Design Document for Liquid-Physics-Simulator

Aidan Dodge (acd2243) Da Won Kim (dk3311) Daanish Khan (dk3472)

May 14, 2026

1 Introduction

This project presents a hardware-accelerated, interactive liquid physics simulator implemented on the DE1-SoC platform using a hardware/software co-design architecture. To overcome the memory and compute bottlenecks of traditional software-based fluid dynamics, we offload a fluid algorithm to a custom digital accelerator on the Cyclone V FPGA. The ARM Cortex-A9 Processor manages high-level system control and user interaction, while the FPGA fabric executes the core physics engine utilizing an optimized 18-bit fixed-point datapath and localized block RAM (BRAM).

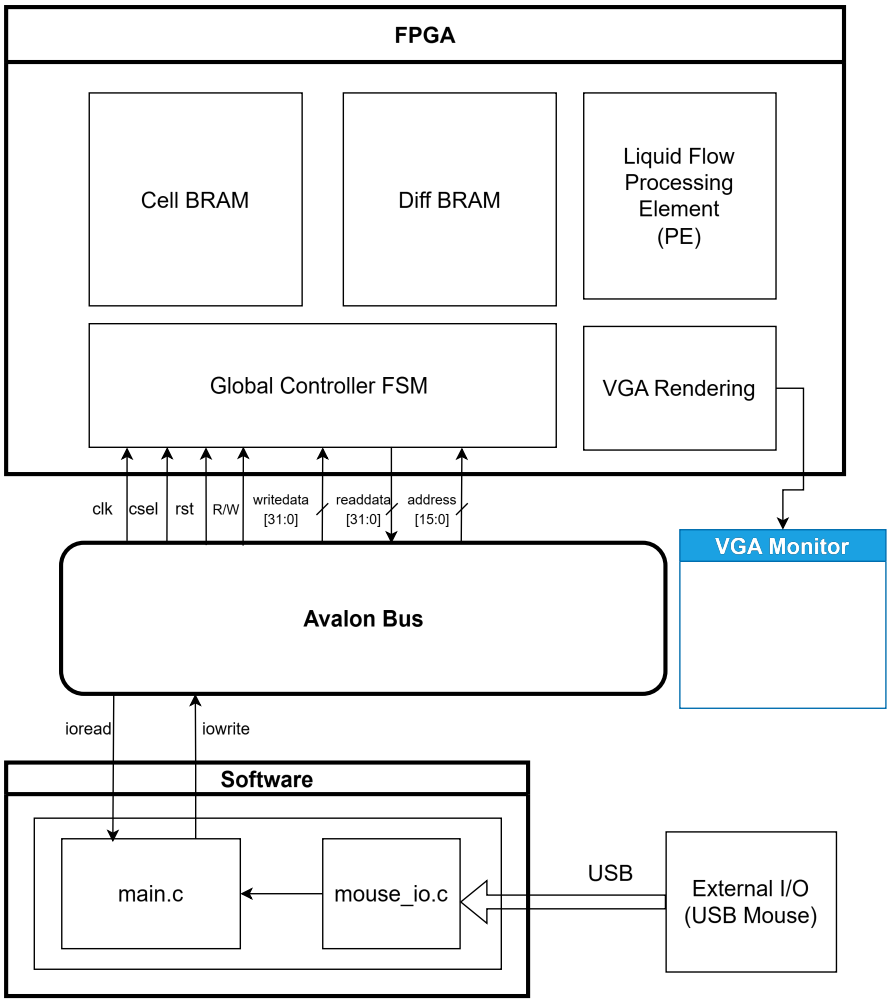

Figure 1: System Block Diagram

Implementation Status

We were unable to complete the integrated system on hardware. The components were developed individually, and integration was not completed within the project timeline. Therefore, what follows is a description of the design and the components written for it. Source files implementing each component are included in the appendix and were written to express the design directly.

2 System Block Diagram

Our system has two halves. The HPS (Hard Processor System) is the ARM Cortex-A9 CPU built into the Cyclone V chip, and it runs our C software. The FPGA fabric is the programmable logic where we place our custom Verilog. The two halves talk to each other over the Avalon bus.

The HPS does not run any physics math. Its only jobs are: read the USB mouse, turn the mouse position into grid coordinates, load the starting cell map at boot, and tell the FPGA when to run each simulation step.

The FPGA contains four blocks. The Particle block is our custom Verilog physics engine. The Global Controller is a top-level FSM; it sequences simulation steps against the VGA refresh and arbitrates traffic on the Avalon bus. The VGA rendering block reads the cell grid and drives the HSYNC, VSYNC, and 8-bit RGB lines to the monitor. The Line block holds the wall geometry, the pixels water cannot pass through, in a dedicated Line Memory.

The cell grid lives in BRAM (Block RAM), which is small on-chip memory inside the FPGA. The Cyclone V provides BRAM in 10 Kb units called M10K blocks. The Particle block stores one 32-bit word per cell, laid out as follows:

Field	Width	Role
`CellType`	1 b	`Blank` (0) or `Solid` (1). The HPS writes this at map load. The physics engine only reads it.
`Liquid`	18 b	Scalar liquid volume in Q2.16 fixed-point, range [0, 1.25].
`Settled`	1 b	Set when the cell has stopped changing. The renderer uses it to hide flow indicators on still water.
`SettleCount`	4 b	Counter that drives the settling heuristic.
`isDownFlowing`	1 b	Set when this cell and its top neighbor both hold liquid. The VGA block uses it to draw falling streams at full opacity.
`Reserved`	7 b	Padding to a 32-bit word for aligned memory access.

A second BRAM of the same size holds the per-cell flow scratch buffer, called `Diffs`. We use it during a simulation tick; Section 3.3 explains how. The Verilog FSM computes each cell's neighbor addresses from its (x, y) index, so we do not need to store neighbor pointers. The Line block holds a third BRAM, Line Memory, which stores the wall geometry.

3 Algorithms

The FPGA runs a 2D cellular-automaton liquid solver. The simulation world is a $W \times H$ grid of cells stored in on-chip BRAM. Each cell holds two pieces of state: a scalar `Liquid` amount, and a one-bit `CellType` flag that marks the cell as `Blank` (can hold water) or `Solid` (a wall). On every tick, a Verilog FSM sweeps the grid. For each `Blank` cell, it applies four flow rules in a fixed order: drain down, spread left, spread right, then push up under pressure.

The work per cell is constant and local. Each cell only reads itself and its four neighbors, so the pipeline produces one evaluated cell per clock once it is full. There is no global solver, no particle list, and no velocity or position state. The only value that changes per step is the `Liquid` amount in each cell.

The HPS never touches the physics. It only handles the control loop: mouse input, startup, Avalon traffic, and frame timing.

We validated our rules and constants against an open-source Unity reference of the same cellular-automaton model[1]. Our Verilog uses identical rules and identical constants. We drop the scaffolding that reference needs for CPU parallelism (per-cell modify fields, entity indexing, sprite-sheet fields), because a single pipelined datapath does not need it.

3.1 Cell State and Algorithm Constants

Each cell holds the 32-bit word described in Section 2. At our target size of $W = H = 64$, the primary cell memory is $4096 \cdot 32 = 131{,}072$ **bits**, or about 13 M10K blocks. The `Diffs` scratch buffer is the same size. We do not store neighbor pointers. The Verilog FSM computes them from each cell's (x, y) index with a small adder at the BRAM address port.

The algorithm has six real-valued constants. Each one fits exactly in 18-bit Q2.16 fixed-point, which is the format our Verilog datapath uses.

Constant	Value	Role
`MaxLiquid`	1.0	Nominal capacity of a cell at atmospheric pressure
`MinLiquid`	0.005	Threshold below which a cell is treated as empty
`MaxCompression`	0.25	Maximum extra liquid a cell below may hold
`MinFlow`	0.005	Flow amounts below this are clamped to zero
`MaxFlow`	4.0	Upper bound on flow per cell per step
`FlowSpeed`	1.0	Damping factor, $(0, 1]$

These constants live in the Water Module as Verilog parameters. The HPS cannot change them at runtime. If we later want to tune `FlowSpeed` from software, we can promote it to a writable Avalon register. For now we treat all six as compile-time values.

3.2 Flow Rules

Every `Blank` cell that is not `Settled` and holds more than `MinLiquid` liquid runs the four rules below, in order. The pipeline does not overwrite the cell's `Liquid` value directly. Instead, each rule writes its flow ϕ into the shared `Diffs` buffer. A separate second pass applies the accumulated deltas (Section 3.3).

Rule 1 — Downward Flow (Gravity)

If the bottom neighbor is `Blank`, the cell tries to fill it to the equilibrium level $V(r, d)$:

$$V(r,d) = \begin{cases} \text{MaxLiquid}, & s \le \text{MaxLiquid} \\ \dfrac{\text{MaxLiquid}^2 + s \cdot \text{MaxCompression}}{\text{MaxLiquid} + \text{MaxCompression}}, & \text{MaxLiquid} < s < 2\,\text{MaxLiquid} + \text{MaxCompression} \\ \dfrac{s + \text{MaxCompression}}{2}, & \text{otherwise} \end{cases} \tag{1}$$

Here r is the liquid left in the source cell, d is the liquid already in the bottom cell, and $s = r + d$. The flow is $\phi = V(r, d) - d$, clamped to $[0,\ \min(\text{MaxFlow}, r)]$ and scaled by `FlowSpeed`.

The three cases describe how pressure stacks up. If the column is under-filled (case 1), the bottom takes everything. If it is a little over-full (case 2), the bottom holds `MaxCompression` more than the top. If it is very over-full (case 3), the bottom splits the excess evenly plus the compression bias.

Rule 2 — Horizontal Flow (Spreading)

With whatever liquid r is left after Rule 1, the cell spreads sideways into any `Blank` neighbor:

$$\phi_\ell = \frac{r - d_\ell}{4}, \qquad \phi_r = \frac{r - d_r}{3}. \tag{2}$$

Each flow is clamped to $[0,\ \min(\text{MaxFlow}, r)]$ and scaled by `FlowSpeed`. The reference divides by 4 on the left and 3 on the right, which gives resting pools a small rightward drift. We make the divisor a parameter in hardware. That way we can either match the reference exactly (for validation) or symmetrize to /4 on both sides (for correctness). Dividing by 4 is a free right shift; dividing by 3 takes one DSP multiplier.

Rule 3 — Upward Flow (Pressure)

If the cell still holds more liquid than it should after Rules 1 and 2, and its top neighbor is `Blank`, it pushes the excess upward:

$$\phi_{\text{top}} \ = \ r \ - \ V(r,\, d_{\text{top}}). \tag{3}$$

Again clamped and `FlowSpeed`-scaled. This reuses the same V function as Rule 1. That matters for hardware: we build V once as a shared combinational block and share it between Rules 1 and 3.

3.3 Two-Pass Update Scheme

One simulation tick runs as two FSM states in the Global Controller. Each state sweeps the whole grid once.

1. **Evaluate.** For every non-`Solid` cell (x, y) in raster order, the physics pipeline computes the four flows. Each flow ϕ subtracts from `Diffs`$[x, y]$ and adds to `Diffs`$[x', y']$, where (x', y') is the neighbor receiving the flow. The `Liquid` BRAM is read but never written during this pass.
2. **Commit.** A second sweep reads each (`Liquid`$[x, y]$, `Diffs`$[x, y]$) pair, adds them, writes the new value into the cell BRAM, and clamps any result below `MinLiquid` to zero. The `Diffs` BRAM is cleared as a side effect.

Splitting evaluate from commit makes the update order-independent. Every cell sees the same snapshot of the grid. This gives us an advantage over how the Unity reference solves the same ordering problem. Instead of one shared `Diffs` buffer, it gives every cell five modify fields (`modifySelf`, `modifyBottom`, `modifyTop`, `modifyLeft`, `modifyRight`). During its parallel evaluate pass, each CPU thread only writes into its own cell's fields, so threads never collide. That works, but it costs about 5× the scratch memory (around 66 Kb versus our 13 Kb). We only process one cell per clock, so the races that scheme is guarding against cannot happen for us, and one shared buffer is enough.

4 Resource Budgets

We are using the DE1-SoC board with the Cyclone V SE A5 chip. The maximum available resources for the chip and board are shown in Figure 2 and Figure **??**.

Product Line		Cyclone V SE SoCs[1]			
		5CSEA2	5CSEA4	5CSEA5	5CSEA6
Resources	LEs (K)	25	40	85	110
	ALMs	9,430	15,880	32,070	41,910
	Registers	37,736	60,376	128,300	166,036
	M10K memory blocks	140	270	397	557
	M10K memory (Kb)	1,400	2,700	3,970	5,570
	MLAB memory (Kb)	138	231	480	621
	Variable-precision DSP blocks	36	84	87	112
	18 x 18 multipliers	72	168	174	224

Figure 2: Maximum Resources for Cyclone V SE A5

For our water particles, we used 32 bits or 4 bytes of data to represent each pixel particle. All arithmetic was performed 18×18 fixed-point format to fit the 18×18 multipliers on the board and limit resource demand. We estimated needed around **20** M10K memory blocks needed, and ended up using **36 M10K memory blocks** (as seen in Fig. 3, still only 9% of what is available on the Cyclone chip.

The results in figure 3 are from the Quartus Prime Fitter report for the full SoC system including our water simulator design module, the Platform Designer-generated interconnect, and the HPS block.

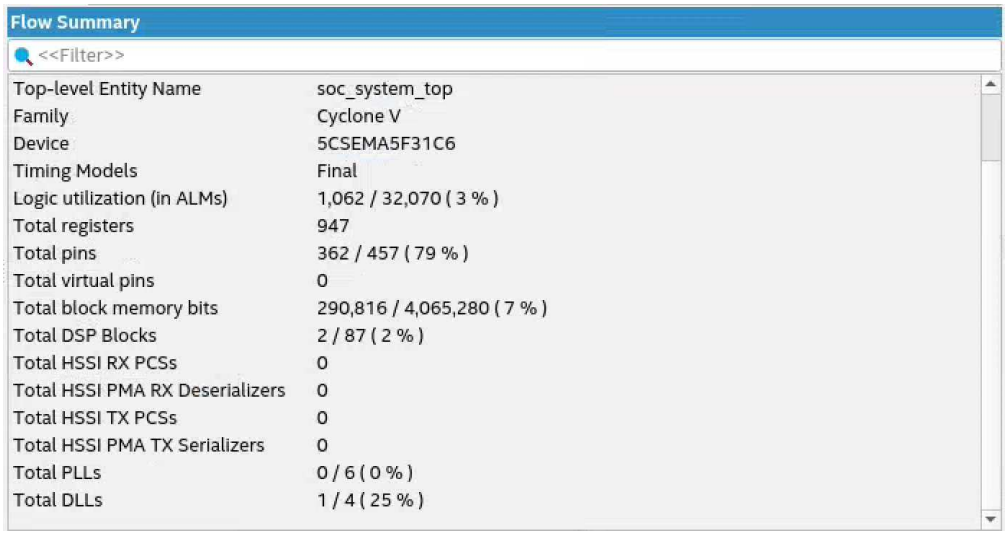

Figure 3: Post-Synthesis Quartus resource report

Out of the 1062 total ALMs used by our system, 797 were dedicated to the whole water simulator block (controller, physics engine, and particle memory). The physics engine (PE) consumed the most in terms of logic with 413 ALMs and 689 combinational LUTs. This was **40%** of the total ALM budget for the water simulator. The high usage is expected because our implementation of the PE used two cascaded Q2.16 multiples, three comparators, and multiple adders.

The BRAM for the cells consumed 26 M10K blocks (208,896 bits for 4096∗32-bit dual-port storage) and the diff BRAM consumed 10 M10K blocks (81,920 bits for 4096∗20-bit signed storage). The VGA renderer block only consumed 45 ALMs and 51 registers.

4.1 Timing Analysis

Timing analysis was performed with the Quartus Prime Timing Analyzer tool, and we used the Slow 1100mV 85 model, similar to lab 3. Figures 4 and 5 show the timing of our system with the *Fmax* and *slack* key variables.

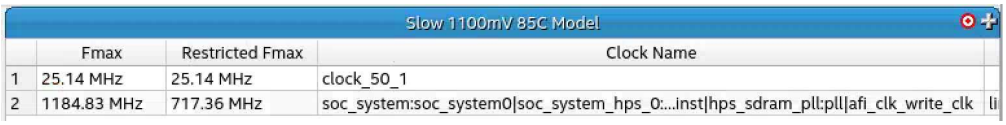

Figure 4: Post-Synthesis Quartus Fmax report

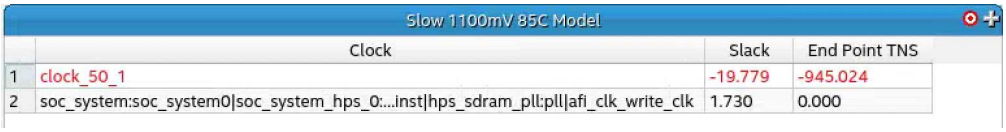

Figure 5: Post-Synthesis Quartus slack report

The −19.779 ns setup slack and 23.53 MHz Fmax figure are below violate our timing conditions: [1] no negative slack, and [2] minimum Fmax of 50 Mhz. We believe the failures are caused by unresolved

cross-clock domain paths involving the clock_50_1 fabric domain (seen in 5). The rest of the paths had a healthy slack above zero but the major negative slack came from a violation in the setup path.

Another estimate for the timing failure is the large combinatorial path taken in the physics engine. We attempted to address this issue by pipelining the path, but our Fmax and slack still fell below the requirements. We successfully converted the .sof file to .rbf for the FPGA's SD card boot to add our module to the board, but our system did not run successfully.

5 Register Map

The HPS sees the simulator as a 32-bit Avalon-MM slave peripheral. Software communicates with the FPGA through a small set of control/status registers plus one bulk memory window that exposes the 64×64 cell grid.

Control registers occupy the low address page. The HPS uses them to start a simulation step, clear or reload the world state, pass in mouse-derived grid coordinates, and configure how the VGA output should render the water state. The bulk cell state itself lives in a BRAM-backed memory window beginning at `GRID_MEM_BASE`. This arrangement is similar to the Lab 3 example. A few low address act like device registers, while a larger memory region holds the state that the custom hardware consumes and renders.

5.1 Register Map

Table 1 summarizes the register map. Each register is 32 bits wide and is accessed at its byte offset from the peripheral base.

