

VeriWave: An FPGA Wave Propagation Visualizer

Final Report

1st Collin Lung

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
lung1@mit.edu

2nd Isaiah Nathaniel

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
isaiah29@mit.edu

Abstract—We present a real-time FPGA-based Wave Propagation Visualizer that simulates two-dimensional wave behavior using a finite-difference method implemented entirely in hardware. Simulating wave propagation is computationally intensive, requiring millions of stencil updates and repeated neighborhood reads per frame; on CPUs and GPUs this often results in high power consumption and memory bandwidth pressure, making low-latency, always-on interactive systems non-trivial to deploy. Our design addresses these challenges by optimizing wave updates through a fixed-point computation pipeline and streaming wave-state data to and from external DDR3 memory, where the simulation grid is stored as a single contiguous block to maximize burst-transfer efficiency and sustain the required update rate. The system incorporates a camera interface that detects a user’s fingertip and injects a sinusoidal wave source at the corresponding location, enabling live interactive manipulation of the simulation domain. A BRAM-backed sine generator provides smooth periodic excitation, and the fixed-point update unit computes spatial derivatives from local neighborhoods each time step. A hardware color-mapping pipeline converts the evolving wave field amplitudes into RGB pixel values for real-time display output. We implement the complete system on the RealDigital Urbana FPGA and evaluate responsiveness, numerical stability, and visualization quality under live user input. We additionally discuss design tradeoffs in memory bandwidth, camera pipeline latency, and numerical precision, and outline potential extensions for higher resolutions and multi-source interactions.

Index Terms—Field-programmable gate arrays (FPGAs), fixed-point arithmetic, wave propagation, finite-difference methods, hardware accelerators, real-time systems, DRAM buffering, computer vision, interactive visualization

I. PHYSICAL CONSTRUCTION

The Wave Propagation Visualizer (WPV) consists of the following major hardware subsystems:

- **RealDigital Urbana FPGA** This serves as the central computation platform, hosting the fixed-point arithmetic unit, the finite-difference wave update pipeline, the sine-wave generator, the camera interface logic, and the external DDR3 memory interface.
- **External DDR3 Memory** The simulation’s wave field is stored in external DDR3 memory as a single contiguous block of memory. The update pipeline reads from and writes to this external DDR3 memory.
- **External Camera Module** A standard OV5640 camera is used to acquire live video frames. A dedicated camera interface block captures raw pixel data feed from the

camera and forwards it to the downstream module that determines the wave-source injection location.

- **Internal BRAM** A BRAM-resident sine lookup table, driven by a phase accumulator, produces a continuous sinusoidal excitation signal. When the fingertip detection module reports a valid coordinate, this sinusoidal value is injected directly into the wave-height field at the corresponding grid location in DDR3.
- **Wave Display / HDMI Output Hardware** A hardware color-mapping pipeline converts simulated wave amplitudes into RGB pixel values. These values are streamed to the HDMI display hardware at video rate, enabling smooth real-time visualization of wave propagation.

Although the prototype is currently assembled using the bare RealDigital Urbana FPGA board with direct connections to the camera and display, future revisions may incorporate a dedicated enclosure to improve durability, maintain consistent camera alignment, and provide cable strain relief during operation.

II. DERIVING THE FINITE-DIFFERENCE WAVE EQUATION

The evolving wave field is governed by the following second order partial differential wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \alpha \frac{\partial u}{\partial t} \quad (1)$$

where $u(x, y, t)$ is the wave height, c is the wave propagation speed, and α is a damping factor used to stabilize the simulation [1], [2], [4].

We discretize the equation using a second-order finite-difference stencil. The Laplacian $\nabla^2 u$ is approximated using the four direct neighbors of each grid cell:

$$\nabla^2 u_{i,j} \approx u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}. \quad (2)$$

This corresponds to a 3×3 neighborhood around (i, j) , where the center cell and its four orthogonal neighbors form the update stencil.

A. Discrete Update Rule

Let $u_{i,j}^{(n)}$ denote the wave height at position (i,j) and time step n . The update rule implemented in hardware is:

$$u_{i,j}^{(n+1)} = 2u_{i,j}^{(n)} - u_{i,j}^{(n-1)} + c^2 \Delta t^2 \left(u_{i+1,j}^{(n)} + u_{i-1,j}^{(n)} + u_{i,j+1}^{(n)} + u_{i,j-1}^{(n)} - 4u_{i,j}^{(n)} \right) - \gamma \left(u_{i,j}^{(n)} - u_{i,j}^{(n-1)} \right) + S_{i,j}^{(n)}. \quad (3)$$

where:

- $S_{i,j}^{(n)}$ is the sinusoidal source term (nonzero only at the fingertip location).
- γ is the discrete damping coefficient,
- Δt is the time step size enforced by the simulation rate.

