N-Body Accelerator

FPGA-Based Gravitational Simulation

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

CSEE 4840 · Spring 2026

Project Overview

Motivation:
N-body Problem: O(N^2) Time Complexity

$$\mathbf{a}_i \approx G \cdot \sum_{j \neq i} \frac{m_j \mathbf{r}_{ij}}{(\|\mathbf{r}_{ij}\|^2 + \epsilon^2)^{3/2}}$$

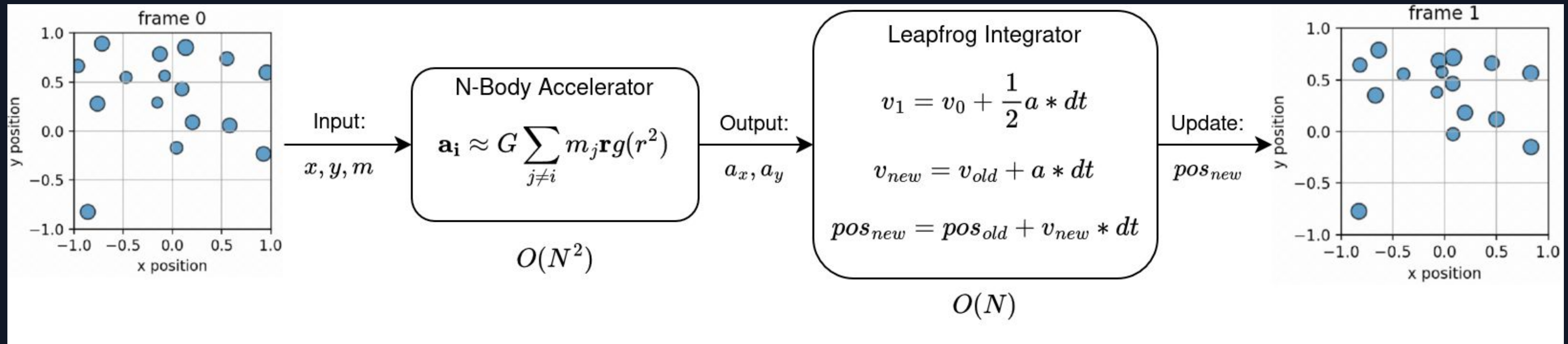
$$\mathbf{a}_i \approx G \sum_{j \neq i} m_j \, \mathbf{r}_{ij} \, g(r_{ij}^2)$$

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

<u>Goal:</u> Accelerate the compute of sum of pairwise interaction force

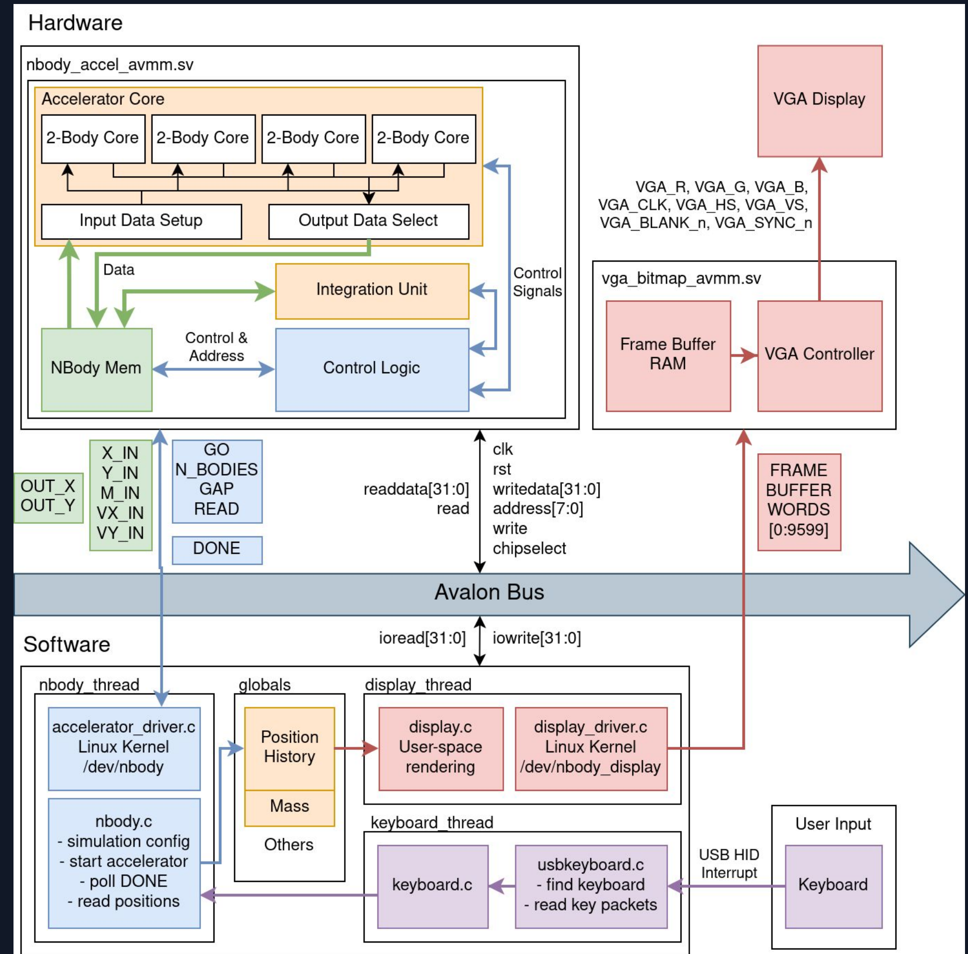
- Parallelism and Pipelining: 4 parallel fully pipelined 2-body core
- Custom 27-bit float (1 18×18 DSP/mult)
- Fast inverse square root (FISR)
- Leapfrog integration in hardware

System Demo:
Interactive 2D simulation sandbox with real-time display (60 Hz refresh rate)



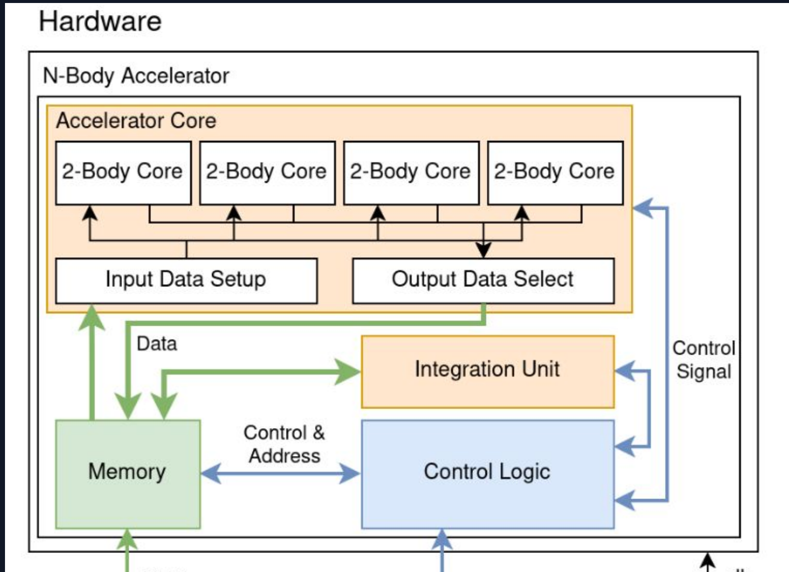
System Block Diagram

- **User-space software:** simulation setup, frame history, display rendering, and keyboard input.
- **Kernel drivers** expose FPGA hardware as Linux device files: /dev/nbody for the accelerator and /dev/nbody_display for VGA.
- **Avalon-MM bus:** connects software-controlled register reads/writes to the FPGA hardware modules.
- **N-body accelerator** receives particle data and control signals, computes updated body positions using parallel 2-body cores, and returns results to software.
- **Display pipeline** converts rendered framebuffer words from software into VGA signals through vga_bitmap_avmm.sv.
- **Keyboard input** allows runtime control such as pause/resume, frame navigation, gap adjustment, reset, and quit.



