

# Designing an Application-Specific Processor to Accelerate the Computation of Discretized Partial Differential Equations Used in Numerical Weather Prediction

Brian Dang

*dept. of Electrical and Computer Engineering*  
*University of California, Davis*  
Davis, CA, USA  
btdang@ucdavis.edu

Kyle Heien

*dept. of Electrical and Computer Engineering*  
*University of California, Davis*  
Davis, CA, USA  
kmheien@ucdavis.edu

Brian Kuhn

*dept. of Electrical and Computer Engineering*  
*University of California, Davis*  
Davis, CA, USA  
bdkuhn@ucdavis.edu

Andrew Lu

*dept. of Electrical and Computer Engineering*  
*University of California, Davis*  
Davis, CA, USA  
awclu@ucdavis.edu

## Abstract

This project aims to accelerate the computation of the Shallow Water Equations; a significant bottleneck in the calculations involved in numerical weather prediction. These equations are computationally intensive partial differential equations that can be approximated using Finite Volume Discretization. This breaks up the calculation to a series of arithmetic equations on a grid. Currently, each point is calculated in series, however it is possible to calculate a number of them in parallel by designing an Application-Specific Processor (ASP) to perform the calculations; which would significantly improve computation time. The ASP is built on an Altera Cyclone FPGA, utilizing a number of DSP blocks and a Cortex A9 ARM processor.

## Index Terms

PDE, parallel computing, FPGA, ASP, Verilog, finite difference method, finite volume method, ALM, DSP, flux, gaussian, geostrophic, gradient

## I. INTRODUCTION

Weather forecasting is a major aspect of meteorology and has a profound impact on the daily lives of billions of people. There is also much that can be learned by analyzing prior atmospheric conditions to plan major changes in climate that may occur in the coming years. The computations that go into these forecasting are very complex and require a high amount of computation power. The National Weather Service currently uses two supercomputers, each with a processing power around 5.78 PFLOPS to run these calculations and generate climate models [1]. The Energy Exascale Earth System Model (E3SM) is a sophisticated simulation that can simulate a 10 year period of the Earth at low resolution<sup>1</sup> in about 24 hours [2]. About 50% of the computation time in the model is occupied by dynamics<sup>2</sup> and tracer transport<sup>3</sup>. Of this 50%; 37% is taken by dynamics, which involves solving the Navier-Stokes Equations, and 63% is occupied by tracer transport calculations. Currently, tracer transport calculations can be sped up by using a Graphics Processing Unit (GPU), and as a result, we have decided to focus our project on accelerating the Navier-Stokes calculations [3].

<sup>1</sup>“Low resolution” is defined as a grid size of 110 km by 110 km.

<sup>2</sup>Flow of water air, and other fluids.

<sup>3</sup>Particulate matter such as molecules and pollutants.

$$\begin{aligned}
F_h &= -\frac{U_{i+1,j} - U_{i-1,j}}{2\Delta x} - \frac{V_{i,j+1} - V_{i,j-1}}{2\Delta x}, \\
F_u &= -\frac{1}{2\Delta x} [U_{i+1,j}^2/h_{i+1,j} + Gh_{i+1,j}^2/2 - U_{i-1,j}^2/h_{i-1,j} - Gh_{i-1,j}^2/2] \\
&\quad - \frac{1}{2\Delta x} [U_{i,j+1}V_{i,j+1}/h_{i,j+1} - U_{i,j-1}V_{i,j-1}/h_{i,j-1}], \\
F_v &= -\frac{1}{2\Delta x} [U_{i+1,j}V_{i+1,j}/h_{i+1,j} - U_{i-1,j}V_{i-1,j}/h_{i-1,j}] \\
&\quad - \frac{1}{2\Delta x} [V_{i,j+1}^2/h_{i,j+1} + Gh_{i,j+1}^2/2 - V_{i,j-1}^2/h_{i,j-1} - Gh_{i,j-1}^2/2]
\end{aligned}$$

Fig. 1. Spatial Semi-Discretization

$$\begin{aligned}
\frac{\partial h_{i,j}}{\partial t} &= F_h \\
\frac{\partial U_{i,j}}{\partial t} &= F_u \\
\frac{\partial V_{i,j}}{\partial t} &= F_v
\end{aligned}$$

Fig. 2. Semi-Discrete Evolution Equations

## II. BACKGROUND

The Navier-Stokes Equations are a series of partial differential equations that arise from Newton's Second Law in the context of fluid motion and describe the motion of viscous fluids [4]. The Shallow Water Equations are a set of partial differential equations that are derived by depth-integrating the Navier-Stokes Equations. A key aspect of these new equations is that the depth of the fluid being studied is much less than its horizontal dimensions [5]. Due to the Earth's atmosphere being far shallower than its surface area, these new equations are a good model for atmospheric predictions.

Solving partial differential equations such as these is difficult, and instead of being solved directly, they can be discretized into a series of algebraic equations via the Finite Difference Method to be solved numerically. The first step in discretizing the Shallow Water Equations is to plot the area being studied on a grid with spacing  $\Delta x$  between each point. The spatial semi-discretized equations are then defined as shown in Figure 1. Each of  $h$ ,  $U$ , and  $V$ , are located at grid points  $(i\Delta x, j\Delta x)$  for non-negative integers  $(i, j)$ , denoted by  $h_{i,j}$ ,  $U_{i,j}$ , and  $V_{i,j}$ , [3]. Using periodic boundary conditions, the semi-discrete equations are then as shown in Figure 2. In order to fully discretize the equations, a time integrator must be chosen. In this case, we used the second-order Runge-Kutta (RK2) method due to its simplicity and compatibility with the semi-discretization above [3]. This works by computing the solutions to the equations a half time step forward and computing the slope between that point and the half time step point. The equations at this half time step are then as shown in Figure 3. The time step size is defined as  $\Delta t$  and the field will be solved at every positive integer multiple of the time step size,  $n\Delta t$ . The solutions for  $h$ ,  $U$ , and  $V$  at each time step are denoted as  $h_{i,j}^n$ ,  $U_{i,j}^n$ , and  $V_{i,j}^n$ . After this is done, a similar calculation is done, but instead the data at the half time step is used as the second point in the slope (Figure 4). This process is repeated for every time step until the end of the simulation.

Our simulation employs the Finite Volume Method; a slight variance on the Finite Difference Method. The Finite Volume Method is very similar, but includes a term for diffusion. This term smooths out high-frequency nodes that occur as a consequence of a finite grid resolution [6].

$$\begin{aligned}
h_{i,j}^* &= h_{i,j}^n + \frac{\Delta t}{2} F_h(h^n, U^n, V^n) \\
U_{i,j}^* &= U_{i,j}^n + \frac{\Delta t}{2} F_u(h^n, U^n, V^n) \\
V_{i,j}^* &= V_{i,j}^n + \frac{\Delta t}{2} F_v(h^n, U^n, V^n)
\end{aligned}$$

Fig. 3. Half Time Step Calculation

$$\begin{aligned}
h_{i,j}^{n+1} &= h_{i,j}^n + \Delta t F_h(h^*, U^*, V^*) \\
U_{i,j}^{n+1} &= U_{i,j}^n + \Delta t F_u(h^*, U^*, V^*) \\
V_{i,j}^{n+1} &= V_{i,j}^n + \Delta t F_v(h^*, U^*, V^*)
\end{aligned}$$

Fig. 4. Whole Time Step Calculation

### III. DESIGN

For the task of speeding up the Shallow Water Simulation, we decided that implementing the calculations fully in hardware would give us the greatest potential of speed up. This is because hardware accelerators are capable of completing their dedicated tasks much faster than generic CPU's. We would then interface the hardware with a C driver that would allow us to compare our design to that of one ran on a generic CPU. We were provided with a DE1-SoC development board for implementation of our hardware. Our design process began with obtaining a working simulation model of the Shallow Water Equations. UC Davis Department of Land, Air, and Water Resources Professor Paul Ullrich provided us with what he named the "Poke-Model" implementation of the Shallow Water Equations coded in MATLAB. This MATLAB code allows a user to simulate over various function types, and also allows for users to view the changing fluid in a visual simulation. Our group was able to use this implementation as a starting point. The hardest part about translating the MATLAB code given to us by Professor Ullrich was translating the function parameters returned by the MATLAB functions. They returned multiple variables at a time when the function is run. We translated this by taking in pointers for the different height and velocity arrays as parameters to the functions. This would update the memory space values as they were being computed in the separate functions. The other difficult part of translating the MATLAB code was translating the array indexing. In MATLAB this can all be done in one line. In C we would need to use for loops which required examination of the ranges needed to traverse in the array. We also merged parts parts of array traversal in the same loop if the indexes were the same instead of separate ones like in the MATLAB code in order to improve performance. Another method to improve performance was occasionally manually indexing through the array if the amount of elements needed to update was small. The MATLAB code was made to easily take in inputs, but the C code currently needs to hardcode in the initial function, final time, resolution, and the x and y domains. The MATLAB code contained functions that created the initial conditions of the Partial Differential Equation into a Gaussian shaped hill, a Geostrophic current, or a Gradient current. We added these as helper functions that took in the height and velocity arrays and calculated the initial conditions using math.h library functions. The initial conditions were stored into three separate arrays for height and the x and y velocities. The rest of the translation was pretty straightforward based on the MATLAB code. Separate helper functions were made for ApplyBoundaryConditions, CalculateFlux, and UpdateFlux. It followed the same structure as well. At the end of main, we print the initial conditions and the values of height and velocity for the last time step.

For the hardware implementation, we split portions of the MATLAB code into different modules. The goal of the hardware is for the hardware to be given the height(h), x-momentum(U), and y-momentum(V) of a given point, then the hardware will calculate these three terms for the next time step. Based off how shallow water simulation

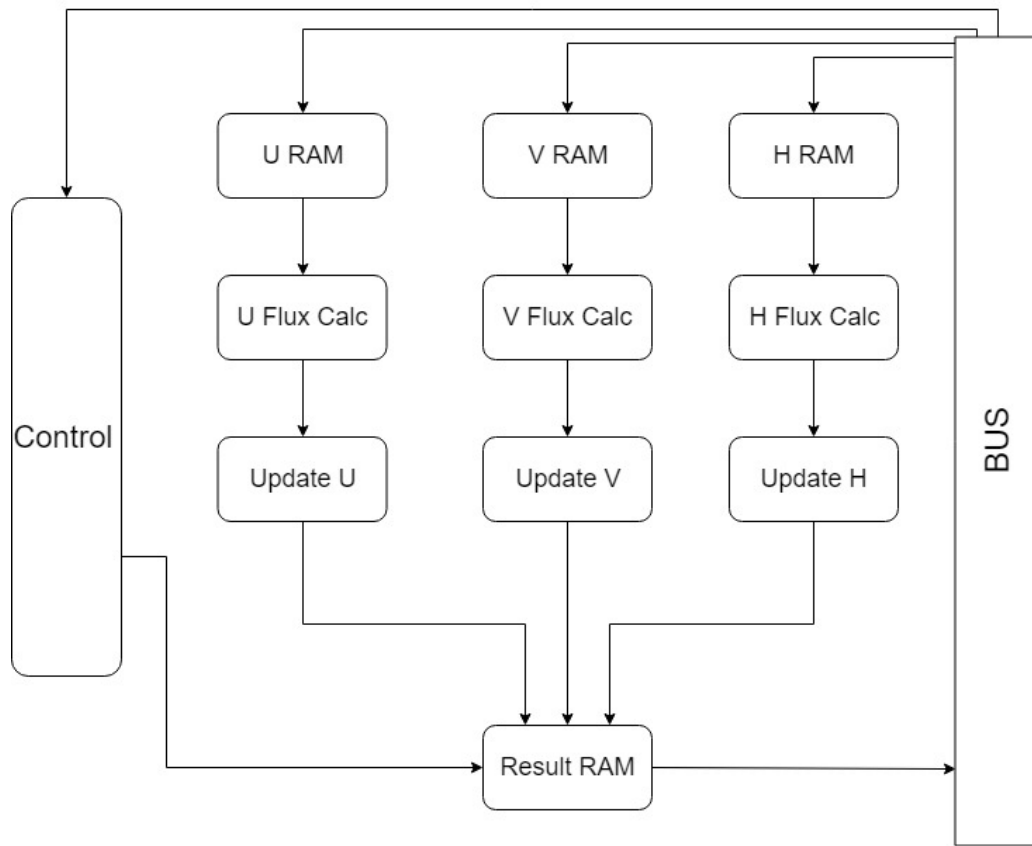


Fig. 5. Whole Time Step Calculation

works, we split the top level into two modules. One module, CalcFlux, calculates the flux of that specific grid point. The second module, Update, uses the flux to calculate the next point in time. CalcFluxes is fed from a RAM that has been filled by the Avalon Bus. Update then is fed the output of CalcFlux as well as the data for the current point. This output is then stored into a RAM which the Avalon Bus can read from. From Figure 5 you can see how the data flows through the modules. An important aspect about this design is that we can run the calculation for h, U and V all in parallel. This will help give us some speedup compared to the C implementation, which is designed to run the calculation for each variable sequentially.

