Hardware Acceleration of Boids Flocking Algorithm

A Design Project Report

Presented to the School of Electrical and Computer Engineering of

Cornell University in Partial Fulfillment of the Requirements for the

Degree of Master of Engineering, Electrical and Computer Engineering

Submitted by

Romano Alexio Tio

M.Eng Field Advisor: Hunter Adams

Degree Date: January 2024

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Hardware Acceleration of Boids Flocking Algorithm

Author: Romano Tio

Abstract: Despite the increase in computing power every year, the simulation of multi-agent systems often suffer from debilitatingly long runtimes. Traditional computers follow the Von Neumann architecture, with a general-purpose processing core which receives inputs and outputs, and is connected to a memory system. The Von Neumann architecture is an inherently serialized and memory restricted architecture, though it is easy to code for. GPUs serve as a more parallelized alternative, but are often very power-hungry and less area efficient compared to more customized solutions on FPGAs. This project tackles the creation of an accelerator for Reynolds' boids algorithm, leveraging principles of hardware-software codesign and providing documentation of the process. The hope is that by documenting the principles and advantages of targeting an FPGA, it can be made more approachable for the enterprising programmer to accelerate other algorithms using FPGAs.

Executive Summary

Through the course of this project, the primary accomplishment was the development and implementation of a hardware accelerator implementing the boids algorithm on an FPGA. Leading up to this we had performed a few preliminary assessments. I had explored the idea of a true random number generator and found it to be not viable, and I had spent time learning how to use the Qsys peripheral configuration tool in Quartus. After I pivoted to this project on boids, I first ported the boids algorithm from the Pi Pico architecture to the ARM core on the DE1-SoC (utilizing its VGA driver to devote the ARM fully to video processing), and established a baseline to target of around 450. I then began designing the accelerator itself, starting with floorplanning a control, datapath, and memory architecture. With these in hand, I then started by designing and verifying the control unit and control logic using simulation-level testing in ModelSim. I then moved on to a similar testing framework for the memory, followed by the datapath, designing and verifying individual components such as the separation-accumulation block and accelerator-writeback block (the primary calculation engines of the accelerator) separately prior to testing them in integration. After designing and passing everything through a basic sim verification, the designs were integrated and compiled onto the FPGA. At this stage more debugging was required to make the design fully functional, but eventually integration issues were resolved and we were able to display boids flying about on a screen (though incorrectly). At this point, we were able to finally proceed with algorithm debugging and parameter tuning, which coincided with stress-testing how large the design could be made. We achieved a maximum compilable size of around 340, which does fall short for two ARM cores but is competitive with one ARM core.

Introduction:

FPGAs, or field-programmable gate arrays, are a class of programmable logic boards that allow for rapid iteration of custom hardware at a lower cost than comparable custom silicon solutions. GPUs and TPUs serve as a higher throughput option which usually provide high performance benefits over traditional CPUs, but they trade programmability for higher power consumption and a lower performance ceiling. For power-constrained environments such as swarm robotics, hardware acceleration can allow for higher compute to be placed on board agents of a swarm, or closer to agents of a swarm on a smaller coordination device. Alternatively, as this project hopes to target, hardware acceleration can be used to improve simulations, in this case by increasing the number of agents reasonably simulatable at once.

To demonstrate hardware acceleration for swarm simulation, we target the boids algorithm as defined by Reynolds in his 1968 paper [1]. It 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 (within the protected range).
- 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 flock within its visual range.

Importantly, the algorithm states that boids should have limited information, and should only consider boids within a certain range of themselves (the visual range). As a precondition to calculating any one of the previous three metrics, the distance between the ‘checking’ boid and its neighbors is measured. If this distance is too small, we accumulate the difference between the checking boid and the neighbor. Otherwise, if the distance is less than our visual range we can consider alignment and cohesion. These properties are calculated in a similar way: As we iterate through the swarm, we accumulate positions and velocities of boids close enough to the checking boid, and increment a counter to record how many boids are neighbors to the checking boid. When all boids in the swarm have been considered, the accumulated values for position and velocities are normalized by the number we recorded in our counter (or are multiplied by zero in the case that no boids were within the visual range). Then, these normalized values are subtracted by the checking boid’s corresponding values for position and velocity. The values we now have prepared for separation, alignment, and cohesion are then multiplied by our avoidfactor, matchfactor, and centerfactor (weights for the weighted average we are taking) respectively, and are then added to the checking boid’s velocity.