Hardware

Avalon Custom Peripheral: nbody_accel_avmm

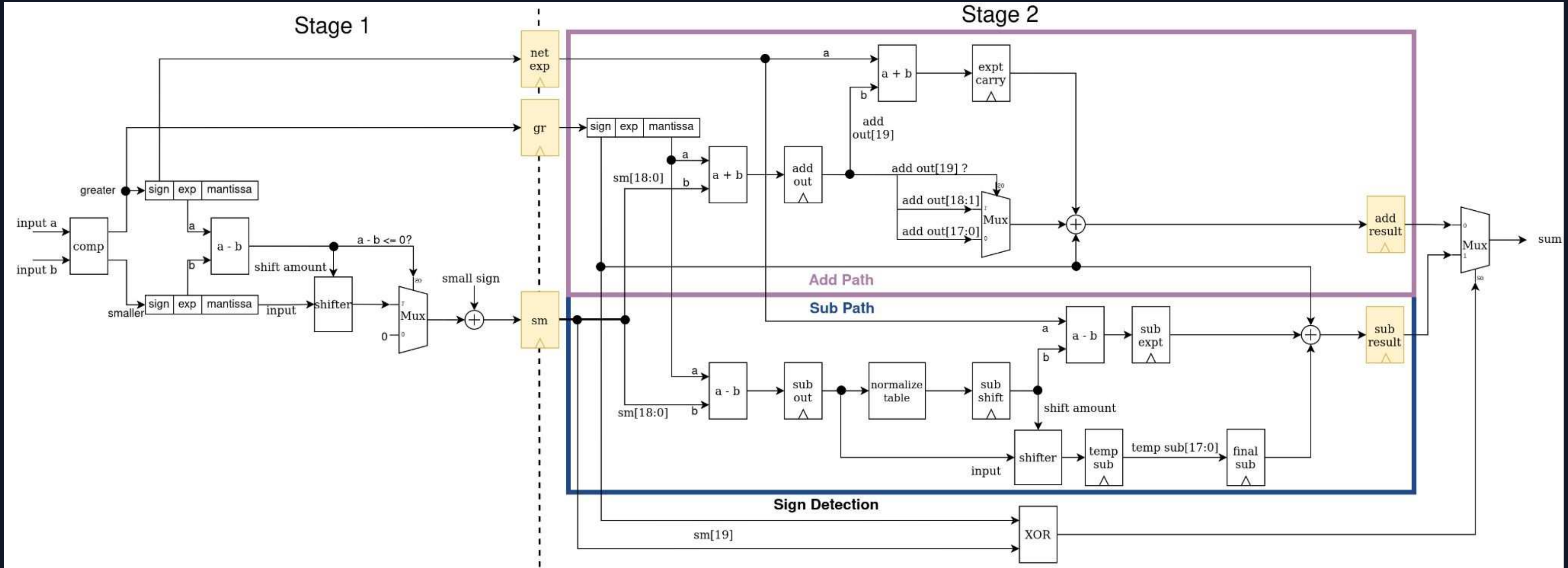


- FPGA accelerator composed of four parallel 2-body computation cores
- Central FSM controller manages scheduling, memory access, and datapath coordination
- Dedicated integration unit updates particle states after force accumulation
- Shared on-chip memory stores particle position, velocity, and mass data
- Input/output buffer logic connected between accelerator core and Avalon interface
- HPS software communicates with the accelerator through memory-mapped registers

Key Computation Building Blocks

- Floating Point Adder
- Floating Point Multiplier
- Fast Inverse Square Root
- Two Body Core
- N-body Memory

Floating Point Adder Module

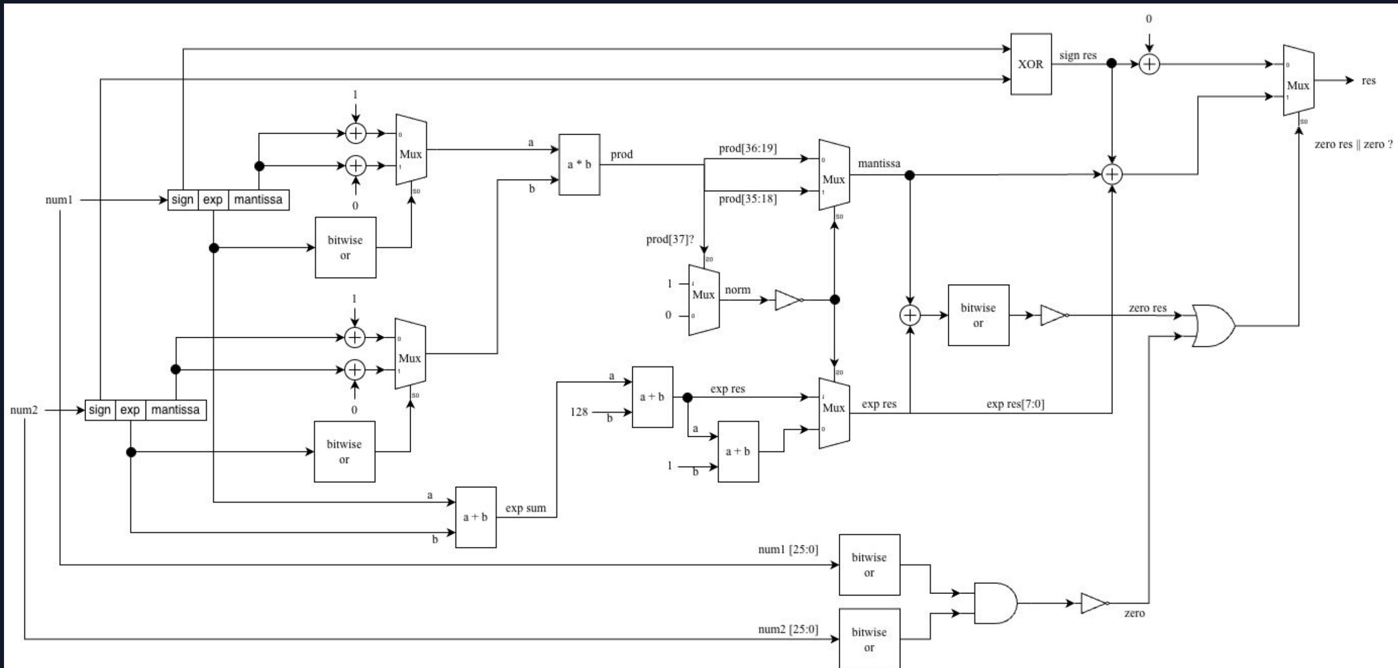


- Two-stage 27-bit custom floating-point adder
  [26] sign | [25:18] exponent (8b) | [17:0] mantissa (18b)
- Stage 1: Compare exponent and shift to align mantissas
- Stage 2: XOR the signs to decide add/sub the shifted mantissas, normalize, round, and pack

Test cases:

- Add zero
- Two inputs cancels
- Different sign
- Add commutativity
- Random tests with 500 vector inputs

Floating Point Multiplier Module



Test cases:

- Multiply by zero
- Different sign
- Multiply commutativity
- Max multiply by max
- Random tests with 500 vector inputs

- Combinational logic 27-bit custom floating-point multiplier
  [26] sign | [25:18] exponent (8b) | [17:0] mantissa (18b)
- XOR the sign bit, ADD exponents, and MULTIPLY mantissas

Fast Inverse Square Root (FP32)

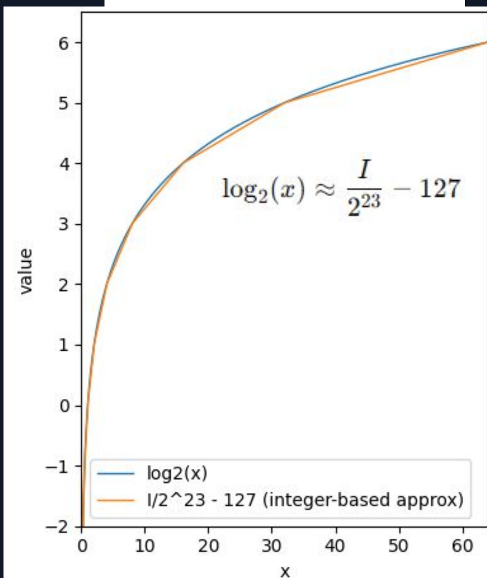
Goal: Compute y≈1/sqrt(x) fast

1. Integer interpretation of a float

S	E	M

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

Integer $I = S \cdot 2^{31} + E \cdot 2^{23} + M$



2. Magic number

$$\log_2(y) \approx -\frac{1}{2}\log_2(x) \approx -\frac{1}{2}\left[\left(\frac{I}{2^{23}} - 127\right) + \sigma\right]$$

$$I_y \approx -\frac{1}{2}I + \frac{3}{2} \cdot 127 \cdot 2^{23} - \frac{1}{2}\sigma\, 2^{23}$$

$$I_y \approx c_0 - \frac{I}{2}$$

Magic Number: $c_0(\sigma) = \frac{3}{2} \cdot 127 \cdot 2^{23} - \frac{1}{2}\sigma\, 2^{23}$

```
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);   // Newton
    return y;
}
```

