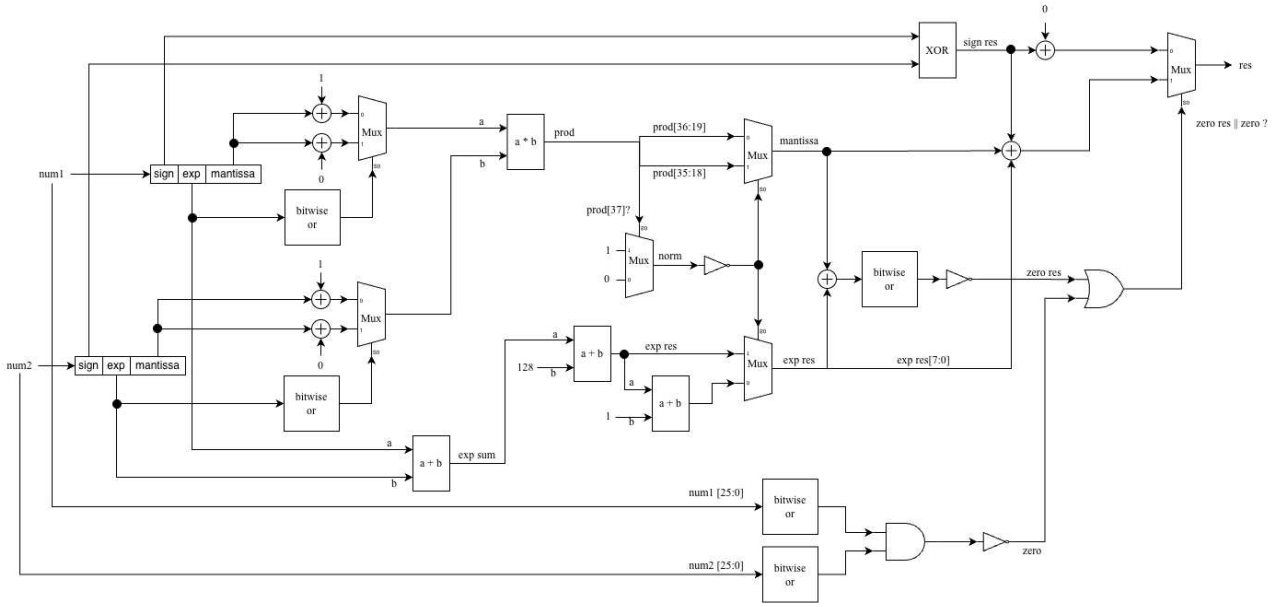

Figure 4: Fpmultiplier Mudule

4.1.3 Fast Inverse Square Root

The core computational bottleneck in the N-body force calculation is evaluating the term:

$$g(r^2) = \frac{1}{(r^2 + \varepsilon^2)^{3/2}}, \tag{7}$$

which requires both a square root and a division. On general-purpose hardware, dedicated square root and division units are expensive in terms of area and latency, making them poorly suited to a

deeply pipelined FPGA design. Instead, we adopt the well-known Fast Inverse Square Root (FISR) algorithm, originally popularized by the Quake III Arena source code, which approximates $1/\sqrt{x}$ using only integer bit manipulation, addition, and multiplication (2; 3).

Step 1: Integer Interpretation of a Float. The key insight is that the bit pattern of an IEEE 754 single-precision float encodes an approximation of its own base-2 logarithm. A 32-bit floating-point number is stored as three fields $[S \mid E \mid M]$, where S is the sign bit, E is the 8-bit biased exponent, and M is the 23-bit mantissa:

$$x = (-1)^S \cdot 2^{E-127} \cdot \left(1 + \frac{M}{2^{23}}\right).$$

Its integer reinterpretation is $I = S \cdot 2^{31} + E \cdot 2^{23} + M$, giving the approximation:

$$\log_2(x) \approx \frac{I}{2^{23}} - 127.$$

Step 2: Magic Number. To compute $y \approx 1/\sqrt{x}$, we require $\log_2(y) \approx -\frac{1}{2}\log_2(x)$. Substituting the integer approximation and solving for I_y gives:

$$I_y \approx c_0 - \frac{I_x}{2},$$

where the magic constant $c_0 =$ `0x5f3759df` accounts for the floating-point bias and a small correction term. The original Quake III implementation is:

```
float Q_rsqrt(float x) {
    float x2 = 0.5f * x;
    int   i  = *(int*)&x;
    i        = 0x5f3759df - (i >> 1);
    float y  = *(float*)&i;
    y        = y * (1.5f - x2 * y * y);
    return y;
}
```

Step 3: Newton's Method. The initial estimate requires refinement using Newton–Raphson iteration (4). Setting $f(y) = 1/y^2 - x = 0$, whose root is $y = 1/\sqrt{x}$, with $f'(y) = -2/y^3$, one Newton–Raphson iteration gives:

$$y_{n+1} = y_n \left(1.5 - 0.5 \cdot x \cdot y_n^2\right). \tag{8}$$

This correction requires two multiplications and one subtraction with no division or square root hardware involved.

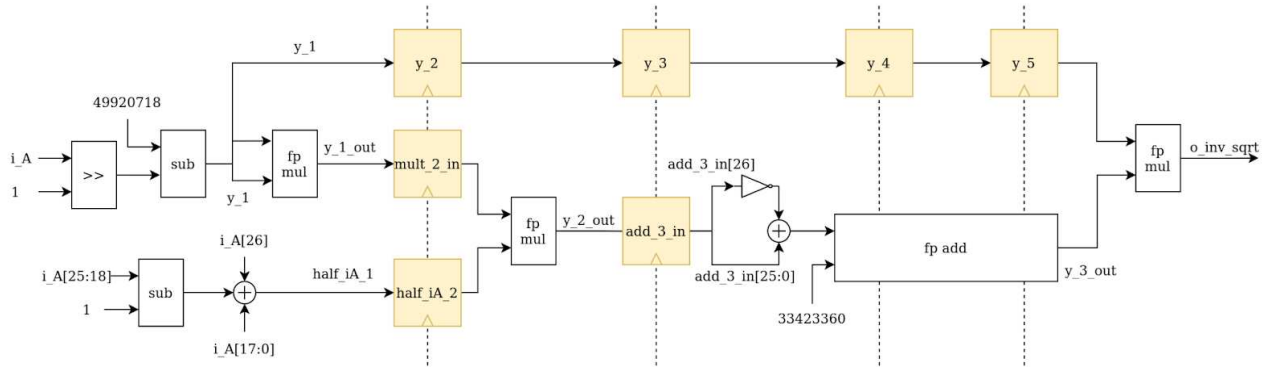

Figure 5: Fp Fast Inverse Square Root Mudule

4.1.4 Pipelined Two-Body Core

Each two-body core implements the full softened pairwise acceleration computation as a single fully pipelined datapath, accepting a new body pair every clock cycle (5). The pipeline proceeds in five stages, illustrated in Figure 6:

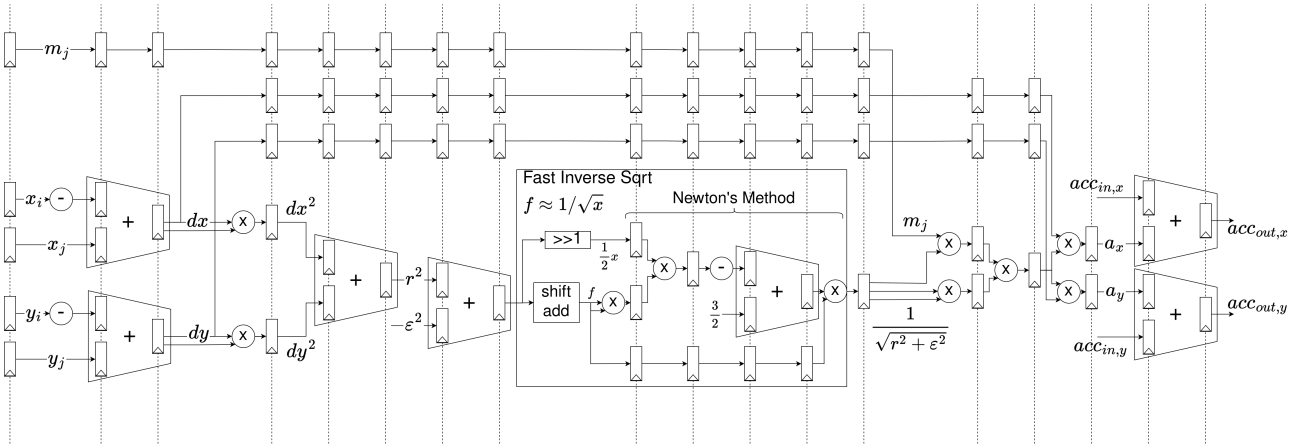

Figure 6: Pipelined datapath of a single two-body compute core. Inputs x_i, x_j, y_i, y_j, m_j enter from the left; the Fast Inverse Square Root unit occupies the central stages; and the final stages scale and accumulate the pairwise acceleration contributions into `acc_out_x` and `acc_out_y`. Vertical dashed lines denote pipeline stage boundaries; rectangular blocks denote pipeline registers.

1. **Displacement:** Compute $\Delta x = x_i - x_j$ and $\Delta y = y_i - y_j$ in parallel using floating-point subtractors.
2. **Squared distance:** Compute Δx^2, Δy^2, sum them, and add ε^2 to obtain the softened $r^2 + \varepsilon^2$.
3. **Fast inverse square root:** Apply the FISR pipeline to compute $f \approx 1/\sqrt{r^2+\varepsilon^2}$, then multiply f^2 and apply one Newton–Raphson correction to obtain $f^3 \approx (r^2+\varepsilon^2)^{-3/2}$.
4. **Force scaling:** Multiply f^3 by m_j to obtain the scalar force coefficient, then multiply separately by Δx and Δy to obtain a_x and a_y.
5. **Accumulation:** Add a_x and a_y to the running partial sums `acc_in_x` and `acc_in_y`, producing `acc_out_x` and `acc_out_y`.

Four such cores operate in parallel, each assigned a distinct body pair at each clock cycle by the controller.

4.2 N-body Memory

The N-body memory module is the main on-chip storage block for the simulation state. It stores each body's position, velocity, mass, and acceleration fields, with the major attributes placed in separate M10K-backed arrays so that different fields can be accessed and updated independently. During initialization, the HPS software writes the initial body records into this memory through the Avalon interface. During hardware execution, the controller reads body data from memory, the acceleration pipeline writes back accumulated acceleration values, and the leapfrog integrator writes back updated position and velocity values. The memory uses synchronous reads, so requested body data becomes valid one clock cycle after the read address is issued. Internally, write-selection logic resolves updates from the initialization path, acceleration writeback path, and integrator writeback path so that each pipeline stage updates only the fields it owns.

```
(* ramstyle = "M10K" *) logic [DATA_W-1:0] x_mem  [0:MAX_BODIES-1];
(* ramstyle = "M10K" *) logic [DATA_W-1:0] y_mem  [0:MAX_BODIES-1];
(* ramstyle = "M10K" *) logic [DATA_W-1:0] m_mem  [0:MAX_BODIES-1];
(* ramstyle = "M10K" *) logic [DATA_W-1:0] vx_mem [0:MAX_BODIES-1];
(* ramstyle = "M10K" *) logic [DATA_W-1:0] vy_mem [0:MAX_BODIES-1];
(* ramstyle = "M10K" *) logic [DATA_W-1:0] ax_mem [0:MAX_BODIES-1];
(* ramstyle = "M10K" *) logic [DATA_W-1:0] ay_mem [0:MAX_BODIES-1];
```

Figure 7: N-body memory block.

4.3 Four-Core Wrapper and Data Flow

The four-core wrapper is the main acceleration datapath used to compute pairwise interactions. Instead of processing one target body at a time, the wrapper works on a tile of 16 target bodies. These 16 bodies are first loaded from N-body memory into a local input bank inside the wrapper. The tile is divided into four groups, where each group contains four target bodies that are mapped onto the four 2-body cores.

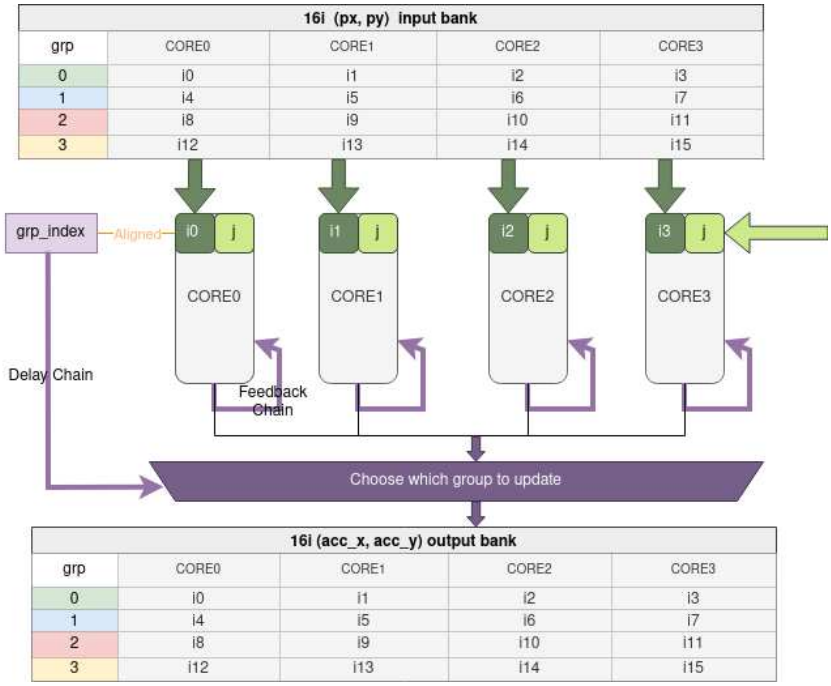

Figure 8: Four-core wrapper dataflow. A 16-body tile is divided into four groups, with each group mapped onto the four 2-body cores.

4.3.1 Tiled Input Organization

The local input bank stores the 16 target bodies as four groups of four bodies. The group index selects which four bodies are currently connected to the four cores. For example, group 0 maps bodies $i0$–$i3$ to cores 0–3, group 1 maps bodies $i4$–$i7$, and so on. This allows the same four physical cores to be reused across all 16 bodies in the tile.

4.3.2 Broadcasted Source Body

During computation, the controller streams one source body j into the wrapper each cycle. The source body's position (x_j, y_j) is broadcast to all four cores, while each core receives one target body (x_i, y_i) from the selected group. The source mass m_j is also broadcast, but it can be masked per lane. This lane mask is used to disable invalid bodies at the end of a tile and to remove self-interactions where $i = j$.

Each 2-body core computes the acceleration contribution from the current source body j to its assigned target body i. Because the cores are fully pipelined, a new source body can be issued every cycle while previous interactions continue moving through the pipeline.

4.3.3 Accumulation and Feedback

Each 2-body core accumulates acceleration over all source bodies. The previous accumulated acceleration is fed back into the core so that each new pairwise contribution is added to the running sum:

$$a_{i,x}^{new} = a_{i,x}^{prev} + a_{ij,x}, \qquad a_{i,y}^{new} = a_{i,y}^{prev} + a_{ij,y}.$$

The feedback path is delayed to align with the pipeline latency of the core. After the pipeline is drained, the accumulated acceleration values represent the total acceleration for the selected group of target bodies.

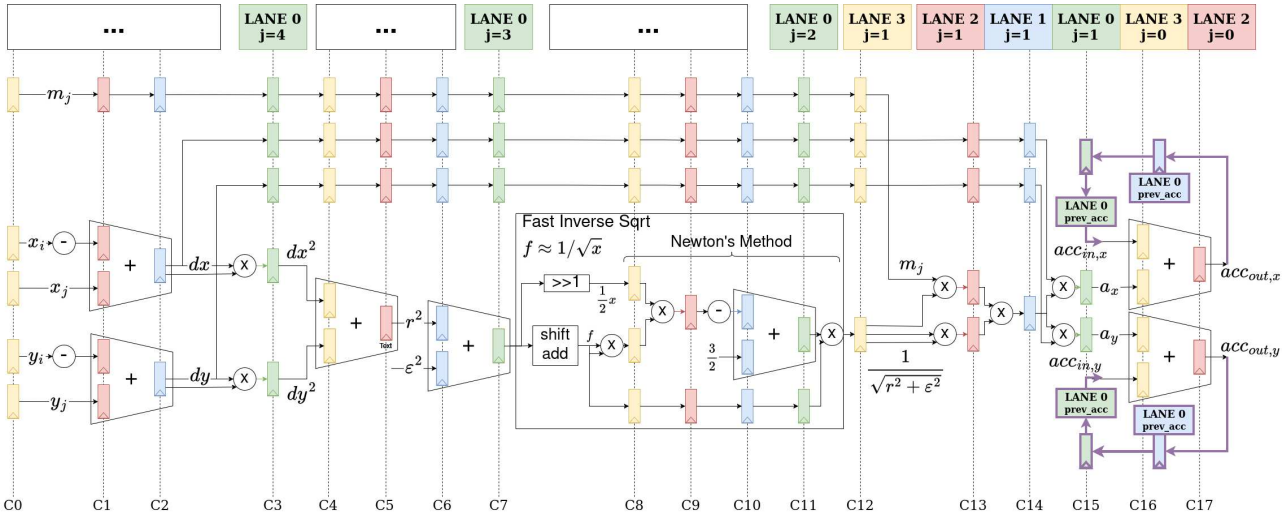

Figure 9: Single 2-body core pipeline. The core computes one pairwise acceleration contribution and feeds the accumulated result back into the pipeline for the next source body.

4.3.4 Output Selection and Writeback

The wrapper stores the accumulated acceleration results for all 16 bodies in a local output bank. Since only four cores are available, the controller selects one group at a time and writes the four corresponding acceleration results back to the N-body acceleration memory. The `grp_index` signal is therefore used both to select which input group is currently being computed and to select which output group is being written back. After all four groups are stored, the controller advances to the next 16-body tile or proceeds to the integration stage.

4.4 Leapfrog Integrator

4.4.1 Leapfrog Integration Algorithm

To update the position and velocity of each body over time, we adopt the leapfrog integration algorithm in hardware, replacing the simpler Euler method (6). Unlike the Euler update:

$$v_{\text{new}} = v_{\text{old}} + a \cdot dt, \qquad \text{pos}_{\text{new}} = \text{pos}_{\text{old}} + v_{\text{new}} \cdot dt,$$

the leapfrog method splits each timestep into three stages:

$$v_{i+1/2} = v_i + \tfrac{1}{2} a_i \cdot dt \tag{9}$$

$$x_{i+1} = x_i + v_{i+1/2} \cdot dt \tag{10}$$

$$v_{i+1} = v_{i+1/2} + \tfrac{1}{2} a_{i+1} \cdot dt \tag{11}$$

This approach uses the mid-interval velocity to update position, yielding better long-term accuracy than Euler integration and preserving energy conservation over extended simulations. Critically, Steps 9, 10, and 11 each require only simple multiply-add operations, while the acceleration computation is handled by the two-body cores. To simplify compute, we combine Step 9 with the next iteration's Step 11, so that leapfrog integration can be done in 2 steps:

$$v_{i+1/2} = v_{i-1/2} + a_i \cdot dt \tag{12}$$

$$x_{i+1} = x_i + v_{i+1/2} \cdot dt \tag{13}$$

except for the first step where

$$v_{1/2} = v_0 + \tfrac{1}{2} a_0 \cdot dt \tag{14}$$

4.4.2 Integrator Implementation

The leapfrog integrator is implemented as a small floating-point datapath controlled by a mini FSM. It takes the current body state (x, y, v_x, v_y) and acceleration (a_x, a_y), first updates the velocity, and then uses the updated velocity to update the position:

$$v'_x = v_x + a_x, \qquad v'_y = v_y + a_y,$$

$$x' = x + v'_x, \qquad y' = y + v'_y.$$

We do not include an explicit multiplication by dt in the integrator because the timestep scaling is absorbed into the acceleration and velocity increments used by the hardware, effectively treating each hardware update as one normalized timestep. The datapath uses pipelined floating-point adders, so the control FSM inserts wait states between the velocity update and the position update. After both add stages complete, the integrator asserts `o_done` to notify the main controller that the updated body state is ready.

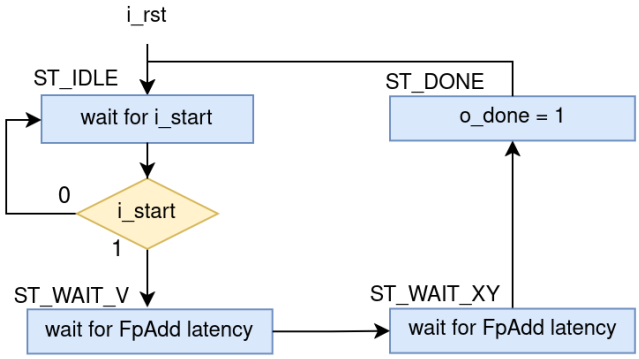

Figure 10: Mini FSM for the leapfrog integrator. The FSM waits for `i_start`, accounts for the floating-point adder latency during the velocity and position update stages, and asserts `o_done` when integration is complete.

4.5 N-body Controller

The accelerator controller is implemented as a finite state machine that sequences the full hardware simulation step. At a high level, the controller loads a tile of up to 16 bodies into the four-core wrapper, streams every source body j through the 2-body cores, stores the accumulated acceleration results, runs the leapfrog integrator for each body, and repeats this process for the requested number of internal timesteps set by `gap`. Figure 11 shows the controller FSM.

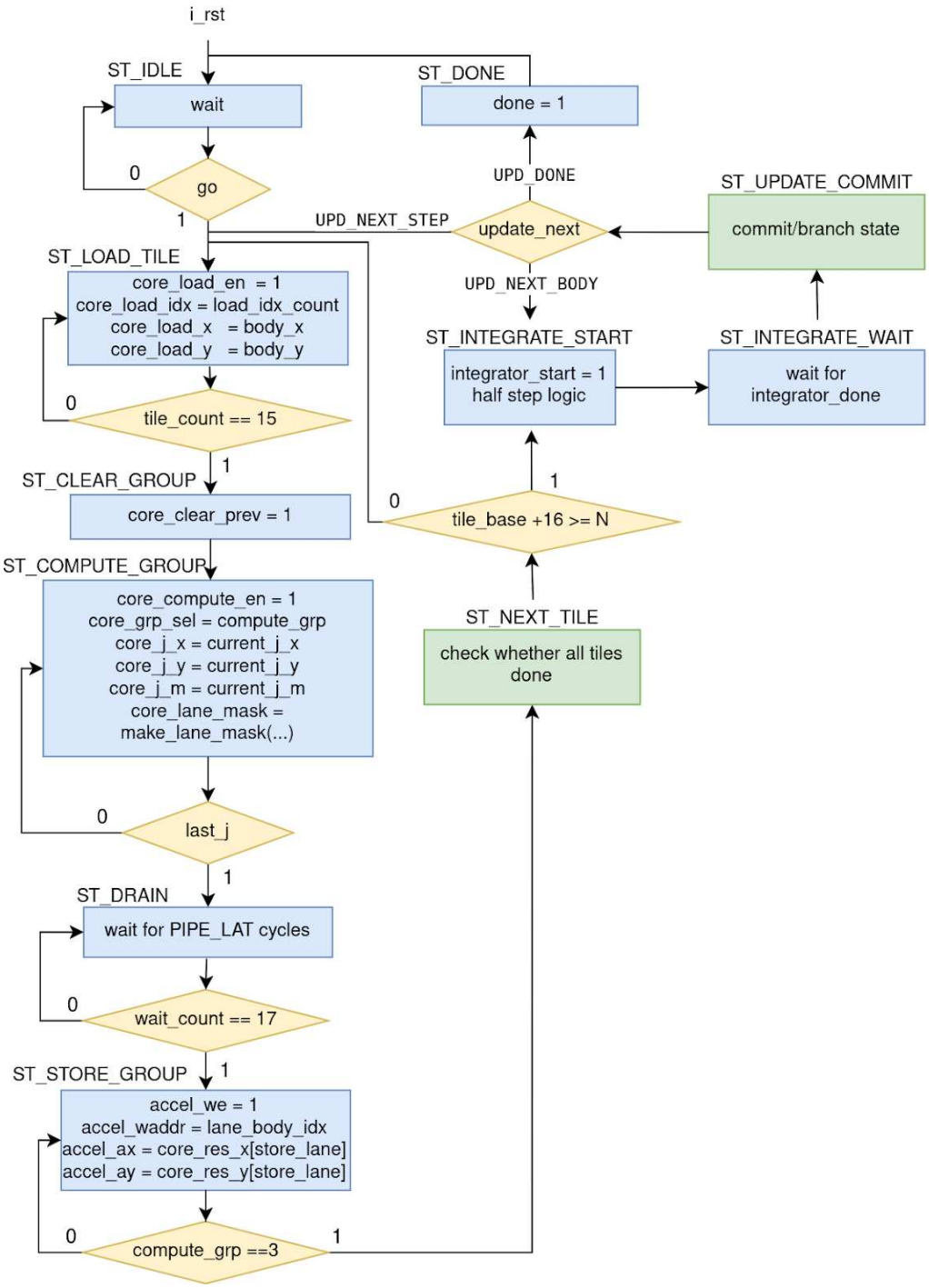

Figure 11: N-body Controller

The `_PRIME` states are one-cycle setup states used before data-dependent operations. They allow the controller to issue a memory read address first, then use the returned body data in the following state. Therefore, `ST_LOAD_TILE_PRIME` is grouped with `ST_LOAD_TILE`, `ST_COMPUTE_PRIME` is grouped with `ST_COMPUTE_GROUP`, and `ST_INTEGRATE_PRIME` is grouped with `ST_INTEGRATE_START`.

The main computation proceeds tile by tile. For each tile, the controller first loads 16 target bodies into the core wrapper. It then broadcasts each source body j to the four 2-body cores and cycles through four compute groups, allowing the hardware to accumulate acceleration contributions for all 16 target bodies. After the last source body has been processed, the controller waits for the pipeline to drain and stores the four-lane acceleration outputs group by group.

Table 1: Main controller state groups.

State group	Function	Main outputs
ST_IDLE	Wait for software to assert go. Initialize counters, tile index, timestep count, and active body count.	-
ST_LOAD_TILE_PRIME / ST_LOAD_TILE	Read body data from N-body memory and load one tile of up to 16 target bodies into the four-core wrapper.	body_raddr, core_load_en, core_load_idx
ST_CLEAR_GROUP	Clear previous accumulated acceleration values before computing a new tile.	core_clear_prev
ST_COMPUTE_PRIME / ST_COMPUTE_GROUP	Stream each source body j through the core wrapper. The controller cycles through four groups and applies lane masks for inactive bodies and self-interactions.	core_compute_en, core_grp_sel, core_lane_mask
ST_DRAIN_GROUP	Wait for the fully pipelined 2-body cores to drain so the final accumulated results are valid.	wait_count
ST_STORE_GROUP / ST_NEXT_GROUP	Store computed acceleration results from each lane into acceleration memory, then advance to the next compute group.	accel_we, accel_waddr, accel_ax/ay
ST_NEXT_TILE	Check whether all 16-body tiles have been processed. If not, advance to the next tile.	tile_base
ST_INTEGRATE_PRIME / ST_INTEGRATE_START / ST_INTEGRATE_WAIT	Run the leapfrog integrator for each body using the stored acceleration values.	integrator_start, body_raddr
ST_UPDATE_COMMIT	Write updated position and velocity back into N-body memory, then branch to the next body, next timestep, or done state.	body_update_we, body_update_addr
ST_DONE	Assert completion so software can read the updated positions.	done

After all tiles have been processed, the controller enters the integration phase. The integrator is started once per body, and when integrator_done is asserted, the updated position and velocity are written back into the N-body memory. The ST_UPDATE_COMMIT state then chooses one of three transitions: continue to the next body, start the next internal timestep if gap has not been reached, or enter ST_DONE when the full update is complete.

5 Software Implementation

The software is responsible for system initialization, user interaction, accelerator control, frame history management, and VGA rendering. The expensive all-pairs N-body computation is offloaded to the FPGA accelerator, while the HPS software coordinates when new simulation frames are computed, read back, stored, and displayed.

The software stack is organized into three user-space threads and two Linux kernel drivers. The user-space application communicates with the accelerator through /dev/nbody and with the VGA framebuffer through /dev/nbody_display. Both device files are implemented as Linux misc devices backed by platform drivers that map the corresponding Avalon-MM peripherals from the device tree.

5.1 Software Architecture

At startup, the program takes two command-line arguments: the number of bodies and the GAP value. The number of bodies is limited to `MAX_BODIES = 1024`, while the GAP value is limited to `1--10`. The GAP value controls how many internal hardware timesteps are executed between successive software readbacks, allowing the user to trade off visual update rate and simulation speed.

After argument checking, the main program allocates the frame history buffer, initializes the random seed, resets the simulation state, and creates three threads: the accelerator thread, the display thread, and the USB keyboard thread. The accelerator thread communicates with the N-body hardware, the display thread renders frames into the VGA framebuffer, and the keyboard thread updates shared simulation state based on user input.

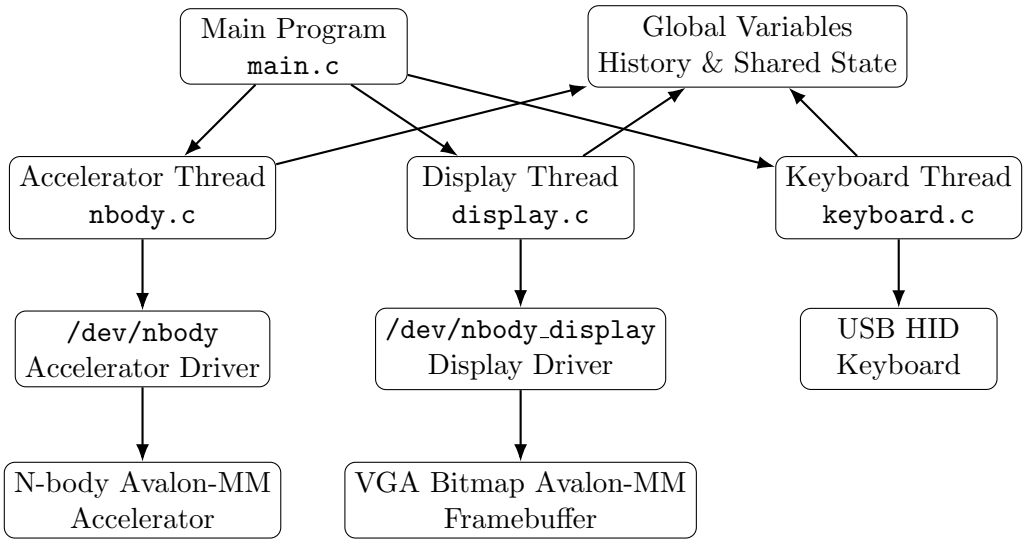

Figure 12: Software architecture. The application uses separate threads for accelerator control, display rendering, and keyboard input. Kernel drivers translate high-level `ioctl` commands into Avalon-MM transactions.

5.2 Real-Time Multi-Threaded Software Design

A key software contribution of this project is that the HPS program is structured as a real-time, multi-threaded N-body simulator rather than a simple sequential hardware test program. The application separates accelerator control, display rendering, and user input into three independent threads. This organization is important because these three tasks have very different timing behavior.

The accelerator thread is hardware-latency dependent. It starts a hardware simulation step, polls the accelerator until completion, reads back the updated body positions, and commits the result into the frame history buffer. The display thread is frame-rate dependent. It periodically selects the current history frame, renders it into a packed software framebuffer, and sends the complete frame to the VGA display driver. The keyboard thread is event-driven. It continuously monitors the USB keyboard and updates shared control state such as pause, reset, GAP, and history-frame index.

If these operations were implemented in a single sequential loop, the user interface would become less responsive. For example, keyboard input could only be processed after a hardware run and a full framebuffer update finished. Similarly, display refreshes could stall while the software is waiting for the accelerator to assert `DONE`. By using three threads, the program allows hardware computation, display refresh, and user interaction to progress concurrently. This makes the final system behave like an interactive simulator rather than a batch-mode accelerator demo.

The thread-level division also decouples the hardware computation rate from the display update rate. The accelerator can compute new simulation frames as quickly as the FPGA allows, while the display thread renders at a controlled visual frame rate. The history buffer sits between these two rates: newly computed frames are appended by the accelerator thread, while the display thread can show either the newest frame or a user-selected historical frame. This producer-consumer structure enables smooth playback, pause/resume, reset, and backward/forward frame navigation without requiring the accelerator to recompute old states.

Shared state is protected using `state_mutex`, while hardware `ioctl` access is protected using a separate `hw_mutex`. This separation keeps long hardware transactions from unnecessarily blocking display-state reads and keyboard updates. As a result, the software design supports real-time interaction while still safely coordinating access to shared simulation data and memory-mapped hardware peripherals.

5.3 Major Software Components

Table 2: Major software files and responsibilities.

File	Responsibility
`main.c`	Parses command-line arguments, allocates the history buffer, initializes the random seed, and starts the accelerator, display, and keyboard threads.
`nbody.c/h`	Maintains global simulation state, initializes random particles, controls pause/reset/history behavior, starts hardware runs, polls for completion, and stores returned body positions.
`accelerator_driver.c`	Implements the `/dev/nbody` kernel driver. It maps the accelerator Avalon-MM registers, converts between IEEE single-precision floats and the custom 27-bit floating-point format, and provides `ioctl` commands for configuration, start, done polling, and result readback.
`display.c/h`	Converts body positions to screen coordinates, draws particle circles and text into a packed 1-bpp software framebuffer, and sends each frame to the display driver.
`display_driver.c`	Implements the `/dev/nbody_display` kernel driver. It copies a packed user-space framebuffer and writes all 9600 32-bit words into the VGA Avalon-MM framebuffer.
`keyboard.c/h` `usbkeyboard./h`	Find a USB HID keyboard using `libusb`, decode keycodes, and call the corresponding simulation-control functions.

5.4 Shared Simulation State and History Buffer

The main shared software state is defined in `nbody.c`. It includes the active body count, current GAP value, pause state, running state, static display radius for each body, and a history buffer storing previously computed frames.

Each history entry stores only body positions:

$$\texttt{body\_pos\_t} = (x, y),$$

where both coordinates are single-precision floating-point values in software. The history buffer contains `MAX_HISTORY = 32768` frame slots, and each slot can store up to `MAX_BODIES = 1024` body

positions. Therefore, the maximum allocated history storage is

$$32768 \times 1024 \times 2 \times 4 = 268{,}435{,}456 \text{ bytes} \approx 256 \text{ MiB}.$$

This relatively large buffer allows the user to scrub through many previously computed frames without recomputing them in hardware.

The history buffer is used as a ring buffer. The variable `h_head` points to the next history slot to be written, `h_count` records how many valid frames are available, and `view_idx` selects the frame currently shown by the display thread. Since `MAX_HISTORY` is a power of two, wrap-around is implemented efficiently using

$$\texttt{slot} = \texttt{h\_head} \;\&\; (\texttt{MAX\_HISTORY} - 1).$$

This avoids the need for expensive modulo operations in the frame-commit path.

Two mutexes are used to protect shared resources. The `state_mutex` protects simulation state variables such as pause state, history counters, current view index, and radius indices. The `hw_mutex` protects hardware access through `ioctl` calls. Separating these locks prevents display and keyboard operations from being blocked unnecessarily by longer hardware transactions.

5.5 Initialization and Reset Flow

The initialization and reset paths share the same software routine, `reset_system()`. During reset, software randomly generates a new set of initial particles. The position range is

$$x, y \in [-10, 10],$$

the mass range is

$$m \in [0.002, 0.01],$$

and the initial velocities are set to

$$v_x = v_y = 0.$$

The initial body positions are written into `history[0]` so that the display can show the first frame immediately after reset. The body mass is also mapped into one of four display radius indices, so heavier bodies are drawn with larger circles.

After random initialization, the software writes the body count and GAP configuration to the accelerator driver using `NBODY_WRITE_CONFIG`. It then writes all body records using `NBODY_WRITE_BODIES`. The kernel driver converts each software `float` into the accelerator's custom S1E8M18 27-bit floating-point format before writing the values to the Avalon-MM input registers.

A generation counter, `sim_generation`, is used to prevent stale hardware results from being stored after a reset. If the user resets the system while the accelerator is still computing an old frame, the returned old result is discarded rather than inserted into the new simulation's history buffer. This makes reset behavior reliable even though the accelerator thread and keyboard thread run concurrently.

5.6 Accelerator Thread

The accelerator thread controls the hardware simulation loop. It opens `/dev/nbody`, loads the initial simulation state, and repeatedly starts hardware computation when the simulation is not paused. For each frame, the thread performs the following sequence:

1. Select the next available history-buffer slot.

2. Issue `NBODY_START_RUN`, which causes the driver to pulse the accelerator `GO` register.
3. Poll `NBODY_CHECK_DONE` until the hardware asserts `DONE = 1`.
4. Read all updated body positions using `NBODY_READ_RESULTS`, asserting `READ = 1` and reading `OUT_X`/`OUT_Y` for each body.
5. Clear the hardware read state using `NBODY_CLEAR_READ`, lowering `READ = 0`.
6. If no reset occurred during this run, commit the frame into the history ring and update `view_idx`; otherwise discard the result.

Figure 13 illustrates this control flow, including the hardware signals exchanged at each step.

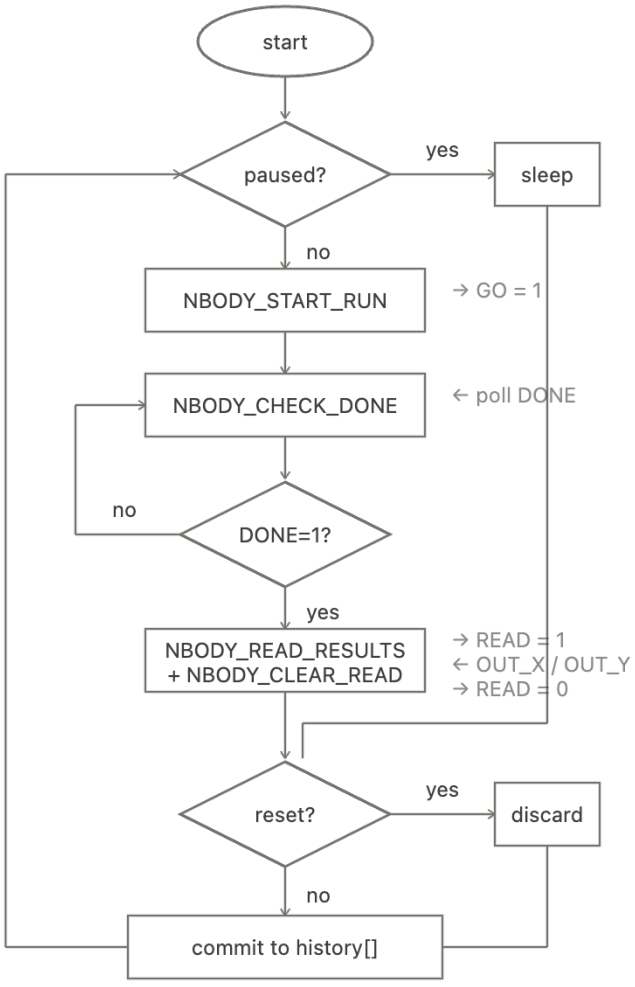

Figure 13: Accelerator thread control flow

The high-level accelerator loop is summarized below.

Listing 1: High-level accelerator-thread control loop.

```
while (running) {
    if (is_paused) {
        sleep briefly;
        continue;
    }
```

```
    next_idx = next history slot;

    ioctl(nbody_fd, NBODY_START_RUN);

    do {
        ioctl(nbody_fd, NBODY_CHECK_DONE, &done);
        sleep briefly if not done;
    } while (running && !done);

    ioctl(nbody_fd, NBODY_READ_RESULTS, history[next_idx]);
    ioctl(nbody_fd, NBODY_CLEAR_READ);

    if (no reset occurred during this run) {
        commit next_idx into the history ring;
        view_idx = newest frame;
    }
}
```

The design uses polling rather than interrupt-driven completion. This keeps the driver simple and is sufficient for the current interactive application, because the polling loop sleeps briefly between DONE checks and the display runs in a separate thread.

5.7 Display Thread and Frame Rendering

The display thread is responsible for converting simulation frames into a 640×480 packed 1-bit-per-pixel framebuffer. The top 448 pixel rows are used for body visualization, while the bottom 32 pixels are reserved for the text-based user interface. This matches two rows of 8×16 font characters.

Before rendering, the display thread copies the currently selected history frame into a local temporary buffer while holding `state_mutex`. The mutex is then released before drawing, so the accelerator and keyboard threads are not blocked during framebuffer generation.

World coordinates are mapped to screen coordinates using linear normalization:

$$x_{\text{screen}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}(640 - 1),$$

$$y_{\text{screen}} = \frac{y - y_{\min}}{y_{\max} - y_{\min}}(448 - 1),$$

where $x_{\min} = y_{\min} = -10$ and $x_{\max} = y_{\max} = 10$. This mapping keeps the simulated world coordinate range aligned with the visible display area.

5.7.1 Precomputed Circular Body Masks

A small but important rendering optimization is the use of precomputed circular body masks. The display code supports four particle radii,

$$r \in \{1, 2, 3, 4\},$$

which correspond to four mass ranges. Heavier bodies are assigned larger radius indices during initialization, so the display gives a simple visual indication of mass without requiring color or grayscale output.

Instead of recomputing the circle equation for every body in every frame, the display thread precomputes all circular masks once during display initialization. These masks are stored in a lookup table:

`body_masks[NUM_BODY_RADII][9][9]`.

The largest supported body radius is 4, so a 9×9 window is sufficient to cover all pixel offsets from -4 to $+4$ in both the horizontal and vertical directions.

For each supported radius, the initialization routine scans all local offsets (x, y) around the particle center and marks a mask entry as active if the point lies inside the filled circle:

$$x^2 + y^2 < r^2 + r.$$

The additional $+r$ term slightly thickens the discrete circle boundary, producing more visually solid particles on a low-resolution 1-bpp display. This is especially useful for small radii, where a strict $x^2 + y^2 < r^2$ test can produce sparse or uneven-looking shapes.

During frame rendering, the function that draws a body no longer evaluates the circle equation. For each particle, the renderer selects the precomputed mask using the particle's `radius_idx`, loops over the local 9×9 mask window, and calls `set_pixel()` only for mask entries that are set. This changes the per-particle drawing operation from "compute geometry and draw" to "lookup mask and draw," which is simpler and more predictable inside the real-time display loop.

Listing 2: Precomputed circular body mask generation and reuse.

```
/* Initialization: done once */
for each radius r in {1, 2, 3, 4}:
    for y = -r to r:
        for x = -r to r:
            if (x*x + y*y < r*r + r)
                body_mask[r][y+r][x+r] = 1;

/* Rendering: reused for every body in every frame */
for each body:
    select mask using radius_idx;
    for each local pixel in the mask:
        if mask pixel is set:
            set_pixel(center_x + x, center_y + y);
```

This optimization is especially useful because the display thread may need to draw up to 1024 particles every frame. Since the body shapes are static and only the particle centers change over time, precomputing the masks removes repeated arithmetic from the frame loop while preserving the appearance of circular particles. It also keeps the display code independent from the accelerator: the FPGA produces updated physical positions, while the HPS software handles the final visualization step efficiently in user space.

The software framebuffer uses the same packed format as the hardware display peripheral:

$$\texttt{word\_index} = y \cdot 20 + \left\lfloor \frac{x}{32} \right\rfloor, \qquad \texttt{bit\_index} = x \bmod 32.$$

A set bit corresponds to a white pixel, while a cleared bit corresponds to a black pixel. After all particles and UI text are drawn, the thread calls `DISPLAY_WRITE_FRAME` to send the complete 9600-word framebuffer to `/dev/nbody_display`. The display thread sleeps for approximately 33.3 ms between frames, giving an intended software rendering rate of about 30 frames/s.

5.8 Keyboard Input and User Interface

The application is controlled entirely through a USB keyboard. The keyboard thread uses `libusb` to locate a USB HID keyboard and reads interrupt-transfer packets from the keyboard endpoint. USB HID keycodes are then translated into simulation actions.

Table 3: Keyboard controls for the simulation.

Key	Action
`SPACE`	Pause or resume the simulation
`R`	Reset the simulation with newly randomized bodies
`W`	Increase GAP, causing more hardware timesteps per displayed readback
`S`	Decrease GAP, causing fewer hardware timesteps per displayed readback
`A`	Step backward by one stored history frame
`D`	Step forward by one stored history frame
`Q`	Quit the application

Manual frame navigation automatically pauses the simulation so that the user can inspect the stored history. The `A` and `D` keys also support long-press repeat: after an initial delay, the keyboard thread repeatedly steps through history at a fixed interval while the key is held. Other keys are de-bounced by comparing the current USB packet against the previous packet, avoiding duplicate actions from a single key press.

The display thread also renders a simple text-based UI in the bottom region of the VGA screen. This status line shows the number of bodies, the current GAP value, the current history-frame index, the latest computed frame, and whether the simulation is running or paused. This makes the FPGA demo self-contained: the user can adjust simulation behavior and immediately see both the visual result and the current software state on the VGA display.

5.9 Kernel Driver Layer

The software uses two kernel drivers to isolate low-level Avalon-MM access from the user-space application.

The accelerator driver registers the device file `/dev/nbody`. It obtains the accelerator's physical address range from the device tree, maps the Avalon-MM register space using `of_iomap()`, and exposes high-level `ioctl` commands for software control. The driver is also responsible for converting data between user-space IEEE-754 single-precision floats and the accelerator's custom 27-bit S1E8M18 format. On input, the 23-bit IEEE mantissa is rounded down to 18 mantissa bits and stored in the lower 27 bits of a 32-bit Avalon word. On output, the 27-bit hardware result is expanded back into a 32-bit software float.

The display driver registers the device file `/dev/nbody_display`. It maps the VGA framebuffer peripheral, allocates a kernel-side framebuffer buffer, and provides two commands: `DISPLAY_WRITE_FRAME` and `DISPLAY_CLEAR`. For each frame update, it copies the 9600-word packed framebuffer from user space and writes each word into the hardware framebuffer using `iowrite32`. The driver does not perform any rendering; all particle drawing and text drawing are handled in user space.

This driver structure gives the user-space application a clean interface to both FPGA peripherals. Instead of directly manipulating physical addresses, the application sends structured `ioctl` requests. The kernel drivers then perform the required memory-mapped I/O operations in the correct order.

5.10 End-to-End Software Flow

Overall, each interactive simulation frame follows this end-to-end flow:

1. User space initializes or resets body positions, masses, and velocities.
2. The accelerator driver converts the body data to 27-bit floating-point format and writes the data into the hardware input registers.
3. The accelerator thread starts a hardware run and polls until `DONE` is asserted.
4. The accelerator driver reads updated positions from the hardware output stream and converts them back to software floats.
5. The accelerator thread stores the returned positions into the history ring buffer.
6. The display thread selects either the newest frame or a user-selected historical frame.
7. The display thread renders particles and UI text into the packed software framebuffer.
8. The display driver copies the packed framebuffer into the VGA peripheral for display.

