Cornell Engineering
Electrical and Computer Engineering

Hardware Acceleration of Boids Flocking Algorithm

Author: Romano Tio Advisor: Van Hunter Adams

Boids

The boids algorithm, as defined by Reynolds in his 1968 paper, is a set of rules governing how a swarm of bird-oid objects (boids) may coordinate with their neighbors based upon three key metrics: separation, alignment, and cohesion.

- Separation: Boids seek to create distance between themselves and boids too close to themselves.
- Alignment: Boids seek to adjust their velocity to be similar to the average velocity of its (visible) neighbors.
- Cohesion: Boids seek to move towards the average position of the (visible) flock.

Importantly, the algorithm states that boids should have limited information, and should only consider boids within a certain range of themselves. We thus consider two key radii: the visual radius and a smaller protected radius. Boids in the protected radius are considered for separation, and boids in the visual radius are considered for alignment and cohesion but not separation. For this specific implementation, we also seek to keep the boids within a certain margin of the screen, turning them back from the edge of the screen as they fly away. This implementation of the algorithm also limits boid velocity to a certain range.

Challenges

There are a number of challenges with achieving good performance of the boids algorithm, both in general and as a hardware implementation. The boids algorithm is inherently O(N^2) time complexity. This cannot be limited easily, but the accelerator aims to make the algorithm as efficient as possible.

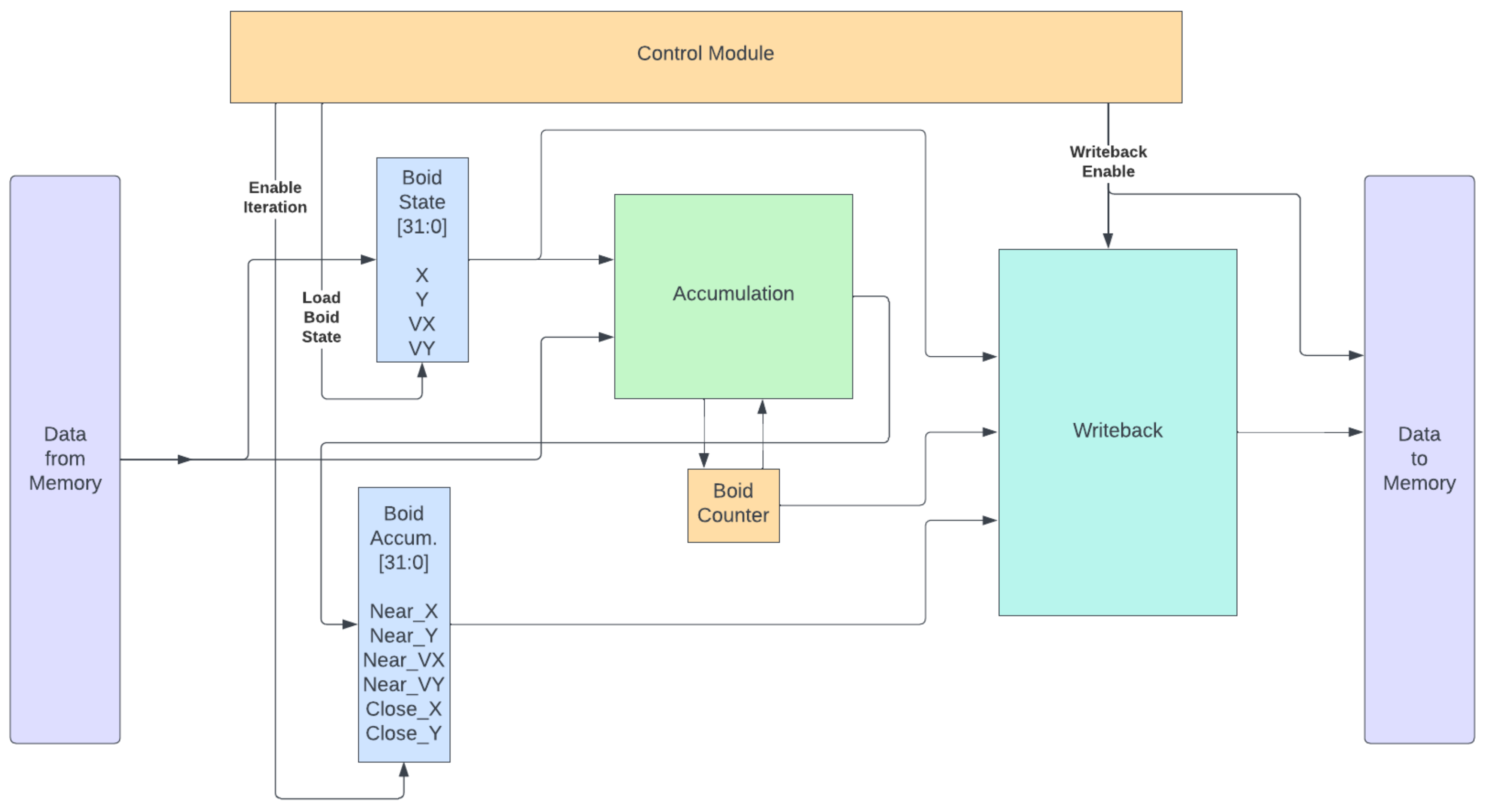
With the introduction of the hardware accelerator, drawing the boids onto a VGA screen becomes more complex. Firstly, we must use a VGA driver on the FPGA directly to drive the screen. This VGA driver provides the index o the next pixel it will draw and accepts color data it will write to that pixel. We compare this value with every value in our boid position registers and identify if a boid exists at that pixel and fill the pixel in if it does.