Offset	Register	R/W	Fields used	Used by
`0x0000`	`CTRL_REG`	W	[1] RESET, [2] LOAD_MAP, [4] BRUSH_APPLY (write-one-pulse)	`reset_fpga()`, `load_empty_map()`, `brush_apply()`
`0x0004`	`STATUS_REG`	R	[0] BUSY, [2] MAP_READY (polled during startup)	`reset_fpga()`, `load_empty_map()`
`0x000C`	`MOUSE_POS_REG`	W	[15:0] x, [31:16] y	`brush_apply()`
`0x0010`	`BRUSH_CFG_REG`	W	[1:0] tool, [15:8] radius, [31:16] liquid (Q2.16 low 16 bits)	`brush_apply()`
`0x0014`	`STEP_CFG_REG`	R/W	[0] AUTO_RUN = 1, [1] FRAME_LOCK = 1, [15:8] STEPS_PER_FRAME = 1	`configure_runtime()`
`0x0040`	`VGA_CTRL_REG`	R/W	[0] ENABLE = 1	`configure_runtime()`
`0x1000–0x4FFC`	`GRID_MEM`	R/W	4096 × 32-bit cell words; cell (x, y) at `0x1000 + 4(64y + x)`	`load_empty_map()`

Table 1: Register interface as exercised by the software.

5.2 Meaning of one `GRID_MEM` word

Each entry in `GRID_MEM` is one 32-bit word, matching the cell layout introduced earlier in the document. Packing the cell into a single word keeps all HPS accesses naturally aligned and makes software initialization straightforward.

Bits	Field	Meaning
`[0]`	`CellType`	0 for blank, 1 for solid. Software sets this when loading the map.
`[18:1]`	`Liquid`	18-bit Q2.16 fixed-point liquid amount used by the physics engine.
`[19]`	`Settled`	Indicates that the cell has reached a quiescent state and may be rendered differently in debug mode.
`[23:20]`	`SettleCount`	Small counter used by the settling heuristic.
`[24]`	`isDownFlowing`	Flow-visualization flag for the VGA block.
`[31:25]`	`Reserved`	Reserved for future use, such as extra visualization or brush metadata.

Table 2: Packing of one cell word in `GRID_MEM`.

6 Software

6.1 Overview

The host-side software runs on the ARM Cortex-A9 of the DEC-SoC and communicates with the simulator peripheral through the Avalon-MM register interface. It is organized into three source files. `water_sim.h` declares the register map, the packed cell word layout, and the `ioctl` commands exposed by the kernel driver. `mouse_io.c` handles raw USB mouse input, accumulating PS/2 packets into a pixel-space cursor and converting that cursor to grid coordinates. `main.c` is the top level program: it brings up the FPGA, enters a steady-state loop that translates mouse events into brush commands, and tears down cleanly on Ctrl-C.

6.2 `main.c`: Top-Level Program

`main.c` runs in three distinct phases, illustrated in Figure 6: a one-time **setup** phase that brings the FPGA up and switches it into autonomous mode; a **runtime** loop that runs until `SIGINT` is received; and a one-time **teardown** phase that resets the FPGA and releases resources.

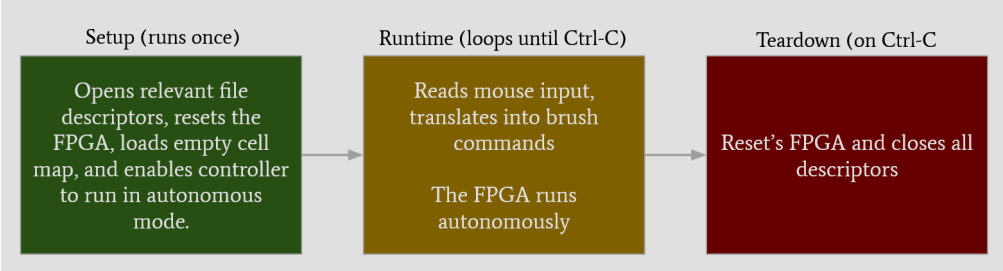

Figure 6: Three-phase structure of `main.c`: setup, runtime, teardown.

The three phases are described in the subsections that follow. Each phase is implemented as a small number of helper functions whose names appear in the `Used by` column of Table 1; together they exercise the subset of the Avalon register interface.

6.2.1 Setup

Setup is a one-time sequence of four function calls executed in order, each gated on the previous one succeeding (Figure 7). The phase moves the FPGA from power-on into a known steady state in which the simulation is running autonomously and the cell grid is fully initialized.

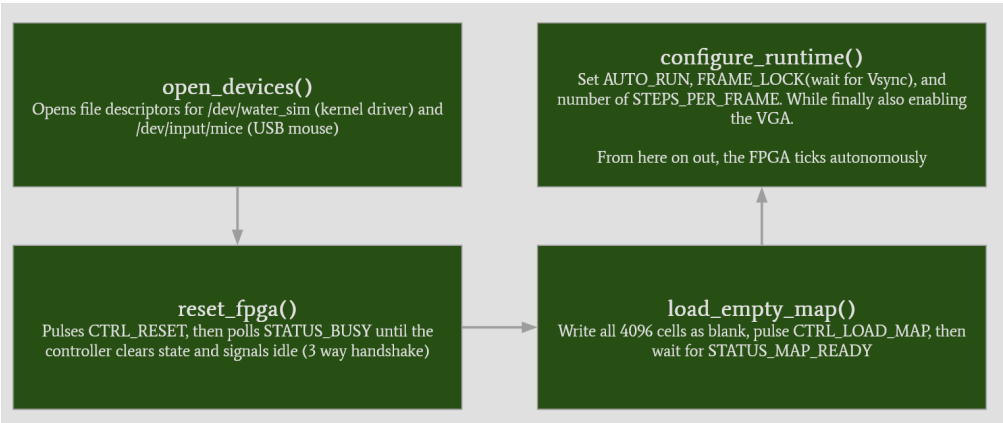

Figure 7: The four setup steps, with the FPGA-facing operations annotated in red.

`open_devices()` Opens two file descriptors: `/dev/water_sim`, the character device exposed by the kernel driver, and `/dev/input/mice`, the USB mouse.

`reset_fpga()` Pulses `CTRL_RESET`, then polls `STATUS_BUSY` with a 100 ms timeout. This ensures the controller has cleared its FSM state and the cell Liquid fields before the next step.

`load_empty_map()` Writes 4096 cells one at a time, each as `PACK_CELL_BLANK(0)` (blank type, zero liquid). Then pulses `CTRL_LOAD_MAP` and polls `STATUS_MAP_READY` to confirm the wall geometry has been latched internally. The cell-by-cell write loop is the slowest step (one `ioctl` per cell), but runs only once at startup; total cost is on the order of 100 ms.

`configure_runtime()` The architectural pivot point of the program. Writes `STEP_CFG_REG` with AUTO_RUN, FRAME_LOCK, and STEPS_PER_FRAME all set to 1, then enables video by writing `VGA_CTRL_REG` bit 0. After this single configuration write, the controller is in autonomous mode: it advances the simulation on its own, one tick per VSYNC, without further intervention from the HPS. The software never issues a STEP command after setup completes.

6.2.2 Runtime

Once setup is complete, the program enters a two-function loop that continues until `SIGINT` clears the `running` flag (Figure 8). The loop's job is narrow: read one mouse event, dispatch it to the appropriate brush command, repeat.

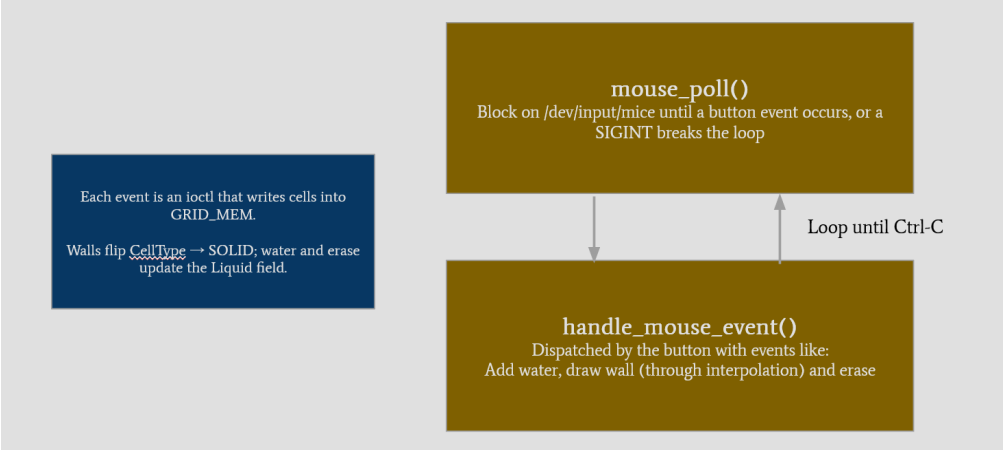

Figure 8: The runtime loop. Each iteration consumes one mouse event and produces one or more `BRUSH_APPLY` `ioctl` calls.

`mouse_poll()` Blocks on the mouse file descriptor until a packet arrives. Returns the parsed event in a `mouse_event_t` struct already populated with grid-space coordinates.

`handle_mouse_event()` Dispatches based on which button is held: left-click events become `brush_apply()` calls with the `ADD` tool, middle-click events with `ERASE`, and right-click events with `WALL`. Right-click drags trigger an additional step: because mouse samples can jump several grid cells between consecutive events, the software interpolates between sample points using a Bresenham line rasterizer and fires one brush per cell along the line. This produces a continuous wall rather than a dotted one.

Each brush event becomes one `ioctl` call that writes `MOUSE_POS_REG`, `BRUSH_CFG_REG`, and pulses `CTRL_BRUSH_APPLY` as a single atomic operation. The kernel driver bundles the three register writes into one compound operation so the FPGA never observes a half-applied brush command. Walls modify each cell's `CellType` field; water and erase events modify the `Liquid` field.

6.2.3 Teardown

When the user sends `SIGINT` (typically via Ctrl-C), the signal handler clears the `running` flag. Because the signal handlers are installed without `SA_RESTART`, the blocking `read()` inside `mouse_poll()` returns `-1` with `errno = EINTR`, breaking the main loop and falling through to teardown

`ctrl_pulse(CTRL_RESET)` Pulses RESET so the next run of the program starts from a clean controller state. Errors are logged but otherwise ignored — the program is exiting anyway, and a failed reset is better than refusing to release the kernel driver.

`close(water_fd)` Releases the kernel driver file descriptor.

`mouse_cleanup()` Closes `/dev/input/mice` and resets internal cursor tracking state.

After teardown returns, `main()` returns 0 and the process exits.

6.3 Mouse I/O

The Mouse I/O module is the input adapter between the raw USB mouse stream and the higher-level brush logic in `main.c`. As shown in Figure 9, it consumes button state and relative movement from `/dev/input/mice`, maintains a cursor in screen space, and returns a compact `mouse_event_t` containing the current grid position and requested action.

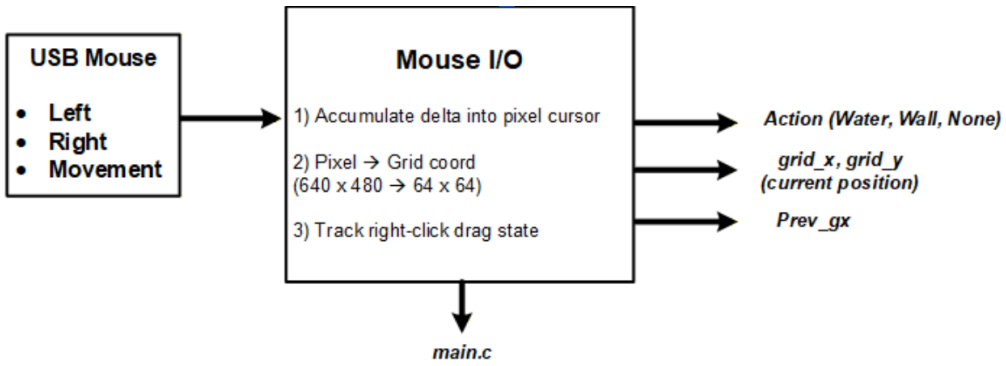

Figure 9: Mouse I/O datapath. Raw mouse movement is accumulated into a pixel cursor, converted to grid coordinates, and combined with button state to produce the action and drag metadata consumed by `main.c`.

`mouse_init()` Opens `/dev/input/mice`, initializes the pixel-space cursor to the center of the 640×480 screen, and clears any previous right-drag state.

`mouse_poll()` Blocks until one 3-byte mouse packet arrives. It treats the X and Y bytes as signed deltas, accumulates them into the pixel cursor, clamps the cursor to the visible screen, and scales it into a 64×64 grid coordinate pair `grid_x`, `grid_y`.

Action mapping The button bits are translated into simulator actions. Left click becomes `MOUSE_ADD_WATER`; right click becomes `MOUSE_DRAW_WALL`; no relevant button becomes `MOUSE_NONE`. The implementation also supports middle click as `MOUSE_ERASE`, which is omitted from the simplified diagram.

Drag metadata For right-click wall drawing, the module stores the previous grid position and returns it as `prev_gx`, `prev_gy`. `main.c` uses the previous and current points to draw a Bresenham line of wall cells, so fast mouse movement still produces continuous walls.

This separation keeps the runtime loop in Section 6.2.2 clean: `main.c` only sees grid-space events and does not need to parse PS/2 packet bits, track screen coordinates, or remember drag state itself.

7 Hardware Implementation (FPGA)

7.1 Overview

The FPGA side implements the parts of the simulator that must run every frame: the liquid update engine, the on-chip memories that hold simulation state, and the VGA renderer. At the top level, `water_simulator` combines the Avalon-MM register interface, the Global Controller FSM, the Liquid Flow Processing Element, Cell BRAM, Diffs BRAM, and the VGA output path. The HPS sends setup and brush commands through registers, while the FPGA performs the repeated evaluate and commit passes locally.

The hardware is organized around the two-pass update scheme from Section 3.3. During evaluate, the controller reads a stable snapshot of the cell grid, runs the combinational flow PE, and accumulates signed deltas in Diffs BRAM. During commit, it applies those deltas back into Cell BRAM and clears the scratch entries for the next tick. The VGA renderer reads the same cell state to display the grid, so the design also has to arbitrate memory access between simulation updates and video output.

7.2 Global Controller FSM

The Global Controller FSM is the scheduler for the FPGA-side simulator. It sits between the Avalon-MM register interface, the Cell BRAM, the Diffs BRAM, the flow-rule PE, and the VGA renderer. In the idle state, the HPS owns Cell BRAM Port A, so software can initialize `GRID_MEM` or issue brush edits. When software pulses a control bit, or when auto-run is enabled and the next frame boundary arrives, the controller takes ownership of the memory ports and runs the requested hardware operation.

FSM state	Role in the simulator
`S_IDLE`	Releases Cell BRAM Port A to the HPS, clears `BUSY`, and waits for `CTRL_RESET`, `CTRL_LOAD_MAP`, `CTRL_BRUSH_APPLY`, a manual step pulse, or an auto-run step.
`S_RESET_SWEEP`	Sweeps all 4096 cells, preserves each cell's `CellType`, clears liquid/bookkeeping fields, and clears the corresponding Diffs BRAM entry.
`S_MAP_LOAD`	Marks the current software-written map as ready by asserting `STATUS_MAP_READY`. In the present RTL, the wall information already lives in the packed Cell BRAM word.
`S_BRUSH_READ`, `S_BRUSH_LATCH`, `S_BRUSH_WRITE`	Performs a small read-modify-write operation for one mouse brush command. Add-water saturates the liquid amount, erase clears liquid, and wall writes a solid cell.
`S_EVALUATE`	Sweeps the grid in raster order, reads the current cell and its neighbors, runs the flow-rule PE, and accumulates the resulting signed deltas into Diffs BRAM.
`S_COMMIT`	Sweeps the grid again, adds each stored diff to the cell's liquid field, writes the updated packed cell word back to Cell BRAM, and clears the diff entry.
`S_VBLANK_WAIT`	Holds completion until the next VSYNC edge when frame-lock is enabled, reducing the chance that VGA reads the grid midway through an update.

Table 3: Global Controller FSM states.

The most important runtime sequence is `S_EVALUATE` followed by `S_COMMIT`. In `S_EVALUATE`, the controller maintains both a two-dimensional cursor (p_x, p_y) and a flattened cell index. For each cell, it issues a Cell BRAM read for the center cell and a Diffs BRAM read for the center diff, then uses Cell BRAM Port B to fetch the bottom, left, and right neighbor words. Boundary flags such as `has_bot_lat` and `has_left_lat` are generated from the cursor position so the PE does not flow outside the 64×64 grid.

After the PE output is valid, the controller registers the PE result before writing it back. This extra pipeline register shortens the critical path from the cell-word registers through the PE logic to the memory write signals. The self-diff is accumulated first at the current cell address. Neighbor diffs are then accumulated through read-modify-write cycles. Since the current Diffs BRAM is single-port, the controller cannot read one diff entry and write another in the same cycle; therefore, bottom, left, and right neighbor updates are serialized using the main `phase` counter and the small `nb_phase` epilogue.

7.3 Liquid Flow Processing Element (Particle Accelerator)

The Processing Element (PE) is the combinational core of the physics engine. Given one cell and the liquid amounts of its four orthogonal neighbors, the PE computes the four flow rules described in Section 3.2 and outputs five signed Q2.16 deltas: one for the cell itself (`diff_self`) and one for each neighbor that received liquid (`diff_bottom`, `diff_left`, `diff_right`, `diff_top`). The PE never writes to the cell BRAM directly; its outputs feed the shared `Diffs` scratch buffer, and the separate commit pass applies them (Section 3.3).

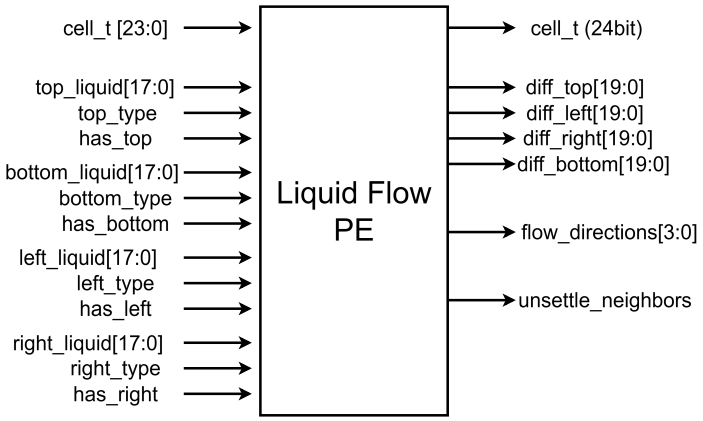

Figure 10: Block-level I/O diagram of liquid flow processing element. The center cell and each neighbor's liquid, type, and boundary-valid inputs are entered on the left. The diagram labels the four neighbor diffs on the right; the RTL port list also includes `diff_self`, which records the corresponding loss from the source cell.

Interface

Table 4 summarizes the PE port list. The `has_x` flags are boundary guards: the Global Controller deasserts them when the cell sits on the corresponding edge of the 64×64 grid, which causes the PE to skip the rule for that direction entirely.