This structure separates the system into clear responsibilities: the FPGA performs the numerical N-body update, the kernel drivers provide safe and structured access to Avalon-MM peripherals, and the user-space application handles control flow, history management, rendering, and interaction. Together, these components form a complete real-time hardware/software N-body simulator rather than only a standalone accelerator benchmark.

6 Hardware/Software Interface

6.1 Accelerator Interface

The accelerator is controlled through a memory-mapped Avalon interface. Each transaction transfers one 32-bit word; for 27-bit floating-point values, bits [26:0] carry the data and bits [31:27] are ignored on write and return zero on read.

Body data are written sequentially through dedicated input registers. For each body, software must write the five input fields in the fixed order `X_IN`, `Y_IN`, `M_IN`, `VX_IN`, `VY_IN`. Writing `VY_IN` is treated as the completion of the current body entry; the hardware commits the assembled body data into internal memory and automatically increments the internal input pointer to the next body.

Output readback is symmetric with input streaming. After `DONE` is asserted, software reads `OUT_X` followed by `OUT_Y` for each body in sequence. Reading `OUT_Y` increments the internal output pointer to the next body. Both the input and output pointers are implicitly reset to zero when `GO` is asserted.

6.2 Display Interface

The display is controlled through a separate memory-mapped Avalon peripheral, `vga_bitmap_avmm`. Both the accelerator and display peripherals are mapped into the HPS lightweight Avalon bridge address space, whose physical base address is `0xFF200000`. To avoid address conflicts, the accelerator peripheral starts at offset `0x00000000`, while the display peripheral starts at offset `0x00010000`. Therefore, software accesses the display frame buffer using addresses relative to the display peripheral base.

Address	Register	R/W	Description
0x00	GO	W	Pulse high to start computation. Implicitly resets both input and output body pointers to 0.
0x01	N_BODIES	W	Number of active bodies in the simulation.
0x02	GAP	W	Number of timesteps executed internally between successive DONE assertions.
0x03	X_IN	W	Input X position for the current body.
0x04	Y_IN	W	Input Y position for the current body.
0x05	M_IN	W	Input mass for the current body.
0x06	VX_IN	W	Input X velocity for the current body.
0x07	VY_IN	W	Input Y velocity for the current body.
0x08	DONE	R	Set high by hardware upon completion of GAP timesteps. Cleared when software lowers READ.
0x09	READ	W	Asserted high by software after DONE is observed. Lowered by software once all outputs have been read, signalling readiness for the next GO.
0x0A	OUT_X	R	Output X position for the current body.
0x0B	OUT_Y	R	Output Y position for the current body.

The display register map is organized as a packed 1-bit-per-pixel frame buffer instead of a set of control registers. Software renders each frame in user space, packs the 640×480 bitmap into 32-bit words, and sends the complete frame to the kernel driver through `/dev/nbody_display`. The display driver then writes the packed frame buffer to the Avalon address space of the VGA peripheral.

Table 4: Display peripheral memory map.

Address Range	Name	R/W	Description
`0x0000-0x257F`	FRAMEBUFFER	W	Packed 640×480 1-bpp frame buffer. Each 32-bit word stores 32 horizontal pixels.

The frame buffer contains 9600 32-bit words, corresponding to

$$640 \times 480/32 = 9600$$

words per frame. Since each row has 640 pixels, each row occupies 20 words. Therefore, the word address and bit index for a pixel at screen coordinate (x, y) are:

$$\texttt{word\_index} = y \cdot 20 + \left\lfloor \frac{x}{32} \right\rfloor, \qquad \texttt{bit\_index} = x \bmod 32.$$

A bit value of 1 displays a white pixel, while a bit value of 0 displays a black pixel.

The hardware VGA controller continuously scans the frame buffer while software updates it through the Avalon interface. The controller generates the 640×480 VGA timing signals and uses the current pixel coordinate to read the corresponding frame-buffer word. The selected bit is converted into RGB output: white for active body pixels and black for the background. Since the frame buffer is implemented as a dual-port memory, software writes and VGA scanout can occur independently.

The kernel driver provides two main operations through `/dev/nbody_display`. The `DISPLAY_WRITE_FRAME` command copies a full packed frame from user space and writes all 9600 words into the hardware frame buffer. The `DISPLAY_CLEAR` command writes zero to every frame-buffer word, clearing the display.

7 Testbench and Verification

7.1 Arithmetic Unit Testbenches

Before verifying the full N-body datapath, we first designed unit-level SystemVerilog testbenches for the major arithmetic and memory modules. The floating-point arithmetic tests were driven by Python-generated test vectors, so that the RTL outputs could be compared against expected software-computed results.

7.1.1 Floating Pint Adder

The floating-point adder testbench verifies the correctness of the two-stage custom 27-bit floating-point adder in S1E8M18 format. The testbench reads Python-generated test vectors from `fpadd_cases.txt`. Each vector contains two 27-bit input operands and one expected 27-bit output.

Since `FpAdd` is a two-stage pipelined module, the testbench applies each input pair, waits for two positive clock edges, and then compares `oSum` with the expected Python output. The test cases include edge cases like addition with zero, cancellation between opposite inputs, different-sign addition, commutativity checks, and randomized vectors. The adder passed all the directed corner-case tests and randomized Python-generated test vectors.

7.1.2 Floating Pint Multiplier

The floating-point multiplier testbench verifies the combinational 27-bit custom floating-point multiplier using the `fpmul_cases.txt`. The multiplier computes the output sign using the XOR of the input signs, adds the two exponents, and multiplies the mantissa fields to produce final results.

The testbench covers multiplication by zero, positive and negative input combinations, multiplication commutativity, maximum-value multiplication, and randomized input vectors. For each test case, the RTL result was compared against the Python-generated expected output, and the multiplier passed all directed and randomized tests.

7.1.3 Fast Inverse Square Root

The fast inverse square root testbench verifies the pipelined module used to approximate $1/\sqrt{x}$. Since the module is pipelined, the testbench also checks output timing and throughput behavior, not only numerical correctness. Our testbench first allows the pipeline to settle and then holds each input constant for `SETTLE_CYCLES = 8` cycles before checking the output. The test cases include accuracy checks, output stability checks, pipeline timing checks, and randomized inputs. For each test case, the RTL output is compared directly against the expected output, and the module passed the Python-generated test cases under the configured pipeline settling delay.

7.1.4 N-body Memory

The N-body memory testbench verifies memory write behavior, read behavior, address independence, field preservation, and write-priority handling of the on-chip memory module that stores the body state. It also verifies the operational behavior such as correct body-data storage, acceleration updates, integrator writeback behavior, independent address access, and simultaneous write-priority

handling. The testbench uses task-based reads and writes to confirm that body updates correctly modify `x/y/vx/vy`, acceleration updates only modify `ax/ay`, and mass values are preserved across different writeback operations. Additional tests verify synchronous read behavior and simultaneous multi-source writes to the same address. The memory module passed all read/write and write-priority verification tests.

7.2 Golden Model

The verification framework uses a Python-based golden model as the software reference for validating the FPGA accelerator outputs. The model emulates the custom FP27 datapath used in the RTL, including the FP27 adder, multiplier, and Fast Inverse Square Root unit. It also follows the same N-body acceleration computation flow and arithmetic ordering as the hardware, allowing the generated acceleration outputs to be directly compared against RTL simulation and FPGA execution results.

To evaluate the numerical accuracy of the custom 27-bit S1E8M18 floating-point format, the accelerator/golden-model outputs are compared against a double-precision Float64 software reference using the same initial frame data. The relative acceleration error is measured across multiple simulation frames.

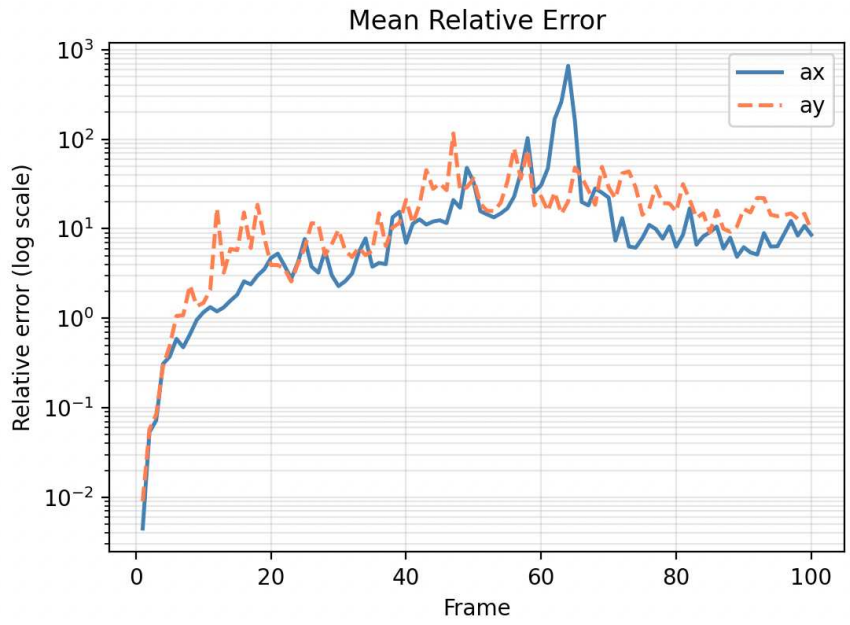

Figure 14: Accuracy validation of the 27-bit S1E8M18 floating-point format against Float64 reference computation.

Figure 14 shows the mean relative error of the computed acceleration components (a_x and a_y) over 100 simulation frames. The first frame captures the intrinsic precision loss introduced by the reduced-precision S1E8M18 format, while the increasing error in later frames is mainly due to the chaotic nature of the N-body system rather than hardware arithmetic mismatch. In the first frame, the custom 27-bit floating-point representation achieves less than 1% mean relative error compared with the Float64 reference, showing that the reduced-precision datapath is sufficiently accurate for this gravitational simulation workload.

7.3 Frame Data Generation

A Python-based frame data generator was developed to produce simulation input datasets for both RTL verification and FPGA testing. The generator creates large-scale particle initialization files containing position, velocity, and mass information for the N-body system using the custom 27-bit floating-point representation adopted in the hardware design.

The generated datasets were used as standardized test inputs throughout the verification flow, enabling consistent evaluation across the golden model, RTL simulation, and FPGA execution. In particular, initialization files such as `frame0_1024binit200_27bits.txt` were used to validate accelerator functionality under large-scale 1024-body simulation workloads.

The frame generator also automated the creation of multiple randomized test cases, allowing extensive functional testing and numerical validation of the hardware datapath under different particle distributions and initial conditions.

7.4 Outputs and Comparison

We designed a set of SystemVerilog testbenches to validate all stages of the computation pipeline against the Python-based golden models. The same generated 1024-body frame-data inputs were used across RTL simulations, controller verification, FPGA-oriented tests, and golden-model execution, allowing direct output comparison throughout the accelerator datapath and control flow.

7.4.1 Acceleration Accumulation Outputs

Using the generated 1024-body frame datasets as simulation inputs, we first verified the standalone pipelined 2-body core against the Python golden model. The golden model reproduced the same custom FP27 arithmetic flow used in hardware, including the FP27 adder, multiplier, and fast inverse square root modules. After validating the single-core acceleration results, we integrated four parallel 2-body cores into the wrapped accelerator datapath and repeated the comparison using the same frame data inputs. The accumulated acceleration outputs (ax, ay) from both the standalone core and the four-core wrapped design matched the golden-model outputs, confirming that the parallel accumulation logic, j-body broadcast scheduling, accumulation feedback chain, and multi-core datapath integration preserved numerical correctness across the full compute pipeline.

Figure 15: Acceleration Accumulation Outputs comparison

7.4.2 Controller-Integrated Outputs

The controller simulation included tile loading, j-body streaming, acceleration accumulation writeback, leapfrog integration, and final body-state updates. The RTL `nbody_control` module was compared

directly against the Python `golden_control.py` reference model, which emulates the full scheduling and state-update behavior of the hardware pipeline. The final outputs, including particle position (`x`, `y`), velocity (`vx`, `vy`), and accumulated acceleration (`ax`, `ay`), matched the golden-model results across the tested frame data. This verified the correctness of the controller FSM sequencing, memory addressing, write-enable control, integration timing, and overall system-level coordination of the N-body accelerator pipeline.

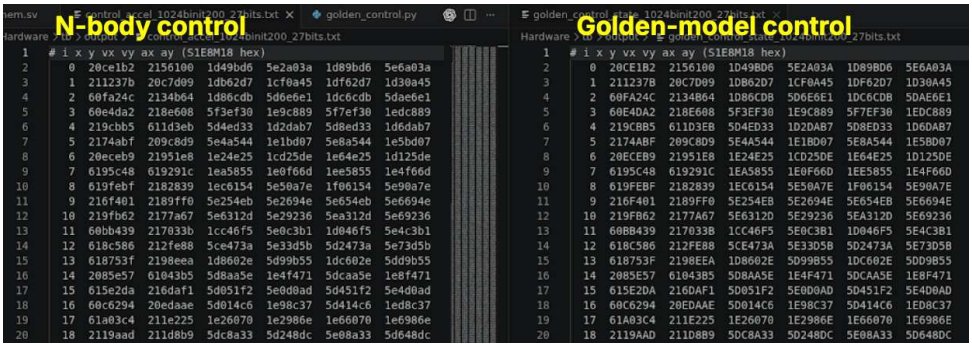

Figure 16: Controller-Integrated Outputs comparison

7.4.3 Software-visible Outputs

We also verified the software-visible outputs throug Avalon-MM interface. The `avmm_frame_xy_dump.c` test program loads particle initialization data into the FPGA accelerator through the memory-mapped registers, starts the computation by asserting `GO`, waits for the `DONE` signal, and then reads back the final `OUT_X` and `OUT_Y` values generated by the hardware. These outputs were compared directly against the Python reference generated by `golden_avmm_xy.py`, which models the same software-visible behavior of the accelerator interface. Using the same generated frame-data inputs, the RTL AVMM interface produced the same final body-position outputs as the Python golden model, verifying the correctness of the hardware/ software interface, Avalon-MM control flow, and final output readback path.

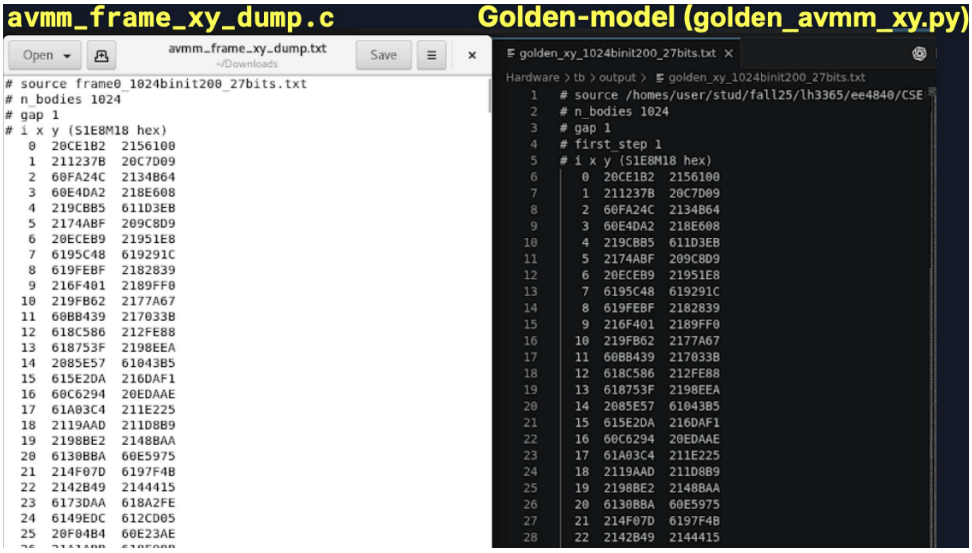

Figure 17: Avmm Frame Outputs comparison

8 Performance and Resource Utilization

8.1 Performance Metrics

The main performance metric for the N-body accelerator is the pairwise interaction throughput, defined as the number of `two_body` interactions computed per second. Since the accelerator instantiates four parallel `two_body` compute units, the theoretical interaction throughput is

$$R_{\text{int}} = 4f_{\text{clk}}.$$

After fitting in Quartus, the worst-case reported FMAX of the full system is

$$f_{\max} = 73.76 \text{ MHz}.$$

This value comes from the slow 1100 mV, 85°C timing model FMAX summary, shown in Figure 18.

```
+-----------------------------------------------------------------------
; Slow 1100mV 85C Model Fmax Summary
+-------------+-----------------+-----------------------------------
; Fmax        ; Restricted Fmax ; Clock Name
+-------------+-----------------+-----------------------------------
; 73.76 MHz   ; 73.76 MHz       ; clock_50_1
; 1184.83 MHz ; 717.36 MHz      ; soc_system:soc_system0|soc_system_h
+-------------+-----------------+-----------------------------------
This panel reports FMAX for every clock in the design, regardless of
```

Figure 18: Quartus post-fit FMAX summary under the slow 1100 mV, 85°C timing model. The worst reported FMAX is 73.76 MHz for the `clock_50_1` domain.

Therefore, the peak theoretical pairwise interaction throughput is

$$R_{\text{int}} = 4 \times 73.76 \times 10^6 = 295.04 \times 10^6$$

two-body interactions per second.

For $N = 1024$ bodies, one all-pairs simulation frame requires approximately

$$N^2 = 1024^2 = 1{,}048{,}576$$

pairwise interactions. Since four interactions are computed per cycle, the frame latency is

$$\frac{N^2}{4} = \frac{1{,}048{,}576}{4} = 262{,}144$$

accelerator cycles. The theoretical compute frame rate is therefore

$$\text{FPS}_{\text{compute}} = \frac{f_{\text{clk}}}{262{,}144}.$$

At 73.76 MHz, this gives

$$\text{FPS}_{\text{compute}} = \frac{73.76 \times 10^6}{262{,}144} \approx 281.4 \text{ frames/s}.$$

The VGA display refresh rate is approximately 60 FPS. Thus, at the post-fit FMAX, the accelerator can compute

$$\frac{281.4}{60} \approx 4.69$$

simulation frames per displayed VGA frame. Equivalently, the display can show approximately every fourth computed frame and skip about three intermediate frames while maintaining 60 FPS. Therefore, for $N = 1024$, the accelerator datapath is theoretically fast enough for real-time 60 FPS visualization, assuming that software overhead, Avalon-MM transfer latency, and framebuffer update time do not dominate.

8.2 Post-Fit FPGA Resource Utilization

Figure 19 shows the post-fit resource utilization reported by Quartus for the complete DE1-SoC system, including the HPS-connected Avalon-MM peripherals, VGA display logic, memory-mapped accelerator interface, and N-body compute datapath.

```
1   Fitter Status : Successful - Tue May 12 15:48:18 2026
2   Quartus Prime Version : 21.1.0 Build 842 10/21/2021 SJ Lite Edition
3   Revision Name : soc_system
4   Top-level Entity Name : soc_system_top
5   Family : Cyclone V
6   Device : 5CSEMA5F31C6
7   Timing Models : Final
8   Logic utilization (in ALMs) : 14,860 / 32,070 ( 46 % )
9   Total registers : 13576
10  Total pins : 362 / 457 ( 79 % )
11  Total virtual pins : 0
12  Total block memory bits : 499,820 / 4,065,280 ( 12 % )
13  Total RAM Blocks : 60 / 397 ( 15 % )
14  Total DSP Blocks : 40 / 87 ( 46 % )
15  Total HSSI RX PCSs : 0
16  Total HSSI PMA RX Deserializers : 0
17  Total HSSI TX PCSs : 0
18  Total HSSI PMA TX Serializers : 0
19  Total PLLs : 0 / 6 ( 0 % )
20  Total DLLs : 1 / 4 ( 25 % )
```

Figure 19: Quartus post-fit resource utilization summary for the complete system.

The final fitted design uses 14,860 out of 32,070 ALMs, corresponding to 46% of the available logic resources. It also uses 13,576 registers and 362 out of 457 pins. The design uses 499,820 block memory bits out of 4,065,280, corresponding to 12% of the available embedded memory, and 60 out of 397 RAM blocks, corresponding to 15% of the available RAM blocks. The final design uses 40 out of 87 DSP blocks, corresponding to 46% DSP utilization.

Table 5: Post-fit resource utilization of the complete system.

Resource	Used	Available	Utilization
ALMs	14,860	32,070	46%
Registers	13,576	–	–
Pins	362	457	79%
Block memory bits	499,820	4,065,280	12%
RAM blocks	60	397	15%
DSP blocks	40	87	46%
PLLs	0	6	0%
DLLs	1	4	25%

The design uses less than half of the available ALM and DSP resources, and only a small fraction of the available embedded memory. Therefore, the current implementation is not limited by memory capacity. The main implementation constraints are arithmetic resource usage, routing complexity, and timing closure.

8.2.1 Custom 27-bit Floating-Point Format

The choice of floating-point precision directly determines how efficiently the design maps onto the Cyclone V DSP blocks (7). The Cyclone V 5CSEA5 FPGA provides 174 independent 18 × 18 bit multipliers in its DSP blocks. The available multiplier configurations and device resource counts are shown in Figure 20.

Variant	Member Code	Variable-precision DSP Block	Independent Input and Output Multiplications Operator			18 x 18 Multiplier Adder Mode	18 x 18 Multiplier Adder Summed with 36 bit Input
			9 x 9 Multiplier	18 x 18 Multiplier	27 x 27 Multiplier		
Cyclone V E	A2	25	75	50	25	25	25
	A4	66	198	132	66	66	66
	A5	150	450	300	150	150	150
	A7	156	468	312	156	156	156
	A9	342	1,026	684	342	342	342
Cyclone V GX	C3	57	171	114	57	57	57
	C4	70	210	140	70	70	70
	C5	150	450	300	150	150	150
	C7	156	468	312	156	156	156
	C9	342	1,026	684	342	342	342
Cyclone V GT	D5	150	450	300	150	150	150
	D7	156	468	312	156	156	156
	D9	342	1,026	684	342	342	342
Cyclone V SE	A2	36	108	72	36	36	36
	A4	84	252	168	84	84	84
	A5	87	261	174	87	87	87
	A6	112	336	224	112	112	112

Figure 20: dsp_blocks_table

The IEEE 754 single-precision floating point uses a 23-bit stored mantissa (24 bits including the implicit leading 1). A 24×24 mantissa multiplication exceeds the native 18×18 DSP width and must be decomposed using the Karatsuba method:

$$24 = 18 + 6 \implies \text{requires 4 DSP blocks per multiplier}$$

since the four partial products (18×18, 18×6, 6×18, 6×6) each occupy one DSP block.

Our custom 27-bit format uses 1 sign bit, 8 exponent bits, and an 18-bit mantissa. An 18×18 mantissa multiplication maps exactly onto a single Cyclone V DSP block with no decomposition required. This is the key motivation for the 27-bit format: it is the largest mantissa width that fits within one native DSP block, maximizing precision per DSP block used.

The resulting multiplier density comparison is:

Format	**DSP blocks per multiplier**	**Max parallel multipliers**
IEEE 754 32-bit	4	$\lfloor 174/4 \rfloor = 43$
Custom 27-bit	1	174

This gives a 4× improvement in available multiplier count, directly translating to more parallel two-body compute cores on the FPGA fabric.

8.2.2 DSP Usage by Arithmetic Submodule

The custom floating-point adder is expected to use approximately zero DSP blocks because it mainly contains exponent comparison, mantissa alignment, add/subtract logic, normalization, and truncation. These operations map primarily to ALMs and registers. The custom floating-point multiplier is

expected to use approximately one DSP block because the main expensive operation is the mantissa multiplication.

The `FastInvSqrt` unit uses a bit-level initial approximation followed by one Newton-Raphson refinement:

$$y \leftarrow y\left(1.5 - 0.5xy^2\right).$$

The initial approximation uses approximately zero DSP blocks. The Newton step requires roughly three multiplications: y^2, xy^2, and the final multiplication by y. Therefore, `FastInvSqrt` is estimated to use approximately three DSP multiplier datapaths. If the following f^3 computation is included,

$$f^2 = f \cdot f, \qquad f^3 = f^2 \cdot f,$$

then two additional multipliers are required.

Table 6: Estimated DSP usage of major arithmetic submodules.

Submodule or operation	Estimated DSP multiplier datapaths
Custom floating-point add	0
Custom floating-point multiply	≈ 1
Fast inverse square root initial guess	0
Fast inverse square root Newton step	≈ 3

A conservative estimate for one fully pipelined `two_body` core is shown in Table 7. This gives approximately 10 DSP multiplier datapaths per core.

Table 7: Estimated DSP multiplier usage per two-body core.

Operation	Estimated multipliers
Squared distance, dx^2 and dy^2	2
FastInvSqrt Newton step	3
f^3 computation	2
Acceleration scaling by m_j, dx, and dy	3
Estimated total per two-body core	10

With four parallel `two_body` cores, the expected DSP usage is

$$4 \times 10 = 40$$

DSP multiplier datapaths. This matches the Quartus post-fit report, which shows 40 out of 87 DSP blocks used. The earlier conservative estimate included additional integration multipliers, but the final result suggests that these were either optimized, shared, implemented using logic, or absorbed into existing arithmetic paths.

8.2.3 Memory Usage

For the accelerator-side body storage, each body stores five 27-bit values: x, y, v_x, v_y, and mass. Additional temporary and output storage is needed for intermediate velocity and acceleration values. The conservative accelerator memory estimate is

$$(n \times 5 \times 27) + n + (4 \times n \times 27) + (n \times 4 \times 27) + n = 353n \text{ bits.}$$

For $n = 1024$,

$$353 \times 1024 = 361{,}472 \text{ bits} \approx 353 \text{ Kbits}.$$

The final Quartus report shows 499,820 block memory bits and 60 RAM blocks used by the complete system. This is larger than the accelerator-only estimate because the fitted design also includes the Avalon-MM peripherals, VGA/display memory, and other system-level storage. However, the total memory utilization is still only 12% of block memory bits and 15% of RAM blocks, so embedded memory is not a limiting resource for this implementation.

9 Contributions and Lessons Learned

Lucy He - My work focused mainly on the hardware implementation and FPGA integration of the N-body accelerator. I collaborated with Xiyuan and Charlotte on designing and debugging the `two_body_core` and helped extend it into the parallel `four_core_wrapper`. I implemented the leapfrog integrator, participated in the controller FSM design, and helped modify and run testbenches to verify that the hardware outputs matched the Python golden model. I also contributed to the hardware/-software memory-map interface, integrated the VGA display interface based on the Lab 3 framework, wrote software tests to check Avalon communication with the accelerator and display peripherals, and handled Quartus compilation, including generating the `.rbf` and device-tree files loaded onto the FPGA. One major lesson I learned came from a difficult bug where fixing the controller made the computed positions correct but unexpectedly broke the VGA display with fractured lines. After checking the code, memory map, software interface, and many possible causes, I concluded that the issue was likely related to post-fit timing or routing changes caused by a small RTL modification, especially because the VGA path is sensitive to stable timing during continuous scanout. This taught me that FPGA debugging goes beyond functional RTL correctness: a design can pass simulation and produce correct numerical outputs while still failing on real hardware due to timing, routing, or integration issues, so post-fit timing analysis and robust system-level testing are essential.

Xiyuan Peng - My work mainly focused on the design of the verification infrastructure and part of the hardware implementation for the N-body accelerator. I developed and modified SystemVerilog testbenches for the floating-point arithmetic units, memory modules, controller logic, and accelerator datapath, and helped build Python-based golden-model comparisons to validate the RTL outputs throughout different computation stages. I also participated in the implementation and debugging of several hardware modules, including parts of the two-body compute pipeline, accumulation flow, and controller integration. Through this project, I learned that hardware verification must go beyond module-level functional correctness. Even when simulation outputs match the golden model, FPGA systems can still fail because of timing, routing, interface, or integration issues after synthesis and place-and-route. This project helped me better understand the importance of systematic verification, timing-aware debugging, and full system-level validation in FPGA design.

Jingzeng Xie - My work focused mainly on the software implementation and real-time visualization of the N-body accelerator system. I collaborated closely with Pengpeng on most parts of the software stack, including the main application structure, simulation control flow, keyboard interaction, display rendering, and hardware/software communication logic, while Pengpeng primarily handled the accelerator driver. I developed the display-side rendering approach for circular bodies, where precomputed circular stamps are used to efficiently draw particles with different radii onto the framebuffer. This reduced repeated per-pixel computation during each frame and made the software display pipeline more suitable for real-time visualization. I also integrated the software threads for simulation, display, and user input so that the accelerator, VGA output, and keyboard controls could work together as an interactive system. One major lesson I learned is the importance of starting hardware/software integration early. Compared with pure software debugging, FPGA-based projects require much

longer iteration cycles because even small hardware changes may require regeneration, recompilation, device-tree updates, and on-board testing. This made debugging slower and more complex than expected, especially when software behavior depended on hardware timing and interface correctness. Starting integration earlier would leave more time to identify interface mismatches, display issues, and system-level bugs before the final stage of the project.

Pengpeng Wang - I implemented the accelerator and display kernel drivers, including the float32-to-27bit conversion and the decision to use a packed 1-bpp framebuffer instead of 32-bpp, which reduced per-frame transfer size from over 1 MB to 38 KB. I also wrote the design document and contributed to the final report. Through this project I gained hands-on experience writing Linux kernel drivers and working with memory-mapped hardware for the first time. Seeing the full stack — from a C program calling ioctl, through the kernel, across the Avalon bus, to actual hardware registers — made the abstractions I had only seen in class feel concrete and real.

Charlotte Chen My work primarily focuses on the testbench implementation and verification of the accelerator, specifically ensuring the high-precision floating-point, including the FP adders, multipliers, and the fast inverse square root modules, maintained bit-accurate consistency with our Python golden model. For the FP adder and multiplier, I developed directed and constrained-random testbenches to verify corner cases such as subnormal handling, overflow, and rounding modes, ensuring that the pipelined datapath maintained throughput without sacrificing precision. Validating the fast inverse square root was particularly challenging, as it required sweeping a wide range of input magnitudes to ensure the hardware's approximation stayed within the required error bounds. This taught me that verification must be "precision-aware," balancing the hardware's area-efficiency constraints against the numerical stability required for physics simulations. Ultimately, seeing how a minor adjustment in the FISR pipeline could alter the routing and timing of the entire SoC emphasized that in high-performance computing, the physical implementation of arithmetic is just as critical as its logical definition.

References

[1] Unknown, "N-body simulation on fpga (ieee paper)," *IEEE*, 2018. Available: https://ieeexplore.ieee.org/document/8445106.

[2] id Software, "Quake III arena source code." `https://github.com/id-Software/Quake-III-Arena`, 1999. Released under GPL v2, 2005.

[3] F. Stokes, "Fast inverse square root." `https://github.com/francisrstokes/githublog/blob/main/2024/5/29/fast-inverse-sqrt.md`, May 2024. Accessed: 2025.

[4] Wikipedia contributors, "Newton's method — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/wiki/Newton%27s_method`, 2025. Accessed: 2025.

[5] Cornell ECE 5760, "Fpga-based n-body simulation final project." `https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2023/raf269_nkg37_tjw234/raf269_nkg37_tjw234/index.html`. Accessed: 2026-04-17.

[6] Wikipedia, "Leapfrog integration." `https://en.wikipedia.org/wiki/Leapfrog_integration`. Accessed: 2026-04-17.

[7] Intel (Altera), "Cyclone v device handbook volume 1." `https://docs.altera.com/r/docs/683375/current/cyclone-v-device-handbook-volume-1-device-interfaces-and-integration/resources`. Accessed: 2026-04-17.

A Code

A.1 Software Source Code

A.1.1 main.c

Listing 3: code/Software/main.c

```c
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "display.h"
#include "keyboard.h"
#include "nbody.h"

int main(int argc, char **argv)
{
    pthread_t accelerator_thread;
    pthread_t vga_thread;
    pthread_t kb_thread;
    int have_keyboard = 0;

    // Check command-line arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <Num Bodies> <Gap>\n", argv[0]);
        return 1;
    }

    // Setting num_bodies and current_gap
    num_bodies = atoi(argv[1]);
    current_gap = atoi(argv[2]);
    if (num_bodies < 5 || num_bodies > MAX_BODIES) {
        fprintf(stderr, "Num Bodies must be between 5 and %d\n", MAX_BODIES);
        return 1;
    }
    if (current_gap < 1 || current_gap > NBODY_GAP_MAX) {
        fprintf(stderr, "Gap must be between 1 and %d\n", NBODY_GAP_MAX);
        return 1;
    }

    // Allocate history buffer frames with body positions
    if (allocate_history() < 0) {
        fprintf(stderr, "Could not allocate history ring (%d frames)\n", MAX_HISTORY);
        return 1;
    }

    // Initialize random seed
    srand((unsigned int) time(NULL));

    // Create Nbody (Accelerator Simulation) Thread
    if (pthread_create(&accelerator_thread, NULL, nbody_thread, NULL) != 0) {
        perror("pthread_create nbody_thread");
        free_history();
        return 1;
    }

    // Create Display Thread
    if (pthread_create(&vga_thread, NULL, display_thread, NULL) != 0) {
        perror("pthread_create display_thread");
        pthread_mutex_lock(&state_mutex);
        running = 0;
        //pthread_cond_broadcast(&frame_ready_cond);
        pthread_mutex_unlock(&state_mutex);
        pthread_join(accelerator_thread, NULL);
        free_history();
        return 1;
```

```
62      }
63
64      // Open keyboard device and create Keyboard Thread
65      if (keyboard_open())
66          have_keyboard = 1;
67      if (have_keyboard && pthread_create(&kb_thread, NULL, keyboard_handler, NULL) != 0) {
68          perror("pthread_create keyboard_handler");
69          have_keyboard = 0;
70          free_history();
71          return 1;
72      }
73
74      // Waiting Nbody thread to be destory
75      pthread_join(accelerator_thread, NULL);
76
77      pthread_mutex_lock(&state_mutex);
78      running = 0;
79      //pthread_cond_broadcast(&frame_ready_cond);
80      pthread_mutex_unlock(&state_mutex);
81
82      // Waiting Keyboard and Display thread to be destory
83      pthread_join(kb_thread, NULL);
84      pthread_join(vga_thread, NULL);
85
86      // Clear the history frame buffer
87      free_history();
88
89      return 0;
90  }
```

A.1.2 nbody.c

Listing 4: code/Software/nbody.c

```
1   #include <errno.h>
2   #include <fcntl.h>
3   #include <pthread.h>
4   #include <stdint.h>
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <sys/ioctl.h>
8   #include <unistd.h>
9
10  #include "nbody.h"
11
12  int num_bodies = 0;
13  int current_gap = 1;
14  int is_paused = 0;
15  int running = 1;
16
17  uint32_t static_masses[MAX_BODIES];
18  body_pos_t *history[MAX_HISTORY];
19  int h_head = 0;
20  int h_count = 0;
21  int view_idx = 0;
22
23  // Used to distinguish between old and new simulations following a reset
24  static unsigned int sim_generation = 0;
25  // Used to pause after a reset, awaiting user input to continue
26  static int reset_waiting_for_play = 0;
27
28  int nbody_fd = -1;
29
30  pthread_mutex_t state_mutex = PTHREAD_MUTEX_INITIALIZER;
31  pthread_cond_t frame_ready_cond = PTHREAD_COND_INITIALIZER;
32  static pthread_mutex_t hw_mutex = PTHREAD_MUTEX_INITIALIZER;
33
34  // Allocate memory for history frames with body positions
35  int allocate_history(void)
```

```
{
    for (int i = 0; i < MAX_HISTORY; i++) {
        history[i] = malloc(MAX_BODIES * sizeof(body_pos_t));
        if (!history[i]) {
            free_history();
            return -1;
        }
    }
    return 0;
}

// Release the memory for all history frames
void free_history(void)
{
    for (int i = 0; i < MAX_HISTORY; i++) {
        free(history[i]);
        history[i] = NULL;
    }
}

// Generate a random float within the range [min, max]
static float random_range(float min, float max)
{
    return min + (max - min) * ((float)rand() / (float)RAND_MAX);
}

// Map the mass to a radius index for display purposes
int get_radius_idx(float mass)
{
    float normalized = (mass - NBODY_MASS_MIN) / (NBODY_MASS_MAX - NBODY_MASS_MIN);

    if (normalized < 0.25f)
        return 0;
    if (normalized < 0.50f)
        return 1;
    if (normalized < 0.75f)
        return 2;
    return 3;
}

// Update current_gap in the software, and REG_GAP/REG_N_BODIES in the hardware
void nbody_set_gap_delta(int delta)
{
    nbody_config_t cfg;

    // Modify current_gap, then save the new configuration to cfg
    pthread_mutex_lock(&state_mutex);
    // If W is pressed, delta > 0, and current_gap is increased
    if (delta > 0 && current_gap < NBODY_GAP_MAX) {
        current_gap++;
    }
    // If S is pressed, delta < 0, and current_gap is decreased
    else if (delta < 0 && current_gap > 1) {
        current_gap--;
    }

    cfg.num_bodies = (uint32_t) num_bodies;
    cfg.gap = (uint32_t) current_gap;
    pthread_mutex_unlock(&state_mutex);

    // Write the new configuration to the hardware device /dev/nbody using ioctl
    if (nbody_fd >= 0) {
        pthread_mutex_lock(&hw_mutex);
        ioctl(nbody_fd, NBODY_WRITE_CONFIG, &cfg);
        pthread_mutex_unlock(&hw_mutex);
    }
}

// Used for browsing history frames
void nbody_show_frame_delta(int delta)
{
    pthread_mutex_lock(&state_mutex);
    // User manually advances the frame, the simulation automatically pauses
```

```
109     is_paused = 1;
110
111     // If A is pressed, delta < 0, and the previous frame is displayed
112     if (delta < 0 && view_idx > 0) {
113         view_idx--;
114     }
115     // If D is pressed, delta > 0, and the next frame is displayed
116     else if (delta > 0 && view_idx < h_count - 1) {
117         view_idx++;
118     }
119     //pthread_cond_broadcast(&frame_ready_cond);
120     pthread_mutex_unlock(&state_mutex);
121 }
122
123 /*
124 Function for SPACE (Play/Pause) being pressed
125 Status: Play -> Paused / Paused -> Play
126 */
127 void nbody_toggle_pause(void)
128 {
129     pthread_mutex_lock(&state_mutex);
130     is_paused = !is_paused;
131     // Clear reset waiting status once user switches from the paused to the play
132     if (!is_paused)
133         reset_waiting_for_play = 0;
134     //pthread_cond_broadcast(&frame_ready_cond);
135     pthread_mutex_unlock(&state_mutex);
136 }
137
138 // Function for R (Reset) being pressed
139 void nbody_request_reset(void)
140 {
141     pthread_mutex_lock(&state_mutex);
142     // If a reset has already been performed and the system is waiting for
143     // the user to press Play, do not perform another reset
144     if (reset_waiting_for_play) {
145         pthread_mutex_unlock(&state_mutex);
146         return;
147     }
148     // The simulation automatically pauses
149     is_paused = 1;
150     reset_waiting_for_play = 1;
151     // Rrevent old results from prior to a reset from being written back to history
152     sim_generation++;
153     //pthread_cond_broadcast(&frame_ready_cond);
154     pthread_mutex_unlock(&state_mutex);
155
156     // Regenerate initial particles and write to hardware
157     reset_system();
158 }
159
160 // Function for Q (Exit) being pressed
161 void nbody_request_quit(void)
162 {
163     pthread_mutex_lock(&state_mutex);
164     // All threads monitor this variable in their main loops
165     // Consequently, this causes the entire program to exit
166     running = 0;
167     //pthread_cond_broadcast(&frame_ready_cond);
168     pthread_mutex_unlock(&state_mutex);
169 }
170
171 // Responsible for initializing a new round of simulation
172 void reset_system(void)
173 {
174     nbody_particle_t *init_particles;
175
176     // Copy global variables to local variables
177     pthread_mutex_lock(&state_mutex);
178     int local_num = num_bodies;
179     int local_gap = current_gap;
180     pthread_mutex_unlock(&state_mutex);
181
```

```
    // Allocate the initial particle array
    init_particles = malloc(local_num * sizeof(*init_particles));
    if (!init_particles) {
        fprintf(stderr, "failed to allocate initial particles\n");
        return;
    }

    // Randomly Generate Initial State
    pthread_mutex_lock(&state_mutex);
    for (int i = 0; i < local_num; i++) {
        // Randomly generate x and y coordinates
        init_particles[i].x = random_range(NBODY_POS_MIN, NBODY_POS_MAX);
        init_particles[i].y = random_range(NBODY_POS_MIN, NBODY_POS_MAX);
        // Save the initial position to history
        history[0][i].x = init_particles[i].x;
        history[0][i].y = init_particles[i].y;

        // Randomly generate mass
        float mass = random_range(NBODY_MASS_MIN, NBODY_MASS_MAX);
        init_particles[i].mass = mass;
        // Determine the display radius based on the mass static_masses
        static_masses[i] = (uint32_t) get_radius_idx(mass);

        // Initialize vx and vy to 0
        init_particles[i].vx = 0.0f;
        init_particles[i].vy = 0.0f;
    }
    pthread_mutex_unlock(&state_mutex);

    // Write the configuration and initial bodies to the hardware
    if (nbody_fd >= 0) {
        nbody_config_t cfg = {
            .num_bodies = (uint32_t)local_num,
            .gap = (uint32_t)local_gap,
        };
        nbody_bodies_arg_t barg = {
            .particles = init_particles,
            .count = (uint32_t)local_num,
        };

        pthread_mutex_lock(&hw_mutex);
        // Clear read status / output pointer
        ioctl(nbody_fd, NBODY_CLEAR_READ);
        //     num_bodies     gap
        if (ioctl(nbody_fd, NBODY_WRITE_CONFIG, &cfg) < 0)
            perror("NBODY_WRITE_CONFIG");
        // Write all initial particles to the hardware input buffer
        if (ioctl(nbody_fd, NBODY_WRITE_BODIES, &barg) < 0)
            perror("NBODY_WRITE_BODIES");
        pthread_mutex_unlock(&hw_mutex);
    }

    // Reset history state
    pthread_mutex_lock(&state_mutex);
    // history[0] stores the initial frame and the next written to history[1]
    h_head = 1;
    h_count = 1;
    view_idx = 0;
    //pthread_cond_broadcast(&frame_ready_cond);
    pthread_mutex_unlock(&state_mutex);

    // Free the memory for array
    free(init_particles);
}

// Nbody (Accelerator Simulation) Thread
void *nbody_thread(void *arg)
{
    // It is merely there to avoid a compiler warning
    (void) arg;

    // Open /dev/nbody to obtain a file descriptor
    // User-space applications communicate with it via ioctl
```

```
    nbody_fd = open("/dev/nbody", O_RDWR);
    if (nbody_fd < 0)
        perror("open /dev/nbody");

    // Regenerate initial particles and write to hardware
    reset_system();

    /*
    Check if the process is paused; if not, select the next history slot
    Initiate hardware computation and wait for completion
    Read the results from the hardware into next_idx and update the history
    */
    while (running) {
        nbody_read_arg_t rarg;

        int done = 0;
        unsigned int local_generation;

        // Copy global variables to local variables
        pthread_mutex_lock(&state_mutex);
        int paused = is_paused;
        int local_num = num_bodies;
        // Subsequently determine whether a reset occurred during this round
        local_generation = sim_generation;
        pthread_mutex_unlock(&state_mutex);

        // If paused or the hardware is not open, sleep for 10 ms
        // Then resume checking in the next iteration
        if (paused || nbody_fd < 0) {
            usleep(10000);
            continue;
        }

        // After writing to the last slot, it wraps back around to the beginning
        pthread_mutex_lock(&state_mutex);
        int next_idx = h_head & (MAX_HISTORY - 1);
        pthread_mutex_unlock(&state_mutex);

        // Have the kernel driver write REG_GO = 1
        // Trigger the hardware to begin computation
        pthread_mutex_lock(&hw_mutex);
        if (ioctl(nbody_fd, NBODY_START_RUN) < 0) {
            perror("NBODY_START_RUN");
            pthread_mutex_unlock(&hw_mutex);
            usleep(10000);
            continue;
        }

        // Polling method waiting for done
        while (running && !done) {
            if (ioctl(nbody_fd, NBODY_CHECK_DONE, &done) < 0) {
                perror("NBODY_CHECK_DONE");
                break;
            }
            if (!done)
                usleep(1000);
        }

        // If process being exit
        if (!running) {
            pthread_mutex_unlock(&hw_mutex);
            break;
        }

        // Once the hardware operations are complete, read the results
        rarg.results = history[next_idx];
        rarg.count = (uint32_t) local_num;
        if (ioctl(nbody_fd, NBODY_READ_RESULTS, &rarg) < 0)
            perror("NBODY_READ_RESULTS");

        // Clear the read state and prepare for the next read
        if (ioctl(nbody_fd, NBODY_CLEAR_READ) < 0)
            perror("NBODY_CLEAR_READ");
```

```
328         pthread_mutex_unlock(&hw_mutex);
329
330         // Update history
331         pthread_mutex_lock(&state_mutex);
332         /*
333         If no reset occurs during this hardware computation cycle,
334         increment normally to the next frame. The old result will not be
335         updated to the history, preventing the old frame position prior
336         to the reset from being marked
337         */
338         if (local_generation == sim_generation) {
339             h_head = (h_head + 1) & (MAX_HISTORY - 1);
340             if (h_count < MAX_HISTORY)
341                 h_count++;
342             view_idx = h_count - 1;
343             //pthread_cond_broadcast(&frame_ready_cond);
344         }
345         pthread_mutex_unlock(&state_mutex);
346     }
347
348     // Clear hardware status
349     if (nbody_fd >= 0) {
350         pthread_mutex_lock(&hw_mutex);
351         ioctl(nbody_fd, NBODY_CLEAR_READ);
352         ioctl(nbody_fd, NBODY_STOP);
353         pthread_mutex_unlock(&hw_mutex);
354         close(nbody_fd);
355         nbody_fd = -1;
356     }
357
358     return NULL;
359 }
```