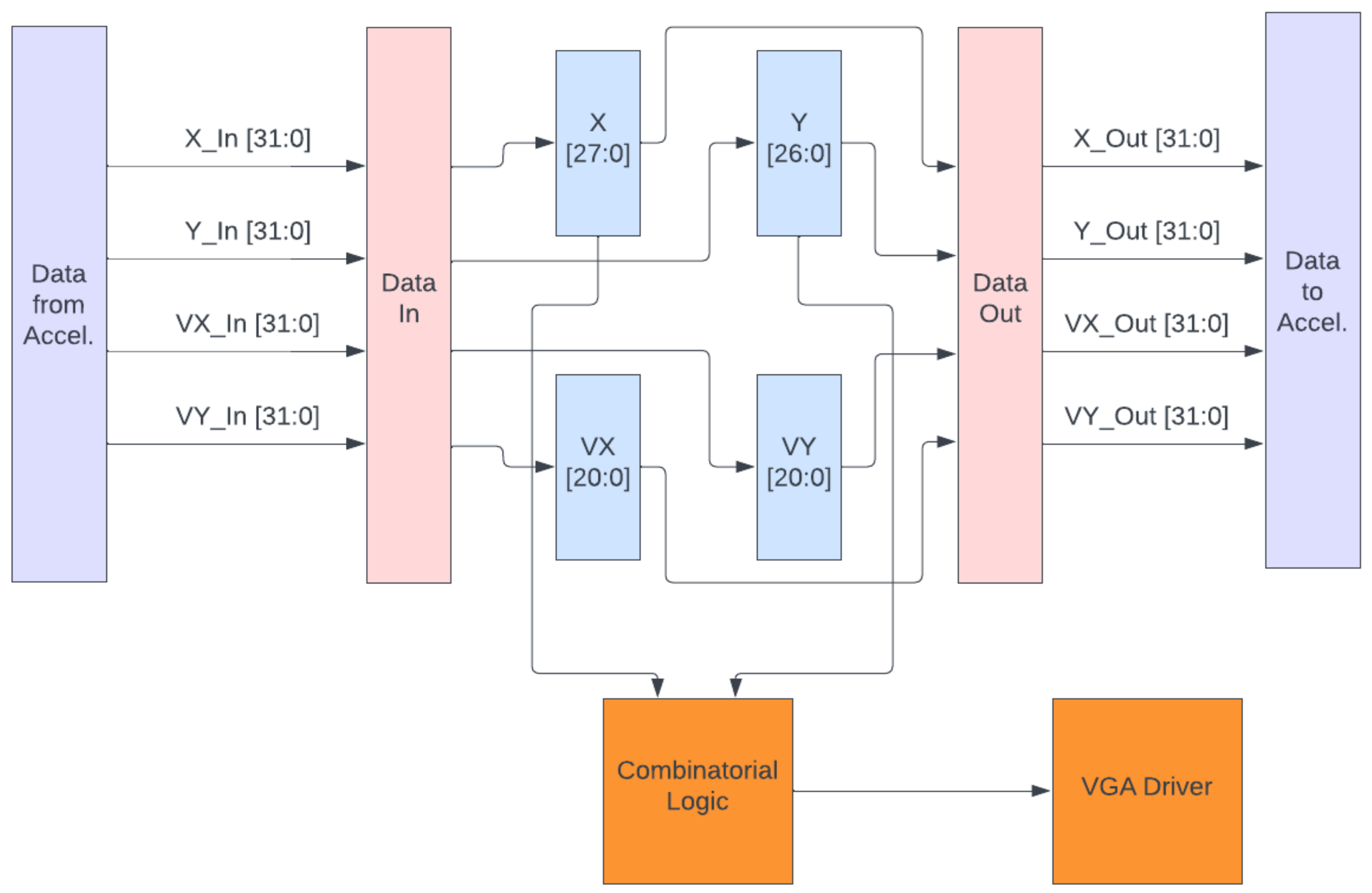
State Machine

The accelerator has five states, however one of these states is unnecessary due to current performance of the memory system. The states and their transition conditions are as follows:

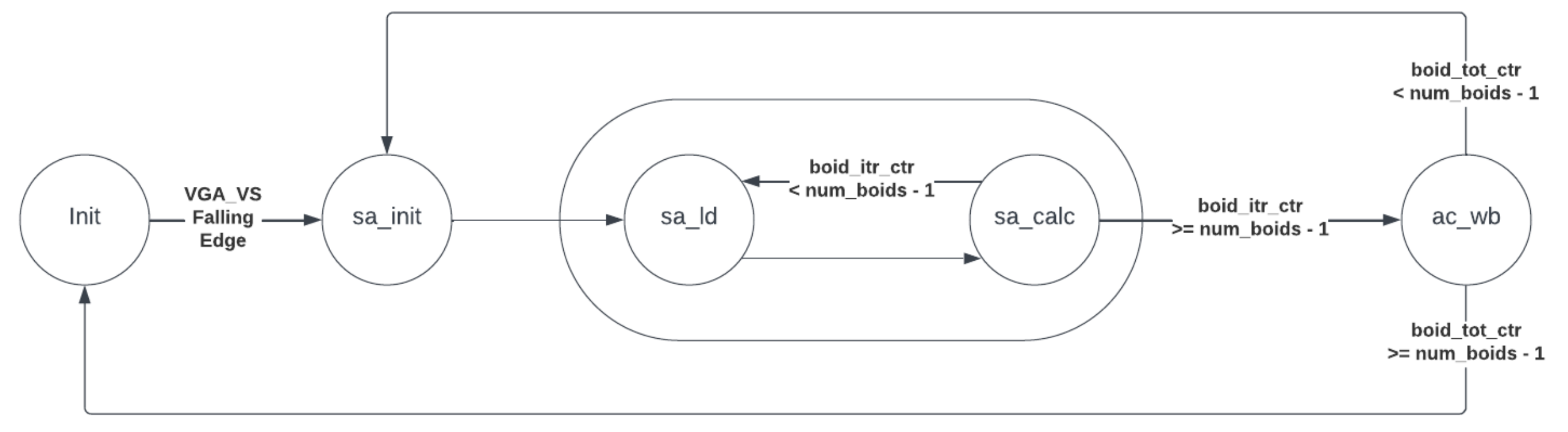
- Init: Initial state, waiting for VGA_VS signal from the VGA driver to show a falling edge. This signal indicates that the driver is drawing a new screen. Transition from init to sa_init
- sa_init: Separation-Accumulation Initial state, load data from the on-board memory to the Boid State registers. Transition from sa_init to sa_ld
- sa_ld: Separation-Accumulation Load state Retrieve iterative data from memory to be used in accumulation. Transition from sa_ld to sa_calc.
- sa_calc: Separation-Accumulation Calculation state, pass data through Accumulation block and latch data into Boid Accumulation registers. Potentially increment Boid Counter if a boid falls within the detection range but not the separation range. Transition to sa_ld if more accumulation must be performed, otherwise transition to ac_wb
- ac_wb: Alignment-Cohesion Writeback state, take data from Boid State and Boid Accumulation, then multiply by relevant normalization factors and update boid statein the Writeback block prior to writing back to memory. Transition to sa_init if not all boids have been updated, otherwise transition to init.