This formulation requires only the current frame $u^{(n)}$, the previous frame $u^{(n-1)}$, and the four neighbors of each cell, making it well-suited for FPGA implementation. The buffer fetch module reads these five values from DDR3 memory, the fixed-point arithmetic unit applies the stencil, and the resulting $u^{(n+1)}$ values are forwarded to the HDMI output pipeline and written back to memory.

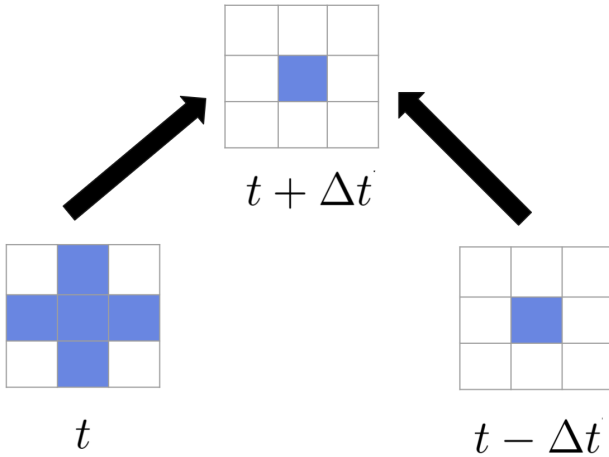


Fig. 1. Data dependencies for update rule

This tightly integrated pipeline allows a user to generate ripples, wave fronts, and interference patterns in real time simply by moving their fingertip in front of the camera.

III. WAVE SOURCE INJECTION

To create an interactive wave field, the system processes a live video stream from the OV5640 camera to determine where the user's fingertip is located. The detected fingertip location is used to inject a sinusoidal disturbance into the wave simulation at the corresponding grid cell. This requires two components: (1) real-time image processing to isolate the fingertip, and (2) center-of-mass calculation.

A. Fingertip Isolation

The camera interface reconstructs incoming pixel values and crosses clock domains from `clk_camera` to `clk_pixel`. This data is then passed through a Gaussian blur convolution module to reduce noise.

To make fingertip detection robust and repeatable, we constructed a simple glove with a bright pink fingertip cover. We design our mask to respond to this color by using a chroma-style red/pink detector similar to the lab05 pipeline from the course. Each pixel is converted to luminance and chroma components and then passed through a thresholding stage tuned to isolate the pink fingertip region.

A binary mask is produced from this thresholding operation. The system then computes both the mass of the mask and the center-of-mass of the mask. These two values provide a robust fingertip coordinate derived from the pink fingertip region rather than arbitrary skin tone or background clutter.

A valid fingertip location corresponds to a center-of-mass coordinate where the total mass meets a certain lower bound. This enforces how close a user's finger must be from the camera to register. If a valid fingertip location (x_s, y_s) is detected, the system injects a sinusoidal source value into that cell of the wave field. Details of how these sinusoidal source values are generated will be discussed in Section IV.

B. Data Pipeline in Detail

The pipeline executes the following stages:

- **Frame Capture** The OV5640 camera provides a continuous stream of pixel data in raster order. A camera interface module reconstructs pixel values and crosses clock domains.
- **Gaussian Blur** Since the camera feed is inherently noisy, the pixel data is passed through a Gaussian blur module, which smooths out sensor noise. This module internally uses BRAM for its line buffer.
- **Thresholding and Mask Generation** The post-blur pixel stream is converted to a luminance–chroma representation and passed through a color threshold tuned to the pink fingertip of our custom glove. Pixels that fall inside this chroma-red/pink band are marked as foreground, producing a binary fingertip mask.
- **Center-of-Mass and Mass Calculation** The coordinates of the fingertip are estimated by computing the center of mass of the binary mask. The total mass (sum of foreground pixels) is also accumulated as an intermediate value. This provides a robust estimate even when the fingertip spans multiple pixels or has a slightly irregular shape.
- **Source Select Module** Once detected, the fingertip coordinates are forwarded to the wave-source-injection/source-select module. If a valid fingertip is present, a sinusoidal excitation derived from a BRAM-based lookup table is written into the wave state at the corresponding grid location. To maintain a valid 3×3 stencil, fingertip locations that fall on the simulation border are rejected: border pixels are forced to zero and

are never treated as valid source locations, and the module only marks coordinates as valid during `active_draw` intervals. To prevent the simulation from being continuously overdriven when the fingertip remains in one place, the source-select module also enforces a configurable cooldown counter, `COOLDOWN_FRAMES`: after injecting a source on a given frame, the module waits a fixed number of frames before allowing another injection. For the double-slit experiment, we additionally treat wall regions as hard boundaries by clamping the wave height to zero whenever propagation reaches a wall cell; the detailed construction of these walls and slits is described in a later section.