Within the CalcFluxes module, there are submodules which assist the CalcFluxes module with its task. Starting with CalcFluxes, it relies on the modules SideStates, CalcSideFluxes and MaxWaveSpeed to operate. SideStates calculates the difference between the current points value and the next point in spaces value. This value is then fed into CalcSideFluxes which calculates the Flux in that particular direction. CalcFluxes is then able to use the flux on both sides to calculate the flux going through that particular point. MaxWaveSpeed calculates the theoretical maximum wave speed based of gravity. This takes into account of the viscosity of the fluid. This value is then incorporated into the algorithm to ensure that flux does not reach very unrealistic values. As the trend of flux be exponential going towards infinity, in reality there are real world constraints preventing this. From a hardware design aspect, this also helps ensure the values of flux do not become so large, they are unable to be stored with our determined bit widths.

For this project, we were limited by the resources on the board when it came to choosing the bit widths of our values. The largest limiting factor is the DSP blocks on the board are only able to hold 27 bits. We decided that our input h, U, and V values would need to be 13 bits. This was due to the fact that when we multiply these two numbers together we would get a 26 bit number, which would be the maximum for our DSP blocks. We also are limited by the lack of floating point support in verilog. It is crucial that our values have some kind of decimal

Flow Summary	
Flow Status	Successful - Mon May 6 12:47:54 2019
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name	divider
Top-level Entity Name	divider
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	689 / 56,480 ( 1 % )
Total registers	213
Total pins	144 / 268 ( 54 % )
Total virtual pins	0
Total block memory bits	0 / 7,024,640 ( 0 % )
Total DSP Blocks	0 / 156 ( 0 % )
Total HSSI RX PCSs	0 / 6 ( 0 % )
Total HSSI PMA RX Deserializers	0 / 6 ( 0 % )
Total HSSI TX PCSs	0 / 6 ( 0 % )
Total HSSI PMA TX Serializers	0 / 6 ( 0 % )
Total PLLs	0 / 13 ( 0 % )
Total DLLs	0 / 4 ( 0 % )

Fig. 6. ALM Resources Used

values. The only option we had then was implementing fixed point. It seemed appropriate to have Q6.7 format, as the values we would test in this project should not be too large. We also would need the data to be signed. The clear option for this would be 2s complement formatting as it is best suited for addition and subtraction.

An important aspect to our project was implementing division for our Q6.7 inputs. Based off of the Shallow Water Equations, to do a single point implementation we would require 5 dividers. Within Quartus, a divider already exists called LPM divide. We originally were going to use this divider; however the output is only that of a whole number. This would be very difficult to implement with fixed point numbers. After looking through GitHub, we found a math library based around fixed point by Sam Skalicky, updated by Tom Burke. Within this library was a suitable dividing function. We had to add a few modifications to allow it to take input and output two's complement, because the original version was designed to take signed magnitude. This divider was also suitable because it only used 1% of ALM resources as seen in Figure 6. This is usually a concern, due to dividers being notorious for using lots of resources.

A square root function is also required to calculate the Max Wave Speed. Our current square root module is an implementation Brian Kuhn had previously written for another class. However; a few modifications were required to get it working with our Q6.7 values. This too only used a small portion of the available resources.

For the bus communication, we decided to use four RAM modules that stored 16 16-bit words. 16-bit words were chosen as C doesn't have 13-bit integers, so it would be easier to transfer 16-bits through the bus, but convert that value from floating point to a Q6.7 fixed point value in C before sending it through the bus. Three of the RAM modules were made writeable from the Avalon Bus which stored the inputs needed to calculate the next h, U, and V. These stored the seven grid points needed to compute h, U, and V for the next time step. Dt was also stored in the h RAM module and dx was stored in the U RAM module. The results RAM module stored 9 outputs from

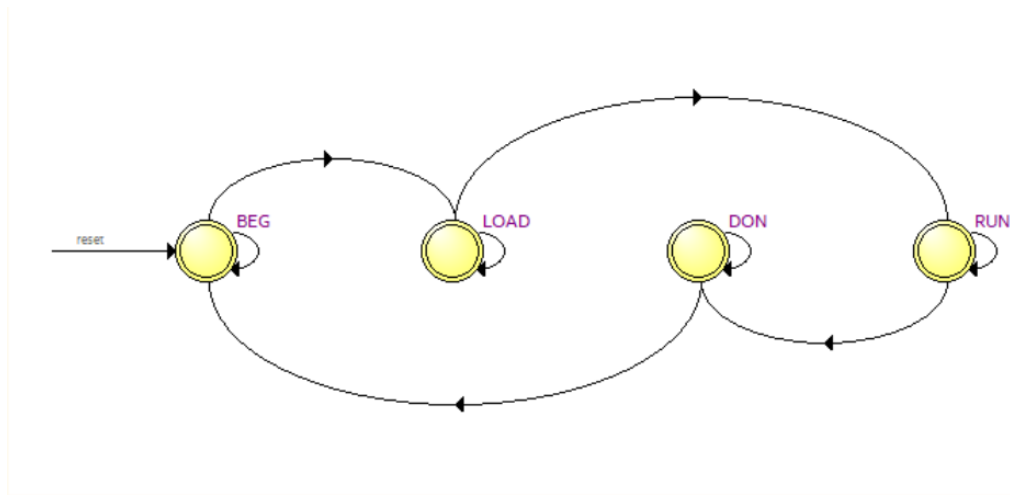


Fig. 7. Finite State Machine

calculating  $h$ ,  $U$ , and  $V$ . Even though, at most the modules need to store 9 words at a time, storing 16 words is easier for indexing and applying offsets since it is a power of 2. The communication is memory mapped, similar to Lab 3. Counting the 16 word space used for control, there would be 80 total words used for the RAM modules. This would be `slave_address`, a 7-bit number. The 4 LSB would be used for RAM module addressing, and the 3 MSB would be used for the offsets. The offsets used were 000 for control, 001 for the height RAM module, 010 for the  $x$  velocity RAM module, 011 for the  $y$  velocity RAM module, and 100 for the results module. The 4 LSB would always start as 0 at the offsets and would allow easy indexing for the RAM modules.

The Finite State Machine for bus communication is shown in Figure 7. In the BEG state, all of the enables for `UpdateFluxes` and `CalcFluxes`, the read and write addresses, and the write enable for the Results RAM module are initialized to 0. The hardware then waits for the RAM modules to be filled up and the start signal to be asserted high from the bus to transition to the LOAD state. The LOAD state is where the values needed are loaded to registers from the RAM modules. In this state the read addresses for the RAM modules are increased until they reach 7 and every clock cycle a new word is read from the RAM modules output into a register. When all data is needed from the modules, the state is switched to the RUN state. The RUN state must first calculate the values of  $h$ ,  $u$ , and  $v$  for half a time step, then a full time step following the RK2 method. This requires running `CalcFlux` and `UpdateFlux` twice using half of  $dt$  first. The RUN state is where all of the `CalcFlux` and `UpdateFlux` modules are ran. First the current input values to calculate  $h$ ,  $U$ , and  $V$  are saved and the enable signals for the `CalcFluxes` submodules are set to high. It then waits for the done signals from all the submodules to finish to enable the `UpdateFluxes` modules. Once the done signals from `UpdateFluxes` is received, the same process is run again using  $dt$  for the full time step value. Once all of the calculation modules are done, it is moved to the DON state. In the DON state, the write address of results is incremented by one and different output of the `UpdateFluxes` modules are loaded into the Results RAM module. Once the start signal is asserted low, it returns back to the BEG state. For the C code driver, in order to run on the FPGA Programmer, we needed to run the C code without `math.h` library functions. Instead of running the Gaussian, Gradient, and Geostrophic functions, we instead loaded the  $h$ ,  $U$ , and  $V$  arrays directly with values found from MATLAB. Each RAM module in verilog takes 32 bytes each, so the offsets for `matrixH`, `matrixHu`, and `matrixHv` to 32 bytes each. This code is similar to the standalone C code except for each time step, it doesn't run any of the calculation submodules of `UpdateFlux` and `CalculateFlux`. Instead for each time step loads the  $h$ ,  $U$ , and  $V$  values, translated from double into fixed point, manually into the RAM modules through the bus. Indexing by plus 1 each time. Then it sets the start signal to high, waits for done to be set to high, and then stores the output values converted back to double into a 2nd array for  $h$ ,  $U$ , and  $V$ . This is done for every element in the  $h$ ,  $U$ , and  $V$  arrays. On the next time step, the values are loaded in from the 2nd array instead of the first by checking the number of time steps modulo 2.

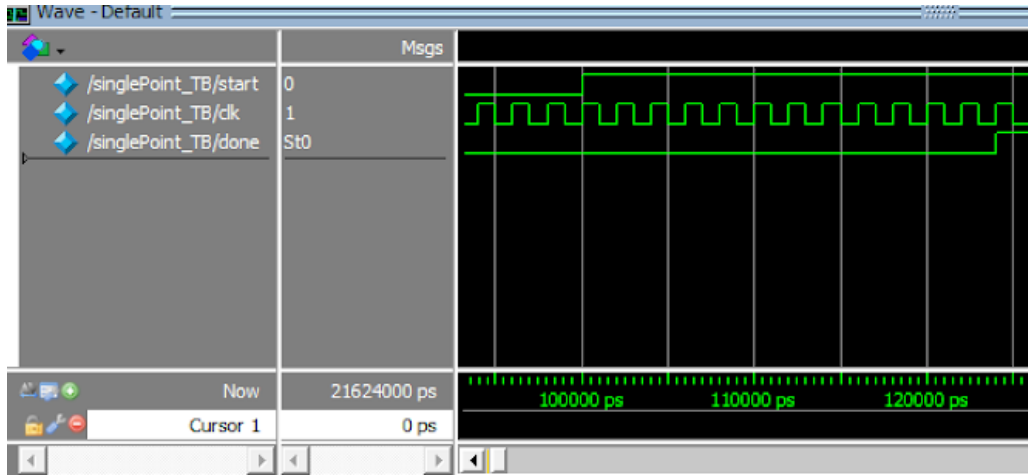


Fig. 8. ModelSim Simulation Showing Clock Cycles

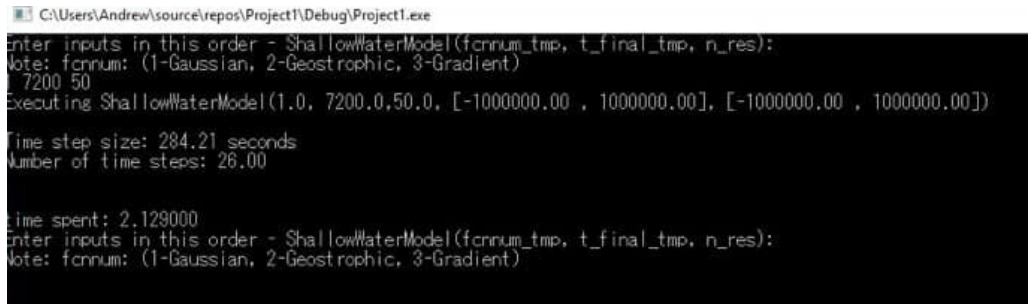


Fig. 9. Output of C Simulation

#### IV. RESULTS

To begin the testing portion of our code, we were able to implement a single point implementation of our hardware. This included a top level code that performs the calculation one point at a time. After compiling and loading this into modelsim, we were able to get the code to successfully update the values for a point and give a done signal. Counting the clock cycles between the start signal and done signal, we can see that the hardware takes 13 cycles to calculate one point's values for one time step. This is shown in Figure 8.

After our successful simulation, we then needed to find out the clock speed to see how long the calculation will be done in seconds. To do this we needed to synthesize the data in Quartus. After synthesizing, we found that the max clock speed is 128 MHz. This is ignoring the fact that the bus clock speed maced at only 100 MHz. Dividing the cycles by the clock speed gives us the number of seconds it will take to complete this calculation in hardware. Our result is  $1.02 \times 10^{-4}$  ms per point time step.

Comparing our hardware to real world models is difficult, because since we did not get our bus interfacing with the hardware correctly we do not know how much time is wasted in overhead. However if we ignore that fact our hardware looks fairly good. In our ported C implementation of this simulation we tested a 50 by 50 grid with 26 time steps as seen in Figure 9. This would require 65,000 points to calculate in total. With a speed of  $1.02 \times 10^{-4}$  ms per point we would get 0.663s which is 3.2 times faster than the 2.13s that the C code takes to do this calculation. Again this speedup representation would be different if we could get the C driver interfacing with the hardware, then we could see how much time is spent on overhead when loading the RAMs and on the setup the C driver preforms.

Applying this to the larger objective when it comes the global simulation, this could reduce the amount of time researchers waste waiting for their code to finish running. Most researchers use a global model that has 42,000 points with 8.5 million time steps. These simulations take researchers up to 24 hours. Again 18.5% of this simulation



$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

Fig. 10. Amdahl's Law

comes directly from doing calculations with the Navier-Stokes equations. Using Amdahl's law (Figure 10) we can see our overall speedup on the Energy Exascale Earth System Mode is 1.14 times faster which mean a 24 hour simulation would then become a 21 hour simulation saving 3 hours.