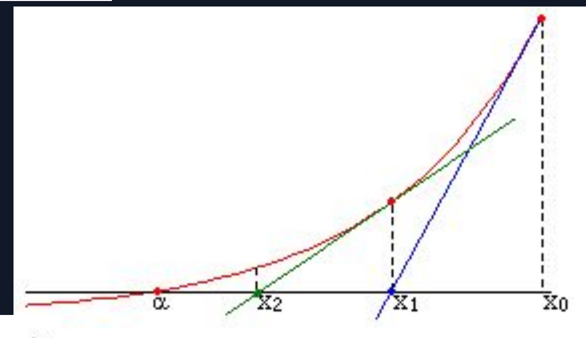
3. Newton's method: Solve f(y) = 0

Update Rule: $y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$

For **y≈1/sqrt(x)**

$f(y) = \frac{1}{y^2} - x$ $f'(y) = -\frac{2}{y^3}$

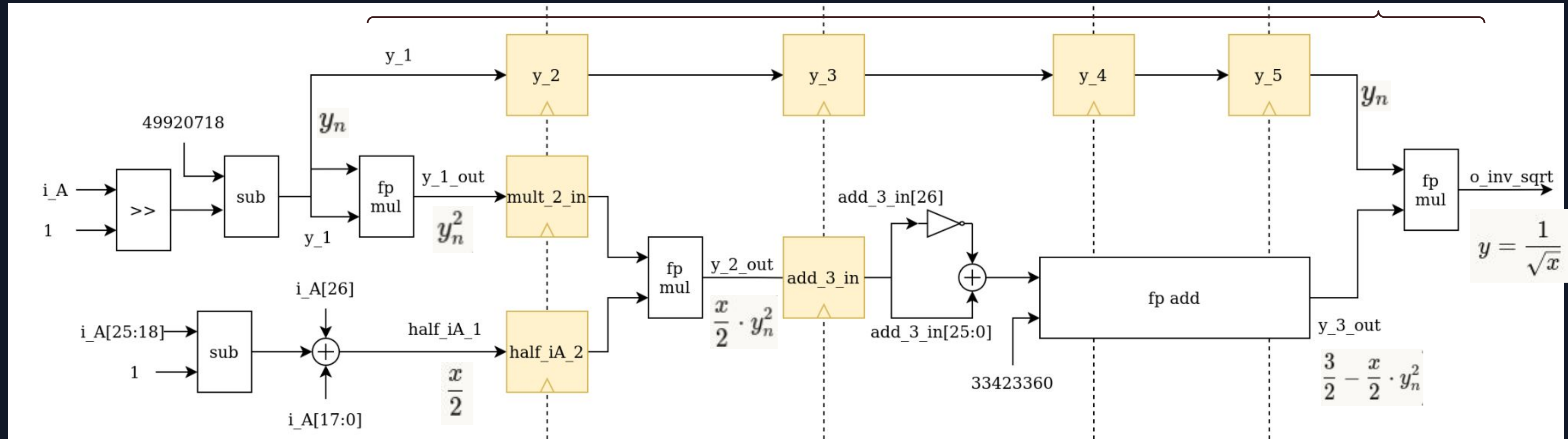
$$y_{n+1} = y_n - \frac{1/y_n^2 - x}{-2/y_n^3} = y_n\left(1.5 - 0.5\, x\, y_n^2\right)$$



Done in only simple addition and multiplication operations!

Floating Point Fast Inverse Square Root Module

$$y_{n+1} = y_n \cdot \left(\frac{3}{2} - \frac{x}{2} \cdot y_n^2\right)$$




```
float Q_rsqrt(float x){
    float x2 = 0.5f * x;
    int i = *(int*)&x;
    i = 49920718 - (i >> 1);
    float y = *(float*)&i;
    y = y * (1.5f - x2 * y * y);    // Newton
    return y;
}
```

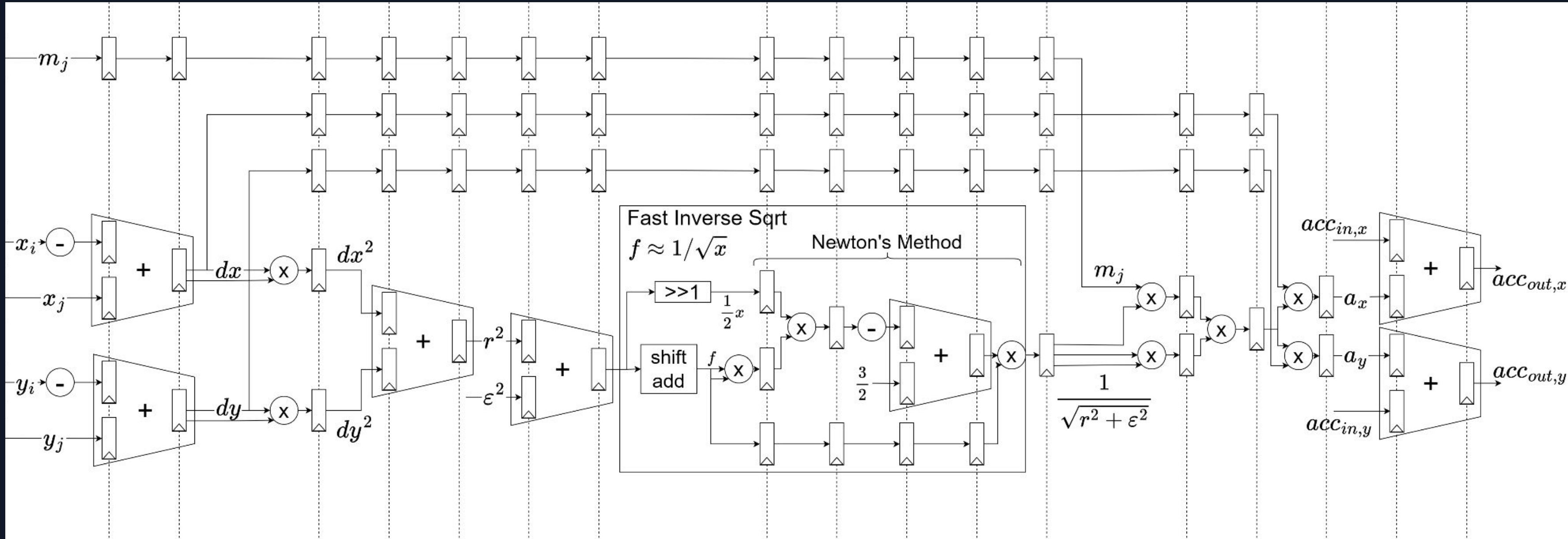
- 5 stage fast inverse square root with Quake III algorithm
- Stage 1 compute initial approximation y_0 through "magic number" bit hack
- Stage 2-5 applies one Newton-Raphson iteration to improve approximation error

Test cases:

- Tolerance test
- Output stability
- Pipeline throughput
- Random tests with 500 vector inputs

Pipelined Two-Body-Core

- 17-stage pipeline
- Two 27-bit Accumulators
- 16-bit floating point inputs → zero-padding → 27-bit floating point intermediates → truncated 16-bit outputs
- Latency = 18 cycles, Throughput = 1 pair/cycle



N-body Memory

- The main on-chip memory module storing body state: `x`, `y`, `mass`, `vx`, `vy`, `ax`, and `ay`.
- Each field uses a separate M10K-backed memory array for independent access and updates.
- The CPU write port loads complete body data and initializes `ax/ay` to zero.
- The synchronous read port returns body data one cycle after `body_raddr` is provided.
- The integrator writeback updates only `x/y/vx/vy`.
- The acceleration writeback updates only `ax/ay`.
- Write-selection logic handles concurrent updates from different pipeline stages.

```
(* 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];
```

Test cases:

- Initial body-data write/read
- Acceleration update
- Integrator position/velocity update
- Mass preservation
- Independent address access
- Simultaneous write priority handling

Dataflow

Constraint:

- **Costly Storage and Update** for partial sums value for all N bodies.
- **2-stage Accumulator**: the **same** i-body cannot be accumulated in consecutive cycles in pipeline.

Solution:

- Keep partial sums value for only one 16-*i* tile locally.
- Broadcast one *j*-body to all 4 cores and reuse it for 4 cycles, while sweeping across 16 *i*-bodies.