Port	Width / Type	Description
`cell_in`	24-bit struct	Packed center cell (`CellType`, `Liquid`, `Settled`, `SettleCount`).
`top_liquid`, `bottom_liquid`, `left_liquid`, `right_liquid`	18 b unsigned	Neighbor liquid amounts (Q2.16).
`top_type` … `right_type`	1 b enum	`CELL_BLANK` or `CELL_SOLID`.
`has_top` … `has_right`	1 b	Boundary valid flags (0 at grid edges).
`cell_out`	24-bit struct	Updated bookkeeping fields (`Settled`, `SettleCount`); `Liquid` unchanged.
`diff_self`, `diff_top`, `diff_bottom`, `diff_left`, `diff_right`	20 b signed	Signed Q2.16 flow deltas for the `Diffs` BRAM.
`flow_directions`	4 b	One-hot mask of directions in which flow occurred.
`unsettle_neighbors`	1 b	Asserted when any flow moved; the controller clears the `Settled` flag on all four neighbors.

Table 4: Liquid Flow PE port summary.

Datapath

The PE is purely combinational. All arithmetic uses a 20-bit signed `flow_t` type (two guard bits above the 18-bit `liquid_t`) to accommodate negative self-deltas without overflow. The four rules execute sequentially within a single `always_comb` block, each one decrementing a running `remaining` register:

1. **Rule 1 — Downward (gravity).** If `has_bottom` and the bottom cell is `CELL_BLANK`, compute the vertical equilibrium target $V(r, d)$ (Equation 1), subtract the bottom cell's current level, clamp via `clamp_positive_flow`, and scale by `FlowSpeed`. The result goes to `diff_bottom`; `remaining` and `diff_self` are decremented by the same amount.

2. **Rule 2a — Spread left.** Compute $\delta = \texttt{remaining} - d_\ell$ and divide by 4 via a two-bit arithmetic right-shift (`fp_div4`), costing zero DSP blocks. Clamp and apply as before.

3. **Rule 2b — Spread right.** Same structure as Rule 2a. When the parameter `USE_RIGHT_DIV3` is set (the default), the divisor is 3 instead of 4, implemented as a multiplication by the reciprocal constant `INV3` $= \lceil 2^{16}/3 \rceil = 21\,845$. This consumes one 18×18 DSP multiplier and faithfully reproduces the slight rightward drift present in the Unity reference. Setting `USE_RIGHT_DIV3 = 0` substitutes a free shift, saving one DSP block at the cost of symmetric spreading.

4. **Rule 3 — Upward (pressure).** Reuses the same `vertical_target` function as Rule 1: $\phi_{\text{top}} = \texttt{remaining} - V(\texttt{remaining}, d_{\text{top}})$. Because V is shared combinational logic, Rules 1 and 3 together need only two instances of the three-case mux and the `INV1P25` multiplier.

After all four rules, the PE runs a settling heuristic. If no flow occurred and `remaining` equals the original `start_value`, `SettleCount` is incremented (saturating at 15). When `SettleCount` reaches the threshold of 10, `Settled` is set, and future ticks skip this cell entirely until a neighbor's flow clears the flag.

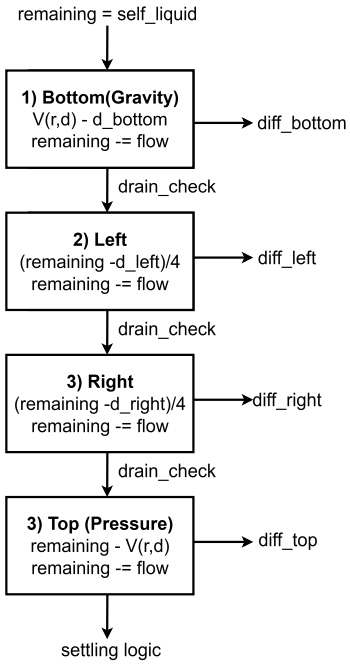

Figure 11: PE internal dataflow. The `remaining` register threads through bottom gravity, left spread, right spread, and top pressure in series. The right-spread box should be read as Rule 2b and the top-pressure box as Rule 3; each stage subtracts its clamped flow from `remaining` and adds it to the corresponding neighbor diff.

Fixed-Point Arithmetic

The package `liquid_flow_pkg` (also duplicated as `flow_rules_pkg`) defines all shared arithmetic. Key functions:

Function	Implementation	DSP cost
`fp_mul(a,b)`	$a \times b \gg 16$ (40-bit intermediate)	1 DSP block
`fp_div2(v)`	$v1$ (arithmetic right-shift)	0
`fp_div4(v)`	$v2$	0
`fp_div3(v)`	`fp_mul(v, INV3)` where INV3 = 21 845	1 DSP block
`clamp_flow`	max(0, min(*candidate*, *maxflow*, *remaining*))	0 (comparators)
`vertical_target`	3-case mux using `fp_mul` and `fp_div2`	shared with Rules 1 & 3

7.4 BRAM Organization

The FPGA design stores the simulation state in two on-chip BRAM structures inside `water_simulator`: the persistent `Cell BRAM` and the per-tick `Diffs BRAM`. Separating these memories is what makes the two-pass update scheme practical. During the evaluate pass, the physics logic reads a stable snapshot from Cell BRAM and writes only signed changes into Diffs BRAM. During the commit pass, the controller combines each cell value with its accumulated diff, writes the updated cell back, and clears the diff entry for the next tick.

Cell BRAM

Cell BRAM is the main grid memory. It is implemented as a true dual-port synchronous memory with 4096 entries, one for each cell in the 64×64 simulation grid. Each entry is a 32-bit packed cell word, so the raw storage cost is $4096 \times 32 = 131{,}072$ bits. The word contains `CellType`, the 18-bit Q2.16 `Liquid` amount, `Settled`, `SettleCount`, `isDownFlowing`, and reserved bits, matching Table 2.

Port A is shared between the HPS Avalon-MM interface and the physics controller. When the FSM is idle, `hps_owns_porta` is asserted, so software can initialize or inspect `GRID_MEM`. During reset, evaluate, commit, and brush operations, the physics controller owns Port A and uses it for cell reads and writes. Port B is read-only in this design. It normally feeds the VGA renderer, but the physics FSM temporarily borrows it during evaluate phases to read neighboring cells without needing extra copies of the grid.

Diffs BRAM

Diffs BRAM is the scratch buffer for one simulation tick. It has the same 4096-entry address space as Cell BRAM, but each entry is a 20-bit signed Q2.16 delta, giving a raw storage cost of $4096 \times 20 = 81{,}920$ bits. The extra signed guard bits let the design represent negative self-deltas such as `diff_self` without overflowing the 18-bit unsigned liquid format.

The evaluate pass accumulates PE outputs into this buffer. For example, if liquid moves from cell (x, y) into its bottom neighbor, the controller adds a negative delta to `Diffs`$[x, y]$ and a positive delta to the bottom neighbor's diff entry. Because the current RTL uses a single-port `diff_bram`, neighbor updates are done as read-modify-write sequences across multiple phases. The commit pass then reads `Cell BRAM`$[i]$ and `Diffs BRAM`$[i]$, computes the new liquid amount, writes the packed cell word back to Cell BRAM, and clears `Diffs BRAM`$[i]$ to zero.

7.5 VGA Renderer

The VGA renderer (`vga_renderer.sv`) is the SystemVerilog module responsible for converting the contents of the cell grid into a continuous 640×480 VGA signal. It reads one 32-bit cell word per pixel from its BRAM read port, decodes the cell's type and liquid amount into an RGB color, and drives the timing signals expected by the DE1-SoC's ADV7123 video DAC. The module exposes 11 ports: `clk`, `reset`, a paired BRAM read interface (`grid_rd_addr`/`grid_rd_data`), and the eight VGA output signals (RGB and the five sync/timing lines). The renderer has no concept of where in the FPGA the cell BRAM lives; address and data are wired at the top level (Figure 12).

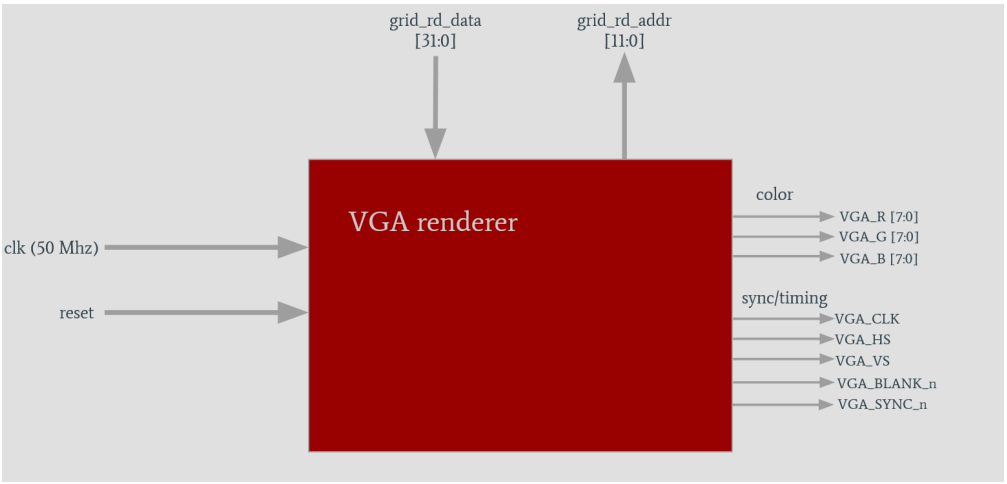

Figure 12: Port interface of the VGA renderer module.

7.5.1 Screen Geometry

The renderer's first design choice is how to map the 64×64 grid onto a 640×480 screen. We render each cell as a 7×7 pixel block: 64 cells × 7 pixels = 448, which fits inside the 480-pixel vertical with a 16-pixel border above and below. Seven is the largest block size for which $64N \leq 480$; an 8-pixel block would overflow. Horizontally, the same 448-pixel sim area is centered in the 640-pixel width with 96-pixel borders on each side (Figure 13).

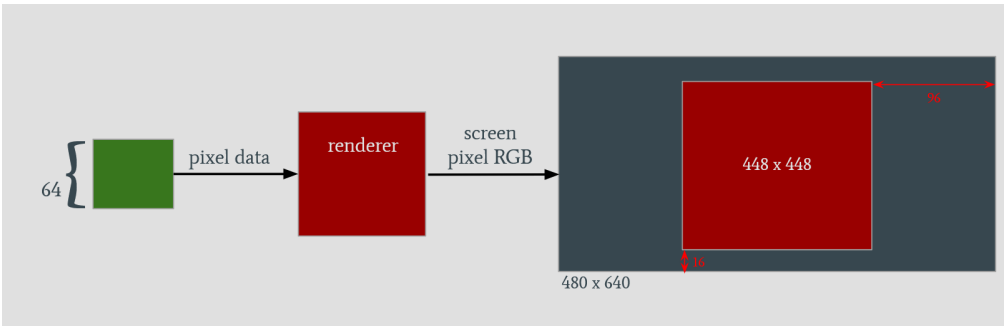

Figure 13: Mapping a 64×64 cell grid into a 448×448 region centered on a 640×480 screen.

7.5.2 Counter Walking for Divide-by-Seven

Computing the cell index from the pixel position naively would require `cell_x = pixel_x / 7`, a non-power-of-two integer divide that would either need a multi-cycle iterative divider or a wide LUT cascade in hardware. The renderer avoids the divide entirely by walking two counters in parallel: `pix_in_cell_x` cycles through $0 \to 6 \to 0$, and on each rollover, `cell_x` increments by one (Figure 14). The rollover *is* the division. In synthesis, this reduces to one comparator and one adder per axis — no DSPs, no iterative state.

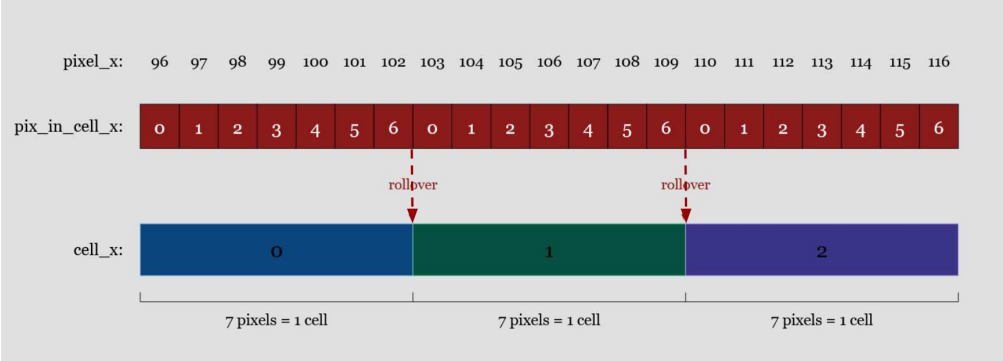

Figure 14: Counter walking: `pix_in_cell_x` sawtooths from 0 to 6, and each rollover increments `cell_x`.

References

[1] BroMayo. unity-dots-ca-watersim: A cellular automata water simulation using Unity DOTS. https://github.com/BroMayo/unity-dots-ca-watersim, 2023.