The results of the combined C code and Verilog were inconclusive because the C program would hang waiting for the done signal to be asserted. One of the most likely reasons is that we are indexing the different RAM modules incorrectly. We are only updating the memory offsets by one in the C code, but that only updates the memory location by one byte when we need to update them by 2 bytes. Also, the bus width is most likely set to 32 bits while I mistakenly thought it was set to 16 bits which would cause the inputs to the bus to be incorrect or update the wrong memory spaces. Another reason for the C code hanging is that the control logic is flawed in verilog, and doesn't update done.

Even if the bus interfacing worked correctly, the current implementation wouldn't be any faster than the standalone C code because we are currently only calculating one by one currently in verilog which is the same method used by the standalone C code.

## V. FUTURE WORK

Our hardware also synthesized with lots of room left on the board. As seen in Figure 11 our hardware implementation only used 8% ALMs, 16% DSPs, and 3% total memory of the board. This single point implementation used less resources than we expected, and would allow us to add more parallelization. With the DSPs being our limiting factor with 16% of them being used, we could conformably add 5 more cloned copies of our single point implementation, to allow for 6 points to be calculated at once. This would cause our hardware to look similar to Figure 12. Again this would just be copies of the single point implementation, each with their own RAMs and modules.

Another way to improve the current hardware implementation is to pipeline the calls to the CalcFluxes and UpdateFluxes modules and load up new grid points every clock cycle. The single point verilog module could then be calculating for multiple grid points each clock cycle. This could be done by instead of loading single points into the module, columns of data could be loaded into the module and it would add new data by going down the column. Our hardware implementation could also benefit with more accurate values. For example, our current square root module has lots of roundoff, due to its inability to only be able to handle integer numbers. Also, with a more advanced FPGA board we could use a larger bit width than the current 13 bit implementation. This would greatly increase our accuracy, and would make our hardware more applicable to real world use. Each of these improvements would increase the surface area of our design, raising the cost of our hardware and requiring lots of power.

On the software side, we would need more robust control logic in order to accommodate running more calculation modules and the higher throughput being outputted. We need a more accurate double to fixed point conversion method because our current implementation is very inaccurate and cuts off bits. Linux needs to be properly running on the FPGA board with our hardware verilog in order to use the math.h functions in the C code to load in the initial values. Software and hardware communication needs to be fixed by fixing the indexing of the RAM modules, and debugging the control logic. Finally the code could add functionality to offload the data into a text file to be read by visualization software. The data could also be offloaded onto a usb in order to have enough space to calculate very large resolutions.

Current computers that run these types of calculations already consume lots of power and cost a lot. The National Oceanic and Atmospheric Administration (NOAA) recently upgraded its datacenter with \$44.5million worth of equipment. Their recent upgrades give them the ability to do 8 quadrillion calculations per second. These facilities



Flow Status	Successful - Mon Jun 03 10:05:47 2019
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	lab1
Top-level Entity Name	lab1
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	2,462 / 32,070 ( 8 % )
Total registers	2890
Total pins	139 / 457 ( 30 % )
Total virtual pins	0
Total block memory bits	133,120 / 4,065,280 ( 3 % )
Total DSP Blocks	14 / 87 ( 16 % )
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 ( 0 % )
Total DLLs	1 / 4 ( 25 % )

Fig. 11. FPGA Resources Used

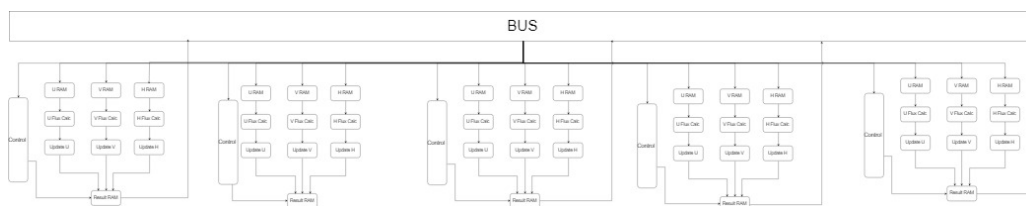


Fig. 12. Parallel Implementation

include water cooling infrastructure to keep all the hardware running at a reasonable temperature. With all these features, it shows how important it would be to climate modeling organizations to reduce the amount of power being consumed by the hardware, as they are already facing overheating issues [7].

#### ACKNOWLEDGMENTS

Special thanks to Professor Paul Ullrich and Professor Joeseeph Biello for their invaluable help and support. Professor Ullrich provided us with the baseline knowledge about the subject and helped us identify a good bottleneck to focus our project on. Professor Biello taught us about PDEs and discretization methods to help us understand the underlying math for our work.

Thank you also to Professor Venkatesh Akella and our teaching assistants, Terry O'Neil and Satyabrata Sarangi for guiding us and helping in the design process.

#### REFERENCES

- [1] US Department of Commerce and NOAA, "About Supercomputers," *National Weather Service*, 03-Aug-2017. [Online]. Available: <https://www.weather.gov/about/supercomputers>. [Accessed: 04-Jun-2019].
- [2] Staff, "What Does it Take to Simulate the Entire Earth?," *Engineering.com*, 11-May-2018. [Online]. Available: <https://www.engineering.com/DesignerEdge/DesignerEdgeArticles/ArticleID/16932/What-Does-it-Take-to-Simulate-the-Entire-Earth.aspx>. [Accessed: 05-Jun-2019].
- [3] P. Ullrich, "RE: Senior Design Project in Weather Prediction," *RE: Senior Design Project in Weather Prediction*, 01-Mar-2019.
- [4] "Navier–Stokes equations," *Wikipedia*, 31-May-2019. [Online]. Available: [https://en.wikipedia.org/wiki/Navier–Stokes\\_equations](https://en.wikipedia.org/wiki/Navier–Stokes_equations). [Accessed: 06-Jun-2019].
- [5] "Shallow water equations," *Wikipedia*, 20-Apr-2019. [Online]. Available: [https://en.wikipedia.org/wiki/Shallow\\_water\\_equations](https://en.wikipedia.org/wiki/Shallow_water_equations). [Accessed: 06-Jun-2019].
- [6] P. Ullrich, "RE: Shallow Water Equation Software," *RE: Shallow Water Equation Software*, 13-Mar-2019.
- [7] "NOAA kicks off 2018 with massive supercomputer upgrade," *National Oceanic and Atmospheric Administration*. [Online]. Available: <https://www.noaa.gov/media-release/noaa-kicks-off-2018-with-massive-supercomputer-upgrade>. [Accessed: 06-Jun-2019].

## APPENDIX: SOURCE CODE

The following pages contain the source code for our project; starting with the software C files and followed by the hardware Verilog files.

```

1 // File: CShallowWater.c
2 // Created by Brian Dang and Andrew Lu
3 // EEC 181
4 // Ported MATLAB code to calculate Shallow Water Flow for different resolutions and time
  frames
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <stdint.h>
9 #include <time.h>
10 #include <math.h>
11 #define RES 50
12 #define HALO 2
13
14 double gravity = 9.80616;
15
16 void GaussianFcn(double x, double y, double *h, double *u, double *v)
17 {
18     double a;
19
20     /* Radius from the center of the hill */
21     a = sqrt(x * x + y * y);
22
23     /* Half-width of the hill */
24     /* Fluid height perturbation */
25     /* Height of the hill at this point */
26     *h = 25.0 + 5.0 * exp(-(a * a) / 1.6E+11);
27
28     /* Zero velocity */
29     *u = 0.0;
30     *v = 0.0;
31 }
32
33 void GeostrophicFcn(double x, double y, double *h, double *u, double *v, double omega)
34 {
35     double h_tmp;
36     (void)x;
37
38     /* Background height */
39     /* Background velocity */
40     /* Latitude (override) */
41     /* Coriolis parameter (override) */
42     /* Wavelength */
43     /* Height of the hill at this point */
44     h_tmp = 6.2831853071795862 * y / 1.0E+6;
45     *h = 25.0 + 30.0 * (2.0 * omega * 0.70710678118654746) / gravity * 1.0E+6 /
46         6.2831853071795862 * cos(h_tmp);
47
48     /* Fixed zonal velocity */
49     *u = 30.0 * sin(h_tmp);
50
51     /* Zero meridional velocity */
52     *v = 0.0;
53 }
54
55 void GradientFcn(double x, double y, double *h, double *u, double *v)
56 {
57     double r;
58     double theta;
59     double uphi;
60     double h_tmp;
61
62     /* Radius from the center of the hill */
63     r = sqrt(x * x + y * y);
64
65     /* Angle from the center of the hill */
66     theta = atan2(y, x);
67
68     /* Half-width of the hill */

```

```

69  /* Fluid height perturbation */
70  /* Background fluid height */
71  /* Calculate fluid height */
72  uphi = r * r;
73  h_tmp = exp(-uphi / 4.0E+10);
74  *h = 25.0 - 20.0 * h_tmp;
75
76  /* Rotational velocity */
77  uphi = sqrt(gravity * 20.0 * (2.0 * uphi / 4.0E+10) * h_tmp);
78  if (r == 0.0) {
79      *u = 0.0;
80      *v = 0.0;
81  } else {
82      *u = -sin(theta) * uphi;
83      *v = cos(theta) * uphi;
84  }
85  }
86
87  void ApplyBoundaryConditions(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES +
2*HALO][RES + 2*HALO], double hv[RES + 2*HALO][RES + 2*HALO]){
88      int i;
89      int j;
90      for(j = 0; j < RES + 2*HALO; j++){
91          h[0][j] = h[RES][j];
92          h[1][j] = h[RES + 1][j];
93          h[RES + HALO][j] = h[HALO][j];
94          h[(RES + 2 * HALO) - 1][j] = h[2 * HALO - 1][j];
95
96          hu[0][j] = hu[RES][j];
97          hu[1][j] = hu[RES + 1][j];
98          hu[RES + HALO][j] = hu[HALO][j];
99          hu[(RES + 2 * HALO) - 1][j] = hu[2 * HALO - 1][j];
100
101          hv[0][j] = hv[RES][j];
102          hv[1][j] = hv[RES + 1][j];
103          hv[RES + HALO][j] = hv[HALO][j];
104          hv[(RES + 2 * HALO) - 1][j] = hv[2 * HALO - 1][j];
105      }
106      for(i = 0; i < RES + 2*HALO; i++){
107          h[i][0] = h[0][RES];
108          h[i][1] = h[1][RES + 1];
109          h[i][RES + HALO] = h[i][HALO];
110          h[i][(RES + 2 * HALO) - 1] = h[i][2 * HALO - 1];
111
112          hu[i][0] = hu[0][RES];
113          hu[i][1] = hu[1][RES + 1];
114          hu[i][RES + HALO] = hu[i][HALO];
115          hu[i][(RES + 2 * HALO) - 1] = hu[i][2 * HALO - 1];
116
117          hv[i][0] = hv[0][RES];
118          hv[i][1] = hv[1][RES + 1];
119          hv[i][RES + HALO] = hv[i][HALO];
120          hv[i][(RES + 2 * HALO) - 1] = hv[i][2 * HALO - 1];
121      }
122  }
123
124
125  void CalculateXFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES
+ 2*HALO],
126      double hv[RES + 2*HALO][RES + 2*HALO], double F[RES + 1][RES][3]){
127      int i;
128      int j;
129      int io;
130      int jo;
131      double h_left;
132      double hu_left;
133      double hv_left;
134      double h_right;
135      double hu_right;