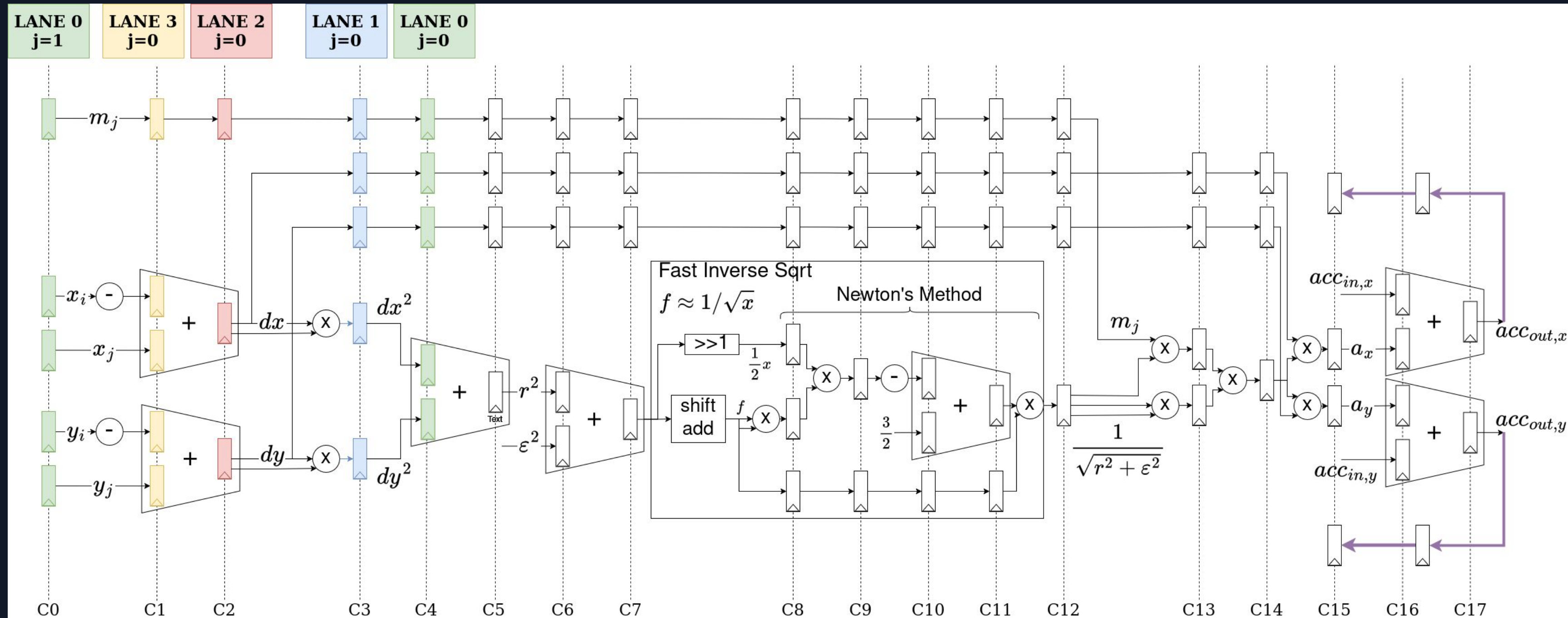
Dataflow Overview:

1. Load one local tile: 16 *i*-bodies.
2. Stream one *j*-body from input over 3 cycles: *x*, *y*, *m*, and broadcast to all 4 cores.
3. Hold *j*-body over 4 cycles and sweep *i_grp_sel* = 0, 1, 2, 3.
4. Repeat for all *j*-bodies, then read out the final accumulated accelerations.
5. Read out final accumulated accelerations of this tile.
6. Reload the next tile and repeat until all N bodies are processed.

cycle	new body	core0		core1		core2		core3	
0	0j	0	0	1	0	2	0	3	0
1		4	0	5	0	6	0	7	0
2		8	0	9	0	10	0	11	0
3		12	0	13	0	14	0	15	0
4	1j	0	1	1	1	2	1	3	1
5		4	1	5	1	6	1	7	1
6		8	1	9	1	10	1	11	1
7		12	1	13	1	14	1	15	1
8	2j	0	2	1	2	2	2	3	2
9		4	2	5	2	6	2	7	2
10		8	2	9	2	10	2	11	2
11		12	2	13	2	14	2	15	2

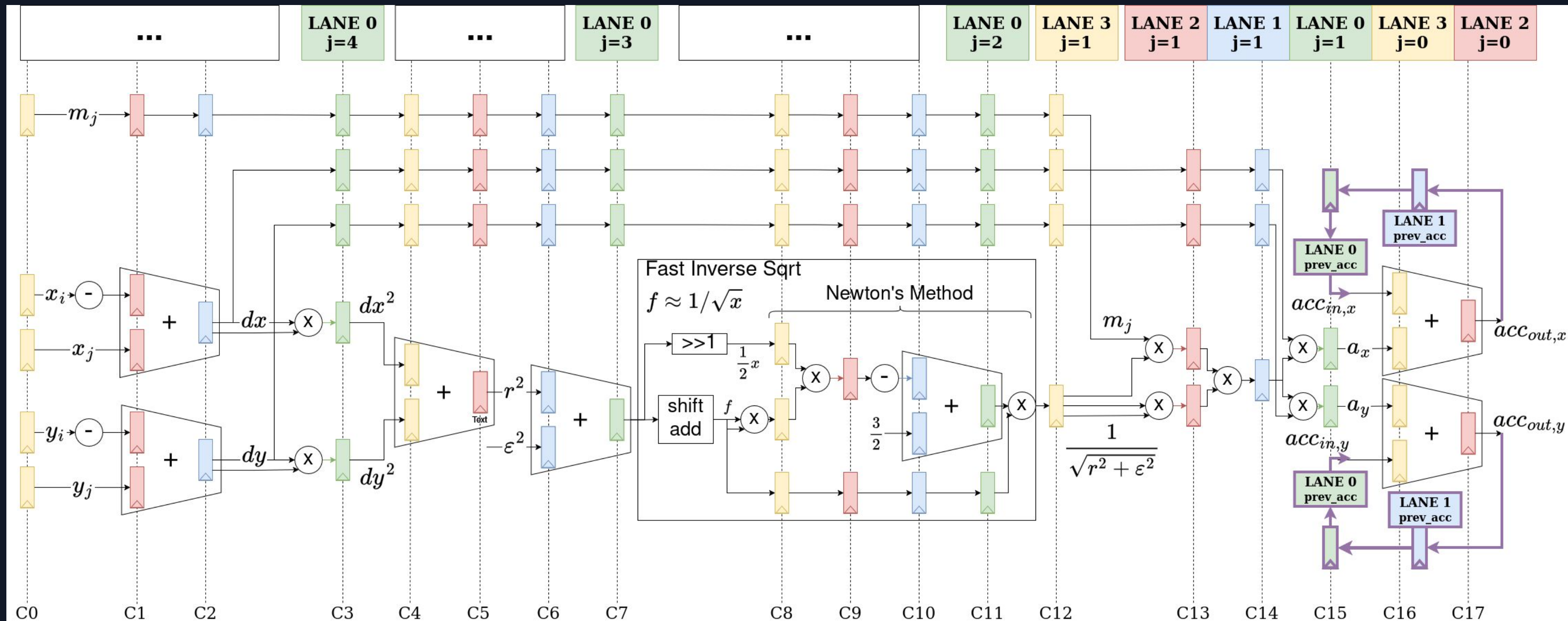
Two-Body-Core Self Accumulation

- Input: 2-stage adder for final accumulating stage → same *i* should not be calculated for next cycle.
- Fixed 2-reg feedback chain
- Example: CORE0 - Cycle 4