A Appendix - Source Code Listings

A.1 RTL(.sv) codes (Hardware(FPGA))

```
// ============================================================================
// liquid_flow_main.sv  ---  Top-level Avalon-MM water simulator peripheral
//
// Instantiates:
//   • vga_renderer    (owns cell BRAM Port B; has internal vga_counters)
//   • flow_rules_v2   (combinational physics PE; imported pkg = flow_rules_pkg_v2)
//   • cell_bram       (true dual-port, 4096 × 32-bit)
//   • diff_bram       (single-port, 4096 × 20-bit signed)
//   • Global Controller FSM
//
// Avalon-MM register map  (word address = byte_offset >> 2)
//   0x0000  CTRL_REG       [0]=step [1]=reset [2]=load_map
//                          [3]=done_ack [4]=brush_apply
//   0x0001  STATUS_REG     [0]=busy [1]=done [2]=map_ready [4:3]=phase
//   0x0002  GRID_SIZE_REG  RO {16'd64, 16'd64}
//   0x0003  MOUSE_POS_REG  [15:0]=grid_x  [31:16]=grid_y
//   0x0004  BRUSH_CFG_REG  [1:0]=tool [15:8]=radius [31:16]=liquid_lo16
//   0x0005  STEP_CFG_REG   [0]=auto_run [1]=frame_lock [15:8]=steps_per_frame
//   0x0006  TICK_COUNT_REG RO
//   0x0010  VGA_CTRL_REG   [0]=vga_enable  (byte 0x0040 >> 2)
//   0x0400--0x13FF GRID_MEM  4096 × 32-bit cell words
//
// Evaluate FSM timing (9 cycles per cell):
//   Ph0 : issue Port-A read (center word) + diff-BRAM read (center diff)
//   Ph1 : latch center word & diff; issue Port-B read → bottom neighbor
//   Ph2 : latch bottom word;        issue Port-B read → left  neighbor
//   Ph3 : latch left word;          issue Port-B read → right neighbor
//   Ph4 : latch right word; PE combinational result valid; REGISTER PE outputs
//   Ph5 : PE outputs stable (registered); write cell_word_out_r (Port-A)
//         write diff_self_r accumulated with center_diff (diff BRAM)
//   Ph6 : diff RMW for bottom neighbor  (uses pe_diff_bottom_r)
//   Ph7 : diff RMW for left  neighbor   (uses pe_diff_left_r)
//   nb_epilogue: diff RMW for right neighbor (uses pe_diff_right_r); advance cell
//
// The one-cycle pipeline register on PE outputs (Ph4→Ph5) breaks the
// critical path:
//   Before: center_word regs → 689 LUTs of PE logic → diff/cell write regs
//           (~42 ns, Fmax 23 MHz)
//   After:  center_word regs → PE logic → pe_*_r regs  (half path, ~20 ns)
//           pe_*_r regs → write regs                   (trivial, ~1 ns)
// ============================================================================

`default_nettype none

module water_simulator
  import flow_rules_pkg_v2::*;
(
  input  logic        clk,
  input  logic        reset,

  // Avalon-MM slave (32-bit data, 14-bit word address)
  input  logic [13:0] address,
  input  logic        write,
  input  logic        read,
  input  logic        chipselect,
  input  logic [31:0] writedata,
  output logic [31:0] readdata,

  // VGA outputs (driven by vga_renderer)
  output logic [7:0]  VGA_R, VGA_G, VGA_B,
  output logic        VGA_CLK, VGA_HS, VGA_VS,
  output logic        VGA_BLANK_n, VGA_SYNC_n
);

  // ------------------------------------------------------------------------
  // Local parameters
  // ------------------------------------------------------------------------
  localparam int GRID_N  = 64 * 64;  // 4096
  localparam int ADDR_W  = 12;       // log2(4096)

  // Avalon word addresses
  localparam logic [13:0] WA_CTRL       = 14'h0000;
  localparam logic [13:0] WA_STATUS     = 14'h0001;
  localparam logic [13:0] WA_GRID_SIZE  = 14'h0002;
  localparam logic [13:0] WA_MOUSE_POS  = 14'h0003;
  localparam logic [13:0] WA_BRUSH_CFG  = 14'h0004;
  localparam logic [13:0] WA_STEP_CFG   = 14'h0005;
  localparam logic [13:0] WA_TICK_COUNT = 14'h0006;
  localparam logic [13:0] WA_VGA_CTRL   = 14'h0010;  // 0x40 >> 2
```

```
  localparam logic [13:0] WA_GRID_BASE  = 14'h0400;  // 0x1000 >> 2
  localparam logic [13:0] WA_GRID_TOP   = 14'h13FF;  // 0x4FFC >> 2

  // ------------------------------------------------------------------------
  // Control / config registers
  // ------------------------------------------------------------------------
  logic       ctrl_step, ctrl_reset, ctrl_load_map,
              ctrl_done_ack, ctrl_brush_apply;

  logic        auto_run;
  logic        frame_lock;
  logic [7:0]  steps_per_frame;
  logic [5:0]  mouse_gx, mouse_gy;
  logic [1:0]  brush_tool;     // 01=add 10=erase 11=wall
  logic [7:0]  brush_radius;
  logic [15:0] brush_liquid;   // low 16 bits of Q2.16 amount
  logic        vga_enable;

  // Status (written by FSM)
  logic        status_busy, status_done, status_map_ready;
  logic [1:0]  status_phase;
  logic [31:0] tick_count;

  // HPS owns Port A when the FSM is idle
  logic        hps_owns_porta;

  // ------------------------------------------------------------------------
  // Avalon register writes
  // ------------------------------------------------------------------------
  always_ff @(posedge clk) begin
    ctrl_step        <= '0;
    ctrl_reset       <= '0;
    ctrl_load_map    <= '0;
    ctrl_done_ack    <= '0;
    ctrl_brush_apply <= '0;

    if (reset) begin
      auto_run        <= '0;
      frame_lock      <= '0;
      steps_per_frame <= 8'd1;
      mouse_gx        <= '0;
      mouse_gy        <= '0;
      brush_tool      <= '0;
      brush_radius    <= '0;
      brush_liquid    <= '0;
      vga_enable      <= '0;
    end else if (chipselect && write) begin
      case (address)
        WA_CTRL: begin
          ctrl_step        <= writedata[0];
          ctrl_reset       <= writedata[1];
          ctrl_load_map    <= writedata[2];
          ctrl_done_ack    <= writedata[3];
          ctrl_brush_apply <= writedata[4];
        end
        WA_MOUSE_POS: begin
          mouse_gx <= writedata[5:0];
          mouse_gy <= writedata[21:16];
        end
        WA_BRUSH_CFG: begin
          brush_tool   <= writedata[1:0];
          brush_radius <= writedata[15:8];
          brush_liquid <= writedata[31:16];
        end
        WA_STEP_CFG: begin
          auto_run        <= writedata[0];
          frame_lock      <= writedata[1];
          steps_per_frame <= writedata[15:8];
        end
        WA_VGA_CTRL: begin
          vga_enable <= writedata[0];
        end
        default: ;
      endcase
    end
  end

  // ------------------------------------------------------------------------
  // Avalon register reads
  // ------------------------------------------------------------------------
  always_comb begin
    readdata = '0;
    if (chipselect && read) begin
      if (address >= WA_GRID_BASE && address <= WA_GRID_TOP)
```

```
      readdata = cell_a_rdata;
    else case (address)
      WA_STATUS:     readdata = {27'h0, status_phase,
                                 status_map_ready, status_done, status_busy};
      WA_GRID_SIZE:  readdata = {16'd64, 16'd64};
      WA_TICK_COUNT: readdata = tick_count;
      default:       readdata = '0;
    endcase
  end
end

// ---------------------------------------------------------------------
// Cell BRAM --- true dual-port, 4096 × 32-bit
//   Port A : physics FSM (r/w)  OR  HPS Avalon (r/w)
//   Port B : vga_renderer (read-only, address driven below)
// ---------------------------------------------------------------------
logic [ADDR_W-1:0] cell_a_addr;
logic [31:0]       cell_a_wdata;
logic              cell_a_we;
logic [31:0]       cell_a_rdata;

logic [ADDR_W-1:0] cell_b_addr;
logic [31:0]       cell_b_rdata;

// HPS-side Port-A signals
logic [ADDR_W-1:0] hps_addr;
logic [31:0]       hps_wdata;
logic              hps_we;
assign hps_addr  = address[ADDR_W-1:0] - WA_GRID_BASE[ADDR_W-1:0];
assign hps_wdata = writedata;
assign hps_we    = chipselect && write
                   && (address >= WA_GRID_BASE)
                   && (address <= WA_GRID_TOP);

// Physics FSM Port-A signals (driven by comb block below)
logic [ADDR_W-1:0] phys_addr;
logic [31:0]       phys_wdata;
logic              phys_we;

// Port-A mux
always_comb begin
  if (hps_owns_porta) begin
    cell_a_addr  = hps_addr;
    cell_a_wdata = hps_wdata;
    cell_a_we    = hps_we;
  end else begin
    cell_a_addr  = phys_addr;
    cell_a_wdata = phys_wdata;
    cell_a_we    = phys_we;
  end
end

cell_bram #(.DEPTH(GRID_N), .WIDTH(32)) u_cell_bram (
  .clk    (clk),
  .a_addr (cell_a_addr), .a_we (cell_a_we),
  .a_din  (cell_a_wdata), .a_dout (cell_a_rdata),
  .b_addr (cell_b_addr),  .b_we (1'b0),
  .b_din  (32'h0),        .b_dout (cell_b_rdata)
);

// ---------------------------------------------------------------------
// Diff BRAM --- single-port, 4096 × 20-bit signed
// ---------------------------------------------------------------------
logic [ADDR_W-1:0] diff_addr;
logic [19:0]       diff_wdata;
logic              diff_we;
logic [19:0]       diff_rdata;

diff_bram #(.DEPTH(GRID_N), .WIDTH(20)) u_diff_bram (
  .clk  (clk),
  .addr (diff_addr), .we (diff_we),
  .din  (diff_wdata), .dout (diff_rdata)
);

// ---------------------------------------------------------------------
// VGA renderer
// ---------------------------------------------------------------------
logic [11:0] renderer_rd_addr;
logic        vga_vs_out;

vga_renderer u_renderer (
  .clk          (clk),
  .reset        (reset),
  .grid_rd_addr (renderer_rd_addr),
```

```
    .grid_rd_data (cell_b_rdata),
    .VGA_R        (VGA_R),    .VGA_G  (VGA_G),    .VGA_B  (VGA_B),
    .VGA_CLK      (VGA_CLK), .VGA_HS (VGA_HS),  .VGA_VS (vga_vs_out),
    .VGA_BLANK_n  (VGA_BLANK_n),
    .VGA_SYNC_n   (VGA_SYNC_n)
  );
  assign VGA_VS = vga_vs_out;

  logic vs_prev, vsync_fall;
  always_ff @(posedge clk) vs_prev <= vga_vs_out;
  assign vsync_fall = vs_prev & ~vga_vs_out;

  // ------------------------------------------------------------------------
  // flow_rules_v2 PE  (purely combinational)
  // ------------------------------------------------------------------------
  cell_word_t pe_cell_word_in;
  logic       pe_has_bottom, pe_has_left, pe_has_right;
  cell_word_t pe_bottom_word, pe_left_word, pe_right_word;

  // Combinational PE outputs
  flow_t      pe_diff_self, pe_diff_bottom, pe_diff_left, pe_diff_right;
  cell_word_t pe_cell_word_out;
  flow_t      pe_remaining_out;
  logic       pe_flow_happened;

  // ------------------------------------------------------------------------
  // Pipeline registers on PE outputs --- break the critical path
  // Registered on the rising edge at phase 4 (when all inputs are stable).
  // Used by the write phases (5, 6, 7, epilogue) one cycle later.
  // ------------------------------------------------------------------------
  flow_t      pe_diff_self_r, pe_diff_bottom_r,
              pe_diff_left_r, pe_diff_right_r;
  cell_word_t pe_cell_word_out_r;
  flow_t      center_diff_r;   // also pipeline center_diff to stay aligned

  always_ff @(posedge clk) begin
    if (state == S_EVALUATE && !in_nb_epilogue && phase == 3'd4) begin
      pe_diff_self_r     <= pe_diff_self;
      pe_diff_bottom_r   <= pe_diff_bottom;
      pe_diff_left_r     <= pe_diff_left;
      pe_diff_right_r    <= pe_diff_right;
      pe_cell_word_out_r <= pe_cell_word_out;
      center_diff_r      <= center_diff;
    end
  end

  flow_rules_v2 #(.USE_RIGHT_DIV3(1'b1)) u_pe (
    .cell_word_in   (pe_cell_word_in),
    .has_bottom     (pe_has_bottom),
    .has_left       (pe_has_left),
    .has_right      (pe_has_right),
    .bottom_word_in (pe_bottom_word),
    .left_word_in   (pe_left_word),
    .right_word_in  (pe_right_word),
    .diff_self      (pe_diff_self),
    .diff_bottom    (pe_diff_bottom),
    .diff_left      (pe_diff_left),
    .diff_right     (pe_diff_right),
    .cell_word_out  (pe_cell_word_out),
    .remaining_out  (pe_remaining_out),
    .flow_happened  (pe_flow_happened)
  );

  // ------------------------------------------------------------------------
  // FSM state encoding
  // ------------------------------------------------------------------------
  typedef enum logic [3:0] {
    S_IDLE        = 4'd0,
    S_RESET_SWEEP = 4'd1,
    S_MAP_LOAD    = 4'd2,
    S_EVALUATE    = 4'd3,
    S_COMMIT      = 4'd4,
    S_VBLANK_WAIT = 4'd5,
    S_BRUSH_READ  = 4'd6,
    S_BRUSH_LATCH = 4'd7,
    S_BRUSH_WRITE = 4'd8
  } state_t;

  state_t state;

  // ------------------------------------------------------------------------
  // Datapath registers
  // ------------------------------------------------------------------------
  logic [5:0]  px, py;
```

```
  logic [11:0] flat;
  assign flat = {py, px};

  logic [11:0] sweep_idx;
  logic [7:0]  step_cnt;
  logic [2:0]  phase;          // 0--7 within evaluate; 0--2 within commit
  logic        in_nb_epilogue; // right-diff epilogue after phase 7

  // Latched neighbor words
  cell_word_t  center_word;
  flow_t       center_diff;
  cell_word_t  bottom_word_lat, left_word_lat, right_word_lat;
  logic        has_bot_lat, has_left_lat, has_right_lat;

  // Registered flat address for the cell being written in Ph5
  // (flat may have advanced if we were to start next cell --- keep a copy)
  logic [11:0] flat_r;   // registered copy of flat, captured at Ph4

  // Latched for commit pass
  cell_word_t  commit_word;
  flow_t       commit_diff;

  // Latched for brush ADD_WATER read-modify-write
  cell_word_t  brush_read_word;

  // ------------------------------------------------------------------------
  // Neighbor flat addresses (combinational)
  // ------------------------------------------------------------------------
  logic [11:0] addr_bot, addr_left, addr_right;
  always_comb begin
    addr_bot   = (py < 63) ? {py + 6'd1, px}   : flat;
    addr_left  = (px > 0)  ? {py, px - 6'd1}   : flat;
    addr_right = (px < 63) ? {py, px + 6'd1}   : flat;
  end

  // Registered neighbor addresses --- captured at Ph4, used in Ph5/6/7/epilogue
  logic [11:0] addr_bot_r, addr_left_r, addr_right_r;
  logic        has_bot_r, has_left_r, has_right_r;

  always_ff @(posedge clk) begin
    if (state == S_EVALUATE && !in_nb_epilogue && phase == 3'd4) begin
      flat_r       <= flat;
      addr_bot_r   <= addr_bot;
      addr_left_r  <= addr_left;
      addr_right_r <= addr_right;
      has_bot_r    <= has_bot_lat;
      has_left_r   <= has_left_lat;
      has_right_r  <= has_right_lat;
    end
  end

  // ------------------------------------------------------------------------
  // Wire PE inputs from latched registers
  // ------------------------------------------------------------------------
  assign pe_cell_word_in = center_word;
  assign pe_has_bottom   = has_bot_lat;
  assign pe_has_left     = has_left_lat;
  assign pe_has_right    = has_right_lat;
  assign pe_bottom_word  = bottom_word_lat;
  assign pe_left_word    = left_word_lat;
  assign pe_right_word   = right_word_lat;

  // ------------------------------------------------------------------------
  // Physics Port-A and diff BRAM combinational control
  // ------------------------------------------------------------------------
  always_comb begin
    phys_addr  = flat;
    phys_wdata = '0;
    phys_we    = 1'b0;
    diff_addr  = flat;
    diff_wdata = '0;
    diff_we    = 1'b0;

    case (state)

      // ---- RESET: preserve CellType, zero everything else ----
      S_RESET_SWEEP: begin
        phys_addr  = sweep_idx;
        phys_wdata = {31'h0, cell_a_rdata[0]};
        phys_we    = (phase == 3'd1);
        diff_addr  = sweep_idx;
        diff_wdata = '0;
        diff_we    = (phase == 3'd1);
      end
```

```
416
417       // ---- EVALUATE ----
418       S_EVALUATE: begin
419         if (!in_nb_epilogue) begin
420           case (phase)
421             // Ph0: issue reads
422             3'd0: begin
423               phys_addr = flat;
424               diff_addr = flat;
425             end
426
427             // Ph1--Ph4: neighbor reads; Port-A / diff idle
428             3'd1, 3'd2, 3'd3, 3'd4: begin
429               phys_addr = flat;
430               diff_addr = flat;
431             end
432
433             // Ph5: PE outputs are now registered and stable.
434             //   • Write updated cell word back via Port A (use flat_r)
435             //   • Write diff_self accumulated with center_diff (use flat_r)
436             3'd5: begin
437               phys_addr  = flat_r;
438               phys_wdata = pe_cell_word_out_r;
439               phys_we    = 1'b1;
440               diff_addr  = flat_r;
441               diff_wdata = flow_t'($signed(center_diff_r) + pe_diff_self_r);
442               diff_we    = 1'b1;
443             end
444
445             // Ph6: RMW bottom neighbor diff
446             // diff_rdata holds bottom's current value (read presented at Ph5
447             // via diff_addr = addr_bot_r; but we wrote flat_r in Ph5.
448             // Since addr_bot_r ≠ flat_r (different cells), the BRAM read
449             // in Ph6 sees the address set in the PREVIOUS cycle (Ph5).
450             // We need to set diff_addr = addr_bot_r in Ph5 for the READ,
451             // but we also need to write flat_r in Ph5.
452             // Resolution: diff BRAM is single-port. We can't read addr_bot_r
453             // and write flat_r simultaneously. Instead:
454             //   Ph5: write flat_r (diff_we=1, diff_addr=flat_r)
455             //        diff_rdata next cycle = flat_r's OLD value (read-first,
456             //        but we are writing flat_r so output is flat_r's old val)
457             //   Ph6: set diff_addr=addr_bot_r (no write) → read arrives Ph7
458             //   Ph7: write addr_bot_r with (diff_rdata + pe_diff_bottom_r)
459             //   Ph8 (epilogue first cycle): set diff_addr=addr_left_r → read Ph_epi1
460             //   Ph_epi1: write addr_left_r
461             //   Ph_epi2: set diff_addr=addr_right_r → read Ph_epi3...
462             //
463             // That extends the epilogue. Simpler: use separate read cycle
464             // before each write. Phases 6/7 become read/write pairs for bot,
465             // then epilogue handles left (read+write) and right (read+write).
466             // This makes each cell take 11 cycles but is correct.
467             //
468             // CHOSEN: Ph6 = issue bot diff READ (addr only, no write)
469             3'd6: begin
470               diff_addr = addr_bot_r;
471               // No write --- just pre-issue the read
472             end
473
474             // Ph7: write bot diff (diff_rdata now has bot's current value)
475             3'd7: begin
476               diff_addr  = addr_bot_r;
477               diff_wdata = flow_t'($signed(flow_t'(diff_rdata)) + pe_diff_bottom_r);
478               diff_we    = has_bot_r;
479             end
480
481             default: ;
482           endcase
483
484         end else begin
485           // nb_epilogue sub-phases for left and right diffs
486           // We use nb_phase to track which sub-step we're in
487           case (nb_phase)
488             2'd0: begin
489               // Issue left diff READ
490               diff_addr = addr_left_r;
491             end
492             2'd1: begin
493               // Write left diff
494               diff_addr  = addr_left_r;
495               diff_wdata = flow_t'($signed(flow_t'(diff_rdata)) + pe_diff_left_r);
496               diff_we    = has_left_r;
497             end
498             2'd2: begin
499               // Issue right diff READ
```

```
            diff_addr = addr_right_r;
          end
          2'd3: begin
            // Write right diff; cell advance happens in sequential block
            diff_addr  = addr_right_r;
            diff_wdata = flow_t'($signed(flow_t'(diff_rdata)) + pe_diff_right_r);
            diff_we    = has_right_r;
          end
          default: ;
        endcase
      end
    end

    // ---- COMMIT ----
    S_COMMIT: begin
      case (phase)
        3'd0: begin
          phys_addr = sweep_idx;
          diff_addr = sweep_idx;
        end
        3'd1: begin
          phys_addr = sweep_idx;
          diff_addr = sweep_idx;
        end
        3'd2: begin
          phys_addr = sweep_idx;
          begin
            flow_t new_liq;
            new_liq = $signed(liq_to_flow(word_liquid(commit_word)))
                    + $signed(commit_diff);
            if ($signed(new_liq) < $signed(MIN_LIQUID))
              phys_wdata = zero_liquid(commit_word);
            else
              phys_wdata = pack_cell(
                             word_cell_type(commit_word),
                             liquid_t'(new_liq[LIQUID_W-1:0]),
                             word_settled(commit_word),
                             word_settle_count(commit_word),
                             1'b0
                           );
          end
          phys_we   = 1'b1;
          diff_addr  = sweep_idx;
          diff_wdata = '0;
          diff_we    = 1'b1;
        end
        default: ;
      endcase
    end

    // ---- BRUSH ----
    S_BRUSH_READ, S_BRUSH_LATCH: begin
      phys_addr = {mouse_gy, mouse_gx};
    end

    S_BRUSH_WRITE: begin
      phys_addr = {mouse_gy, mouse_gx};
      phys_we   = 1'b1;
      case (brush_tool)
        2'b01: begin  // ADD_WATER
          logic [19:0] sum_liq;
          sum_liq    = {2'b0, word_liquid(brush_read_word)}
                     + {4'b0, brush_liquid};
          phys_wdata = pack_cell(
                         CELL_BLANK,
                         (sum_liq > 20'(MAX_VALUE))
                           ? liquid_t'(MAX_VALUE[LIQUID_W-1:0])
                           : liquid_t'(sum_liq[LIQUID_W-1:0]),
                         word_settled(brush_read_word),
                         word_settle_count(brush_read_word),
                         1'b0
                       );
        end
        2'b10: begin  // ERASE
          phys_wdata = zero_liquid(brush_read_word);
        end
        2'b11: begin  // WALL
          phys_wdata = pack_cell(CELL_SOLID, '0, 1'b1, '0, 1'b0);
        end
        default: phys_we = 1'b0;
      endcase
    end

    default: ;
```

```
    endcase
  end

  // ---------------------------------------------------------------------
  // nb_phase: 2-bit sub-counter for left/right diff epilogue
  // ---------------------------------------------------------------------
  logic [1:0] nb_phase;

  // ---------------------------------------------------------------------
  // FSM sequential logic
  // ---------------------------------------------------------------------
  always_ff @(posedge clk) begin
    if (reset) begin
      state          <= S_IDLE;
      status_busy    <= '0;
      status_done    <= '0;
      status_map_ready <= '0;
      status_phase   <= '0;
      tick_count     <= '0;
      hps_owns_porta <= 1'b1;
      px             <= '0;
      py             <= '0;
      sweep_idx      <= '0;
      step_cnt       <= '0;
      phase          <= '0;
      nb_phase       <= '0;
      in_nb_epilogue <= '0;
    end else begin
      case (state)

        // -----------------------------------------------------------------
        S_IDLE: begin
          status_busy    <= '0;
          status_phase   <= 2'd0;
          hps_owns_porta <= 1'b1;

          if (ctrl_done_ack)   status_done <= '0;

          if (ctrl_reset) begin
            status_done      <= '0;
            status_map_ready <= '0;
            sweep_idx        <= '0;
            phase            <= '0;
            hps_owns_porta   <= '0;
            status_busy      <= 1'b1;
            state            <= S_RESET_SWEEP;
          end else if (ctrl_load_map) begin
            state <= S_MAP_LOAD;
          end else if (ctrl_brush_apply) begin
            hps_owns_porta <= '0;
            state          <= S_BRUSH_READ;
          end else if (ctrl_step || (auto_run &&
                      (!frame_lock || vsync_fall))) begin
            px             <= '0;
            py             <= '0;
            step_cnt       <= '0;
            phase          <= '0;
            nb_phase       <= '0;
            in_nb_epilogue <= '0;
            status_busy    <= 1'b1;
            status_done    <= '0;
            status_phase   <= 2'd1;
            hps_owns_porta <= '0;
            state          <= S_EVALUATE;
          end
        end

        // -----------------------------------------------------------------
        // RESET_SWEEP
        // -----------------------------------------------------------------
        S_RESET_SWEEP: begin
          if (phase == 3'd0) begin
            phase <= 3'd1;
          end else begin
            phase <= 3'd0;
            if (sweep_idx == 12'(GRID_N - 1)) begin
              sweep_idx      <= '0;
              hps_owns_porta <= 1'b1;
              status_busy    <= '0;
              state          <= S_IDLE;
            end else begin
              sweep_idx <= sweep_idx + 1'b1;
            end
          end
```

```
668         end
669
670         // ------------------------------------------------------------------
671         S_MAP_LOAD: begin
672           status_map_ready <= 1'b1;
673           state            <= S_IDLE;
674         end
675
676         // ------------------------------------------------------------------
677         // EVALUATE: 8 main phases + 4-cycle nb epilogue
678         // Ph0--Ph4: reads and PE latching
679         // Ph4: PE outputs registered into pe_*_r (done in separate always_ff)
680         // Ph5: write cell + diff_self (from registered PE outputs)
681         // Ph6: issue bot diff read
682         // Ph7: write bot diff → then enter nb_epilogue
683         // nb_epilogue nb_phase 0: issue left diff read
684         //             nb_phase 1: write left diff
685         //             nb_phase 2: issue right diff read
686         //             nb_phase 3: write right diff; advance cell
687         // ------------------------------------------------------------------
688         S_EVALUATE: begin
689           if (!in_nb_epilogue) begin
690             phase <= phase + 3'd1;
691
692             case (phase)
693               3'd0: ; // reads issued
694
695               3'd1: begin
696                 center_word   <= cell_a_rdata;
697                 center_diff   <= flow_t'(diff_rdata);
698                 has_bot_lat   <= (py < 63);
699                 has_left_lat  <= (px > 0);
700                 has_right_lat <= (px < 63);
701               end
702
703               3'd2: begin
704                 bottom_word_lat <= cell_b_rdata;
705               end
706
707               3'd3: begin
708                 left_word_lat <= cell_b_rdata;
709               end
710
711               3'd4: begin
712                 // Right word arrives; PE is combinationally valid.
713                 // pe_*_r pipeline registers capture PE outputs this cycle
714                 // (done in the separate always_ff block above).
715                 right_word_lat <= cell_b_rdata;
716               end
717
718               3'd5: begin
719                 // Cell word and diff_self written by comb block.
720               end
721
722               3'd6: begin
723                 // Bot diff read issued by comb block (no write).
724               end
725
726               3'd7: begin
727                 // Bot diff written by comb block.
728                 // Transition to nb_epilogue for left and right diffs.
729                 in_nb_epilogue <= 1'b1;
730                 nb_phase       <= 2'd0;
731                 phase          <= 3'd0;  // reset for next cell after epilogue
732               end
733
734               default: ;
735             endcase
736
737           end else begin
738             // nb_epilogue: 4 sub-phases
739             nb_phase <= nb_phase + 2'd1;
740
741             case (nb_phase)
742               2'd0: ; // left diff read issued
743               2'd1: ; // left diff written
744               2'd2: ; // right diff read issued
745               2'd3: begin
746                 // Right diff written. Advance to next cell.
747                 in_nb_epilogue <= 1'b0;
748                 nb_phase       <= '0;
749
750                 if (px == 6'd63) begin
751                   px <= '0;
```

```
                if (py == 6'd63) begin
                  py           <= '0;
                  sweep_idx    <= '0;
                  phase        <= 3'd0;
                  status_phase <= 2'd2;
                  state        <= S_COMMIT;
                end else begin
                  py <= py + 6'd1;
                end
              end else begin
                px <= px + 6'd1;
              end
            end
            default: ;
          endcase
        end
      end

      // ------------------------------------------------------------------
      // COMMIT: 3-phase per cell
      // ------------------------------------------------------------------
      S_COMMIT: begin
        phase <= phase + 3'd1;

        case (phase)
          3'd0: ; // reads issued

          3'd1: begin
            commit_word <= cell_a_rdata;
            commit_diff <= flow_t'(diff_rdata);
          end

          3'd2: begin
            phase <= 3'd0;
            if (sweep_idx == 12'(GRID_N - 1)) begin
              sweep_idx  <= '0;
              tick_count <= tick_count + 1'b1;
              step_cnt   <= step_cnt + 8'd1;

              if (step_cnt + 8'd1 < steps_per_frame) begin
                px             <= '0;
                py             <= '0;
                phase          <= 3'd0;
                nb_phase       <= '0;
                in_nb_epilogue <= '0;
                status_phase   <= 2'd1;
                state          <= S_EVALUATE;
              end else if (frame_lock) begin
                status_phase <= 2'd3;
                state        <= S_VBLANK_WAIT;
              end else begin
                status_done    <= 1'b1;
                status_busy    <= '0;
                status_phase   <= 2'd0;
                hps_owns_porta <= 1'b1;
                state          <= S_IDLE;
              end
            end else begin
              sweep_idx <= sweep_idx + 1'b1;
            end
          end

          default: ;
        endcase
      end

      // ------------------------------------------------------------------
      S_VBLANK_WAIT: begin
        if (vsync_fall) begin
          status_done    <= 1'b1;
          status_busy    <= '0;
          status_phase   <= 2'd0;
          hps_owns_porta <= 1'b1;
          state          <= S_IDLE;
        end
      end

      // ------------------------------------------------------------------
      S_BRUSH_READ: begin
        state <= S_BRUSH_LATCH;
      end

      S_BRUSH_LATCH: begin
        brush_read_word <= cell_a_rdata;
```

```
          state           <= S_BRUSH_WRITE;
        end

        S_BRUSH_WRITE: begin
          hps_owns_porta <= 1'b1;
          state          <= S_IDLE;
        end

        default: state <= S_IDLE;
      endcase
    end
  end

  // ------------------------------------------------------------------------
  // Port-B address mux
  // Phases 1--4: physics FSM reads bottom/left/right neighbors via Port B.
  // All other times: renderer drives Port B.
  // ------------------------------------------------------------------------
  logic        phys_owns_portb;
  logic [11:0] nb_portb_addr;

  assign phys_owns_portb = (state == S_EVALUATE) && !in_nb_epilogue
                           && (phase >= 3'd1) && (phase <= 3'd4);

  always_comb begin
    case (phase)
      3'd1: nb_portb_addr = addr_bot;
      3'd2: nb_portb_addr = addr_left;
      3'd3: nb_portb_addr = addr_right;
      default: nb_portb_addr = flat;
    endcase
  end

  assign cell_b_addr = phys_owns_portb ? nb_portb_addr : renderer_rd_addr;

endmodule


// ============================================================================
// cell_bram --- true dual-port synchronous BRAM  (Quartus M10K inference)
// Read-first: a_dout / b_dout present old value on the cycle of a write.
// ============================================================================
module cell_bram #(
  parameter int DEPTH = 4096,
  parameter int WIDTH = 32
)(
  input  logic                      clk,
  input  logic [$clog2(DEPTH)-1:0]  a_addr,
  input  logic                      a_we,
  input  logic [WIDTH-1:0]          a_din,
  output logic [WIDTH-1:0]          a_dout,
  input  logic [$clog2(DEPTH)-1:0]  b_addr,
  input  logic                      b_we,
  input  logic [WIDTH-1:0]          b_din,
  output logic [WIDTH-1:0]          b_dout
);
  (* ramstyle = "M10K" *)
  logic [WIDTH-1:0] mem [0:DEPTH-1];

  always_ff @(posedge clk) begin
    a_dout <= mem[a_addr];
    if (a_we) mem[a_addr] <= a_din;
  end

  always_ff @(posedge clk) begin
    b_dout <= mem[b_addr];
    if (b_we) mem[b_addr] <= b_din;
  end
endmodule


// ============================================================================
// diff_bram --- single-port synchronous BRAM, read-first
// ============================================================================
module diff_bram #(
  parameter int DEPTH = 4096,
  parameter int WIDTH = 20
)(
  input  logic                      clk,
  input  logic [$clog2(DEPTH)-1:0]  addr,
  input  logic                      we,
  input  logic [WIDTH-1:0]          din,
  output logic [WIDTH-1:0]          dout
);
```

```
  (* ramstyle = "M10K" *)
  logic [WIDTH-1:0] mem [0:DEPTH-1];

  always_ff @(posedge clk) begin
    dout <= mem[addr];
    if (we) mem[addr] <= din;
  end
endmodule

`default_nettype wire
```