This processing pipeline allows the user to create ripples, interference patterns, and wave fronts directly on the simulated surface by moving their finger in front of the camera. Because the entire pipeline is hardware-accelerated, interaction remains smooth and responsive even at the high update rates required by the wave solver.

IV. SINE WAVE GENERATOR

To inject a stable periodic source into the wave simulation, our hardware implements a deterministic sine wave generator. Instead of evaluating $\sin(\theta)$ directly, which would be extremely expensive in hardware, we precompute a high-resolution lookup table stored in BRAM. This approach allows the system to output a continuous sinusoidal waveform using only integer arithmetic and a simple address indexing scheme.

A. Sine Wave Lookup Table

The lookup table contains uniformly sampled points from one full period of a sine wave. The system keeps track of a phase index which corresponds to which value of the precomputed lookup table the source injection should use. The phase index is incremented with wrap-around upon receiving a `new_frame` signal. By adjusting the number of samples for the look up table, the generator produces different output frequencies and subsequently different simulation behavior.

B. Selecting Number of Samples

All lookup table entries are precomputed offline in Python using the same fixed-point format as the wave simulation (signed Q1.7.17). This ensures numerical compatibility when the sine source is added directly into the finite-difference update equation. We evaluated lookup tables of size 20, 60, and 120 samples per period to compare BRAM usage and waveform smoothness. Although 120 samples produces the most visually continuous waveform, it requires additional memory, and 20 samples introduces noticeable quantization noise.

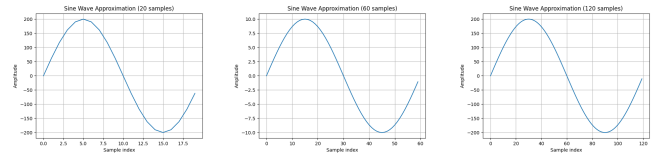


Fig. 2. Comparison of sine lookup table resolutions. Left: 20 samples per period produces visible quantization. Middle: 60 samples per period (our final choice) balances smoothness and BRAM usage. Right: 120 samples per period yields a smooth waveform but uses more memory.

After comparing all options, we selected a 60-sample lookup table as the final design point. It provides a smooth-looking sinusoid when injected into the wave field and rendered on the HDMI display, while consuming only a modest amount of BRAM.

This lookup table based oscillator integrates cleanly into the simulation pipeline, offering a compact, stable, and low-overhead source signal for exciting the wave field.

V. STATE BUFFERS

As stated previously, our finite difference method requires two frames in time to calculate the next value. This requires a tunable memory interface to tackle this.

A. State Buffer Storage Format

In order to simplify the memory interface, a singular DDR3 Memory entity is used instead of storing the two state buffers in separate DDR3 memory entities. Furthermore, in order to avoid a more complicated protocol that exchanges addresses, we organized our data so that the AXI Streaming protocol is sufficient.

The following format is used to pack the values for a singular coordinate:



The padding bits are needed to ensure that the memory interface has no trouble packing adjacent values into a 128-bit transaction. This produces a deterministic and predictable read pattern.

B. Buffer Write

The Buffer Write module contains the following sub-components:

- **Tlast Logic** The logic tags the last values of a frame with a bit. The purpose of this is to allow downstream address synchronization in the Traffic Controller to avoid error propagation.
- **Stacker** Since the UberDDR3 module expects data to be packed in 128-bit transactions, the role of the stacker is to take 64-bit packed values and stack them into a 128-bit transaction.
- **Clock Crossing AXIS FIFO** The Clock Crossing AXIS FIFO then takes care of crossing from the `clk_pixel` domain to the `clk_controller` domain. This FIFO is

how the traffic controller receives the values to be written to memory.

C. Buffer Fetch

The Buffer Fetch module contains the following sub-components:

- **Clock Crossing AXIS FIFO** The Clock Crossing AXIS FIFO takes care of crossing from the `clk_controller` domain to the `clk_pixel` domain. This FIFO is how the core pipeline receives the values read from memory and outputted by the traffic controller.
- **Unstacker** Since the UberDDR3 module outputs data in a packed 128-bit transactions, the role of the unstacker is to take to 128-bit transactions values and return original 64 bit values.
- **Tready Logic** The pipeline assumes that the unstacker will always have some valid output in steady state. With that said, some synchronization errors can occur. In order to isolate these errors, the Tready logic ensures two things: (1) once a Tlast bit is seen, the pipeline only consumes data when the last pixel of a frame is being drawn, (2) the pipeline only consumes data when pixels are being drawn
- **Line Buffer** As seen in fig.1, the calculations for the next amplitude value relies on a 3×3 stencil. As such, a line buffer is used to cache and store previous lines so that this stencil can be updated accordingly. This module internally uses BRAM to store 4 lines of the image.