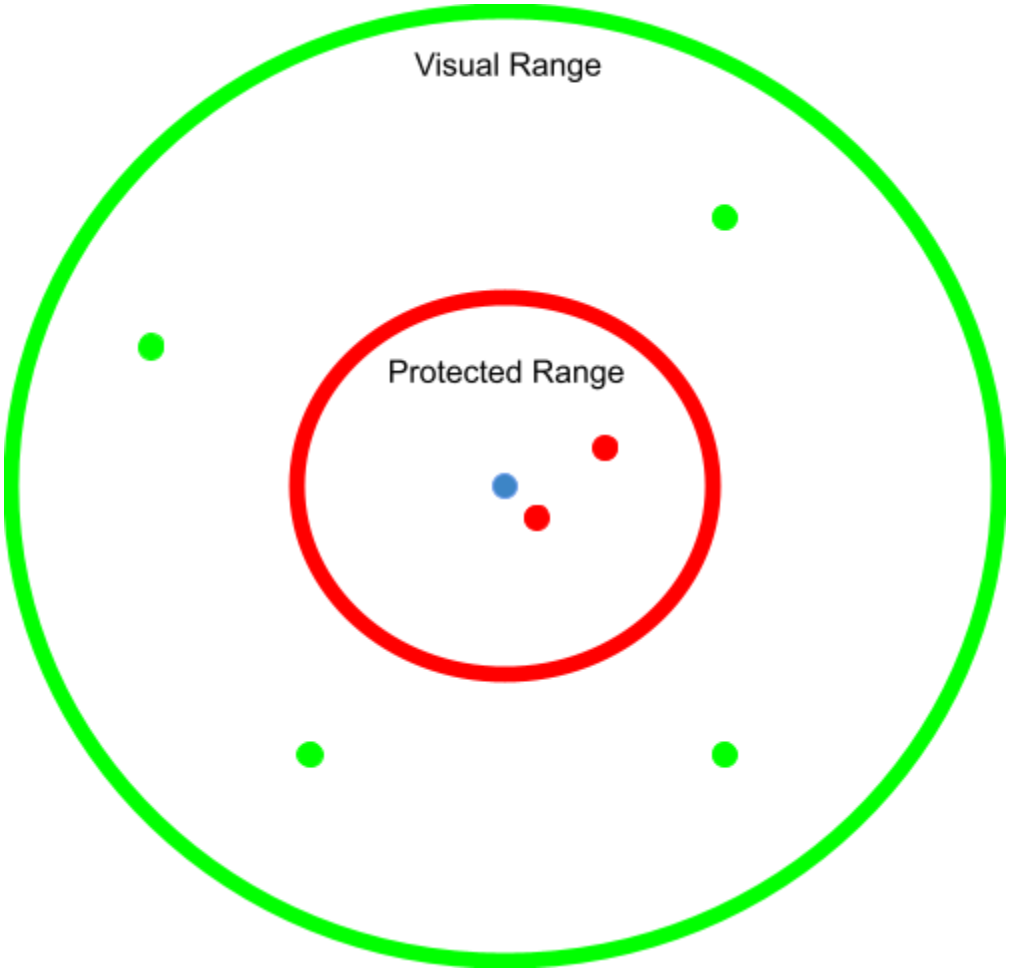

Figure 1: Illustration of boid protected and visual range. Boids/dots in red are considered for separation, boids/dots in green are considered for alignment and cohesion.

We additionally implement rules specific to our implementation to control when our boid nears the edge of the screen. We direct the boid back to the center of the screen by modifying its velocity to turn it back toward the screen, if the boid passes a margin near any edge of the screen. The amount we adjust the velocity by in that case is the turnfactor. We furthermore maintain that a boid's speed must be within a certain minimum and maximum value. We compute the speed of a boid with an approximation of the square root, by first computing the absolute value (multiplying by -1 if a number is less than 1) of the x and y velocities. Then, the smaller of the two is arithmetically shifted right by 2 bits and summed with the larger, and this number is the approximate speed. If this number is larger than 6 in our fixed-point representation (15.16 bits with 1 sign bit), we scale the velocity down, and if this number is smaller than 3 we scale the velocity up. Lastly, this final velocity is added to position for both x and y and these updated position and velocity values are written back to memory. This process is iteratively repeated for every boid in the swarm.

We target the boids algorithm for this project for a few reasons. First, it is a very well-established algorithm which many in the field of swarm robotics understand. It is useful beyond the

simulation of swarms as well, serving as the foundation for the particle swarm optimization algorithm. Acceleration of PSO could have applicability for machine learning tasks. Lastly, Reynolds' boids algorithm is relatively simple to implement, though not without its difficulties as we will discuss in more detail.

Although the body of work based upon the boids algorithm by Reynolds [1] is not the strongest in the sector of hardware acceleration, it is nonetheless a classic in swarm control algorithm. Much in contrast to the original boids paper, however, is Kennedy and Eberhart's paper on the particle swarm optimization algorithm [2]. PSO is essentially a generalization of the boids algorithm to function with any fitness function aside from the one which boids defines, though PSO only operates with the cohesion rule of the boids algorithm (moving towards a global best solution in the solution space). There is an accordingly large body of work focusing on algorithmic improvements to PSO which primarily seek to address PSO's propensity for finding local maxima that are not globally optimal, including reinitializing particles in the swarm when they are too close to each other [4], redefining particle grouping within a certain radius (making a composite agent of multiple sub-agents) [5], and reintroducing the alignment rule to PSO [7]. As for specific hardware implementations of PSO or boids, they do exist in both FPGA [8] and ASIC [9] implementations, but these implementations primarily target floating-point data or otherwise are focused on absolute performance metrics. Swarm robotics implementations of PSO also exist [3], however these solutions optimize to a static target instead of a dynamic one as this accelerator hopes to implement. As a result, this solution hopes to bring faster fixed-point mathematics and

Implementation

This project is primarily coded in synthesizable SystemVerilog, though the base project it is derived from is primarily coded in Verilog. We utilize the VGA_TO_M10K project from Hunter Adams' website as the foundation of this algorithm, which instantiates Intel IP to speak with the board peripherals and a VGA driver to deliver outputs to the VGA screen (CITATION). This implementation of the boids algorithm is organized into three distinct components: control, datapath, and memory interface. Each of these components are organized into their own modules. Broadly, the control unit directs the execution of the loop, the datapath performs arithmetic operations, and the memory layer both holds data and is the interface between the accelerator and the VGA driver.