Listing 1: Top-level Avalon-MM water simulator peripheral, controller FSM, BRAMs, and VGA/physics arbitration.

```
// ============================================================================
// flow_rules_pkg_v2.sv
//
// Primary package for the v2 physics path.  Imported by:
//   • flow_rules_v2.sv   (the optimised PE)
//   • vga_renderer.sv    (colour decode)
//   • liquid_flow_main.sv (controller pack/unpack)
//
// Defines the 32-bit BRAM cell word layout, all fixed-point types,
// algorithm constants, bit-field indices, and helper functions.
//
// Cell word layout (Table 2, design doc):
//   [0]      CellType        0 = BLANK, 1 = SOLID
//   [18:1]   Liquid          18-bit Q2.16 unsigned
//   [19]     Settled
//   [23:20]  SettleCount     4-bit saturating counter
//   [24]     isDownFlowing   VGA hint
//   [31:25]  Reserved
// ============================================================================

package flow_rules_pkg_v2;

  // ------------------------------------------------------------------------
  // Fixed-point widths
  // ------------------------------------------------------------------------
  parameter int FRAC_BITS = 16;
  parameter int LIQUID_W  = 18;   // Q2.16 unsigned
  parameter int FLOW_W    = 20;   // Q2.16 signed (2 extra bits for headroom)
  parameter int SETTLE_W  = 4;    // settle_count width

  // ------------------------------------------------------------------------
  // Types
  // ------------------------------------------------------------------------
  typedef logic [31:0]              cell_word_t;
  typedef logic [LIQUID_W-1:0]      liquid_t;
  typedef logic signed [FLOW_W-1:0] flow_t;

  typedef enum logic {
    CELL_BLANK = 1'b0,
    CELL_SOLID = 1'b1
  } cell_type_t;

  // ------------------------------------------------------------------------
  // Bit-field indices into cell_word_t
  // Named constants so flow_rules_v2 can use part-selects by name.
  // ------------------------------------------------------------------------
  parameter int F_CELL_TYPE      =  0;
  parameter int F_LIQUID_LO      =  1;
  parameter int F_LIQUID_HI      = 18;
  parameter int F_SETTLED        = 19;
  parameter int F_SETTLE_CNT_LO  = 20;
  parameter int F_SETTLE_CNT_HI  = 23;
  parameter int F_IS_DOWN_FLOWING = 24;

  // ------------------------------------------------------------------------
  // Algorithm constants (Q2.16 fixed-point, stored as signed flow_t)
  //
  //   MAX_VALUE       = 1.0    = 0x00010000 = 65536
  //   MIN_LIQUID      = 0.005  = 0x00000148 = 328
  //   MAX_COMPRESSION = 0.25   = 0x00004000 = 16384
  //   MAX_FLOW        = 4.0    = 0x00040000 = 262144  (unused in v2)
  //
  // Pre-computed vertical_target constants (change A+B from design doc):
  //   VERT_THRESH_A = 2*MAX_VALUE + MAX_COMPRESSION = 2*65536 + 16384 = 147456
  //   ML_SQ_Q216   = MAX_VALUE^2 in Q2.16
  //                = (65536 * 65536) >> 16 = 65536  (same as MAX_VALUE)
  //   INV_1P25     = 1 / 1.25 in Q2.16 = 0.8 * 65536 = 52429
  //   INV3         = 1 / 3   in Q2.16  ≈ 21845
  // ------------------------------------------------------------------------
```

```
  localparam flow_t MAX_VALUE      = 20'sd65536;
  localparam flow_t MIN_LIQUID     = 20'sd328;
  localparam flow_t MAX_COMPRESSION= 20'sd16384;

  // Vertical-target thresholds (constant-folded, saves adder + DSP vs v1)
  localparam flow_t VERT_THRESH_A  = 20'sd147456;  // 2*MAX_VALUE + MAX_COMPRESSION
  localparam flow_t ML_SQ_Q216     = 20'sd65536;   // MAX_VALUE^2 in Q2.16

  // Reciprocals for multiply-based division
  localparam flow_t INV_1P25       = 20'sd52429;   // ~1/1.25
  localparam flow_t INV3           = 20'sd21845;   // ~1/3

  // Settling heuristic
  localparam logic [SETTLE_W-1:0] SETTLE_THRESHOLD = 4'd10;

  // ------------------------------------------------------------------------
  // Fixed-point arithmetic helpers
  // ------------------------------------------------------------------------

  // Zero-extend liquid_t (unsigned 18-bit) to signed flow_t (20-bit).
  function automatic flow_t liq_to_flow(input liquid_t value);
    return flow_t'({{(FLOW_W-LIQUID_W){1'b0}}, value});
  endfunction

  // Q2.16 multiply: (a * b) >> 16, both signed.
  function automatic flow_t fp_mul(input flow_t a, input flow_t b);
    logic signed [(2*FLOW_W)-1:0] product;
    product = $signed(a) * $signed(b);
    return flow_t'(product >>> FRAC_BITS);
  endfunction

  // Divide by 2 (arithmetic right-shift 1).
  function automatic flow_t fp_div2(input flow_t value);
    return flow_t'($signed(value) >>> 1);
  endfunction

  // Divide by 4 (arithmetic right-shift 2).  Free --- no DSP.
  function automatic flow_t fp_div4(input flow_t value);
    return flow_t'($signed(value) >>> 2);
  endfunction

  // Divide by 3 via reciprocal multiply.  One DSP.
  function automatic flow_t fp_div3(input flow_t value);
    return fp_mul(value, INV3);
  endfunction

  // Clamp flow to [0, available].
  // v2 drops the MAX_FLOW upper bound (always < 4.0 by construction).
  function automatic flow_t clamp_pos_flow(
    input flow_t candidate,
    input flow_t available
  );
    flow_t c;
    begin
      // Floor at zero
      c = ($signed(candidate) > 0) ? candidate : '0;
      // Cap at available liquid
      c = ($signed(c) < $signed(available)) ? c : available;
      return c;
    end
  endfunction

  // ------------------------------------------------------------------------
  // vertical_target --- three-case piecewise equilibrium level V(r, d)
  //
  //   s = r + d
  //   case 1: s ≤ MAX_VALUE          → V = MAX_VALUE
  //   case 2: s < VERT_THRESH_A      → V = (ML_SQ_Q216 + s*MAX_COMPRESSION)
  //                                        / (MAX_VALUE + MAX_COMPRESSION)
  //                                    = fp_mul(numer, INV_1P25)
  //   case 3: s ≥ VERT_THRESH_A      → V = (s + MAX_COMPRESSION) >> 1
  //
  // Pre-computed constants fold away the adder and one DSP vs v1.
  // ------------------------------------------------------------------------
  function automatic flow_t vertical_target(
    input flow_t source_liquid,
    input flow_t dest_liquid
  );
    flow_t sum;
    flow_t numer;
    begin
      sum = source_liquid + dest_liquid;

      if ($signed(sum) <= $signed(MAX_VALUE)) begin
```

```
        return MAX_VALUE;
      end

      if ($signed(sum) < $signed(VERT_THRESH_A)) begin
        numer = ML_SQ_Q216 + fp_mul(sum, MAX_COMPRESSION);
        return fp_mul(numer, INV_1P25);
      end

      return fp_div2(sum + MAX_COMPRESSION);
    end
  endfunction

  // ---------------------------------------------------------------------
  // Cell word accessors (used by vga_renderer and controller)
  // ---------------------------------------------------------------------

  function automatic cell_type_t word_cell_type(input cell_word_t w);
    return cell_type_t'(w[F_CELL_TYPE]);
  endfunction

  function automatic liquid_t word_liquid(input cell_word_t w);
    return liquid_t'(w[F_LIQUID_HI : F_LIQUID_LO]);
  endfunction

  function automatic logic word_settled(input cell_word_t w);
    return w[F_SETTLED];
  endfunction

  function automatic logic [SETTLE_W-1:0] word_settle_count(input cell_word_t w);
    return w[F_SETTLE_CNT_HI : F_SETTLE_CNT_LO];
  endfunction

  function automatic logic word_down_flowing(input cell_word_t w);
    return w[F_IS_DOWN_FLOWING];
  endfunction

  // ---------------------------------------------------------------------
  // Pack / unpack between cell fields and cell_word_t
  // ---------------------------------------------------------------------

  function automatic cell_word_t pack_cell(
    input cell_type_t          ctype,
    input liquid_t             liquid,
    input logic                settled,
    input logic [SETTLE_W-1:0] settle_count,
    input logic                down_flowing
  );
    cell_word_t w;
    w = '0;
    w[F_CELL_TYPE]                    = logic'(ctype);
    w[F_LIQUID_HI  : F_LIQUID_LO]    = liquid;
    w[F_SETTLED]                      = settled;
    w[F_SETTLE_CNT_HI:F_SETTLE_CNT_LO] = settle_count;
    w[F_IS_DOWN_FLOWING]              = down_flowing;
    return w;
  endfunction

  function automatic cell_word_t zero_liquid(input cell_word_t w);
    // Zero the liquid, settled, and settle_count fields; preserve CellType.
    cell_word_t r;
    r = '0;
    r[F_CELL_TYPE] = w[F_CELL_TYPE];
    return r;
  endfunction

endpackage
```

