Design Document for Liquid-Physics-Simulator

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

April 17, 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).

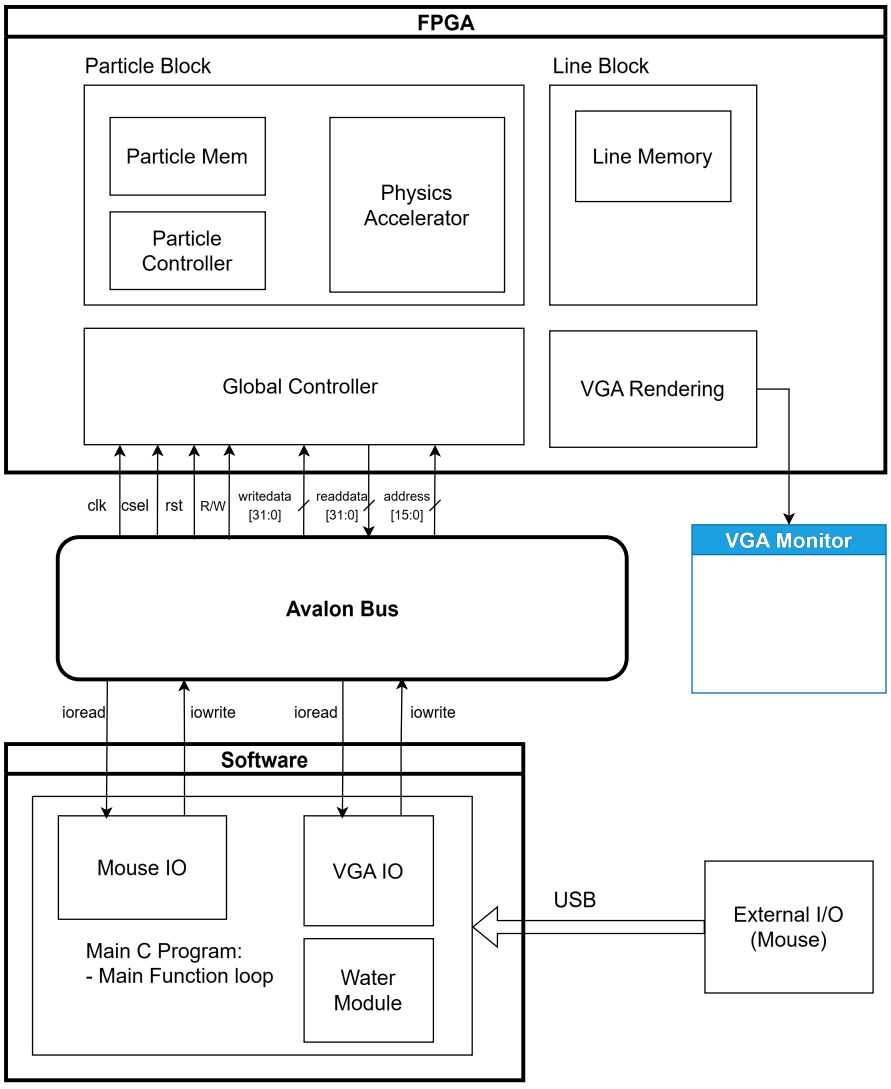

Figure 1: System Block Diagram

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 3.

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 will be using 32 bits, or 4 bytes of data to represent each pixel particle. All arithmetic will be in 18×18 fixed-point format to fit the 18×18 multipliers on the board and limit resource demand. If we choose a 64×64 bit display for our VGA, our maximum amount of possible particles will be 4096, with 4 bytes per particle making our maximum particle requirement **131,072 Kb** or **16.384 KB of memory**. With the same amount of 18 bit arrays for arithmetic we have

another $64 \times 64 \times 18 = 73{,}728$ Kb of memory. Putting these together we estimate about **20** M10K memory blocks needed, only 5% of the total on the Cyclone chip.

Another hardware particle physics implementation found in a published IEEE symposium on Field-Programmable Custom Computing Machines paper [2] has a resource utilization found in figure 3 using an FPGA with a much smaller resource budget. A notable difference is that this group used a 3-D scale (while our system is 2-D) and held additional particle variables, so we used these synthesis results as a general guideline for our resource estimates. We would expect our budget to fall well below the marks of figure 3.

	Logic cells	Registers	18x18 Mult.
Avail. in FPGA	41,250	44,860	56
Entire Pipe	30,283	27,853	28
Force System	5,157	4,962	3
Coll. Detection	2,472	2,301	0
Coll. Response	12,699	12,031	12
Rendering	8,382	7,034	13
Nios μController	6,397	2424	1

Figure 3: Beeckler and Gross FPGA Synthesis Results

5 Hardware/Software Interface

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.