A.1.3 nbody.h

Listing 5: code/Software/nbody.h

```
1  #ifndef _NBODY_H
2  #define _NBODY_H
3
4  #include <pthread.h>
5  #include <stdint.h>
6  #include "nbody_ioctl.h"
7
8  #define MAX_BODIES NBODY_MAX_BODIES
9  #define MAX_HISTORY 32768
10 #define BODY_DISPLAY_W 640
11 #define BODY_DISPLAY_H 448
12 #define NBODY_POS_MIN (-10.0f)
13 #define NBODY_POS_MAX 10.0f
14 #define NBODY_MASS_MIN 0.002f
15 #define NBODY_MASS_MAX 0.01f
16 #define NBODY_GAP_MAX 10
17
18 extern int num_bodies;
19 extern int current_gap;
20 extern int is_paused;
21 extern int running;
22
23 extern uint32_t static_masses[MAX_BODIES];
24 extern body_pos_t *history[MAX_HISTORY];
25 extern int h_head;
26 extern int h_count;
27 extern int view_idx;
28
29 extern int nbody_fd;
30
31 extern pthread_mutex_t state_mutex;
32 extern pthread_cond_t frame_ready_cond;
```

```
33
34 int allocate_history(void);
35 void free_history(void);
36 int get_radius_idx(float mass);
37 void nbody_set_gap_delta(int delta);
38 void nbody_show_frame_delta(int delta);
39 void nbody_toggle_pause(void);
40 void nbody_request_reset(void);
41 void nbody_request_quit(void);
42 void reset_system(void);
43 void *nbody_thread(void *arg);
44
45 #endif
```

A.1.4 display.c

Listing 6: code/Software/display.c

```
1  #include <ctype.h>
2  #include <fcntl.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/ioctl.h>
8  #include <unistd.h>
9
10 #include "display.h"
11 #include "nbody.h"
12
13 static int display_fd = -1;
14 static uint32_t framebuffer[DISPLAY_WORDS];
15 static uint8_t body_masks[NUM_BODY_RADII][9][9];
16
17 static const int BODY_RADII[NUM_BODY_RADII] = { 1, 2, 3, 4 };
18
19 // Set a specific pixel is on or off (1/0)
20 static inline void set_pixel(int x, int y, int on)
21 {
22     // Check if the coordinates are out of bounds
23     if (x < 0 || x >= DISPLAY_WIDTH || y < 0 || y >= DISPLAY_HEIGHT)
24         return;
25
26     /*
27     Packed display format
28     frame index = y * 20 + x / 32, bit = x & 31 (x % 32)
29     one LSB-first bit per pixel
30     */
31     if (on) {
32         framebuffer[y * DISPLAY_WORDS_PER_ROW + x / 32] |= (1u << (x & 31));
33     } else {
34         framebuffer[y * DISPLAY_WORDS_PER_ROW + x / 32] &= ~(1u << (x & 31));
35     }
36 }
37
38 // Returning the 5-bit character set of a given character in a given line
39 static uint8_t glyph_row(char c, int row)
40 {
41     // Each character consists of 7 rows, with each row represented by 5 bits
42     static const uint8_t blank[7] = { 0, 0, 0, 0, 0, 0, 0 };
43     static const uint8_t glyphs[][7] = {
44         ['0'] = { 0x0e, 0x11, 0x13, 0x15, 0x19, 0x11, 0x0e },
45         ['1'] = { 0x04, 0x0c, 0x04, 0x04, 0x04, 0x04, 0x0e },
46         ['2'] = { 0x0e, 0x11, 0x01, 0x02, 0x04, 0x08, 0x1f },
47         ['3'] = { 0x1e, 0x01, 0x01, 0x0e, 0x01, 0x01, 0x1e },
48         ['4'] = { 0x02, 0x06, 0x0a, 0x12, 0x1f, 0x02, 0x02 },
49         ['5'] = { 0x1f, 0x10, 0x10, 0x1e, 0x01, 0x01, 0x1e },
50         ['6'] = { 0x0e, 0x10, 0x10, 0x1e, 0x11, 0x11, 0x0e },
51         ['7'] = { 0x1f, 0x01, 0x02, 0x04, 0x08, 0x08, 0x08 },
```

```
        ['8'] = { 0x0e, 0x11, 0x11, 0x0e, 0x11, 0x11, 0x0e },
        ['9'] = { 0x0e, 0x11, 0x11, 0x0f, 0x01, 0x01, 0x0e },
        ['A'] = { 0x0e, 0x11, 0x11, 0x1f, 0x11, 0x11, 0x11 },
        ['B'] = { 0x1e, 0x11, 0x11, 0x1e, 0x11, 0x11, 0x1e },
        ['C'] = { 0x0f, 0x10, 0x10, 0x10, 0x10, 0x10, 0x0f },
        ['D'] = { 0x1e, 0x11, 0x11, 0x11, 0x11, 0x11, 0x1e },
        ['E'] = { 0x1f, 0x10, 0x10, 0x1e, 0x10, 0x10, 0x1f },
        ['F'] = { 0x1f, 0x10, 0x10, 0x1e, 0x10, 0x10, 0x10 },
        ['G'] = { 0x0f, 0x10, 0x10, 0x13, 0x11, 0x11, 0x0f },
        ['H'] = { 0x11, 0x11, 0x11, 0x1f, 0x11, 0x11, 0x11 },
        ['I'] = { 0x0e, 0x04, 0x04, 0x04, 0x04, 0x04, 0x0e },
        ['J'] = { 0x07, 0x02, 0x02, 0x02, 0x12, 0x12, 0x0c },
        ['K'] = { 0x11, 0x12, 0x14, 0x18, 0x14, 0x12, 0x11 },
        ['L'] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x1f },
        ['M'] = { 0x11, 0x1b, 0x15, 0x15, 0x11, 0x11, 0x11 },
        ['N'] = { 0x11, 0x19, 0x15, 0x13, 0x11, 0x11, 0x11 },
        ['O'] = { 0x0e, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0e },
        ['P'] = { 0x1e, 0x11, 0x11, 0x1e, 0x10, 0x10, 0x10 },
        ['Q'] = { 0x0e, 0x11, 0x11, 0x11, 0x15, 0x12, 0x0d },
        ['R'] = { 0x1e, 0x11, 0x11, 0x1e, 0x14, 0x12, 0x11 },
        ['S'] = { 0x0f, 0x10, 0x10, 0x0e, 0x01, 0x01, 0x1e },
        ['T'] = { 0x1f, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04 },
        ['U'] = { 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0e },
        ['V'] = { 0x11, 0x11, 0x11, 0x11, 0x11, 0x0a, 0x04 },
        ['W'] = { 0x11, 0x11, 0x11, 0x15, 0x15, 0x15, 0x0a },
        ['X'] = { 0x11, 0x11, 0x0a, 0x04, 0x0a, 0x11, 0x11 },
        ['Y'] = { 0x11, 0x11, 0x0a, 0x04, 0x04, 0x04, 0x04 },
        ['Z'] = { 0x1f, 0x01, 0x02, 0x04, 0x08, 0x10, 0x1f },
        [':'] = { 0x00, 0x04, 0x04, 0x00, 0x04, 0x04, 0x00 },
        ['/'] = { 0x01, 0x01, 0x02, 0x04, 0x08, 0x10, 0x10 },
        ['['] = { 0x0e, 0x08, 0x08, 0x08, 0x08, 0x08, 0x0e },
        [']'] = { 0x0e, 0x02, 0x02, 0x02, 0x02, 0x02, 0x0e },
        ['|'] = { 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04 },
        ['-'] = { 0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00 },
        [' '] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    };
    const uint8_t *g;

    // Converts lowercase letters to uppercase
    if (c >= 'a' && c <= 'z')
        c = (char)toupper((unsigned char)c);

    // If the character is not supported, display a blank space
    if ((unsigned char) c >= sizeof(glyphs) / sizeof(glyphs[0])) {
        g = blank;
    } else {
        g = glyphs[(unsigned char) c];
    }

    return g[row];
}

// Convert the simulation's world coordinates into screen pixel coordinates
int world_to_screen_x(float x)
{
    float normalized = (x - NBODY_POS_MIN) / (NBODY_POS_MAX - NBODY_POS_MIN);
    return (int) (normalized * (float)(BODY_DISPLAY_W - 1) + 0.5f);
}

// Convert the simulation's world coordinates into screen pixel coordinates
int world_to_screen_y(float y)
{
    float normalized = (y - NBODY_POS_MIN) / (NBODY_POS_MAX - NBODY_POS_MIN);
    return (int) (normalized * (float)(BODY_DISPLAY_H - 1) + 0.5f);
}

// Clear the entire framebuffer to 0, resulting in a black screen
void display_clear(void)
{
    memset(framebuffer, 0, sizeof(framebuffer));
}

/*
```

```
The rendering thread needs to draw numerous bodies in every frame
Recalculating the circular geometry each time a body is drawn is time-consuming
Instead, pre-generate the masks beforehand
then simply perform a direct lookup when rendering
*/
void display_init_bodyshape(void)
{
    memset(body_masks, 0, sizeof(body_masks));

    for (int i = 0; i < NUM_BODY_RADII; i++) {
        int r = BODY_RADII[i];
        for (int y = -r; y <= r; y++)
            for (int x = -r; x <= r; x++)
                if (x * x + y * y < r * r + r)
                    body_masks[i][y + r][x + r] = 1;
    }
}

// Draw a body at position (cx, cy)
void display_draw_body(int cx, int cy, int radius_idx)
{
    int r = BODY_RADII[radius_idx];
    for (int y = -r; y <= r; y++) {
        int sy = cy + y;

        // Check if the coordinates are out of bounds
        if (sy < 0 || sy >= BODY_DISPLAY_H)
            continue;
        for (int x = -r; x <= r; x++)
            if (body_masks[radius_idx][y + r][x + r])
                set_pixel(cx + x, sy, 1);
    }
}

// Draw a character at a specific row and column within a text grid
void display_putchar(char c, int row, int col)
{
    // Check if the coordinates are out of bounds
    if (row < 0 || row >= TEXT_ROWS || col < 0 || col >= TEXT_COLS)
        return;

    // Render 5 x 7 character to 8    16 pixels
    for (int y = 0; y < FONT_HEIGHT; y++) {
        uint8_t bits = 0;

        if (y >= 2 && y < 16)
            bits = glyph_row(c, (y - 2) / 2);

        for (int x = 0; x < FONT_WIDTH; x++) {
            int on = (x >= 1 && x <= 5) && (bits & (1u << (5 - x)));
            set_pixel(col * FONT_WIDTH + x, row * FONT_HEIGHT + y, on);
        }
    }
}

// Draw Strings Continuously
void display_puts(const char *s, int row, int col)
{
    while (*s && col < TEXT_COLS) {
        display_putchar(*s++, row, col++);
    }
}

// Send the software framebuffer to the kernel driver
// If /dev/nbody_display is not open, return -1
int display_present(void)
{
    if (display_fd < 0)
        return -1;
    return ioctl(display_fd, DISPLAY_WRITE_FRAME, framebuffer);
}

// Display Thread
```

```c
void *display_thread(void *arg)
{
    // It is merely there to avoid a compiler warning
    (void) arg;

    // Allocate a temporary array to store the positions of the current frame
    body_pos_t *render_positions = malloc(MAX_BODIES * sizeof(*render_positions));
    if (!render_positions)
        return NULL;

    // Open display device
    display_fd = open("/dev/nbody_display", O_RDWR);
    if (display_fd < 0)
        perror("open /dev/nbody_display");

    // Initialize circular mask for the body
    display_init_bodyshape();

    while (running) {
        // Copy a local variables from the global variables
        pthread_mutex_lock(&state_mutex);
        int local_num = num_bodies;
        int local_gap = current_gap;
        int local_count = h_count;
        int local_view = view_idx;
        int local_paused = is_paused;

        // Select the current frame to be displayed from history frames
        if (local_count > 0) {
            int slot = (local_count < MAX_HISTORY) ? local_view : (h_head + local_view) & (MAX_HISTORY - 1);
            memcpy(render_positions, history[slot], local_num * sizeof(*render_positions));
        }
        pthread_mutex_unlock(&state_mutex);

        // Clear the display framebuffer
        display_clear();

        // If historical frames exist
        if (local_count > 0) {
            for (int i = 0; i < local_num; i++) {
                // Retrieve world x/y coordinates and convert to screen x/y coordinates
                int x = world_to_screen_x(render_positions[i].x);
                int y = world_to_screen_y(render_positions[i].y);
                display_draw_body(x, y, (int)static_masses[i]);
            }
        }

        // Display the UI in last two rows in screen/VGA
        char line1[TEXT_COLS + 1];
        snprintf(line1, sizeof(line1), "Bodies: %4d/%d | Gap: %2d/%d | Frame: %d/%d | Status: %s",
                 local_num, MAX_BODIES, local_gap, NBODY_GAP_MAX, local_view + 1, local_count,
                 local_paused ? "PAUSED" : "RUNNING");
        display_puts(line1, UI_START_ROW, 0);
        display_puts("[SPACE] Play/Pause | [W/S] Gap | [A/D] Frame | [R] Reset | [Q] Exit",
                     UI_START_ROW + 1, 0);

        // Write the entire framebuffer to the display hardware
        display_present();

        // It sleeps for 33.333 ms, the display thread is running at approximately 30 fps
        usleep(33333);
    }

    // Clear the screen once before exiting and close the display device
    if (display_fd >= 0) {
        display_clear();
        display_present();
        close(display_fd);
        display_fd = -1;
    }
```

```
268
269     // Release temporary array
270     free(render_positions);
271
272     return NULL;
273 }
```

A.1.5 display.h

Listing 7: code/Software/display.h

```
1  #ifndef _DISPLAY_H
2  #define _DISPLAY_H
3
4  #include "display_ioctl.h"
5
6  #define FONT_WIDTH 8
7  #define FONT_HEIGHT 16
8  #define TEXT_COLS 80
9  #define TEXT_ROWS 30
10 #define UI_START_ROW 28
11 #define BODY_DISPLAY_H 448
12 #define NUM_BODY_RADII 4
13
14 int world_to_screen_x(float x);
15 int world_to_screen_y(float y);
16 void display_clear(void);
17 void display_init_bodyshape(void);
18 void display_draw_body(int cx, int cy, int radius_idx);
19 void display_putchar(char c, int row, int col);
20 void display_puts(const char *s, int row, int col);
21 int display_present(void);
22 void *display_thread(void *arg);
23
24 #endif
```

A.1.6 keyboard.c

Listing 8: code/Software/keyboard.c

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <time.h>
5
6  #include "keyboard.h"
7  #include "nbody.h"
8  #include "usbkeyboard.h"
9
10 static struct libusb_device_handle *keyboard = NULL;
11 static uint8_t endpoint_address;
12
13 // Open keyboard device
14 int keyboard_open(void)
15 {
16     // This function is implemented in usbkeyboard.c
17     // it scans USB devices to locate a HID keyboard
18     keyboard = openkeyboard(&endpoint_address);
19     if (!keyboard)
20         fprintf(stderr, "Warning: did not find a USB keyboard\n");
21
22     return keyboard != NULL;
23 }
24
25 /*
```

```
This function returns the current monotonic time in milliseconds
It uses CLOCK_MONOTONIC rather than standard wall-clock time
The monotonic clock does not jump or shift when the system time is adjusted
More suitable for timing key-repeat intervals
*/
static unsigned long long monotonic_ms(void)
{
    struct timespec ts;

    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (unsigned long long)ts.tv_sec * 1000ull +
           (unsigned long long)ts.tv_nsec / 1000000ull;
}

// Converting USB HID Keycodes into Program Actions
static void handle_keycode(uint8_t keycode)
{
    // SPACE - Pause/Play
    if (keycode == 0x2c) {
        nbody_toggle_pause();
    }
    // W - Gap Increase
    else if (keycode == 0x1a) {
        nbody_set_gap_delta(1);
    }
    // S - Gap Decrease
    else if (keycode == 0x16) {
        nbody_set_gap_delta(-1);
    }
    // A - Frame Backward
    else if (keycode == 0x04) {
        nbody_show_frame_delta(-1);
    }
    // D - Frame Forward
    else if (keycode == 0x07) {
        nbody_show_frame_delta(1);
    }
    // R - Reset
    else if (keycode == 0x15) {
        nbody_request_reset();
    }
    // Q - Exit
    else if (keycode == 0x14) {
        nbody_request_quit();
    }
}

/* Keyboard Thread
Read a packet from the USB keyboard, waiting for a maximum of 50 ms
If a timeout occurs: if the 'A' or 'D' key is currently being held down
and the repeat interval has elapsed, trigger the corresponding action
Otherwise, proceed to the next iteration
If the read operation fails, proceed to the next iteration
Check if the current packet is identical to the previous packet
If no key is currently pressed, clear the repeat state and proceed to the next iteration
If the key is A/D, enable long-press repeat functionality and proceed to the next iteration
For all other keys: if the current packet is identical to the previous packet, ignore it
Otherwise, process the key press once
*/
void *keyboard_handler(void *arg)
{
    // It is merely there to avoid a compiler warning
    (void) arg;

    /*
    These two values    are    used to scroll through historical frames when holding down A/
    D
    After the initial press of A/D, a 350 ms delay before continuous repetition begins
    // Once repetition starts, a frame-scrolling event is triggered every 35 ms
    */
    enum {
        REPEAT_DELAY_MS = 350,
        REPEAT_INTERVAL_MS = 35,
```

```
    };

    // Data packets read from a USB keyboard
    struct usb_keyboard_packet packet;
    uint8_t last_packet[8] = { 0 };
    uint8_t held_repeat_key = 0;
    unsigned long long next_repeat_ms = 0;

    while (running) {
        /*
        Read a keyboard packet from the USB keyboard's interrupt endpoint
        If no keyboard event occurs within 50 ms, it will time out
        rather than blocking indefinitely
        */
        int transferred = 0;
        int rc = libusb_interrupt_transfer(keyboard, endpoint_address,
                                           (unsigned char *)&packet, sizeof(packet),
                                           &transferred, 50);
        unsigned long long now = monotonic_ms();

        /*
        If no new keyboard packet is received this time, but a  k e y specifically  A/D
        was previously being held down (repeating), then as soon as the designated
        time interval elapses, trigger the action once again
        */
        if (rc == LIBUSB_ERROR_TIMEOUT) {
            if (held_repeat_key && now >= next_repeat_ms) {
                handle_keycode(held_repeat_key);
                next_repeat_ms = now + REPEAT_INTERVAL_MS;
            }
            continue;
        }

        //If a USB transfer error occurs, or if the number of bytes read is incorrect
        if (rc != 0 || transferred != sizeof(packet))
            continue;

        // Determines whether the current packet is identical to the previous one
        // in order to prevent a single key press from being processed repeatedly
        int same_packet = (memcmp(&packet, last_packet, sizeof(packet)) == 0);
        if (!same_packet)
            memcpy(last_packet, &packet, sizeof(packet));

        // Take only the first keycode
        uint8_t keycode = packet.keycode[0];
        if (keycode == 0) {
            held_repeat_key = 0;
            continue;
        }

        // A and D are special because they support long-press repeat
        if (keycode == 0x04 || keycode == 0x07) {
            if (!same_packet || held_repeat_key != keycode) {
                handle_keycode(keycode);
                held_repeat_key = keycode;
                next_repeat_ms = now + REPEAT_DELAY_MS;
            } else if (now >= next_repeat_ms) {
                handle_keycode(keycode);
                next_repeat_ms = now + REPEAT_INTERVAL_MS;
            }
            continue;
        }

        // If the current packet is identical to the previous one
        // it indicates that the key is still being held down
        // then do not trigger a duplicate event
        held_repeat_key = 0;
        if (same_packet)
            continue;

        handle_keycode(keycode);
    }

```

```
171     return NULL;
172 }
```

A.1.7 keyboard.h

Listing 9: code/Software/keyboard.h

```
1  #ifndef _KEYBOARD_H
2  #define _KEYBOARD_H
3
4  int keyboard_open(void);
5  void *keyboard_handler(void *arg);
6
7  #endif
```

A.1.8 usbkeyboard.c

Listing 10: code/Software/usbkeyboard.c

```
1  #include "usbkeyboard.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* References on libusb 1.0 and the USB HID/keyboard protocol
7   *
8   * http://libusb.org
9   * https://web.archive.org/web/20210302095553/https://www.dreamincode.net/forums/topic
       /148707-introduction-to-using-libusb-10/
10  *
11  * https://www.usb.org/sites/default/files/documents/hid1_11.pdf
12  *
13  * https://usb.org/sites/default/files/hut1_5.pdf
14  */
15
16 /*
17  * Find and return a USB keyboard device or NULL if not found
18  * The argument con
19  *
20  */
21 struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {
22   libusb_device **devs;
23   struct libusb_device_handle *keyboard = NULL;
24   struct libusb_device_descriptor desc;
25   ssize_t num_devs, d;
26   uint8_t i, k;
27
28   /* Start the library */
29   if ( libusb_init(NULL) < 0 ) {
30     fprintf(stderr, "Error: libusb_init failed\n");
31     exit(1);
32   }
33
34   /* Enumerate all the attached USB devices */
35   if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
36     fprintf(stderr, "Error: libusb_get_device_list failed\n");
37     exit(1);
38   }
39
40   /* Look at each device, remembering the first HID device that speaks
41      the keyboard protocol */
42
43   for (d = 0 ; d < num_devs ; d++) {
44     libusb_device *dev = devs[d];
45     if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
```

```
46        fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
47        exit(1);
48      }
49
50      if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
51        struct libusb_config_descriptor *config;
52        libusb_get_config_descriptor(dev, 0, &config);
53        for (i = 0 ; i < config->bNumInterfaces ; i++)
54      for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
55        const struct libusb_interface_descriptor *inter =
56          config->interface[i].altsetting + k ;
57        if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
58             inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {
59          int r;
60          if ((r = libusb_open(dev, &keyboard)) != 0) {
61            fprintf(stderr, "Error: libusb_open failed: %d\n", r);
62            exit(1);
63          }
64          if (libusb_kernel_driver_active(keyboard,i))
65            libusb_detach_kernel_driver(keyboard, i);
66          libusb_set_auto_detach_kernel_driver(keyboard, i);
67          if ((r = libusb_claim_interface(keyboard, i)) != 0) {
68            fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
69            exit(1);
70          }
71          *endpoint_address = inter->endpoint[0].bEndpointAddress;
72          goto found;
73        }
74      }
75      }
76    }
77
78   found:
79    libusb_free_device_list(devs, 1);
80
81    return keyboard;
82  }
```

A.1.9 usbkeyboard.h

Listing 11: code/Software/usbkeyboard.h

```
1  #ifndef _USBKEYBOARD_H
2  #define _USBKEYBOARD_H
3
4  #include <libusb-1.0/libusb.h>
5
6  #define USB_HID_KEYBOARD_PROTOCOL 1
7
8  /* Modifier bits */
9  #define USB_LCTRL  (1 << 0)
10 #define USB_LSHIFT (1 << 1)
11 #define USB_LALT   (1 << 2)
12 #define USB_LGUI   (1 << 3)
13 #define USB_RCTRL  (1 << 4)
14 #define USB_RSHIFT (1 << 5)
15 #define USB_RALT   (1 << 6)
16 #define USB_RGUI   (1 << 7)
17
18 struct usb_keyboard_packet {
19   uint8_t modifiers;
20   uint8_t reserved;
21   uint8_t keycode[6];
22 };
23
24 /* Find and open a USB keyboard device.  Argument should point to
25    space to store an endpoint address.  Returns NULL if no keyboard
26    device was found. */
27 extern struct libusb_device_handle *openkeyboard(uint8_t *);
```

```
28
29 #endif
```

A.1.10 accelerator_driver.c

Listing 12: code/Software/accelerator_driver.c

```
1  /*
2   * N-body Avalon-MM Linux driver.
3   *
4   * This follows nbody_accel_avmm.sv exactly. Register addresses below are
5   * byte offsets for ioread32/iowrite32; the hardware Avalon address is a
6   * 32-bit word address, so every register is word_address * 4.
7   */
8
9  #include <linux/errno.h>
10 #include <linux/fs.h>
11 #include <linux/init.h>
12 #include <linux/io.h>
13 #include <linux/kernel.h>
14 #include <linux/miscdevice.h>
15 #include <linux/module.h>
16 #include <linux/of.h>
17 #include <linux/of_address.h>
18 #include <linux/platform_device.h>
19 #include <linux/slab.h>
20 #include <linux/string.h>
21 #include <linux/uaccess.h>
22 #include "nbody_ioctl.h"
23
24 #define DRIVER_NAME "nbody"
25
26 #define REG_GO        (0x00 * 4)
27 #define REG_N_BODIES  (0x01 * 4)
28 #define REG_GAP       (0x02 * 4)
29 #define REG_X_IN      (0x03 * 4)
30 #define REG_Y_IN      (0x04 * 4)
31 #define REG_M_IN      (0x05 * 4)
32 #define REG_VX_IN     (0x06 * 4)
33 #define REG_VY_IN     (0x07 * 4)
34 #define REG_DONE      (0x08 * 4)
35 #define REG_READ      (0x09 * 4)
36 #define REG_OUT_X     (0x0A * 4)
37 #define REG_OUT_Y     (0x0B * 4)
38
39 struct nbody_dev {
40     struct resource res;
41     void __iomem *virtbase;
42     uint32_t num_bodies;
43     uint32_t gap;
44 };
45
46 static struct nbody_dev dev;
47
48 /*
49  * Convert a userspace IEEE-754 single-precision float into the accelerator's
50  * custom S1E8M18 27-bit floating-point format.
51  *
52  * The converted value is stored in the lower 27 bits of a 32-bit Avalon word.
53  * The upper 5 bits are unused padding.
54  */
55 static uint32_t f32_to_f27_bits(const float *value)
56 {
57     uint32_t bits;
58     uint32_t sign;
59     uint32_t exp;
60     uint32_t mant;
61     uint32_t rounded_mant;
62
```

```
    memcpy(&bits, value, sizeof(bits));

    sign = (bits >> 31) & 1u;
    exp = (bits >> 23) & 0xffu;
    mant = bits & 0x007fffffu;

    if (exp == 0)
        return 0;
    if (exp == 0xffu)
        return (sign << NBODY_F27_SIGN_SHIFT) | (0xfeu << NBODY_F27_EXP_SHIFT) |
               NBODY_F27_MANT_MASK;

    rounded_mant = (mant + 0x10u) >> 5;
    if (rounded_mant == (1u << 18)) {
        rounded_mant = 0;
        exp++;
        if (exp >= 0xffu) {
            exp = 0xfeu;
            rounded_mant = NBODY_F27_MANT_MASK;
        }
    }

    return (sign << NBODY_F27_SIGN_SHIFT) |
           (exp << NBODY_F27_EXP_SHIFT) |
           (rounded_mant & NBODY_F27_MANT_MASK);
}

/*
 * Convert a raw 27-bit S1E8M18 hardware result back into an IEEE-754
 * single-precision float for userspace.
 */
static void f27_bits_to_f32(uint32_t raw, float *value)
{
    uint32_t bits;
    uint32_t sign;
    uint32_t exp;
    uint32_t mant;

    raw &= NBODY_DATA_MASK;
    if (raw == 0) {
        bits = 0;
    } else {
        sign = (raw >> NBODY_F27_SIGN_SHIFT) & 1u;
        exp = (raw >> NBODY_F27_EXP_SHIFT) & 0xffu;
        mant = raw & NBODY_F27_MANT_MASK;

        if (exp == 0) {
            bits = 0;
        } else {
            bits = (sign << 31) | (exp << 23) | (mant << 5);
        }
    }

    memcpy(value, &bits, sizeof(bits));
}

/*
 * Copy the initial particle array from userspace, convert each field from
 * float32 to S1E8M18, and stream the data into the hardware input registers.
 *
 * The hardware commits one complete body when VY_IN is written, so the fields
 * must be written in the expected X, Y, mass, VX, VY order.
 */
static int nbody_write_bodies(unsigned long arg)
{
    nbody_bodies_arg_t barg;
    nbody_particle_t *particles;
    size_t bytes;
    uint32_t i;

    if (copy_from_user(&barg, (void __user *)arg, sizeof(barg)))
        return -EFAULT;

```

```
    if (barg.count == 0 || barg.count > NBODY_MAX_BODIES || !barg.particles)
        return -EINVAL;

    if (dev.num_bodies == 0 || barg.count != dev.num_bodies)
        return -EINVAL;

    bytes = barg.count * sizeof(*particles);
    particles = memdup_user((const void __user *)barg.particles, bytes);
    if (IS_ERR(particles))
        return PTR_ERR(particles);

    /*
     * Userspace passes IEEE-754 single-precision values. Hardware payloads are
     * rounded to S1E8M18 in the low 27 bits of each 32-bit Avalon word; the
     * upper 5 bits are zero padding. Writing VY_IN commits the current body
     * and advances input_ptr.
     */
    for (i = 0; i < barg.count; i++) {
        iowrite32(f32_to_f27_bits(&particles[i].x), dev.virtbase + REG_X_IN);
        iowrite32(f32_to_f27_bits(&particles[i].y), dev.virtbase + REG_Y_IN);
        iowrite32(f32_to_f27_bits(&particles[i].mass), dev.virtbase + REG_M_IN);
        iowrite32(f32_to_f27_bits(&particles[i].vx), dev.virtbase + REG_VX_IN);
        iowrite32(f32_to_f27_bits(&particles[i].vy), dev.virtbase + REG_VY_IN);
    }

    kfree(particles);
    return 0;
}

/*
 * Read computed body positions from the accelerator output registers.
 *
 * Writing READ=1 prepares the hardware output stream. For each body, OUT_X is
 * read first and OUT_Y is read second. The hardware output pointer advances
 * only after OUT_Y is read.
 */
static int nbody_read_results(unsigned long arg)
{
    nbody_read_arg_t rarg;
    nbody_result_t *results;
    size_t bytes;
    uint32_t i;
    int ret = 0;

    if (copy_from_user(&rarg, (void __user *)arg, sizeof(rarg)))
        return -EFAULT;

    if (rarg.count == 0 || rarg.count > NBODY_MAX_BODIES ||
        rarg.count > dev.num_bodies || !rarg.results)
        return -EINVAL;

    bytes = rarg.count * sizeof(*results);
    results = kmalloc(bytes, GFP_KERNEL);
    if (!results)
        return -ENOMEM;

    iowrite32(1, dev.virtbase + REG_READ);

    /*
     * Read OUT_X first, then OUT_Y. Only the OUT_Y read increments the
     * hardware output pointer.
     */
    for (i = 0; i < rarg.count; i++) {
        f27_bits_to_f32(ioread32(dev.virtbase + REG_OUT_X), &results[i].x);
        f27_bits_to_f32(ioread32(dev.virtbase + REG_OUT_Y), &results[i].y);
    }

    if (copy_to_user((void __user *)rarg.results, results, bytes))
        ret = -EFAULT;

    kfree(results);
    return ret;
}
```

```
209
210 /*
211  * Main ioctl dispatch function for /dev/nbody.
212  *
213  * Each command corresponds to one userspace accelerator operation, such as
214  * writing configuration, loading input bodies, starting a run, checking DONE,
215  * reading output positions, or clearing the hardware read state.
216  */
217 static long nbody_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
218 {
219     nbody_config_t cfg;
220     int done;
221
222     switch (cmd) {
223     case NBODY_WRITE_CONFIG:
224         if (copy_from_user(&cfg, (void __user *)arg, sizeof(cfg)))
225             return -EFAULT;
226         if (cfg.num_bodies > NBODY_MAX_BODIES || cfg.gap == 0)
227             return -EINVAL;
228
229         dev.num_bodies = cfg.num_bodies;
230         dev.gap = cfg.gap;
231         iowrite32(cfg.num_bodies, dev.virtbase + REG_N_BODIES);
232         iowrite32(cfg.gap, dev.virtbase + REG_GAP);
233         return 0;
234
235     case NBODY_WRITE_BODIES:
236         return nbody_write_bodies(arg);
237
238     case NBODY_START_RUN:
239         iowrite32(1, dev.virtbase + REG_GO);
240         return 0;
241
242     case NBODY_CHECK_DONE:
243         done = ioread32(dev.virtbase + REG_DONE) & 1;
244         if (copy_to_user((int __user *)arg, &done, sizeof(done)))
245             return -EFAULT;
246         return 0;
247
248     case NBODY_READ_RESULTS:
249         return nbody_read_results(arg);
250
251     case NBODY_CLEAR_READ:
252         iowrite32(0, dev.virtbase + REG_READ);
253         return 0;
254
255     case NBODY_STOP:
256         /* GO is pulse-based in SV; best-effort stop is clearing READ. */
257         iowrite32(0, dev.virtbase + REG_READ);
258         return 0;
259
260     case NBODY_SOFT_RESET:
261         /*
262          * There is no full software reset register. Clear READ/output_ptr;
263          * userspace must reload config and bodies for a fresh simulation.
264          */
265         iowrite32(0, dev.virtbase + REG_READ);
266         return 0;
267
268     default:
269         return -EINVAL;
270     }
271 }
272
273 static const struct file_operations nbody_fops = {
274     .owner = THIS_MODULE,
275     .unlocked_ioctl = nbody_ioctl,
276 };
277
278 static struct miscdevice nbody_misc_device = {
279     .minor = MISC_DYNAMIC_MINOR,
280     .name = "nbody",
281     .fops = &nbody_fops,
```

```
};

/*
 * Platform driver probe function.
 *
 * This function is called when the matching device-tree node is found. It maps
 * the accelerator register space, registers /dev/nbody as a misc device, and
 * initializes basic hardware control registers.
 */
static int nbody_probe(struct platform_device *pdev)
{
    int ret;

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret)
        return ret;

    if (!request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME))
        return -EBUSY;

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (!dev.virtbase) {
        ret = -ENOMEM;
        goto out_release;
    }

    ret = misc_register(&nbody_misc_device);
    if (ret)
        goto out_unmap;

    iowrite32(0, dev.virtbase + REG_READ);
    iowrite32(1, dev.virtbase + REG_GAP);
    pr_info("nbody driver registered /dev/nbody\n");
    return 0;

out_unmap:
    iounmap(dev.virtbase);
out_release:
    release_mem_region(dev.res.start, resource_size(&dev.res));
    return ret;
}

/*
 * Platform driver remove function.
 *
 * Release the misc device, unmap the hardware register space, and free the
 * reserved memory region.
 */
static int nbody_remove(struct platform_device *pdev)
{
    misc_deregister(&nbody_misc_device);
    if (dev.virtbase)
        iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    return 0;
}

static const struct of_device_id nbody_of_match[] = {
    { .compatible = "csee4840,nbody_accel_avmm-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, nbody_of_match);

static struct platform_driver nbody_driver = {
    .probe = nbody_probe,
    .remove = nbody_remove,
    .driver = {
        .name = DRIVER_NAME,
        .of_match_table = nbody_of_match,
    },
};

module_platform_driver(nbody_driver);
```

```
355
356 MODULE_LICENSE("GPL");
357 MODULE_AUTHOR("CSEE4840 Group");
358 MODULE_DESCRIPTION("N-body Avalon-MM accelerator driver");
```

A.1.11 nbody_ioctl.h

Listing 13: code/Software/nbody_ioctl.h

```
1  #ifndef _NBODY_IOCTL_H
2  #define _NBODY_IOCTL_H
3
4  #ifdef __KERNEL__
5  #include <linux/ioctl.h>
6  #include <linux/types.h>
7  #else
8  #include <sys/ioctl.h>
9  #include <stdint.h>
10 #endif
11
12 #define NBODY_MAX_BODIES 1024
13 #define NBODY_DATA_W 27
14 #define NBODY_DATA_MASK 0x07ffffffu
15 #define NBODY_F27_SIGN_SHIFT 26
16 #define NBODY_F27_EXP_SHIFT 18
17 #define NBODY_F27_MANT_MASK 0x0003ffffu
18 #define NBODY_MAGIC 'n'
19
20 typedef struct {
21     float x;
22     float y;
23     float mass;
24     float vx;
25     float vy;
26 } nbody_particle_t;
27
28 typedef struct {
29     float x;
30     float y;
31 } nbody_result_t;
32
33 typedef nbody_result_t body_pos_t;
34
35 typedef struct {
36     uint32_t num_bodies;
37     uint32_t gap;
38 } nbody_config_t;
39
40 typedef struct {
41     const nbody_particle_t *particles;
42     uint32_t count;
43 } nbody_bodies_arg_t;
44
45 typedef struct {
46     nbody_result_t *results;
47     uint32_t count;
48 } nbody_read_arg_t;
49
50 #define NBODY_WRITE_CONFIG   _IOW(NBODY_MAGIC, 1, nbody_config_t)
51 #define NBODY_WRITE_BODIES   _IOW(NBODY_MAGIC, 2, nbody_bodies_arg_t)
52 #define NBODY_START_RUN      _IO(NBODY_MAGIC, 3)
53 #define NBODY_CHECK_DONE     _IOR(NBODY_MAGIC, 4, int)
54 #define NBODY_READ_RESULTS   _IOWR(NBODY_MAGIC, 5, nbody_read_arg_t)
55 #define NBODY_CLEAR_READ     _IO(NBODY_MAGIC, 6)
56 #define NBODY_STOP           _IO(NBODY_MAGIC, 7)
57 #define NBODY_SOFT_RESET     _IO(NBODY_MAGIC, 8)
58
59 #define NBODY_WRITE_INIT_DATA NBODY_WRITE_BODIES
60 #define NBODY_READ_POSITIONS  NBODY_READ_RESULTS
```

```
61
62 #endif
```

A.1.12 display_driver.c

Listing 14: code/Software/display_driver.c

```
1  /*
2   * Packed bitmap display Avalon-MM Linux driver.
3   *
4   * The kernel only transfers the userspace-rendered 640x480 1-bpp framebuffer
5   * to vga_bitmap_avmm.sv. It does not render particles or text.
6   */
7
8  #include <linux/errno.h>
9  #include <linux/fs.h>
10 #include <linux/init.h>
11 #include <linux/io.h>
12 #include <linux/kernel.h>
13 #include <linux/miscdevice.h>
14 #include <linux/module.h>
15 #include <linux/of.h>
16 #include <linux/of_address.h>
17 #include <linux/platform_device.h>
18 #include <linux/slab.h>
19 #include <linux/uaccess.h>
20 #include "display_ioctl.h"
21
22 #define DRIVER_NAME "nbody_display"
23
24 struct display_dev {
25     struct resource res;
26     void __iomem *virtbase;
27     uint32_t *kbuf;
28 };
29
30 static struct display_dev dev;
31
32 /*
33  * Main ioctl dispatch function for /dev/nbody_display.
34  *
35  * DISPLAY_WRITE_FRAME copies a full packed framebuffer from userspace and
36  * writes it into the hardware framebuffer. DISPLAY_CLEAR clears every hardware
37  * framebuffer word to zero.
38  */
39 static long display_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
40 {
41     uint32_t i;
42
43     switch (cmd) {
44     case DISPLAY_WRITE_FRAME:
45         if (copy_from_user(dev.kbuf, (uint32_t __user *)arg,
46                            DISPLAY_WORDS * sizeof(uint32_t)))
47             return -EFAULT;
48
49         /*
50          * vga_bitmap_avmm.sv uses word addresses 0..9599, so byte offset
51          * is i * 4. Framebuffer packing is word = y * 20 + x / 32,
52          * bit = x % 32, LSB-first.
53          */
54         for (i = 0; i < DISPLAY_WORDS; i++)
55             iowrite32(dev.kbuf[i], dev.virtbase + i * 4);
56         return 0;
57
58     case DISPLAY_CLEAR:
59         for (i = 0; i < DISPLAY_WORDS; i++)
60             iowrite32(0, dev.virtbase + i * 4);
61         return 0;
62
```

```
    default:
        return -EINVAL;
    }
}

static const struct file_operations display_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = display_ioctl,
};

static struct miscdevice display_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "nbody_display",
    .fops = &display_fops,
};

/*
 * Platform driver probe function.
 *
 * This function is called when the matching display device-tree node is found.
 * It maps the Avalon-MM framebuffer, allocates a kernel-side frame buffer, and
 * registers /dev/nbody_display as a misc device.
 */
static int display_probe(struct platform_device *pdev)
{
    int ret;

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret)
        return ret;

    if (!request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME))
        return -EBUSY;

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (!dev.virtbase) {
        ret = -ENOMEM;
        goto out_release;
    }

    dev.kbuf = kmalloc(DISPLAY_WORDS * sizeof(uint32_t), GFP_KERNEL);
    if (!dev.kbuf) {
        ret = -ENOMEM;
        goto out_unmap;
    }

    ret = misc_register(&display_misc_device);
    if (ret)
        goto out_free;

    pr_info("display driver registered /dev/nbody_display\n");
    return 0;

out_free:
    kfree(dev.kbuf);
out_unmap:
    iounmap(dev.virtbase);
out_release:
    release_mem_region(dev.res.start, resource_size(&dev.res));
    return ret;
}

/*
 * Platform driver remove function.
 *
 * Release /dev/nbody_display, free the kernel frame buffer, unmap the hardware
 * framebuffer, and release the memory region.
 */
static int display_remove(struct platform_device *pdev)
{
    misc_deregister(&display_misc_device);
    kfree(dev.kbuf);
    if (dev.virtbase)
```

```
        iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    return 0;
}

static const struct of_device_id display_of_match[] = {
    { .compatible = "csee4840,vga_bitmap_avmm-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, display_of_match);

static struct platform_driver display_driver = {
    .probe = display_probe,
    .remove = display_remove,
    .driver = {
        .name = DRIVER_NAME,
        .of_match_table = display_of_match,
    },
};

module_platform_driver(display_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CSEE4840 Group");
MODULE_DESCRIPTION("Packed VGA bitmap Avalon-MM display driver");
```

A.1.13 display_ioctl.h

Listing 15: code/Software/display_ioctl.h

```
#ifndef _DISPLAY_IOCTL_H
#define _DISPLAY_IOCTL_H

#ifdef __KERNEL__
#include <linux/ioctl.h>
#include <linux/types.h>
#else
#include <sys/ioctl.h>
#include <stdint.h>
#endif

#define DISPLAY_WIDTH 640
#define DISPLAY_HEIGHT 480
#define DISPLAY_WORDS_PER_ROW 20
#define DISPLAY_WORDS 9600

#define DISPLAY_MAGIC 'd'

#define DISPLAY_WRITE_FRAME _IOW(DISPLAY_MAGIC, 1, uint32_t *)
#define DISPLAY_CLEAR       _IO(DISPLAY_MAGIC, 2)

#endif
```

A.1.14 Makefile

Listing 16: code/Software/Makefile

```
ifneq ($(KERNELRELEASE),)

obj-m += accelerator_driver.o
obj-m += display_driver.o

else

CC ?= cc
```

```
KDIR ?= /usr/src/linux-headers-$(shell uname -r)
PWD := $(shell pwd)

CFLAGS += -Wall -Wextra -pthread
LDLIBS += -lusb-1.0 -pthread

APP = nbody_app
OBJECTS = main.o nbody.o display.o keyboard.o usbkeyboard.o

TARFILES = Makefile main.c nbody.c nbody.h nbody_ioctl.h \
        display.h display.c display_ioctl.h \
        keyboard.h keyboard.c \
        usbkeyboard.h usbkeyboard.c accelerator_driver.c display_driver.c

.PHONY: all app modules modules-clean clean distclean

all: app

app: $(APP)

$(APP): $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LDLIBS)

main.o: main.c display.h keyboard.h nbody.h nbody_ioctl.h display_ioctl.h
nbody.o: nbody.c nbody.h nbody_ioctl.h
display.o: display.c display.h display_ioctl.h nbody.h nbody_ioctl.h
keyboard.o: keyboard.c keyboard.h usbkeyboard.h nbody.h nbody_ioctl.h
usbkeyboard.o: usbkeyboard.c usbkeyboard.h

modules:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

modules-clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

nbody_app.tar.gz: $(TARFILES)
    rm -rf nbody_app_dir
    mkdir nbody_app_dir
    ln $(TARFILES) nbody_app_dir
    tar zcf nbody_app.tar.gz nbody_app_dir
    rm -rf nbody_app_dir

clean:
    rm -rf $(OBJECTS) $(APP) nbody_app.tar.gz nbody_app_dir

distclean: clean modules-clean

endif
```

A.2 Software Test Code

A.2.1 avmm_smoke_test.c

Listing 17: code/Software/test/avmm_smoke_test.c

```
/*
 * avmm_smoke_test.c
 *
 * Minimal Linux userspace smoke test for DE1-SoC HPS -> FPGA lightweight
 * Avalon-MM peripherals:
 *   - nbody_accel_avmm
 *   - vga_bitmap_avmm
 *
 * Build on the board:
 *   gcc -O2 -Wall -Wextra -o avmm_smoke_test avmm_smoke_test.c
 *
 * Run as root / with permission to open /dev/mem:
 *   ./avmm_smoke_test <nbody_offset_hex> <vga_offset_hex>
```

```
 */

#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define LW_BRIDGE_BASE 0xFF200000u
#define LW_BRIDGE_SPAN 0x00200000u

/* nbody_accel_avmm register word indices */
enum {
    NB_GO       = 0x00,
    NB_N_BODIES = 0x01,
    NB_GAP      = 0x02,
    NB_X_IN     = 0x03,
    NB_Y_IN     = 0x04,
    NB_M_IN     = 0x05,
    NB_VX_IN    = 0x06,
    NB_VY_IN    = 0x07,
    NB_DONE     = 0x08,
    NB_READ     = 0x09,
    NB_OUT_X    = 0x0A,
    NB_OUT_Y    = 0x0B,
};

/* vga_bitmap_avmm constants */
#define VGA_WIDTH         640u
#define VGA_HEIGHT        480u
#define VGA_WORDS_PER_ROW 20u
#define VGA_FB_WORDS      (VGA_HEIGHT * VGA_WORDS_PER_ROW)

#define FP27_SIGN_SHIFT 26u
#define FP27_EXP_SHIFT  18u
#define FP27_MANT_MASK  0x0003FFFFu

static inline void mmio_write(volatile uint32_t *base, uint32_t word_index, uint32_t value)
    {
    base[word_index] = value;
    __sync_synchronize();
}

static inline uint32_t mmio_read(volatile uint32_t *base, uint32_t word_index) {
    uint32_t value = base[word_index];
    __sync_synchronize();
    return value;
}

static uint32_t f32_to_f27(float value) {
    uint32_t bits;
    uint32_t sign;
    uint32_t exp;
    uint32_t mant;
    uint32_t rounded_mant;

    memcpy(&bits, &value, sizeof(bits));
    sign = (bits >> 31) & 1u;
    exp = (bits >> 23) & 0xffu;
    mant = bits & 0x007fffffu;

    if (exp == 0u) {
        return 0u;
    }
    if (exp == 0xffu) {
        return (sign << FP27_SIGN_SHIFT) | (0xfeu << FP27_EXP_SHIFT) | FP27_MANT_MASK;
    }

```