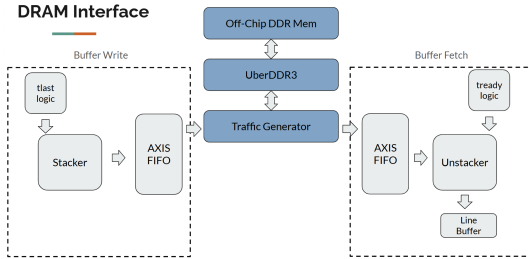


Fig. 3. Resulting DRAM interface

VI. FIXED POINT UNIT

Before implementing the finite-difference wave simulator in hardware, we developed a complete fixed-point reference model in Python using Pygame. This software model replicates the wave propagation equation in a controllable environment and allows rapid iteration on boundary behavior, damping, and numerical stability. The Python simulation uses fixed-point values with parametrized layouts for fast testing. Furthermore, the Python simulation provides visual confirmation that our discretization and update stencil produce realistic wave behavior. These simulations served as the ground truth against which the FPGA implementation was validated.

A. Floating-Point to Fixed-Point Conversion

To translate the Python model into synthesizable hardware, all arithmetic was converted from floating point to fixed point. We implemented helper functions in Python to quantize the floating-point values into various fixed-point formats, enabling direct comparisons between floating-point updates and their fixed-point equivalents. This workflow allowed us to evaluate rounding error, stability, and visual fidelity as a function of bit width, and to experiment with fractional precision before committing to a hardware representation [3], [5].

Through extensive experimentation, we determined that a format with 1 sign bit, 7 integer bits, and 17 fractional bits (25 bits total) provides more than sufficient precision for wave heights, Laplacian values, and temporal updates. Smaller fractional widths resulted in visible artifacts or instability in the simulated wave field, while larger widths provided no meaningful visual improvement but increased hardware costs.

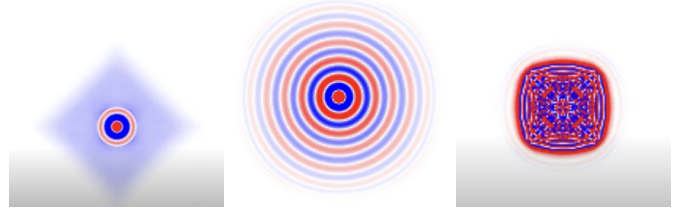


Fig. 4. Comparison of wave simulation results under different fixed-point configurations. Each image shows the simulated propagation using different fixed-point representations: 10 fractional bits and 12 integer bits (left), 17 fractional bits and 7 integer bits (center), 22 fractional bits and 2 integer bits (right). Too little fractional bits leads to weird underflow patterns occurring in the background (left) where as too few integer bits leads to overflow and a distorted wave pattern (right).

B. DSP-Aware Bit Width Selection

The choice of a 25-bit fixed-point format is driven not only by numerical precision but also by the structure of the DSP blocks on the RealDigital Urbana FPGA. Each DSP block provides a 25×18 multiplier, so packing the wave amplitude into a 25-bit value allows the entire finite-difference update to fit naturally into the DSP hardware without additional tiling or lookup table based partial multiplication. This mapping greatly simplifies the arithmetic pipeline and improves timing reliability.

For the finite-difference computation, the dominant multiplications are by constant coefficients. The update equation contains multiplications by:

- the wave speed constant $C = 0.15$,
- the damping factor $DAMPING = 0.995$.

Both constants fit cleanly into a 18-bit fixed-point representation (1 sign bit and 17 fractional bits), enabling the full 25×18 multiplication pathway in a singular DSP unit. All remaining multiplications in the finite-difference scheme are by powers of two (bit shifts), since setting $\Delta t = \Delta x = \Delta y = 1$ simplifies the wave equation without reducing numerical stability. These shifts are handled directly in logic and do not consume DSP resources.

C. Numerical Stability and Parameter Choices

The parameters used in the Python simulation, including $C = 0.15$ and $DAMPING = 0.995$, were verified for stability using the Courant condition for the 2D wave equation [6], [7]. The stability of the discretized update rule was tested in Python before being committed to hardware. Because the hardware implementation mirrors the Python model exactly, the resulting FPGA-based solver preserves the stability and dynamics of the software reference model.

This fixed-point representation, combined with DSP-aware arithmetic and verified stability, forms the foundation of the Fixed Point Unit, enabling real-time wave propagation on the FPGA.

VII. FIXED-POINT TO RGB MAPPING

The wave height at each grid cell is stored in a 25-bit signed fixed-point format Q1.7.17: one sign bit, seven integer bits, and seventeen fractional bits. This format provides a dynamic range of approximately $[-128, 128)$ with a resolution of $2^{-17} \approx 7.6 \times 10^{-6}$, which is more than sufficient for the amplitudes produced by the finite-difference solver. For visualization, these fixed-point amplitudes must be converted into 8-bit RGB values suitable for the HDMI output.