Register map

Byte offset	Register name	Description
0x0000	CTRL_REG	Write-one-pulse control bits for step, reset, map load, done acknowledge, and brush apply.
0x0004	STATUS_REG	Reports busy, done, map-ready, current FSM phase, and optional debug count fields.
0x0008	GRID_SIZE_REG	Holds grid width and height in cells; default is 64×64.
0x000C	MOUSE_POS_REG	Current mouse position expressed as simulation-cell coordinates (x,y).
0x0010	BRUSH_CFG_REG	Selects the brush tool, brush radius, and liquid amount used by interactive edits.
0x0014	STEP_CFG_REG	Configures auto-run, frame lock, and the number of simulation steps per frame.
0x0018	TICK_COUNT_REG	Read-only counter of completed simulation ticks.
0x0040	VGA_CTRL_REG	Controls video enable, liquid-layer clear, and normal versus debug display mode.
0x1000--0x4FFC	GRID_MEM	4096-word BRAM window storing the packed 64×64 cell array.

Table 1: Proposed 32-bit Avalon-MM register map for the water simulator peripheral.

The table above keeps the register map compact, while the detailed behavior of each register is described here in prose. CTRL_REG is the main command port. CTRL_REG[0] starts one full simulation tick, meaning one evaluate pass followed by one commit pass. CTRL_REG[1] performs a soft reset of the controller and clears the liquid state. CTRL_REG[2] issues LOAD_MAP, which copies the CellType bits currently stored in GRID_MEM into the internal wall or line memory used by the physics and VGA logic. CTRL_REG[3] acknowledges and clears the sticky DONE condition, and CTRL_REG[4] applies the brush command currently stored in MOUSE_POS_REG and BRUSH_CFG_REG. All of these control bits are intended to be write-one pulses rather than persistent mode bits.

STATUS_REG is the corresponding readback register. STATUS_REG[0] is BUSY while the global controller is sweeping the grid. STATUS_REG[1] is DONE after a full simulation tick completes and remains asserted until software acknowledges it through CTRL_REG[3]. STATUS_REG[2] is MAP_READY, indicating that the current wall map has been latched internally. Bits [4:3] can be used to report the current FSM phase such as idle, evaluate, commit, or vertical-blank wait. The upper bits may also be used for optional debug fields such as a free-running counter.

The remaining low-address registers carry runtime parameters from software. GRID_SIZE_REG stores GRID_W in bits [15:0] and GRID_H in bits [31:16]; for the current design both are expected to default to 64. MOUSE_POS_REG stores the mouse position already converted by software into simulation-cell coordinates, so the FPGA never needs to interpret raw USB packets directly. BRUSH_CFG_REG holds the interactive edit command: bits [1:0] select the tool (00 none, 01 add water, 10 erase water, 11 draw wall), bits [15:8] store the brush radius in cells, and bits [31:16] store the amount of liquid to inject as an unsigned fractional value. STEP_CFG_REG controls the scheduler, where bit [0] enables AUTO_RUN, bit [1] enables FRAME_LOCK, and bits [15:8] may hold STEPS_PER_FRAME. TICK_COUNT_REG is a read-only counter incremented after every completed simulation tick. VGA_CTRL_REG controls presentation only: bit [0] enables video output, bit [1] clears the visible liquid layer without changing the wall map, and bit [2] selects between a normal fill view and a debug rendering mode based on flags such as Settled and isDownFlowing.

The bulk state is stored in GRID_MEM, a BRAM-backed memory window beginning at byte address 0x1000. This region contains 4096 words, one for each cell in the 64×64 grid, and cell (x, y) is stored at byte address 0x1000 + 4*(y*64 + x). Software writes this window during initialization to define blank and solid cells and to seed any initial liquid values. It may also read the same region back for

debug, state dumps, or software-side inspection.

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.

In software, the typical sequence is: (1) write the initial 64×64 map into GRID_MEM; (2) pulse CTRL_REG[2] so the internal wall memory is updated; (3) write MOUSE_POS_REG, BRUSH_CFG_REG, and VGA_CTRL_REG as needed during runtime; (4) either pulse CTRL_REG[0] for a single step or enable AUTO_RUN; and (5) poll STATUS_REG or TICK_COUNT_REG to determine when the next frame is ready.

References

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

[2] J.S. Beeckler and W.J. Gross. Fpga particle graphics hardware. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 85–94, 2005.