It is worth briefly mentioning the metrics I will be using to discuss the performance of the FPGA throughout this paper. An FPGA instantiates our design on its Adaptive Logic Modules, which can be utilized either as Adaptive Lookup Tables (ALUTs) or Logic Registers (LRs). These metrics are what we will use to evaluate the design area. Performance is evaluated by the capacity of the system to support boids in a swarm. Many things go into this capability, but the ones we primarily encounter are memory capacity and the function of cycles per boid for a swarm of a given size. For reference, a normal Von Neumann-style ARM processor is able to support anywhere from 200 to 2000 boids in a swarm. The program we are comparing against

(graphics_16bit_boids.c) takes 111 machine instructions (with 8 branches) to perform each accumulation step and 273 cycles (with 16 branches) to perform the final writeback, when processed through Godbolt armv8-a clang 17.0.1.

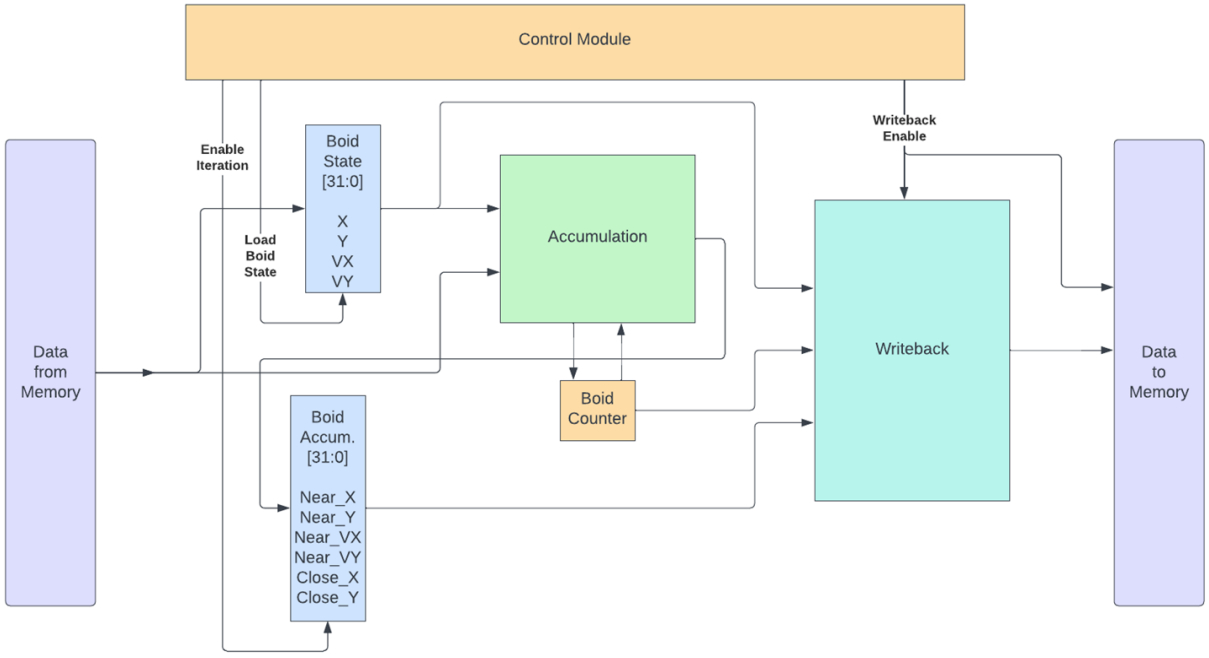

Figure 2: Accelerator datapath

First, it is worth briefly going over our method of data representation that our accelerator implements. This accelerator implements its calculations using fixed-point mathematics. We represent our numbers according to the diagram in Figure 3, reducing our dynamic range to 2^16 but increasing resolution while simplifying the hardware design. For our purposes, we are not particularly interested in numbers much larger than 1024 given the VGA screen we use has dimensions of 640 by 480 pixels, but we are interested in higher precision for small velocities in the range of 2-8 pixels, as otherwise our boids' movement would be very crude and jumpy. Fixed-point numbers otherwise act like signed integers (and would in fact be represented as signed int in C). For our purposes, we constrain the range further in memory while holding the decimal point constant, and sign extend it back to the full range for calculations.