Listing 2: Optimized fixed-point package used by the integrated RTL PE.

```
// ============================================================================
// flow_rules_v2.sv  ---  hardware-optimised flow-rules processing element
//
// Implements Rule 1 (downward/gravity) and Rule 2 (horizontal spreading)
// from Section 3.2 of the VGA_Water_Design design document.
//
// Changes from flow_rules.sv (v1) --- all three come from constant folding:
//
//  (C) FLOW_SPEED multiplications removed
//        V1 Rule 1:
//          if (bot_liq > 0 && flow > MIN_FLOW)
//            flow = fp_mul(flow, FLOW_SPEED);   // <- 1 DSP multiply
//        V2: entire block deleted.
//        FLOW_SPEED = 1.0 in Q2.16.  Multiplying any value by 1.0 is the
//        identity.  There is no damping at the current parameter setting,
```

```
//        so the DSP and its surrounding comparator are pure dead logic.
//        Same elimination applied to Rule 2a and Rule 2b (3 DSPs total).
//
//  (D) MAX_FLOW clamp removed
//        V1: clamp_flow(flow, MAX_FLOW, remaining)
//                          ↑ also tests flow < MAX_FLOW = 4.0
//        V2: clamp_pos_flow(flow, remaining)
//                          no MAX_FLOW comparison
//        Flow values are derived from liquid amounts bounded by
//        MAX_LIQUID + MAX_COMPRESSION = 1.25, which is always < MAX_FLOW = 4.0,
//        so fp_min(flow, MAX_FLOW) is never the active branch.
//        Saves three comparators (one per rule direction).
//
//  (A+B) vertical_target uses pre-computed constants --- see flow_rules_pkg_v2.
//
// Net hardware savings vs v1:
//   DSP multipliers : −4  (3 × FLOW_SPEED, 1 × MAX_LIQUID²)
//   Comparators     : −6  (3 × MIN_FLOW guard, 3 × MAX_FLOW clamp)
//   Adder + shift   : −1  (VERT_THRESH_A pre-computed)
//
// Interface is identical to flow_rules.sv so it is a drop-in replacement.
// ============================================================================

module flow_rules_v2
#(
  // 1 → right spread uses ÷3  (reference-exact, slight rightward bias, 1 DSP)
  // 0 → right spread uses ÷4  (symmetric, saves the INV3 DSP multiply)
  parameter bit USE_RIGHT_DIV3 = 1'b1
)(
  // ---- Current cell (packed 32-bit GRID_MEM word) ----------------------
  input  flow_rules_pkg_v2::cell_word_t  cell_word_in,

  // ---- Boundary valid flags (0 when neighbour is outside the grid) -----
  input  logic        has_bottom,
  input  logic        has_left,
  input  logic        has_right,

  // ---- Neighbour packed cell words -------------------------------------
  input  flow_rules_pkg_v2::cell_word_t  bottom_word_in,
  input  flow_rules_pkg_v2::cell_word_t  left_word_in,
  input  flow_rules_pkg_v2::cell_word_t  right_word_in,

  // ---- Signed Q2.16 diffs for the Diffs scratch buffer ----------------
  output flow_rules_pkg_v2::flow_t       diff_self,
  output flow_rules_pkg_v2::flow_t       diff_bottom,
  output flow_rules_pkg_v2::flow_t       diff_left,
  output flow_rules_pkg_v2::flow_t       diff_right,

  // ---- Updated cell word (Settled / SettleCount / isDownFlowing only) --
  output flow_rules_pkg_v2::cell_word_t  cell_word_out,

  // ---- Liquid remaining after downward + horizontal rules --------------
  output flow_rules_pkg_v2::flow_t       remaining_out,

  // ---- Asserted when any flow occurred this cycle ----------------------
  output logic        flow_happened
);
  import flow_rules_pkg_v2::*;
  always_comb begin : flow_rules_proc

    // ------------------------------------------------------------------
    // Local variables (implicitly automatic inside always_comb)
    // ------------------------------------------------------------------
    cell_type_t          self_type;
    liquid_t             self_liq_raw;
    logic                self_settled;
    logic [SETTLE_W-1:0] settle_cnt;
    flow_t               self_liquid;
    flow_t               bot_liq, lft_liq, rgt_liq;
    flow_t               start_value, remaining;
    flow_t               flow, target, delta;
    logic [SETTLE_W-1:0] sc_next;
    logic                any_flow;

    // ------------------------------------------------------------------
    // Output defaults
    // ------------------------------------------------------------------
    cell_word_out = cell_word_in;
    diff_self     = '0;
    diff_bottom   = '0;
    diff_left     = '0;
    diff_right    = '0;
    remaining_out = '0;
    flow_happened = 1'b0;
```

```
    // ------------------------------------------------------------------
    // Unpack current-cell fields
    // ------------------------------------------------------------------
    self_type    = word_cell_type(cell_word_in);
    self_liq_raw = word_liquid(cell_word_in);
    self_settled = word_settled(cell_word_in);
    settle_cnt   = word_settle_count(cell_word_in);
    self_liquid  = liq_to_flow(self_liq_raw);

    // ------------------------------------------------------------------
    // Unpack neighbour liquid amounts
    // ------------------------------------------------------------------
    bot_liq = liq_to_flow(word_liquid(bottom_word_in));
    lft_liq = liq_to_flow(word_liquid(left_word_in));
    rgt_liq = liq_to_flow(word_liquid(right_word_in));

    // Silence lint tools
    start_value = '0;
    remaining   = '0;
    flow        = '0;
    target      = '0;
    delta       = '0;
    sc_next     = '0;
    any_flow    = 1'b0;

    // ==================================================================
    // Skip conditions
    // ==================================================================

    if (self_type == CELL_SOLID) begin
      // Solid: clear liquid and lock as settled
      cell_word_out[F_LIQUID_HI    : F_LIQUID_LO]     = '0;
      cell_word_out[F_SETTLED]                         = 1'b1;
      cell_word_out[F_SETTLE_CNT_HI: F_SETTLE_CNT_LO] = '0;
      cell_word_out[F_IS_DOWN_FLOWING]                 = 1'b0;

    end else if ((self_liquid == '0) || self_settled) begin
      ; // empty or settled --- pass through

    end else if ($signed(self_liquid) < $signed(MIN_LIQUID)) begin
      // Sub-threshold: drain residual via diff
      diff_self = -self_liquid;

    end else begin

      // ==================================================================
      // Active cell --- Rule 1 then Rule 2
      // ==================================================================
      start_value = self_liquid;
      remaining   = self_liquid;

      // ------------------------------------------------------------------
      // Rule 1 --- Downward flow (gravity)           [Eq. 1, PDF §3.2]
      //
      //   target = V(remaining, d_bottom)
      //            ↪ uses VERT_THRESH_A (147456) and ML_SQ_Q216 (65536)
      //              as hardwired constants --- no adder, no extra DSP
      //   flow   = target − d_bottom
      //
      //   (C) FLOW_SPEED damping block removed:
      //       V1 had:  if (bot_liq>0 && flow>MIN_FLOW) flow = fp_mul(flow,FLOW_SPEED);
      //       FLOW_SPEED=1.0 → identity multiply → deleted entirely.
      //
      //   (D) Clamp: clamp_pos_flow(flow, remaining)
      //       V1 used clamp_flow(flow, MAX_FLOW, remaining); MAX_FLOW test removed.
      // ------------------------------------------------------------------
      if (has_bottom && (word_cell_type(bottom_word_in) == CELL_BLANK)) begin

        target = vertical_target(self_liquid, bot_liq);
        flow   = target - bot_liq;
        // flow can be negative when bot_liq > target; clamp_pos_flow floors at 0.

        flow = clamp_pos_flow(flow, remaining);  // change (D): no MAX_FLOW test

        if (flow != '0) begin
          remaining   = remaining   - flow;
          diff_self   = diff_self   - flow;
          diff_bottom = diff_bottom + flow;
          any_flow    = 1'b1;

          // isDownFlowing: both cells hold liquid → falling stream
          // (VGA renderer draws at full opacity when this flag is set)
          cell_word_out[F_IS_DOWN_FLOWING] =
```

```
            ($signed(bot_liq) > 0) ? 1'b1 : 1'b0;
        end

      end // Rule 1

      // Drain sub-threshold remnant before horizontal spread
      if ($signed(remaining) < $signed(MIN_LIQUID)) begin
        diff_self = diff_self - remaining;
        remaining = '0;
      end else begin

        // ------------------------------------------------------------
        // Rule 2a --- Leftward spread                [Eq. 2, PDF §3.2]
        //
        //   φ_left = (remaining − d_left) / 4
        //
        //   ÷4 is a free arithmetic right-shift (no DSP).
        //
        //   (C) FLOW_SPEED block removed:
        //       V1 had:  if (flow>MIN_FLOW) flow = fp_mul(flow, FLOW_SPEED);
        //       FLOW_SPEED=1.0 → deleted.  Saves 1 DSP + 1 comparator.
        //
        //   (D) Clamp: clamp_pos_flow (no MAX_FLOW test).
        //       flow = delta/4 > 0 always here, so fp_max(f,0) is also a
        //       no-op in practice, but kept inside clamp_pos_flow for
        //       uniformity and safety.
        // ------------------------------------------------------------
        if (has_left && (word_cell_type(left_word_in) == CELL_BLANK)) begin

          delta = remaining - lft_liq;

          if ($signed(delta) > 0) begin
            flow = fp_div4(delta);                  // free right-shift
            flow = clamp_pos_flow(flow, remaining);  // change (D)

            if (flow != '0) begin
              remaining = remaining - flow;
              diff_self = diff_self - flow;
              diff_left = diff_left + flow;
              any_flow  = 1'b1;
            end
          end

        end // Rule 2a

        // Drain sub-threshold remnant
        if ($signed(remaining) < $signed(MIN_LIQUID)) begin
          diff_self = diff_self - remaining;
          remaining = '0;
        end else begin

          // ------------------------------------------------------------
          // Rule 2b --- Rightward spread              [Eq. 2, PDF §3.2]
          //
          //   φ_right = (remaining − d_right) / 3  [USE_RIGHT_DIV3 = 1]
          //           = (remaining − d_right) / 4  [USE_RIGHT_DIV3 = 0]
          //
          //   ÷3: one DSP multiply via fp_mul(delta, INV3).
          //   ÷4: free right-shift, same as left spread.
          //   The /3 (right) vs /4 (left) asymmetry matches the Unity
          //   reference implementation's slight rightward bias.
          //
          //   (C) FLOW_SPEED block removed:
          //       V1 had:  if (flow>MIN_FLOW) flow = fp_mul(flow, FLOW_SPEED);
          //       FLOW_SPEED=1.0 → deleted.  Saves 1 DSP + 1 comparator.
          //
          //   (D) Clamp: clamp_pos_flow (no MAX_FLOW test).
          // ------------------------------------------------------------
          if (has_right && (word_cell_type(right_word_in) == CELL_BLANK)) begin

            delta = remaining - rgt_liq;

            if ($signed(delta) > 0) begin
              // USE_RIGHT_DIV3 is a parameter constant → synthesis keeps
              // only one branch and discards the other entirely.
              flow = USE_RIGHT_DIV3 ? fp_div3(delta)   // 1 DSP (INV3 multiply)
                                    : fp_div4(delta);  // free right-shift
              flow = clamp_pos_flow(flow, remaining);  // change (D)

              if (flow != '0) begin
                remaining  = remaining  - flow;
                diff_self  = diff_self  - flow;
                diff_right = diff_right + flow;
                any_flow   = 1'b1;
```

```
            end
          end

        end // Rule 2b

        // Drain any final sub-threshold remnant
        if ($signed(remaining) < $signed(MIN_LIQUID)) begin
          diff_self = diff_self - remaining;
          remaining = '0;
        end

      end // rightward-spread block
    end // leftward-spread block

    // ------------------------------------------------------------
    // Settling heuristic
    //
    // any_flow = 1 → liquid moved: clear Settled + SettleCount so the
    //                cell (and its neighbours, handled upstream) resumes.
    //
    // no flow AND remaining == start_value → nothing moved at all:
    //                saturating-increment SettleCount; mark Settled
    //                when the threshold is reached.
    // ------------------------------------------------------------
    if (any_flow) begin
      cell_word_out[F_SETTLED]                       = 1'b0;
      cell_word_out[F_SETTLE_CNT_HI: F_SETTLE_CNT_LO] = '0;
    end else if (remaining == start_value) begin
      sc_next = settle_cnt;
      if (sc_next != {SETTLE_W{1'b1}})   // saturating: stop at 4'b1111
        sc_next = sc_next + 1'b1;
      cell_word_out[F_SETTLE_CNT_HI: F_SETTLE_CNT_LO] = sc_next;
      if (sc_next >= SETTLE_THRESHOLD)
        cell_word_out[F_SETTLED] = 1'b1;
    end

    flow_happened = any_flow;
    remaining_out = remaining;

  end // active-cell block

end : flow_rules_proc

endmodule
```

Listing 3: Optimized combinational flow-rules PE instantiated by the top-level controller.

```
// ============================================================================
// vga_pkg.sv  ---  VGA renderer package
//
// Constants, types, and helpers specific to the VGA rendering path.
// ============================================================================

package vga_pkg;

  // ------------------------------------------------------------------------
  // Display geometry
  //
  //                  +-------+-------------+-------+
  //                  |  16px |             |  16px |   Y_OFF (top border)
  //                  +-------+-------------+-------+
  //                  |       |             |       |
  //                  |  96px |  448 × 448  |  96px |
  //                  |border |   sim area  |border |
  //                  |       |             |       |
  //                  +-------+-------------+-------+
  //                  |  16px |             |  16px |   bottom border
  //                  +-------+-------------+-------+
  //                            640 × 480
  // ------------------------------------------------------------------------
  parameter int SCREEN_W = 640;
  parameter int SCREEN_H = 480;

  parameter int GRID_W   = 64;            // cells across
  parameter int GRID_H   = 64;            // cells down

  parameter int BLOCK_PX = 7;             // pixels per cell side
  parameter int SIM_W    = GRID_W * BLOCK_PX;        // 448
  parameter int SIM_H    = GRID_H * BLOCK_PX;        // 448
  parameter int X_OFF    = (SCREEN_W - SIM_W) / 2;   // 96
  parameter int Y_OFF    = (SCREEN_H - SIM_H) / 2;   // 16
  parameter int X_END    = X_OFF + SIM_W;            // 544
  parameter int Y_END    = Y_OFF + SIM_H;            // 464

```

```
  // Counter widths derived from the geometry.
  // BLOCK_PX = 7 fits in 3 bits, GRID_W/H = 64 fits in 6 bits.
  parameter int PIX_IN_CELL_W = 3;
  parameter int CELL_IDX_W    = 6;


  // 64 × 64 = 4096 cells → 12-bit address.
  parameter int GRID_ADDR_W = 12;
  typedef logic [GRID_ADDR_W-1:0] grid_addr_t;

  typedef logic [23:0] rgb_t;

  parameter rgb_t COLOR_OFFSCREEN = 24'h00_00_00;   // mandatory black
  parameter rgb_t COLOR_BORDER    = 24'h18_1A_20;   // dark slate
  parameter rgb_t COLOR_WALL      = 24'h00_00_00;   // black
  parameter rgb_t COLOR_EMPTY     = 24'hFF_FF_FF;   // white
  parameter rgb_t COLOR_LIQUID    = 24'h20_60_FF;   // blue

  // ------------------------------------------------------------------------
  // Helpers
  // ------------------------------------------------------------------------

  // Pack {cell_y, cell_x} into a flat BRAM address.
  function automatic grid_addr_t cell_to_addr(
    input logic [CELL_IDX_W-1:0] cell_x,
    input logic [CELL_IDX_W-1:0] cell_y
  );
    return {cell_y, cell_x};
  endfunction

  // Sim-area predicates.  Combinational, no DSPs, no comparators with
  // anything but synthesis-time constants --- these turn into a small
  // tree of LUTs.
  function automatic logic in_sim_area_x(input logic [9:0] pixel_x);
    return (pixel_x >= X_OFF) && (pixel_x < X_END);
  endfunction

  function automatic logic in_sim_area_y(input logic [9:0] pixel_y);
    return (pixel_y >= Y_OFF) && (pixel_y < Y_END);
  endfunction

endpackage
```

Listing 4: Package used for VGA renderer.

```
// ============================================================================
// vga_renderer.sv  ---  VGA output for the liquid simulator
//
// Reads cells from the grid BRAM and renders them as 7×7 blocks on a
// 640×480 display, centred in a 448×448 sim area.
// ============================================================================

module vga_renderer


(
    input  logic        clk,           // 50 MHz system clock
    input  logic        reset,

    // ---- Read port to cell BRAM ----
    output vga_pkg::grid_addr_t  grid_rd_addr,  // 12-bit cell index
    input  flow_rules_pkg_v2::cell_word_t  grid_rd_data,  // 32-bit packed cell
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
                        VGA_BLANK_n, VGA_SYNC_n
);
    import vga_pkg::*;
    import flow_rules_pkg_v2::*;

    logic [10:0] hcount;
    logic [9:0]  vcount;

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

    wire [9:0] pixel_x = hcount[10:1];   // 0..639
    wire [9:0] pixel_y = vcount;         // 0..479

    // Sim-area detection (combinational, from package helpers)
    wire in_sim_x = in_sim_area_x(pixel_x);
    wire in_sim_y = in_sim_area_y(pixel_y);
    wire in_sim   = in_sim_x && in_sim_y;

    // Cell coordinate counters --- avoid divide-by-7 by walking counters
    logic [PIX_IN_CELL_W-1:0] pix_in_cell_x;
```

```
    logic [CELL_IDX_W-1:0]    cell_x;

    always_ff @(posedge clk) begin
        if (reset) begin
            pix_in_cell_x <= '0;
            cell_x <= '0;
        end else if (hcount[0]) begin
            // Once-per-pixel update.
            if (pixel_x == X_OFF - 1) begin
                // About to enter sim area on the NEXT pixel:
                // present cell 0's address now so the BRAM read can
                // complete in time.
                pix_in_cell_x <= '0;
                cell_x <= '0;
            end else if (in_sim_x) begin
                if (pix_in_cell_x == BLOCK_PX - 1) begin
                    pix_in_cell_x <= '0;
                    cell_x        <= cell_x + 1'b1;
                end else begin
                    pix_in_cell_x <= pix_in_cell_x + 1'b1;
                end
            end
        end
    end

    logic [PIX_IN_CELL_W-1:0] pix_in_cell_y;
    logic [CELL_IDX_W-1:0]    cell_y;
    wire end_of_line = (hcount == 11'd1599);   // HTOTAL - 1

    always_ff @(posedge clk) begin
        if (reset) begin
            pix_in_cell_y <= '0;
            cell_y        <= '0;
        end else if (end_of_line) begin
            if (pixel_y == Y_OFF - 1) begin
                pix_in_cell_y <= '0;
                cell_y        <= '0;
            end else if (in_sim_y) begin
                if (pix_in_cell_y == BLOCK_PX - 1) begin
                    pix_in_cell_y <= '0;
                    cell_y        <= cell_y + 1'b1;
                end else begin
                    pix_in_cell_y <= pix_in_cell_y + 1'b1;
                end
            end
        end
    end


    // BRAM read address
    assign grid_rd_addr = cell_to_addr(cell_x, cell_y);

    // Latency-matching pipeline registers
    logic in_sim_d1;
    logic blank_n_d1;

    always_ff @(posedge clk) begin
        in_sim_d1  <= in_sim;
        blank_n_d1 <= VGA_BLANK_n;
    end

    // Colour decoding (combinational, on the BRAM-aligned signals)
    cell_type_t cell_type;
    liquid_t cell_liquid;
    rgb_t rgb;

    assign cell_type = word_cell_type(grid_rd_data);
    assign cell_liquid = word_liquid(grid_rd_data);

    always_comb begin
        if (!blank_n_d1)                 rgb = COLOR_OFFSCREEN;
        else if (!in_sim_d1)             rgb = COLOR_BORDER;
        else if (cell_type == CELL_SOLID) rgb = COLOR_WALL;
        else if (cell_liquid == 18'd0)    rgb = COLOR_EMPTY;
        else                              rgb = COLOR_LIQUID;
    end

    assign {VGA_R, VGA_G, VGA_B} = rgb;

endmodule


// ============================================================================
```

```
// 640×480 @ 60 Hz timing from a 50 MHz input clock. (Similar to Lab 3)
// One pixel every other system cycle; hcount[0] is the 25 MHz pixel clock.
module vga_counters(
    input  logic clk50, reset,
    output logic [10:0]  hcount,    // hcount[10:1] = pixel column
    output logic [9:0]   vcount,    // pixel row
    output logic         VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n
);

    parameter HACTIVE      = 11'd1280,
              HFRONT_PORCH = 11'd32,
              HSYNC        = 11'd192,
              HBACK_PORCH  = 11'd96,
              HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 1600

    parameter VACTIVE      = 10'd480,
              VFRONT_PORCH = 10'd10,
              VSYNC        = 10'd2,
              VBACK_PORCH  = 10'd33,
              VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

    logic endOfLine;
    always_ff @(posedge clk50 or posedge reset)
        if (reset)          hcount <= 11'd0;
        else if (endOfLine) hcount <= 11'd0;
        else                hcount <= hcount + 11'd1;
    assign endOfLine = (hcount == HTOTAL - 1);

    logic endOfField;
    always_ff @(posedge clk50 or posedge reset)
        if (reset)              vcount <= 10'd0;
        else if (endOfLine)
            if (endOfField)     vcount <= 10'd0;
            else                vcount <= vcount + 10'd1;
    assign endOfField = (vcount == VTOTAL - 1);

    assign VGA_HS       = !((hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
    assign VGA_VS       = !(vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
    assign VGA_SYNC_n   = 1'b0;
    assign VGA_BLANK_n  = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                          !( vcount[9]  | (vcount[8:5] == 4'b1111) );
    assign VGA_CLK      = hcount[0];

endmodule
```

Listing 5: VGA renderer for the 64×64 simulation grid on a 640×480 display.