Our color-mapping module implements a four-step pipeline adapted from an online floating point amplitude visualization scheme [5], modified for our 25-bit Q1.7.17 representation.

A. Step 1: Absolute Value (Magnitude)

The first stage computes the magnitude of the wave amplitude. In software, this is simply

$$\text{abs_amp} = |\text{amp}|.$$

In hardware, we implement the same operation with a conditional two's-complement conversion:

- Let $\text{amp}[24]$ be the sign bit of the 25-bit value.
- If $\text{amp}[24] == 0$, then $\text{abs_amp} = \text{amp}$.
- If $\text{amp}[24] == 1$, then $\text{abs_amp} = \sim \text{amp} + 1$.

This produces a 25-bit nonnegative magnitude while preserving the original sign for later use in the color mapping.

B. Step 2: Multiply by 256

Next, we scale the magnitude into a range suitable for RGB conversion by multiplying the amplitude by a factor of 256. Multiplying by 256 is equivalent to a left shift of 8 bits. In Q1.7.17 form, we risk overflow, so we can add bits to the front of our absolute value to ensure we don't overflow:

$$\text{scaled} = \{4'h0, \text{abs_amp}\} \ll 8$$

In hardware this is implemented as a simple right shift; no DSP resources are required.

C. Step 3: Clipping to [0, 255]

The next steps require integers. We turn our previous fixed point value into an integer by simply indexing for the integer bits. Because large wave amplitudes or transient numerical spikes could exceed the 8-bit range, we clamp the scaled magnitude to the interval $[0, 255]$:

$$\text{intensity} = \begin{cases} 255, & \text{if scaled} > 255, \\ \text{scaled}[24:17], & \text{otherwise.} \end{cases}$$

In hardware this is implemented with simple comparisons and a multiplexer selecting either 255 or the integer bits of scaled . The result is a single 8-bit intensity value that encodes the magnitude of the wave at the current pixel.

D. Step 4: Signed Color Mapping

Finally, we map the sign of the original wave amplitude to color. We use a diverging blue–white–red scheme: positive amplitudes are drawn in red, negative amplitudes in blue, and values near zero appear white. Let I denote the 8-bit intensity from the previous step and $\text{sign} = \text{amp}[24]$.

- If $\text{sign} == 1$ (negative amplitude):

$$R = 255 - I, \quad G = 255 - I, \quad B = 255.$$

This produces bright blue for large negative values and fades toward white as the magnitude decreases.

- If $\text{sign} == 0$ (positive amplitude):

$$R = 255, \quad G = 255 - I, \quad B = 255 - I.$$

This produces bright red for large positive values and again fades toward white near zero.

All three channels R , G , and B are 8-bit values and are driven directly into the HDMI pipeline. Because the operations consist only of absolute value, shifts, clipping, and a small number of add/subtract operations, the entire fixed-point-to-RGB pipeline is inexpensive in logic and introduces negligible latency. The mapping preserves both the magnitude and sign of the wave amplitude in a visually intuitive way, resulting in a clear real-time depiction of the propagating wave field.

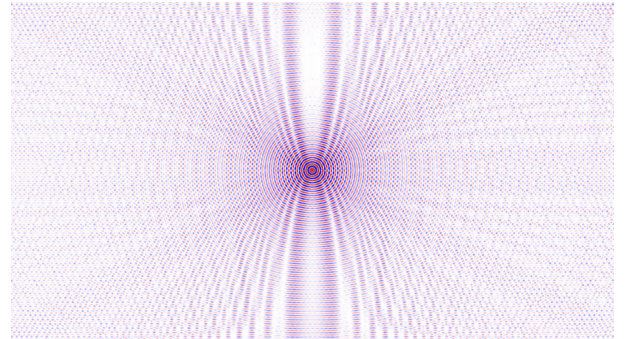


Fig. 5. High-resolution (1280x720) real-time wave propagation achieved as part of our stretch goal. The solver stably computes and displays the evolving wavefield at 60 fps, validating both the numerical fidelity and the memory bandwidth of our FPGA architecture.

VIII. DOUBLE-SLIT EXPERIMENT IMPLEMENTATION

To demonstrate the wave interference patterns predicted by classical wave physics, we implemented an interactive version of the double-slit experiment directly within the simulation domain. The experiment is activated in hardware through `switch[0]`; when the switch is turned on, the system overlays a static wall with two narrow openings onto the wave field. This wall geometry is registered in hardware as a mask evaluated during the finite-difference update stage.

When a wall cell is encountered during wave propagation, the corresponding amplitude is forced to zero, effectively creating a rigid boundary that reflects incoming waves. Only the two slit openings are left unmasked, allowing the injected sinusoidal source to propagate through them. This produces two coherent wavefronts that interfere downstream, forming the characteristic fringe pattern.