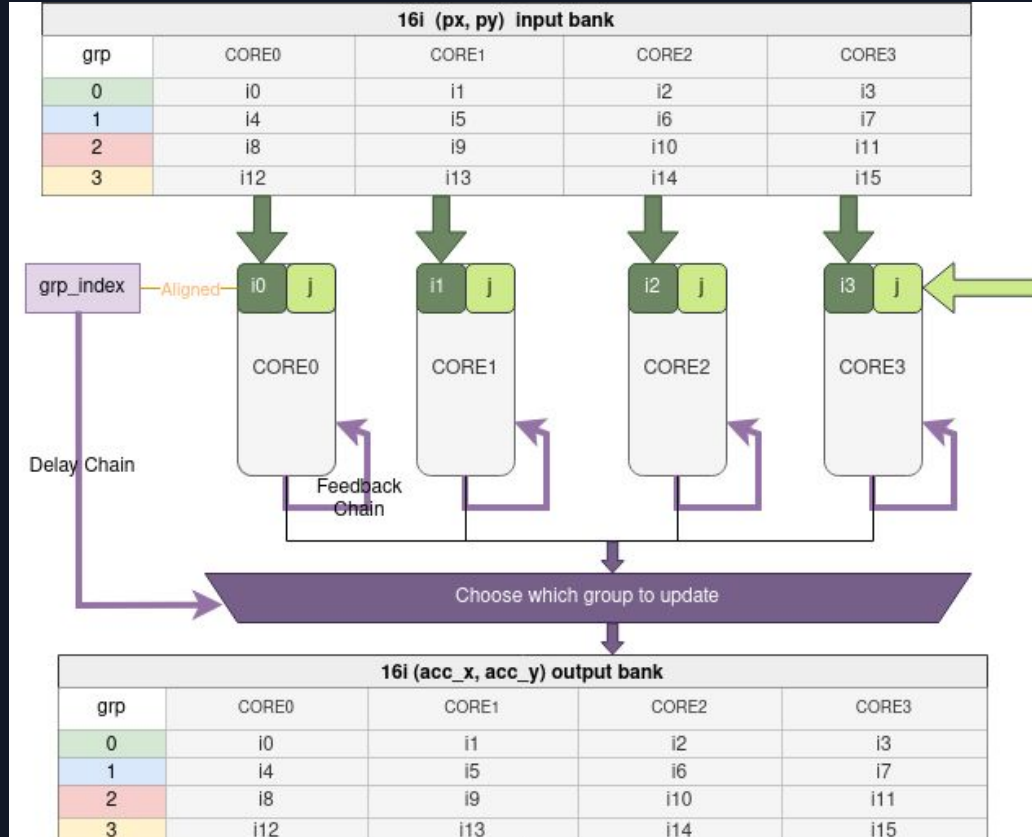


Two-Body-Core Self Accumulation

- Input: 2-stage adder for final accumulating stage → same *i* should not be calculated for next cycle.
- Fixed 2-reg feedback chain
- Example: CORE0 - Cycle 19 (First Self Accumulating)



Four Two-Body Cores Dataflow



LOAD phase:

- Initialize 16 *i* input bank.
- Flush pipeline
- Store positions of 16 bodies

COMPUTE phase:

- Update 16 *i* output bank
- *grp_idx* **must loop** as 0-3
- *j* comes from ports and changes every **4** cycles
- Broadcast positions of *j*
- Each core has its **own** mass of *j* as mask, the *j_mass* of this lane may be set as 0 to hold acc value if (mask == 1)

VALID_OUT phase:

- Keep mask as 4'1111 to hold value.
- Wait enough cycles after COMPUTE.

Ideal-case time complexity:

$O(N^2) \to O(N^2/4)$

On-chip result registers:

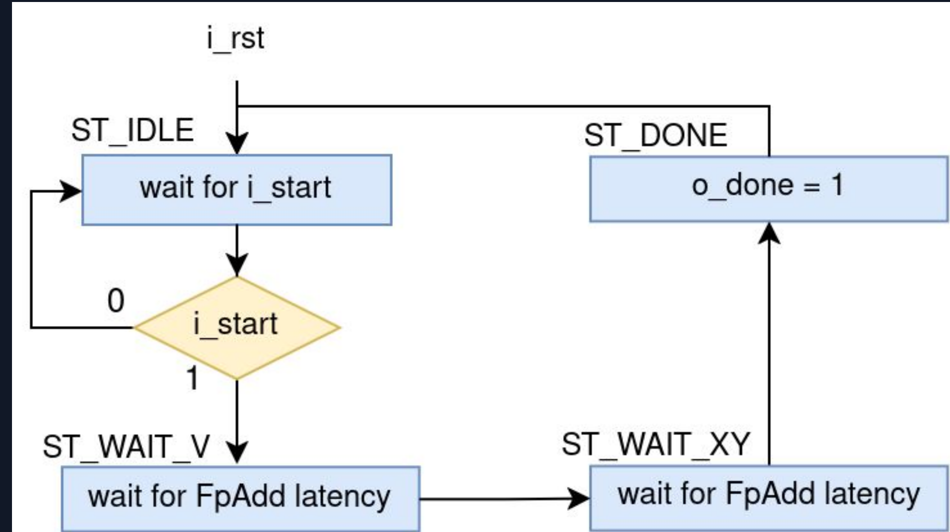
$2N \to 16 \times 2$

Leapfrog Integrator

Per-body update engine that applies the leapfrog velocity and position update using pipelined floating-point adders.

first step:
$v_{1/2} = v_0 + 0.5\, a_0$

later steps:
$v_{n+1/2} = v_{n-1/2} + a_n$
$x_{n+1} = x_n + v_{n+1/2}$

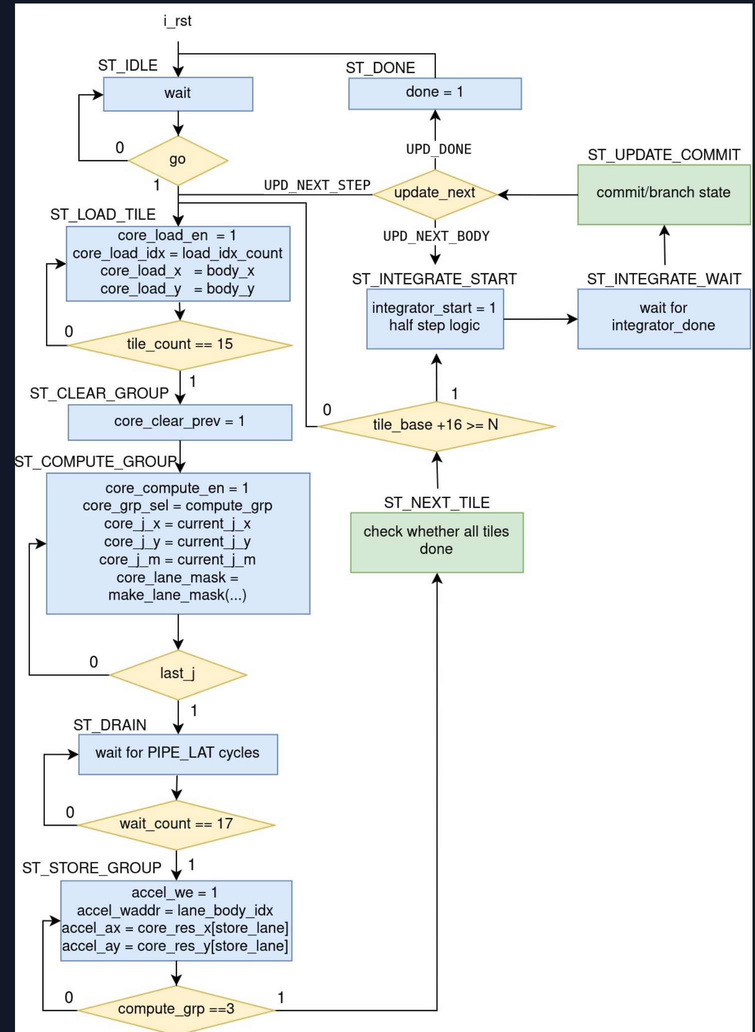


N-body Control

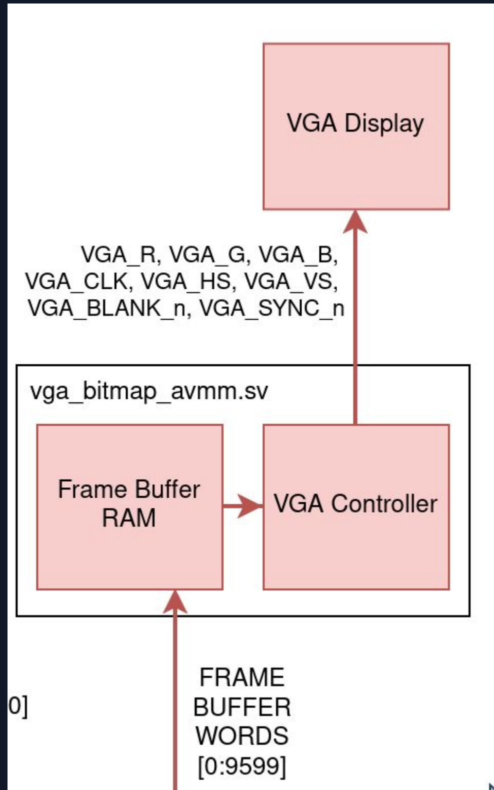
Top-level scheduler that coordinates tile loading, force computation, acceleration writeback, and body integration across each timestep.

1. Load body tiles into the compute cores
2. Stream each j-body through the four-core force calculation pipeline
3. Store the accumulated accelerations
4. Launch the integrator for each body

Manage timestep progress, initial half-step handling, memory addresses, write enables, and the final “done” signal.