```
    rounded_mant = (mant + 0x10u) >> 5;
    if (rounded_mant == (1u << 18)) {
        rounded_mant = 0u;
        exp++;
        if (exp >= 0xffu) {
            exp = 0xfeu;
            rounded_mant = FP27_MANT_MASK;
        }
    }

    return (sign << FP27_SIGN_SHIFT) | (exp << FP27_EXP_SHIFT) |
           (rounded_mant & FP27_MANT_MASK);
}

static uint32_t parse_hex_arg(const char *s, const char *name) {
    char *end = NULL;
    errno = 0;
    unsigned long value = strtoul(s, &end, 0);
    if (errno != 0 || end == s || *end != '\0' || value > 0xFFFFFFFFul) {
        fprintf(stderr, "Bad %s: %s\n", name, s);
        exit(2);
    }
    if ((value & 0x3u) != 0) {
        fprintf(stderr, "Warning: %s is not 4-byte aligned: 0x%08lx\n", name, value);
    }
    return (uint32_t)value;
}

static bool poll_done(volatile uint32_t *nb, unsigned max_iters) {
    for (unsigned i = 0; i < max_iters; ++i) {
        if (mmio_read(nb, NB_DONE) & 0x1u) {
            return true;
        }
    }
    return false;
}

static void vga_clear(volatile uint32_t *vga, uint32_t value) {
    for (uint32_t i = 0; i < VGA_FB_WORDS; ++i) {
        mmio_write(vga, i, value);
    }
}

static void vga_draw_test_pattern(volatile uint32_t *vga) {
    /* Clear screen. */
    vga_clear(vga, 0x00000000u);

    /* Draw a simple border plus a diagonal-ish line. */
    for (uint32_t y = 0; y < VGA_HEIGHT; ++y) {
        for (uint32_t xword = 0; xword < VGA_WORDS_PER_ROW; ++xword) {
            uint32_t word = 0;
            uint32_t x0 = xword * 32u;
            for (uint32_t b = 0; b < 32u; ++b) {
                uint32_t x = x0 + b;
                bool border = (x == 0u || x == VGA_WIDTH - 1u || y == 0u || y == VGA_HEIGHT
    - 1u);
                bool diag = (x < VGA_WIDTH && (x / 2u) == y);
                if (border || diag) {
                    word |= (1u << b); /* LSB-first, matching hardware comment. */
                }
            }
            mmio_write(vga, y * VGA_WORDS_PER_ROW + xword, word);
        }
    }
}

static int test_nbody_zero_body_done(volatile uint32_t *nb) {
    printf("[nbody] zero-body GO/DONE/READ smoke test...\n");

    mmio_write(nb, NB_READ, 0u);
    mmio_write(nb, NB_N_BODIES, 0u);
    mmio_write(nb, NB_GAP, 1u);
    mmio_write(nb, NB_GO, 1u);
```

```
158
159     if (!poll_done(nb, 1000000u)) {
160         fprintf(stderr, "[FAIL] DONE did not go high for N_BODIES=0.\n");
161         return 1;
162     }
163     printf("[pass] DONE went high.\n");
164
165     mmio_write(nb, NB_READ, 0u);
166     for (unsigned i = 0; i < 1000u; ++i) {
167         if ((mmio_read(nb, NB_DONE) & 0x1u) == 0u) {
168             printf("[pass] DONE cleared after READ=0.\n");
169             return 0;
170         }
171     }
172
173     fprintf(stderr, "[FAIL] DONE did not clear after READ=0.\n");
174     return 1;
175 }
176
177 static int test_nbody_one_body_output(volatile uint32_t *nb) {
178     printf("[nbody] one-body input/output smoke test...\n");
179
180     /* Load one body: x=1.0, y=2.0, m=1.0, vx=0, vy=0. */
181     const uint32_t x_in = f32_to_f27(1.0f);
182     const uint32_t y_in = f32_to_f27(2.0f);
183     const uint32_t m_in = f32_to_f27(1.0f);
184     const uint32_t zero = f32_to_f27(0.0f);
185
186     mmio_write(nb, NB_READ, 0u);
187     mmio_write(nb, NB_N_BODIES, 1u);
188     mmio_write(nb, NB_GAP, 1u);
189     mmio_write(nb, NB_X_IN, x_in);
190     mmio_write(nb, NB_Y_IN, y_in);
191     mmio_write(nb, NB_M_IN, m_in);
192     mmio_write(nb, NB_VX_IN, zero);
193     mmio_write(nb, NB_VY_IN, zero); /* commits body 0 */
194     mmio_write(nb, NB_GO, 1u);
195
196     if (!poll_done(nb, 10000000u)) {
197         fprintf(stderr, "[FAIL] DONE did not go high for N_BODIES=1.\n");
198         return 1;
199     }
200
201     uint32_t out_x = mmio_read(nb, NB_OUT_X) & 0x07FFFFFFu;
202     uint32_t out_y = mmio_read(nb, NB_OUT_Y) & 0x07FFFFFFu; /* increments output pointer */
203
204     printf("[info] OUT_X = 0x%08" PRIx32 ", OUT_Y = 0x%08" PRIx32 "\n", out_x, out_y);
205     printf("[info] Expected for current N=1 zero-velocity smoke case is usually x=0x%08x, y
    =0x%08x.\n",
206            x_in, y_in);
207
208     if (out_x == x_in && out_y == y_in) {
209         printf("[pass] one-body output matches input position.\n");
210     } else {
211         printf("[warn] one-body output did not exactly match. This may indicate either a
    real issue or that the integrator/first-step path changed the value.\n");
212     }
213
214     mmio_write(nb, NB_READ, 0u);
215     return 0;
216 }
217
218 int main(int argc, char **argv) {
219     if (argc != 3) {
220         fprintf(stderr,
221                 "Usage: %s <nbody_offset_hex> <vga_offset_hex>\n"
222                 "Example: %s 0x00000 0x01000\n",
223                 argv[0], argv[0]);
224         return 2;
225     }
226
227     uint32_t nbody_offset = parse_hex_arg(argv[1], "nbody offset");
228     uint32_t vga_offset   = parse_hex_arg(argv[2], "vga offset");
```

```
229
230     if (nbody_offset >= LW_BRIDGE_SPAN || vga_offset >= LW_BRIDGE_SPAN) {
231         fprintf(stderr, "Offsets must be inside LW bridge span 0x%08x.\n", LW_BRIDGE_SPAN);
232         return 2;
233     }
234
235     int fd = open("/dev/mem", O_RDWR | O_SYNC);
236     if (fd < 0) {
237         perror("open /dev/mem");
238         return 1;
239     }
240
241     void *map = mmap(NULL, LW_BRIDGE_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
    LW_BRIDGE_BASE);
242     if (map == MAP_FAILED) {
243         perror("mmap lightweight bridge");
244         close(fd);
245         return 1;
246     }
247
248     volatile uint32_t *nbody = (volatile uint32_t *)((volatile uint8_t *)map + nbody_offset)
    ;
249     volatile uint32_t *vga   = (volatile uint32_t *)((volatile uint8_t *)map + vga_offset);
250
251     printf("LW bridge base: 0x%08x\n", LW_BRIDGE_BASE);
252     printf("nbody offset  : 0x%08" PRIx32 "\n", nbody_offset);
253     printf("vga offset    : 0x%08" PRIx32 "\n", vga_offset);
254
255     int failures = 0;
256
257     /* VGA readback is intentionally always zero in current RTL. */
258     printf("[vga] readback check: word0=0x%08" PRIx32 " word123=0x%08" PRIx32 " (current RTL
     should read 0)\n",
259            mmio_read(vga, 0), mmio_read(vga, 123));
260     printf("[vga] drawing border + diagonal test pattern...\n");
261     vga_draw_test_pattern(vga);
262     printf("[vga] done. Check monitor for white border and diagonal on black background.\n")
    ;
263
264     failures += test_nbody_zero_body_done(nbody);
265     failures += test_nbody_one_body_output(nbody);
266
267     munmap(map, LW_BRIDGE_SPAN);
268     close(fd);
269
270     if (failures == 0) {
271         printf("All smoke tests completed.\n");
272         return 0;
273     }
274
275     fprintf(stderr, "%d smoke test(s) failed.\n", failures);
276     return 1;
277 }
```

A.2.2 avmm_frame_xy_dump.c

Listing 18: code/Software/test/avmm_frame_xy_dump.c

```
1  #define _DEFAULT_SOURCE
2
3  /*
4   * avmm_frame_xy_dump.c
5   *
6   * Board-side N-body Avalon-MM test that loads a frame-input txt file, runs one
7   * accelerator pass, and writes the resulting X/Y positions to a txt file for
8   * comparison against a golden output.
9   *
10  * Input format matches the txt files in Hardware/tb/frame_input:
11  *   # i  px  py  vx  vy  m   (S1E8M18 hex)
```

```
 *      0  20CE1A0  2156136  0000000  0000000  1FA8587
 *
 * Build on the board:
 *   gcc -O2 -Wall -Wextra -std=c11 -o avmm_frame_xy_dump avmm_frame_xy_dump.c
 *
 * Run as root / with permission to open /dev/mem:
 *   ./avmm_frame_xy_dump <nbody_offset_hex> <input_frame.txt> <output_xy.txt> [n_bodies] [gap]
 *
 * Example:
 *   ./avmm_frame_xy_dump 0x00000 frame0_1024binit200_27bits.txt hw_xy_1024.txt 1024 1
 */

#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define LW_BRIDGE_BASE 0xFF200000u
#define LW_BRIDGE_SPAN 0x00200000u

#define NBODY_MAX_BODIES 1024u
#define NBODY_DATA_MASK  0x07FFFFFFu

enum {
    NB_GO       = 0x00,
    NB_N_BODIES = 0x01,
    NB_GAP      = 0x02,
    NB_X_IN     = 0x03,
    NB_Y_IN     = 0x04,
    NB_M_IN     = 0x05,
    NB_VX_IN    = 0x06,
    NB_VY_IN    = 0x07,
    NB_DONE     = 0x08,
    NB_READ     = 0x09,
    NB_OUT_X    = 0x0A,
    NB_OUT_Y    = 0x0B,
};

typedef struct {
    uint32_t x;
    uint32_t y;
    uint32_t vx;
    uint32_t vy;
    uint32_t m;
    bool present;
} frame_body_t;

static inline void mmio_write(volatile uint32_t *base, uint32_t word_index, uint32_t value)
{
    base[word_index] = value;
    __sync_synchronize();
}

static inline uint32_t mmio_read(volatile uint32_t *base, uint32_t word_index)
{
    uint32_t value = base[word_index];
    __sync_synchronize();
    return value;
}

static uint32_t parse_u32_arg(const char *s, const char *name)
{
    char *end = NULL;
    errno = 0;
    unsigned long value = strtoul(s, &end, 0);
    if (errno != 0 || end == s || *end != '\0' || value > 0xFFFFFFFFul) {
```

```
        fprintf(stderr, "Bad %s: %s\n", name, s);
        exit(2);
    }
    return (uint32_t)value;
}

static uint32_t parse_count_arg(const char *s, const char *name)
{
    uint32_t value = parse_u32_arg(s, name);
    if (value == 0u || value > NBODY_MAX_BODIES) {
        fprintf(stderr, "%s must be in 1..%u, got %" PRIu32 "\n",
                name, NBODY_MAX_BODIES, value);
        exit(2);
    }
    return value;
}

static uint32_t parse_gap_arg(const char *s)
{
    uint32_t value = parse_u32_arg(s, "gap");
    if (value == 0u) {
        fprintf(stderr, "gap must be nonzero\n");
        exit(2);
    }
    return value;
}

static uint32_t parse_offset_arg(const char *s)
{
    uint32_t value = parse_u32_arg(s, "nbody offset");
    if ((value & 0x3u) != 0u) {
        fprintf(stderr, "Warning: nbody offset is not 4-byte aligned: 0x%08" PRIx32 "\n",
                value);
    }
    if (value >= LW_BRIDGE_SPAN) {
        fprintf(stderr, "nbody offset must be inside LW bridge span 0x%08x\n",
                LW_BRIDGE_SPAN);
        exit(2);
    }
    return value;
}

static uint32_t infer_count(const frame_body_t bodies[NBODY_MAX_BODIES])
{
    uint32_t count = 0;

    for (uint32_t i = 0; i < NBODY_MAX_BODIES; ++i) {
        if (bodies[i].present) {
            count = i + 1u;
        }
    }

    return count;
}

static int read_frame_file(const char *path, frame_body_t bodies[NBODY_MAX_BODIES],
                           uint32_t *inferred_count)
{
    FILE *fp;
    char line[512];
    unsigned line_no = 0;

    memset(bodies, 0, sizeof(frame_body_t) * NBODY_MAX_BODIES);

    fp = fopen(path, "r");
    if (!fp) {
        perror(path);
        return 1;
    }

    while (fgets(line, sizeof(line), fp)) {
        int idx;
        unsigned px;
```

```
        unsigned py;
        unsigned vx;
        unsigned vy;
        unsigned m;
        char tail;
        int got;
        char *p = line;

        line_no++;
        while (*p == ' ' || *p == '\t') {
            p++;
        }
        if (*p == '\0' || *p == '\n' || *p == '#') {
            continue;
        }

        got = sscanf(p, "%d %x %x %x %x %x %c", &idx, &px, &py, &vx, &vy, &m, &tail);
        if (got != 6) {
            fprintf(stderr, "%s:%u: expected six columns: i px py vx vy m\n",
                    path, line_no);
            fclose(fp);
            return 1;
        }
        if (idx < 0 || idx >= (int)NBODY_MAX_BODIES) {
            fprintf(stderr, "%s:%u: body index %d outside 0..%u\n",
                    path, line_no, idx, NBODY_MAX_BODIES - 1u);
            fclose(fp);
            return 1;
        }
        if ((px | py | vx | vy | m) & ~NBODY_DATA_MASK) {
            fprintf(stderr, "%s:%u: S1E8M18 fields must fit in 27 bits\n",
                    path, line_no);
            fclose(fp);
            return 1;
        }
        if (bodies[idx].present) {
            fprintf(stderr, "%s:%u: duplicate body index %d\n", path, line_no, idx);
            fclose(fp);
            return 1;
        }

        bodies[idx].x = px;
        bodies[idx].y = py;
        bodies[idx].vx = vx;
        bodies[idx].vy = vy;
        bodies[idx].m = m;
        bodies[idx].present = true;
    }

    if (ferror(fp)) {
        perror(path);
        fclose(fp);
        return 1;
    }

    fclose(fp);
    *inferred_count = infer_count(bodies);
    if (*inferred_count == 0u) {
        fprintf(stderr, "%s: no body rows found\n", path);
        return 1;
    }

    return 0;
}

static int validate_frame_prefix(const frame_body_t bodies[NBODY_MAX_BODIES],
                                 uint32_t count)
{
    for (uint32_t i = 0; i < count; ++i) {
        if (!bodies[i].present) {
            fprintf(stderr, "input frame is missing required body index %" PRIu32 "\n", i);
            return 1;
        }
```

```
    }
    return 0;
}

static void load_frame(volatile uint32_t *nb, const frame_body_t bodies[NBODY_MAX_BODIES],
                       uint32_t count, uint32_t gap)
{
    mmio_write(nb, NB_READ, 0u);
    mmio_write(nb, NB_N_BODIES, count);
    mmio_write(nb, NB_GAP, gap);

    for (uint32_t i = 0; i < count; ++i) {
        mmio_write(nb, NB_X_IN, bodies[i].x);
        mmio_write(nb, NB_Y_IN, bodies[i].y);
        mmio_write(nb, NB_M_IN, bodies[i].m);
        mmio_write(nb, NB_VX_IN, bodies[i].vx);
        mmio_write(nb, NB_VY_IN, bodies[i].vy);
    }
}

static bool poll_done(volatile uint32_t *nb, uint32_t max_iters)
{
    for (uint32_t i = 0; i < max_iters; ++i) {
        if (mmio_read(nb, NB_DONE) & 0x1u) {
            return true;
        }
        if ((i & 0x3fffu) == 0u) {
            usleep(1000);
        }
    }
    return false;
}

static int run_and_write_xy(volatile uint32_t *nb, const char *out_path,
                            const char *input_path, uint32_t count, uint32_t gap)
{
    FILE *out;

    mmio_write(nb, NB_GO, 1u);
    if (!poll_done(nb, 20000000u)) {
        fprintf(stderr, "[FAIL] DONE did not go high for count=%" PRIu32 ", gap=%" PRIu32 "\
    n",
                count, gap);
        return 1;
    }

    out = fopen(out_path, "w");
    if (!out) {
        perror(out_path);
        return 1;
    }

    fprintf(out, "# source %s\n", input_path);
    fprintf(out, "# n_bodies %" PRIu32 "\n", count);
    fprintf(out, "# gap %" PRIu32 "\n", gap);
    fprintf(out, "# i x y (S1E8M18 hex)\n");

    mmio_write(nb, NB_READ, 1u);
    for (uint32_t i = 0; i < count; ++i) {
        uint32_t x = mmio_read(nb, NB_OUT_X) & NBODY_DATA_MASK;
        uint32_t y = mmio_read(nb, NB_OUT_Y) & NBODY_DATA_MASK;
        fprintf(out, "%4" PRIu32 "  %07" PRIX32 "  %07" PRIX32 "\n", i, x, y);
    }
    mmio_write(nb, NB_READ, 0u);

    if (fclose(out) != 0) {
        perror(out_path);
        return 1;
    }

    return 0;
}

```

```c
int main(int argc, char **argv)
{
    frame_body_t bodies[NBODY_MAX_BODIES];
    uint32_t nbody_offset;
    uint32_t inferred_count;
    uint32_t count;
    uint32_t gap;
    int fd;
    void *map;
    volatile uint32_t *nbody;
    int ret;

    if (argc < 4 || argc > 6) {
        fprintf(stderr,
                "Usage: %s <nbody_offset_hex> <input_frame.txt> <output_xy.txt> [n_bodies] [gap]\n"
                "Example: %s 0x00000 frame0_1024binit200_27bits.txt hw_xy_1024.txt 1024 1\n",
                argv[0], argv[0]);
        return 2;
    }

    nbody_offset = parse_offset_arg(argv[1]);
    gap = (argc >= 6) ? parse_gap_arg(argv[5]) : 1u;

    if (read_frame_file(argv[2], bodies, &inferred_count) != 0) {
        return 1;
    }

    count = (argc >= 5) ? parse_count_arg(argv[4], "n_bodies") : inferred_count;
    if (count > inferred_count) {
        fprintf(stderr,
                "requested n_bodies=%" PRIu32 " but frame only contains rows through index %" PRIu32 "\n",
                count, inferred_count - 1u);
        return 2;
    }
    if (validate_frame_prefix(bodies, count) != 0) {
        return 1;
    }

    fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) {
        perror("open /dev/mem");
        return 1;
    }

    map = mmap(NULL, LW_BRIDGE_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED, fd, LW_BRIDGE_BASE);
    if (map == MAP_FAILED) {
        perror("mmap lightweight bridge");
        close(fd);
        return 1;
    }

    nbody = (volatile uint32_t *)((volatile uint8_t *)map + nbody_offset);

    printf("LW bridge base: 0x%08x\n", LW_BRIDGE_BASE);
    printf("nbody offset  : 0x%08" PRIx32 "\n", nbody_offset);
    printf("input frame   : %s\n", argv[2]);
    printf("output xy     : %s\n", argv[3]);
    printf("n_bodies      : %" PRIu32 "\n", count);
    printf("gap           : %" PRIu32 "\n", gap);
    printf("[nbody] loading frame...\n");

    load_frame(nbody, bodies, count, gap);
    printf("[nbody] running accelerator and dumping X/Y results...\n");
    ret = run_and_write_xy(nbody, argv[3], argv[2], count, gap);

    munmap(map, LW_BRIDGE_SPAN);
    close(fd);

    if (ret == 0) {
```

```
371         printf("[pass] wrote %s\n", argv[3]);
372     }
373     return ret;
374 }
```

A.2.3 Test Makefile

Listing 19: code/Software/test/Makefile

```
1  # Makefile for direct userspace Avalon-MM smoke test on DE1-SoC
2  #
3  # This is intentionally simpler than the Lab 3 driver Makefile:
4  # it only builds a userspace program that opens /dev/mem and writes/reads
5  # the FPGA lightweight bridge directly. No .ko driver is built here.
6  #
7  # Usage on DE1-SoC:
8  #   make
9  #   make run NBODY_OFFSET=0x00000 VGA_OFFSET=0x01000
10 #   make run-frame NBODY_OFFSET=0x00000 FRAME_INPUT=frame0_1024binit200_27bits.txt
       FRAME_OUTPUT=hw_xy_1024.txt
11 #
12 # Replace the offsets with Platform Designer assigned offsets
13 # relative to the HPS lightweight bridge base 0xFF200000.
14
15 CC      ?= cc
16 CFLAGS  ?= -O2 -Wall -Wextra -std=c11
17 SMOKE_TARGET := avmm_smoke_test
18 FRAME_TARGET := avmm_frame_xy_dump
19 TARGETS      := $(SMOKE_TARGET) $(FRAME_TARGET)
20
21 # Override these at the command line:
22 #   make run NBODY_OFFSET=0x00000 VGA_OFFSET=0x01000
23 NBODY_OFFSET ?= 0x00000
24 VGA_OFFSET   ?= 0x01000
25 FRAME_INPUT  ?= ../../Hardware/tb/frame_input/frame0_1024binit200_27bits.txt
26 FRAME_OUTPUT ?= hw_xy_1024.txt
27 FRAME_N      ?= 1024
28 FRAME_GAP    ?= 1
29
30 .PHONY: all run clean help
31
32 all: $(TARGETS)
33
34 $(SMOKE_TARGET): avmm_smoke_test.c
35     $(CC) $(CFLAGS) -o $@ $^
36
37 $(FRAME_TARGET): avmm_frame_xy_dump.c
38     $(CC) $(CFLAGS) -o $@ $^
39
40 run: $(SMOKE_TARGET)
41     ./$(SMOKE_TARGET) $(NBODY_OFFSET) $(VGA_OFFSET)
42
43 run-frame: $(FRAME_TARGET)
44     ./$(FRAME_TARGET) $(NBODY_OFFSET) $(FRAME_INPUT) $(FRAME_OUTPUT) $(FRAME_N) $(FRAME_GAP)
45
46 clean:
47     $(RM) $(TARGETS) *.o
48
49 help:
50     @echo "Build: make"
51     @echo "Run:   make run NBODY_OFFSET=0x<nbody_offset> VGA_OFFSET=0x<vga_offset>"
52     @echo "Example: make run NBODY_OFFSET=0x00000 VGA_OFFSET=0x01000"
53     @echo "Frame: make run-frame NBODY_OFFSET=0x<nbody_offset> FRAME_INPUT=<frame.txt>
       FRAME_OUTPUT=<xy.txt> FRAME_N=<n> FRAME_GAP=<gap>"
54     @echo "Example: make run-frame NBODY_OFFSET=0x00000 FRAME_INPUT=../../Hardware/tb/
       frame_input/frame0_1024binit200_27bits.txt FRAME_OUTPUT=hw_xy_1024.txt FRAME_N=1024
       FRAME_GAP=1"
```

A.3 Hardware Source Code

A.3.1 nbody_accel_avmm.sv

Listing 20: code/Hardware/code/nbody_accel_avmm.sv

```
// Avalon-MM top-level shell for the N-body accelerator.
//
// Register map, 32-bit words:
//   0x00 GO        W  Write bit 0 high to start one accelerator run. This also
//                  resets input/output body pointers to 0.
//   0x01 N_BODIES  W  Number of active bodies in the simulation.
//   0x02 GAP       W  Number of timesteps executed internally between DONE pulses.
//   0x03 X_IN      W  Input X position for the current body.
//   0x04 Y_IN      W  Input Y position for the current body.
//   0x05 M_IN      W  Input mass for the current body.
//   0x06 VX_IN     W  Input X velocity for the current body.
//   0x07 VY_IN     W  Input Y velocity for the current body. Writing this register
//                  commits the current body and increments the input pointer.
//   0x08 DONE      R  High when GAP timesteps have completed.
//   0x09 READ      W  Write 1 while software is reading DONE/output data. Write
//                  0 after reading all outputs; this clears DONE and arms the
//                  next GO.
//   0x0A OUT_X     R  Output X position for the current output body.
//   0x0B OUT_Y     R  Output Y position for the current output body. Reading this
//                  register increments the output pointer.
// Data uses the lower 27 bits of each 32-bit Avalon word.

module nbody_accel_avmm #(
    parameter int MAX_BODIES = 1024
) (
    input  logic        clk,
    input  logic        reset,       // active-high Platform Designer reset

    input  logic        chipselect,
    input  logic        read,
    input  logic        write,
    input  logic [7:0]  address,
    input  logic [31:0] writedata,
    output logic [31:0] readdata
);

    localparam logic [7:0] REG_GO       = 8'h00;
    localparam logic [7:0] REG_N_BODIES = 8'h01;
    localparam logic [7:0] REG_GAP      = 8'h02;
    localparam logic [7:0] REG_X_IN     = 8'h03;
    localparam logic [7:0] REG_Y_IN     = 8'h04;
    localparam logic [7:0] REG_M_IN     = 8'h05;
    localparam logic [7:0] REG_VX_IN    = 8'h06;
    localparam logic [7:0] REG_VY_IN    = 8'h07;
    localparam logic [7:0] REG_DONE     = 8'h08;
    localparam logic [7:0] REG_READ     = 8'h09;
    localparam logic [7:0] REG_OUT_X    = 8'h0A;
    localparam logic [7:0] REG_OUT_Y    = 8'h0B;

    localparam int DATA_W = 27;
    localparam int PAD_W = 32 - DATA_W;
    localparam int PTR_W = $clog2(MAX_BODIES);
    localparam logic [PTR_W-1:0] MAX_BODY_PTR = PTR_W'(MAX_BODIES - 1);

    logic        go_pulse;
    logic        read_reg;
    logic        out_y_read_d;
    logic        done;
    logic        first_step_pending;
    logic        first_step_for_run;

    logic [31:0] n_bodies_reg;
    logic [31:0] gap_reg;

    logic [PTR_W-1:0] input_ptr;
```

```
66      logic [PTR_W-1:0] output_ptr;
67
68      logic [DATA_W-1:0] x_in_shadow;
69      logic [DATA_W-1:0] y_in_shadow;
70      logic [DATA_W-1:0] m_in_shadow;
71      logic [DATA_W-1:0] vx_in_shadow;
72      logic [DATA_W-1:0] vy_in_shadow;
73
74      logic              cpu_body_we;
75      logic [PTR_W-1:0] cpu_body_waddr;
76      logic [PTR_W-1:0] control_body_raddr;
77      logic [PTR_W-1:0] mem_body_raddr;
78      logic [DATA_W-1:0] body_x;
79      logic [DATA_W-1:0] body_y;
80      logic [DATA_W-1:0] body_m;
81      logic [DATA_W-1:0] body_vx;
82      logic [DATA_W-1:0] body_vy;
83      logic [DATA_W-1:0] body_ax;
84      logic [DATA_W-1:0] body_ay;
85
86      logic              body_update_we;
87      logic [PTR_W-1:0] body_update_addr;
88      logic [DATA_W-1:0] body_update_x;
89      logic [DATA_W-1:0] body_update_y;
90      logic [DATA_W-1:0] body_update_vx;
91      logic [DATA_W-1:0] body_update_vy;
92
93      logic              accel_we;
94      logic [PTR_W-1:0]  accel_waddr;
95      logic [DATA_W-1:0] accel_ax;
96      logic [DATA_W-1:0] accel_ay;
97
98      assign mem_body_raddr = done ? output_ptr : control_body_raddr;
99
100     nbody_mem #(
101         .MAX_BODIES(MAX_BODIES),
102         .DATA_W(DATA_W),
103         .PTR_W(PTR_W)
104     ) u_mem (
105         .clk             (clk),
106
107         .cpu_body_we     (cpu_body_we),
108         .cpu_body_waddr  (cpu_body_waddr),
109         .cpu_x           (x_in_shadow),
110         .cpu_y           (y_in_shadow),
111         .cpu_m           (m_in_shadow),
112         .cpu_vx          (vx_in_shadow),
113         .cpu_vy          (vy_in_shadow),
114
115         .body_raddr      (mem_body_raddr),
116         .body_x          (body_x),
117         .body_y          (body_y),
118         .body_m          (body_m),
119         .body_vx         (body_vx),
120         .body_vy         (body_vy),
121         .body_ax         (body_ax),
122         .body_ay         (body_ay),
123
124         .body_update_we  (body_update_we),
125         .body_update_addr(body_update_addr),
126         .body_update_x   (body_update_x),
127         .body_update_y   (body_update_y),
128         .body_update_vx  (body_update_vx),
129         .body_update_vy  (body_update_vy),
130
131         .accel_we        (accel_we),
132         .accel_waddr     (accel_waddr),
133         .accel_ax        (accel_ax),
134         .accel_ay        (accel_ay)
135     );
136
137     nbody_control #(
138         .MAX_BODIES(MAX_BODIES),
```

```
        .DATA_W(DATA_W)
    ) u_control (
        .clk             (clk),
        .reset           (reset),

        .go                          (go_pulse),
        .read_enable                 (read_reg),
        .first_step                  (first_step_for_run),
        .n_bodies                    (n_bodies_reg),
        .gap                         (gap_reg),
        .done                        (done),

        .body_raddr      (control_body_raddr),
        .body_x          (body_x),
        .body_y          (body_y),
        .body_m          (body_m),
        .body_vx         (body_vx),
        .body_vy         (body_vy),
        .body_ax         (body_ax),
        .body_ay         (body_ay),

        .body_update_we  (body_update_we),
        .body_update_addr(body_update_addr),
        .body_update_x   (body_update_x),
        .body_update_y   (body_update_y),
        .body_update_vx  (body_update_vx),
        .body_update_vy  (body_update_vy),

        .accel_we        (accel_we),
        .accel_waddr     (accel_waddr),
        .accel_ax        (accel_ax),
        .accel_ay        (accel_ay)
    );

    always_ff @(posedge clk) begin
        if (reset) begin
            go_pulse      <= 1'b0;
            read_reg      <= 1'b0;
            out_y_read_d   <= 1'b0;
            first_step_pending <= 1'b1;
            first_step_for_run <= 1'b1;
            n_bodies_reg  <= 32'd0;
            gap_reg       <= 32'd0;
            input_ptr     <= '0;
            output_ptr    <= '0;
            x_in_shadow   <= '0;
            y_in_shadow   <= '0;
            m_in_shadow   <= '0;
            vx_in_shadow  <= '0;
            vy_in_shadow  <= '0;
            cpu_body_we    <= 1'b0;
            cpu_body_waddr <= '0;
        end else begin
            go_pulse    <= 1'b0;
            cpu_body_we <= 1'b0;
            out_y_read_d <= chipselect && read && (address == REG_OUT_Y);

            if (chipselect && write) begin
                unique case (address)
                    REG_GO: begin
                        if (writedata[0]) begin
                            go_pulse   <= 1'b1;
                            read_reg   <= 1'b1;
                            first_step_for_run <= first_step_pending;
                            first_step_pending <= 1'b0;
                            input_ptr  <= '0;
                            output_ptr <= '0;
                        end
                    end

                    REG_N_BODIES: n_bodies_reg <= writedata;
                    REG_GAP:      gap_reg      <= writedata;
                    REG_X_IN:     x_in_shadow  <= writedata[DATA_W-1:0];
```

```
                    REG_Y_IN:     y_in_shadow  <= writedata[DATA_W-1:0];
                    REG_M_IN:     m_in_shadow  <= writedata[DATA_W-1:0];
                    REG_VX_IN:    vx_in_shadow <= writedata[DATA_W-1:0];

                    REG_VY_IN: begin
                        vy_in_shadow  <= writedata[DATA_W-1:0];
                        cpu_body_we    <= 1'b1;
                        cpu_body_waddr <= input_ptr;
                        first_step_pending <= 1'b1;

                        if (input_ptr != MAX_BODY_PTR) begin
                            input_ptr <= input_ptr + 1'b1;
                        end
                    end

                    REG_READ: begin
                        read_reg <= writedata[0];
                        if (!writedata[0]) begin
                            output_ptr <= '0;
                        end
                    end

                    default: begin
                    end
                endcase
            end

            if ((chipselect && read && (address == REG_OUT_Y)) && !out_y_read_d) begin
                if (output_ptr != MAX_BODY_PTR) begin
                    output_ptr <= output_ptr + 1'b1;
                end
            end
        end
    end

    always_comb begin
        unique case (address)
            REG_DONE:  readdata = {31'd0, done};
            REG_OUT_X: readdata = {{PAD_W{1'b0}}, body_x};
            REG_OUT_Y: readdata = {{PAD_W{1'b0}}, body_y};
            default:   readdata = 32'd0;
        endcase
    end

endmodule
```

A.3.2 vga_bitmap_avmm.sv

Listing 21: code/Hardware/code/vga_bitmap_avmm.sv

```
/*
 * Avalon memory-mapped VGA bitmap peripheral.
 *
 * This is a separate Platform Designer/Qsys component from the n-body
 * accelerator. Software writes a packed 640x480 1-bpp framebuffer through
 * Avalon-MM; hardware only scans the bitmap out to VGA.
 *
 * Register map, 32-bit word addresses:
 *   0..9599  framebuffer words, write-only from software
 *   all reads from software return 0
 *
 * Bitmap packing:
 *   WIDTH = 640, HEIGHT = 480, WORDS_PER_ROW = 20, FB_WORDS = 9600
 *   word_index = y * 20 + x / 32
 *   bit_index  = x % 32
 *   frame[y * 20 + x / 32][x % 32] = pixel(x, y)
 *
 * Pixel value 0 displays black. Pixel value 1 displays white.
 */
```

```
20
21 module vga_bitmap_avmm (
22     input  logic         clk,
23     input  logic         reset,       // active-high Platform Designer reset
24
25     input  logic         chipselect,
26     input  logic         read,
27     input  logic         write,
28     input  logic [13:0]  address,     // word address, valid framebuffer range 0..9599
29     input  logic [31:0]  writedata,
30     output logic [31:0]  readdata,
31
32     output logic [7:0]   VGA_R,
33     output logic [7:0]   VGA_G,
34     output logic [7:0]   VGA_B,
35     output logic         VGA_CLK,
36     output logic         VGA_HS,
37     output logic         VGA_VS,
38     output logic         VGA_BLANK_n,
39     output logic         VGA_SYNC_n
40 );
41
42     localparam int WIDTH         = 640;
43     localparam int HEIGHT        = 480;
44     localparam int WORDS_PER_ROW = 20;
45     localparam int FB_WORDS      = HEIGHT * WORDS_PER_ROW;
46     localparam int ADDR_W        = 14;
47
48     localparam logic [ADDR_W-1:0] FB_LAST_WORD = ADDR_W'(FB_WORDS - 1);
49
50     logic [10:0] hcount;
51     logic [9:0]  vcount;
52
53     logic        av_addr_valid;
54     logic [13:0] av_ram_addr;
55
56     logic        vga_blank_raw;
57     logic        active_video;
58     logic        active_video_d;
59     logic [9:0]  pixel_x;
60     logic [8:0]  pixel_y;
61     logic [13:0] pixel_y_ext;
62     logic [13:0] scan_addr;
63     logic [4:0]  bit_index;
64     logic [4:0]  bit_index_d;
65     logic [31:0] scan_readdata;
66     logic        pixel_bit;
67
68     assign av_addr_valid = (address <= FB_LAST_WORD);
69     assign av_ram_addr   = av_addr_valid ? address : 14'd0;
70
71     // Reuse the Lab 3 VGA timing generator from vga_ball.sv.
72     // hcount[10:1] is the 640-pixel column and vcount is the 480-pixel row.
73     vga_counters counters (
74         .clk50      (clk),
75         .reset      (reset),
76         .hcount     (hcount),
77         .vcount     (vcount),
78         .VGA_CLK    (VGA_CLK),
79         .VGA_HS     (VGA_HS),
80         .VGA_VS     (VGA_VS),
81         .VGA_BLANK_n(vga_blank_raw),
82         .VGA_SYNC_n (VGA_SYNC_n)
83     );
84
85     assign active_video = vga_blank_raw;
86     assign pixel_x      = hcount[10:1];
87     assign pixel_y      = vcount[8:0];
88     assign pixel_y_ext  = {5'd0, pixel_y};
89
90     // word_index = y * 20 + x / 32 = (y << 4) + (y << 2) + x[9:5]
91     assign scan_addr = active_video ? ((pixel_y_ext << 4) + (pixel_y_ext << 2) +
92                                        {9'd0, pixel_x[9:5]}) :
```

```
                                        14'd0;
    assign bit_index = pixel_x[4:0];

    framebuffer_ram #(
        .FB_WORDS(FB_WORDS),
        .ADDR_W  (ADDR_W)
    ) u_framebuffer (
        .clk             (clk),

        .port_a_we       (chipselect && write && av_addr_valid),
        .port_a_addr     (av_ram_addr),
        .port_a_writedata(writedata),

        .port_b_addr     (scan_addr),
        .port_b_readdata (scan_readdata)
    );

    always_ff @(posedge clk) begin
        if (reset) begin
            active_video_d  <= 1'b0;
            bit_index_d     <= 5'd0;
        end else begin
            active_video_d  <= active_video;
            bit_index_d     <= bit_index;
        end
    end

    assign readdata    = 32'd0;
    assign VGA_BLANK_n = vga_blank_raw;
    assign pixel_bit   = scan_readdata[bit_index_d];

    always_comb begin
        if (active_video_d && pixel_bit) 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,
```

```
166               VFRONT_PORCH = 10'd 10,
167               VSYNC        = 10'd 2,
168               VBACK_PORCH  = 10'd 33,
169               VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
170                              VBACK_PORCH; // 525
171
172     logic endOfLine;
173
174     always_ff @(posedge clk50)
175       if (reset)          hcount <= 0;
176       else if (endOfLine) hcount <= 0;
177       else                hcount <= hcount + 11'd 1;
178
179     assign endOfLine = hcount == HTOTAL - 1;
180
181     logic endOfField;
182
183     always_ff @(posedge clk50)
184       if (reset)          vcount <= 0;
185       else if (endOfLine)
186         if (endOfField)   vcount <= 0;
187         else              vcount <= vcount + 10'd 1;
188
189     assign endOfField = vcount == VTOTAL - 1;
190
191     // Horizontal sync: from 0x520 to 0x5DF (0x57F)
192     // 101 0010 0000 to 101 1101 1111
193     assign VGA_HS = !( (hcount[10:8] == 3'b101) &
194                        !(hcount[7:5] == 3'b111));
195     assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
196
197     assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused
198
199     // Horizontal active: 0 to 1279     Vertical active: 0 to 479
200     // 101 0000 0000  1280              01 1110 0000  480
201     // 110 0011 1111  1599              10 0000 1100  524
202     assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
203                          !( vcount[9] | (vcount[8:5] == 4'b1111) );
204
205     /* VGA_CLK is 25 MHz
206      *             __    __    __
207      * clk50    __|  |__|  |__|
208      *
209      *             _____       __
210      * hcount[0]__|     |_____|
211      */
212     assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive
213
214 endmodule
```

A.3.3 nbody_control.sv

Listing 22: code/Hardware/code/nbody_control.sv

```
1  module nbody_control #(
2      parameter int MAX_BODIES = 1024,
3      parameter int DATA_W = 27
4  ) (
5      input  logic clk,
6      input  logic reset,
7
8      input  logic        go,
9      input  logic        read_enable,
10     input  logic        first_step,
11     input  logic [31:0] n_bodies,
12     input  logic [31:0] gap,
13     output logic        done,
14
15     output logic [$clog2(MAX_BODIES)-1:0] body_raddr,
```

```
16      input  logic [DATA_W-1:0]              body_x,
17      input  logic [DATA_W-1:0]              body_y,
18      input  logic [DATA_W-1:0]              body_m,
19      input  logic [DATA_W-1:0]              body_vx,
20      input  logic [DATA_W-1:0]              body_vy,
21      input  logic [DATA_W-1:0]              body_ax,
22      input  logic [DATA_W-1:0]              body_ay,
23
24      output logic                           body_update_we,
25      output logic [$clog2(MAX_BODIES)-1:0] body_update_addr,
26      output logic [DATA_W-1:0]              body_update_x,
27      output logic [DATA_W-1:0]              body_update_y,
28      output logic [DATA_W-1:0]              body_update_vx,
29      output logic [DATA_W-1:0]              body_update_vy,
30
31      output logic                           accel_we,
32      output logic [$clog2(MAX_BODIES)-1:0] accel_waddr,
33      output logic [DATA_W-1:0]              accel_ax,
34      output logic [DATA_W-1:0]              accel_ay
35  );
36
37      typedef enum logic [3:0] {
38          ST_IDLE,
39          ST_LOAD_TILE_PRIME,
40          ST_LOAD_TILE,
41          ST_CLEAR_GROUP,
42          ST_COMPUTE_PRIME,
43          ST_COMPUTE_GROUP,
44          ST_DRAIN_GROUP,
45          ST_STORE_GROUP,
46          ST_NEXT_GROUP,
47          ST_NEXT_TILE,
48          ST_INTEGRATE_PRIME,
49          ST_INTEGRATE_START,
50          ST_INTEGRATE_WAIT,
51          ST_UPDATE_COMMIT,
52          ST_DONE
53      } state_t;
54
55      typedef enum logic [1:0] {
56          UPD_NEXT_BODY,
57          UPD_NEXT_STEP,
58          UPD_DONE
59      } update_next_t;
60
61      localparam int PTR_W = $clog2(MAX_BODIES);
62      localparam logic [PTR_W-1:0] MAX_BODY_PTR = PTR_W'(MAX_BODIES - 1);
63      localparam logic [PTR_W-1:0] TILE_STRIDE  = PTR_W'(16);
64      localparam int ACTIVE_W = PTR_W + 1;
65      localparam logic [31:0] MAX_BODIES_U32 = 32'(MAX_BODIES);
66      localparam int PIPE_LAT = 18;
67      localparam int WAIT_W = $clog2(PIPE_LAT + 4);
68
69      state_t state;
70      update_next_t update_next;
71
72      logic [31:0] timestep_count;
73      logic run_initial_half_step;
74      logic [PTR_W-1:0] tile_base;
75      logic [PTR_W-1:0] j_body_idx;
76      logic [PTR_W-1:0] current_j_idx;
77      logic [PTR_W-1:0] integrate_idx;
78      logic [WAIT_W-1:0] wait_count;
79      logic [1:0] compute_grp;
80      logic [1:0] store_lane;
81      logic [ACTIVE_W-1:0] active_count;
82      logic [ACTIVE_W-1:0] active_count_next;
83      logic core_rst_n;
84
85      logic        core_clear_prev;
86      logic        core_load_en;
87      logic        core_compute_en;
88      logic [3:0]  load_idx_count;
```

```
    logic [3:0]  core_load_idx;
    logic [DATA_W-1:0] core_load_x;
    logic [DATA_W-1:0] core_load_y;
    logic [1:0]  core_grp_sel;
    logic [DATA_W-1:0] core_j_x;
    logic [DATA_W-1:0] core_j_y;
    logic [DATA_W-1:0] core_j_m;
    logic [3:0]  core_lane_mask;

    logic [DATA_W-1:0] current_j_x;
    logic [DATA_W-1:0] current_j_y;
    logic [DATA_W-1:0] current_j_m;

    logic [DATA_W-1:0] core_res_x [4];
    logic [DATA_W-1:0] core_res_y [4];
    logic        unused_core_res_vld;

    logic        integrator_start;
    logic        integrator_done;
    logic        integrator_half_step;
    logic [7:0]  integrator_ax_half_exp;
    logic [7:0]  integrator_ay_half_exp;
    logic [DATA_W-1:0] integrator_ax_in;
    logic [DATA_W-1:0] integrator_ay_in;
    logic [DATA_W-1:0] integrator_x_out;
    logic [DATA_W-1:0] integrator_y_out;
    logic [DATA_W-1:0] integrator_vx_out;
    logic [DATA_W-1:0] integrator_vy_out;

    function automatic logic [ACTIVE_W-1:0] ptr_to_active(input logic [PTR_W-1:0] value);
        ptr_to_active = {{(ACTIVE_W-PTR_W){1'b0}}, value};
    endfunction

    function automatic logic [ACTIVE_W-1:0] load_idx_to_active(
        input logic [3:0] value
    );
        load_idx_to_active = ACTIVE_W'(value);
    endfunction

    function automatic logic [ACTIVE_W-1:0] lane_idx_to_active(
        input logic [PTR_W-1:0] base,
        input logic [1:0]       grp,
        input logic [1:0]       lane
    );
        lane_idx_to_active = ptr_to_active(base) + ACTIVE_W'({grp, 2'b00}) + ACTIVE_W'(lane)
    ;
    endfunction

    function automatic logic is_last_active_body(
        input logic [PTR_W-1:0] idx,
        input logic [ACTIVE_W-1:0] active_bodies
    );
        is_last_active_body = (idx == MAX_BODY_PTR) ||
                              (ptr_to_active(idx) + ACTIVE_W'(1) >= active_bodies);
    endfunction

    function automatic logic [PTR_W-1:0] lane_body_idx(
        input logic [PTR_W-1:0] base,
        input logic [1:0]       grp,
        input logic [1:0]       lane
    );
        lane_body_idx = base + PTR_W'({grp, 2'b00}) + PTR_W'(lane);
    endfunction

    function automatic logic lane_is_active(
        input logic [PTR_W-1:0] base,
        input logic [1:0]       grp,
        input logic [1:0]       lane,
        input logic [ACTIVE_W-1:0] active_bodies
    );
        logic [ACTIVE_W-1:0] lane_idx;
        begin
            lane_idx = lane_idx_to_active(base, grp, lane);
```

```
            lane_is_active = (lane_idx < active_bodies);
        end
    endfunction

    function automatic logic [3:0] make_lane_mask(
        input logic [PTR_W-1:0] base,
        input logic [1:0]       grp,
        input logic [PTR_W-1:0] j_idx,
        input logic [ACTIVE_W-1:0] active_bodies
    );
        logic [ACTIVE_W-1:0] lane0;
        logic [ACTIVE_W-1:0] lane1;
        logic [ACTIVE_W-1:0] lane2;
        logic [ACTIVE_W-1:0] lane3;
        logic [ACTIVE_W-1:0] j_idx_active;
        begin
            lane0 = lane_idx_to_active(base, grp, 2'd0);
            lane1 = lane_idx_to_active(base, grp, 2'd1);
            lane2 = lane_idx_to_active(base, grp, 2'd2);
            lane3 = lane_idx_to_active(base, grp, 2'd3);
            j_idx_active = ptr_to_active(j_idx);

            make_lane_mask[0] = (lane0 >= active_bodies) || (lane0 == j_idx_active);
            make_lane_mask[1] = (lane1 >= active_bodies) || (lane1 == j_idx_active);
            make_lane_mask[2] = (lane2 >= active_bodies) || (lane2 == j_idx_active);
            make_lane_mask[3] = (lane3 >= active_bodies) || (lane3 == j_idx_active);
        end
    endfunction

    always_comb begin
        if (n_bodies > MAX_BODIES_U32) begin
            active_count_next = ACTIVE_W'(MAX_BODIES);
        end else begin
            active_count_next = n_bodies[ACTIVE_W-1:0];
        end
    end

    always_comb begin
        body_raddr = '0;

        unique case (state)
            ST_LOAD_TILE_PRIME: begin
                body_raddr = tile_base + PTR_W'(load_idx_count);
            end