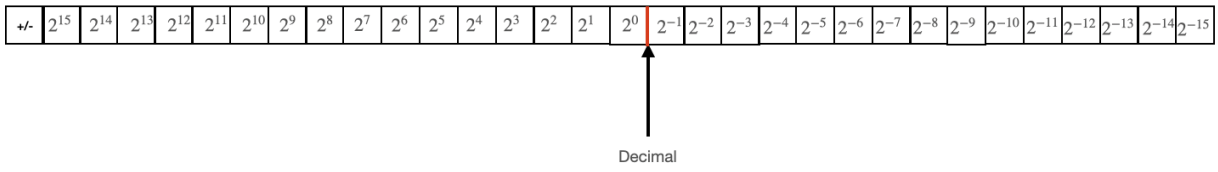

Figure 3: Fixed-point number representation

Datapath

The datapath contains four primary structures: the Accumulation unit, the Writeback unit, the Boid State registers, and the Boid Accumulation registers.The operations of these components are dictated by inputs from the Control Module and the Boid Counter, and data is received from the memory into the Boid State or Accumulation block. Completed data is sent back to the memory after post-processing in the Writeback block. The control unit will enable the boid state registers to read, and the control unit will enable the boid accumulation registers to accept feedback from the Accumulation block when appropriate. Due to the current register-based nature of the memory, we can safely refrain from placing a register between the memory and the Accumulation unit. However, if we ever anticipated memory access delays we would need to add boid iteration registers to retain this data for processing in the Accumulation module. We mux the inputs of the Writeback block with 0 to reduce activity in that section of the hardware, and when we direct the accelerator to write back to memory we mux in the values in Boid State and Boid Accumulation. Value from the Writeback stage is written back to memory directly (without any intermediate registers) with the higher bits truncated as we are able to without losing information. We keep 28 bits for X and 27 bits for Y, enough to capture the full range of values which could be on the screen (up to 1024 for X and 512 for Y) and we keep 21 bits for velocity in each direction (from +7 to -8).