A.2 C(.c, .h) codes (Software)

```
/* ============================================================================
 * main.c --- HPS main loop for the liquid simulator
 *
 *   1. Opens /dev/water_sim  (the FPGA peripheral, via the kernel driver)
 *      and /dev/input/mice    (the USB mouse).
 *   2. Resets the FPGA's water simulator to a known state.
 *   3. Loads an EMPTY cell map into the FPGA's cell BRAM
 *   4. Enables auto-run mode: the FPGA advances the simulation one
 *      tick per VGA frame, autonomously.Auto-run means the
 *      controller starts each new tick on its own, timed against VSYNC.
 *   5. Runs a loop that translates each mouse event into an FPGA
 *      "brush" command so the user can paint water, walls, and
 *      erasures interactively.
 *   6. On Ctrl-C, resets the FPGA and exits cleanly.
 *
 *
 * The HPS main loop:
 *   read mouse → translate to brush → write registers → repeat.
 *
 * ----------------------------------------------------------------------
 * FOR THE FPGA CONTROLLER
 * ----------------------------------------------------------------------
 *  *
 *   CTRL_RESET   (CTRL_REG[1] --- write-one-pulse, software sets it,
 *                 controller self-clears)
 *     Controller MUST:
 *       (a) clear its FSM state, settle counts, and the Diffs BRAM
 *       (b) zero the Liquid field of every cell in the cell BRAM
 *       (c) leave CellType (wall map) untouched
 *           --- software does CellType bookkeeping through LOAD_MAP
 *       (d) self-clear the bit when done
 *       (e) deassert STATUS_BUSY when fully reset
```

```
 *
 *   CTRL_LOAD_MAP   (CTRL_REG[2])
 *     Pulsed by software after we've written all 4096 cell words.
 *     Controller MUST latch the CellType bits into the internal wall
 *     map (if the renderer or physics keeps a private copy), set
 *     STATUS_MAP_READY, and self-clear the bit.
 *
 *   CTRL_BRUSH_APPLY   (CTRL_REG[4])
 *     Pulsed by software after writing MOUSE_POS_REG and BRUSH_CFG_REG.
 *     Controller MUST:
 *       - read MOUSE_POS_REG → (grid_x, grid_y)
 *       - read BRUSH_CFG_REG → (tool, radius, liquid_amount)
 *       - apply the brush to the affected disc of cells:
 *           tool = ADD    → set CellType=BLANK, add liquid_amount
 *           tool = ERASE  → set CellType=BLANK, liquid=0
 *           tool = WALL   → set CellType=SOLID, liquid=0
 *           radius = 0    → single cell at (x, y)
 *           radius > 0    → disc of cells, Chebyshev or Euclidean
 *                           (we use radius=0 in this file; that
 *                            simplifies your job, you can ignore the
 *                            disc rasterisation for now)
 *       - self-clear the bit when done
 *
 *   STEP_CFG_REG.AUTO_RUN     (bit 0, persistent)
 *     Software sets this once at startup.  While set, controller runs
 *     one evaluate + commit tick per VGA frame autonomously --- no
 *     software step pulses required.
 *
 *   STEP_CFG_REG.FRAME_LOCK   (bit 1, persistent)
 *     When set, controller waits for VSYNC before starting the next
 *     tick.  This prevents VGA tearing because the renderer can't
 *     observe the cell BRAM mid-update.  We set this from software.
 *
 *   VGA_CTRL_REG.ENABLE        (bit 0, persistent)
 *     The renderer should drive RGB outputs only when this bit is set.
 *     Lets us safely load the map (no garbage visible) before kicking
 *     off the sim.
 *
 *   In auto-run mode, we DO NOT poll STATUS_DONE or pulse
 *   CTRL_DONE_ACK.  The controller can leave DONE sticky-set if it
 *   wants --- software just ignores it.  Manual-step mode would care
 *   about DONE, but we're not using that here.
 *
 * ------------------------------------------------------------------------
 * BUILD AND RUN
 * ------------------------------------------------------------------------
 *   See README.md and Makefile.  Briefly:
 *     $ make
 *     $ sudo insmod /path/to/water_sim_drv.ko
 *     $ sudo ./liquid-hps
 *
 *   Needs root (or input group + cap_sys_rawio) to open /dev/input/mice
 *   and /dev/water_sim.
 * ======================================================================== */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <stdint.h>

#include "water_sim.h"     /* register map and ioctl interface */
#include "mouse_io.h"      /* mouse → grid event abstraction */


/* ========================================================================
 * Compile-time tuning
 * ======================================================================== */

/* How much liquid one "add water" click injects, in Q2.16.
 *
 * BRUSH_CFG_REG's liquid field is 16 bits wide (bits [31:16]).  Q216_HALF
 * is 0x00008000 --- its low 16 bits are 0x8000.  When the controller
 * reads the field, it should treat it as the low 16 bits of a Q2.16
 * value, which represents 0.5 (half a cell's nominal capacity).
*/
#define WATER_BRUSH_AMOUNT     (Q216_HALF & 0xFFFFu)   /* = 0x8000 */

#define BRUSH_RADIUS 0
```

```c
#define STEPS_PER_FRAME        1



// Globals
static int water_fd = -1;        /* /dev/water_sim file descriptor */
static volatile int running  = 1;         /* Set to 0 by SIGINT to exit     */


/* ========================================================================
 * Signal handling
 *
 * On Ctrl-C or SIGTERM, set the running flag to 0.  We deliberately
 * register the handler WITHOUT SA_RESTART, so a signal arriving during
 * a blocking read() on /dev/input/mice causes the read to return -1
 * with errno = EINTR.  That breaks the main loop out of mouse_poll()
 * so we can check `running` and exit cleanly.
 * ======================================================================== */

static void sig_handler(int sig)
{
    (void)sig;
    running = 0;
}

static void install_signal_handlers(void)
{
    struct sigaction sa = {0};
    sa.sa_handler = sig_handler;
    sa.sa_flags   = 0;                /* no SA_RESTART --- let read() return EINTR */
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT,  &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);
}


/* Write a 32-bit value to register `offset` (one of the WATER_REG_*
 * constants from water_sim.h).  Returns 0 on success, -1 on error. */
static int reg_write(uint32_t offset, uint32_t value)
{
    water_sim_reg_t r = { .reg = offset, .value = value };
    if (ioctl(water_fd, WATER_SIM_WRITE_REG, &r) < 0) {
        fprintf(stderr, "WRITE_REG(0x%04x = 0x%08x) failed: %s\n",
                offset, value, strerror(errno));
        return -1;
    }
    return 0;
}

/* Read the 32-bit value at register `offset` into *out. */
static int reg_read(uint32_t offset, uint32_t *out)
{
    water_sim_reg_t r = { .reg = offset, .value = 0 };
    if (ioctl(water_fd, WATER_SIM_READ_REG, &r) < 0) {
        fprintf(stderr, "READ_REG(0x%04x) failed: %s\n",
                offset, strerror(errno));
        return -1;
    }
    *out = r.value;
    return 0;
}

/* Pulse a write-one-pulse control bit by writing the appropriate CTRL_*
 * mask.  The controller self-clears the bit when it has consumed the
 * pulse, so this single write is sufficient. */
static int ctrl_pulse(uint32_t bit_mask)
{
    return reg_write(WATER_REG_CTRL, bit_mask);
}

/* Write one cell directly into the FPGA's cell BRAM.  Used at startup
 * to load the initial map.  Slow (~one ioctl per cell), but only
 * happens once. */
static int cell_write(int x, int y, uint32_t cell_word)
{
    water_sim_cell_t c = { .x = x, .y = y, .cell_word = cell_word };
    if (ioctl(water_fd, WATER_SIM_WRITE_CELL, &c) < 0) {
        fprintf(stderr, "WRITE_CELL(%d, %d) failed: %s\n",
                x, y, strerror(errno));
        return -1;
    }
    return 0;
}

```

```c

/* ============================================================================
 * Setup --- runs once at startup
 * ============================================================================ */

/* Open /dev/water_sim and initialise the mouse subsystem.
 * Returns 0 on success, -1 on failure. */
static int open_devices(void)
{
    water_fd = open("/dev/water_sim", O_RDWR);
    if (water_fd < 0) {
        fprintf(stderr, "Cannot open /dev/water_sim: %s\n", strerror(errno));
        fprintf(stderr, "  Did you load the kernel module?\n");
        fprintf(stderr, "  Try: sudo insmod water_sim_drv.ko\n");
        return -1;
    }

    if (mouse_init("/dev/input/mice") < 0) {
        /* mouse_init prints its own error via perror() */
        close(water_fd);
        water_fd = -1;
        return -1;
    }

    return 0;
}

/* Reset the FPGA peripheral and wait for it to come back idle.
 *
 * Controller-side contract: after we pulse CTRL_RESET, the controller
 * should clear all internal state, zero every cell's liquid field
 * (leaving CellType alone), and deassert STATUS_BUSY when finished. */
static int reset_fpga(void)
{
    printf("Resetting FPGA peripheral... ");
    fflush(stdout);

    if (ctrl_pulse(CTRL_RESET) < 0) return -1;

    /* Poll STATUS_BUSY for up to ~100 ms.  Reset should complete in
     * microseconds; this is just defensive. */
    for (int tries = 0; tries < 1000; tries++) {
        uint32_t status;
        if (reg_read(WATER_REG_STATUS, &status) < 0) return -1;
        if ((status & STATUS_BUSY) == 0) {
            printf("done.\n");
            return 0;
        }
        usleep(100);   /* 100 µs */
    }

    fprintf(stderr, "Reset timed out --- STATUS_BUSY still asserted after 100 ms.\n");
    return -1;
}

/* Write an all-blank, no-liquid cell map into the cell BRAM, then
 * pulse CTRL_LOAD_MAP to tell the controller "the map is ready."
 *
 * Controller-side contract: on LOAD_MAP, latch the CellType bits from
 * each cell word into whatever internal wall storage the renderer /
 * physics modules use, then set STATUS_MAP_READY and self-clear the
 * CTRL bit.
 *
 * Since every cell here has CellType = BLANK and liquid = 0, this is
 * effectively the "clear the world" path.  It's still important to do
 * the write because the BRAM powers up with arbitrary contents. */
static int load_empty_map(void)
{
    printf("Loading empty cell map (%d cells)... ", GRID_N);
    fflush(stdout);

    uint32_t blank = PACK_CELL_BLANK(0);     /* CellType=BLANK, liquid=0 */

    for (int y = 0; y < GRID_H; y++) {
        for (int x = 0; x < GRID_W; x++) {
            if (cell_write(x, y, blank) < 0) return -1;
        }
    }

    /* Tell the controller the map is loaded. */
    if (ioctl(water_fd, WATER_SIM_LOAD_MAP) < 0) {
        fprintf(stderr, "LOAD_MAP ioctl failed: %s\n", strerror(errno));
        return -1;
    }
```

```
285
286     /* Wait for STATUS_MAP_READY (briefly).  This isn't strictly
287      * required --- the next operation would block on STATUS_BUSY
288      * anyway --- but it gives us a clear failure point if the
289      * controller never asserts MAP_READY. */
290     for (int tries = 0; tries < 1000; tries++) {
291         uint32_t status;
292         if (reg_read(WATER_REG_STATUS, &status) < 0) return -1;
293         if (status & STATUS_MAP_READY) {
294             printf("ready.\n");
295             return 0;
296         }
297         usleep(100);
298     }
299
300     fprintf(stderr, "MAP_READY not asserted within 100 ms after LOAD_MAP.\n");
301     return -1;
302 }
303
304 /* Configure auto-run mode, frame-lock, and enable VGA output. */
305 static int configure_runtime(void)
306 {
307     /* STEP_CFG_REG:
308      *   AUTO_RUN    = 1  →  controller advances on its own
309      *   FRAME_LOCK  = 1  →  controller waits for VSYNC before each tick
310      *   STEPS_PER_FRAME  →  number of evaluate+commit cycles per frame
311      */
312     uint32_t step_cfg = PACK_STEP_CFG(1, 1, STEPS_PER_FRAME);
313     if (reg_write(WATER_REG_STEP_CFG, step_cfg) < 0) return -1;
314
315     /* VGA_CTRL_REG: enable video. */
316     if (reg_write(WATER_REG_VGA_CTRL, VGA_CTRL_ENABLE) < 0) return -1;
317
318     printf("Auto-run enabled (%d step%s per frame, frame-locked).\n",
319            STEPS_PER_FRAME, STEPS_PER_FRAME == 1 ? "" : "s");
320     return 0;
321 }
322
323
324 /* ========================================================================
325  * Runtime --- mouse → brush translation
326  * ======================================================================== */
327
328 /* Send one brush command to the FPGA.  Wraps the WATER_SIM_BRUSH_APPLY
329  * ioctl, which in turn writes MOUSE_POS_REG, BRUSH_CFG_REG, and pulses
330  * CTRL_BRUSH_APPLY --- all in one driver-side operation. */
331 static void brush_apply(int x, int y, uint32_t tool, uint32_t liquid)
332 {
333     if (x < 0 || x >= GRID_W || y < 0 || y >= GRID_H) return;
334
335     water_sim_brush_t b = {
336         .grid_x = (uint32_t)x,
337         .grid_y = (uint32_t)y,
338         .tool   = tool,
339         .radius = BRUSH_RADIUS,
340         .liquid = liquid,
341     };
342
343     if (ioctl(water_fd, WATER_SIM_BRUSH_APPLY, &b) < 0) {
344         /* Non-fatal --- log and keep going.  A failed brush is preferable
345          * to crashing the whole loop because the user clicked weird. */
346         fprintf(stderr, "BRUSH_APPLY(%d, %d) failed: %s\n",
347                 x, y, strerror(errno));
348     }
349 }
350
351 /* Rasterise a wall along the line from (x0, y0) to (x1, y1), one cell
352  * at a time, via per-cell brush applies.  Standard Bresenham.  Used
353  * for right-click drag wall drawing so a fast mouse move doesn't leave
354  * gaps between sample points.
355  *
356  * The FPGA's brush is per-cell; it has no concept of "draw a line."
357  * Line interpolation is a software concern. */
358 static void draw_wall_line(int x0, int y0, int x1, int y1)
359 {
360     int dx =  (x1 > x0) ? (x1 - x0) : (x0 - x1);
361     int dy = -((y1 > y0) ? (y1 - y0) : (y0 - y1));
362     int sx = (x0 < x1) ? 1 : -1;
363     int sy = (y0 < y1) ? 1 : -1;
364     int err = dx + dy;
365
366     while (1) {
367         brush_apply(x0, y0, BRUSH_TOOL_WALL, 0);
368         if (x0 == x1 && y0 == y1) break;
```

```
        int e2 = 2 * err;
        if (e2 >= dy) { err += dy; x0 += sx; }
        if (e2 <= dx) { err += dx; y0 += sy; }
    }
}

/* Translate one mouse_event_t into FPGA brush command(s). */
static void handle_mouse_event(const mouse_event_t *ev)
{
    switch (ev->action) {

    case MOUSE_ADD_WATER:
        brush_apply(ev->grid_x, ev->grid_y,
                    BRUSH_TOOL_ADD, WATER_BRUSH_AMOUNT);
        break;

    case MOUSE_DRAW_WALL:
        /* If this is a continuation of a drag (prev_gx/prev_gy valid),
         * rasterise a line so fast mouse motion fills in cleanly.
         * Otherwise it's the first point of a new drag --- just one cell. */
        if (ev->prev_gx >= 0 && ev->prev_gy >= 0) {
            draw_wall_line(ev->prev_gx, ev->prev_gy,
                           ev->grid_x,  ev->grid_y);
        } else {
            brush_apply(ev->grid_x, ev->grid_y, BRUSH_TOOL_WALL, 0);
        }
        break;

    case MOUSE_ERASE:
        brush_apply(ev->grid_x, ev->grid_y, BRUSH_TOOL_ERASE, 0);
        break;

    case MOUSE_NONE:
    default:
        /* mouse_poll() can fill in MOUSE_NONE when the cursor moves
         * with no button held --- nothing to send to the FPGA. */
        break;
    }
}


/* ========================================================================
 * Shutdown
 * ======================================================================== */

static void teardown(void)
{
    if (water_fd >= 0) {
        /* Best effort: reset the FPGA so the next run starts clean.
         * Ignore errors --- we're exiting anyway. */
        ctrl_pulse(CTRL_RESET);
        close(water_fd);
        water_fd = -1;
    }
    mouse_cleanup();
}


int main(void)
{
    printf("Liquid sim --- HPS main loop starting.\n");

    install_signal_handlers();

    if (open_devices()      < 0) return 1;
    if (reset_fpga()        < 0) { teardown(); return 1; }
    if (load_empty_map()    < 0) { teardown(); return 1; }
    if (configure_runtime() < 0) { teardown(); return 1; }

    printf("\n");
    printf("Controls:\n");
    printf("  Left  click           : add water\n");
    printf("  Right click + drag    : draw walls\n");
    printf("  Middle click          : erase\n");
    printf("  Ctrl-C                : quit\n");
    printf("\n");
    printf("Simulation is auto-running on the FPGA.\n");
    printf("Watch the VGA monitor.\n\n");

    /* ------------------------------------------------------------------
     * Main loop
     *
     * mouse_poll() blocks on read(/dev/input/mice).
     * On Ctrl-C, the signal handler clears `running`, and SIGINT
```

```
     * causes the blocking read() to return -1 with errno = EINTR
     * We treat EINTR as the exit signal.
     * ----------------------------------------------------------------- */
    while (running) {
        mouse_event_t ev;

        if (mouse_poll(&ev) == 0) {
            handle_mouse_event(&ev);
        } else {
            /* mouse_poll returned -1.  Two likely causes:
             *   EINTR: signal arrived --- loop will see !running and exit.
             *   anything else: real I/O error --- log and bail. */
            if (errno != EINTR) {
                perror("mouse_poll");
                break;
            }
        }
    }

    printf("\nShutting down...\n");
    teardown();
    printf("Goodbye.\n");
    return 0;
}
```