            ST_LOAD_TILE: begin
                if (load_idx_count == 4'd15) begin
                    body_raddr = tile_base + PTR_W'(load_idx_count);
                end else begin
                    body_raddr = tile_base + PTR_W'(load_idx_count) + PTR_W'(1);
                end
            end

            ST_COMPUTE_PRIME: begin
                body_raddr = j_body_idx;
            end

            ST_COMPUTE_GROUP: begin
                if ((compute_grp == 2'd2) &&
                    !is_last_active_body(j_body_idx, active_count)) begin
                    body_raddr = j_body_idx + 1'b1;
                end else begin
                    body_raddr = j_body_idx;
                end
            end

            ST_INTEGRATE_PRIME,
            ST_INTEGRATE_START,
            ST_INTEGRATE_WAIT: begin
                body_raddr = integrate_idx;
            end

            default: begin
```

```
                body_raddr = '0;
            end
        endcase
    end

    // Leapfrog starts with one half-step velocity kick. Later kicks are full-step.
    assign integrator_half_step    = run_initial_half_step && (timestep_count == 32'd0);
    assign integrator_ax_half_exp  = (body_ax[25:18] > 8'd1) ? (body_ax[25:18] - 8'd1) : 8'
    d0;
    assign integrator_ay_half_exp  = (body_ay[25:18] > 8'd1) ? (body_ay[25:18] - 8'd1) : 8'
    d0;
    assign integrator_ax_in        = integrator_half_step ? {body_ax[26],
    integrator_ax_half_exp, body_ax[17:0]} : body_ax;
    assign integrator_ay_in        = integrator_half_step ? {body_ay[26],
    integrator_ay_half_exp, body_ay[17:0]} : body_ay;

    four_core_wrapper #(
        .DATA_W(DATA_W)
    ) u_core (
        .i_clk       (clk),
        .i_rst       (core_rst_n),
        .i_clear_prev(core_clear_prev),
        .i_load_en   (core_load_en),
        .i_compute_en(core_compute_en),
        .i_load_idx  (core_load_idx),
        .i_load_x    (core_load_x),
        .i_load_y    (core_load_y),
        .i_grp_sel   (core_grp_sel),
        .i_j_x       (core_j_x),
        .i_j_y       (core_j_y),
        .i_j_m       (core_j_m),
        .i_lane_mask (core_lane_mask),
        .o_res0_x    (core_res_x[0]),
        .o_res0_y    (core_res_y[0]),
        .o_res1_x    (core_res_x[1]),
        .o_res1_y    (core_res_y[1]),
        .o_res2_x    (core_res_x[2]),
        .o_res2_y    (core_res_y[2]),
        .o_res3_x    (core_res_x[3]),
        .o_res3_y    (core_res_y[3]),
        .o_res_vld   (unused_core_res_vld)
    );

    nbody_integrator #(
        .DATA_W(DATA_W)
    ) u_integrator (
        .clk        (clk),
        .reset      (reset),
        .i_start    (integrator_start),
        .o_done     (integrator_done),
        .i_x        (body_x),
        .i_y        (body_y),
        .i_vx       (body_vx),
        .i_vy       (body_vy),
        .i_ax       (integrator_ax_in),
        .i_ay       (integrator_ay_in),
        .o_x        (integrator_x_out),
        .o_y        (integrator_y_out),
        .o_vx       (integrator_vx_out),
        .o_vy       (integrator_vy_out)
    );

    // Main FSM
    always_ff @(posedge clk) begin
        if (reset) begin
            core_rst_n            <= 1'b0;
            state                 <= ST_IDLE;
            update_next           <= UPD_DONE;
            done                  <= 1'b0;
            timestep_count        <= 32'd0;
            run_initial_half_step <= 1'b0;
            tile_base             <= '0;
            j_body_idx            <= '0;
```

```
            current_j_idx         <= '0;
            integrate_idx         <= '0;
            wait_count            <= '0;
            compute_grp           <= 2'd0;
            store_lane            <= 2'd0;
            active_count          <= '0;
            core_clear_prev       <= 1'b0;
            core_load_en          <= 1'b0;
            core_compute_en       <= 1'b0;
            load_idx_count        <= 4'd0;
            core_load_idx         <= 4'd0;
            core_load_x           <= '0;
            core_load_y           <= '0;
            core_grp_sel          <= 2'd0;
            core_j_x              <= '0;
            core_j_y              <= '0;
            core_j_m              <= '0;
            core_lane_mask        <= 4'hF;
            current_j_x           <= '0;
            current_j_y           <= '0;
            current_j_m           <= '0;
            integrator_start      <= 1'b0;
            body_update_we        <= 1'b0;
            body_update_addr      <= '0;
            body_update_x         <= '0;
            body_update_y         <= '0;
            body_update_vx        <= '0;
            body_update_vy        <= '0;
            accel_we              <= 1'b0;
            accel_waddr           <= '0;
            accel_ax              <= '0;
            accel_ay              <= '0;
        end else begin
            core_clear_prev  <= 1'b0;
            core_load_en     <= 1'b0;
            core_compute_en  <= 1'b0;
            integrator_start <= 1'b0;
            body_update_we   <= 1'b0;
            accel_we         <= 1'b0;
            core_rst_n       <= 1'b1;

            unique case (state)
                ST_IDLE: begin
                    done <= 1'b0;
                    if (go) begin
                        tile_base             <= '0;
                        load_idx_count        <= 4'd0;
                        core_load_idx         <= 4'd0;
                        compute_grp           <= 2'd0;
                        j_body_idx            <= '0;
                        current_j_idx         <= '0;
                        integrate_idx         <= '0;
                        timestep_count        <= 32'd0;
                        run_initial_half_step <= first_step;
                        active_count          <= active_count_next;
                        if (active_count_next == '0) begin
                            state <= ST_DONE;
                        end else begin
                            state <= ST_LOAD_TILE_PRIME;
                        end
                    end
                end

                ST_LOAD_TILE_PRIME: begin
                    state <= ST_LOAD_TILE;
                end

                ST_LOAD_TILE: begin
                    core_load_en  <= 1'b1;
                    core_load_idx <= load_idx_count;

                    if (ptr_to_active(tile_base) + load_idx_to_active(load_idx_count) <
active_count) begin
```

```
                        core_load_x <= body_x;
                        core_load_y <= body_y;
                    end else begin
                        core_load_x <= '0;
                        core_load_y <= '0;
                    end

                    if (load_idx_count == 4'd15) begin
                        load_idx_count  <= 4'd0;
                        compute_grp     <= 2'd0;
                        j_body_idx      <= '0;
                        state           <= ST_CLEAR_GROUP;
                    end else begin
                        load_idx_count <= load_idx_count + 1'b1;
                    end
                end

                ST_CLEAR_GROUP: begin
                    core_clear_prev <= 1'b1;
                    core_grp_sel    <= 2'd0;
                    j_body_idx      <= '0;
                    current_j_idx   <= '0;
                    wait_count      <= '0;
                    state           <= ST_COMPUTE_PRIME;
                end

                ST_COMPUTE_PRIME: begin
                    current_j_x   <= body_x;
                    current_j_y   <= body_y;
                    current_j_m   <= body_m;
                    current_j_idx <= j_body_idx;
                    compute_grp   <= 2'd0;
                    state <= ST_COMPUTE_GROUP;
                end

                ST_COMPUTE_GROUP: begin
                    core_compute_en <= 1'b1;
                    core_grp_sel    <= compute_grp;
                    core_j_x        <= current_j_x;
                    core_j_y        <= current_j_y;
                    core_j_m        <= current_j_m;
                    core_lane_mask  <= make_lane_mask(tile_base, compute_grp, current_j_idx,
    active_count);

                    if (compute_grp == 2'd3) begin
                        if (is_last_active_body(j_body_idx, active_count)) begin
                            wait_count <= '0;
                            state      <= ST_DRAIN_GROUP;
                        end else begin
                            j_body_idx    <= j_body_idx + 1'b1;
                            current_j_idx <= j_body_idx + 1'b1;
                            current_j_x   <= body_x;
                            current_j_y   <= body_y;
                            current_j_m   <= body_m;
                            compute_grp   <= 2'd0;
                        end
                    end else begin
                        compute_grp <= compute_grp + 1'b1;
                    end
                end

                ST_DRAIN_GROUP: begin
                    core_grp_sel <= 2'd0;

                    if (wait_count == WAIT_W'(PIPE_LAT + 1)) begin
                        compute_grp <= 2'd0;
                        store_lane <= 2'd0;
                        state      <= ST_STORE_GROUP;
                    end else begin
                        wait_count <= wait_count + 1'b1;
                    end
                end

```

```
                ST_STORE_GROUP: begin
                    core_grp_sel <= compute_grp;

                    if (lane_is_active(tile_base, compute_grp, store_lane, active_count)) begin
                        accel_we    <= 1'b1;
                        accel_waddr <= lane_body_idx(tile_base, compute_grp, store_lane);
                        accel_ax    <= core_res_x[store_lane];
                        accel_ay    <= core_res_y[store_lane];
                    end

                    if (store_lane == 2'd3) begin
                        state <= ST_NEXT_GROUP;
                    end else begin
                        store_lane <= store_lane + 1'b1;
                    end
                end

                ST_NEXT_GROUP: begin
                    if (compute_grp == 2'd3) begin
                        state <= ST_NEXT_TILE;
                    end else begin
                        compute_grp <= compute_grp + 1'b1;
                        core_grp_sel <= compute_grp + 1'b1;
                        store_lane  <= 2'd0;
                        state       <= ST_STORE_GROUP;
                    end
                end

                ST_NEXT_TILE: begin
                    if (ptr_to_active(tile_base) + ACTIVE_W'(16) >= active_count) begin
                        integrate_idx <= '0;
                        state         <= ST_INTEGRATE_PRIME;
                    end else begin
                        tile_base     <= tile_base + TILE_STRIDE;
                        load_idx_count <= 4'd0;
                        core_load_idx <= 4'd0;
                        compute_grp   <= 2'd0;
                        j_body_idx    <= '0;
                        current_j_idx <= '0;
                        state         <= ST_LOAD_TILE_PRIME;
                    end
                end

                ST_INTEGRATE_PRIME: begin
                    state <= ST_INTEGRATE_START;
                end

                ST_INTEGRATE_START: begin
                    integrator_start <= 1'b1;
                    state            <= ST_INTEGRATE_WAIT;
                end

                ST_INTEGRATE_WAIT: begin
                    if (integrator_done) begin
                        body_update_we   <= 1'b1;
                        body_update_addr <= integrate_idx;
                        body_update_x    <= integrator_x_out;
                        body_update_y    <= integrator_y_out;
                        body_update_vx   <= integrator_vx_out;
                        body_update_vy   <= integrator_vy_out;

                        if (is_last_active_body(integrate_idx, active_count)) begin
                            if (timestep_count + 32'd1 >= gap) begin
                                update_next <= UPD_DONE;
                            end else begin
                                update_next    <= UPD_NEXT_STEP;
                                timestep_count <= timestep_count + 1'b1;
                            end
                        end else begin
                            update_next   <= UPD_NEXT_BODY;
                            integrate_idx <= integrate_idx + 1'b1;
                        end
```

```
519
520                         state <= ST_UPDATE_COMMIT;
521                     end
522                 end
523
524                 ST_UPDATE_COMMIT: begin
525                     unique case (update_next)
526                         UPD_NEXT_BODY: begin
527                             state <= ST_INTEGRATE_PRIME;
528                         end
529
530                         UPD_NEXT_STEP: begin
531                             tile_base     <= '0;
532                             load_idx_count <= 4'd0;
533                             core_load_idx <= 4'd0;
534                             compute_grp   <= 2'd0;
535                             j_body_idx    <= '0;
536                             current_j_idx <= '0;
537                             integrate_idx <= '0;
538                             state         <= ST_LOAD_TILE_PRIME;
539                         end
540
541                         default: begin
542                             state <= ST_DONE;
543                         end
544                     endcase
545                 end
546
547                 ST_DONE: begin
548                     done <= 1'b1;
549                     if (!read_enable) begin
550                         done  <= 1'b0;
551                         state <= ST_IDLE;
552                     end
553                 end
554
555                 default: begin
556                     state <= ST_IDLE;
557                 end
558             endcase
559         end
560     end
561
562 endmodule
```

A.3.4 four_core_wrapper.sv

Listing 23: code/Hardware/code/four_core_wrapper.sv

```
1  module four_core_wrapper #(
2      parameter int DATA_W = 27
3  ) (
4      input  logic         i_clk,
5      input  logic         i_rst,   // active-low synchronous reset
6
7      // external phase control
8      input  logic         i_clear_prev,
9      input  logic         i_load_en,
10     input  logic         i_compute_en,
11
12     // load 16 i-bodies into local wrapper memory
13     input  logic [3:0]   i_load_idx,
14     input  logic [DATA_W-1:0] i_load_x,
15     input  logic [DATA_W-1:0] i_load_y,
16
17     // external group select: combinationally selects which 4 i-bodies feed the datapath
18     // or which cached 4-lane result group is read out
19     input  logic [1:0]   i_grp_sel,
20
```

```
21      // broadcasted j input
22      input  logic [DATA_W-1:0] i_j_x,
23      input  logic [DATA_W-1:0] i_j_y,
24      input  logic [DATA_W-1:0] i_j_m,
25      input  logic [3:0]  i_lane_mask,
26
27      // accumulated outputs
28      output logic [DATA_W-1:0] o_res0_x,
29      output logic [DATA_W-1:0] o_res0_y,
30      output logic [DATA_W-1:0] o_res1_x,
31      output logic [DATA_W-1:0] o_res1_y,
32      output logic [DATA_W-1:0] o_res2_x,
33      output logic [DATA_W-1:0] o_res2_y,
34      output logic [DATA_W-1:0] o_res3_x,
35      output logic [DATA_W-1:0] o_res3_y,
36
37      // selected group results valid
38      output logic        o_res_vld
39  );
40
41      localparam int PIPE_LAT = 18;
42      integer i;
43
44      // local i-body memory (16 entries)
45      logic [DATA_W-1:0] i_x_mem [16];
46      logic [DATA_W-1:0] i_y_mem [16];
47
48      // internal cached accumulated outputs (real wrapper outputs)
49      logic [DATA_W-1:0] acc_x_bank [16];
50      logic [DATA_W-1:0] acc_y_bank [16];
51      logic [3:0]  grp_written;
52
53      // fixed feedback chain (no per-group selection here)
54      logic [DATA_W-1:0] prev0_x_d0, prev0_x_d1, prev0_y_d0, prev0_y_d1;
55      logic [DATA_W-1:0] prev1_x_d0, prev1_x_d1, prev1_y_d0, prev1_y_d1;
56      logic [DATA_W-1:0] prev2_x_d0, prev2_x_d1, prev2_y_d0, prev2_y_d1;
57      logic [DATA_W-1:0] prev3_x_d0, prev3_x_d1, prev3_y_d0, prev3_y_d1;
58
59      // c0->c17 pipelined delay for writeback address alignment
60      logic       vld_pipe [PIPE_LAT];
61      logic [1:0] grp_pipe [PIPE_LAT];
62
63      // current group selection for datapath/readout
64      logic [3:0] grp_base;
65      logic [3:0] wb_base;
66
67      assign grp_base = {i_grp_sel, 2'b00};
68      assign wb_base  = {grp_pipe[PIPE_LAT-1], 2'b00};
69
70      logic [DATA_W-1:0] cur_i0_x;
71      logic [DATA_W-1:0] cur_i0_y;
72      logic [DATA_W-1:0] cur_i1_x;
73      logic [DATA_W-1:0] cur_i1_y;
74      logic [DATA_W-1:0] cur_i2_x;
75      logic [DATA_W-1:0] cur_i2_y;
76      logic [DATA_W-1:0] cur_i3_x;
77      logic [DATA_W-1:0] cur_i3_y;
78
79      assign cur_i0_x = i_x_mem[grp_base + 4'd0];
80      assign cur_i0_y = i_y_mem[grp_base + 4'd0];
81      assign cur_i1_x = i_x_mem[grp_base + 4'd1];
82      assign cur_i1_y = i_y_mem[grp_base + 4'd1];
83      assign cur_i2_x = i_x_mem[grp_base + 4'd2];
84      assign cur_i2_y = i_y_mem[grp_base + 4'd2];
85      assign cur_i3_x = i_x_mem[grp_base + 4'd3];
86      assign cur_i3_y = i_y_mem[grp_base + 4'd3];
87
88      // active-high mask: 1 => self/invalid/hold, so new term is nulled by mj=0
89      logic [DATA_W-1:0] j0_m_eff;
90      logic [DATA_W-1:0] j1_m_eff;
91      logic [DATA_W-1:0] j2_m_eff;
92      logic [DATA_W-1:0] j3_m_eff;
93
```

```
    assign j0_m_eff = i_lane_mask[0] ? '0 : i_j_m;
    assign j1_m_eff = i_lane_mask[1] ? '0 : i_j_m;
    assign j2_m_eff = i_lane_mask[2] ? '0 : i_j_m;
    assign j3_m_eff = i_lane_mask[3] ? '0 : i_j_m;

    // outputs of datapath instances
    logic [DATA_W-1:0] dp_out0_x, dp_out0_y;
    logic [DATA_W-1:0] dp_out1_x, dp_out1_y;
    logic [DATA_W-1:0] dp_out2_x, dp_out2_y;
    logic [DATA_W-1:0] dp_out3_x, dp_out3_y;

    fourcore_bcj_datapath #(
        .DATA_W(DATA_W)
    ) u_dp (
        .i_clk    (i_clk),
        .i_rst    (i_rst),
        .i_j_x    (i_compute_en ? i_j_x    : '0),
        .i_j_y    (i_compute_en ? i_j_y    : '0),
        .i_j0_m   (i_compute_en ? j0_m_eff : '0),
        .i_j1_m   (i_compute_en ? j1_m_eff : '0),
        .i_j2_m   (i_compute_en ? j2_m_eff : '0),
        .i_j3_m   (i_compute_en ? j3_m_eff : '0),
        .i_i0_x   (i_compute_en ? cur_i0_x : '0),
        .i_i0_y   (i_compute_en ? cur_i0_y : '0),
        .i_prev0_x(prev0_x_d1),
        .i_prev0_y(prev0_y_d1),
        .o_out0_x (dp_out0_x),
        .o_out0_y (dp_out0_y),
        .i_i1_x   (i_compute_en ? cur_i1_x : '0),
        .i_i1_y   (i_compute_en ? cur_i1_y : '0),
        .i_prev1_x(prev1_x_d1),
        .i_prev1_y(prev1_y_d1),
        .o_out1_x (dp_out1_x),
        .o_out1_y (dp_out1_y),
        .i_i2_x   (i_compute_en ? cur_i2_x : '0),
        .i_i2_y   (i_compute_en ? cur_i2_y : '0),
        .i_prev2_x(prev2_x_d1),
        .i_prev2_y(prev2_y_d1),
        .o_out2_x (dp_out2_x),
        .o_out2_y (dp_out2_y),
        .i_i3_x   (i_compute_en ? cur_i3_x : '0),
        .i_i3_y   (i_compute_en ? cur_i3_y : '0),
        .i_prev3_x(prev3_x_d1),
        .i_prev3_y(prev3_y_d1),
        .o_out3_x (dp_out3_x),
        .o_out3_y (dp_out3_y)
    );

    task automatic clear_wrapper_state;
        for (i = 0; i < 16; i = i + 1) begin
            acc_x_bank[i] <= '0;
            acc_y_bank[i] <= '0;
        end

        grp_written <= 4'b0000;

        prev0_x_d0 <= '0; prev0_x_d1 <= '0; prev0_y_d0 <= '0; prev0_y_d1 <= '0;
        prev1_x_d0 <= '0; prev1_x_d1 <= '0; prev1_y_d0 <= '0; prev1_y_d1 <= '0;
        prev2_x_d0 <= '0; prev2_x_d1 <= '0; prev2_y_d0 <= '0; prev2_y_d1 <= '0;
        prev3_x_d0 <= '0; prev3_x_d1 <= '0; prev3_y_d0 <= '0; prev3_y_d1 <= '0;

        for (i = 0; i < PIPE_LAT; i = i + 1) begin
            vld_pipe[i] <= 1'b0;
            grp_pipe[i] <= 2'd0;
        end
    endtask

    // sequential state updates
    always_ff @(posedge i_clk) begin
        if (!i_rst) begin
            for (i = 0; i < 16; i = i + 1) begin
                i_x_mem[i] <= '0;
                i_y_mem[i] <= '0;
```

```
            end

            clear_wrapper_state();
        end else if (i_load_en) begin
            // load current i entry
            i_x_mem[i_load_idx] <= i_load_x;
            i_y_mem[i_load_idx] <= i_load_y;

            // clear wrapper-side cached outputs / feedback / metadata
            // during tile load so no stale values pollute the next compute phase.
            clear_wrapper_state();
        end else if (i_clear_prev) begin
            // explicit clear between phases / after sufficient drain
            clear_wrapper_state();
        end else begin
            // fixed feedback transport
            prev0_x_d1 <= prev0_x_d0; prev0_y_d1 <= prev0_y_d0; prev0_x_d0 <= dp_out0_x;
    prev0_y_d0 <= dp_out0_y;
            prev1_x_d1 <= prev1_x_d0; prev1_y_d1 <= prev1_y_d0; prev1_x_d0 <= dp_out1_x;
    prev1_y_d0 <= dp_out1_y;
            prev2_x_d1 <= prev2_x_d0; prev2_y_d1 <= prev2_y_d0; prev2_x_d0 <= dp_out2_x;
    prev2_y_d0 <= dp_out2_y;
            prev3_x_d1 <= prev3_x_d0; prev3_y_d1 <= prev3_y_d0; prev3_x_d0 <= dp_out3_x;
    prev3_y_d0 <= dp_out3_y;

            // sample current external group as the true c0 group when this issue enters
    core
            vld_pipe[0] <= i_compute_en;
            grp_pipe[0] <= i_grp_sel;
            for (int i = 0; i < PIPE_LAT-1; i++) begin
                vld_pipe[i+1] <= vld_pipe[i];
                grp_pipe[i+1] <= grp_pipe[i];
            end

            // c17-aligned out_mem writeback (wrapper result storage)
            if (vld_pipe[PIPE_LAT-1]) begin
                acc_x_bank[wb_base + 4'd0] <= dp_out0_x;
                acc_y_bank[wb_base + 4'd0] <= dp_out0_y;
                acc_x_bank[wb_base + 4'd1] <= dp_out1_x;
                acc_y_bank[wb_base + 4'd1] <= dp_out1_y;
                acc_x_bank[wb_base + 4'd2] <= dp_out2_x;
                acc_y_bank[wb_base + 4'd2] <= dp_out2_y;
                acc_x_bank[wb_base + 4'd3] <= dp_out3_x;
                acc_y_bank[wb_base + 4'd3] <= dp_out3_y;
                grp_written[grp_pipe[PIPE_LAT-1]] <= 1'b1;
            end
        end
    end

    // real output is always selected from out_mem, using the same single grp_sel
    assign o_res0_x = acc_x_bank[grp_base + 4'd0];
    assign o_res0_y = acc_y_bank[grp_base + 4'd0];
    assign o_res1_x = acc_x_bank[grp_base + 4'd1];
    assign o_res1_y = acc_y_bank[grp_base + 4'd1];
    assign o_res2_x = acc_x_bank[grp_base + 4'd2];
    assign o_res2_y = acc_y_bank[grp_base + 4'd2];
    assign o_res3_x = acc_x_bank[grp_base + 4'd3];
    assign o_res3_y = acc_y_bank[grp_base + 4'd3];
    assign o_res_vld = grp_written[i_grp_sel];

endmodule
```

A.3.5 fourcore_bcj_datapath.sv

Listing 24: code/Hardware/code/fourcore_bcj_datapath.sv

```
module fourcore_bcj_datapath #(
    parameter int DATA_W = 27
) (
```

```
    input  logic         i_clk,
    input  logic         i_rst,

    // broadcasted j position, per-lane j mass
    input  logic [DATA_W-1:0] i_j_x,
    input  logic [DATA_W-1:0] i_j_y,
    input  logic [DATA_W-1:0] i_j0_m,
    input  logic [DATA_W-1:0] i_j1_m,
    input  logic [DATA_W-1:0] i_j2_m,
    input  logic [DATA_W-1:0] i_j3_m,

    // lane 0
    input  logic [DATA_W-1:0] i_i0_x,
    input  logic [DATA_W-1:0] i_i0_y,
    input  logic [DATA_W-1:0] i_prev0_x,
    input  logic [DATA_W-1:0] i_prev0_y,
    output logic [DATA_W-1:0] o_out0_x,
    output logic [DATA_W-1:0] o_out0_y,

    // lane 1
    input  logic [DATA_W-1:0] i_i1_x,
    input  logic [DATA_W-1:0] i_i1_y,
    input  logic [DATA_W-1:0] i_prev1_x,
    input  logic [DATA_W-1:0] i_prev1_y,
    output logic [DATA_W-1:0] o_out1_x,
    output logic [DATA_W-1:0] o_out1_y,

    // lane 2
    input  logic [DATA_W-1:0] i_i2_x,
    input  logic [DATA_W-1:0] i_i2_y,
    input  logic [DATA_W-1:0] i_prev2_x,
    input  logic [DATA_W-1:0] i_prev2_y,
    output logic [DATA_W-1:0] o_out2_x,
    output logic [DATA_W-1:0] o_out2_y,

    // lane 3
    input  logic [DATA_W-1:0] i_i3_x,
    input  logic [DATA_W-1:0] i_i3_y,
    input  logic [DATA_W-1:0] i_prev3_x,
    input  logic [DATA_W-1:0] i_prev3_y,
    output logic [DATA_W-1:0] o_out3_x,
    output logic [DATA_W-1:0] o_out3_y
);

    two_body_core #(
        .DATA_W(DATA_W)
    ) u_core0 (
        .i_clk   (i_clk),
        .i_rst   (i_rst),
        .i_b1_x  (i_i0_x),
        .i_b1_y  (i_i0_y),
        .i_b2_x  (i_j_x),
        .i_b2_y  (i_j_y),
        .i_m_b2  (i_j0_m),
        .i_a_b1_x(i_prev0_x),
        .i_a_b1_y(i_prev0_y),
        .o_a_b1_x(o_out0_x),
        .o_a_b1_y(o_out0_y)
    );

    two_body_core #(
        .DATA_W(DATA_W)
    ) u_core1 (
        .i_clk   (i_clk),
        .i_rst   (i_rst),
        .i_b1_x  (i_i1_x),
        .i_b1_y  (i_i1_y),
        .i_b2_x  (i_j_x),
        .i_b2_y  (i_j_y),
        .i_m_b2  (i_j1_m),
        .i_a_b1_x(i_prev1_x),
        .i_a_b1_y(i_prev1_y),
        .o_a_b1_x(o_out1_x),
```

```
        .o_a_b1_y(o_out1_y)
    );

    two_body_core #(
        .DATA_W(DATA_W)
    ) u_core2 (
        .i_clk   (i_clk),
        .i_rst   (i_rst),
        .i_b1_x  (i_i2_x),
        .i_b1_y  (i_i2_y),
        .i_b2_x  (i_j_x),
        .i_b2_y  (i_j_y),
        .i_m_b2  (i_j2_m),
        .i_a_b1_x(i_prev2_x),
        .i_a_b1_y(i_prev2_y),
        .o_a_b1_x(o_out2_x),
        .o_a_b1_y(o_out2_y)
    );

    two_body_core #(
        .DATA_W(DATA_W)
    ) u_core3 (
        .i_clk   (i_clk),
        .i_rst   (i_rst),
        .i_b1_x  (i_i3_x),
        .i_b1_y  (i_i3_y),
        .i_b2_x  (i_j_x),
        .i_b2_y  (i_j_y),
        .i_m_b2  (i_j3_m),
        .i_a_b1_x(i_prev3_x),
        .i_a_b1_y(i_prev3_y),
        .o_a_b1_x(o_out3_x),
        .o_a_b1_y(o_out3_y)
    );

endmodule
```

A.3.6 two_body_core.sv

Listing 25: code/Hardware/code/two_body_core.sv

```
//============================================================================
// two_body_core
//
// Timing summary:
//   c0      : capture b1/b2/m2 inputs
//   c0->c2  : dx, dy
//   c2->c3  : dx2, dy2
//   c3->c5  : r2
//   c5->c7  : r2 + eps^2
//   c7->c11 : invsqrt raw
//   c11->c12: register s
//   c12->c13: s2, m2*s
//   c13->c14: k = m2*s^3
//   c14->c15: term = dx*k, dy*k (mul comb + REG)
//   c15->c17: out = prev + term_reg (FpAdd = 2)
//
// prev must be driven from outside for the c15 sampling edge.
//============================================================================

module two_body_core #(
    parameter int DATA_W = 27
) (
    input  logic        i_clk,
    input  logic        i_rst,   // active-low synchronous reset

    // 27-bit S1E8M18 inputs
    input  logic [DATA_W-1:0] i_b1_x,
    input  logic [DATA_W-1:0] i_b1_y,
```

```
    input  logic [DATA_W-1:0] i_b2_x,
    input  logic [DATA_W-1:0] i_b2_y,
    input  logic [DATA_W-1:0] i_m_b2,

    // 27-bit previous accel
    input  logic [DATA_W-1:0] i_a_b1_x,
    input  logic [DATA_W-1:0] i_a_b1_y,

    // 27-bit output accel
    output logic [DATA_W-1:0] o_a_b1_x,
    output logic [DATA_W-1:0] o_a_b1_y
);

    // eps^2 = 0.25
    localparam logic [26:0] EPSILON_SQUARE = {1'b0, 8'd125, 18'd0};

    // Body inputs
    logic [26:0] b1_x_27;
    logic [26:0] b1_y_27;
    logic [26:0] b2_x_27;
    logic [26:0] b2_y_27;
    logic [26:0] m2_27;

    assign b1_x_27 = i_b1_x;
    assign b1_y_27 = i_b1_y;
    assign b2_x_27 = i_b2_x;
    assign b2_y_27 = i_b2_y;
    assign m2_27   = i_m_b2;

    // c0: register core-format inputs
    logic [26:0] r_b1_x_c0;
    logic [26:0] r_b1_y_c0;
    logic [26:0] r_b2_x_c0;
    logic [26:0] r_b2_y_c0;
    logic [26:0] r_m2_c0;

    always_ff @(posedge i_clk) begin
        if (!i_rst) begin
            r_b1_x_c0 <= 27'd0;
            r_b1_y_c0 <= 27'd0;
            r_b2_x_c0 <= 27'd0;
            r_b2_y_c0 <= 27'd0;
            r_m2_c0   <= 27'd0;
        end else begin
            r_b1_x_c0 <= b1_x_27;
            r_b1_y_c0 <= b1_y_27;
            r_b2_x_c0 <= b2_x_27;
            r_b2_y_c0 <= b2_y_27;
            r_m2_c0   <= m2_27;
        end
    end

    // Reset-mux for fplib blocks (they have no reset)
    logic en;
    logic [26:0] b2x_in;
    logic [26:0] b2y_in;

    assign en     = i_rst;
    assign b2x_in = en ? r_b2_x_c0 : 27'd0;
    assign b2y_in = en ? r_b2_y_c0 : 27'd0;

    // c0 comb: negate b1 so dx = x2 + (-x1), dy = y2 + (-y1)
    logic [26:0] w_b1_x_neg;
    logic [26:0] w_b1_y_neg;

    FpNegate u_neg_b1x (
        .iA       (en ? r_b1_x_c0 : 27'd0),
        .oNegative(w_b1_x_neg)
    );

    FpNegate u_neg_b1y (
        .iA       (en ? r_b1_y_c0 : 27'd0),
        .oNegative(w_b1_y_neg)
```

```
    );

    // c0 -> c2: dx, dy (FpAdd = 2 cycles)
    logic [26:0] w_dx_c2;
    logic [26:0] w_dy_c2;

    FpAdd u_add_dx (
        .iCLK(i_clk),
        .iA  (b2x_in),
        .iB  (w_b1_x_neg),
        .oSum(w_dx_c2)
    );

    FpAdd u_add_dy (
        .iCLK(i_clk),
        .iA  (b2y_in),
        .iB  (w_b1_y_neg),
        .oSum(w_dy_c2)
    );

    // c2 -> c3: dx2, dy2 (mul comb + reg = 1 cycle)
    logic [26:0] w_dx2_comb;
    logic [26:0] w_dy2_comb;
    logic [26:0] r_dx2_c3;
    logic [26:0] r_dy2_c3;

    FpMul u_mul_dx2 (
        .iA   (w_dx_c2),
        .iB   (w_dx_c2),
        .oProd(w_dx2_comb)
    );

    FpMul u_mul_dy2 (
        .iA   (w_dy_c2),
        .iB   (w_dy_c2),
        .oProd(w_dy2_comb)
    );

    always_ff @(posedge i_clk) begin
        if (!i_rst) begin
            r_dx2_c3 <= 27'd0;
            r_dy2_c3 <= 27'd0;
        end else begin
            r_dx2_c3 <= w_dx2_comb;
            r_dy2_c3 <= w_dy2_comb;
        end
    end

    // c3 -> c5: r2 = dx2 + dy2 (FpAdd = 2 cycles)
    logic [26:0] w_r2_c5;

    FpAdd u_add_r2 (
        .iCLK(i_clk),
        .iA  (en ? r_dx2_c3 : 27'd0),
        .iB  (en ? r_dy2_c3 : 27'd0),
        .oSum(w_r2_c5)
    );

    // c5 -> c7: r2e = r2 + eps2 (FpAdd = 2 cycles)
    logic [26:0] w_r2e_c7;

    FpAdd u_add_r2e (
        .iCLK(i_clk),
        .iA  (en ? w_r2_c5 : 27'd0),
        .iB  (en ? EPSILON_SQUARE : 27'd0),
        .oSum(w_r2e_c7)
    );

    // c7 -> c11: s_raw = inv_sqrt(r2e)
    logic [26:0] w_s_c11;

    FpInvSqrt u_invsqrt (
        .iCLK    (i_clk),
```

```
        .iA      (en ? w_r2e_c7 : 27'd0),
        .oInvSqrt(w_s_c11)
    );

    // c11 -> c12: register s before mul usage
    logic [26:0] r_s_c12;

    always_ff @(posedge i_clk) begin
        if (!i_rst) begin
            r_s_c12 <= 27'd0;
        end else begin
            r_s_c12 <= w_s_c11;
        end
    end

    // Align m2 to c12
    logic [26:0] r_m2_pipe [12];

    always_ff @(posedge i_clk) begin
        if (!i_rst) begin
            for (int i = 0; i < 12; i++) begin
                r_m2_pipe[i] <= 27'd0;
            end
        end else begin
            r_m2_pipe[0] <= r_m2_c0;
            for (int i = 0; i < 11; i++) begin
                r_m2_pipe[i+1] <= r_m2_pipe[i];
            end
        end
    end

    logic [26:0] m2_c12;
    assign m2_c12 = r_m2_pipe[11];

    // c12 -> c13: s2 = s*s, t = m2*s
    logic [26:0] w_s2_comb;
    logic [26:0] w_t_comb;
    logic [26:0] r_s2_c13;
    logic [26:0] r_t_c13;

    FpMul u_mul_s2 (
        .iA   (r_s_c12),
        .iB   (r_s_c12),
        .oProd(w_s2_comb)
    );

    FpMul u_mul_t (
        .iA   (m2_c12),
        .iB   (r_s_c12),
        .oProd(w_t_comb)
    );

    always_ff @(posedge i_clk) begin
        if (!i_rst) begin
            r_s2_c13 <= 27'd0;
            r_t_c13  <= 27'd0;
        end else begin
            r_s2_c13 <= w_s2_comb;
            r_t_c13  <= w_t_comb;
        end
    end

    // c13 -> c14: k = m2*s^3
    logic [26:0] w_k_comb;
    logic [26:0] r_k_c14;

    FpMul u_mul_k (
        .iA   (r_t_c13),
        .iB   (r_s2_c13),
        .oProd(w_k_comb)
    );

    always_ff @(posedge i_clk) begin
```

```
248         if (!i_rst) begin
249             r_k_c14 <= 27'd0;
250         end else begin
251             r_k_c14 <= w_k_comb;
252         end
253     end
254 
255     // Align dx/dy to c14
256     logic [26:0] r_dx_pipe [12];
257     logic [26:0] r_dy_pipe [12];
258 
259     always_ff @(posedge i_clk) begin
260         if (!i_rst) begin
261             for (int i = 0; i < 12; i++) begin
262                 r_dx_pipe[i] <= 27'd0;
263                 r_dy_pipe[i] <= 27'd0;
264             end
265         end else begin
266             r_dx_pipe[0] <= w_dx_c2;
267             r_dy_pipe[0] <= w_dy_c2;
268             for (int i = 0; i < 11; i++) begin
269                 r_dx_pipe[i+1] <= r_dx_pipe[i];
270                 r_dy_pipe[i+1] <= r_dy_pipe[i];
271             end
272         end
273     end
274 
275     logic [26:0] dx_c14;
276     logic [26:0] dy_c14;
277 
278     assign dx_c14 = r_dx_pipe[11];
279     assign dy_c14 = r_dy_pipe[11];
280 
281     // c14 -> c15: pair term = dx*k, dy*k (mul comb + reg)
282     logic [26:0] w_ax_term_c15;
283     logic [26:0] w_ay_term_c15;
284     logic [26:0] r_ax_term_c15;
285     logic [26:0] r_ay_term_c15;
286 
287     FpMul u_mul_ax (
288         .iA   (dx_c14),
289         .iB   (r_k_c14),
290         .oProd(w_ax_term_c15)
291     );
292 
293     FpMul u_mul_ay (
294         .iA   (dy_c14),
295         .iB   (r_k_c14),
296         .oProd(w_ay_term_c15)
297     );
298 
299     always_ff @(posedge i_clk) begin
300         if (!i_rst) begin
301             r_ax_term_c15 <= 27'd0;
302             r_ay_term_c15 <= 27'd0;
303         end else begin
304             r_ax_term_c15 <= w_ax_term_c15;
305             r_ay_term_c15 <= w_ay_term_c15;
306         end
307     end
308 
309     // c15 -> c17: out = prev + term_reg (FpAdd = 2 cycles)
310     logic [26:0] w_out_x_c17;
311     logic [26:0] w_out_y_c17;
312 
313     FpAdd u_add_outx (
314         .iCLK(i_clk),
315         .iA  (en ? i_a_b1_x : 27'd0),
316         .iB  (en ? r_ax_term_c15 : 27'd0),
317         .oSum(w_out_x_c17)
318     );
319 
320     FpAdd u_add_outy (
```

```
321         .iCLK(i_clk),
322         .iA  (en ? i_a_b1_y : 27'd0),
323         .iB  (en ? r_ay_term_c15 : 27'd0),
324         .oSum(w_out_y_c17)
325     );
326
327     assign o_a_b1_x = w_out_x_c17;
328     assign o_a_b1_y = w_out_y_c17;
329
330 endmodule
```

A.3.7 nbody_integrator.sv

Listing 26: code/Hardware/code/nbody_integrator.sv

```
1  // Integration unit
2  //
3  // One-body update:
4  //   vx' = vx + ax
5  //   vy' = vy + ay
6  //   x'  = x  + vx'
7  //   y'  = y  + vy'
8  //
9  // The control unit supplies ax/ay as either a half-step or full-step
10 // acceleration increment for leapfrog integration
11
12 module nbody_integrator #(
13     parameter int DATA_W = 27
14 ) (
15     input  logic         clk,
16     input  logic         reset,   // active-high reset
17
18     input  logic         i_start,
19     output logic         o_done,
20
21     input  logic [DATA_W-1:0] i_x,
22     input  logic [DATA_W-1:0] i_y,
23     input  logic [DATA_W-1:0] i_vx,
24     input  logic [DATA_W-1:0] i_vy,
25     input  logic [DATA_W-1:0] i_ax,
26     input  logic [DATA_W-1:0] i_ay,
27
28     output logic [DATA_W-1:0] o_x,
29     output logic [DATA_W-1:0] o_y,
30     output logic [DATA_W-1:0] o_vx,
31     output logic [DATA_W-1:0] o_vy
32 );
33
34     typedef enum logic [1:0] {
35         ST_IDLE,
36         ST_WAIT_V,
37         ST_WAIT_XY,
38         ST_DONE
39     } state_t;
40
41     state_t state;
42     logic [2:0] wait_count;
43
44     logic [DATA_W-1:0] x_27;
45     logic [DATA_W-1:0] y_27;
46     logic [DATA_W-1:0] vx_27;
47     logic [DATA_W-1:0] vy_27;
48     logic [DATA_W-1:0] ax_27;
49     logic [DATA_W-1:0] ay_27;
50
51     logic [DATA_W-1:0] vx_new_27;
52     logic [DATA_W-1:0] vy_new_27;
53     logic [DATA_W-1:0] x_new_27;
54     logic [DATA_W-1:0] y_new_27;
```

```
55
56      FpAdd u_add_vx (
57          .iCLK(clk),
58          .iA  (vx_27),
59          .iB  (ax_27),
60          .oSum(vx_new_27)
61      );
62
63      FpAdd u_add_vy (
64          .iCLK(clk),
65          .iA  (vy_27),
66          .iB  (ay_27),
67          .oSum(vy_new_27)
68      );
69
70      FpAdd u_add_x (
71          .iCLK(clk),
72          .iA  (x_27),
73          .iB  (vx_new_27),
74          .oSum(x_new_27)
75      );
76
77      FpAdd u_add_y (
78          .iCLK(clk),
79          .iA  (y_27),
80          .iB  (vy_new_27),
81          .oSum(y_new_27)
82      );
83
84      always_ff @(posedge clk) begin
85          if (reset) begin
86              state      <= ST_IDLE;
87              wait_count <= 3'd0;
88              o_done     <= 1'b0;
89              x_27       <= '0;
90              y_27       <= '0;
91              vx_27      <= '0;
92              vy_27      <= '0;
93              ax_27      <= '0;
94              ay_27      <= '0;
95              o_x        <= '0;
96              o_y        <= '0;
97              o_vx       <= '0;
98              o_vy       <= '0;
99          end else begin
100             o_done <= 1'b0;
101
102             unique case (state)
103                 ST_IDLE: begin
104                     if (i_start) begin
105                         x_27       <= i_x;
106                         y_27       <= i_y;
107                         vx_27      <= i_vx;
108                         vy_27      <= i_vy;
109                         ax_27      <= i_ax;
110                         ay_27      <= i_ay;
111                         wait_count <= 3'd0;
112                         state      <= ST_WAIT_V;
113                     end
114                 end
115
116                 ST_WAIT_V: begin
117                     if (wait_count == 3'd3) begin
118                         wait_count <= 3'd0;
119                         state      <= ST_WAIT_XY;
120                     end else begin
121                         wait_count <= wait_count + 1'b1;
122                     end
123                 end
124
125                 ST_WAIT_XY: begin
126                     if (wait_count == 3'd3) begin
127                         o_vx   <= vx_new_27;
```

```
                        o_vy   <= vy_new_27;
                        o_x    <= x_new_27;
                        o_y    <= y_new_27;
                        state  <= ST_DONE;
                    end else begin
                        wait_count <= wait_count + 1'b1;
                    end
                end

                ST_DONE: begin
                    o_done <= 1'b1;
                    state  <= ST_IDLE;
                end

                default: begin
                    state <= ST_IDLE;
                end
            endcase
        end
    end

endmodule
```

A.3.8 nbody_mem.sv

Listing 27: code/Hardware/code/nbody_mem.sv

```
module nbody_mem #(
    parameter int MAX_BODIES = 1024,
    parameter int DATA_W = 27,
    parameter int PTR_W = $clog2(MAX_BODIES)
) (
    input  logic clk,

    // CPU writes one complete body when VY_IN is committed.
    input  logic              cpu_body_we,
    input  logic [PTR_W-1:0]  cpu_body_waddr,
    input  logic [DATA_W-1:0] cpu_x,
    input  logic [DATA_W-1:0] cpu_y,
    input  logic [DATA_W-1:0] cpu_m,
    input  logic [DATA_W-1:0] cpu_vx,
    input  logic [DATA_W-1:0] cpu_vy,

    // Shared synchronous body read port. Data is valid one clock after body_raddr is
    sampled.
    input  logic [PTR_W-1:0] body_raddr,
    output logic [DATA_W-1:0] body_x,
    output logic [DATA_W-1:0] body_y,
    output logic [DATA_W-1:0] body_m,
    output logic [DATA_W-1:0] body_vx,
    output logic [DATA_W-1:0] body_vy,
    output logic [DATA_W-1:0] body_ax,
    output logic [DATA_W-1:0] body_ay,

    // Integrator writeback port.
    input  logic              body_update_we,
    input  logic [PTR_W-1:0]  body_update_addr,
    input  logic [DATA_W-1:0] body_update_x,
    input  logic [DATA_W-1:0] body_update_y,
    input  logic [DATA_W-1:0] body_update_vx,
    input  logic [DATA_W-1:0] body_update_vy,

    // Serialized acceleration writeback port.
    input  logic              accel_we,
    input  logic [PTR_W-1:0]  accel_waddr,
    input  logic [DATA_W-1:0] accel_ax,
    input  logic [DATA_W-1:0] accel_ay
);

```

```
42      (* ramstyle = "M10K" *) logic [DATA_W-1:0] x_mem  [0:MAX_BODIES-1];
43      (* ramstyle = "M10K" *) logic [DATA_W-1:0] y_mem  [0:MAX_BODIES-1];
44      (* ramstyle = "M10K" *) logic [DATA_W-1:0] m_mem  [0:MAX_BODIES-1];
45      (* ramstyle = "M10K" *) logic [DATA_W-1:0] vx_mem [0:MAX_BODIES-1];
46      (* ramstyle = "M10K" *) logic [DATA_W-1:0] vy_mem [0:MAX_BODIES-1];
47      (* ramstyle = "M10K" *) logic [DATA_W-1:0] ax_mem [0:MAX_BODIES-1];
48      (* ramstyle = "M10K" *) logic [DATA_W-1:0] ay_mem [0:MAX_BODIES-1];
49
50      logic             x_we;
51      logic             y_we;
52      logic             m_we;
53      logic             vx_we;
54      logic             vy_we;
55      logic             ax_we;
56      logic             ay_we;
57      logic [PTR_W-1:0] x_waddr;
58      logic [PTR_W-1:0] y_waddr;
59      logic [PTR_W-1:0] m_waddr;
60      logic [PTR_W-1:0] vx_waddr;
61      logic [PTR_W-1:0] vy_waddr;
62      logic [PTR_W-1:0] ax_waddr;
63      logic [PTR_W-1:0] ay_waddr;
64      logic [DATA_W-1:0] x_wdata;
65      logic [DATA_W-1:0] y_wdata;
66      logic [DATA_W-1:0] m_wdata;
67      logic [DATA_W-1:0] vx_wdata;
68      logic [DATA_W-1:0] vy_wdata;
69      logic [DATA_W-1:0] ax_wdata;
70      logic [DATA_W-1:0] ay_wdata;
71
72      always_comb begin
73          x_we    = cpu_body_we;
74          y_we    = cpu_body_we;
75          m_we    = cpu_body_we;
76          vx_we   = cpu_body_we;
77          vy_we   = cpu_body_we;
78          ax_we   = cpu_body_we;
79          ay_we   = cpu_body_we;
80          x_waddr = cpu_body_waddr;
81          y_waddr = cpu_body_waddr;
82          m_waddr = cpu_body_waddr;
83          vx_waddr = cpu_body_waddr;
84          vy_waddr = cpu_body_waddr;
85          ax_waddr = cpu_body_waddr;
86          ay_waddr = cpu_body_waddr;
87          x_wdata = cpu_x;
88          y_wdata = cpu_y;
89          m_wdata = cpu_m;
90          vx_wdata = cpu_vx;
91          vy_wdata = cpu_vy;
92          ax_wdata = '0;
93          ay_wdata = '0;
94
95          if (body_update_we) begin
96              x_we    = 1'b1;
97              y_we    = 1'b1;
98              vx_we   = 1'b1;
99              vy_we   = 1'b1;
100             x_waddr = body_update_addr;
101             y_waddr = body_update_addr;
102             vx_waddr = body_update_addr;
103             vy_waddr = body_update_addr;
104             x_wdata = body_update_x;
105             y_wdata = body_update_y;
106             vx_wdata = body_update_vx;
107             vy_wdata = body_update_vy;
108         end
109
110         if (accel_we) begin
111             ax_we    = 1'b1;
112             ay_we    = 1'b1;
113             ax_waddr = accel_waddr;
114             ay_waddr = accel_waddr;
```