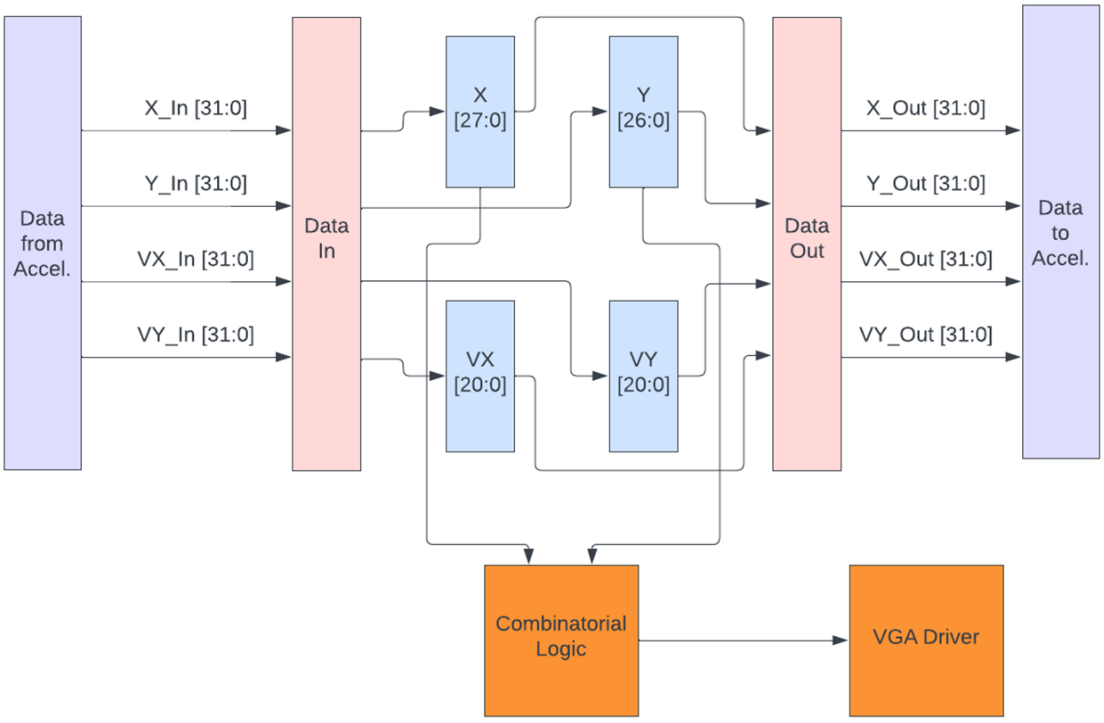

Figure 4: Accelerator memory interface

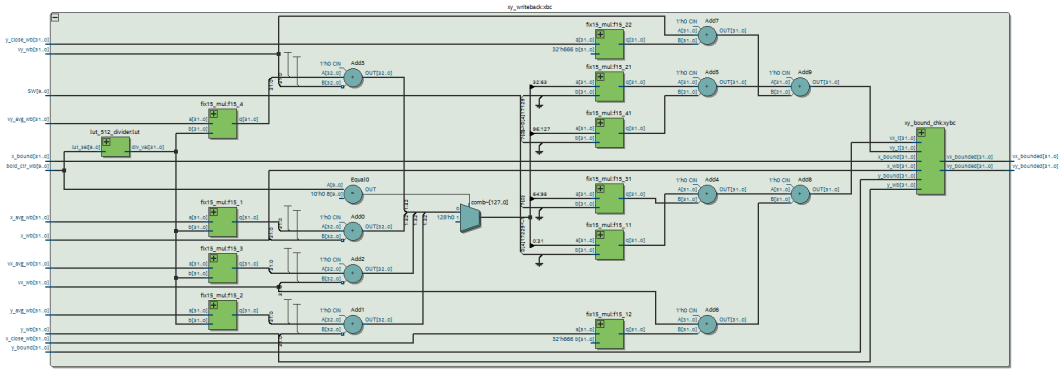

Figure 5: Writeback module (generated from Quartus Netlist Viewer)

The Writeback block is the most expensive component we currently instantiate, representing around 50% of the design area and 70% of the DSP blocks the design utilizes. This block compresses all calculations that are required for boid position update, executing them in parallel (within the same cycle) where applicable. Accordingly, we perform all the normalizations of our summed data in parallel then collapse that data down into an updated velocity. We then adjust this velocity in accordance with our position (turning back towards the boundaries if we cross them, and normalizing speed if it is too high) before adding velocity to position and writing back to memory.

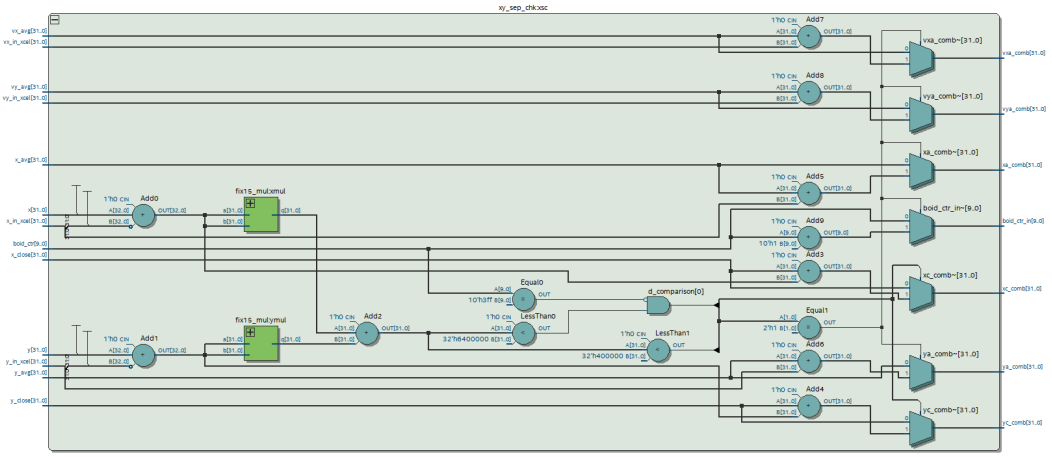

Figure 6: Accumulation module (generated from Quartus Netlist Viewer)

The Accumulation block, in contrast to the Writeback block, is much smaller coming in at 329 ALUTs. This block squares the difference in X and Y position between the boid in the Boid State registers and the boid we have pulled from memory for this iteration of the loop. These values are then summed before being compared to two thresholds. If the sum of the squared distances indicates the boids are ‘too close’ (defined as ‘the sum of the squared differences is less than 8’), then we accumulate this difference onto our separation registers (the x_close and y_close register) and do not update any other registers. If the sum of the squared distances is greater than our ‘too close’ parameter but less than our visual range parameter (defined as 1600) and if the boid counter is not currently at its maximum value of all 1s, we add the memory boid’s

position and velocity onto our alignment and cohesion registers and increment the boid counter. As we use a lookup table divider in Writeback, we want to limit the number of boids we can accumulate in some way, however because we need to check for collisions we cannot simply break the iteration once the boid counter is saturated.

Control Unit

The control unit is structured as a canonical finite state machine, with a current-state combinatorial output, a next-state combinatorial update, and a synchronous current state register alongside some counters (boid_tot_ctr, boid_itr_ctr) to control loop iteration. 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 state in the Writeback block prior to writing back to memory. Transition to sa_init if not all boids have been updated, otherwise transition to init.

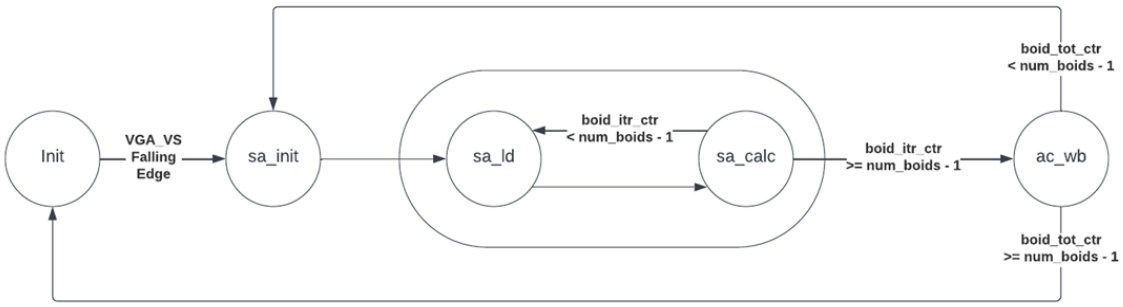

Figure 7: FSM flow chart

The state machine instantiates only a handful of components. As we control exiting the init state with the falling edge of the VGA_VS signal, we instantiate a simple falling edge detector. This circuit records the value of a signal and the previous value of the signal, and outputs 1 if the current value is negative but the previous value is positive. Besides VGA_VS, clock, and reset, we do not need to accept other external signals, however if we wanted to take into account