To ensure numerical stability and visual clarity, we tuned several parameters: the slit width and separation, the strength and frequency of the injected sinusoidal source, and the damping factor near boundaries. These adjustments allowed the interference fringes to remain sharp while still integrating cleanly with the live fingertip-driven interaction. Because all masking, boundary enforcement, and wave updates occur inside the hardware pipeline, the double-slit pattern appears in real time at 60 fps without any software post-processing.

This feature showcases how physical wave phenomena can be recreated and explored interactively on our FPGA-based solver, further validating the correctness and flexibility of our underlying simulation architecture.

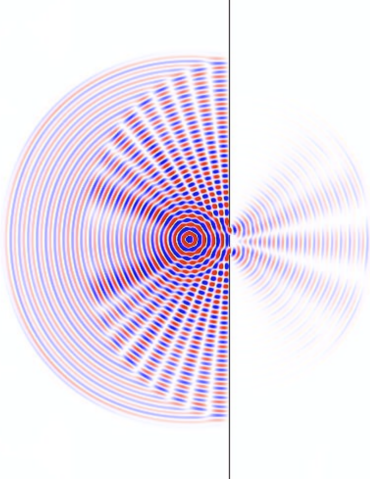


Fig. 6. Simulated double-slit interference pattern generated by our FPGA solver. A sinusoidal source propagates toward a wall with two narrow slits, producing the expected downstream fringe structure.

IX. OVERALL SYSTEM EVALUATION

A. Throughput, Latency, and Timing Requirements

Meeting timing was essential for sustaining real-time 60 fps wave simulation at our stretch-goal resolution of 1280×720 . The design spans multiple clock domains—including `clk_pixel` for the finite-difference

pipeline and HDMI output, and `clk_controller` for the DDR3 interface—so timing closure required careful attention to BRAM access paths, unpacking logic, and the pipelined arithmetic stages.

According to the post-implementation timing report, our design achieved a positive Worst Negative Slack (WNS) of 0.294 ns, indicating that all synchronous paths meet their required timing constraints. No additional pipelining was required beyond what was already introduced for the fixed-point computation unit and DDR3 traffic controller. Achieving this margin was nontrivial given the depth of the stencil arithmetic and the wide 128-bit burst interface used for memory streaming.

Latency in the system is dominated by DRAM round-trip delays: each frame requires fetching the packed previous time steps and writing back the updated values. However, this latency does not accumulate across frames, and our pipeline produces one wave update per pixel clock, allowing us to maintain a stable 60 fps output. The finite-difference arithmetic and fixed-point-to-RGB conversion are fully pipelined, so neither contributes to a throughput bottleneck.

Overall, the system comfortably meets timing and sustains the throughput required for full real-time operation. The positive slack and stable frame rate validate the architectural choices behind our memory streaming scheme, fixed-point configuration, and hardware pipelining strategy.

B. Memory

Our project required careful coordination between on-chip BRAM, the external DDR3 memory, and the DSP-aware fixed-point formats used throughout the finite-difference solver. Because our final system runs a 720p (1280×720) simulation grid at 60 fps, the wave field cannot fit entirely in BRAM and must be stored in external DDR3. Following the design in our preliminary report, we pack two time levels of the wave height (the current and previous frame) into a single DRAM word per grid cell using a format of `[buff0][buff1][padding]`, and store the full state as one contiguous block in memory. The `buffer_write` and `buffer_fetch` modules stream this packed state through the UberDDR3 controller using burst-oriented 128-bit AXI transactions, giving us a deterministic access pattern and sufficient bandwidth for real-time updates. On-chip BRAM is reserved for low-latency structures such as the Gaussian blur line buffers, the multi-line stencil cache used by the finite-difference unit, and the 60-entry sine lookup table, so the core pipeline can continuously consume data without stalling.

To reduce memory footprint and better match the Urbana board's 25×18 DSP blocks, we represent each wave amplitude in a 25-bit fixed-point format rather than a 32-bit floating-point value. Packing 25-bit samples into the DRAM interface width (with minimal padding) lowers the per-pixel bandwidth cost and allows the dominant multiplications by the wave speed constant $C = 0.15$ and damping factor 0.995 to fit cleanly into a single DSP unit. All remaining stencil multiplications are powers of two and are implemented as

shifts in logic, consuming no extra DSP or memory resources. In retrospect, additional optimizations such as more aggressive BRAM sharding for wider line-buffer parallelism, bank-aware DRAM tiling to reduce row-activation overheads, or lightweight compression of near-zero background regions could further reduce bandwidth. Nonetheless, the implemented memory architecture was sufficient to sustain a 1280×720 grid at 60 fps with interactive fingertip control, meeting our original commitment, main goal, and stretch goal for the project.