Listing 6: main.c code. HPS setup, runtime loop, and brush-command interface.

```
/* ============================================================================
 * mouse_io.c --- USB mouse input module for the water simulator
 *
 * Reads 3-byte PS/2 packets from /dev/input/mice, accumulates relative
 * deltas into a pixel-space cursor (640×480), and converts to grid
 * coordinates (64×64).
 *
 * Provides a simple polling API --- see mouse_io.h for usage.
 * ============================================================================ */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include "mouse_io.h"

/* ---------------------------------------------------------------------------
 * Screen and grid constants
 * --------------------------------------------------------------------------- */
#define SCREEN_W    640
#define SCREEN_H    480
#define GRID_W      64
#define GRID_H      64

/* PS/2 byte-0 button masks */
#define BTN_LEFT    0x01
#define BTN_RIGHT   0x02
#define BTN_MIDDLE  0x04

/* ---------------------------------------------------------------------------
 * Internal state  (file-scoped --- one mouse per process)
 * --------------------------------------------------------------------------- */
static int fd_mouse = -1;

/* Pixel-space cursor (accumulated from relative deltas) */
static int cur_x = SCREEN_W / 2;
static int cur_y = SCREEN_H / 2;

/* Previous grid position for right-click drag tracking */
static int drag_gx = -1;
static int drag_gy = -1;

/* ---------------------------------------------------------------------------
 * Helpers
 * --------------------------------------------------------------------------- */
static inline int clamp(int v, int lo, int hi)
{
  if (v < lo) return lo;
  if (v > hi) return hi;
  return v;
}

/* ---------------------------------------------------------------------------
 * Public API
 * --------------------------------------------------------------------------- */

int mouse_init(const char *device_path)
{
```

```
  fd_mouse = open(device_path, O_RDONLY);
  if (fd_mouse < 0) {
    perror(device_path);
    return -1;
  }

  /* Reset cursor to screen center */
  cur_x = SCREEN_W / 2;
  cur_y = SCREEN_H / 2;
  drag_gx = -1;
  drag_gy = -1;

  return 0;
}

int mouse_poll(mouse_event_t *ev)
{
  unsigned char buf[3];
  ssize_t n;

  /* Blocking read of one 3-byte PS/2 packet */
  n = read(fd_mouse, buf, 3);
  if (n != 3)
    return -1;

  /* Signed movement deltas */
  int dx = (int)(signed char)buf[1];
  int dy = (int)(signed char)buf[2];

  /* Accumulate pixel cursor.
     PS/2: positive Y = mouse pushed away (physical up).
     Screen: positive Y = downward.  So subtract dy. */
  cur_x = clamp(cur_x + dx, 0, SCREEN_W - 1);
  cur_y = clamp(cur_y - dy, 0, SCREEN_H - 1);

  /* Convert to grid coordinates */
  int gx = clamp(cur_x * GRID_W / SCREEN_W, 0, GRID_W - 1);
  int gy = clamp(cur_y * GRID_H / SCREEN_H, 0, GRID_H - 1);

  ev->grid_x  = gx;
  ev->grid_y  = gy;
  ev->prev_gx = -1;
  ev->prev_gy = -1;
  ev->action  = MOUSE_NONE;

  /* ---- Determine action from button state ---- */

  if (buf[0] & BTN_LEFT) {
    ev->action = MOUSE_ADD_WATER;
  }

  if (buf[0] & BTN_RIGHT) {
    ev->action  = MOUSE_DRAW_WALL;
    ev->prev_gx = drag_gx;     /* -1 on first point */
    ev->prev_gy = drag_gy;
    drag_gx = gx;
    drag_gy = gy;
  } else {
    /* Right button released --- reset drag state */
    drag_gx = -1;
    drag_gy = -1;
  }

  if (buf[0] & BTN_MIDDLE) {
    ev->action = MOUSE_ERASE;
  }

  return 0;
}

void mouse_cleanup(void)
{
  if (fd_mouse >= 0) {
    close(fd_mouse);
    fd_mouse = -1;
  }
  drag_gx = -1;
  drag_gy = -1;
}
```

Listing 7: USB mouse packet handling and grid-event translation, mouse-io.c code.

```
/* ========================================================================
 * mouse_io.h --- USB mouse input module for the water simulator
```

```
 *
 * Reads /dev/input/mice, tracks cursor position in pixel space,
 * converts to 64×64 grid coordinates, and reports button actions.
 *
 * Usage from main loop (Daanish's main.c):
 *
 *     #include "mouse_io.h"
 *
 *     mouse_init("/dev/input/mice");
 *
 *     while (running) {
 *         mouse_event_t ev;
 *         if (mouse_poll(&ev) == 0) {
 *             if (ev.action == MOUSE_ADD_WATER)
 *                 // write ev.grid_x, ev.grid_y to FPGA ...
 *             if (ev.action == MOUSE_DRAW_WALL)
 *                 // draw wall line from (ev.prev_gx,ev.prev_gy)
 *                 //                 to (ev.grid_x, ev.grid_y) ...
 *         }
 *     }
 *
 *     mouse_cleanup();
 * ========================================================================== */

#ifndef _MOUSE_IO_H
#define _MOUSE_IO_H

/* ------------------------------------------------------------------------
 * Action codes --- what the user intends to do this event
 * ------------------------------------------------------------------------ */
typedef enum {
  MOUSE_NONE       = 0, /* mouse moved but no relevant button held   */
  MOUSE_ADD_WATER  = 1, /* left  click  --- spawn water               */
  MOUSE_DRAW_WALL  = 2, /* right click  --- draw solid wall           */
  MOUSE_ERASE      = 3  /* middle click --- erase cell (set to blank)  */
} mouse_action_t;

/* ------------------------------------------------------------------------
 * Event reported by mouse_poll()
 *
 * grid_x / grid_y  :  current cursor in grid coordinates [0..63]
 * prev_gx / prev_gy:  previous grid position when dragging (DRAW_WALL).
 *                     Set to -1 on the first point of a new drag.
 *                     The caller should use Bresenham between
 *                     (prev_gx, prev_gy) → (grid_x, grid_y) to
 *                     fill in the wall line without gaps.
 * ------------------------------------------------------------------------ */
typedef struct {
  mouse_action_t action;
  int grid_x;        /* current  grid column [0 .. GRID_W-1] */
  int grid_y;        /* current  grid row    [0 .. GRID_H-1] */
  int prev_gx;       /* previous grid column (-1 = drag start) */
  int prev_gy;       /* previous grid row    (-1 = drag start) */
} mouse_event_t;

/* ------------------------------------------------------------------------
 * API
 * ------------------------------------------------------------------------ */

/*
 * mouse_init --- open the mouse device and reset cursor to screen center.
 *
 *   device_path : typically "/dev/input/mice"  (needs root or input group)
 *   returns     : 0 on success, -1 on error (errno set)
 */
int mouse_init(const char *device_path);

/*
 * mouse_poll --- blocking read of one 3-byte PS/2 packet.
 *
 * Updates the internal cursor, converts to grid coordinates, and fills
 * in *ev with the resulting action and position.
 *
 *   returns : 0 on success, -1 on read error
 */
int mouse_poll(mouse_event_t *ev);

/*
 * mouse_cleanup --- close the mouse device and reset state.
 */
void mouse_cleanup(void);

#endif /* _MOUSE_IO_H */
```

Listing 8: USB mouse packet handling and grid-event translation, mouse-io.h code.

```c
/* ============================================================================
 * water_sim.h --- Shared kernel / userspace header for the water-simulator
 *                 FPGA peripheral on the DE1-SoC.
 *
 * Defines:
 *   - Avalon-MM register map        (Table 1, VGA_Water_Design §5)
 *   - CTRL / STATUS / BRUSH bit-fields
 *   - 32-bit GRID_MEM cell word layout (Table 2)
 *   - ioctl command numbers and argument structs
 *
 * Include from the kernel driver (water_sim_drv.c) and from any userspace
 * program that talks to /dev/water_sim (mouse_io.c).
 * ============================================================================ */

#ifndef _WATER_SIM_H
#define _WATER_SIM_H

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

/* ----------------------------------------------------------------------------
 * Avalon-MM register byte offsets  (Table 1, design document §5)
 *
 *  Offset      Name             Access    Description
 *  ----------  ---------------  ------    -----------------------------------
 *  0x0000      CTRL_REG         W         Write-one-pulse control bits
 *  0x0004      STATUS_REG       R         Busy, done, phase, debug
 *  0x0008      GRID_SIZE_REG    R/W       Grid W [15:0], H [31:16]
 *  0x000C      MOUSE_POS_REG    W         Cell coords:  X [15:0], Y [31:16]
 *  0x0010      BRUSH_CFG_REG    W         Tool, radius, liquid amount
 *  0x0014      STEP_CFG_REG     R/W       Auto-run, frame-lock, steps/frame
 *  0x0018      TICK_COUNT_REG   R         Free-running tick counter
 *  0x0040      VGA_CTRL_REG     R/W       Video enable, clear, debug view
 *  0x1000--     GRID_MEM         R/W       4096 × 32-bit packed cell words
 * ---------------------------------------------------------------------------- */
#define WATER_REG_CTRL          0x0000
#define WATER_REG_STATUS        0x0004
#define WATER_REG_GRID_SIZE     0x0008
#define WATER_REG_MOUSE_POS     0x000C
#define WATER_REG_BRUSH_CFG     0x0010
#define WATER_REG_STEP_CFG      0x0014
#define WATER_REG_TICK_COUNT    0x0018
#define WATER_REG_VGA_CTRL      0x0040
#define WATER_GRID_MEM_BASE     0x1000
#define WATER_GRID_MEM_SIZE     (4096 * 4)      /* 64×64 cells × 4 bytes */

/* Total byte span the driver must memory-map */
#define WATER_PERIPH_SPAN       (WATER_GRID_MEM_BASE + WATER_GRID_MEM_SIZE)

/* ----------------------------------------------------------------------------
 * CTRL_REG  [0x0000]  --- write-one-pulse command bits
 *
 * Each bit is self-clearing in the FPGA; software writes 1 to fire,
 * the Global Controller acts on it, then hardware clears it.
 * ---------------------------------------------------------------------------- */
#define CTRL_STEP           (1u << 0)   /* Start one evaluate+commit tick  */
#define CTRL_RESET          (1u << 1)   /* Soft-reset controller + grid    */
#define CTRL_LOAD_MAP       (1u << 2)   /* Latch CellType bits → Line Mem  */
#define CTRL_DONE_ACK       (1u << 3)   /* Acknowledge & clear sticky DONE */
#define CTRL_BRUSH_APPLY    (1u << 4)   /* Apply MOUSE_POS + BRUSH_CFG     */

/* ----------------------------------------------------------------------------
 * STATUS_REG  [0x0004]  --- read-only status
 * ---------------------------------------------------------------------------- */
#define STATUS_BUSY         (1u << 0)
#define STATUS_DONE         (1u << 1)
#define STATUS_MAP_READY    (1u << 2)
#define STATUS_PHASE_SHIFT  3
#define STATUS_PHASE_MASK   (0x3u << STATUS_PHASE_SHIFT)

/* ----------------------------------------------------------------------------
 * MOUSE_POS_REG  [0x000C]
 *   [15: 0]  grid X  (column, 0-based)
 *   [31:16]  grid Y  (row,    0-based)
 * ---------------------------------------------------------------------------- */
```

```
#define PACK_MOUSE_POS(x, y) \
    (((x) & 0xFFFFu) | (((y) & 0xFFFFu) << 16))

/* ------------------------------------------------------------------------
 * BRUSH_CFG_REG  [0x0010]
 *   [ 1: 0]  tool      --- 0 = none, 1 = add water, 2 = erase, 3 = wall
 *   [15: 8]  radius    --- brush radius in cells (0 = single cell)
 *   [31:16]  liquid    --- Q2.16 unsigned amount to inject (add-water only)
 * ------------------------------------------------------------------------ */
#define BRUSH_TOOL_NONE     0u
#define BRUSH_TOOL_ADD      1u
#define BRUSH_TOOL_ERASE    2u
#define BRUSH_TOOL_WALL     3u

#define BRUSH_RADIUS_SHIFT  8
#define BRUSH_LIQUID_SHIFT  16

#define PACK_BRUSH_CFG(tool, radius, liquid) \
    (((tool) & 0x3u)                          | \
     (((radius) & 0xFFu) << BRUSH_RADIUS_SHIFT) | \
     (((liquid) & 0xFFFFu) << BRUSH_LIQUID_SHIFT))

/* ------------------------------------------------------------------------
 * STEP_CFG_REG  [0x0014]
 *   [0]      AUTO_RUN    --- 1 = auto-step after each VGA frame
 *   [1]      FRAME_LOCK  --- 1 = wait for VSYNC before starting next tick
 *   [15:8]   STEPS_PER_FRAME --- number of sim ticks per video frame
 * ------------------------------------------------------------------------ */
#define STEP_CFG_AUTO_RUN       (1u << 0)
#define STEP_CFG_FRAME_LOCK     (1u << 1)
#define STEP_CFG_SPF_SHIFT      8

#define PACK_STEP_CFG(auto_run, frame_lock, steps_per_frame) \
    (((auto_run) ? 1u : 0u)                                | \
     ((frame_lock) ? 2u : 0u)                               | \
     (((steps_per_frame) & 0xFFu) << STEP_CFG_SPF_SHIFT))

/* ------------------------------------------------------------------------
 * VGA_CTRL_REG  [0x0040]
 *   [0]  Video enable
 *   [1]  Clear visible liquid layer (without touching wall map)
 *   [2]  Debug rendering mode (show Settled / isDownFlowing flags)
 * ------------------------------------------------------------------------ */
#define VGA_CTRL_ENABLE         (1u << 0)
#define VGA_CTRL_CLEAR_LIQUID   (1u << 1)
#define VGA_CTRL_DEBUG_MODE     (1u << 2)

/* ------------------------------------------------------------------------
 * Grid constants
 * ------------------------------------------------------------------------ */
#define GRID_W      64
#define GRID_H      64
#define GRID_N      (GRID_W * GRID_H)   /* 4096 cells */

/* ------------------------------------------------------------------------
 * 32-bit GRID_MEM cell word layout  (Table 2, design document §5)
 *
 *   Bits     Field            Width   Meaning
 *   ------   --------------   -----   -------------------------------------
 *   [0]      CellType         1 b     0 = Blank, 1 = Solid
 *   [18:1]   Liquid           18 b    Q2.16 unsigned liquid amount
 *   [19]     Settled          1 b     Quiescent flag
 *   [23:20]  SettleCount      4 b     Settling-heuristic counter
 *   [24]     isDownFlowing    1 b     VGA stream-drawing hint
 *   [31:25]  Reserved         7 b
 * ------------------------------------------------------------------------ */
#define CELL_TYPE_BIT       0
#define CELL_LIQUID_LO      1
#define CELL_LIQUID_HI      18
#define CELL_SETTLED_BIT    19
#define CELL_SETTLE_LO      20
#define CELL_SETTLE_HI      23
#define CELL_DOWN_FLOW_BIT  24

#define CELL_TYPE_BLANK     0u
#define CELL_TYPE_SOLID     1u

/* Q2.16 literal helpers */
#define Q216_ONE            0x00010000u          /* 1.0 in Q2.16 */
#define Q216_HALF           0x00008000u          /* 0.5 in Q2.16 */

/* Pack a blank cell with a given Q2.16 liquid amount */
#define PACK_CELL_BLANK(liquid) \
    (CELL_TYPE_BLANK | (((liquid) & 0x3FFFFu) << CELL_LIQUID_LO))
```

```c

/* Pack a solid (wall) cell --- no liquid */
#define PACK_CELL_SOLID     (CELL_TYPE_SOLID)

/* ------------------------------------------------------------------------
 * ioctl interface --- argument structures
 * ------------------------------------------------------------------------ */

/* Generic register read / write */
typedef struct {
    uint32_t reg;           /* byte offset (e.g. WATER_REG_CTRL) */
    uint32_t value;         /* value to write, or value read back */
} water_sim_reg_t;

/* High-level brush command (MOUSE_POS + BRUSH_CFG + apply, in one call) */
typedef struct {
    uint32_t grid_x;        /* cell column [0 .. GRID_W-1] */
    uint32_t grid_y;        /* cell row    [0 .. GRID_H-1] */
    uint32_t tool;          /* BRUSH_TOOL_ADD / ERASE / WALL */
    uint32_t radius;        /* brush radius in cells        */
    uint32_t liquid;        /* Q2.16 liquid amount (add-water only) */
} water_sim_brush_t;

/* Single grid-cell read / write */
typedef struct {
    uint32_t x, y;          /* cell coordinates */
    uint32_t cell_word;     /* packed 32-bit word (Table 2)  */
} water_sim_cell_t;

/* ------------------------------------------------------------------------
 * ioctl command numbers
 * ------------------------------------------------------------------------ */
#define WATER_SIM_MAGIC         'w'

#define WATER_SIM_WRITE_REG     _IOW (WATER_SIM_MAGIC, 1, water_sim_reg_t)
#define WATER_SIM_READ_REG      _IOWR(WATER_SIM_MAGIC, 2, water_sim_reg_t)
#define WATER_SIM_BRUSH_APPLY   _IOW (WATER_SIM_MAGIC, 3, water_sim_brush_t)
#define WATER_SIM_STEP          _IO  (WATER_SIM_MAGIC, 4)
#define WATER_SIM_RESET         _IO  (WATER_SIM_MAGIC, 5)
#define WATER_SIM_LOAD_MAP      _IO  (WATER_SIM_MAGIC, 6)
#define WATER_SIM_WRITE_CELL    _IOW (WATER_SIM_MAGIC, 7, water_sim_cell_t)
#define WATER_SIM_READ_CELL     _IOWR(WATER_SIM_MAGIC, 8, water_sim_cell_t)
#define WATER_SIM_READ_STATUS   _IOR (WATER_SIM_MAGIC, 9, uint32_t)

#endif /* _WATER_SIM_H */
```

Listing 9: Shared userspace/kernel register map and ioctl definitions.