potential memory delays we would need to accept a control signal to notify the state machine when a memory request is successfully fulfilled. Otherwise, the state machine increments boid_itr_ctr and boid_tot_ctr. These two counters are used as array indices to extract values from the memory system. boid_tot_ctr is used to load data into Boid State and write data back to memory, and boid_itr_ctr is used to load data for accumulation.

When the control unit was initially designed for two boid operations, it did not require the boid_itr_ctr and could simply read the inverse of the one-bit boid_tot_ctr for reading the state of the other boid. However, to show correct behavior for more than 2 boids it is necessary to keep a second counter. If boid_itr_ctr would increment to be equal to boid_tot_ctr, we skip ahead one to save two iterations. Although not skipping this value would not create incorrect behavior, as this would lead to a 0 (a result of subtracting the values in Boid State from itself) being accumulated into close_x and close_y, we can easily save two cycles per loop by being conscious of this. boid_itr_ctr currently resets to 0 at the beginning of each loop, however it could be configured to reset to 1+boid_tot_ctr and to terminate at boid_tot_ctr. Doing so does not provide any benefit right now, and would require explicit handling of overflow to roll back over to 0, but would be a small optimization if memory writing wasn't single-cycle. This change would guarantee that the last boid to be written to memory (at boid_tot_ctr - 1) will be the last boid accessed in the calculation of the subsequent boid (at boid_tot_ctr). Currently, every boid looks at every other boid in the swarm, and we do not allow boids to mutually update each other. Mutual updating of boids (updating boid B's velocity/position as it is accumulated onto boid A) would require the allocation of additional memory to record these pre-accumulated values, alongside additional logic to load data from memory into Boid Accumulation registers.

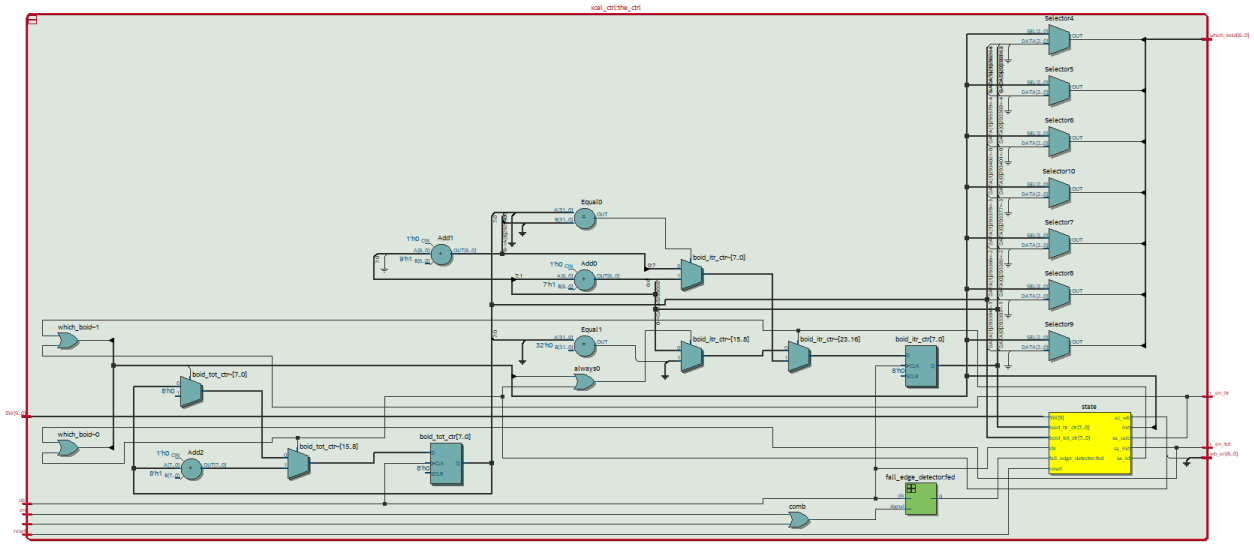

Figure 8: Control unit block diagram for 100 boids (generated from Quartus Netlist Viewer)

Memory System

From our datapath unit, we send data through a translation layer which truncates bits from the position and velocity, and latch this data into the appropriate registers. When reading data from the registers, though, we must sign extend it appropriately. We have a configurable sign extension module we instantiate to sign extend x, y, vx, and vy as they are read from the memory into the datapath. Furthermore, with the current memory system, we directly interface from the x and y registers to the VGA driver. To do this, we logically or every register in the x and y array with a next_x and next_y output from the VGA driver (the driver outputs the next position on the screen it will write, and accepts the color it will write to that pixel as an input). This is, admittedly, a crude and unsustainable method for interfacing with the VGA driver if we hope to expand the system to a larger memory.