```

```

136     double hv_right;
137     double h_flux_left;
138     double hu_flux_left;
139     double hv_flux_left;
140     double h_flux_right;
141     double hu_flux_right;
142     double hv_flux_right;
143     double max_wave_speed;
144
145     for(i = 0; i < RES + 1; i++){
146         for(j = 0; j < RES; j++){
147             io = i + HALO;
148             jo = j + HALO;
149
150             // Calculate left state (to 2nd order accuracy)
151             h_left = 0.25 * h[io][jo] + h[io-1][jo] - 0.25 * h[io-2][jo];
152             hu_left = 0.25 * hu[io][jo] + hu[io-1][jo] - 0.25 * hu[io-2][jo];
153             hv_left = 0.25 * hv[io][jo] + hv[io-1][jo] - 0.25 * hv[io-2][jo];
154
155             // Calculate right state (to 2nd order accuracy)
156             h_right = 0.25 * h[io-1][jo] + h[io][jo] - 0.25 * h[io+1][jo];
157             hu_right = 0.25 * hu[io-1][jo] + hu[io][jo] - 0.25 * hu[io+1][jo];
158             hv_right = 0.25 * hv[io-1][jo] + hv[io][jo] - 0.25 * hv[io+1][jo];
159
160             // Left flux
161             h_flux_left = hu_left;
162             hu_flux_left = hu_left * hu_left / h_left + 0.5 * gravity * h_left * h_left;
163             hv_flux_left = hu_left * hv_left / h_left;
164
165             h_flux_right = hu_right;
166             hu_flux_right = hu_right * hu_right / h_right + 0.5 * gravity * h_right *
167             h_right;
168             hv_flux_right = hu_right * hv_right / h_right;
169
170             // Max wave speed
171             max_wave_speed = abs(0.5 * (hu_left / h_left + hu_right / h_right)) +
172             sqrt(gravity * 0.5 * (h_left + h_right));
173
174             // Calculate flux
175             F[i][j][0] = 0.5 * (h_flux_left + h_flux_right) - 0.5 * max_wave_speed *
176             (h_right - h_left);
177             F[i][j][1] = 0.5 * (hu_flux_left + hu_flux_right) - 0.5 * max_wave_speed *
178             (hu_right - hu_left);
179             F[i][j][2] = 0.5 * (hv_flux_left + hv_flux_right) - 0.5 * max_wave_speed *
180             (hv_right - hv_left);
181
182         }
183     }
184
185     void UpdateXFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES +
186     2*HALO],
187     double hv[RES + 2*HALO][RES + 2*HALO], double XFlux[RES + 1][RES][3], double dx,
188     double dt){
189         int io;
190         int jo;
191         int i;
192         int j;
193         for(i = 0; i < RES + 1; i++){
194             for(j = 0; j < RES; j++){
195                 io = i + HALO;
196                 jo = j + HALO;
197
198                 h[io][jo] = h[io][jo] + dt / dx * XFlux[i][j][0];
199                 h[io-1][jo] = h[io-1][jo] - dt / dx * XFlux[i][j][0];
200
201                 hu[io][jo] = hu[io][jo] + dt / dx * XFlux[i][j][1];
202                 hu[io-1][jo] = hu[io-1][jo] - dt / dx * XFlux[i][j][1];

```

```

198         hv[io][jo] = hv[io][jo] + dt / dx * XFlux[i][j][2];
199         hv[io-1][jo] = hv[io-1][jo] - dt / dx * XFlux[i][j][2];
200
201     }
202 }
203 }
204
205 void CalculateYFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES
+ 2*HALO],
206     double hv[RES + 2*HALO][RES + 2*HALO], double F[RES][RES + 1][3]){
207     int i;
208     int j;
209     int io;
210     int jo;
211     double h_left;
212     double hu_left;
213     double hv_left;
214     double h_right;
215     double hu_right;
216     double hv_right;
217     double h_flux_left;
218     double hu_flux_left;
219     double hv_flux_left;
220     double h_flux_right;
221     double hu_flux_right;
222     double hv_flux_right;
223     double max_wave_speed;
224
225     for(i = 0; i < RES; i++){
226         for(j = 0; j < RES + 1; j++){
227             io = i + HALO;
228             jo = j + HALO;
229
230             // Calculate left state (to 2nd order accuracy)
231             h_left = 0.25 * h[io][jo] + h[io][jo-1] - 0.25 * h[io][jo-2];
232             hu_left = 0.25 * hu[io][jo] + hu[io][jo-1] - 0.25 * hu[io][jo-2];
233             hv_left = 0.25 * hv[io][jo] + hv[io][jo-1] - 0.25 * hv[io][jo-2];
234
235             // Calculate right state (to 2nd order accuracy)
236             h_right = 0.25 * h[io][jo-1] + h[io][jo] - 0.25 * h[io][jo+1];
237             hu_right = 0.25 * hu[io][jo-1] + hu[io][jo] - 0.25 * hu[io][jo+1];
238             hv_right = 0.25 * hv[io][jo-1] + hv[io][jo] - 0.25 * hv[io][jo+1];
239
240             // Left flux
241             h_flux_left = hu_left;
242             hv_flux_left = hu_left * hu_left / h_left + 0.5 * gravity * h_left * h_left;
243             hu_flux_left = hu_left * hv_left / h_left;
244
245             h_flux_right = hu_right;
246             hv_flux_right = hu_right * hu_right / h_right + 0.5 * gravity * h_right *
h_right;
247             hu_flux_right = hu_right * hv_right / h_right;
248
249             // Max wave speed
250             max_wave_speed = abs(0.5 * (hu_left / h_left + hu_right / h_right)) +
sqrt(gravity * 0.5 * (h_left + h_right));
251
252             // Calculate flux
253             F[i][j][0] = 0.5 * (h_flux_left + h_flux_right) - 0.5 * max_wave_speed *
(h_right - h_left);
254             F[i][j][1] = 0.5 * (hu_flux_left + hu_flux_right) - 0.5 * max_wave_speed *
(hu_right - hu_left);
255             F[i][j][2] = 0.5 * (hv_flux_left + hv_flux_right) - 0.5 * max_wave_speed *
(hv_right - hv_left);
256
257         }
258     }
259 }
260

```



```

261 void UpdateYFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES +
262 2*HALO],
263     double hv[RES + 2*HALO][RES + 2*HALO], double YFlux[RES][RES + 1][3], double dx,
264     double dt){
265     int io;
266     int jo;
267     int i;
268     int j;
269
270     for(i = 0; i < RES; i++){
271         for(j = 0; j < RES + 1; j++){
272             io = i + HALO;
273             jo = j + HALO;
274
275             h[io][jo] = h[io][jo] + dt / dx * YFlux[i][j][0];
276             h[io][jo-1] = h[io][jo-1] - dt / dx * YFlux[i][j][0];
277
278             hu[io][jo] = hu[io][jo] + dt / dx * YFlux[i][j][1];
279             hu[io][jo-1] = hu[io][jo-1] - dt / dx * YFlux[i][j][1];
280
281             hv[io][jo] = hv[io][jo] + dt / dx * YFlux[i][j][2];
282             hv[io][jo-1] = hv[io][jo-1] - dt / dx * YFlux[i][j][2];
283         }
284     }
285
286     int offset(int x, int y, int t){
287         return (t * RES * RES) + (y * RES) + x;
288     }
289
290     double ShallowWaterModel(int fcnnum, double t_final, int n_res, double x_domain[2],
291     double y_domain[2], double* X, double* Y, double **T, double Minith[RES][RES], double
292     Minithu[RES][RES], double Minithv[RES][RES], double **Mh, double **Mhu, double **Mhv){
293         double phi0 = 0.78539816339744828; //latitude
294         double cfl = 0.5;
295         double omega = pow(7.292, -5); //rotation rate
296
297         double f = 2 * omega * sin(phi0);
298
299         int halo = 2;
300
301         double dx = (x_domain[1] - x_domain[0]) / n_res;
302         double dy = (y_domain[1] - y_domain[0]) / n_res;
303
304         int i;
305         int j;
306         double XEdge[n_res + 2];
307         double YEdge[n_res + 2];
308         double XFlux[RES + 1][RES][3];
309         double YFlux[RES][RES + 1][3];
310
311         for(i = 0; i <= n_res + 1; i++){
312             XEdge[i] = i * dx + x_domain[0];
313             YEdge[i] = i * dy + y_domain[0];
314         }
315
316         for(i = 0; i < n_res; i++){
317             X[i] = 0.5 * (XEdge[i] + XEdge[i+1]);
318             Y[i] = 0.5 * (YEdge[i] + YEdge[i+1]);
319         }
320
321         double h[RES + 2*halo][RES + 2*halo]; //height and momentums
322         double hu[RES + 2*halo][RES + 2*halo];
323         double hv[RES + 2*halo][RES + 2*halo];
324         double h_old[RES + 2*halo][RES + 2*halo]; //height and momentums
325         double hu_old[RES + 2*halo][RES + 2*halo];
326         double hv_old[RES + 2*halo][RES + 2*halo];

```

```

326     int io;
327     int jo;
328
329     for(i = 0; i < n_res; i++){
330         for(j = 0; j < n_res; j++){
331             io = i + halo;
332             jo = j + halo;
333
334             GaussianFcn(X[i], Y[i], &(h[io][jo]), &(hu[io][jo]), &(hv[io][jo]));
335
336             hu[io][jo] *= h[io][jo];
337             hv[io][jo] *= h[io][jo];
338
339             //doing Minit initial states with less loops
340             if((io >= halo && io < n_res + halo) && (jo >= halo && jo < n_res + halo)){
341                 Minith[i][j] = h[io][jo];
342                 Minithu[i][j] = hu[io][jo] / Minith[i][j];
343                 Minithv[i][j] = hv[io][jo] / Minith[i][j];
344             }
345         }
346     }
347 }
348
349 //calculate initial timestep
350 double max_wave_speed = 0;
351 double wave_speed;
352 for(i = 0; i < n_res; i++){
353     for(j = 0; j < n_res; j++){
354         wave_speed = sqrt(hu[i][j] * hu[i][j]) / h[i][j] + sqrt(gravity * h[i][j]);
355
356         if(wave_speed > max_wave_speed)
357             max_wave_speed = wave_speed;
358     }
359 }
360
361 double dt = cfl * dx / max_wave_speed;
362
363 double nt = ceil(t_final / dt);
364
365 printf("Time step size: %f seconds\n", dt);
366 printf("Number of time steps: %f\n", nt);
367
368 *T = (double *)malloc(nt * sizeof(double));
369 *Mh = (double *)malloc(nt * RES * RES * sizeof(double));
370 *Mhu = (double *)malloc(nt * RES * RES * sizeof(double));
371 *Mhv = (double *)malloc(nt * RES * RES * sizeof(double));
372
373 if(*T)
374     printf("T malloc success\n");
375 if(*Mh)
376     printf("Mh malloc success\n");
377 if(*Mhu)
378     printf("Mhu malloc success\n");
379 if(*Mhv)
380     printf("Mhv malloc success\n");
381
382
383 double t_current = 0;
384 int t;
385
386 for(t = 0; t < (int)nt; t++){
387     //special handling for final timestep
388     if(t == nt)
389         dt = t_final - t_current;
390
391     ApplyBoundaryConditions(h, hu, hv);
392
393     //store old values

```

```

395     for(i = 0; i < RES + 2*halo; i++){
396         for(j = 0; j < RES + 2*halo; j++){
397             h_old[i][j] = h[i][j];
398             hu_old[i][j] = hu[i][j];
399             hv_old[i][j] = hv[i][j];
400         }
401     }
402
403     //predictor step
404     CalculateXFluxes(h, hu, hv, XFlux);
405     CalculateYFluxes(h, hu, hv, YFlux);
406
407     //update half a timestep
408     UpdateXFluxes(h, hu, hv, XFlux, dx, 0.5 * dt);
409     UpdateYFluxes(h, hu, hv, YFlux, dx, 0.5 * dt);
410
411     // Apply source terms
412     for(i = 0; i < RES; i++){
413         for(j = 0; j < RES; j++){
414             io = i + halo;
415             jo = j + halo;
416
417             hu[io][jo] = hu[io][jo] + 0.5 * dt * f * hv_old[io][jo];
418             hv[io][jo] = hv[io][jo] - 0.5 * dt * f * hu_old[io][jo];
419         }
420     }
421
422     ApplyBoundaryConditions(h, hu, hv);
423
424     CalculateXFluxes(h, hu, hv, XFlux);
425     CalculateYFluxes(h, hu, hv, YFlux);
426
427     for(i = 0; i < RES + 2*halo; i++){
428         for(j = 0; j < RES + 2*halo; j++){
429             h[i][j] = h_old[i][j];
430             hu[i][j] = hu_old[i][j];
431             hv[i][j] = hv_old[i][j];
432         }
433     }
434
435     UpdateXFluxes(h, hu, hv, XFlux, dx, dt);
436     UpdateYFluxes(h, hu, hv, YFlux, dx, dt);
437
438     t_current = t_current + dt;
439
440     (*T)[t] = t_current;
441
442     int off = 0;
443     //store h, u, v for current timestep
444     for(i = 0; i < RES; i++){
445         for(j = 0; j < RES; j++){
446             off = offset(i, j, t);
447             (*Mh)[off] = h[HALO + i][HALO + j];
448             //printf("h: %f\n", h[HALO + i][HALO + j]);
449             (*Mhu)[off] = hu[HALO + i][HALO + j] / h[HALO + i][HALO + j];
450             //printf("hu: %f\n", hu[HALO + i][HALO + j]);
451             (*Mhv)[off] = hv[HALO + i][HALO + j] / h[HALO + i][HALO + j];
452             //printf("hv: %f\n", hv[HALO + i][HALO + j]);
453         }
454     }
455
456 }
457
458 return nt;
459 }
460
461 int main(){
462     int fcnum = 0;
463     int n_res = RES;