C. Checklist

We were able to successfully achieve all elements of our original commitment, implementing a fully interactive wave-propagation system that reaches 60 fps at a resolution of 180×360, matching the performance of our NumPy reference. We also met our primary goal by integrating fingertip-controlled wave injection through the camera pipeline and scaling the simulation to 60 fps at 360×640 resolution, exceeding the performance of the software baseline. Moreover, we completed our stretch goal by extending the system to run at 60 fps at 720×1280 resolution and demonstrating a hardware-accelerated double-slit experiment within the FPGA environment. Together, these results show that our design not only met but surpassed the targets outlined in our project plan.

D. Implementation Insights

One of the most challenging aspects of the project was getting the external DDR3 memory interface to operate reliably at the throughput required for real-time wave updates. Much of our early development time was spent debugging subtle synchronization issues between the `clk_pixel` and `clk_controller` domains, as well as understanding the constraints imposed by the UberDDR3 controller. Small mistakes in packing order, alignment, or `tlast` signaling often resulted in corrupted frames or stalled bursts, and it took repeated refinement of our `buffer_fetch` and `buffer_write` modules before the system behaved deterministically. In hindsight, bringing up DRAM earlier and isolating it in a standalone test environment would have accelerated development considerably.

A related implementation pitfall was reset behavior. Because both the wave state and the camera pipeline depend on correctly initialized DRAM and line buffers, partial or unsynchronized resets sometimes left the system in a corrupted state: on some resets, `active_draw` appeared shifted to the right or the first few frames contained garbage values instead of a clean quiescent wave field. Fixing this required enforcing a consistent global reset sequence, explicitly clearing the DRAM state buffers, and ensuring that all clock domains observed reset deassertion at well-defined boundaries. The experience reinforced how fragile complex multi-clock designs can be when reset is treated as an afterthought.

A second major lesson came from the design of the fixed-point arithmetic pipeline. Our Python simulation was indispensable here: it allowed us to sweep bit widths, visualize numerical artifacts, and experiment with stability parameters

long before committing to hardware. Several early hardware implementations produced either overflow or underflow patterns, and the Python reference model made it clear which configurations were numerically viable. The eventual choice of a 25-bit amplitude format and 18-bit constants was the result of this iterative loop between software simulation and hardware synthesis.

We also learned the importance of architectural flexibility. Because the frame-buffer layout in DRAM evolved as we better understood the bandwidth and packing constraints, many modules—especially the line buffer, unstacker, and fixed-point unit—had to be adapted multiple times. Certain design ideas that seemed reasonable early on (such as more complex buffer separation or storing values in different address regions) proved incompatible with the DRAM access patterns or AXI streaming semantics. Being willing to redesign interfaces and rework earlier modules was essential to making the final system stable.

Integrating the camera pipeline with the wave solver highlighted how even small mismatches in timing, coordinate systems, or mask definitions can cascade into large visual artifacts. We found that adding a simple pink fingertip glove, introducing `COOLDOWN_FRAMES`, and rejecting border coordinates were not just user-experience improvements but also important steps toward preventing numerical blowups in the finite-difference solver. At the same time, our choice of a 60-sample sine lookup table interacts subtly with the cooldown logic: because injections occur only every `COOLDOWN_FRAMES` cycles, the phase of the sine wave is effectively sampled sparsely, which can lead to visible variation in the apparent source intensity over time. In future revisions, we would like to make this path more robust and parametrizable—for example, by supporting different LUT sizes, decoupling the source phase progression from the injection cadence, or normalizing the injected energy—so that source behavior remains consistent across a wider range of parameter settings.

Overall, the most valuable insight from the project was that building a real-time hardware system requires rapid iteration, strong internal interfaces, and a willingness to revisit foundational design decisions when new constraints emerge. The final system is the result of many such revisions, each prompted by an unexpected interaction between memory, timing, and numerical behavior.

X. REPOSITORY

The full project source code is available at:
<https://github.mit.edu/6205F25/fa25-6205-team03>

XI. INDIVIDUAL CONTRIBUTIONS

A. Code

Isaiah implemented the Python wave simulation used as the reference model and designed the FPGA modules for buffer fetch, buffer write, the BRAM-backed sine-wave generator, and the double-slit experiment logic.

Collin implemented the FPGA modules for the fixed-point

computation unit, source-select logic, and fixed-point-to-RGB conversion, and integrated the fingertip source propagation with the camera-based detection pipeline.

B. Report

Collin drafted Sections I, III, VI, VII, IX, and X and created Figure 4. Isaiah authored Sections II, IV, V, and VIII and produced Figures 1, 2, and 3. We jointly designed the overall system block diagram.

XII. ACKNOWLEDGMENTS

We thank Joe Steinmeyer, our team mentor, Hasan Yeldiz, and the 6.205 teaching team for their continued assistance and advice on implementing this project.