Algorithmic Assumptions

In the name of allowing for a reduction of hardware complexity, we performed some simplifications and assumptions. The first and most impactful is the decision to target fixed-point numbers instead of floating-point numbers. Fixed-point numbers shift where the decimal point is considered to be, sacrificing an integer's dynamic range to enable calculation of fractional values. Despite our use of a hardware accelerator, floating-point math would require more complicated and power-hungry hardware throughout. A fixed-point solution allows for the usage of integer hardware components with some degree of modifications (namely that multiplies capture a different set of bits), simplifying the constructs that an FPGA would need to instantiate.

Secondly, we perform a few simplifications that reduce accuracy somewhat but eliminate costly requirements of calculation. Our speed normalization required a division before, but now we simply add/subtract our pre-normalized speed by itself arithmetic right-shifted by 2, depending on whether we need to grow or shrink it. If we did not simplify this, the required division would represent a massive slow-down and a requirement for complicated division hardware.

Lastly, one of the more impactful assumptions stems from the requirement to collect averages (and thus divide) which the algorithm calls for. We can sidestep the need to implement a complicated and time-wasting hardware divider by utilizing the value from the boid counter as an index to a lookup table of pre-calculated division coefficients as opposed to a denominator of a division directly. We can then perform a fixed-point multiplication to achieve division.

Design Area

Swarm Size - Component	Comb. ALUTs	Logic Registers	DSP Blocks
200 - Control	61	19400	0
200 - Datapath	2447	25	42
200 - Memory	9781	299	0
200 - Total	**12289**	**19724**	**42**
100 - Control	54	23	0
100 - Datapath	2039	299	42
100 - Memory	7039	9700	0
100 - Total	**7039**	**10022**	**42**
50 - Control	57	21	0
50 - Datapath	2040	299	42
50 - Memory	2465	4850	0
50 - Total	**4562**	**5170**	**42**
10 - Control	31	17	0
10 - Datapath	2044	299	42
10 - Memory	577	970	0
10 - Total	**2652**	**1286**	**42**
2 - Control	18	13	0
2 - Datapath	2045	299	42
2 - Memory	177	194	0
2 - Total	**2240**	**506**	**42**

Table 1: Feature size for given swarm sizes. All of these design sizes were built.

The design area that the accelerator control and datapath occupies for a variety of swarm sizes from 2 to 100 is visible in Table 1. The control unit will grow in size slightly as the swarm increases in size due to a wider boid_itr_ctr and boid_tot_ctr, with occasional fluctuations derived from the precise swarm size. However, the datapath does not change much in size with swarm size, with slight fluctuations in combinatorial ALU count and a static requirement for 299 logic registers and 42 DSP blocks. The largest component in the datapath by far is the xy_writeback module, which is where 30 of the DSP blocks and 949 of the ALUTs are allocated to. The FPGA supports a maximum of 85k logic elements, making the accelerator's use of general logic elements very minimal, but its use of DSP blocks significant (the FPGA we have designed for contains only 87 DSP blocks). Thus, without utilizing multiple FPGAs, our maximum degree of parallelism is 2 parallel accelerators before we must make considerations about our use of multipliers.

Performance and Extrapolations

With the current configuration of the state machine, Alignment-Cohesion Writeback and Separation-Accumulation Initialization occur once per boid in the swarm. To calculate the updated position and velocity for each boid, we must read the state of every other boid in the swarm, which currently takes 2 additional cycles per boid. We could, in theory, eliminate one of these states, however if we were to be made to contend with memory latency we would need two states here to separate waiting for data from operating upon the data. We can accurately predict our accelerator to have an execution time $N(2 + 2(N - 1))$ cycles per each full calculation loop for a swarm of N boids.

Though in its current state, the processing of boids is not parallelized, we could consider multiple boids in parallel by instantiating multiple copies of the datapath (and Boid Counter) and modifying the control unit slightly to increment boid_tot_ctr by the number of datapath and to increment boid_itr_ctr up to the number of boids minus the number of datapaths. We would need to structure the memory to allow for parallel reads and writes (likely by splitting the memory into sub-arrays), and we would need multiple copies of the interface. With these alterations, we could achieve an execution time of $\lceil(N/K)\rceil(2 + 2(N - 1))$ for N boids with K parallel datapaths. Notably, this does not reduce our algorithm time complexity, but it does reduce the coefficients significantly for larger N.