```

```

464     double t_final = 7200;
465     double x_domain[2] = {-1000000, 1000000};
466     double y_domain[2] = {-1000000, 1000000};
467     double X[RES];
468     double Y[RES];
469     double *T;
470     double *Mh;
471     double *Mhu;
472     double *Mhv;
473     double Minith[n_res][n_res];
474     double Minithu[n_res][n_res];
475     double Minithv[n_res][n_res];
476     int i;
477     int j;
478     double nt;
479     clock_t t;
480
481     t = clock();
482
483     nt = ShallowWaterModel(fcnnum, t_final, n_res, x_domain, y_domain, X, Y, &T, Minith,
484                             Minithu, Minithv, &Mh, &Mhu, &Mhv);
485
486     t = clock() - t;
487     double time_taken = ((double)t)/CLOCKS_PER_SEC;
488
489     //print initial values
490     for(i = 0; i < n_res; i++){
491         for(j = 0; j < n_res; j++){
492             printf("Minith: ");
493             printf("%f ", Minith[i][j]);
494             if(j == n_res - 1)
495                 printf("\n");
496         }
497     }
498     printf("\n");
499
500     for(i = 0; i < n_res; i++){
501         for(j = 0; j < n_res; j++){
502             printf("Minithu: ");
503             printf("%f ", Minithu[i][j]);
504             if(j == n_res - 1)
505                 printf("\n");
506         }
507     }
508
509     printf("\n");
510
511     for(i = 0; i < n_res; i++){
512         for(j = 0; j < n_res; j++){
513             printf("Minithv: ");
514             printf("%f ", Minithv[i][j]);
515             if(j == n_res - 1)
516                 printf("\n");
517         }
518     }
519
520     printf("\n");
521
522     printf("nt: %f", nt);
523
524     //print values for final timestep
525     for(i = 0; i < n_res; i++){
526         for(j = 0; j < n_res; j++){
527             int off = offset(i, j, nt - 1);
528             printf("Mh: ");
529             printf("%f ", Mh[off]);
530             if(j == n_res - 1)
531                 printf("\n");

```

```

532     }
533 }
534
535 for(i = 0; i < n_res; i++){
536     for(j = 0; j < n_res; j++){
537         int off = offset(i, j, nt - 1);
538         printf("Mhu: ");
539         printf("%f ", Mhu[off]);
540         if(j == n_res - 1)
541             printf("\n");
542     }
543 }
544
545 for(i = 0; i < n_res; i++){
546     for(j = 0; j < n_res; j++){
547         int off = offset(i, j, nt - 1);
548         printf("Mhv: ");
549         printf("%f ", Mhv[off]);
550         if(j == n_res - 1)
551             printf("\n");
552     }
553 }
554
555 printf("Program took %f seconds to execute \n", time_taken);
556
557 free(T);
558 free(Mh);
559 free(Mhv);
560 free(Mhu);
561
562 return 0;
563 }

```

```

1 // File: CShallowWater.c
2 // Created by Brian Dang
3 // EEC 181
4 // C code to run with hardware Verilog, doesn't use math.h libraries
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <stdint.h>
9
10 #define RES 4
11 #define HALO 2
12 #define FIXED_POINT_FRACTIONAL_BITS 7
13
14 typedef uint16_t fixed_point_t;
15
16 double gravity = 9.80616;
17 volatile int *control = (int*) 0xFF200000;
18 volatile int *matrixH = (int*) 0xFF200020;
19 volatile int *matrixHu = (int*) 0xFF200040;
20 volatile int *matrixHv = (int*) 0xFF200060;
21 volatile int *matrixResult = (int*) 0xFF200080;
22 volatile int *led = (int*) 0xFF200140;
23 volatile int *switches = (int*) 0xFF200130;
24 volatile int *buttons = (int*) 0xFF200100;
25 volatile int *hex3_hex0 = (int*) 0xFF200120;
26 volatile int *hex5_hex4 = (int*) 0xFF200110;
27
28 inline fixed_point_t float_to_fixed(double input)
29 {
30     return (fixed_point_t)(input * (1 << FIXED_POINT_FRACTIONAL_BITS));
31 }
32
33 double fixed16_to_double(uint16_t input, uint8_t fractional_bits)
34 {
35     return ((double)input / (double)(1 << fractional_bits));
36 }
37
38 void ApplyBoundaryConditions(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES +
2*HALO][RES + 2*HALO], double hv[RES + 2*HALO][RES + 2*HALO]){
39     int i;
40     int j;
41     for(j = 0; j < RES + 2*HALO; j++){
42         h[0][j] = h[RES][j];
43         h[1][j] = h[RES + 1][j];
44         h[RES + HALO][j] = h[HALO][j];
45         h[(RES + 2 * HALO) - 1][j] = h[2 * HALO - 1][j];
46
47         hu[0][j] = hu[RES][j];
48         hu[1][j] = hu[RES + 1][j];
49         hu[RES + HALO][j] = hu[HALO][j];
50         hu[(RES + 2 * HALO) - 1][j] = hu[2 * HALO - 1][j];
51
52         hv[0][j] = hv[RES][j];
53         hv[1][j] = hv[RES + 1][j];
54         hv[RES + HALO][j] = hv[HALO][j];
55         hv[(RES + 2 * HALO) - 1][j] = hv[2 * HALO - 1][j];
56     }
57     for(i = 0; i < RES + 2*HALO; i++){
58         h[i][0] = h[0][RES];
59         h[i][1] = h[1][RES + 1];
60         h[i][RES + HALO] = h[i][HALO];
61         h[i][(RES + 2 * HALO) - 1] = h[i][2 * HALO - 1];
62
63         hu[i][0] = hu[0][RES];
64         hu[i][1] = hu[1][RES + 1];
65         hu[i][RES + HALO] = hu[i][HALO];
66         hu[i][(RES + 2 * HALO) - 1] = hu[i][2 * HALO - 1];
67
68         hv[i][0] = hv[0][RES];

```

```

69         hv[i][1] = hv[1][RES + 1];
70         hv[i][RES + HALO] = hv[i][HALO];
71         hv[i][(RES + 2 * HALO) - 1] = hv[i][2 * HALO - 1];
72     }
73
74 }
75
76 int offset(int x, int y, int t){
77     return (t * RES * RES) + (y * RES) + x;
78 }
79
80 double ShallowWaterModel(int fcnnum, double t_final, int n_res, double x_domain[2],
double y_domain[2], double* X, double* Y, double **T, double Minith[RES][RES], double
Minithu[RES][RES], double Minithv[RES][RES], double **Mh, double **Mhu, double **Mhv){
81     double phi0 = 0.78539816339744828;//latitude
82     double cfl = 0.5;
83
84     int halo = 2;
85
86     double dx = (x_domain[1] - x_domain[0]) / n_res;
87     double dy = (y_domain[1] - y_domain[0]) / n_res;
88
89     int i;
90     int j;
91     double XEdge[n_res + 2];
92     double YEdge[n_res + 2];
93     double XFlux[RES + 1][RES][3];
94     double YFlux[RES][RES + 1][3];
95
96
97     for(i = 0; i <= n_res + 1; i++){
98         XEdge[i] = i * dx + x_domain[0];
99         YEdge[i] = i * dy + y_domain[0];
100     }
101
102     for(i = 0; i < n_res; i++){
103         X[i] = 0.5 * (XEdge[i] + XEdge[i+1]);
104         Y[i] = 0.5 * (YEdge[i] + YEdge[i+1]);
105     }
106
107     double h[RES + 2*halo][RES + 2*halo];//height and momentums
108     double hu[RES + 2*halo][RES + 2*halo];
109     double hv[RES + 2*halo][RES + 2*halo];
110     double h2[RES + 2*halo][RES + 2*halo];//height and momentums
111     double hu2[RES + 2*halo][RES + 2*halo];
112     double hv2[RES + 2*halo][RES + 2*halo];
113     int io;
114     int jo;
115
116     //loading up my own values because I can't run math.h from GaussianFcn
117     for(i = 0; i < RES + 2*halo; i++){
118         for(j = 0; j < RES + 2*halo; j++){
119             if(i == 0 || j == 0 || i == 1 || j == 1 || i == RES + 2*halo - 1 || j == RES +
2*halo - 1 || i == RES + 2*halo - 2 || i == RES + 2*halo - 2){
120                 h[i][j] = 0;
121             }
122             else{
123                 h[i][j] = 30;
124             }
125             hu[i][j] = 0;
126             hv[i][j] = 0;
127         }
128     }
129     for(i = 0; i < n_res; i++){
130         for(j = 0; j < n_res; j++){
131             io = i + halo;
132             jo = j + halo;
133
134             //doing Minit initial states with less loops

```



```

135         if((io >= halo && io < n_res + halo) && (jo >= halo && jo < n_res + halo)){
136             Minith[i][j] = h[io][jo];
137             Minithu[i][j] = hu[io][jo] / Minith[i][j];
138             Minithv[i][j] = hv[io][jo] / Minith[i][j];
139         }
140     }
141 }
142 }
143 }
144
145 for(i = 0; i < RES + 2*halo; i++){
146     for(j = 0; j < RES + 2*halo; j++){
147         printf("h: %f ", h[i][j]);
148         if(j == (RES + 2*halo) - 1)
149             printf("\n");
150     }
151 }
152
153 double dt = 1.46;
154 double nt = 7;
155
156 printf("Time step size: %f seconds\n", dt);
157 printf("Number of time steps: %f\n", nt);
158
159 *T = (double *)malloc(nt * sizeof(double));
160 *Mh = (double *)malloc(nt * RES * RES * sizeof(double));
161 *Mhu = (double *)malloc(nt * RES * RES * sizeof(double));
162 *Mhv = (double *)malloc(nt * RES * RES * sizeof(double));
163
164 if(*T)
165     printf("T malloc success\n");
166 if(*Mh)
167     printf("Mh malloc success\n");
168 if(*Mhu)
169     printf("Mhu malloc success\n");
170 if(*Mhv)
171     printf("Mhv malloc success\n");
172
173
174 double t_current = 0;
175 int t;
176
177 for(t = 0; t < (int)nt; t++){
178     //special handling for final timestep
179     if(t == nt)
180         dt = t_final - t_current;
181
182     //starting verilog code, if statement to load from h or h2 array
183     if((int)nt % 2 == 0){//first case
184         ApplyBoundaryConditions(h, hu, hv);
185         for(i = 0; i < RES + 1; i++){
186             for(j = 0; j < RES + 1; j++){
187                 io = i + HALO;
188                 jo = j + HALO;
189
190                 //load values through bus
191                 *matrixH = float_to_fixed(h[io - 2][jo]);
192                 *(matrixH + 1) = float_to_fixed(h[io - 1][jo]);
193                 *(matrixH + 2) = float_to_fixed(h[io + 1][jo]);
194                 *(matrixH + 3) = float_to_fixed(h[io][jo]);
195                 *(matrixH + 4) = float_to_fixed(h[io][jo - 2]);
196                 *(matrixH + 5) = float_to_fixed(h[io][jo - 1]);
197                 *(matrixH + 6) = float_to_fixed(h[io][jo + 1]);
198                 *(matrixH + 7) = float_to_fixed(dt);
199
200                 *matrixHu = float_to_fixed(hu[io - 2][jo]);
201                 *(matrixHu + 1) = float_to_fixed(hu[io - 1][jo]);
202                 *(matrixHu + 2) = float_to_fixed(hu[io + 1][jo]);
203                 *(matrixHu + 3) = float_to_fixed(hu[io][jo]);

```

```

204 * (matrixHu + 4) = float_to_fixed(hu[io][jo - 2]);
205 * (matrixHu + 5) = float_to_fixed(hu[io][jo - 1]);
206 * (matrixHu + 6) = float_to_fixed(hu[io][jo + 1]);
207 * (matrixHu + 7) = float_to_fixed(dx);
208
209 *matrixHv = float_to_fixed(hv[io - 2][jo]);
210 * (matrixHv + 1) = float_to_fixed(hv[io - 1][jo]);
211 * (matrixHv + 2) = float_to_fixed(hv[io + 1][jo]);
212 * (matrixHv + 3) = float_to_fixed(hv[io][jo]);
213 * (matrixHv + 4) = float_to_fixed(hv[io][jo - 2]);
214 * (matrixHv + 5) = float_to_fixed(hv[io][jo - 1]);
215 * (matrixHv + 6) = float_to_fixed(hv[io][jo + 1]);
216
217 *control = 0x0002; //set start
218
219 //wait for done to go high
220 while(*control != 0xB555) {
221     *led = 0x0000AAAA;
222 }
223
224 *control = 0x00000000; //set start to 0
225
226 // Verify that done goes low
227 while (*control != 0xB554) {
228     *led = 0x0000000F;
229 }
230
231 //store output of verilog
232 h2[io][jo] = fixed16_to_double(*(matrixResult), 7);
233 h2[io-1][jo] = fixed16_to_double(*(matrixResult + 1), 7);
234 hu2[io][jo] = fixed16_to_double(*(matrixResult + 2), 7);
235 hu2[io-1][jo] = fixed16_to_double(*(matrixResult+3), 7);
236 hv2[io][jo] = fixed16_to_double(*(matrixResult+4), 7);
237 hv2[io-1][jo] = fixed16_to_double(*(matrixResult+5), 7);
238 h2[io][jo-1] = fixed16_to_double(*(matrixResult+6), 7);
239 hu2[io][jo-1] = fixed16_to_double(*(matrixResult+7), 7);
240 hv2[io][jo-1] = fixed16_to_double(*(matrixResult+8), 7);
241
242 }
243 }
244 int off = 0;
245 //store outputs into M arrays by timestep
246 for(i = 0; i < RES; i++){
247     for(j = 0; j < RES; j++){
248         off = offset(i, j, t);
249         (*Mh)[off] = h2[HALO + i][HALO + j];
250         //printf("h: %f\n", h[HALO + i][HALO + j]);
251         (*Mhu)[off] = hu2[HALO + i][HALO + j] / h[HALO + i][HALO + j];
252         //printf("hu: %f\n", hu[HALO + i][HALO + j]);
253         (*Mhv)[off] = hv2[HALO + i][HALO + j] / h[HALO + i][HALO + j];
254         //printf("hv: %f\n", hv[HALO + i][HALO + j]);
255     }
256 }
257
258 }
259 else{ //loading from 2nd array
260     for(i = 0; i < RES + 1; i++){
261         for(j = 0; j < RES + 1; j++){
262             io = i + HALO;
263             jo = j + HALO;
264
265             //load values through bus
266             *matrixH = float_to_fixed(h2[io - 2][jo]);
267             * (matrixH + 1) = float_to_fixed(h2[io - 1][jo]);
268             * (matrixH + 2) = float_to_fixed(h2[io + 1][jo]);
269             * (matrixH + 3) = float_to_fixed(h2[io][jo]);
270             * (matrixH + 4) = float_to_fixed(h2[io][jo - 2]);
271             * (matrixH + 5) = float_to_fixed(h2[io][jo - 1]);
272             * (matrixH + 6) = float_to_fixed(h2[io][jo + 1]);