Accelerator Diagram





State Machine Diagram


Init
VGA_VS Falling Edge
sa_init
sa_ld
boid_itr_ctr < num_boids - 1
sa_calc
boid_itr_ctr >= num_boids - 1
ac_wb
boid_tot_ctr < num_boids - 1
boid_tot_ctr >= num_boids - 1

Acknowledgements

This accelerator implements Reynold's boids algorithm, as described in his paper:

Craig W. Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. SIGGRAPH Comput. Graph. 21, 4 (July 1987), 25–34. https://doi.org/10.1145/37402.37406

Thank you to my advisor, Van Hunter Adams

Approximations

- The algorithm is built using fixed-point arithmetic to allow for the use of integer math hardware structures
- Data is stored in memory using only as many bits as absolutely required to not lose information. The true bit-widths of the memories are labelled on the accelerator diagram. These values are sign-extended when read and truncated when written.
- The boid visual and protected ranges are calculated as squares to simplify the calculation (and required hardware) of these areas.
- Boids will only accumulate up to thirty-one of its neighbors. This allows us to perform divisions as multiplications by pre-calculated fractional values sourced from a lookup table. We index into this lookup table with the Boid Counter module.

Extrapolations

- The state machine spends 2 cycles (which could be reduced to 1 if AMAL of no more than 1 cycle can be guaranteed) per boid accumulation and 2 additional cycles to load from and store to memory. Accordingly, our cycle time to fully calculate N boids is N(2 + 2(N-1)) cycles
- The throughput of the accelerator can be improved by adding multiple boid calculators, but the accelerator must still check every boid. This equation looks like [2/K (N)] + 2N(N- K) for N boids and K accelerators.
- Each boid contains 97b (0.776kb) of state information. However, each datapath gets its own 'scratch pad' to record where the boids it has handled should be (adding an additional 38.4kB per accelerator, ~7.9% of the available memory). We should be able to represent 578 boids in theory before memory limitations are encountered.
- Area for multi-core operation should be a linear increase. We are instantiating multiple datapaths, but not instantiating multiple control units or memories.

Future Work

Currently, the calculation of each boid and updating of each boid's position has been sped up as much as is practical. However, the system is currently severely memory constrained and requires a few additional logical structures to allow for memory system expansion. In an attempt to fix this, an auxiliary memory was designed that would have been a 2D array the size of the VGA screen, which would have mapped position to boid presence and have been updated every time a boid was calculated. Unfortunately, this system did not fit directly on the FPGA, and would have required buffers to increment and decrement each element in this boid position map array. With this added, additional M10K memory blocks could be allocated up to the limit described in the Extrapolations section.

To expand beyond this, as we know our memory access pattern and can predict what memory locations we will need in advance, we can potentially utilize our M10K blocks as a cache and create a memory prefetcher to load data into the M10K memories and take advantage of their fast read speed compared to a larger and slower memory and continue expanding the number of boids we operate upon while mitigating memory access latency.

Lastly, for very large swarms, it could be possible to linearize the algorithm by creating a virtual memory map that associates each boid's position more strongly with its location in memory. If this was done, then the loop could be changed to check every memory position in the visual range to see if a boid is present as opposed to check if each boid is within the visual range or not. The constant coefficient would be much larger, so this solution would only be suitable for large swarms.