```
            ax_wdata = accel_ax;
            ay_wdata = accel_ay;
        end
    end

    always_ff @(posedge clk) begin
        body_x  <= x_mem[body_raddr];
        body_y  <= y_mem[body_raddr];
        body_m  <= m_mem[body_raddr];
        body_vx <= vx_mem[body_raddr];
        body_vy <= vy_mem[body_raddr];
        body_ax <= ax_mem[body_raddr];
        body_ay <= ay_mem[body_raddr];

        if (x_we)  x_mem[x_waddr]   <= x_wdata;
        if (y_we)  y_mem[y_waddr]   <= y_wdata;
        if (m_we)  m_mem[m_waddr]   <= m_wdata;
        if (vx_we) vx_mem[vx_waddr] <= vx_wdata;
        if (vy_we) vy_mem[vy_waddr] <= vy_wdata;
        if (ax_we) ax_mem[ax_waddr] <= ax_wdata;
        if (ay_we) ay_mem[ay_waddr] <= ay_wdata;
    end

endmodule
```

A.3.9 framebuffer_ram.sv

Listing 28: code/Hardware/code/framebuffer_ram.sv

```
// Simple dual-port 1-bit-per-pixel framebuffer storage.
//
// Port A is used by the Avalon-MM slave for CPU writes.
// Port B is used by the VGA scanout logic for continuous reads.

module framebuffer_ram #(
    parameter int FB_WORDS = 9600,
    parameter int ADDR_W   = 14
) (
    input  logic              clk,

    input  logic              port_a_we,
    input  logic [ADDR_W-1:0] port_a_addr,
    input  logic [31:0]       port_a_writedata,

    input  logic [ADDR_W-1:0] port_b_addr,
    output logic [31:0]       port_b_readdata
);

    (* ramstyle = "M10K" *) logic [31:0] mem [0:FB_WORDS-1];

    always_ff @(posedge clk) begin
        if (port_a_we) begin
            mem[port_a_addr] <= port_a_writedata;
        end

        port_b_readdata <= mem[port_b_addr];
    end

endmodule
```

A.3.10 FpAdd.sv

Listing 29: code/Hardware/code/FpAdd.sv

```
/*************************************************************************
```

```
 * Floating Point Adder                                                  *
 * 2-stage pipeline                                                      *
 ************************************************************************/
module FpAdd (
    input  logic        iCLK,
    input  logic [26:0] iA,
    input  logic [26:0] iB,
    output logic [26:0] oSum
);

    // Extract fields of A and B.
    logic        A_s;
    logic [7:0]  A_e;
    logic [17:0] A_f;
    logic        B_s;
    logic [7:0]  B_e;
    logic [17:0] B_f;
    logic        A_larger;

    assign A_s = iA[26];
    assign A_e = iA[25:18];
    assign A_f = {1'b1, iA[17:1]};
    assign B_s = iB[26];
    assign B_e = iB[25:18];
    assign B_f = {1'b1, iB[17:1]};

    // Shift fractions of A and B so that they align.
    logic [7:0]  exp_diff_A;
    logic [7:0]  exp_diff_B;
    logic [7:0]  larger_exp;
    logic [36:0] A_f_shifted;
    logic [36:0] B_f_shifted;

    assign exp_diff_A = B_e - A_e; // if B bigger
    assign exp_diff_B = A_e - B_e; // if A bigger

    assign larger_exp = (B_e > A_e) ? B_e : A_e;

    assign A_f_shifted = A_larger              ? {1'b0, A_f, 18'b0} :
                         (exp_diff_A > 8'd35) ? 37'b0 :
                         ({1'b0, A_f, 18'b0} >> exp_diff_A);
    assign B_f_shifted = ~A_larger             ? {1'b0, B_f, 18'b0} :
                         (exp_diff_B > 8'd35) ? 37'b0 :
                         ({1'b0, B_f, 18'b0} >> exp_diff_B);

    // Determine which of A, B is larger.
    assign A_larger =    (A_e > B_e)                  ? 1'b1 :
                         ((A_e == B_e) && (A_f > B_f)) ? 1'b1 :
                         1'b0;

    // Calculate sum or difference of shifted fractions.
    logic [36:0] pre_sum;
    assign pre_sum = ((A_s ^ B_s) &  A_larger) ? A_f_shifted - B_f_shifted :
                     ((A_s ^ B_s) & ~A_larger) ? B_f_shifted - A_f_shifted :
                     A_f_shifted + B_f_shifted;

    // Buffer midway results.
    logic [36:0] buf_pre_sum;
    logic [7:0]  buf_larger_exp;
    logic        buf_A_e_zero;
    logic        buf_B_e_zero;
    logic [26:0] buf_A;
    logic [26:0] buf_B;
    logic        buf_oSum_s;

    always_ff @(posedge iCLK) begin
        buf_pre_sum    <= pre_sum;
        buf_larger_exp <= larger_exp;
        buf_A_e_zero   <= (A_e == 8'b0);
        buf_B_e_zero   <= (B_e == 8'b0);
        buf_A          <= iA;
        buf_B          <= iB;
        buf_oSum_s     <= A_larger ? A_s : B_s;
```

```
    end

    // Convert to positive fraction and a sign bit.
    logic [36:0] pre_frac;
    assign pre_frac = buf_pre_sum;

    // Determine output fraction and exponent change with position of first 1.
    logic [17:0] oSum_f;
    logic [7:0]  shft_amt;

    assign shft_amt = pre_frac[36] ? 8'd0  : pre_frac[35] ? 8'd1  :
                      pre_frac[34] ? 8'd2  : pre_frac[33] ? 8'd3  :
                      pre_frac[32] ? 8'd4  : pre_frac[31] ? 8'd5  :
                      pre_frac[30] ? 8'd6  : pre_frac[29] ? 8'd7  :
                      pre_frac[28] ? 8'd8  : pre_frac[27] ? 8'd9  :
                      pre_frac[26] ? 8'd10 : pre_frac[25] ? 8'd11 :
                      pre_frac[24] ? 8'd12 : pre_frac[23] ? 8'd13 :
                      pre_frac[22] ? 8'd14 : pre_frac[21] ? 8'd15 :
                      pre_frac[20] ? 8'd16 : pre_frac[19] ? 8'd17 :
                      pre_frac[18] ? 8'd18 : pre_frac[17] ? 8'd19 :
                      pre_frac[16] ? 8'd20 : pre_frac[15] ? 8'd21 :
                      pre_frac[14] ? 8'd22 : pre_frac[13] ? 8'd23 :
                      pre_frac[12] ? 8'd24 : pre_frac[11] ? 8'd25 :
                      pre_frac[10] ? 8'd26 : pre_frac[9]  ? 8'd27 :
                      pre_frac[8]  ? 8'd28 : pre_frac[7]  ? 8'd29 :
                      pre_frac[6]  ? 8'd30 : pre_frac[5]  ? 8'd31 :
                      pre_frac[4]  ? 8'd32 : pre_frac[3]  ? 8'd33 :
                      pre_frac[2]  ? 8'd34 : pre_frac[1]  ? 8'd35 :
                      pre_frac[0]  ? 8'd36 : 8'd37;

    logic [53:0] pre_frac_shft;
    logic [53:0] uflow_shift;

    // The shift +1 is because high order bit is not stored, but implied.
    assign pre_frac_shft = {pre_frac, 17'b0} << (shft_amt + 8'd1);
    assign uflow_shift   = {pre_frac, 17'b0} << shft_amt;
    assign oSum_f        = pre_frac_shft[53:36];

    logic [7:0] oSum_e;
    assign oSum_e = buf_larger_exp - shft_amt + 8'd1;

    // Detect underflow. If top bit of mantissa is not set, then denormalize
    logic underflow;
    assign underflow = ~uflow_shift[53];

    always_ff @(posedge iCLK) begin
        oSum <= (buf_A_e_zero && buf_B_e_zero) ? 27'b0 :
                buf_A_e_zero                   ? buf_B :
                buf_B_e_zero                   ? buf_A :
                underflow                      ? 27'b0 :
                (pre_frac == 37'b0)            ? 27'b0 :
                {buf_oSum_s, oSum_e, oSum_f};
    end

endmodule
```

A.3.11 FpMul.sv

Listing 30: code/Hardware/code/FpMul.sv

```
/*****************************************************************************
 * Floating Point Multiplier                                                 *
 * Fully Combinational                                                       *
 *****************************************************************************/
module FpMul (
    input  logic [26:0] iA,    // First input
    input  logic [26:0] iB,    // Second input
    output logic [26:0] oProd  // Product
);
```

```
10
11      // Extract fields of A and B.
12      logic        A_s;
13      logic [7:0]  A_e;
14      logic [17:0] A_f;
15      logic        B_s;
16      logic [7:0]  B_e;
17      logic [17:0] B_f;
18
19      assign A_s = iA[26];
20      assign A_e = iA[25:18];
21      assign A_f = {1'b1, iA[17:1]};
22      assign B_s = iB[26];
23      assign B_e = iB[25:18];
24      assign B_f = {1'b1, iB[17:1]};
25
26      // XOR sign bits to determine product sign.
27      logic oProd_s;
28      assign oProd_s = A_s ^ B_s;
29
30      // Multiply the fractions of A and B.
31      logic [35:0] pre_prod_frac;
32      assign pre_prod_frac = A_f * B_f;
33
34      // Add exponents of A and B.
35      logic [8:0] pre_prod_exp;
36      assign pre_prod_exp = A_e + B_e;
37
38      // If top bit of product frac is 0, shift left one.
39      logic [7:0]  oProd_e;
40      logic [17:0] oProd_f;
41
42      assign oProd_e = pre_prod_frac[35] ? (pre_prod_exp - 9'd126) :
43                                           (pre_prod_exp - 9'd127);
44      assign oProd_f = pre_prod_frac[35] ? pre_prod_frac[34:17] :
45                                           pre_prod_frac[33:16];
46
47      // Detect underflow.
48      logic underflow;
49      assign underflow = pre_prod_exp < 9'h80;
50
51      // Detect zero conditions (either product frac doesn't start with 1, or underflow).
52      assign oProd = underflow     ? 27'b0 :
53                     (B_e == 8'd0) ? 27'b0 :
54                     (A_e == 8'd0) ? 27'b0 :
55                     {oProd_s, oProd_e, oProd_f};
56
57  endmodule
```

A.3.12 FpInvSqrt.sv

Listing 31: code/Hardware/code/FpInvSqrt.sv

```
1   /*****************************************************************************
2    * Floating Point Fast Inverse Square Root                                   *
3    * 5-stage pipeline                                                          *
4    * http://en.wikipedia.org/wiki/Fast_inverse_square_root                     *
5    * Magic number 27'd49920718                                                 *
6    * 1.5 = 27'd33423360                                                        *
7    *****************************************************************************/
8   module FpInvSqrt (
9       input  logic        iCLK,
10      input  logic [26:0] iA,
11      output logic [26:0] oInvSqrt
12  );
13
14      // Extract fields of A.
15      logic        A_s;
16      logic [7:0]  A_e;
```

```
17      logic [17:0] A_f;
18
19      assign A_s = iA[26];
20      assign A_e = iA[25:18];
21      assign A_f = iA[17:0];
22
23      // Stage 1.
24      logic [26:0] y_1;
25      logic [26:0] y_1_out;
26      logic [26:0] half_iA_1;
27
28      assign y_1       = 27'd49920718 - (iA >> 1);
29      assign half_iA_1 = {A_s, A_e - 8'd1, A_f};
30
31      FpMul s1_mult (
32          .iA   (y_1),
33          .iB   (y_1),
34          .oProd(y_1_out)
35      );
36
37      // Stage 2.
38      logic [26:0] y_2;
39      logic [26:0] mult_2_in;
40      logic [26:0] half_iA_2;
41      logic [26:0] y_2_out;
42
43      FpMul s2_mult (
44          .iA   (half_iA_2),
45          .iB   (mult_2_in),
46          .oProd(y_2_out)
47      );
48
49      // Stage 3.
50      logic [26:0] y_3;
51      logic [26:0] add_3_in;
52      logic [26:0] y_3_out;
53
54      FpAdd s3_add (
55          .iCLK(iCLK),
56          .iA  ({~add_3_in[26], add_3_in[25:0]}),
57          .iB  (27'd33423360),
58          .oSum(y_3_out)
59      );
60
61      // Stage 4.
62      logic [26:0] y_4;
63
64      // Stage 5.
65      logic [26:0] y_5;
66
67      FpMul s5_mult (
68          .iA   (y_5),
69          .iB   (y_3_out),
70          .oProd(oInvSqrt)
71      );
72
73      always_ff @(posedge iCLK) begin
74          // Stage 1 to 2.
75          y_2       <= y_1;
76          mult_2_in <= y_1_out;
77          half_iA_2 <= half_iA_1;
78
79          // Stage 2 to 3.
80          y_3       <= y_2;
81          add_3_in  <= y_2_out;
82
83          // Stage 3 to 4.
84          y_4 <= y_3;
85
86          // Stage 4 to 5.
87          y_5 <= y_4;
88      end
89
```

```
endmodule
```

A.3.13 FpNegate.sv

Listing 32: code/Hardware/code/FpNegate.sv

```
/*****************************************************************************
 * Floating Point sign negation                                              *
 * Combinational                                                             *
 *****************************************************************************/
module FpNegate (
    input  logic [26:0] iA,
    output logic [26:0] oNegative
);

    // Extract fields of A.
    logic        A_s;
    logic [7:0]  A_e;
    logic [17:0] A_f;

    assign A_s = iA[26];
    assign A_e = iA[25:18];
    assign A_f = iA[17:0];

    // Flip bit 26.
    assign oNegative = {~A_s, A_e, A_f};

endmodule
```

A.4 Hardware Testbenches

A.4.1 tb_nbody_control.sv

Listing 33: code/Hardware/tb/tb_nbody_control.sv

```
`timescale 1ns/1ps

module tb_nbody_control;

  localparam int MAX_BODIES = 1024;
  localparam int DATA_W = 27;
  localparam int PTR_W = $clog2(MAX_BODIES);
  localparam int RUN_TIMEOUT_CYCLES = 400000;

  localparam string INPUT_FILE = "tb/frame_input/frame0_1024binit200_27bits.txt";
  localparam string OUT_FILE   = "tb/output/control_accel_1024binit200_27bits.txt";

  logic clk;
  logic reset;

  logic        go;
  logic        read_enable;
  logic        first_step;
  logic [31:0] n_bodies;
  logic [31:0] gap;
  logic        done;

  logic [PTR_W-1:0]  body_raddr;
  logic [DATA_W-1:0] body_x;
  logic [DATA_W-1:0] body_y;
  logic [DATA_W-1:0] body_m;
  logic [DATA_W-1:0] body_vx;
  logic [DATA_W-1:0] body_vy;
  logic [DATA_W-1:0] body_ax;
```

```
30    logic [DATA_W-1:0] body_ay;
31
32    logic              body_update_we;
33    logic [PTR_W-1:0]  body_update_addr;
34    logic [DATA_W-1:0] body_update_x;
35    logic [DATA_W-1:0] body_update_y;
36    logic [DATA_W-1:0] body_update_vx;
37    logic [DATA_W-1:0] body_update_vy;
38
39    logic              accel_we;
40    logic [PTR_W-1:0]  accel_waddr;
41    logic [DATA_W-1:0] accel_ax;
42    logic [DATA_W-1:0] accel_ay;
43
44    logic              cpu_body_we;
45    logic [PTR_W-1:0]  cpu_body_waddr;
46    logic [DATA_W-1:0] cpu_x;
47    logic [DATA_W-1:0] cpu_y;
48    logic [DATA_W-1:0] cpu_m;
49    logic [DATA_W-1:0] cpu_vx;
50    logic [DATA_W-1:0] cpu_vy;
51
52    logic [DATA_W-1:0] init_x  [0:MAX_BODIES-1];
53    logic [DATA_W-1:0] init_y  [0:MAX_BODIES-1];
54    logic [DATA_W-1:0] init_m  [0:MAX_BODIES-1];
55    logic [DATA_W-1:0] init_vx [0:MAX_BODIES-1];
56    logic [DATA_W-1:0] init_vy [0:MAX_BODIES-1];
57    logic [DATA_W-1:0] final_x  [0:MAX_BODIES-1];
58    logic [DATA_W-1:0] final_y  [0:MAX_BODIES-1];
59    logic [DATA_W-1:0] final_vx [0:MAX_BODIES-1];
60    logic [DATA_W-1:0] final_vy [0:MAX_BODIES-1];
61    logic [DATA_W-1:0] final_ax [0:MAX_BODIES-1];
62    logic [DATA_W-1:0] final_ay [0:MAX_BODIES-1];
63    logic              accel_seen [0:MAX_BODIES-1];
64    logic              update_seen [0:MAX_BODIES-1];
65
66    int err_count;
67    int accel_count;
68    int update_count;
69    int fo;
70
71    initial clk = 1'b0;
72    always #5 clk = ~clk;
73
74    nbody_control #(
75        .MAX_BODIES(MAX_BODIES),
76        .DATA_W(DATA_W)
77    ) dut_control (
78        .clk             (clk),
79        .reset           (reset),
80        .go              (go),
81        .read_enable     (read_enable),
82        .first_step      (first_step),
83        .n_bodies        (n_bodies),
84        .gap             (gap),
85        .done            (done),
86        .body_raddr      (body_raddr),
87        .body_x          (body_x),
88        .body_y          (body_y),
89        .body_m          (body_m),
90        .body_vx         (body_vx),
91        .body_vy         (body_vy),
92        .body_ax         (body_ax),
93        .body_ay         (body_ay),
94        .body_update_we  (body_update_we),
95        .body_update_addr(body_update_addr),
96        .body_update_x   (body_update_x),
97        .body_update_y   (body_update_y),
98        .body_update_vx  (body_update_vx),
99        .body_update_vy  (body_update_vy),
100       .accel_we        (accel_we),
101       .accel_waddr     (accel_waddr),
102       .accel_ax        (accel_ax),
```

```
        .accel_ay        (accel_ay)
    );

    nbody_mem #(
        .MAX_BODIES(MAX_BODIES),
        .DATA_W(DATA_W),
        .PTR_W(PTR_W)
    ) dut_mem (
        .clk             (clk),
        .cpu_body_we     (cpu_body_we),
        .cpu_body_waddr  (cpu_body_waddr),
        .cpu_x           (cpu_x),
        .cpu_y           (cpu_y),
        .cpu_m           (cpu_m),
        .cpu_vx          (cpu_vx),
        .cpu_vy          (cpu_vy),
        .body_raddr      (body_raddr),
        .body_x          (body_x),
        .body_y          (body_y),
        .body_m          (body_m),
        .body_vx         (body_vx),
        .body_vy         (body_vy),
        .body_ax         (body_ax),
        .body_ay         (body_ay),
        .body_update_we  (body_update_we),
        .body_update_addr(body_update_addr),
        .body_update_x   (body_update_x),
        .body_update_y   (body_update_y),
        .body_update_vx  (body_update_vx),
        .body_update_vy  (body_update_vy),
        .accel_we        (accel_we),
        .accel_waddr     (accel_waddr),
        .accel_ax        (accel_ax),
        .accel_ay        (accel_ay)
    );

    task automatic clear_inputs;
      begin
        go = 1'b0;
        read_enable = 1'b1;
        first_step = 1'b1;
        n_bodies = MAX_BODIES;
        gap = 32'd1;
        cpu_body_we = 1'b0;
        cpu_body_waddr = '0;
        cpu_x = '0;
        cpu_y = '0;
        cpu_m = '0;
        cpu_vx = '0;
        cpu_vy = '0;
      end
    endtask

    task automatic read_frame_file(input string fname);
      int fd;
      string line;
      int idx;
      logic [DATA_W-1:0] lpx, lpy, lvx, lvy, lm;
      int got;
      begin
        for (int i = 0; i < MAX_BODIES; i++) begin
          init_x[i] = '0;
          init_y[i] = '0;
          init_m[i] = '0;
          init_vx[i] = '0;
          init_vy[i] = '0;
          final_x[i] = '0;
          final_y[i] = '0;
          final_vx[i] = '0;
          final_vy[i] = '0;
          final_ax[i] = '0;
          final_ay[i] = '0;
          accel_seen[i] = 1'b0;
```

```
        update_seen[i] = 1'b0;
      end

      fd = $fopen(fname, "r");
      if (fd == 0) $fatal(1, "ERROR: cannot open INPUT_FILE=%s", fname);

      while (!$feof(fd)) begin
        line = "";
        void'($fgets(line, fd));

        if (line.len() == 0) continue;
        if (line.substr(0, 0) == "#") continue;

        got = $sscanf(line, "%d %h %h %h %h %h",
                      idx, lpx, lpy, lvx, lvy, lm);

        if (got == 6 && idx >= 0 && idx < MAX_BODIES) begin
          init_x[idx] = lpx;
          init_y[idx] = lpy;
          init_vx[idx] = lvx;
          init_vy[idx] = lvy;
          init_m[idx] = lm;
        end
      end

      $fclose(fd);
    end
  endtask

  task automatic cpu_write_body(
    input [PTR_W-1:0] addr,
    input [DATA_W-1:0] x,
    input [DATA_W-1:0] y,
    input [DATA_W-1:0] mass,
    input [DATA_W-1:0] vx,
    input [DATA_W-1:0] vy
  );
    begin
      @(negedge clk);
      cpu_body_we = 1'b1;
      cpu_body_waddr = addr;
      cpu_x = x;
      cpu_y = y;
      cpu_m = mass;
      cpu_vx = vx;
      cpu_vy = vy;
      @(posedge clk);
      @(negedge clk);
      cpu_body_we = 1'b0;
    end
  endtask

  task automatic preload_memory;
    begin
      for (int i = 0; i < MAX_BODIES; i++) begin
        cpu_write_body(PTR_W'(i), init_x[i], init_y[i], init_m[i], init_vx[i], init_vy[i]);
      end
    end
  endtask

  task automatic pulse_go;
    begin
      @(negedge clk);
      go = 1'b1;
      @(posedge clk);
      @(negedge clk);
      go = 1'b0;
    end
  endtask

  task automatic run_and_capture;
    int cycles;
    begin
```

```
249         cycles = 0;
250         accel_count = 0;
251         update_count = 0;
252 
253         pulse_go();
254 
255         while (done !== 1'b1 && cycles < RUN_TIMEOUT_CYCLES) begin
256           @(posedge clk);
257           #1;
258           cycles++;
259 
260           if (accel_we) begin
261             if (accel_waddr >= MAX_BODIES) begin
262               $display("ERROR accel_waddr out of range: %0d", accel_waddr);
263               err_count++;
264             end else begin
265               if (accel_seen[accel_waddr]) begin
266                 $display("ERROR duplicate accel write addr=%0d", accel_waddr);
267                 err_count++;
268               end
269               accel_seen[accel_waddr] = 1'b1;
270               final_ax[accel_waddr] = accel_ax;
271               final_ay[accel_waddr] = accel_ay;
272               accel_count++;
273             end
274           end
275 
276           if (body_update_we) begin
277             if (body_update_addr >= MAX_BODIES) begin
278               $display("ERROR body_update_addr out of range: %0d", body_update_addr);
279               err_count++;
280             end else begin
281               if (update_seen[body_update_addr]) begin
282                 $display("ERROR duplicate body update addr=%0d", body_update_addr);
283                 err_count++;
284               end
285               update_seen[body_update_addr] = 1'b1;
286               final_x[body_update_addr] = body_update_x;
287               final_y[body_update_addr] = body_update_y;
288               final_vx[body_update_addr] = body_update_vx;
289               final_vy[body_update_addr] = body_update_vy;
290               update_count++;
291             end
292           end
293         end
294 
295         if (done !== 1'b1) begin
296           $display("ERROR timed out waiting for done after %0d cycles", cycles);
297           err_count++;
298         end
299 
300         if (accel_count != MAX_BODIES) begin
301           $display("ERROR accel_count=%0d expected %0d", accel_count, MAX_BODIES);
302           err_count++;
303         end
304 
305         if (update_count != MAX_BODIES) begin
306           $display("ERROR update_count=%0d expected %0d", update_count, MAX_BODIES);
307           err_count++;
308         end
309 
310         for (int i = 0; i < MAX_BODIES; i++) begin
311           if (!accel_seen[i]) begin
312             if (err_count < 32) begin
313               $display("ERROR missing accel write addr=%0d", i);
314             end
315             err_count++;
316           end
317 
318           if (!update_seen[i]) begin
319             if (err_count < 32) begin
320               $display("ERROR missing body update addr=%0d", i);
321             end
```

```
            err_count++;
          end
        end

        fo = $fopen(OUT_FILE, "w");
        if (fo == 0) $fatal(1, "ERROR: cannot open OUT_FILE=%s", OUT_FILE);
        $fwrite(fo, "# i x y vx vy ax ay (S1E8M18 hex)\n");

        for (int i = 0; i < MAX_BODIES; i++) begin
          $fwrite(fo, "%4d  %07h  %07h  %07h  %07h  %07h  %07h\n",
                  i, final_x[i], final_y[i], final_vx[i], final_vy[i],
                  final_ax[i], final_ay[i]);
        end

        $fclose(fo);

        $display("DONE. cycles=%0d accel_count=%0d update_count=%0d wrote %s",
                 cycles, accel_count, update_count, OUT_FILE);
      end
    endtask

    initial begin
      err_count = 0;
      reset = 1'b1;
      clear_inputs();
      read_frame_file(INPUT_FILE);

      repeat (4) @(posedge clk);
      @(negedge clk);
      reset = 1'b0;

      repeat (2) @(posedge clk);
      preload_memory();
      repeat (2) @(posedge clk);

      run_and_capture();

      if (err_count == 0) begin
        $display("PASS: tb_nbody_control completed without errors");
      end else begin
        $display("FAIL: tb_nbody_control saw %0d error(s)", err_count);
      end

      $finish;
    end

endmodule
```

A.4.2 tb_four_core_wrapper.sv

Listing 34: code/Hardware/tb/tb_four_core_wrapper.sv

```
1  `timescale 1ns/1ps
2
3  // Full-frame testbench for four_core_wrapper.
4  //
5  // This drives the wrapper the same way nbody_control intends to use it:
6  //   - load one 16-body tile into wrapper local memory
7  //   - clear wrapper accumulation state once for the tile
8  //   - for each j body, hold j stable while groups 0,1,2,3 are issued
9  //   - after the pipeline drains, sample all 16 accumulated lane outputs
10 //
11 // Output format:
12 //   # i ax ay (S1E8M18 hex)
13
14 module tb_four_core_wrapper;
15
16   localparam int DATA_W = 27;
17   localparam int N_BODIES = 1024;
```

```
18    localparam int TILE_SIZE = 16;
19    localparam int GROUP_SIZE = 4;
20    localparam int N_GROUPS = TILE_SIZE / GROUP_SIZE;
21    localparam int PIPE_LAT = 18;
22    localparam int RUN_TIMEOUT_CYCLES = 2000000;
23
24    localparam string INPUT_FILE = "tb/frame_input/frame0_1024binit200_27bits.txt";
25    localparam string OUT_FILE   = "tb/output/four_core_wrapper_accel_1024binit200_27bits.txt"
        ;
26
27    logic clk;
28    logic rst_n;
29    logic i_clear_prev;
30    logic i_load_en;
31    logic i_compute_en;
32    logic [3:0] i_load_idx;
33    logic [DATA_W-1:0] i_load_x;
34    logic [DATA_W-1:0] i_load_y;
35    logic [1:0] i_grp_sel;
36    logic [DATA_W-1:0] i_j_x;
37    logic [DATA_W-1:0] i_j_y;
38    logic [DATA_W-1:0] i_j_m;
39    logic [3:0] i_lane_mask;
40
41    wire [DATA_W-1:0] o_res0_x;
42    wire [DATA_W-1:0] o_res0_y;
43    wire [DATA_W-1:0] o_res1_x;
44    wire [DATA_W-1:0] o_res1_y;
45    wire [DATA_W-1:0] o_res2_x;
46    wire [DATA_W-1:0] o_res2_y;
47    wire [DATA_W-1:0] o_res3_x;
48    wire [DATA_W-1:0] o_res3_y;
49    wire              o_res_vld;
50
51    logic [DATA_W-1:0] px [0:N_BODIES-1];
52    logic [DATA_W-1:0] py [0:N_BODIES-1];
53    logic [DATA_W-1:0] m  [0:N_BODIES-1];
54    logic [DATA_W-1:0] ax [0:N_BODIES-1];
55    logic [DATA_W-1:0] ay [0:N_BODIES-1];
56
57    int err_count;
58    int fo;
59    int cycles;
60
61    initial clk = 1'b0;
62    always #5 clk = ~clk;
63
64    four_core_wrapper #(
65        .DATA_W(DATA_W)
66    ) dut (
67        .i_clk       (clk),
68        .i_rst       (rst_n),
69        .i_clear_prev(i_clear_prev),
70        .i_load_en   (i_load_en),
71        .i_compute_en(i_compute_en),
72        .i_load_idx  (i_load_idx),
73        .i_load_x    (i_load_x),
74        .i_load_y    (i_load_y),
75        .i_grp_sel   (i_grp_sel),
76        .i_j_x       (i_j_x),
77        .i_j_y       (i_j_y),
78        .i_j_m       (i_j_m),
79        .i_lane_mask (i_lane_mask),
80        .o_res0_x    (o_res0_x),
81        .o_res0_y    (o_res0_y),
82        .o_res1_x    (o_res1_x),
83        .o_res1_y    (o_res1_y),
84        .o_res2_x    (o_res2_x),
85        .o_res2_y    (o_res2_y),
86        .o_res3_x    (o_res3_x),
87        .o_res3_y    (o_res3_y),
88        .o_res_vld   (o_res_vld)
89    );
```

```
90
91    task automatic drive_idle;
92      begin
93        i_clear_prev = 1'b0;
94        i_load_en = 1'b0;
95        i_compute_en = 1'b0;
96        i_load_idx = 4'd0;
97        i_load_x = '0;
98        i_load_y = '0;
99        i_j_x = '0;
100       i_j_y = '0;
101       i_j_m = '0;
102       i_lane_mask = 4'h0;
103     end
104   endtask
105
106   task automatic read_frame_file(input string fname);
107     int fd;
108     string line;
109     int idx;
110     logic [DATA_W-1:0] lpx, lpy, lvx, lvy, lm;
111     int got;
112     begin
113       for (int t = 0; t < N_BODIES; t++) begin
114         px[t] = '0;
115         py[t] = '0;
116         m[t] = '0;
117         ax[t] = '0;
118         ay[t] = '0;
119       end
120
121       fd = $fopen(fname, "r");
122       if (fd == 0) $fatal(1, "ERROR: cannot open INPUT_FILE=%s", fname);
123
124       while (!$feof(fd)) begin
125         line = "";
126         void'($fgets(line, fd));
127
128         if (line.len() == 0) continue;
129         if (line.substr(0, 0) == "#") continue;
130
131         got = $sscanf(line, "%d %h %h %h %h %h",
132                       idx, lpx, lpy, lvx, lvy, lm);
133
134         if (got == 6 && idx >= 0 && idx < N_BODIES) begin
135           px[idx] = lpx;
136           py[idx] = lpy;
137           m[idx] = lm;
138         end
139       end
140
141       $fclose(fd);
142     end
143   endtask
144
145   task automatic load_tile(input int tile_base);
146     int lane;
147     int body_idx;
148     begin
149       for (lane = 0; lane < TILE_SIZE; lane++) begin
150         body_idx = tile_base + lane;
151         @(negedge clk);
152         drive_idle();
153         i_load_en = 1'b1;
154         i_load_idx = lane[3:0];
155         if (body_idx < N_BODIES) begin
156           i_load_x = px[body_idx];
157           i_load_y = py[body_idx];
158         end else begin
159           i_load_x = '0;
160           i_load_y = '0;
161         end
162         @(posedge clk);
```

```
      end

      @(negedge clk);
      drive_idle();
    end
  endtask

  task automatic pulse_clear_prev;
    begin
      @(negedge clk);
      drive_idle();
      i_grp_sel = 2'd0;
      i_clear_prev = 1'b1;
      @(posedge clk);
      @(negedge clk);
      i_clear_prev = 1'b0;
    end
  endtask

  function automatic logic [3:0] make_lane_mask(input int tile_base, input int grp, input int j_idx);
    int lane_body;
    begin
      for (int lane = 0; lane < GROUP_SIZE; lane++) begin
        lane_body = tile_base + grp * GROUP_SIZE + lane;
        make_lane_mask[lane] = (lane_body >= N_BODIES) || (lane_body == j_idx);
      end
    end
  endfunction

  task automatic issue_one_compute(
    input [1:0] grp_sel,
    input [3:0] lane_mask,
    input [DATA_W-1:0] jx,
    input [DATA_W-1:0] jy,
    input [DATA_W-1:0] jm
  );
    begin
      @(negedge clk);
      drive_idle();
      i_grp_sel = grp_sel;
      i_lane_mask = lane_mask;
      i_j_x = jx;
      i_j_y = jy;
      i_j_m = jm;
      i_compute_en = 1'b1;
      @(posedge clk);
    end
  endtask

  task automatic drain_tile;
    begin
      @(negedge clk);
      drive_idle();
      i_grp_sel = 2'd0;

      for (int cyc = 0; cyc < PIPE_LAT + 4; cyc++) begin
        @(posedge clk);
      end

      for (int grp = 0; grp < N_GROUPS; grp++) begin
        @(negedge clk);
        drive_idle();
        i_grp_sel = grp[1:0];
        #1;
        if (o_res_vld !== 1'b1) begin
          $display("ERROR: o_res_vld not set for tile group %0d at t=%0t", grp, $time);
          err_count++;
        end
      end
    end
  endtask

```

```
  task automatic save_group(input int tile_base, input int grp);
    int base;
    begin
      base = tile_base + grp * GROUP_SIZE;
      if (base + 0 < N_BODIES) begin ax[base + 0] = o_res0_x; ay[base + 0] = o_res0_y; end
      if (base + 1 < N_BODIES) begin ax[base + 1] = o_res1_x; ay[base + 1] = o_res1_y; end
      if (base + 2 < N_BODIES) begin ax[base + 2] = o_res2_x; ay[base + 2] = o_res2_y; end
      if (base + 3 < N_BODIES) begin ax[base + 3] = o_res3_x; ay[base + 3] = o_res3_y; end
    end
  endtask

  task automatic run_tile(input int tile_base);
    begin
      load_tile(tile_base);
      repeat (2) @(posedge clk);
      pulse_clear_prev();

      for (int j = 0; j < N_BODIES; j++) begin
        for (int grp = 0; grp < N_GROUPS; grp++) begin
          issue_one_compute(grp[1:0],
                            make_lane_mask(tile_base, grp, j),
                            px[j], py[j], m[j]);
          cycles++;
          if (cycles > RUN_TIMEOUT_CYCLES) begin
            $fatal(1, "ERROR: timeout while streaming wrapper computes");
          end
        end
      end

      drain_tile();
      for (int grp = 0; grp < N_GROUPS; grp++) begin
        @(negedge clk);
        drive_idle();
        i_grp_sel = grp[1:0];
        #1;
        save_group(tile_base, grp);
      end
    end
  endtask

  task automatic write_output(input string fname);
    begin
      fo = $fopen(fname, "w");
      if (fo == 0) $fatal(1, "ERROR: cannot open OUT_FILE=%s", fname);

      $fwrite(fo, "# i ax ay (S1E8M18 hex)\n");
      for (int i = 0; i < N_BODIES; i++) begin
        $fwrite(fo, "%4d  %07h  %07h\n", i, ax[i], ay[i]);
      end

      $fclose(fo);
    end
  endtask

  initial begin
    rst_n = 1'b0;
    i_grp_sel = 2'd0;
    drive_idle();
    err_count = 0;
    cycles = 0;

    read_frame_file(INPUT_FILE);

    repeat (5) @(posedge clk);
    @(negedge clk);
    rst_n = 1'b1;
    repeat (5) @(posedge clk);

    for (int tile_base = 0; tile_base < N_BODIES; tile_base += TILE_SIZE) begin
      run_tile(tile_base);
    end

    write_output(OUT_FILE);
```

```
308
309     if (err_count == 0) begin
310       $display("PASS: tb_four_core_wrapper completed without errors");
311     end else begin
312       $display("FAIL: tb_four_core_wrapper saw %0d error(s)", err_count);
313     end
314     $display("DONE. cycles=%0d wrote %s", cycles, OUT_FILE);
315     $finish;
316   end
317
318 endmodule
```

A.4.3 tb_core_accel.sv

Listing 35: code/Hardware/tb/tb_core_accel.sv

```
1  `timescale 1ns/1ps
2
3  //-------------------------------------------------------------------------------
4  // TB: multi-body streaming scheduler driving a 2-body core
5  // - Reads frame0 file:  idx px py vx vy m   (all hex, S1E8M18 for px/py/vx/vy/m)
6  // - Streams all pairs (b1 accumulates contributions from every b2!=b1)
7  // - Core interface:
8  //     inputs: 27-bit S1E8M18 (pos/mass), prev accel: 27-bit S1E8M18
9  //     outputs: 27-bit S1E8M18
10 // - Core has fixed epsilon_square internally
11
12 module tb_core_accel;
13
14   // ----------------------------
15   // User knobs
16   // ----------------------------
17   localparam int N_BODIES  = 1024;   // <-- set any N you want
18   localparam int PIPE_LAT  = 18;
19   parameter  int PREV_LAT  = 16;
20
21   localparam string INPUT_FILE = "tb/frame_input/frame0_1024binit200_27bits.txt";
22   localparam string OUT_FILE_27bits = "tb/output/eps025_accel_27bits_1024binit200.txt";
23
24   logic        i_clk;
25   logic        i_rst;  // active-low reset
26
27   logic [26:0] i_b1_x, i_b1_y;
28   logic [26:0] i_b2_x, i_b2_y;
29   logic [26:0] i_m_b2;
30
31   wire  [26:0] i_a_b1_x, i_a_b1_y;       // 27-bit prev accel, late-aligned
32   wire  [26:0] o_a_b1_x, o_a_b1_y;       // 27-bit DUT accel out
33
34   two_body_core dut (
35     .i_clk    (i_clk),
36     .i_rst    (i_rst),
37
38     .i_b1_x   (i_b1_x),
39     .i_b1_y   (i_b1_y),
40     .i_b2_x   (i_b2_x),
41     .i_b2_y   (i_b2_y),
42     .i_m_b2   (i_m_b2),
43
44     .i_a_b1_x (i_a_b1_x),
45     .i_a_b1_y (i_a_b1_y),
46
47     .o_a_b1_x (o_a_b1_x),
48     .o_a_b1_y (o_a_b1_y)
49   );
50
51   // Clock
52   initial i_clk = 1'b0;
53   always #5 i_clk = ~i_clk; // 100MHz
```

```
54
55    // Frame storage
56    logic [26:0] px [0:N_BODIES-1];
57    logic [26:0] py [0:N_BODIES-1];
58    logic [26:0] m  [0:N_BODIES-1];
59
60
61
62    // Streaming control
63    logic in_valid;
64    logic [PIPE_LAT-1:0] vsh;
65    logic out_valid;
66    assign out_valid = vsh[PIPE_LAT-1];
67
68    logic [26:0] launch_prev_x_27, launch_prev_y_27;
69    logic [26:0] prev_x_pipe [0:PREV_LAT-1];
70    logic [26:0] prev_y_pipe [0:PREV_LAT-1];
71
72    always_ff @(posedge i_clk) begin
73      if (!i_rst) begin
74        vsh <= '0;
75        for (int p = 0; p < PREV_LAT; p++) begin
76          prev_x_pipe[p] <= 27'd0;
77          prev_y_pipe[p] <= 27'd0;
78        end
79      end else begin
80        vsh <= {vsh[PIPE_LAT-2:0], in_valid};
81
82        prev_x_pipe[0] <= in_valid ? launch_prev_x_27 : 27'd0;
83        prev_y_pipe[0] <= in_valid ? launch_prev_y_27 : 27'd0;
84        for (int p = 0; p < PREV_LAT-1; p++) begin
85          prev_x_pipe[p+1] <= prev_x_pipe[p];
86          prev_y_pipe[p+1] <= prev_y_pipe[p];
87        end
88      end
89    end
90
91    assign i_a_b1_x = prev_x_pipe[PREV_LAT-1];
92    assign i_a_b1_y = prev_y_pipe[PREV_LAT-1];
93
94    // Helper tasks for driving input and reading frame files
95    task automatic drive_idle();
96      begin
97        i_b1_x   <= 27'h1FC0000;
98        i_b1_y   <= 27'h0000000;
99        i_b2_x   <= 27'h2000000;
100       i_b2_y   <= 27'h0000000;
101       i_m_b2   <= 27'h1FC0000;
102
103       launch_prev_x_27 <= 27'd0;
104       launch_prev_y_27 <= 27'd0;
105
106       in_valid <= 1'b0;
107     end
108   endtask
109
110   // Helper task to drive one transaction (one pair of bodies, plus prev accel)
111   task automatic drive_txn(
112     input int b1,
113     input int b2,
114     input logic [26:0] a_prev_x_27,
115     input logic [26:0] a_prev_y_27
116   );
117     begin
118       i_b1_x   <= px[b1];
119       i_b1_y   <= py[b1];
120       i_b2_x   <= px[b2];
121       i_b2_y   <= py[b2];
122       i_m_b2   <= m[b2];
123
124       launch_prev_x_27 <= a_prev_x_27;
125       launch_prev_y_27 <= a_prev_y_27;
126
```

```
      in_valid <= 1'b1;
    end
  endtask

  // Read input frame file
  // line format: idx px py vx vy m (hex)
  task automatic read_frame_file(input string fname);
    int fd;
    string line;
    int idx;
    logic [26:0] lpx, lpy, lvx, lvy, lm;
    int got;
    begin
      for (int t = 0; t < N_BODIES; t++) begin
        px[t] = 27'h0000000;
        py[t] = 27'h0000000;
        m[t]  = 27'h0000000;
      end

      fd = $fopen(fname, "r");
      if (fd == 0) $fatal(1, "ERROR: cannot open INPUT_FILE=%s", fname);

      while (!$feof(fd)) begin
        line = "";
        void'($fgets(line, fd));

        if (line.len() == 0) continue;
        if (line.substr(0,0) == "#") continue;

        got = $sscanf(line, "%d %h %h %h %h %h",
                      idx, lpx, lpy, lvx, lvy, lm);

        if (got == 6 && idx >= 0 && idx < N_BODIES) begin
          px[idx] = lpx;
          py[idx] = lpy;
          m[idx]  = lm;
        end
      end
      $fclose(fd);
    end
  endtask

  // Auto-alignment queue (tracks which txn produced each output)
  typedef struct packed {
    int b1;
    int b2;
  } txn_t;

  txn_t q[$];

  // Scheduling state
  // We keep per-b1 running accel in 27-bit (as the core output is "prev+term")
  // busy[b1] prevents issuing next (b1, b2) until previous output for b1 returns.
  logic [26:0] a_acc_x_27 [0:N_BODIES-1];
  logic [26:0] a_acc_y_27 [0:N_BODIES-1];
  logic        busy       [0:N_BODIES-1];
  int          next_j     [0:N_BODIES-1];

  function automatic int pick_b1();
    int b;
    begin
      pick_b1 = -1;
      for (b = 0; b < N_BODIES; b++) begin
        if (!busy[b]) begin
          // skip self-pair
          while (next_j[b] < N_BODIES && next_j[b] == b) next_j[b]++;
          if (next_j[b] < N_BODIES) begin
            pick_b1 = b;
            return pick_b1;
          end
        end
      end
    end
```

```
  endfunction

  function automatic bit all_done();
    int b;
    begin
      all_done = 1'b1;
      for (b = 0; b < N_BODIES; b++) begin
        int tmp;
        tmp = next_j[b];
        while (tmp < N_BODIES && tmp == b) tmp++;
        if (tmp < N_BODIES) all_done = 1'b0;
        if (busy[b])        all_done = 1'b0;
      end
      if (q.size() != 0) all_done = 1'b0;
      if (vsh != '0)     all_done = 1'b0;
    end
  endfunction

  integer fo;
  integer b;
  integer b1;
  integer b2;

  initial begin
    drive_idle();

    // reset (active-low)
    i_rst = 1'b0;
    repeat (5) @(posedge i_clk);
    i_rst = 1'b1;
    repeat (5) @(posedge i_clk);

    read_frame_file(INPUT_FILE);

    // init per-body accumulators and sched
    for (b = 0; b < N_BODIES; b++) begin
      a_acc_x_27[b] = 27'd0;
      a_acc_y_27[b] = 27'd0;
      busy[b]       = 1'b0;
      next_j[b]     = 0;
    end

    // streaming loop
    while (!all_done()) begin
      @(posedge i_clk); #1;

      // 1) receive aligned output (27-bit), update per-b1 accumulator
      if (out_valid) begin
        txn_t t;
        if (q.size() == 0) $fatal(1, "out_valid but txn queue empty (alignment bug)");
        t = q.pop_front();