```

```

273         *(matrixH + 7) = float_to_fixed(dt);
274
275         *matrixHu = float_to_fixed(hu2[io - 2][jo]);
276         *(matrixHu + 1) = float_to_fixed(hu2[io - 1][jo]);
277         *(matrixHu + 2) = float_to_fixed(hu2[io + 1][jo]);
278         *(matrixHu + 3) = float_to_fixed(hu2[io][jo]);
279         *(matrixHu + 4) = float_to_fixed(hu2[io][jo - 2]);
280         *(matrixHu + 5) = float_to_fixed(hu2[io][jo - 1]);
281         *(matrixHu + 6) = float_to_fixed(hu2[io][jo + 1]);
282         *(matrixHu + 7) = float_to_fixed(dx);
283
284         *matrixHv = float_to_fixed(hv2[io - 2][jo]);
285         *(matrixHv + 1) = float_to_fixed(hv2[io - 1][jo]);
286         *(matrixHv + 2) = float_to_fixed(hv2[io + 1][jo]);
287         *(matrixHv + 3) = float_to_fixed(hv2[io][jo]);
288         *(matrixHv + 4) = float_to_fixed(hv2[io][jo - 2]);
289         *(matrixHv + 5) = float_to_fixed(hv2[io][jo - 1]);
290         *(matrixHv + 6) = float_to_fixed(hv2[io][jo + 1]);
291
292         *control = 0x0002; //set start
293
294         //wait for done to go high
295         while(*control != 0xb555555) {
296             *led = 0x000AAAA;
297         }
298
299         *control = 0x0000000; //set start to 0
300
301         // Verify that done goes low
302         while (*control != 0xb5555554) {
303             *led = 0x0000000F;
304         }
305
306         //store outputs
307         h[io][jo] = fixed16_to_double(*(matrixResult), 7);
308         h[io-1][jo] = fixed16_to_double(*(matrixResult + 1), 7);
309         hu[io][jo] = fixed16_to_double(*(matrixResult + 2), 7);
310         hu[io-1][jo] = fixed16_to_double(*(matrixResult+3), 7);
311         hv[io][jo] = fixed16_to_double(*(matrixResult+4), 7);
312         hv[io-1][jo] = fixed16_to_double(*(matrixResult+5), 7);
313         h[io][jo-1] = fixed16_to_double(*(matrixResult+6), 7);
314         hu[io][jo-1] = fixed16_to_double(*(matrixResult+7), 7);
315         hv[io][jo-1] = fixed16_to_double(*(matrixResult+8), 7);
316
317     }
318
319 }
320 int off = 0;
321 //store h, u ,v for current timestep
322 for(i = 0; i < RES; i++){
323     for(j = 0; j < RES; j++){
324         off = offset(i, j, t);
325         (*Mh)[off] = h[HALO + i][HALO + j];
326         //printf("h: %f\n", h[HALO + i][HALO + j]);
327         (*Mhu)[off] = hu[HALO + i][HALO + j] / h[HALO + i][HALO + j];
328         //printf("hu: %f\n", hu[HALO + i][HALO + j]);
329         (*Mhv)[off] = hv[HALO + i][HALO + j] / h[HALO + i][HALO + j];
330         //printf("hv: %f\n", hv[HALO + i][HALO + j]);
331     }
332 }
333
334 }
335
336 t_current = t_current + dt;
337 //store current timestep
338 (*T)[t] = t_current;
339
340 }
341

```

```

342     return nt;
343 }
344
345 int main(){
346     int fcnum = 0;
347     int n_res = RES;
348     double t_final = 10;
349     double x_domain[2] = {-100, 100};
350     double y_domain[2] = {-100, 100};
351     double X[RES];
352     double Y[RES];
353     double *T;
354     double *Mh;
355     double *Mhu;
356     double *Mhv;
357     double Minith[n_res][n_res];
358     double Minithu[n_res][n_res];
359     double Minithv[n_res][n_res];
360     int i;
361     int j;
362     double nt;
363
364     nt = ShallowWaterModel(fcnum, t_final, n_res, x_domain, y_domain, X, Y, &T, Minith,
365                             Minithu, Minithv, &Mh, &Mhu, &Mhv);
366
367     for(i = 0; i < n_res; i++){
368         for(j = 0; j < n_res; j++){
369             printf("Minith: ");
370             printf("%f ", Minith[i][j]);
371             if(j == n_res - 1)
372                 printf("\n");
373         }
374     }
375     printf("\n");
376
377     for(i = 0; i < n_res; i++){
378         for(j = 0; j < n_res; j++){
379             printf("Minithu: ");
380             printf("%f ", Minithu[i][j]);
381             if(j == n_res - 1)
382                 printf("\n");
383         }
384     }
385
386     printf("\n");
387
388     for(i = 0; i < n_res; i++){
389         for(j = 0; j < n_res; j++){
390             printf("Minithv: ");
391             printf("%f ", Minithv[i][j]);
392             if(j == n_res - 1)
393                 printf("\n");
394         }
395     }
396
397     printf("\n");
398
399     printf("nt: %f", nt);
400
401     for(i = 0; i < n_res; i++){
402         for(j = 0; j < n_res; j++){
403             int off = offset(i, j, nt - 1);
404             printf("Mh: ");
405             printf("%f ", Mh[off]);
406             if(j == n_res - 1)
407                 printf("\n");
408         }
409     }

```

```

410
411     for(i = 0; i < n_res; i++){
412         for(j = 0; j < n_res; j++){
413             int off = offset(i, j, nt - 1);
414             printf("Mhu: ");
415             printf("%f ", Mhu[off]);
416             if(j == n_res - 1)
417                 printf("\n");
418         }
419     }
420
421     for(i = 0; i < n_res; i++){
422         for(j = 0; j < n_res; j++){
423             int off = offset(i, j, nt - 1);
424             printf("Mhv: ");
425             printf("%f ", Mhv[off]);
426             if(j == n_res - 1)
427                 printf("\n");
428         }
429     }
430
431     free(T);
432     free(Mh);
433     free(Mhv);
434     free(Mhu);
435
436     return 0;
437 }

```

```

1  /*
2  * bussedAndFSMTop.v - the top level module to communicate with C code and run
  computation modules
3  * Brian Dang - Shallow Water Simulation - EEC 181
4  * May 2019
5  */
6
7  `define H 2'b00
8  `define HU 2'b01
9  `define HV 2'b10
10
11  `define X 1'b0
12  `define Y 1'b1
13
14  // constants and computation parameters
15  `define HALO 2'd2
16  `define GRAVITY 13'b0010011100111// 9.80616 in decimal Q6.7 format - 13'b001001_1100111
17
18  `define CONTROL_OFFSET 3'b000
19  `define MATRIX_H_ADDRESS_OFFSET 3'b001
20  `define MATRIX_HU_ADDRESS_OFFSET 3'b010
21  `define MATRIX_HV_ADDRESS_OFFSET 3'b011
22  `define MATRIX_RESULT_ADDRESS_OFFSET 3'b100
23  `define CLEAR_INDEX 2'b0
24  `define MULTIPLY_INDEX 2'b01
25
26  module bussedAndFSMTop (
27      // NUM_DATA is number of initial data points
28      // NUM_GRID_SIMULT is number of grid points we'll be calculating at the same time
29      // signals to connect to an Avalon clock source interface
30      clk,
31      reset,
32
33      // signals to connect to an Avalon-MM slave interface
34      slave_address,
35      slave_read,
36      slave_write,
37      slave_readdata,
38      slave_writedata,
39      slave_byteenable
40  );
41      parameter DATA_WIDTH = 16;
42      parameter BEG = 2'b00;
43      parameter LOAD = 2'b01;
44      parameter RUN = 2'b11;
45      parameter DON = 2'b10;
46
47      input clk;
48      input reset;
49
50      input [6:0] slave_address;
51      input slave_read;
52      input slave_write;
53
54      output reg [DATA_WIDTH-1:0] slave_readdata;
55      input [DATA_WIDTH-1:0] slave_writedata;
56      input [(DATA_WIDTH/8)-1:0] slave_byteenable;
57
58      reg [7:0] dt_in;
59      reg [12:0] dx_in;
60      reg [7:0] dt_in_c;
61      reg [12:0] dx_in_c;
62      reg done;
63      reg done_c;
64
65      reg start;
66
67      // data matrices - MIGHT NEED TO GET REWORKED WITH BUS STUFF
68      reg signed [12:0] h [6:0]; // i-2, i-1, i+1, 0, j-2, j-2, j+1

```

```

69 reg signed [12:0] hu [6:0];
70 reg signed [12:0] hv [6:0];
71 reg signed [12:0] h_old [6:0];
72 reg signed [12:0] hu_old [6:0];
73 reg signed [12:0] hv_old [6:0];
74
75 reg signed [12:0] h_c [6:0]; // i-2, i-1, i+1, 0, j-2, j-2, j+1
76 reg signed [12:0] hu_c [6:0];
77 reg signed [12:0] hv_c [6:0];
78 reg signed [12:0] h_old_c [6:0];
79 reg signed [12:0] hu_old_c [6:0];
80 reg signed [12:0] hv_old_c [6:0];
81
82 // intermediate data
83 wire signed [25:0] h_flux_left, h_flux_right, hu_flux_left, hu_flux_right,
hv_flux_left, hv_flux_right;
84 wire signed [12:0] h_left, h_right, hu_left, hu_right, hv_left, hv_right;
85 wire signed [12:0] Fh_x, Fhu_x, Fhv_x, Fh_y, Fhu_y, Fhv_y;
86
87 // all the start control signals
88 //reg start_boundCon;
89 reg start_calcXFlux, start_calcYFlux;
90 reg start_upXFlux, start_upYFlux;
91 reg start_calcXFlux_c, start_calcYFlux_c;
92 reg start_upXFlux_c, start_upYFlux_c;
93
94 // all the done control signals
95 reg done_boundCon_h, done_boundCon_hu, done_boundCon_hv;
96 reg done_boundCon_h_c, done_boundCon_hu_c, done_boundCon_hv_c;
97 wire done_flux_left;
98 wire done_fh_x, done_fhu_x, done_fhv_x, done_fh_y, done_fhu_y, done_fhv_y;
99 wire done_uph_x, done_uphu_x, done_uphv_x, done_uph_y, done_uphu_y, done_uphv_y;
100
101 // other control signals
102 reg update_half; // for UpdateFluxes | 0 = whole, 1 = half
103 reg update_half_c; // for UpdateFluxes | 0 = whole, 1 = half
104
105 // buffer signals
106 wire signed [12:0] foobar, h_0_new, h_1_new_x, h_1_new_y, hu_0_new, hu_1_new_x,
hu_1_new_y, hv_0_new, hv_1_new_x, hv_1_new_y;
107
108 reg [3:0] rd_addressH;
109 reg [3:0] rd_addressHu;
110 reg [3:0] rd_addressHv;
111
112 reg [3:0] rd_addressH_c;
113 reg [3:0] rd_addressHu_c;
114 reg [3:0] rd_addressHv_c;
115
116 wire [DATA_WIDTH-1:0] matrixH_dout;
117 wire [DATA_WIDTH-1:0] matrixHu_dout;
118 wire [DATA_WIDTH-1:0] matrixHv_dout;
119 wire [DATA_WIDTH-1:0] matrixResult_dout;
120 reg [DATA_WIDTH-1:0] fluxData; //data to write to results vector
121 reg [DATA_WIDTH-1:0] fluxData_c;
122
123 wire matrixH_wren = slave_write & (slave_address[6:4] == `MATRIX_H_ADDRESS_OFFSET);
124 wire matrixHu_wren = slave_write & (slave_address[6:4] == `MATRIX_HU_ADDRESS_OFFSET);
125 wire matrixHv_wren = slave_write & (slave_address[6:4] == `MATRIX_HV_ADDRESS_OFFSET);
126
127 reg Rwr_en; //write enable for results
128 reg Rwr_en_c;
129 reg wr_addr;
130 reg wr_addr_c;
131
132 reg [1:0] state;
133 reg [1:0] state_c;
134
135 newhRam matrixH(