As a point of comparison, as stated earlier we found accumulation to take 111 cycles and writeback to take 273 cycles to update each boid (excluding memory access latency and branch misprediction time). On a Raspberry Pi Pico (the device most students have used to implement boids at Cornell), the programmer has access to two cores operating at 133MHz each. Cycle-time normalized from the 133MHz max clock of the Pi Pico to the 50MHz our FPGA runs at, this would result in a performance equation of $N(47.6 + 102.6(N - 1))$ if we only consider 'accumulation' and 'writeback' components. Of course, the algorithm I used to compare this was

a minimally optimal one, and could likely be surpassed, but the architecture of this accelerator was based on the same code. Furthermore, the Pi Pico must also spend CPU time updating the VGA screen whereas we offload that to a completely parallel device. The advantage which the Pi Pico has, however, is a much larger memory as a baseline, and the CPU not requiring that we define a memory mapping for the program (leaving such a thing for the compiler). As a result, while our design runs into a wall with regard to available memory, Pi Pico solutions can often and easily surpass this accelerator. We could alleviate this with memory improvements which we discuss in Future Work.

Future Work

Currently, the calculation of each boid and updating of each boid's position has been sped up as much as is practical without a major paradigm shift from the basic algorithm we implemented. 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.

Our theoretical limit for the number of boids we can calculate, assuming a perfect memory and a 50MHz clock to the FPGA, we have a budget of ~1.6 million cycles per frame, at 30 frames per second. Given this cycle budget, we could theoretically support 912 boids at 30 frames per second. This is a theoretical improvement over the baseline we have established (which runs at 800MHz for comparison), and it may also be worth considering restructuring some of these components as an accelerator attached to the CPU. From the perspective of the faster-clocked CPU, it would take 32 CPU cycles to receive data from the separation pipe or write-back pipe of the accelerator.

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.

To further enable parallelization, and reduce the amount of DSP blocks used by the writeback stage, the writeback stage could be pipelined to use a smaller number of multipliers over a larger number of cycles to perform all of its necessary calculations and reductions. Natural numbers to choose would be four (the largest number of multiplies that launch at the same point in the logic), two, and one. If we perform the slight access restructure

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 checking 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.

Conclusion

The use of FPGA-based solutions for swarm robotics tasks may have merit in certain situations. CPUs are versatile and easy to program for, but often a swarm robot only needs to execute a very narrow set of functions, and in a case where this is true it may be worth looking towards hardware acceleration. Custom silicon solutions are often prohibitively expensive to develop for research use (requiring a very expensive and logistically difficult tape-out), but FPGAs can be reprogrammed swifty and still make many of the benefits of ASICs available. Though this implementation falls short of increasing absolute performance over a microcontroller, it is likely that with further optimization it could surpass the performance of any device operating at a similar power figure. The performance it offers is competitive with one ARM Cortex-M0+, but fails to surpass two ARM cores acting in concert.

References

[1] Craig W. Reynolds, ”Flocks, herds and schools: A distributed behavioral model,” Proceedings of SIGGRAPH'87 - 14th annual conference on Computer graphics and interactive techniques, New York, NY, USA, 1987, pp. 25-34, doi: 10.1145/37402.37406
[2] J. Kennedy and R. Eberhart, ”Particle swarm optimization,” Proceedings of ICNN'95 International Conference on Neural Networks, Perth, WA Australia, 1995, pp. 1942-1948 vol.4, doi: 10.1109/ICNN.1995.488968.
[3] C. Greenhagen, T. Krentz, J. Wigal and S. Khorbotly, ”A real-life robotic application of the particle swarm optimization algorithm,” 2016 Swarm/Human Blended Intelligence Workshop (SHBI), Cleveland, OH, USA, 2016, pp. 1-5, doi: 10.1109/SHBI.2016.7780281.
[4] J. J. Liang and B. Y. Qu, ”Large-scale portfolio optimization using multiobjective dynamic mutli-swarm particle swarm optimizer,” 2013 IEEE Symposium on Swarm Intelligence (SIS), Singapore, 2013, pp. 1-6, doi: 10.1109/SIS.2013.6615152.
[5] Wang Guang-Hui, Chen Jie and Pan Feng, ”Cooperative Multi-Swarms Particle Swarm Optimizer for dynamic environment optimization,” 2008 27th Chinese Control Conference, Kunming, 2008, pp. 43-48, doi: 10.1109/CHICC.2008.4605456.
[6] M. Munlin and M. Anantathanavit, ”New social-based radius particle swarm optimization,” 2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA), Siem Reap, Cambodia, 2017, pp. 838-843, doi: 10.1109/ICIEA.2017.8282956.

[7] Z. Cui, ”Alignment particle swarm optimization,” 2009 8th IEEE International Conference on Cognitive Informatics, Hong Kong, China, 2009, pp. 497-501, doi: 10.1109/COGINF.2009.5250688.
[8] T. Tala´ska, R. Długosz and W. Pedrycz, ”Hardware implementation of the particle swarm optimization algorithm,” 2017 MIXDES- 24th International Conference ”Mixed Design of Integrated Circuits and Systems, Bydgoszcz, Poland, 2017, pp. 521-526, doi: 10.23919/MIXDES.2017.8005267.
[9] A. Rathod and R. A. Thakker, ”FPGA realization of Particle Swarm Optimization algorithm using floating point arithmetic,” 2014 International Conference on High Performance Computing and Applications (ICHPCA), Bhubaneswar, India, 2014, pp. 1-6, doi: 10.1109/ICHPCA.2014.7045338