        // core returns prev+term already, so we overwrite accumulator
        a_acc_x_27[t.b1] = o_a_b1_x;
        a_acc_y_27[t.b1] = o_a_b1_y;

        busy[t.b1] = 1'b0;
      end

      // 2) issue one txn if possible
      b1 = pick_b1();
      if (b1 >= 0) begin
        b2 = next_j[b1];
        while (b2 < N_BODIES && b2 == b1) b2++;

        if (b2 < N_BODIES) begin
          drive_txn(b1, b2, a_acc_x_27[b1], a_acc_y_27[b1]);

          q.push_back('{b1:b1, b2:b2});

          busy[b1]   = 1'b1;
          next_j[b1] = b2 + 1;
        end else begin
```

```
273             drive_idle();
274           end
275         end else begin
276           drive_idle();
277         end
278       end
279
280       // write 27-bit accumulated output
281       fo = $fopen(OUT_FILE_27bits, "w");
282       if (fo == 0) $fatal(1, "ERROR: cannot open OUT_FILE_27bits=%s", OUT_FILE_27bits);
283       $fwrite(fo, "# i ax27 ay27 (S1E8M18 hex)\n");
284       for (b = 0; b < N_BODIES; b++) begin
285         $fwrite(fo, "%4d  %07h  %07h\n", b, a_acc_x_27[b], a_acc_y_27[b]);
286       end
287       $fclose(fo);
288
289       $display("DONE. Wrote %s", OUT_FILE_27bits);
290       $finish;
291     end
292
293   endmodule
```

A.4.4 tb_nbody_integrator.sv

Listing 36: code/Hardware/tb/tb_nbody_integrator.sv

```
1   `timescale 1ns/1ps
2
3   module tb_nbody_integrator;
4
5     localparam int DATA_W = 27;
6     localparam int SIGN_SHIFT = 26;
7
8     logic clk;
9     logic reset;
10    logic i_start;
11    logic o_done;
12
13    logic [DATA_W-1:0] i_x;
14    logic [DATA_W-1:0] i_y;
15    logic [DATA_W-1:0] i_vx;
16    logic [DATA_W-1:0] i_vy;
17    logic [DATA_W-1:0] i_ax;
18    logic [DATA_W-1:0] i_ay;
19
20    logic [DATA_W-1:0] o_x;
21    logic [DATA_W-1:0] o_y;
22    logic [DATA_W-1:0] o_vx;
23    logic [DATA_W-1:0] o_vy;
24
25    int err_count;
26
27    initial clk = 1'b0;
28    always #5 clk = ~clk;
29
30    nbody_integrator #(
31        .DATA_W(DATA_W)
32    ) dut (
33        .clk    (clk),
34        .reset  (reset),
35        .i_start(i_start),
36        .o_done (o_done),
37        .i_x    (i_x),
38        .i_y    (i_y),
39        .i_vx   (i_vx),
40        .i_vy   (i_vy),
41        .i_ax   (i_ax),
42        .i_ay   (i_ay),
43        .o_x    (o_x),
```

```
      .o_y    (o_y),
      .o_vx   (o_vx),
      .o_vy   (o_vy)
  );

  function automatic logic [17:0] rtl_frac18_from_u27(input logic [DATA_W-1:0] u);
    begin
      rtl_frac18_from_u27 = {1'b1, u[17:1]};
    end
  endfunction

  function automatic int lod37_shift_amt(input logic [36:0] x);
    begin
      lod37_shift_amt = 37;
      for (int bit_idx = 36; bit_idx >= 0; bit_idx--) begin
        if (x[bit_idx] && lod37_shift_amt == 37) begin
          lod37_shift_amt = 36 - bit_idx;
        end
      end
    end
  endfunction

  function automatic logic [DATA_W-1:0] fp27_add_model(
    input logic [DATA_W-1:0] a,
    input logic [DATA_W-1:0] b
  );
    logic sa, sb;
    int ea, eb;
    logic [17:0] af, bf;
    logic a_larger;
    int exp_diff_a, exp_diff_b;
    int larger_exp;
    logic [36:0] a_ext, b_ext;
    logic [36:0] a_shifted, b_shifted;
    logic [36:0] pre_sum;
    int shft_amt;
    logic [53:0] pre_frac_54;
    logic [53:0] pre_frac_shft;
    logic [53:0] uflow_shift;
    logic [17:0] osum_f;
    int osum_e;
    logic underflow;
    logic out_sign;
    begin
      sa = a[SIGN_SHIFT];
      sb = b[SIGN_SHIFT];
      ea = a[25:18];
      eb = b[25:18];
      af = rtl_frac18_from_u27(a);
      bf = rtl_frac18_from_u27(b);

      a_larger = ((ea > eb) || ((ea == eb) && (af > bf)));
      exp_diff_a = (eb - ea) & 8'hff;
      exp_diff_b = (ea - eb) & 8'hff;
      larger_exp = (eb > ea) ? eb : ea;

      a_ext = {1'b0, af, 18'b0};
      b_ext = {1'b0, bf, 18'b0};
      a_shifted = a_larger ? a_ext : ((exp_diff_a > 35) ? 37'd0 : (a_ext >> exp_diff_a));
      b_shifted = !a_larger ? b_ext : ((exp_diff_b > 35) ? 37'd0 : (b_ext >> exp_diff_b));

      if ((sa ^ sb) && a_larger) begin
        pre_sum = a_shifted - b_shifted;
      end else if ((sa ^ sb) && !a_larger) begin
        pre_sum = b_shifted - a_shifted;
      end else begin
        pre_sum = a_shifted + b_shifted;
      end

      shft_amt = lod37_shift_amt(pre_sum);
      pre_frac_54 = {pre_sum, 17'b0};
      pre_frac_shft = pre_frac_54 << (shft_amt + 1);
      uflow_shift = pre_frac_54 << shft_amt;
```

```
      osum_f = pre_frac_shft[53:36];
      osum_e = (larger_exp - shft_amt + 1) & 8'hff;
      underflow = ~uflow_shift[53];
      out_sign = a_larger ? sa : sb;

      if ((ea == 0) && (eb == 0)) begin
        fp27_add_model = '0;
      end else if (ea == 0) begin
        fp27_add_model = b;
      end else if (eb == 0) begin
        fp27_add_model = a;
      end else if (underflow) begin
        fp27_add_model = '0;
      end else if (pre_sum == 0) begin
        fp27_add_model = '0;
      end else begin
        fp27_add_model = {out_sign, osum_e[7:0], osum_f};
      end
    end
  endfunction

  task automatic clear_inputs;
    begin
      i_start = 1'b0;
      i_x = '0;
      i_y = '0;
      i_vx = '0;
      i_vy = '0;
      i_ax = '0;
      i_ay = '0;
    end
  endtask

  task automatic run_case(
    input string name,
    input logic [DATA_W-1:0] x,
    input logic [DATA_W-1:0] y,
    input logic [DATA_W-1:0] vx,
    input logic [DATA_W-1:0] vy,
    input logic [DATA_W-1:0] ax,
    input logic [DATA_W-1:0] ay
  );
    logic [DATA_W-1:0] exp_vx;
    logic [DATA_W-1:0] exp_vy;
    logic [DATA_W-1:0] exp_x;
    logic [DATA_W-1:0] exp_y;
    int wait_cycles;
    begin
      exp_vx = fp27_add_model(vx, ax);
      exp_vy = fp27_add_model(vy, ay);
      exp_x = fp27_add_model(x, exp_vx);
      exp_y = fp27_add_model(y, exp_vy);

      @(negedge clk);
      i_x = x;
      i_y = y;
      i_vx = vx;
      i_vy = vy;
      i_ax = ax;
      i_ay = ay;
      i_start = 1'b1;
      @(posedge clk);
      @(negedge clk);
      i_start = 1'b0;

      wait_cycles = 0;
      do begin
        @(posedge clk);
        #1;
        wait_cycles++;
      end while (o_done !== 1'b1 && wait_cycles < 20);

      if (o_done !== 1'b1) begin
```

```
        $display("ERROR %s timed out waiting for o_done", name);
        err_count++;
      end else if (wait_cycles != 9) begin
        $display("ERROR %s done latency got %0d expected 9", name, wait_cycles);
        err_count++;
      end

      if (o_vx !== exp_vx || o_vy !== exp_vy || o_x !== exp_x || o_y !== exp_y) begin
        $display("ERROR %s mismatch", name);
        $display("  got vx=%07h vy=%07h x=%07h y=%07h", o_vx, o_vy, o_x, o_y);
        $display("  exp vx=%07h vy=%07h x=%07h y=%07h", exp_vx, exp_vy, exp_x, exp_y);
        err_count++;
      end else begin
        $display("PASS %s vx=%07h vy=%07h x=%07h y=%07h", name, o_vx, o_vy, o_x, o_y);
      end

      @(posedge clk);
      #1;
      if (o_done !== 1'b0) begin
        $display("ERROR %s o_done did not deassert after one cycle", name);
        err_count++;
      end
    end
  endtask

  initial begin
    err_count = 0;
    reset = 1'b1;
    clear_inputs();

    repeat (3) @(posedge clk);
    #1;
    if (o_done !== 1'b0 || o_x !== '0 || o_y !== '0 || o_vx !== '0 || o_vy !== '0) begin
      $display("ERROR reset outputs not zero");
      err_count++;
    end

    @(negedge clk);
    reset = 1'b0;

    run_case("zero", 27'h0000000, 27'h0000000, 27'h0000000, 27'h0000000,
             27'h0000000, 27'h0000000);

    run_case("positive", 27'h3fc0000, 27'h4040000, 27'h3f80000, 27'h3f00000,
             27'h3e80000, 27'h3e00000);

    run_case("signed", 27'h4040000, 27'h6040000, 27'h3fc0000, 27'h5fc0000,
             27'h5fc0000, 27'h3fc0000);

    run_case("mantissa", 27'h3fc1234, 27'h4045678, 27'h3f8abcd, 27'h5f81234,
             27'h3e85555, 27'h3e8aaaa);

    if (err_count == 0) begin
      $display("PASS: tb_nbody_integrator completed without errors");
    end else begin
      $display("FAIL: tb_nbody_integrator saw %0d error(s)", err_count);
    end

    $finish;
  end

endmodule
```

A.4.5 tb_nbody_mem.sv

Listing 37: code/Hardware/tb/tb_nbody_mem.sv

```
`timescale 1ns/1ps

```

```
3  module tb_nbody_mem;
4
5    localparam int MAX_BODIES = 16;
6    localparam int DATA_W = 27;
7    localparam int PTR_W = 4;
8
9    logic clk;
10
11   logic             cpu_body_we;
12   logic [PTR_W-1:0] cpu_body_waddr;
13   logic [DATA_W-1:0] cpu_x;
14   logic [DATA_W-1:0] cpu_y;
15   logic [DATA_W-1:0] cpu_m;
16   logic [DATA_W-1:0] cpu_vx;
17   logic [DATA_W-1:0] cpu_vy;
18
19   logic [PTR_W-1:0] body_raddr;
20   logic [DATA_W-1:0] body_x;
21   logic [DATA_W-1:0] body_y;
22   logic [DATA_W-1:0] body_m;
23   logic [DATA_W-1:0] body_vx;
24   logic [DATA_W-1:0] body_vy;
25   logic [DATA_W-1:0] body_ax;
26   logic [DATA_W-1:0] body_ay;
27
28   logic             body_update_we;
29   logic [PTR_W-1:0] body_update_addr;
30   logic [DATA_W-1:0] body_update_x;
31   logic [DATA_W-1:0] body_update_y;
32   logic [DATA_W-1:0] body_update_vx;
33   logic [DATA_W-1:0] body_update_vy;
34
35   logic              accel_we;
36   logic [PTR_W-1:0]  accel_waddr;
37   logic [DATA_W-1:0] accel_ax;
38   logic [DATA_W-1:0] accel_ay;
39
40   int err_count;
41
42   initial clk = 1'b0;
43   always #5 clk = ~clk;
44
45   nbody_mem #(
46       .MAX_BODIES(MAX_BODIES),
47       .DATA_W(DATA_W),
48       .PTR_W(PTR_W)
49   ) dut (
50       .clk             (clk),
51       .cpu_body_we     (cpu_body_we),
52       .cpu_body_waddr  (cpu_body_waddr),
53       .cpu_x           (cpu_x),
54       .cpu_y           (cpu_y),
55       .cpu_m           (cpu_m),
56       .cpu_vx          (cpu_vx),
57       .cpu_vy          (cpu_vy),
58       .body_raddr      (body_raddr),
59       .body_x          (body_x),
60       .body_y          (body_y),
61       .body_m          (body_m),
62       .body_vx         (body_vx),
63       .body_vy         (body_vy),
64       .body_ax         (body_ax),
65       .body_ay         (body_ay),
66       .body_update_we  (body_update_we),
67       .body_update_addr(body_update_addr),
68       .body_update_x   (body_update_x),
69       .body_update_y   (body_update_y),
70       .body_update_vx  (body_update_vx),
71       .body_update_vy  (body_update_vy),
72       .accel_we        (accel_we),
73       .accel_waddr     (accel_waddr),
74       .accel_ax        (accel_ax),
75       .accel_ay        (accel_ay)
```

```
76    );
77
78    task automatic clear_inputs;
79      begin
80        cpu_body_we      = 1'b0;
81        cpu_body_waddr   = '0;
82        cpu_x            = '0;
83        cpu_y            = '0;
84        cpu_m            = '0;
85        cpu_vx           = '0;
86        cpu_vy           = '0;
87        body_raddr       = '0;
88        body_update_we   = 1'b0;
89        body_update_addr = '0;
90        body_update_x    = '0;
91        body_update_y    = '0;
92        body_update_vx   = '0;
93        body_update_vy   = '0;
94        accel_we         = 1'b0;
95        accel_waddr      = '0;
96        accel_ax         = '0;
97        accel_ay         = '0;
98      end
99    endtask
100
101   task automatic cpu_write_body(
102     input [PTR_W-1:0] addr,
103     input [DATA_W-1:0] x,
104     input [DATA_W-1:0] y,
105     input [DATA_W-1:0] mass,
106     input [DATA_W-1:0] vx,
107     input [DATA_W-1:0] vy
108   );
109     begin
110       @(negedge clk);
111       cpu_body_we    = 1'b1;
112       cpu_body_waddr = addr;
113       cpu_x          = x;
114       cpu_y          = y;
115       cpu_m          = mass;
116       cpu_vx         = vx;
117       cpu_vy         = vy;
118       @(posedge clk);
119       @(negedge clk);
120       cpu_body_we = 1'b0;
121     end
122   endtask
123
124   task automatic update_body(
125     input [PTR_W-1:0] addr,
126     input [DATA_W-1:0] x,
127     input [DATA_W-1:0] y,
128     input [DATA_W-1:0] vx,
129     input [DATA_W-1:0] vy
130   );
131     begin
132       @(negedge clk);
133       body_update_we   = 1'b1;
134       body_update_addr = addr;
135       body_update_x    = x;
136       body_update_y    = y;
137       body_update_vx   = vx;
138       body_update_vy   = vy;
139       @(posedge clk);
140       @(negedge clk);
141       body_update_we = 1'b0;
142     end
143   endtask
144
145   task automatic write_accel(
146     input [PTR_W-1:0] addr,
147     input [DATA_W-1:0] ax,
148     input [DATA_W-1:0] ay
```

```
  );
    begin
      @(negedge clk);
      accel_we    = 1'b1;
      accel_waddr = addr;
      accel_ax    = ax;
      accel_ay    = ay;
      @(posedge clk);
      @(negedge clk);
      accel_we = 1'b0;
    end
  endtask

  task automatic check_body(
    input [PTR_W-1:0] addr,
    input [DATA_W-1:0] exp_x,
    input [DATA_W-1:0] exp_y,
    input [DATA_W-1:0] exp_m,
    input [DATA_W-1:0] exp_vx,
    input [DATA_W-1:0] exp_vy,
    input [DATA_W-1:0] exp_ax,
    input [DATA_W-1:0] exp_ay
  );
    begin
      @(negedge clk);
      body_raddr = addr;
      @(posedge clk);
      #1;
      if (body_x !== exp_x || body_y !== exp_y || body_m !== exp_m ||
          body_vx !== exp_vx || body_vy !== exp_vy ||
          body_ax !== exp_ax || body_ay !== exp_ay) begin
        $display("ERROR addr=%0d got x=%07h y=%07h m=%07h vx=%07h vy=%07h ax=%07h ay=%07h",
                 addr, body_x, body_y, body_m, body_vx, body_vy, body_ax, body_ay);
        $display("              exp x=%07h y=%07h m=%07h vx=%07h vy=%07h ax=%07h ay=%07h",
                 exp_x, exp_y, exp_m, exp_vx, exp_vy, exp_ax, exp_ay);
        err_count++;
      end
    end
  endtask

  task automatic simultaneous_cpu_update_accel;
    begin
      @(negedge clk);
      cpu_body_we      = 1'b1;
      cpu_body_waddr   = 4'd7;
      cpu_x            = 27'h1111111;
      cpu_y            = 27'h1222222;
      cpu_m            = 27'h1333333;
      cpu_vx           = 27'h1444444;
      cpu_vy           = 27'h1555555;

      body_update_we   = 1'b1;
      body_update_addr = 4'd7;
      body_update_x    = 27'h2111111;
      body_update_y    = 27'h2222222;
      body_update_vx   = 27'h2444444;
      body_update_vy   = 27'h2555555;

      accel_we         = 1'b1;
      accel_waddr      = 4'd7;
      accel_ax         = 27'h2666666;
      accel_ay         = 27'h2777777;

      @(posedge clk);
      @(negedge clk);
      cpu_body_we    = 1'b0;
      body_update_we = 1'b0;
      accel_we       = 1'b0;
    end
  endtask

  initial begin
    err_count = 0;
```

```
    clear_inputs();
    repeat (2) @(posedge clk);

    cpu_write_body(4'd3, 27'h1010001, 27'h1020002, 27'h1030003, 27'h1040004, 27'h1050005);
    check_body(4'd3, 27'h1010001, 27'h1020002, 27'h1030003, 27'h1040004, 27'h1050005,
               27'h0000000, 27'h0000000);

    write_accel(4'd3, 27'h1a60006, 27'h1a70007);
    check_body(4'd3, 27'h1010001, 27'h1020002, 27'h1030003, 27'h1040004, 27'h1050005,
               27'h1a60006, 27'h1a70007);

    update_body(4'd3, 27'h2010001, 27'h2020002, 27'h2040004, 27'h2050005);
    check_body(4'd3, 27'h2010001, 27'h2020002, 27'h1030003, 27'h2040004, 27'h2050005,
               27'h1a60006, 27'h1a70007);

    cpu_write_body(4'd5, 27'h3010001, 27'h3020002, 27'h3030003, 27'h3040004, 27'h3050005);
    check_body(4'd5, 27'h3010001, 27'h3020002, 27'h3030003, 27'h3040004, 27'h3050005,
               27'h0000000, 27'h0000000);
    check_body(4'd3, 27'h2010001, 27'h2020002, 27'h1030003, 27'h2040004, 27'h2050005,
               27'h1a60006, 27'h1a70007);

    simultaneous_cpu_update_accel();
    check_body(4'd7, 27'h2111111, 27'h2222222, 27'h1333333, 27'h2444444, 27'h2555555,
               27'h2666666, 27'h2777777);

    if (err_count == 0) begin
      $display("PASS: tb_nbody_mem completed without errors");
    end else begin
      $display("FAIL: tb_nbody_mem saw %0d error(s)", err_count);
    end

    $finish;
  end

endmodule
```

A.4.6 tb_fpadd.sv

Listing 38: code/Hardware/tb/tb_fpadd.sv

```
`timescale 1ns/1ps

module tb_fpadd;

    reg         iCLK;
    reg  [26:0] iA;
    reg  [26:0] iB;
    wire [26:0] oSum;
    reg  [26:0] exp_out;

    integer fd_in;
    integer ret;
    integer case_idx;
    integer pass_cnt;
    integer fail_cnt;

    FpAdd dut (
        .iCLK(iCLK),
        .iA  (iA),
        .iB  (iB),
        .oSum(oSum)
    );

    initial begin
        iCLK = 0;
        forever #5 iCLK = ~iCLK;
    end

    initial begin
```

```
30          iA = 0;
31          iB = 0;
32          exp_out = 0;
33          case_idx = 0;
34          pass_cnt = 0;
35          fail_cnt = 0;
36
37          fd_in = $fopen("tb/frame_input/fpadd_cases.txt", "r");
38          if (fd_in == 0) begin
39              $display("ERROR: cannot open fpadd_cases.txt");
40              $finish;
41          end
42
43          @(negedge iCLK);
44
45          while (!$feof(fd_in)) begin
46              ret = $fscanf(fd_in, "%h %h %h\n", iA, iB, exp_out);
47
48              if (ret == 3) begin
49                  // FpAdd is two-stage pipelined, wait 2 cycles for output to settle
50                  @(posedge iCLK);
51                  @(posedge iCLK);
52                  #1;
53
54                  if (oSum === exp_out) begin
55                      pass_cnt = pass_cnt + 1;
56                      $display("[%0d] a=%07h b=%07h rtl=%07h py=%07h PASS",
57                               case_idx, iA, iB, oSum, exp_out);
58                  end
59                  else begin
60                      fail_cnt = fail_cnt + 1;
61                      $display("[%0d] a=%07h b=%07h rtl=%07h py=%07h FAIL",
62                               case_idx, iA, iB, oSum, exp_out);
63                      // Uncomment below to stop on the first failure
64                      // $stop;
65                  end
66
67                  case_idx = case_idx + 1;
68
69                  // Load next input group with a clean negedge clock edge
70                  @(negedge iCLK);
71              end
72              else begin
73                  ret = $fgetc(fd_in);
74              end
75          end
76
77          $display("========================================");
78          $display("tb_fpadd done");
79          $display("total = %0d, pass = %0d, fail = %0d",
80                   case_idx, pass_cnt, fail_cnt);
81          $display("========================================");
82
83          $fclose(fd_in);
84          $finish;
85      end
86
87  endmodule
```

A.4.7 tb_fpmul.sv

Listing 39: code/Hardware/tb/tb_fpmul.sv

```
1   `timescale 1ns/1ps
2
3   module tb_fpmul;
4
5       reg  [26:0] iA;
6       reg  [26:0] iB;
```

```
    wire [26:0] oProd;
    reg  [26:0] exp_out;

    integer fd_in;
    integer ret;
    integer case_idx;

    FpMul dut (
        .iA(iA),
        .iB(iB),
        .oProd(oProd)
    );

    initial begin
        iA = 0;
        iB = 0;
        exp_out = 0;

        fd_in = $fopen("tb/frame_input/fpmul_cases.txt", "r");
        if (fd_in == 0) begin
            $display("ERROR: cannot open fpmul_cases.txt");
            $finish;
        end

        case_idx = 0;

        while (!$feof(fd_in)) begin
            ret = $fscanf(fd_in, "%h %h %h\n", iA, iB, exp_out);
            $display("ret=%0d iA=%h iB=%h exp=%h", ret, iA, iB, exp_out);

            if (ret == 3) begin
                #1;
                $display("[%0d] rtl=%h py=%h %s",
                         case_idx, oProd, exp_out,
                         (oProd === exp_out) ? "PASS" : "FAIL");
                case_idx = case_idx + 1;
            end
            else begin
                // consume one bad line if needed
                ret = $fgetc(fd_in);
            end
        end

        $fclose(fd_in);
        $finish;
    end

endmodule
```

A.4.8 tb_fpinvsqrt.sv

Listing 40: code/Hardware/tb/tb_fpinvsqrt.sv

```
`timescale 1ns/1ps

module tb_fpinvsqrt;

    reg         iCLK;
    reg  [26:0] iA;
    wire [26:0] oInvSqrt;
    reg  [26:0] exp_out;

    integer fd_in;
    integer ret;
    integer case_idx;
    integer pass_cnt;
    integer fail_cnt;
    integer k;

```

```
    localparam integer SETTLE_CYCLES = 8;

    FpInvSqrt dut (
        .iCLK    (iCLK),
        .iA      (iA),
        .oInvSqrt(oInvSqrt)
    );

    initial begin
        iCLK = 1'b0;
        forever #5 iCLK = ~iCLK;
    end

    initial begin
        iA = 27'd0;
        exp_out = 27'd0;
        case_idx = 0;
        pass_cnt = 0;
        fail_cnt = 0;

        fd_in = $fopen("tb/frame_input/fpinvsqrt_cases.txt", "r");
        if (fd_in == 0) begin
            $display("ERROR: cannot open fpinvsqrt_cases.txt");
            $finish;
        end

        repeat (3) @(posedge iCLK); // let pipeline settle

        while (!$feof(fd_in)) begin
            ret = $fscanf(fd_in, "%h %h\n", iA, exp_out);

            if (ret == 2) begin
                // hold this input constant long enough
                for (k = 0; k < SETTLE_CYCLES; k = k + 1)
                    @(posedge iCLK);
                #1;

                if (oInvSqrt === exp_out) begin
                    pass_cnt = pass_cnt + 1;
                    $display("[%0d] x=%07h rtl=%07h py=%07h PASS",
                             case_idx, iA, oInvSqrt, exp_out);
                end
                else begin
                    fail_cnt = fail_cnt + 1;
                    $display("[%0d] x=%07h rtl=%07h py=%07h FAIL",
                             case_idx, iA, oInvSqrt, exp_out);

                    // first-fail stop is very useful here
                    $display("---- first fail detail ----");
                    $display("x      = %07h", iA);
                    $display("rtl    = %07h", oInvSqrt);
                    $display("python = %07h", exp_out);
                    $stop;
                end

                case_idx = case_idx + 1;
            end
            else begin
                ret = $fgetc(fd_in);
            end
        end

        $display("========================================");
        $display("tb_fpinvsqrt done");
        $display("total = %0d, pass = %0d, fail = %0d",
                 case_idx, pass_cnt, fail_cnt);
        $display("========================================");

        $fclose(fd_in);
        $finish;
    end

endmodule
```

A.5 Golden Model and Input Generation Code

A.5.1 golden.py

Listing 41: code/Golden/golden.py

```python
# ## 3. Python Model for FP modules

# ### 3.1 FP27 Add
# fp27_add_model.py
# Standalone RTL-mimic model for FpAdd (S1E8M18 core, 27-bit)

import random

# ============================================================
# format constants
# ============================================================
U27_MASK = (1 << 27) - 1
SIGN_SHIFT = 26
EXP_SHIFT  = 18
E_MASK     = 0xFF
M18_MASK   = (1 << 18) - 1


def u27_pack(sign: int, exp: int, mant: int) -> int:
    return ((sign & 1) << SIGN_SHIFT) | ((exp & E_MASK) << EXP_SHIFT) | (mant & M18_MASK)


def u27_fields(u: int):
    u &= U27_MASK
    s = (u >> SIGN_SHIFT) & 1
    e = (u >> EXP_SHIFT) & E_MASK
    m = u & M18_MASK
    return s, e, m


def u27_hex(u: int) -> str:
    return f"{u & U27_MASK:07X}"


def rtl_frac18_from_u27(u: int) -> int:
    """
    Match RTL:
      assign A_f = {1'b1, iA[17:1]};
    => internal 18-bit mantissa used by RTL add
    """
    return (1 << 17) | ((u >> 1) & 0x1FFFF)


def lod37_shift_amt(x: int) -> int:
    """
    Match RTL shft_amt:
      pre_frac[36] ? 0 :
      pre_frac[35] ? 1 :
      ...
      pre_frac[0]  ? 36 :
                     37;
    """
    x &= (1 << 37) - 1
    if x == 0:
        return 37
    for bit in range(36, -1, -1):
        if (x >> bit) & 1:
            return 36 - bit
    return 37


def fp27_add_rtl(a: int, b: int, debug: bool = False):
    """
    Bit-exact mimic of RTL FpAdd.
    Functionally collapses 2 pipeline stages into one Python call.
```

```
    """
    a &= U27_MASK
    b &= U27_MASK

    sa, ea, _ = u27_fields(a)
    sb, eb, _ = u27_fields(b)

    A_f = rtl_frac18_from_u27(a)  # 18-bit
    B_f = rtl_frac18_from_u27(b)  # 18-bit

    # ------------------------------------------------------------
    # stage 1 combinational
    # ------------------------------------------------------------
    A_larger = 1 if ((ea > eb) or ((ea == eb) and (A_f > B_f))) else 0

    exp_diff_A = (int(eb) - int(ea)) & 0xFF   # 9-bit-ish spirit
    exp_diff_B = (int(ea) - int(eb)) & 0xFF
    larger_exp = eb if (eb > ea) else ea

    # RTL:
    # {1'b0, A_f, 18'b0}  -> 37 bits
    A_ext = (A_f << 18) & ((1 << 37) - 1)
    B_ext = (B_f << 18) & ((1 << 37) - 1)

    if A_larger:
        A_f_shifted = A_ext
    else:
        A_f_shifted = 0 if exp_diff_A > 35 else (A_ext >> exp_diff_A)

    if not A_larger:
        B_f_shifted = B_ext
    else:
        B_f_shifted = 0 if exp_diff_B > 35 else (B_ext >> exp_diff_B)

    if ((sa ^ sb) & A_larger):
        pre_sum = (A_f_shifted - B_f_shifted) & ((1 << 37) - 1)
    elif ((sa ^ sb) & (1 - A_larger)):
        pre_sum = (B_f_shifted - A_f_shifted) & ((1 << 37) - 1)
    else:
        pre_sum = (A_f_shifted + B_f_shifted) & ((1 << 37) - 1)

    # ------------------------------------------------------------
    # stage 1 registers
    # ------------------------------------------------------------
    buf_pre_sum    = pre_sum
    buf_larger_exp = larger_exp
    buf_A_e_zero   = (ea == 0)
    buf_B_e_zero   = (eb == 0)
    buf_A          = a
    buf_B          = b
    buf_oSum_s     = sa if A_larger else sb

    # ------------------------------------------------------------
    # stage 2 combinational / output logic
    # ------------------------------------------------------------
    pre_frac = buf_pre_sum
    shft_amt = lod37_shift_amt(pre_frac)

    # RTL:
    # pre_frac_shft = {pre_frac,17'b0} << (shft_amt+1)
    # uflow_shift   = {pre_frac,17'b0} << (shft_amt)
    pre_frac_54  = (pre_frac << 17) & ((1 << 54) - 1)
    pre_frac_shft = (pre_frac_54 << (shft_amt + 1)) & ((1 << 54) - 1)
    uflow_shift   = (pre_frac_54 << shft_amt) & ((1 << 54) - 1)

    oSum_f = (pre_frac_shft >> 36) & M18_MASK
    oSum_e = (int(buf_larger_exp) - int(shft_amt) + 1) & 0xFF

    # RTL:
    # assign underflow = ~uflow_shift[53];
    underflow = 1 if (((uflow_shift >> 53) & 1) == 0) else 0

    if buf_A_e_zero and buf_B_e_zero:
```

```
        out = 0
    elif buf_A_e_zero:
        out = buf_B
    elif buf_B_e_zero:
        out = buf_A
    elif underflow:
        out = 0
    elif pre_frac == 0:
        out = 0
    else:
        out = u27_pack(buf_oSum_s, oSum_e, oSum_f)

    if debug:
        return {
            "a": a,
            "b": b,
            "sa": sa,
            "sb": sb,
            "ea": ea,
            "eb": eb,
            "A_f": A_f,
            "B_f": B_f,
            "A_larger": A_larger,
            "exp_diff_A": exp_diff_A,
            "exp_diff_B": exp_diff_B,
            "larger_exp": larger_exp,
            "A_f_shifted": A_f_shifted,
            "B_f_shifted": B_f_shifted,
            "pre_sum": pre_sum,
            "buf_pre_sum": buf_pre_sum,
            "pre_frac": pre_frac,
            "shft_amt": shft_amt,
            "pre_frac_54": pre_frac_54,
            "pre_frac_shft": pre_frac_shft,
            "uflow_shift": uflow_shift,
            "oSum_f": oSum_f,
            "oSum_e": oSum_e,
            "underflow": underflow,
            "buf_oSum_s": buf_oSum_s,
            "out": out,
            "out_hex": u27_hex(out),
        }
    return out


def make_case_file(path: str, n_random: int = 500):
    cases = []

    # ------------------------------------------------------------
    # directed cases
    # ------------------------------------------------------------
    directed = [
        # zeros
        (0x0000000, 0x0000000),
        (0x0000000, 0x3FC0000),
        (0x3FC0000, 0x0000000),

        # same-sign additions
        (0x3FC0000, 0x3FC0000),
        (0x4040000, 0x3FC0000),
        (0x4040000, 0x4040000),

        # sign-opposite / cancellation-ish
        (0x4440000, 0xC040000),
        (0xC040000, 0x4040000),
        (0x4040000, 0xC040000),
        (0x4040000, 0xC040001),  # test bit0-ish weirdness
        (0x3FC0000, 0xBFC0000),  # exact cancel
        (0x3FC0002, 0xBFC0000),  # near cancel

        # same exponent / different frac
        (0x3FC0001, 0x3FC0000),
        (0x3FC0002, 0x3FC0004),
```

```
        (0x7E7FFFF, 0x3FC0000),

        # large exponent diff
        (0x7C40000, 0x0440000),
        (0x0440000, 0x7C40000),

        # random looking hand-picked
        (0x11B31C6, 0x6AB377F),
        (0x390A3CF, 0x0F3EFAC),
        (0x4F2FC12, 0x0899422),
        (0x2C8993B, 0x4E9E669),
    ]
    cases.extend(directed)

    # ------------------------------------------------------------
    # random normals / zeros
    # ------------------------------------------------------------
    def rand_u27():
        s = random.getrandbits(1)
        e = 0 if random.random() < 0.08 else random.randint(1, 254)
        m = random.getrandbits(18)
        return u27_pack(s, e, m)

    for _ in range(n_random):
        a = rand_u27()
        b = rand_u27()
        cases.append((a, b))

    with open(path, "w") as f:
        for a, b in cases:
            y = fp27_add_rtl(a, b)
            f.write(f"{u27_hex(a)} {u27_hex(b)} {u27_hex(y)}\n")

    print(f"Wrote {len(cases)} cases to {path}")


# if __name__ == "__main__":
#     make_case_file("fpadd_cases.txt", n_random=500)


# ### 3.2 FP27 Mul
# fp27_mul_model.py
# Standalone RTL-mimic model for FpMul (S1E8M18 core, 27-bit)

import random

# ============================================================
# format constants
# ============================================================
U27_MASK = (1 << 27) - 1
SIGN_SHIFT = 26
EXP_SHIFT  = 18
E_MASK     = 0xFF
M18_MASK   = (1 << 18) - 1

# NOTE:
# RTL uses:
#   A_f = {1'b1, iA[17:1]}
# so bit0 is ignored by the multiplier datapath.


def u27_pack(sign: int, exp: int, mant: int) -> int:
    return ((sign & 1) << SIGN_SHIFT) | ((exp & E_MASK) << EXP_SHIFT) | (mant & M18_MASK)


def u27_fields(u: int):
    u &= U27_MASK
    s = (u >> SIGN_SHIFT) & 1
    e = (u >> EXP_SHIFT) & E_MASK
    m = u & M18_MASK
    return s, e, m
```

```
def u27_hex(u: int) -> str:
    return f"{u & U27_MASK:07X}"


def rtl_frac18_from_u27(u: int) -> int:
    """
    Match RTL:
      assign A_f = {1'b1, iA[17:1]};
    Effective 18-bit mantissa used internally:
      bit17 = hidden 1
      bit16:0 = stored bits [17:1]
    """
    return (1 << 17) | ((u >> 1) & 0x1FFFF)


def fp27_mul_rtl(a: int, b: int, debug: bool = False):
    """
    Bit-exact mimic of RTL FpMul:
      oProd_s = A_s ^ B_s
      pre_prod_frac = A_f * B_f
      pre_prod_exp  = A_e + B_e
      if pre_prod_frac[35]:
          oProd_e = pre_prod_exp - 126
          oProd_f = pre_prod_frac[34:17]
      else:
          oProd_e = pre_prod_exp - 127
          oProd_f = pre_prod_frac[33:16]
      underflow = pre_prod_exp < 128
      if underflow or A_e==0 or B_e==0: out=0
    """
    a &= U27_MASK
    b &= U27_MASK

    sa, ea, _ = u27_fields(a)
    sb, eb, _ = u27_fields(b)

    A_f = rtl_frac18_from_u27(a)
    B_f = rtl_frac18_from_u27(b)

    oProd_s = sa ^ sb
    pre_prod_frac = A_f * B_f    # 36-bit
    pre_prod_exp  = ea + eb      # 9-bit in RTL spirit

    if ((pre_prod_frac >> 35) & 1) == 1:
        oProd_e = (pre_prod_exp - 126) & 0xFF
        oProd_f = (pre_prod_frac >> 17) & M18_MASK   # [34:17]
        path = "hi"
    else:
        oProd_e = (pre_prod_exp - 127) & 0xFF
        oProd_f = (pre_prod_frac >> 16) & M18_MASK   # [33:16]
        path = "lo"

    underflow = pre_prod_exp < 0x80

    if underflow or (ea == 0) or (eb == 0):
        out = 0
    else:
        out = u27_pack(oProd_s, oProd_e, oProd_f)

    if debug:
        return {
            "a": a,
            "b": b,
            "sa": sa,
            "sb": sb,
            "ea": ea,
            "eb": eb,
            "A_f": A_f,
            "B_f": B_f,
            "pre_prod_frac": pre_prod_frac,
            "pre_prod_exp": pre_prod_exp,
            "oProd_s": oProd_s,
            "oProd_e": oProd_e,
```

```
            "oProd_f": oProd_f,
            "underflow": underflow,
            "path": path,
            "out": out,
            "out_hex": u27_hex(out),
        }
    return out


def make_case_file(path: str, n_random: int = 200):
    """
    Write:
      a_hex b_hex expected_hex
    """
    cases = []

    # -------- directed cases --------
    directed = [
        (0x0000000, 0x0000000),
        (0x0000000, 0x3FC0000),
        (0x3FC0000, 0x0000000),
        (0x3FC0000, 0x3FC0000),
        (0x4040000, 0x3FC0000),
        (0x4040000, 0x4040000),
        (0x43C0000, 0x3FC0000),
        (0x7E7FFFF, 0x3FC0000),
        (0x3FC0001, 0x3FC0001),  # test bit0 ignored or not
        (0x3FC0002, 0x3FC0002),
        (0x7C40000, 0x0440000),  # possible underflow-ish
        (0x4440000, 0xC040000),  # sign mix
        (0xC040000, 0xC040000),
    ]
    cases.extend(directed)

    # -------- random normals / zeros --------
    for _ in range(n_random):
        def rand_u27():
            s = random.getrandbits(1)
            # mostly normal exp, sometimes zero
            e = 0 if random.random() < 0.08 else random.randint(1, 254)
            m = random.getrandbits(18)
            return u27_pack(s, e, m)

        a = rand_u27()
        b = rand_u27()
        cases.append((a, b))

    with open(path, "w") as f:
        f.write("# a_hex b_hex expected_hex\n")
        for a, b in cases:
            y = fp27_mul_rtl(a, b)
            f.write(f"{u27_hex(a)} {u27_hex(b)} {u27_hex(y)}\n")

    print(f"Wrote {len(cases)} cases to {path}")


# if __name__ == "__main__":
#     make_case_file("fpmul_cases.txt", n_random=500)


# ### 3.3 FP27 InvSqrt
# fp27_invsqrt_model.py
import random

# ------------------------------------------------------------
# import your already-validated standalone models
# ------------------------------------------------------------
# from fp27_mul_model import fp27_mul_rtl
# from fp27_add_model import fp27_add_rtl

# ------------------------------------------------------------
# format constants
# ------------------------------------------------------------
```

```
U27_MASK    = (1 << 27) - 1
SIGN_SHIFT  = 26
EXP_SHIFT   = 18
E_MASK      = 0xFF
M18_MASK    = (1 << 18) - 1

# from your current model
MAGIC_C = 49920718   # 27'd49920718
ADD_K   = 33423360   # 27'd33423360


def u27_pack(sign: int, exp: int, mant: int) -> int:
    return ((sign & 1) << SIGN_SHIFT) | ((exp & E_MASK) << EXP_SHIFT) | (mant & M18_MASK)


def u27_fields(u: int):
    u &= U27_MASK
    s = (u >> SIGN_SHIFT) & 1
    e = (u >> EXP_SHIFT) & E_MASK
    m = u & M18_MASK
    return s, e, m


def u27_hex(u: int) -> str:
    return f"{u & U27_MASK:07X}"


def u27_neg(u: int) -> int:
    """Match FpNegate behavior for normal finite-style custom float."""
    u &= U27_MASK
    #if u == 0:
    #   return 0
    return u ^ (1 << SIGN_SHIFT)


def fast_inv_sqrt_u27_rtl(i_x: int, debug: bool = False) -> int:
    i_x &= U27_MASK
    w_sign, w_exp, w_man = u27_fields(i_x)

    # Stage 1
    w_s12_c = (MAGIC_C - (i_x >> 1)) & U27_MASK

    # IMPORTANT:
    # RTL does NOT flush exp==0 here.
    # It literally does {A_s, A_e-8'd1, A_f}, so exponent wraps.
    w_s12_b = u27_pack(w_sign, (w_exp - 1) & E_MASK, w_man)

    # Stage 1 mul
    w_s12_a = fp27_mul_rtl(w_s12_c, w_s12_c)

    # Stage 2 mul
    w_s23_a = fp27_mul_rtl(w_s12_b, w_s12_a)
    w_s23_b = w_s12_c

    # Stage 3 add: iA({~add_3_in[26], add_3_in[25:0]}), iB(ADD_K)
    w_s45_a = fp27_add_rtl(u27_neg(w_s23_a), ADD_K)
    w_s45_b = w_s23_b

    # Final mul
    out = fp27_mul_rtl(w_s45_b, w_s45_a)

    if debug:
        return {
            "i_x": i_x,
            "w_sign": w_sign,
            "w_exp": w_exp,
            "w_man": w_man,
            "w_s12_c": w_s12_c,
            "w_s12_b": w_s12_b,
            "w_s12_a": w_s12_a,
            "w_s23_a": w_s23_a,
            "w_s23_b": w_s23_b,
            "w_s45_a": w_s45_a,
```

```
            "w_s45_b": w_s45_b,
            "out": out,
            "out_hex": f"{out:07X}",
        }
    return out


def make_case_file(path: str, n_random: int = 500):
    cases = []

    # ------------------------------------------------------------
    # directed cases
    # ------------------------------------------------------------
    directed = [
        0x0000000,  # zero
        0x3FC0000,  # 1.0-ish
        0x4040000,  # 2.0-ish
        0x3C40000,  # 0.5-ish
        0x43C0000,  # 8.0-ish
        0x44C0000,
        0x7E7FFFF,  # large normal-ish
        0x0440000,  # tiny normal-ish
        0x3FC0001,  # bit0 perturb
        0x3FC0002,
        0x4040001,
        0x7C40000,
        0x0100000,
        0x1000000,
        0x2000000,
    ]
    for x in directed:
        cases.append(x)

    # ------------------------------------------------------------
    # random normals / zeros
    # ------------------------------------------------------------
    def rand_u27():
        s = 0  # invsqrt input should usually be non-negative r2/r2e
        e = 0 if random.random() < 0.08 else random.randint(1, 254)
        m = random.getrandbits(18)
        return u27_pack(s, e, m)

    for _ in range(n_random):
        cases.append(rand_u27())

    with open(path, "w") as f:
        for x in cases:
            y = fast_inv_sqrt_u27_rtl(x)
            f.write(f"{u27_hex(x)} {u27_hex(y)}\n")

    print(f"Wrote {len(cases)} cases to {path}")


# if __name__ == "__main__":
#     make_case_file("fpinvsqrt_cases.txt", n_random=500)



# ## 4. Full Hardware Model

# v4real_chiplike_f187_output.py
# IO:   27-bit custom float S1E8M18 (sign 1, exp 8, mant 18)
# CORE: 27-bit custom float S1E8M18 (sign 1, exp 8, mant 18)
#
# Outputs (N_FRAMES steps, frames numbered 0,1,2,...):
#   OUT_ROOT/
#     frame_chip_input/    : chip input frames (HEX S1E8M18). frame0.txt copied from INPUT_FILE
#     accel_chip_output/   : chip output accelerations (HEX S1E8M18). accel0.txt, accel1.txt, ...
#     accel_core27_output/ : internal accumulated accel (HEX S1E8M18). accel27_0.txt, accel27_1.txt, ...
#     frame_host_output/   : host frames (DECIMAL). frame0.txt, frame1.txt, ...
```

```
#
# Physics convention:
# - Chip accel is "chiplike" and MUST NOT include G0.
# - Host update uses:
#     K_HOST = G0 * DT   (only used in v-update)
#     v += a_chip * K_HOST
#     p += v * DT        (DT remains here; do NOT multiply G0 into this)
#
# IMPORTANT (BIT-EXACT CORE):
# - Core ops (add/mul/neg/fast_inv_sqrt) NEVER use float.
# - Core arithmetic is imported from standalone RTL-mimic models.
# - Host boundary quantization still uses TRUNC; denorm flush to 0.
# - Overflow clamps to max finite (exp=254, mant=all-1).

import os
import shutil
import numpy as np

# ============================================================
# IMPORT BIT-EXACT CORE (match standalone RTL-mimic modules)
# ============================================================
# The functions fp27_add_rtl, fp27_mul_rtl, and fast_inv_sqrt_u27_rtl
# are defined in earlier cells. We will assign them directly to aliases.
fp27_add = fp27_add_rtl
fp27_mul = fp27_mul_rtl
fast_inv_sqrt_u27 = fast_inv_sqrt_u27_rtl

# ============================================================
# Top-level knobs
# ============================================================
INPUT_FILE = "../Hardware/tb/frame_input/frame0_1024binit200_27bits.txt"
OUT_ROOT = "../Hardware/tb/output/eps025_v4real_chiplike_1024binit200"
# INPUT_FILE = "frame0_8binit10.txt"
# OUT_ROOT = "eps025_v4real_chiplike_8binit10"

DIR_FRAME_HOST   = os.path.join(OUT_ROOT, "frame_host_output")     # decimal
DIR_FRAME_CHIP   = os.path.join(OUT_ROOT, "frame_chip_input")      # hex (S1E8M18)
DIR_ACCEL_CHIP   = os.path.join(OUT_ROOT, "accel_chip_output")     # hex (S1E8M18)
DIR_ACCEL_CORE27 = os.path.join(OUT_ROOT, "accel_core27_output")   # hex (S1E8M18)

N_FRAMES = 4

# Host integrator settings
DT = 0.1
G0 = 10.0
K_HOST = G0 * DT

HOST_DTYPE = np.float32

# EPS^2 bit-exact constant in u27.
# Matches RTL: localparam logic [26:0] EPSILON_SQUARE = {1'b0, 8'd125, 18'd0};
# This encodes 2^(125 - 127) = 0.25, so epsilon = 0.5.
EPS_SQUARE_U27 = (0 << 26) | (125 << 18) | 0

# ============================================================
# Shared float layout params
# ============================================================
BIAS = 127
E_BITS = 8
E_MASK = (1 << E_BITS) - 1