```



```

136         .clock ( clk),
137         .data ( slave_writedata ),
138         .rdaddress (rd_addressH),
139         .waddress (slave_address[3:0]),
140         .wren (matrixH_wren),
141         .q (matrixH_dout)
142     );
143
144     newhRam matrixHu(
145         .clock ( clk),
146         .data ( slave_writedata ),
147         .rdaddress (rd_addressHu),
148         .waddress (slave_address[3:0]),
149         .wren (matrixHu_wren),
150         .q (matrixHu_dout)
151     );
152
153     newhRam matrixHv(
154         .clock ( clk),
155         .data ( slave_writedata ),
156         .rdaddress (rd_addressHv),
157         .waddress (slave_address[3:0]),
158         .wren (matrixHv_wren),
159         .q (matrixHv_dout)
160     );
161
162     newhRam matrixResult(
163         .clock ( clk),
164         .data (fluxData),
165         .rdaddress (slave_address[3:0]),
166         .waddress (wr_addr),
167         .wren (Rwr_en),
168         .q (matrixResult_dout)
169     );
170
171     CalcFluxes XFluxH (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
172         .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
173         .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
174         .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
175         .halo(`HALO), .gravity(`GRAVITY),
176         .F(Fh_x), .done(done_fh_x)
177     );
178     CalcFluxes XFluxHU (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
179         .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
180         .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
181         .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
182         .halo(`HALO), .gravity(`GRAVITY),
183         .F(Fhu_x), .done(done_fhu_x)
184     );
185     CalcFluxes XFluxHV (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
186         .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
187         .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
188         .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
189         .halo(`HALO), .gravity(`GRAVITY),
190         .F(Fhv_x), .done(done_fhv_x)
191     );
192
193     CalcFluxes YFluxH (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
194         .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[6]),
195         .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[6]),
196         .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[6]),
197         .halo(`HALO), .gravity(`GRAVITY),
198         .F(Fh_y), .done(done_fh_y)
199     );
200     CalcFluxes YFluxHU (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
201         .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[6]),
202         .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[6]),
203         .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[6]),
204         .halo(`HALO), .gravity(`GRAVITY),

```

```

205         .F(Fhu_y), .done(done_fhu_y)
206     );
207     CalcFluxes YFluxHV (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
208         .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[6]),
209         .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[6]),
210         .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[6]),
211         .halo(`HALO), .gravity(`GRAVITY),
212         .F(Fhv_y), .done(done_fhv_y)
213     );
214
215     // update fluxes
216     UpdateFluxes XUpH (.clk(clk), .start(start_upXFlux), .half(update_half),
217         .h_0 (h[3]), .h_1(h[1]),
218         .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
219         .h_0_new(h_0_new), .h_1_new(h_1_new_x),
220         .done(done_uph_x)
221     );
222     UpdateFluxes XUpHU (.clk(clk), .start(start_upXFlux), .half(update_half),
223         .h_0 (hu[3]), .h_1(hu[1]),
224         .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
225         .h_0_new(hu_0_new), .h_1_new(hu_1_new_x),
226         .done(done_uphu_x)
227     );
228     UpdateFluxes XUpHV (.clk(clk), .start(start_upXFlux), .half(update_half),
229         .h_0 (hv[3]), .h_1(hv[1]),
230         .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
231         .h_0_new(hv_0_new), .h_1_new(hv_1_new_x),
232         .done(done_uphv_x)
233     );
234
235     UpdateFluxes YUpH (.clk(clk), .start(start_upYFlux), .half(update_half),
236         .h_0 (h[3]), .h_1(h[5]),
237         .dt(dt_in), .dx(dx_in), .f_h(Fh_y),
238         .h_0_new(foobar), .h_1_new(h_1_new_y),
239         .done(done_uph_y)
240     );
241     UpdateFluxes YUpHU (.clk(clk), .start(start_upYFlux), .half(update_half),
242         .h_0 (hu[3]), .h_1(hu[5]),
243         .dt(dt_in), .dx(dx_in), .f_h(Fhu_y),
244         .h_0_new(foobar), .h_1_new(hu_1_new_y),
245         .done(done_uphu_y)
246     );
247     UpdateFluxes YUpHV (.clk(clk), .start(start_upYFlux), .half(update_half),
248         .h_0 (h[3]), .h_1(hv[5]),
249         .dt(dt_in), .dx(dx_in), .f_h(Fhv_y),
250         .h_0_new(foobar), .h_1_new(hv_1_new_y),
251         .done(done_uphv_y)
252     );
253
254     //FSM for controlling bus
255     always @(*) begin
256         state_c = state;
257         rd_addressH_c = rd_addressH;
258         rd_addressHu_c = rd_addressHu;
259         rd_addressHv_c = rd_addressHv;
260         wr_addr_c = wr_addr;
261         Rwr_en_c = Rwr_en;
262         case(state)
263             BEG: begin//beginning state
264                 if(start == 1'b0) begin //if we are still loading
265                     start_calcXFlux_c = 1'b0;
266                     start_calcYFlux_c = 1'b0;
267                     start_upXFlux_c = 1'b0;
268                     start_upYFlux_c = 1'b0;
269                     done_boundCon_h_c = 1'b0;
270                     done_boundCon_hu_c = 1'b0;
271                     done_boundCon_hv_c = 1'b0;
272                     done_c = 1'b0;
273                     update_half_c = 1'b0;

```

```

274         Rwr_en_c = 1'b0;
275     end
276     if(start == 1'b1) begin //done writing values and can load
277         rd_addressH_c = 1'b0;
278         rd_addressHu_c = 1'b0;
279         rd_addressHv_c = 1'b0;
280         state_c = LOAD;
281     end
282 end
283 LOAD: begin
284     if(rd_addressH < 3'd6 && rd_addressHu < 3'd6 && rd_addressHv < 3'd6)
285         begin //get 7 h values
286             h_c[rd_addressH] = matrixH_dout;
287             hu_c[rd_addressH] = matrixHu_dout;
288             hv_c[rd_addressH] = matrixHv_dout;
289             rd_addressH_c = rd_addressH + 1'b1;
290             rd_addressHu_c = rd_addressHu + 1'b1;
291             rd_addressHv_c = rd_addressHv + 1'b1;
292         end
293         if(rd_addressH == 3'd7 && rd_addressHu == 3'd7 && rd_addressHv == 3'd7)
294             begin //get dt and dx
295                 dt_in_c = matrixH_dout;
296                 dx_in_c = matrixHu_dout;
297                 done_boundCon_h_c = 1'b1;
298                 done_boundCon_hu_c = 1'b1;
299                 done_boundCon_hv_c = 1'b1;
300                 state_c = RUN;
301             end
302         end
303     RUN: begin //run the algorithm
304         // store old variables
305         if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv) begin
306             h_old_c[0] = h[0];
307             h_old_c[1] = h[1];
308             h_old_c[2] = h[2];
309             h_old_c[3] = h[3];
310             h_old_c[4] = h[4];
311             h_old_c[5] = h[5];
312             h_old_c[6] = h[6];
313
314             hu_old_c[0] = hu[0];
315             hu_old_c[1] = hu[1];
316             hu_old_c[2] = hu[2];
317             hu_old_c[3] = hu[3];
318             hu_old_c[4] = hu[4];
319             hu_old_c[5] = hu[5];
320             hu_old_c[6] = hu[6];
321
322             hv_old_c[0] = hv[0];
323             hv_old_c[1] = hv[1];
324             hv_old_c[2] = hv[2];
325             hv_old_c[3] = hv[3];
326             hv_old_c[4] = hv[4];
327             hv_old_c[5] = hv[5];
328             hv_old_c[6] = hv[6];
329             // calculate fluxes (predictor step)
330             start_calcXFlux_c = 1'b1;
331             start_calcYFlux_c = 1'b1;
332             done_boundCon_h_c = 1'b0;
333             done_boundCon_hu_c = 1'b0;
334             done_boundCon_hv_c = 1'b0;
335         end
336
337         // update half a timestep
338         if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
339             done_fhv_y) begin
340             start_calcXFlux_c = 1'b0;
341             start_calcYFlux_c = 1'b0;
342             update_half_c = 1'b1;

```

```

340         start_upXFlux_c = 1'b1;
341         start_upYFlux_c = 1'b1;
342     end
343
344     // apply boundary conditions
345     if (done_uph_x && done_uphu_x && done_uphv_x && done_uph_y &&
done_uphu_y && done_uphv_y) begin
346         start_upXFlux_c = 0;
347         start_upYFlux_c = 0;
348         h_c[1] = h_1_new_x;
349         h_c[3] = h_0_new;
350         h_c[5] = h_1_new_y;
351         hu_c[1] = hu_1_new_x;
352         hu_c[3] = hu_0_new;
353         hu_c[5] = hu_1_new_y;
354         hv_c[1] = hv_1_new_x;
355         hv_c[3] = hv_0_new;
356         hv_c[5] = hv_1_new_y;
357         done_boundCon_h_c = 1'b1;
358         done_boundCon_hu_c = 1'b1;
359         done_boundCon_hv_c = 1'b1;
360     end
361
362     // calculate new fluxes
363     if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv &&
update_half) begin
364         start_calcXFlux_c = 1'b1;
365         start_calcYFlux_c = 1'b1;
366     end
367
368     // update a full timestep
369     if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
done_fhv_y && update_half) begin
370         start_calcXFlux_c = 1'b0;
371         start_calcYFlux_c = 1'b0;
372         h_c[0] = h_old[0];
373         h_c[1] = h_old[1];
374         h_c[2] = h_old[2];
375         h_c[3] = h_old[3];
376         h_c[4] = h_old[4];
377         h_c[5] = h_old[5];
378         h_c[6] = h_old[6];
379
380         hu_c[0] = hu_old[0];
381         hu_c[1] = hu_old[1];
382         hu_c[2] = hu_old[2];
383         hu_c[3] = hu_old[3];
384         hu_c[4] = hu_old[4];
385         hu_c[5] = hu_old[5];
386         hu_c[6] = hu_old[6];
387
388         hv_c[0] = hv_old[0];
389         hv_c[1] = hv_old[1];
390         hv_c[2] = hv_old[2];
391         hv_c[3] = hv_old[3];
392         hv_c[4] = hv_old[4];
393         hv_c[5] = hv_old[5];
394         hv_c[6] = hv_old[6];
395         update_half_c = 1'b0;
396         start_upXFlux_c = 1'b1;
397         start_upYFlux_c = 1'b1;
398     end
399
400     if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv &&
!update_half) begin
401         start_upXFlux_c = 1'b0;
402         start_upYFlux_c = 1'b0;
403         state_c = DON;//change to finish and load final values
404         wr_addr_c = 1'b0;

```

```

405         Rwr_en_c = 1'b1;
406         fluxData_c = h_0_new;
407     end
408 end
409 DON: begin
410     if(start == 1'b1) begin
411         Rwr_en_c = 1'b1;
412         wr_addr_c = wr_addr + 1'b1;
413         case(wr_addr)
414             1'd1: begin
415                 fluxData_c = h_1_new_x;
416             end
417             2'd2: begin
418                 fluxData_c = hu_0_new;
419             end
420             2'd3: begin
421                 fluxData_c = hu_1_new_x;
422             end
423             3'd4: begin
424                 fluxData_c = hv_0_new;
425             end
426             3'd5: begin
427                 fluxData_c =
428                     hv_1_new_x;
429             end
430             3'd6: begin
431                 fluxData_c = h_1_new_y;
432             end
433             3'd7: begin
434                 fluxData_c = hu_1_new_y;
435             end
436             4'd8: begin
437                 fluxData_c = hv_1_new_y;
438                 done_c = 1'b1;
439             end
440         endcase
441     end
442     if(start == 1'b0) begin
443         Rwr_en_c = 1'b0;
444         done_c = 1'b0;
445         state_c = BEG;//back to beginning
446     end
447 endcase
448 end
449
450 //for reading values
451 always @(slave_address or matrixH_dout or matrixHu_dout or matrixHv_dout or
matrixResult_dout or done) begin
452     case(slave_address[6:4])
453         `MATRIX_H_ADDRESS_OFFSET: slave_readdata = matrixH_dout;
454         `MATRIX_HU_ADDRESS_OFFSET: slave_readdata = matrixHu_dout;
455         `MATRIX_HV_ADDRESS_OFFSET: slave_readdata = matrixHv_dout;
456         `MATRIX_RESULT_ADDRESS_OFFSET: slave_readdata = matrixResult_dout;
457         `CONTROL_OFFSET: slave_readdata = {16'b101_1010_1010_1010, done};
458     endcase // case (slave_address[9:8])
459 end
460
461 always @(posedge clk) begin
462     state <= state_c;
463     wr_addr <= wr_addr_c;
464     rd_addressH <= rd_addressH_c;
465     rd_addressHu <= rd_addressHu_c;
466     rd_addressHv <= rd_addressHv_c;
467     done_boundCon_h <= done_boundCon_h_c;
468     done_boundCon_hu <= done_boundCon_hu_c;
469     done_boundCon_hv <= done_boundCon_hv_c;
470     done <= done_c;
471     Rwr_en <= Rwr_en_c;