Avalon Custom Peripheral: vga_bitmap_avmm



Dual-Port Frame Buffer RAM

- Port A: write from Avalon bus
- Port B: read by VGA controller

- Software renderer writes simulation frames into framebuffer memory
- VGA controller continuously reads framebuffer data for real-time display

Frame buffer:

640∗480 1-bit pixel map stored as 9600 32-bit words (20 words per row)

Testbenches

System verilog simulation

- FpAdd
- FpMul
- FpInvSqrt
- N-body mem
- N-body integrator
- N-body control

Software test program

- Avalon bus (avmm_smoke_test.c)
- Actual outputs generated by the FPGA (avmm_frame_xy_dump.c)

Golden model simulation

- accelerator accumulation simulation (golden.py)
- N-body control simulation (golden_control.py)
- simulate avmm_accel and software-visible outputs (golden_avmm_xy.py)

System verilog simulation

Frame Data

- Generated simulation input frames for 1024-body using custom Python-based input generators.
- Produced 27-bit formatted frame initialization files for RTL simulation and FPGA testing.
- Created initialization datasets (`frame0_1024binit200_27bits.txt`) used as hardware test inputs

e > tb > frame_input > frame0_1024binit200_27bits.txt

```
# i   px      py      vx      vy      m   (S1E8M18 hex)
  0   20CE1A0  2156136  0000000  0000000  1FA8587
  1   211235F  20C7D02  0000000  0000000  1F443B7
  2   60FA270  2134B72  0000000  0000000  1FBD6AC
  3   60E3DC4  218E5AC  0000000  0000000  1F3A965
```

-Pass all the arithmetic computation units and N-body mem tests using generated frame data.

Golden-model Simulation

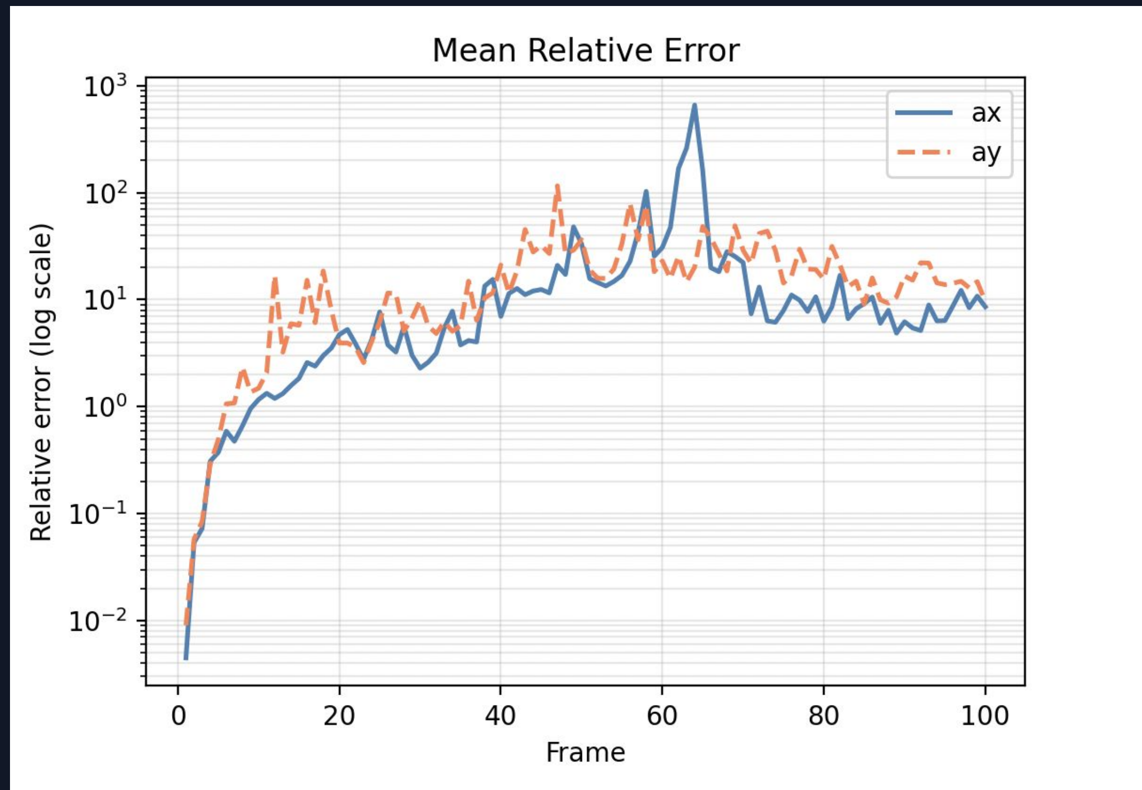
Golden Model: Python-based reference model used to generate expected hardware outputs

- Bit-accurate mimic of the custom 27-bit floating-point RTL modules:
  - FP27 Adder
  - FP27 Multiplier
  - FP27 Fast Inverse Square Root
- Uses custom S1E8M18 floating-point format:
  - 1 sign bit
  - 8 exponent bits
  - 18 mantissa bits
- Computes N-body accelerations using the same arithmetic flow as the FPGA datapath:

$$\mathbf{a}_i \approx G \cdot \sum_{j \neq i} \frac{m_j \mathbf{r}_{ij}}{(\|\mathbf{r}_{ij}\|^2 + \epsilon^2)^{3/2}}$$

- Generates “golden” acceleration outputs for each particle using the same frame data inputs

Accuracy Validation: 27-bit vs Float64




```
Summary (frame 1 → 100)

Mean rel error ax:  2.293e+01
Mean rel error ay:  1.802e+01
Max rel error  ax:  1.636e+05
Max rel error  ay:  2.227e+04

Mean abs error ax:  5.781e-02
Mean abs error ay:  5.801e-02

N bodies : 256
EPS²     : 0.25
G0       : 10.0   DT : 0.1
Format   : S1E8M18 (27-bit)
Ref      : float64
```

Acceleration Error (Frame 1, same initial state)
Mean relative error ax: 0.45% Mean relative error ay: 0.90%

Key observations:
- Frame 1 error reflects pure 27-bit format precision
- Error grows after frame 1 due to N-body chaotic divergence, not hardware inaccuracy
- 27-bit S1E8M18 format achieves < 1% mean error vs float64
- Acceptable for gravitational simulation

Outputs Comparison

- **Acceleration accumulation outputs**

2-body core

Hardware > tb > output > eps025_accel_1024binit20

```
# i ax ay (S1E8M18 hex)
  0  1d89bd6  5e6a03a
  1  1df62d7  1d30a45
  2  1dc6cdb  5dae6e1
  3  5f7ef30  1edc889
  4  5d8ed33  1d6dab7
  5  5e8a544  1e5bd07
  6  1e64e25  1d125de
  7  1ee5855  1e4f66d
  8  1f06154  5e90a7e
  9  5e654eb  5e6694e
 10  5ea312d  5e69236
 11  1d046f5  5e4c3b1
 12  5d2473a  5e73d5b
 13  1dc602e  5dd9b55
 14  5dcaa5e  1e8f471
```

four 2-body cores wrapped

Hardware > tb > output > four_core_wrapper_accel_1024bir

```
# i ax ay (S1E8M18 hex)
  0  1d89bd6  5e6a03a
  1  1df62d7  1d30a45
  2  1dc6cdb  5dae6e1
  3  5f7ef30  1edc889
  4  5d8ed33  1d6dab7
  5  5e8a544  1e5bd07
  6  1e64e25  1d125de
  7  1ee5855  1e4f66d
  8  1f06154  5e90a7e
  9  5e654eb  5e6694e
 10  5ea312d  5e69236
 11  1d046f5  5e4c3b1
 12  5d2473a  5e73d5b
 13  1dc602e  5dd9b55
 14  5dcaa5e  1e8f471
```

Golden-model

Hardware > tb > output > golden_accel_1024binit

```
# i ax ay (S1E8M18 hex)
  0  1D89BD6  5E6A03A
  1  1DF62D7  1D30A45
  2  1DC6CDB  5DAE6E1
  3  5F7EF30  1EDC889
  4  5D8ED33  1D6DAB7
  5  5E8A544  1E5BD07
  6  1E64E25  1D125DE
  7  1EE5855  1E4F66D
  8  1F06154  5E90A7E
  9  5E654EB  5E6694E
 10  5EA312D  5E69236
 11  1D046F5  5E4C3B1
 12  5D2473A  5E73D5B
 13  1DC602E  5DD9B55
 14  5DCAA5E  1E8F471
```

- Pass the outputs comparison of 2-body core, 4-Core wrapper with Golden model.