# ============================================================
# CORE format: S1E8M18 (27-bit)
# ============================================================
M18_BITS = 18
U27_MASK = (1 << 27) - 1
SIGN27_SHIFT = 26
EXP27_SHIFT = M18_BITS
M18_MASK = (1 << M18_BITS) - 1

def u27_fields(u27: int):
    u27 &= U27_MASK
    s = (u27 >> SIGN27_SHIFT) & 1
```

```
    e = (u27 >> EXP27_SHIFT) & E_MASK
    m = u27 & M18_MASK
    return s, e, m

def u27_pack(s: int, e: int, m: int) -> int:
    return ((s & 1) << SIGN27_SHIFT) | ((e & E_MASK) << EXP27_SHIFT) | (m & M18_MASK)

def u27_neg(u27: int) -> int:
    # FIX: match RTL FpNegate exactly: {~A_s, A_e, A_f}
    # RTL unconditionally flips bit26, no zero guard.
    return (u27 ^ (1 << SIGN27_SHIFT)) & U27_MASK

def u27_to_hex(u27: int) -> str:
    return f"{u27 & U27_MASK:07X}"

def u27_from_hex(h: str) -> int:
    return int(h, 16) & U27_MASK

def u27_to_float(u27: int) -> float:
    """For host-side view only."""
    u27 &= U27_MASK
    if u27 == 0:
        return 0.0
    s, e, m = u27_fields(u27)
    if e == 0:
        return -0.0 if s else 0.0
    return (-1.0 if s else 1.0) * (2.0 ** (int(e) - BIAS)) * (1.0 + (m / (1 << M18_BITS)))

def u27_from_float_trunc(x: float) -> int:
    """TRUNC into S1E8M18. Denorm flush to 0. Overflow clamp."""
    if x == 0.0 or not np.isfinite(x):
        if x == 0.0:
            return 0
        sign = 1 if np.signbit(x) else 0
        return u27_pack(sign, 254, M18_MASK)

    sign = 1 if x < 0 else 0
    ax = abs(x)
    if ax == 0.0:
        return 0

    m, e = np.frexp(ax)
    m2 = m * 2.0
    e2 = e - 1
    exp = int(e2 + BIAS)

    if exp <= 0:
        return 0
    if exp >= 255:
        return u27_pack(sign, 254, M18_MASK)

    mant = int(np.floor((m2 - 1.0) * (1 << M18_BITS)))
    if mant < 0:
        mant = 0
    if mant > M18_MASK:
        mant = M18_MASK

    return u27_pack(sign, exp, mant)

# ============================================================
# File IO helpers
# ============================================================
def load_frame0_u27(path: str):
    idx, px, py, vx, vy, mm = [], [], [], [], [], []
    with open(path, "r") as f:
        for line in f:
            line = line.strip()
            if (not line) or line.startswith("#"):
                continue
            parts = line.split()
            idx.append(int(parts[0]))
            px.append(u27_from_hex(parts[1]))
            py.append(u27_from_hex(parts[2]))
```

```
            vx.append(u27_from_hex(parts[3]))
            vy.append(u27_from_hex(parts[4]))
            mm.append(u27_from_hex(parts[5]))
    return (
        np.array(idx, dtype=np.int32),
        np.array(px, dtype=np.uint32),
        np.array(py, dtype=np.uint32),
        np.array(vx, dtype=np.uint32),
        np.array(vy, dtype=np.uint32),
        np.array(mm, dtype=np.uint32),
    )

def write_frame_chip_hex(path: str, idx, px_u27, py_u27, vx_u27, vy_u27, m_u27):
    with open(path, "w") as f:
        f.write("# i  px  py  vx  vy  m   (S1E8M18 hex)\n")
        for i in range(len(idx)):
            f.write(
                f"{int(idx[i])}  {u27_to_hex(int(px_u27[i]))}  {u27_to_hex(int(py_u27[i]))}
 "
                f"{u27_to_hex(int(vx_u27[i]))}  {u27_to_hex(int(vy_u27[i]))}  {u27_to_hex(
int(m_u27[i]))}\n"
            )

def write_accel_core27_hex(path: str, ax_u27, ay_u27):
    with open(path, "w") as f:
        f.write("# i  ax27  ay27   (S1E8M18 hex)\n")
        for i in range(len(ax_u27)):
            f.write(f"{i}  {u27_to_hex(int(ax_u27[i]))}  {u27_to_hex(int(ay_u27[i]))}\n")

def write_frame_host_decimal(path: str, idx, px_f, py_f, vx_f, vy_f, m_f, g0: float, dt:
float, k_host: float):
    with open(path, "w") as f:
        f.write(f"# G0 = {g0:.12e}\n")
        f.write(f"# DT = {dt:.12e}\n")
        f.write(f"# K_HOST = G0*DT = {k_host:.12e}\n")
        f.write("# i  px  py  vx  vy  m   (decimal float)\n")
        for i in range(len(idx)):
            f.write(
                f"{int(idx[i])}  {float(px_f[i]): .12e}  {float(py_f[i]): .12e}  "
                f"{float(vx_f[i]): .12e}  {float(vy_f[i]): .12e}  {float(m_f[i]): .12e}\n"
            )

# ============================================================
# Core compute: chip hex -> internal u27 -> accel
# ============================================================
def compute_acc_once_u27io_accum27(px_u27, py_u27, m_u27, eps2_u27: int):
    """
    - input is already 27-bit S1E8M18
    - all ops in 27-bit bit-exact core
    - sum_ax/sum_ay accumulated in 27-bit
    """
    N = len(px_u27)
    ax_u27 = np.zeros(N, dtype=np.uint32)
    ay_u27 = np.zeros(N, dtype=np.uint32)

    px = [int(v) & U27_MASK for v in px_u27]
    py = [int(v) & U27_MASK for v in py_u27]
    mm = [int(v) & U27_MASK for v in m_u27]

    for i in range(N):
        sum_ax = 0
        sum_ay = 0

        for j in range(N):
            if j == i:
                continue

            dx = fp27_add(px[j], u27_neg(px[i]))
            dy = fp27_add(py[j], u27_neg(py[i]))

            dx2 = fp27_mul(dx, dx)
            dy2 = fp27_mul(dy, dy)
            r2  = fp27_add(dx2, dy2)
```

```
            r2e = fp27_add(r2, eps2_u27)

            # FIX: match RTL two_body_core.v pipeline mul order exactly:
            #   c12: s2 = s*s  AND  t = m2*s  (parallel)
            #   c13: k  = t*s2        (m2*s) * (s*s)
            # NOT the same as m2 * (s*s*s) due to FpMul bit0 truncation
            s  = fast_inv_sqrt_u27(r2e)
            s2 = fp27_mul(s, s)       # s
            t  = fp27_mul(mm[j], s)   #   m  s    (parallel with s2 in RTL)
            k  = fp27_mul(t, s2)      # (  m  s  )  s

            term_x = fp27_mul(k, dx)
            term_y = fp27_mul(k, dy)

            sum_ax = fp27_add(sum_ax, term_x)
            sum_ay = fp27_add(sum_ay, term_y)

        ax_u27[i] = sum_ax & U27_MASK
        ay_u27[i] = sum_ay & U27_MASK

    return ax_u27, ay_u27

# ============================================================
# Main loop
# ============================================================
def main():
    os.makedirs(DIR_FRAME_HOST, exist_ok=True)
    os.makedirs(DIR_FRAME_CHIP, exist_ok=True)
    os.makedirs(DIR_ACCEL_CHIP, exist_ok=True)
    os.makedirs(DIR_ACCEL_CORE27, exist_ok=True)

    print("Imported core modules:")
    print("  fp27_add         ->", fp27_add.__module__)
    print("  fp27_mul         ->", fp27_mul.__module__)
    print("  fast_inv_sqrt_u27->", fast_inv_sqrt_u27.__module__)

    eps2_u27 = EPS_SQUARE_U27

    # Load frame0 (27-bit hex)
    idx, px0_u27, py0_u27, vx0_u27, vy0_u27, m0_u27 = load_frame0_u27(INPUT_FILE)

    # Copy frame0 into frame_chip_input/frame0.txt
    frame0_chip_path = os.path.join(DIR_FRAME_CHIP, "frame0.txt")
    try:
        shutil.copyfile(INPUT_FILE, frame0_chip_path)
    except Exception:
        write_frame_chip_hex(frame0_chip_path, idx, px0_u27, py0_u27, vx0_u27, vy0_u27,
    m0_u27)

    # Host state (float) initialized from frame0 hex --- FP32
    px_f = np.array([u27_to_float(int(v)) for v in px0_u27], dtype=HOST_DTYPE)
    py_f = np.array([u27_to_float(int(v)) for v in py0_u27], dtype=HOST_DTYPE)
    vx_f = np.array([u27_to_float(int(v)) for v in vx0_u27], dtype=HOST_DTYPE)
    vy_f = np.array([u27_to_float(int(v)) for v in vy0_u27], dtype=HOST_DTYPE)
    m_f  = np.array([u27_to_float(int(v)) for v in m0_u27],  dtype=HOST_DTYPE)

    # Chip state (hex) initialized from frame0
    px_u27 = px0_u27.copy()
    py_u27 = py0_u27.copy()
    vx_u27 = vx0_u27.copy()
    vy_u27 = vy0_u27.copy()
    m_u27  = m0_u27.copy()

    # Dump host decimal frame0
    host0_path = os.path.join(DIR_FRAME_HOST, "frame0.txt")
    write_frame_host_decimal(host0_path, idx, px_f, py_f, vx_f, vy_f, m_f, G0, DT, K_HOST)

    # Produce frames 0..N_FRAMES, and accels 0..N_FRAMES-1
    for k in range(N_FRAMES):
        # 1) Chip computes accel for current frame k (NO G0 inside)
        ax_u27, ay_u27 = compute_acc_once_u27io_accum27(
            px_u27, py_u27, m_u27, eps2_u27
        )
```

```
862
863         accel_path = os.path.join(DIR_ACCEL_CHIP, f"accel{k}.txt")
864         write_accel_core27_hex(accel_path, ax_u27, ay_u27)
865
866         accel27_path = os.path.join(DIR_ACCEL_CORE27, f"accel27_{k}.txt")
867         write_accel_core27_hex(accel27_path, ax_u27, ay_u27)
868
869         # 2) Host update (FP32) using accel from chip (apply K_HOST only here)
870         ax_f = np.array([u27_to_float(int(v)) for v in ax_u27], dtype=HOST_DTYPE)
871         ay_f = np.array([u27_to_float(int(v)) for v in ay_u27], dtype=HOST_DTYPE)
872
873         k_host_f = HOST_DTYPE(K_HOST)
874         dt_f     = HOST_DTYPE(DT)
875
876         vx_f = (vx_f + ax_f * k_host_f).astype(HOST_DTYPE, copy=False)
877         vy_f = (vy_f + ay_f * k_host_f).astype(HOST_DTYPE, copy=False)
878         px_f = (px_f + vx_f * dt_f).astype(HOST_DTYPE, copy=False)
879         py_f = (py_f + vy_f * dt_f).astype(HOST_DTYPE, copy=False)
880
881         # 3) Quantize updated host frame back to 27-bit chip hex for next frame using TRUNC
882         px_u27 = np.array([u27_from_float_trunc(float(x)) for x in px_f], dtype=np.uint32)
883         py_u27 = np.array([u27_from_float_trunc(float(x)) for x in py_f], dtype=np.uint32)
884         vx_u27 = np.array([u27_from_float_trunc(float(x)) for x in vx_f], dtype=np.uint32)
885         vy_u27 = np.array([u27_from_float_trunc(float(x)) for x in vy_f], dtype=np.uint32)
886
887         # mass stays constant
888         m_u27 = m_u27
889
890         # 4) Write next chip input frame and host decimal frame
891         chip_path = os.path.join(DIR_FRAME_CHIP, f"frame{k+1}.txt")
892         write_frame_chip_hex(chip_path, idx, px_u27, py_u27, vx_u27, vy_u27, m_u27)
893
894         host_path = os.path.join(DIR_FRAME_HOST, f"frame{k+1}.txt")
895         write_frame_host_decimal(host_path, idx, px_f, py_f, vx_f, vy_f, m_f, G0, DT, K_HOST
    )
896
897     print(f"[OK] Done. OUT_ROOT={OUT_ROOT}")
898     print(f"     EPS_SQUARE_U27=0x{eps2_u27:07X}")
899     print(f"     G0={G0}  DT={DT}  K_HOST={K_HOST}")
900     print(f"     HOST_DTYPE={HOST_DTYPE}")
901     print(f"     Frames written: frame0..frame{N_FRAMES}")
902     print(f"     Accels written: accel0..accel{N_FRAMES-1}")
903     print(f"     Core27 written: accel27_0..accel27_{N_FRAMES-1}")
904
905 # if __name__ == "__main__":
906 #     main()
907
908
909 # ============================================================
910 # TB-compatible one-shot golden output
911 # Match tb_core_accel input/output behavior
912 # ============================================================
913
914 TB_INPUT_FILE = "../Hardware/tb/frame_input/frame0_1024binit200_27bits.txt"
915 TB_GOLDEN_OUT = "../Hardware/tb/output/golden_accel_1024binit200_27bits.txt"
916
917 def run_tb_compatible_golden():
918     idx, px_u27, py_u27, vx_u27, vy_u27, m_u27 = load_frame0_u27(TB_INPUT_FILE)
919
920     ax_u27, ay_u27 = compute_acc_once_u27io_accum27(
921         px_u27,
922         py_u27,
923         m_u27,
924         EPS_SQUARE_U27
925     )
926
927     with open(TB_GOLDEN_OUT, "w") as f:
928         f.write("# i ax ay (S1E8M18 hex)\n")
929         for i in range(len(idx)):
930             f.write(f"{int(idx[i]):4d}  {u27_to_hex(int(ax_u27[i]))}  {u27_to_hex(int(ay_u27
    [i]))}\n")
931
932     print(f"[OK] Wrote TB-compatible golden output: {TB_GOLDEN_OUT}")
```

```python

if __name__ == "__main__":
    run_tb_compatible_golden()


# ## 5. Compare
import os

# ============================================================
# USER CONFIG
# ============================================================
RTL_FILE = "../Hardware/tb/output/eps025_accel_1024binit200_27bits.txt"
PY_FILE  = "../Hardware/tb/output/golden_accel_1024binit200_27bits.txt"
# RTL_FILE = "eps025_temp_27bits_8binit10.txt"
# PY_FILE  = "eps025_v4real_chiplike_8binit10/accel_core27_output/accel27_0.txt"

# if you want, save mismatch report
REPORT_FILE = "compare_27bit_report.txt"

# ============================================================
# HELPERS
# ============================================================
U27_MASK = (1 << 27) - 1

def popcount(x: int) -> int:
    return bin(x).count("1")

def load_27bit_txt(path: str):
    """
    Expected format:
      # i  ax27  ay27 ...
      0  3ABCDE1  0123456
      1  1234567  7654321
    Returns:
      dict[idx] = (ax27_int, ay27_int)
    """
    data = {}
    with open(path, "r") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            parts = line.split()
            if len(parts) < 3:
                continue
            idx = int(parts[0])
            ax = int(parts[1], 16) & U27_MASK
            ay = int(parts[2], 16) & U27_MASK
            data[idx] = (ax, ay)
    return data

# ============================================================
# MAIN
# ============================================================
def main():
    rtl = load_27bit_txt(RTL_FILE)
    py  = load_27bit_txt(PY_FILE)

    rtl_keys = set(rtl.keys())
    py_keys  = set(py.keys())

    only_in_rtl = sorted(rtl_keys - py_keys)
    only_in_py  = sorted(py_keys - rtl_keys)
    common      = sorted(rtl_keys & py_keys)

    lines = []
    lines.append(f"RTL_FILE = {RTL_FILE}")
    lines.append(f"PY_FILE  = {PY_FILE}")
    lines.append("")

    if only_in_rtl:
        lines.append(f"indices only in RTL: {only_in_rtl}")
    if only_in_py:
```

```
        lines.append(f"indices only in PY : {only_in_py}")
    if only_in_rtl or only_in_py:
        lines.append("")

    mismatch_count = 0
    max_ax_bits = -1
    max_ay_bits = -1
    max_ax_idx = None
    max_ay_idx = None

    total_ax_bits = 0
    total_ay_bits = 0

    for i in common:
        rtl_ax, rtl_ay = rtl[i]
        py_ax,  py_ay  = py[i]

        diff_ax = rtl_ax ^ py_ax
        diff_ay = rtl_ay ^ py_ay

        bits_ax = popcount(diff_ax)
        bits_ay = popcount(diff_ay)

        total_ax_bits += bits_ax
        total_ay_bits += bits_ay

        if bits_ax > max_ax_bits:
            max_ax_bits = bits_ax
            max_ax_idx = i

        if bits_ay > max_ay_bits:
            max_ay_bits = bits_ay
            max_ay_idx = i

        if diff_ax != 0 or diff_ay != 0:
            mismatch_count += 1
            lines.append(
                f"{i:4d} | "
                f"{rtl_ax:07X},{rtl_ay:07X} -> {py_ax:07X},{py_ay:07X} | "
                f"{bits_ax},{bits_ay}"
            )

    lines.append("")
    lines.append("Summary")
    lines.append("-------")
    lines.append(f"common indices      : {len(common)}")
    lines.append(f"mismatch rows       : {mismatch_count}")
    lines.append(f"total ax bit diffs  : {total_ax_bits}")
    lines.append(f"total ay bit diffs  : {total_ay_bits}")
    lines.append(f"max ax bit diff     : {max_ax_bits}  (index {max_ax_idx})")
    lines.append(f"max ay bit diff     : {max_ay_bits}  (index {max_ay_idx})")

    report = "\n".join(lines)
    print(report)

    with open(REPORT_FILE, "w") as f:
        f.write(report)

    print(f"\nWrote report to: {REPORT_FILE}")

# if __name__ == "__main__":
#     main()
```

A.5.2 golden_control.py

Listing 42: code/Golden/golden_control.py

```
#!/usr/bin/env python3
"""
```

```
Golden model for nbody_control + four_core_wrapper + nbody_integrator.

Default output matches Hardware/tb/tb_nbody_control.sv:
    # i x y vx vy ax ay (S1E8M18 hex)
       0  0123456  0ABCDEF  0000000  0000000  0123000  0ABC000

The model reuses the bit-exact FP/core helpers from golden.py, then applies the
same control-level ordering: 16-body tiles, four 4-lane groups per tile, and
ascending lane writeback. It also updates the body memory with the RTL
integrator semantics so --gap and --first-step match nbody_control.
"""

import argparse
import os
import sys
from typing import Iterable, List, Optional, Tuple


SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
REPO_ROOT = os.path.abspath(os.path.join(SCRIPT_DIR, ".."))
if SCRIPT_DIR not in sys.path:
    sys.path.insert(0, SCRIPT_DIR)

import golden as core  # noqa: E402


U27_MASK = (1 << 27) - 1
SIGN_SHIFT = 26
EXP_SHIFT = 18
E_MASK = 0xFF
M18_MASK = (1 << 18) - 1
TILE_STRIDE = 16
GROUP_SIZE = 4
EPS_SQUARE_U27 = (0 << 26) | (125 << 18) | 0

DEFAULT_INPUT = os.path.join(
    REPO_ROOT, "Hardware", "tb", "frame_input", "frame0_1024binit200_27bits.txt"
)
DEFAULT_STATE_OUT = os.path.join(
    REPO_ROOT, "Hardware", "tb", "output", "golden_control_state_1024binit200_27bits.txt"
)


class BodyState:
    def __init__(
        self,
        x: List[int],
        y: List[int],
        vx: List[int],
        vy: List[int],
        m: List[int],
        ax: List[int],
        ay: List[int],
    ) -> None:
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.m = m
        self.ax = ax
        self.ay = ay


def u27_hex(value: int) -> str:
    return f"{value & U27_MASK:07X}"


def u27_from_hex(text: str) -> int:
    return int(text, 16) & U27_MASK


def u27_pack(sign: int, exp: int, mant: int) -> int:
    return ((sign & 1) << SIGN_SHIFT) | ((exp & E_MASK) << EXP_SHIFT) | (mant & M18_MASK)
```

```python


def u27_fields(value: int) -> Tuple[int, int, int]:
    value &= U27_MASK
    return (value >> SIGN_SHIFT) & 1, (value >> EXP_SHIFT) & E_MASK, value & M18_MASK


def half_step_accel(value: int) -> int:
    """Match nbody_control's exponent-only half-step acceleration adjustment."""
    sign, exp, mant = u27_fields(value)
    half_exp = exp - 1 if exp > 1 else 0
    return u27_pack(sign, half_exp, mant)


def integrator_step(
    x: int,
    y: int,
    vx: int,
    vy: int,
    ax: int,
    ay: int,
    half_step: bool = False,
) -> Tuple[int, int, int, int]:
    """Match nbody_integrator.sv: vx'=vx+ax, x'=x+vx'."""
    if half_step:
        ax = half_step_accel(ax)
        ay = half_step_accel(ay)

    vx_new = core.fp27_add_rtl(vx, ax)
    vy_new = core.fp27_add_rtl(vy, ay)
    x_new = core.fp27_add_rtl(x, vx_new)
    y_new = core.fp27_add_rtl(y, vy_new)
    return x_new, y_new, vx_new, vy_new


def load_frame(path: str, max_bodies: int) -> Tuple[BodyState, int]:
    x = [0] * max_bodies
    y = [0] * max_bodies
    vx = [0] * max_bodies
    vy = [0] * max_bodies
    m = [0] * max_bodies
    highest_idx = -1

    with open(path, "r") as f:
        for line_no, line in enumerate(f, start=1):
            line = line.strip()
            if not line or line.startswith("#"):
                continue

            parts = line.split()
            if len(parts) < 6:
                raise ValueError(f"{path}:{line_no}: expected 'idx x y vx vy m'")

            idx = int(parts[0])
            if idx < 0 or idx >= max_bodies:
                continue

            x[idx] = u27_from_hex(parts[1])
            y[idx] = u27_from_hex(parts[2])
            vx[idx] = u27_from_hex(parts[3])
            vy[idx] = u27_from_hex(parts[4])
            m[idx] = u27_from_hex(parts[5])
            highest_idx = max(highest_idx, idx)

    return BodyState(x=x, y=y, vx=vx, vy=vy, m=m, ax=[0] * max_bodies, ay=[0] * max_bodies),
     highest_idx + 1


def compute_one_body_accel(state: BodyState, i: int, active_count: int) -> Tuple[int, int]:
    """Bit-exact model of one accumulated four_core_wrapper output lane."""
    sum_ax = 0
    sum_ay = 0

```

```
    for j in range(active_count):
        if j == i:
            continue

        dx = core.fp27_add_rtl(state.x[j], core.u27_neg(state.x[i]))
        dy = core.fp27_add_rtl(state.y[j], core.u27_neg(state.y[i]))

        dx2 = core.fp27_mul_rtl(dx, dx)
        dy2 = core.fp27_mul_rtl(dy, dy)
        r2 = core.fp27_add_rtl(dx2, dy2)
        r2e = core.fp27_add_rtl(r2, EPS_SQUARE_U27)

        s = core.fast_inv_sqrt_u27_rtl(r2e)
        s2 = core.fp27_mul_rtl(s, s)
        t = core.fp27_mul_rtl(state.m[j], s)
        k = core.fp27_mul_rtl(t, s2)

        term_x = core.fp27_mul_rtl(k, dx)
        term_y = core.fp27_mul_rtl(k, dy)
        sum_ax = core.fp27_add_rtl(sum_ax, term_x)
        sum_ay = core.fp27_add_rtl(sum_ay, term_y)

    return sum_ax & U27_MASK, sum_ay & U27_MASK


def control_write_order(active_count: int) -> Iterable[int]:
    """Yield body indices in the same tile/group/lane order as nbody_control."""
    for tile_base in range(0, active_count, TILE_STRIDE):
        for group in range(TILE_STRIDE // GROUP_SIZE):
            group_base = tile_base + group * GROUP_SIZE
            for lane in range(GROUP_SIZE):
                idx = group_base + lane
                if idx < active_count:
                    yield idx


def compute_accel_pass(state: BodyState, active_count: int) -> List[Tuple[int, int, int]]:
    writes: List[Tuple[int, int, int]] = []

    for i in control_write_order(active_count):
        ax, ay = compute_one_body_accel(state, i, active_count)
        state.ax[i] = ax
        state.ay[i] = ay
        writes.append((i, ax, ay))

    return writes


def integrate_pass(state: BodyState, active_count: int, half_step: bool) -> None:
    for i in range(active_count):
        x, y, vx, vy = integrator_step(
            state.x[i],
            state.y[i],
            state.vx[i],
            state.vy[i],
            state.ax[i],
            state.ay[i],
            half_step=half_step,
        )
        state.x[i] = x
        state.y[i] = y
        state.vx[i] = vx
        state.vy[i] = vy


def run_control_model(
    state: BodyState,
    active_count: int,
    gap: int,
    first_step: bool,
) -> Tuple[List[Tuple[int, int, int]], BodyState]:
    """
    Run the control-level model.
```

```
221
222     Returns the first timestep's accel writes because tb_nbody_control.sv uses
223     gap=1 and writes one flat accel output file. For gap>1, later timesteps are
224     still integrated into state, but not appended to that flat file.
225     """
226     if gap <= 0 or active_count <= 0:
227         return [], state
228
229     first_writes: Optional[List[Tuple[int, int, int]]] = None
230     for timestep in range(gap):
231         writes = compute_accel_pass(state, active_count)
232         if first_writes is None:
233             first_writes = writes
234         integrate_pass(state, active_count, half_step=first_step and timestep == 0)
235
236     return first_writes or [], state
237
238
239 def write_accel_file(path: str, writes: Iterable[Tuple[int, int, int]]) -> None:
240     os.makedirs(os.path.dirname(os.path.abspath(path)), exist_ok=True)
241     with open(path, "w") as f:
242         f.write("# i ax ay (S1E8M18 hex)\n")
243         for idx, ax, ay in writes:
244             f.write(f"{idx:4d}  {u27_hex(ax)}  {u27_hex(ay)}\n")
245
246
247 def write_frame_file(path: str, state: BodyState, active_count: int) -> None:
248     os.makedirs(os.path.dirname(os.path.abspath(path)), exist_ok=True)
249     with open(path, "w") as f:
250         f.write("# i  px  py  vx  vy  m   (S1E8M18 hex)\n")
251         for i in range(active_count):
252             f.write(
253                 f"{i:4d}  {u27_hex(state.x[i])}  {u27_hex(state.y[i])}  "
254                 f"{u27_hex(state.vx[i])}  {u27_hex(state.vy[i])}  {u27_hex(state.m[i])}\n"
255             )
256
257
258 def write_state_file(path: str, state: BodyState, active_count: int) -> None:
259     os.makedirs(os.path.dirname(os.path.abspath(path)), exist_ok=True)
260     with open(path, "w") as f:
261         f.write("# i x y vx vy ax ay (S1E8M18 hex)\n")
262         for i in range(active_count):
263             f.write(
264                 f"{i:4d}  {u27_hex(state.x[i])}  {u27_hex(state.y[i])}  "
265                 f"{u27_hex(state.vx[i])}  {u27_hex(state.vy[i])}  "
266                 f"{u27_hex(state.ax[i])}  {u27_hex(state.ay[i])}\n"
267             )
268
269
270 def parse_args() -> argparse.Namespace:
271     parser = argparse.ArgumentParser(
272         description="Generate a golden output for nbody_control.sv."
273     )
274     parser.add_argument("--input", default=DEFAULT_INPUT, help="Input frame file.")
275     parser.add_argument(
276         "--output",
277         default=DEFAULT_STATE_OUT,
278         help="Output final x/y/vx/vy/ax/ay file matching tb_nbody_control.sv.",
279     )
280     parser.add_argument("--frame-output", default=None, help="Optional final updated frame
    output.")
281     parser.add_argument(
282         "--accel-output",
283         default=None,
284         help="Optional acceleration-only write stream file.",
285     )
286     parser.add_argument("--max-bodies", type=int, default=1024, help="Memory capacity.")
287     parser.add_argument("--n-bodies", type=int, default=None, help="Active body count.")
288     parser.add_argument("--gap", type=int, default=1, help="Number of timesteps to run.")
289     parser.add_argument(
290         "--first-step",
291         action="store_true",
292         help="Apply the initial leapfrog half-step kick on timestep 0.",
```

```
    )
    return parser.parse_args()


def main() -> None:
    args = parse_args()
    state, loaded_count = load_frame(args.input, args.max_bodies)
    active_count = args.n_bodies if args.n_bodies is not None else loaded_count
    active_count = max(0, min(active_count, args.max_bodies))

    writes, final_state = run_control_model(
        state=state,
        active_count=active_count,
        gap=args.gap,
        first_step=args.first_step,
    )
    write_state_file(args.output, final_state, active_count)

    if args.frame_output:
        write_frame_file(args.frame_output, final_state, active_count)

    if args.accel_output:
        write_accel_file(args.accel_output, writes)

    print(f"[OK] input        : {args.input}")
    print(f"[OK] n_bodies     : {active_count}")
    print(f"[OK] gap          : {args.gap}")
    print(f"[OK] first_step   : {int(args.first_step)}")
    print(f"[OK] state output : {args.output}")
    if args.frame_output:
        print(f"[OK] frame output : {args.frame_output}")
    if args.accel_output:
        print(f"[OK] accel output : {args.accel_output}")


if __name__ == "__main__":
    main()
```

A.5.3 golden_avmm_xy.py

Listing 43: code/Golden/golden_avmm_xy.py

```
#!/usr/bin/env python3
"""
Golden X/Y stream for the current nbody_accel_avmm hardware path.

This script takes the same six-column S1E8M18 frame input file that software
loads into the accelerator, runs the bit-level Python model of the current RTL,
and writes the X/Y positions that software reads from OUT_X/OUT_Y after DONE.

The default first-step behavior matches nbody_accel_avmm.sv: loading bodies via
VY_IN marks the next GO as the initial leapfrog half-step run.
"""

import argparse
import os
import sys
from typing import Optional


SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
REPO_ROOT = os.path.abspath(os.path.join(SCRIPT_DIR, ".."))
if SCRIPT_DIR not in sys.path:
    sys.path.insert(0, SCRIPT_DIR)

import golden_control as hw  # noqa: E402


DEFAULT_INPUT = os.path.join(
```

```
    REPO_ROOT, "Hardware", "tb", "frame_input", "frame0_1024binit200_27bits.txt"
)
DEFAULT_OUTPUT = os.path.join(
    REPO_ROOT, "Hardware", "tb", "output", "golden_xy_1024binit200_27bits.txt"
)


def checked_active_count(loaded_count: int, requested_count: Optional[int], max_bodies: int) -> int:
    if requested_count is None:
        active_count = loaded_count
    else:
        active_count = requested_count

    if active_count <= 0:
        raise ValueError(f"n_bodies must be positive, got {active_count}")
    if active_count > max_bodies:
        raise ValueError(f"n_bodies={active_count} exceeds max_bodies={max_bodies}")
    if active_count > loaded_count:
        raise ValueError(
            f"n_bodies={active_count} but input only contains rows through index {loaded_count - 1}"
        )

    return active_count


def write_xy_file(path: str, input_path: str, active_count: int, gap: int, first_step: bool,
                  state: hw.BodyState) -> None:
    out_dir = os.path.dirname(os.path.abspath(path))
    if out_dir:
        os.makedirs(out_dir, exist_ok=True)

    with open(path, "w") as f:
        f.write(f"# source {input_path}\n")
        f.write(f"# n_bodies {active_count}\n")
        f.write(f"# gap {gap}\n")
        f.write(f"# first_step {int(first_step)}\n")
        f.write("# i x y (S1E8M18 hex)\n")
        for i in range(active_count):
            f.write(f"{i:4d}  {hw.u27_hex(state.x[i])}  {hw.u27_hex(state.y[i])}\n")


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description=(
            "Generate the golden OUT_X/OUT_Y stream returned by current "
            "nbody_accel_avmm hardware after one GO/DONE run."
        )
    )
    parser.add_argument("--input", default=DEFAULT_INPUT, help="Input frame txt file.")
    parser.add_argument("--output", default=DEFAULT_OUTPUT, help="Output X/Y txt file.")
    parser.add_argument("--max-bodies", type=int, default=1024, help="Hardware memory capacity.")
    parser.add_argument("--n-bodies", type=int, default=None, help="Active body count.")
    parser.add_argument("--gap", type=int, default=1, help="Hardware GAP register value.")
    parser.add_argument(
        "--no-first-step",
        action="store_true",
        help="Model a later GO without reloading bodies. Default models GO after body load.",
    )
    parser.add_argument(
        "--frame-output",
        default=None,
        help="Optional full final frame output with x/y/vx/vy/m.",
    )
    parser.add_argument(
        "--accel-output",
        default=None,
        help="Optional acceleration write stream output, matching tb_nbody_control.sv.",
    )
    return parser.parse_args()
```

```

def main() -> None:
    args = parse_args()

    if args.max_bodies <= 0:
        raise ValueError(f"max_bodies must be positive, got {args.max_bodies}")
    if args.gap <= 0:
        raise ValueError(f"gap must be positive, got {args.gap}")

    state, loaded_count = hw.load_frame(args.input, args.max_bodies)
    active_count = checked_active_count(loaded_count, args.n_bodies, args.max_bodies)
    first_step = not args.no_first_step

    accel_writes, final_state = hw.run_control_model(
        state=state,
        active_count=active_count,
        gap=args.gap,
        first_step=first_step,
    )

    write_xy_file(
        path=args.output,
        input_path=args.input,
        active_count=active_count,
        gap=args.gap,
        first_step=first_step,
        state=final_state,
    )

    if args.frame_output:
        hw.write_frame_file(args.frame_output, final_state, active_count)

    if args.accel_output:
        hw.write_accel_file(args.accel_output, accel_writes)

    print(f"[OK] input        : {args.input}")
    print(f"[OK] n_bodies     : {active_count}")
    print(f"[OK] gap          : {args.gap}")
    print(f"[OK] first_step   : {int(first_step)}")
    print(f"[OK] xy output    : {args.output}")
    if args.frame_output:
        print(f"[OK] frame output : {args.frame_output}")
    if args.accel_output:
        print(f"[OK] accel output : {args.accel_output}")


if __name__ == "__main__":
    main()
```

A.5.4 FP64.py

Listing 44: code/Golden/FP64.py

```
# ## 2. FP64 Golden Reference


# fp64_reference_loop.py
# Pure float64 N-body reference with "chip-accel convention":
#   - accel_chip_output: DOES NOT include G0
#   - host update uses:  K_HOST = G0 * DT   (v += a_chip * K_HOST)
#
# Output structure same as chiplike version (3 folders)

import os
import shutil
import numpy as np

# ============================================================
```

```
# User knobs
# ============================================================
INPUT_FILE = "../Hardware/tb/frame_input/frame0_256binit200.txt"  # f187 hex input

OUT_ROOT = "./Hardware/tb/output/tempfp64_256binit200_outputs"
DIR_FRAME_HOST = os.path.join(OUT_ROOT, "frame_host_output")
DIR_FRAME_CHIP = os.path.join(OUT_ROOT, "frame_chip_input")
DIR_ACCEL_CHIP = os.path.join(OUT_ROOT, "accel_chip_output")

N_FRAMES = 10
DT = 0.1
G0 = 10.0

# IMPORTANT: this is epsilon^2 added to r^2, use very small value for closer-to-realistic accel (but not too small to cause underflow issues in fp64)
EPS_SQ = 2.0 ** (-27)

# Host update gain (this is where G0 goes)
K_HOST = G0 * DT

# ============================================================
# S1E8M7 helpers (same as before)
# ============================================================
BIAS = 127
E_BITS = 8
M_BITS = 7
TOTAL_BITS = 16
E_MASK = (1 << E_BITS) - 1
M_MASK = (1 << M_BITS) - 1
SIGN_SHIFT = TOTAL_BITS - 1
EXP_SHIFT = M_BITS
U16_MASK = (1 << TOTAL_BITS) - 1

def f16_pack(sign: int, exp: int, mant: int) -> int:
    return ((sign & 1) << SIGN_SHIFT) | ((exp & E_MASK) << EXP_SHIFT) | (mant & M_MASK)

def f16_to_hex(u16: int) -> str:
    return f"{u16 & U16_MASK:04X}"

def f16_from_hex(h: str) -> int:
    return int(h, 16) & U16_MASK

def f16_to_float(u16: int) -> float:
    if u16 == 0:
        return 0.0
    sign = (u16 >> SIGN_SHIFT) & 1
    exp  = (u16 >> EXP_SHIFT) & E_MASK
    mant = u16 & M_MASK
    if exp == 0:
        return 0.0
    val = (1.0 + mant / (1 << M_BITS)) * (2.0 ** (exp - BIAS))
    return -val if sign else val

def f16_from_float(x: float) -> int:
    if x == 0.0:
        return 0
    sign = 1 if x < 0 else 0
    ax = abs(x)

    m, e = np.frexp(ax)  # ax = m * 2^e, m in [0.5, 1)
    m2 = m * 2.0
    e2 = e - 1
    exp = e2 + BIAS

    if exp <= 0:
        return 0
    if exp >= 255:
        return f16_pack(sign, 254, M_MASK)

    frac = m2 - 1.0
    mant_f = frac * (1 << M_BITS)
    mant = int(np.floor(mant_f + 0.5))  # round-to-nearest

```

```
    if mant == (1 << M_BITS):
        mant = 0
        exp += 1
        if exp >= 255:
            exp = 254
            mant = M_MASK

    return f16_pack(sign, exp, mant)

# ============================================================
# IO helpers
# ============================================================
def load_frame0_hex(path: str):
    idx, px, py, vx, vy, mm = [], [], [], [], [], []
    with open(path, "r") as f:
        for line in f:
            if line.startswith("#") or not line.strip():
                continue
            parts = line.split()
            idx.append(int(parts[0]))
            px.append(f16_to_float(f16_from_hex(parts[1])))
            py.append(f16_to_float(f16_from_hex(parts[2])))
            vx.append(f16_to_float(f16_from_hex(parts[3])))
            vy.append(f16_to_float(f16_from_hex(parts[4])))
            mm.append(f16_to_float(f16_from_hex(parts[5])))
    return (np.array(idx, dtype=np.int32),
            np.array(px, dtype=np.float64),
            np.array(py, dtype=np.float64),
            np.array(vx, dtype=np.float64),
            np.array(vy, dtype=np.float64),
            np.array(mm, dtype=np.float64))

def write_frame_host(path, idx, px, py, vx, vy, m):
    with open(path, "w") as f:
        f.write("# i px py vx vy m (float64)\n")
        for i in range(len(idx)):
            f.write(f"{idx[i]} {px[i]: .12e} {py[i]: .12e} "
                    f"{vx[i]: .12e} {vy[i]: .12e} {m[i]: .12e}\n")

def write_frame_chip(path, idx, px, py, vx, vy, m):
    with open(path, "w") as f:
        f.write("# i px py vx vy m (S1E8M7 hex)\n")
        for i in range(len(idx)):
            f.write(f"{idx[i]} "
                    f"{f16_to_hex(f16_from_float(px[i]))} "
                    f"{f16_to_hex(f16_from_float(py[i]))} "
                    f"{f16_to_hex(f16_from_float(vx[i]))} "
                    f"{f16_to_hex(f16_from_float(vy[i]))} "
                    f"{f16_to_hex(f16_from_float(m[i]))}\n")

def write_accel_chip(path, idx, ax, ay):
    # accel is "chip convention" (NO G0), but written as S1E8M7 hex for chip-output compare
    with open(path, "w") as f:
        f.write("# i ax ay (S1E8M7 hex)   [chip accel: NO G0]\n")
        for i in range(len(idx)):
            f.write(f"{idx[i]} "
                    f"{f16_to_hex(f16_from_float(ax[i]))} "
                    f"{f16_to_hex(f16_from_float(ay[i]))}\n")

# ============================================================
# FP64 N-body compute (chip accel: NO G0)
# ============================================================
def compute_acc_chip(px, py, m):
    """
    Compute chip-style acceleration (NO G0):
      a_chip = sum_j m[j] * d / (r^2 + eps^2)^(3/2)
    """
    N = len(px)
    ax = np.zeros(N, dtype=np.float64)
    ay = np.zeros(N, dtype=np.float64)

    for i in range(N):
        axi = 0.0
```

```
        ayi = 0.0
        pxi = px[i]
        pyi = py[i]
        for j in range(N):
            if i == j:
                continue
            dx = px[j] - pxi
            dy = py[j] - pyi
            r2 = dx*dx + dy*dy + EPS_SQ
            inv3 = 1.0 / (r2 ** 1.5)
            factor = m[j] * inv3  # <-- NO G0 here
            axi += factor * dx
            ayi += factor * dy
        ax[i] = axi
        ay[i] = ayi
    return ax, ay

# ============================================================
# Main
# ============================================================
def main():
    os.makedirs(DIR_FRAME_HOST, exist_ok=True)
    os.makedirs(DIR_FRAME_CHIP, exist_ok=True)
    os.makedirs(DIR_ACCEL_CHIP, exist_ok=True)

    idx, px, py, vx, vy, m = load_frame0_hex(INPUT_FILE)

    # frame0 outputs
    write_frame_host(os.path.join(DIR_FRAME_HOST, "frame0.txt"),
                     idx, px, py, vx, vy, m)
    shutil.copyfile(INPUT_FILE,
                    os.path.join(DIR_FRAME_CHIP, "frame0.txt"))

    for k in range(N_FRAMES):
        # 1) chip accel (no G0)
        ax_chip, ay_chip = compute_acc_chip(px, py, m)

        # 2) write chip accel as hex (for compare with RTL chip output)
        write_accel_chip(os.path.join(DIR_ACCEL_CHIP, f"accel{k}.txt"),
                         idx, ax_chip, ay_chip)

        # 3) host update uses K_HOST = G0 * DT
        vx = vx + ax_chip * K_HOST
        vy = vy + ay_chip * K_HOST
        px = px + vx * DT
        py = py + vy * DT

        # 4) write next frame (float64 host) + chip-input hex view
        write_frame_host(os.path.join(DIR_FRAME_HOST, f"frame{k+1}.txt"),
                         idx, px, py, vx, vy, m)

        write_frame_chip(os.path.join(DIR_FRAME_CHIP, f"frame{k+1}.txt"),
                         idx, px, py, vx, vy, m)

    print("[OK] FP64 reference generated (chip-accel convention, K_HOST used).")
    print(f"     EPS_SQ={EPS_SQ}  DT={DT}  G0={G0}  K_HOST={K_HOST}")

if __name__ == "__main__":
    main()
```

A.5.5 input_gen_s1e8m18.py

Listing 45: code/Hardware/tb/frame_input/input_gen_s1e8m18.py

```
# gen_input_s1e8m18.py
# Output 1: HEX file (S1E8M18)
# Output 2: DEC file (quantized decimal)

import numpy as np
```

```
6
7  # =========================
8  # User knobs
9  # =========================
10 N_PARTICLE    = 1024
11 POS_MIN       = -200.0
12 POS_MAX       =  200.0
13 VEL_INIT_ZERO = True
14 VEL_MIN       = -0.1
15 VEL_MAX       =  0.1
16 MASS_MIN      =  0.2
17 MASS_MAX      =  1.0
18 RANDOM_SEED   = 0
19
20 OUTPUT_HEX = "frame0_1024binit200_27bits.txt"
21
22 # =========================
23 # S1E8M18 helpers
24 # =========================
25 BIAS = 127
26 E_BITS = 8
27 M_BITS = 18
28 TOTAL_BITS = 27
29 E_MASK = (1 << E_BITS) - 1
30 M_MASK = (1 << M_BITS) - 1
31 SIGN_SHIFT = TOTAL_BITS - 1
32 EXP_SHIFT = M_BITS
33 U27_MASK = (1 << TOTAL_BITS) - 1
34
35 def f27_pack(sign: int, exp: int, mant: int) -> int:
36     return ((sign & 1) << SIGN_SHIFT) | ((exp & E_MASK) << EXP_SHIFT) | (mant & M_MASK)
37
38 def f27_to_hex(u27: int) -> str:
39     return f"{u27 & U27_MASK:07X}"
40
41 def f27_to_float(u27: int) -> float:
42     if u27 == 0:
43         return 0.0
44     sign = (u27 >> SIGN_SHIFT) & 1
45     exp  = (u27 >> EXP_SHIFT) & E_MASK
46     mant = u27 & M_MASK
47     if exp == 0:
48         return 0.0
49     val = (1.0 + mant / (1 << M_BITS)) * (2.0 ** (exp - BIAS))
50     return -val if sign else val
51
52 def f27_from_float(x: float) -> int:
53     if x == 0.0:
54         return 0
55     sign = 1 if x < 0 else 0
56     ax = abs(x)
57
58     m, e = np.frexp(ax)
59     m2 = m * 2.0
60     e2 = e - 1
61     exp = e2 + BIAS
62
63     if exp <= 0:
64         return 0
65     if exp >= 255:
66         return f27_pack(sign, 254, M_MASK)
67
68     frac = m2 - 1.0
69     mant_f = frac * (1 << M_BITS)
70
71     mant = int(np.floor(mant_f + 0.5))
72     if mant == (1 << M_BITS):
73         mant = 0
74         exp += 1
75         if exp >= 255:
76             exp = 254
77             mant = M_MASK
78
```

```
    return f27_pack(sign, exp, mant)

# ==========================
# Main
# ==========================
def main():
    np.random.seed(RANDOM_SEED)

    pos = np.random.uniform(POS_MIN, POS_MAX, size=(N_PARTICLE, 2))
    if VEL_INIT_ZERO:
        vel = np.zeros((N_PARTICLE, 2), dtype=np.float64)
    else:
        vel = np.random.uniform(VEL_MIN, VEL_MAX, size=(N_PARTICLE, 2))

    mass = np.random.uniform(MASS_MIN, MASS_MAX, size=(N_PARTICLE,))

    with open(OUTPUT_HEX, "w") as f_hex:

        f_hex.write("# i  px  py  vx  vy  m   (S1E8M18 hex)\n")

        for i in range(N_PARTICLE):

            # original float
            px_f = float(pos[i, 0])
            py_f = float(pos[i, 1])
            vx_f = float(vel[i, 0])
            vy_f = float(vel[i, 1])
            m_f  = float(mass[i])

            # quantize
            px_u = f27_from_float(px_f)
            py_u = f27_from_float(py_f)
            vx_u = f27_from_float(vx_f)
            vy_u = f27_from_float(vy_f)
            m_u  = f27_from_float(m_f)

            # write HEX
            f_hex.write(
                f"{i:4d}  "
                f"{f27_to_hex(px_u)}  {f27_to_hex(py_u)}  "
                f"{f27_to_hex(vx_u)}  {f27_to_hex(vy_u)}  "
                f"{f27_to_hex(m_u)}\n"
            )


    print(f"[OK] Generated:")
    print(f"     {OUTPUT_HEX}")
    print(f"     N={N_PARTICLE}")

if __name__ == "__main__":
    main()
```