REFERENCES

- [1] I. Berg, “The 2D wave equation,” Solving the 2D wave equation with the finite difference method, https://beltoforion.de/en/recreational_mathematics/2d-wave-equation.php
- [2] H. Langtangen, “Finite Difference Methods for Wave Motion,” Department of Informatics, University of Oslo, 2016.
- [3] A. P. Shanthi, “Floating Point Arithmetic Unit – Computer Architecture,” University of Maryland Department of Computer Science. Available: <https://www.cs.umd.edu/meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html>
- [4] Warrington, Don & Wynn, Robert. (2000). Comparison of Numerical Methods to Closed Form Solution for Wave Equation Analysis of Piling.
- [5] W. Green, “Fixed Point Numbers in Verilog,” Project F, 2020. Available: <https://projectf.io/posts/fixed-point-numbers-in-verilog/>
- [6] U.S. Army Corps of Engineers, “Variable Time Step Capabilities,” HEC-RAS 2D User’s Manual, 2025. Available: <https://www.hec.usace.army.mil/confluence/rasdocs/r2dum/6.6/running-a-model-with-2d-flow-areas/variable-time-step-capabilities>
- [7] U.S. Army Corps of Engineers, “Computational Time Step,” HEC-RAS Hydraulic Reference Manual, 2025. Available: <https://www.hec.usace.army.mil/confluence/rasdocs/ras1dtechref/6.6/performing-a-dam-break-study-with-hec-ras/computational-time-step>

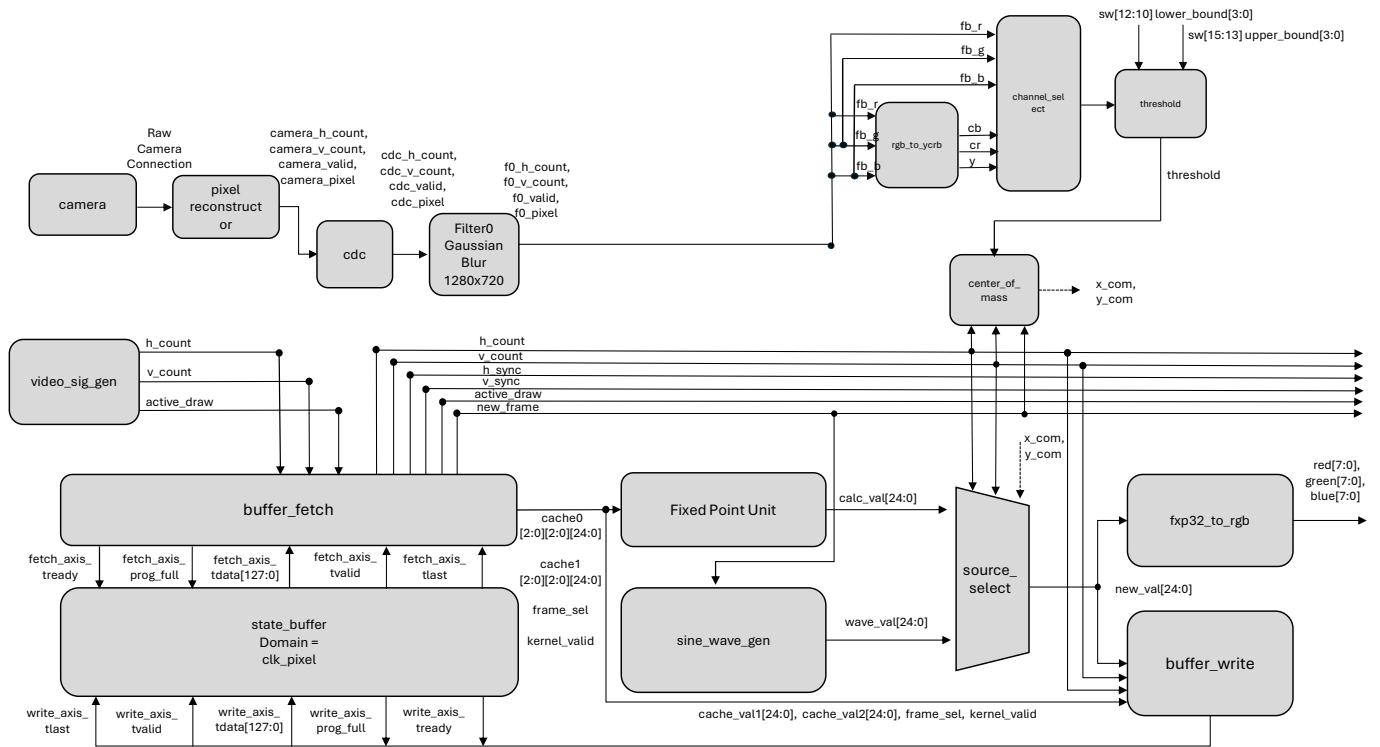


Fig. 7. Overall VeriWave Block Diagram.