- **Control outputs**

N-body control

mem.sv | control_accel_1024binit200_27bits.txt | golden_control.py

Hardware > tb > output > control_accel_1024binit200_27bits.txt

```
# i x y vx vy ax ay (S1E8M18 hex)
  0  20ce1b2  2156100  1d49bd6  5e2a03a  1d89bd6  5e6a03a
  1  211237b  20c7d09  1db62d7  1cf0a45  1df62d7  1d30a45
  2  60fa24c  2134b64  1d86cdb  5d6e6e1  1dc6cdb  5dae6e1
  3  60e4da2  218e608  5f3ef30  1e9c889  5f7ef30  1edc889
  4  219cbb5  611d3eb  5d4ed33  1d2dab7  5d8ed33  1d6dab7
  5  2174abf  209c8d9  5e4a544  1e1bd07  5e8a544  1e5bd07
  6  20eceb9  21951e8  1e24e25  1cd25de  1e64e25  1d125de
  7  6195c48  619291c  1ea5855  1e0f66d  1ee5855  1e4f66d
  8  619febf  2182839  1ec6154  5e50a7e  1f06154  5e90a7e
  9  216f401  2189ff0  5e254eb  5e2694e  5e654eb  5e6694e
 10  219fb62  2177a67  5e6312d  5e29236  5ea312d  5e69236
 11  60bb439  217033b  1cc46f5  5e0c3b1  1d046f5  5e4c3b1
 12  618c586  212fe88  5ce473a  5e33d5b  5d2473a  5e73d5b
 13  618753f  2198eea  1d8602e  5d99b55  1dc602e  5dd9b55
 14  2085e57  61043b5  5d8aa5e  1e4f471  5dcaa5e  1e8f471
 15  615e2da  216daf1  5d051f2  5e0d0ad  5d451f2  5e4d0ad
 16  60c6294  20edaae  5d014c6  1e98c37  5d414c6  1ed8c37
 17  61a03c4  211e225  1e26070  1e2986e  1e66070  1e6986e
 18  2119aad  211d8b9  5dc8a33  5d248dc  5e08a33  5d648dc
```

Golden-model control

golden_control_state_1024binit200_27bits.txt

Hardware > tb > output > golden_control_state_1024binit200_27bits.txt

```
# i x y vx vy ax ay (S1E8M18 hex)
  0  20CE1B2  2156100  1D49BD6  5E2A03A  1D89BD6  5E6A03A
  1  211237B  20C7D09  1DB62D7  1CF0A45  1DF62D7  1D30A45
  2  60FA24C  2134B64  1D86CDB  5D6E6E1  1DC6CDB  5DAE6E1
  3  60E4DA2  218E608  5F3EF30  1E9C889  5F7EF30  1EDC889
  4  219CBB5  611D3EB  5D4ED33  1D2DAB7  5D8ED33  1D6DAB7
  5  2174ABF  209C8D9  5E4A544  1E1BD07  5E8A544  1E5BD07
  6  20ECEB9  21951E8  1E24E25  1CD25DE  1E64E25  1D125DE
  7  6195C48  619291C  1EA5855  1E0F66D  1EE5855  1E4F66D
  8  619FEBF  2182839  1EC6154  5E50A7E  1F06154  5E90A7E
  9  216F401  2189FF0  5E254EB  5E2694E  5E654EB  5E6694E
 10  219FB62  2177A67  5E6312D  5E29236  5EA312D  5E69236
 11  60BB439  217033B  1CC46F5  5E0C3B1  1D046F5  5E4C3B1
 12  618C586  212FE88  5CE473A  5E33D5B  5D2473A  5E73D5B
 13  618753F  2198EEA  1D8602E  5D99B55  1DC602E  5DD9B55
 14  2085E57  61043B5  5D8AA5E  1E4F471  5DCAA5E  1E8F471
 15  615E2DA  216DAF1  5D051F2  5E0D0AD  5D451F2  5E4D0AD
 16  60C6294  20EDAAE  5D014C6  1E98C37  5D414C6  1ED8C37
 17  61A03C4  211E225  1E26070  1E2986E  1E66070  1E6986E
 18  2119AAD  211D8B9  5DC8A33  5D248DC  5E08A33  5D648DC
```

- Pass the outputs comparison of n-body control with Golden model.

Software

Software Architecture

Thread 1
Accelerator

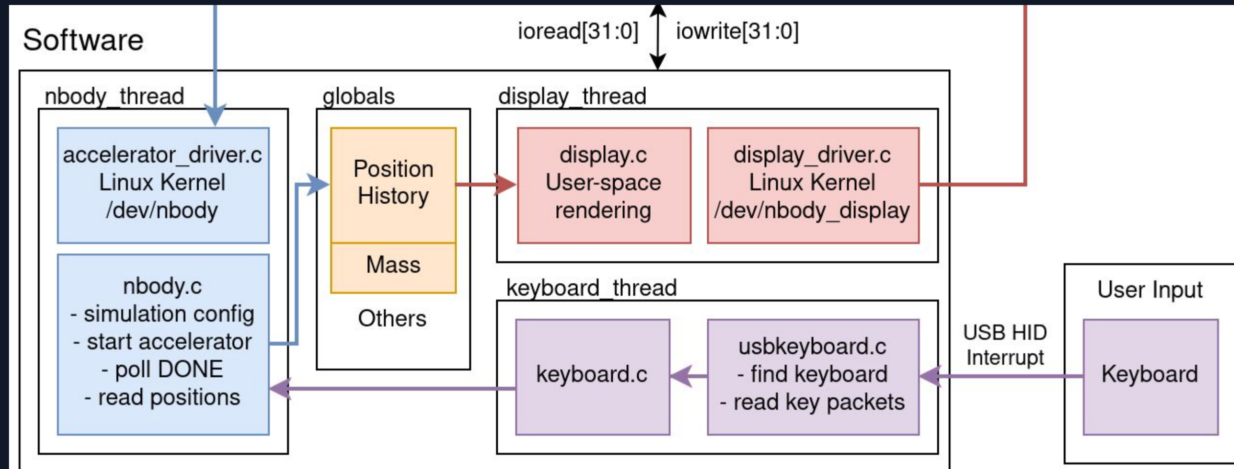
/dev/nbody

Thread 2
Display

/dev/nbody_display

Thread 3
USB Keyboard

libusb



Display Thread

- Selects the current frame from the shared position-history buffer
- Converts body positions into screen coordinates
- Renders particles/UI into a software framebuffer
- Sends framebuffer data to /dev/nbody_display
- Continuously updates VGA output while simulation runs

Display 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;
    }
}

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;
        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);
    }
}
```

USB Keyboard

- Reads keyboard input using the USB keyboard helper
- Updates shared control state based on key presses
- Supports pause/resume, reset, quit, frame navigation, and gap changes
- Uses condition/broadcast signals to wake other threads when state changes

Key	Action
SPACE	Pause / Resume the simulation
R	Reset to initial state with new random bodies
W / S	Increase / decrease GAP (simulation speed)
A / D	Step backward / forward through history by one readback frame
Q	Quit the simulation

Hardware/Software Interface

Display Hardware/Software Interface

Memory Layout

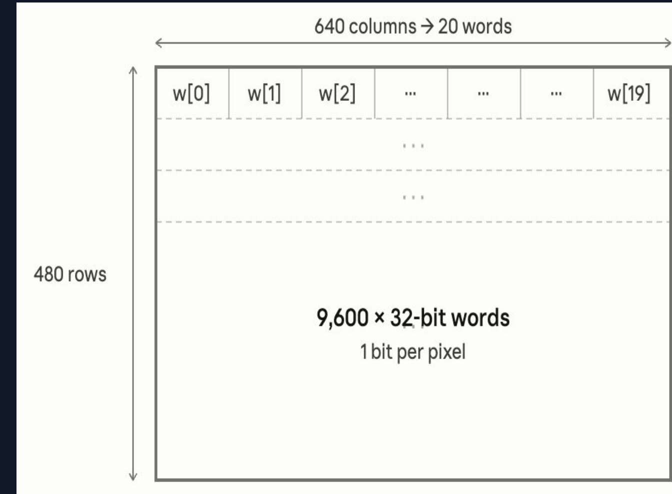
640 × 480 display, 1 bit per pixel
9,600 uint32 words total

Layout:
  word = y × 20 + x / 32
  bit = x % 32

  bit=1 → white pixel
  bit=0 → black pixel

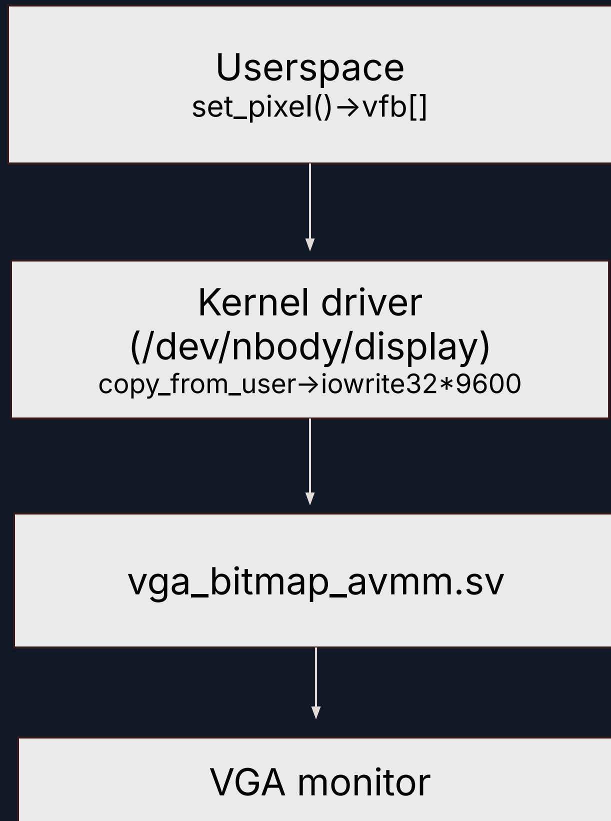
address calculation

```
/* pixel (x, y) maps to: */
word = y * 20 + x / 32;
bit  = x % 32;

/* set pixel white */
vfb[word] |= (1u << bit);

/* set pixel black */
vfb[word] &= ~(1u << bit);
```



9,600 words × 4 bytes per word = 38,400 bytes ≈ 38 KB

Display Software Interface



display_driver.c

```
case DISPLAY_WRITE_FRAME:
    copy_from_user(dev.kbuf,
                   (uint32_t __user *)arg,
                   DISPLAY_WORDS *
sizeof(uint32_t));

    for (i = 0; i < DISPLAY_WORDS; i++)
        iowrite32(dev.kbuf[i],
                  dev.virtbase + i * 4);
    return 0;
```

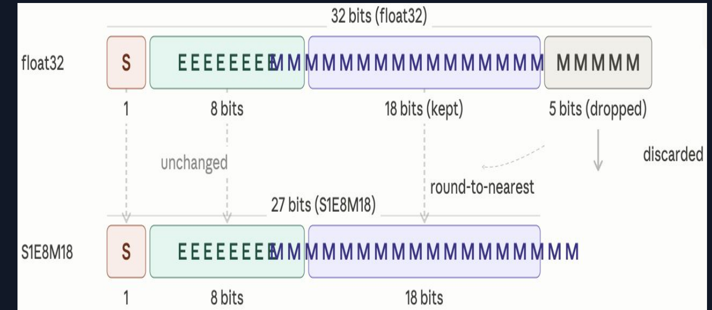
Accelerator Interface

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.

float conversion

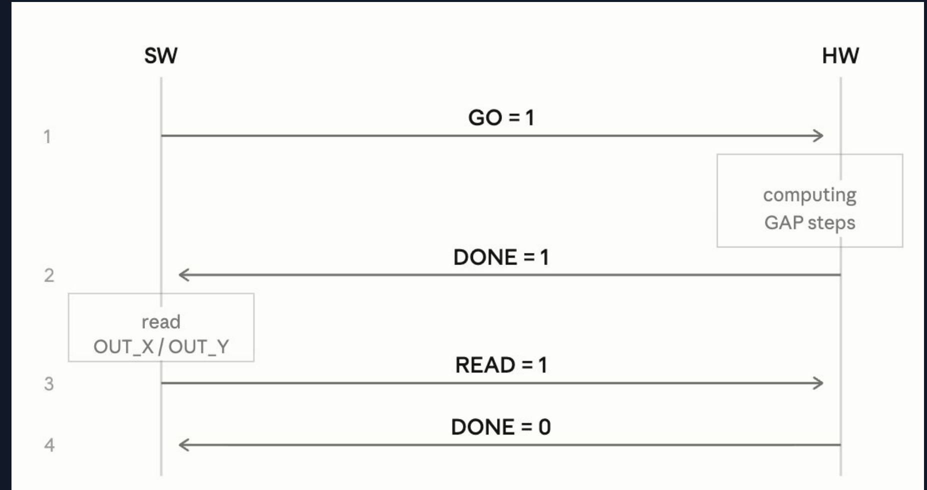
Userspace → IEEE-754 float32
Hardware → Custom S1E8M18 (27-bit)

f32_to_f27_bits() on write
f27_bits_to_f32() on read



Driver Control Flow

ioctl	Action
NBODY_WRITE_CONFIG	Set N_BODIES and GAP
NBODY_WRITE_BODIES	Stream X,Y,M,VX,VY × N
NBODY_START_RUN	Pulse GO = 1
NBODY_CHECK_DONE	Poll DONE register
NBODY_READ_RESULTS	Read OUT_X, OUT_Y × N
NBODY_CLEAR_READ	Lower READ = 0



Software Test Program

avmm_smoke_test.c

- Tests basic HPS-to-FPGA Avalon-MM communication through `/dev/mem`.
- Verifies the `nbody_accel_avmm` control path using a simple zero-body run.
- Checks correct `GO` / `DONE` behavior and `DONE` clearing.
- Loads a simple one-body test case and reads back output X/Y values.
- Writes a VGA test pattern through `vga_bitmap_avmm`
- for display verification.

- `pass:` verified Avalon-MM communication and basic control/read paths.

avmm_frame_xy_dump.c

- Tests the real N-body accelerator on FPGA hardware using an input frame file.
- Loads body data through Avalon-MM registers and starts computation with `GO`.
- Waits for `DONE`, then reads back final `X/Y` positions.
- Dumps the hardware output for comparison against the golden reference file.

`-pass`: means the accelerator completed and hardware results were successfully dumped.

Outputs Comparison

- **avmm_frame_xy outputs**

It models what the software would see after loading particle data into the accelerator, starting the computation with GO, waiting for DONE, and reading back OUT_X and OUT_Y.

avmm_frame_xy_dump.c

Open | avmm_frame_xy_dump.txt ~/Downloads | Save

```
# source frame0_1024binit200_27bits.txt
# n_bodies 1024
# gap 1
# i x y (S1E8M18 hex)
   0  20CE1B2  2156100
   1  211237B  20C7D09
   2  60FA24C  2134B64
   3  60E4DA2  218E608
   4  219CBB5  611D3EB
   5  2174ABF  209C8D9
   6  20ECEB9  21951E8
   7  6195C48  619291C
   8  619FEBF  2182839
   9  216F401  2189FF0
  10  219FB62  2177A67
  11  60BB439  217033B
  12  618C586  212FE88
  13  618753F  2198EEA
  14  2085E57  61043B5
  15  615E2DA  216DAF1
  16  60C6294  20EDAAE
  17  61A03C4  211E225
  18  2119AAD  211D8B9
  19  2198BE2  2148BAA
  20  6130BBA  60E5975
  21  214F07D  6197F4B
  22  2142B49  2144415
  23  6173DAA  618A2FE
  24  6149EDC  612CD05
  25  20F04B4  60E23AE
  26  21A1ABB  618F98B
  27  61746D7  6183BB9
```

Golden-model (golden_avmm_xy.py)

golden_xy_1024binit200_27bits.txt

Hardware > tb > output > golden_xy_1024binit200_27bits.txt

```
1   # source /homes/user/stud/fall25/lh3365/ee4840/CSE
2   # n_bodies 1024
3   # gap 1
4   # first_step 1
5   # i x y (S1E8M18 hex)
6      0  20CE1B2  2156100
7      1  211237B  20C7D09
8      2  60FA24C  2134B64
9      3  60E4DA2  218E608
10     4  219CBB5  611D3EB
11     5  2174ABF  209C8D9
12     6  20ECEB9  21951E8
13     7  6195C48  619291C
14     8  619FEBF  2182839
15     9  216F401  2189FF0
16    10  219FB62  2177A67
17    11  60BB439  217033B
18    12  618C586  212FE88
19    13  618753F  2198EEA
20    14  2085E57  61043B5
21    15  615E2DA  216DAF1
22    16  60C6294  20EDAAE
23    17  61A03C4  211E225
24    18  2119AAD  211D8B9
25    19  2198BE2  2148BAA
26    20  6130BBA  60E5975
27    21  214F07D  6197F4B
28    22  2142B49  2144415
29    23  6173DAA  618A2FE
30    24  6149EDC  612CD05
```

- Verified the RTL AVMM interface producing the same software-visible results as the Python golden model.

DEMO

CSEE 4840 · Spring 2026