```

```

472     fluxData <= fluxData_c;
473     h[0] <= h_c[0];
474     h[1] <= h_c[1];
475     h[2] <= h_c[2];
476     h[3] <= h_c[3];
477     h[4] <= h_c[4];
478     h[5] <= h_c[5];
479     h[6] <= h_c[6];
480     hu[0] <= hu_c[0];
481     hu[1] <= hu_c[1];
482     hu[2] <= hu_c[2];
483     hu[3] <= hu_c[3];
484     hu[4] <= hu_c[4];
485     hu[5] <= hu_c[5];
486     hu[6] <= hu_c[6];
487     hv[0] <= hv_c[0];
488     hv[1] <= hv_c[1];
489     hv[2] <= hv_c[2];
490     hv[3] <= hv_c[3];
491     hv[4] <= hv_c[4];
492     hv[5] <= hv_c[5];
493     hv[6] <= hv_c[6];
494     h_old[0] <= h_old_c[0];
495     h_old[1] <= h_old_c[1];
496     h_old[2] <= h_old_c[2];
497     h_old[3] <= h_old_c[3];
498     h_old[4] <= h_old_c[4];
499     h_old[5] <= h_old_c[5];
500     h_old[6] <= h_old_c[6];
501     hu_old[0] <= hu_old_c[0];
502     hu_old[1] <= hu_old_c[1];
503     hu_old[2] <= hu_old_c[2];
504     hu_old[3] <= hu_old_c[3];
505     hu_old[4] <= hu_old_c[4];
506     hu_old[5] <= hu_old_c[5];
507     hu_old[6] <= hu_old_c[6];
508     hv_old[0] <= hv_old_c[0];
509     hv_old[1] <= hv_old_c[1];
510     hv_old[2] <= hv_old_c[2];
511     hv_old[3] <= hv_old_c[3];
512     hv_old[4] <= hv_old_c[4];
513     hv_old[5] <= hv_old_c[5];
514     hv_old[6] <= hv_old_c[6];
515     start_calcXFlux <= start_calcXFlux_c;
516     start_calcYFlux <= start_calcYFlux_c;
517     start_upXFlux <= start_upXFlux_c;
518     start_upYFlux <= start_upYFlux_c;
519     update_half <= update_half_c;
520     if ((slave_write == 1) && (slave_address[6:4] == `CONTROL_OFFSET)) begin
521         case (slave_writedata[1])
522             1'b0: begin
523                 start <= 1'b0;
524             end
525
526             1'b1: begin
527                 start <= 1'b1;
528             end
529         endcase // case (slave_address[1:0])
530     end
531 end
532
533 endmodule

```

```

1 // megafunction wizard: %RAM: 2-PORT%
2 // GENERATION: STANDARD
3 // VERSION: WM1.0
4 // MODULE: altsyncram
5
6 // =====
7 // File Name: newhRam.v
8 // Megafunction Name(s):
9 //     altsyncram
10 //
11 // Simulation Library Files(s):
12 //     altera_mf
13 // =====
14 // *****
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 17.0.0 Build 595 04/25/2017 SJ Lite Edition
18 // *****
19
20
21 //Copyright (C) 2017 Intel Corporation. All rights reserved.
22 //Your use of Intel Corporation's design tools, logic functions
23 //and other software and tools, and its AMPP partner logic
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Intel Program License
28 //Subscription Agreement, the Intel Quartus Prime License Agreement,
29 //the Intel MegaCore Function License Agreement, or other
30 //applicable license agreement, including, without limitation,
31 //that your use is for the sole purpose of programming logic
32 //devices manufactured by Intel and sold by Intel or its
33 //authorized distributors. Please refer to the applicable
34 //agreement for further details.
35
36
37 // synopsys translate_off
38 `timescale 1 ps / 1 ps
39 // synopsys translate_on
40 module newhRam (
41     clock,
42     data,
43     rdaddress,
44     wraddress,
45     wren,
46     q);
47
48     input    clock;
49     input    [15:0] data;
50     input    [3:0] rdaddress;
51     input    [3:0] wraddress;
52     input    wren;
53     output   [15:0] q;
54 `ifndef ALTERA_RESERVED_QIS
55 // synopsys translate_off
56 `endif
57     tri1    clock;
58     tri0    wren;
59 `ifndef ALTERA_RESERVED_QIS
60 // synopsys translate_on
61 `endif
62
63     wire [15:0] sub_wire0;
64     wire [15:0] q = sub_wire0[15:0];
65
66     altsyncram altsyncram_component (
67         .address_a (wraddress),
68         .address_b (rdaddress),
69         .clock0 (clock),

```



```

70         .data_a (data),
71         .wren_a (wren),
72         .q_b (sub_wire0),
73         .aclr0 (1'b0),
74         .aclr1 (1'b0),
75         .addressstall_a (1'b0),
76         .addressstall_b (1'b0),
77         .byteena_a (1'b1),
78         .byteena_b (1'b1),
79         .clock1 (1'b1),
80         .clocken0 (1'b1),
81         .clocken1 (1'b1),
82         .clocken2 (1'b1),
83         .clocken3 (1'b1),
84         .data_b ({16{1'b1}}),
85         .eccstatus (),
86         .q_a (),
87         .rden_a (1'b1),
88         .rden_b (1'b1),
89         .wren_b (1'b0));
90
91 defparam
92     altsyncram_component.address_aclr_b = "NONE",
93     altsyncram_component.address_reg_b = "CLOCK0",
94     altsyncram_component.clock_enable_input_a = "BYPASS",
95     altsyncram_component.clock_enable_input_b = "BYPASS",
96     altsyncram_component.clock_enable_output_b = "BYPASS",
97     altsyncram_component.intended_device_family = "Cyclone V",
98     altsyncram_component.lpm_type = "altsyncram",
99     altsyncram_component.numwords_a = 16,
100    altsyncram_component.numwords_b = 16,
101    altsyncram_component.operation_mode = "DUAL_PORT",
102    altsyncram_component.outdata_aclr_b = "NONE",
103    altsyncram_component.outdata_reg_b = "CLOCK0",
104    altsyncram_component.power_up_uninitialized = "FALSE",
105    altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
106    altsyncram_component.widthad_a = 4,
107    altsyncram_component.widthad_b = 4,
108    altsyncram_component.width_a = 16,
109    altsyncram_component.width_b = 16,
110    altsyncram_component.width_byteena_a = 1;
111
112 endmodule
113
114 // =====
115 // CNX file retrieval info
116 // =====
117 // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
118 // Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
119 // Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
120 // Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
121 // Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
122 // Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
123 // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
124 // Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
125 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
126 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
127 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
128 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
129 // Retrieval info: PRIVATE: CLRdata NUMERIC "0"
130 // Retrieval info: PRIVATE: CLRq NUMERIC "0"
131 // Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
132 // Retrieval info: PRIVATE: CLRrren NUMERIC "0"
133 // Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
134 // Retrieval info: PRIVATE: CLRwren NUMERIC "0"
135 // Retrieval info: PRIVATE: Clock NUMERIC "0"
136 // Retrieval info: PRIVATE: Clock_A NUMERIC "0"
137 // Retrieval info: PRIVATE: Clock_B NUMERIC "0"
138 // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"

```

```

139 // Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
140 // Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "0"
141 // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_B"
142 // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
143 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
144 // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
145 // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
146 // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
147 // Retrieval info: PRIVATE: MEMSIZE NUMERIC "256"
148 // Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
149 // Retrieval info: PRIVATE: MIFfilename STRING ""
150 // Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "2"
151 // Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
152 // Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
153 // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
154 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "2"
155 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
156 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
157 // Retrieval info: PRIVATE: REGdata NUMERIC "1"
158 // Retrieval info: PRIVATE: REGq NUMERIC "1"
159 // Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
160 // Retrieval info: PRIVATE: REGrren NUMERIC "1"
161 // Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
162 // Retrieval info: PRIVATE: REGwren NUMERIC "1"
163 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
164 // Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
165 // Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
166 // Retrieval info: PRIVATE: VarWidth NUMERIC "0"
167 // Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "16"
168 // Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "16"
169 // Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "16"
170 // Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "16"
171 // Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
172 // Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "0"
173 // Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
174 // Retrieval info: PRIVATE: enable NUMERIC "0"
175 // Retrieval info: PRIVATE: rden NUMERIC "0"
176 // Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
177 // Retrieval info: CONSTANT: ADDRESS_ACLR_B STRING "NONE"
178 // Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
179 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
180 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
181 // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
182 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
183 // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
184 // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16"
185 // Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "16"
186 // Retrieval info: CONSTANT: OPERATION_MODE STRING "DUAL_PORT"
187 // Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
188 // Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK0"
189 // Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
190 // Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS STRING "DONT_CARE"
191 // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "4"
192 // Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "4"
193 // Retrieval info: CONSTANT: WIDTH_A NUMERIC "16"
194 // Retrieval info: CONSTANT: WIDTH_B NUMERIC "16"
195 // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
196 // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
197 // Retrieval info: USED_PORT: data 0 0 16 0 INPUT NODEFVAL "data[15..0]"
198 // Retrieval info: USED_PORT: q 0 0 16 0 OUTPUT NODEFVAL "q[15..0]"
199 // Retrieval info: USED_PORT: rdaddress 0 0 4 0 INPUT NODEFVAL "rdaddress[3..0]"
200 // Retrieval info: USED_PORT: wraddress 0 0 4 0 INPUT NODEFVAL "wraddress[3..0]"
201 // Retrieval info: USED_PORT: wren 0 0 0 0 INPUT GND "wren"
202 // Retrieval info: CONNECT: @address_a 0 0 4 0 wraddress 0 0 4 0
203 // Retrieval info: CONNECT: @address_b 0 0 4 0 rdaddress 0 0 4 0
204 // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
205 // Retrieval info: CONNECT: @data_a 0 0 16 0 data 0 0 16 0
206 // Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
207 // Retrieval info: CONNECT: q 0 0 16 0 @q_b 0 0 16 0

```

```
208 // Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.v TRUE
209 // Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.inc FALSE
210 // Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.cmp FALSE
211 // Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.bsf FALSE
212 // Retrieval info: GEN_FILE: TYPE_NORMAL newhRam_inst.v FALSE
213 // Retrieval info: GEN_FILE: TYPE_NORMAL newhRam_bb.v TRUE
214 // Retrieval info: LIB_FILE: altera_mf
215
```

```

1  /*
2  * singlePoint.v - the top level module to get a proof the hw works in serial
3  * Brian Kuhn, Brian Dang and Kyle Heien - Shallow Water Simulation - EEC 181
4  * May 2019
5  */
6
7  `define H 2'b00
8  `define HU 2'b01
9  `define HV 2'b10
10
11  `define X 1'b0
12  `define Y 1'b1
13
14  // constants and computation parameters
15  `define HALO 2'd2
16  `define GRAVITY 13'b0010011100111// 9.80616 in decimal Q6.7 format - 13'b001001_1100111
17
18  `define CONTROL_OFFSET 2'b00
19  `define MATRIX_A_ADDRESS_OFFSET 2'b01
20  `define MATRIX_B_ADDRESS_OFFSET 2'b10
21  `define MATRIX_RESULT_ADDRESS_OFFSET 2'b11
22
23  module singlePoint #( () (
24      // signals to connect to an Avalon clock source interface
25      clk,
26      reset,
27
28      // signals to connect to an Avalon-MM slave interface
29      slave_address,
30      slave_read,
31      slave_write,
32      slave_readdata,
33      slave_writedata,
34      slave_byteenable
35  );
36
37      parameter DATA_WIDTH = 16;
38
39      input clk,
40      reg start,
41      input reset,
42
43      // slave interface
44      input [9:0] slave_address;
45      input slave_read;
46      input slave_write;
47      output reg [DATA_WIDTH-1:0] slave_readdata;
48      input [DATA_WIDTH-1:0] slave_writedata;
49      input [(DATA_WIDTH/8)-1:0] slave_byteenable;
50
51      input [12:0] h_0_in, // (hio, jo)
52      input [12:0] h_1_in_x, // (i0-1)
53      input [12:0] h_2_in_x, // (io-2)
54      input [12:0] h_f_in_x, // (io+1)
55      input [12:0] h_1_in_y,
56      input [12:0] h_2_in_y,
57      input [12:0] h_f_in_y,
58      input [12:0] hu_0_in,
59      input [12:0] hu_1_in_x,
60      input [12:0] hu_2_in_x,
61      input [12:0] hu_f_in_x,
62      input [12:0] hu_1_in_y,
63      input [12:0] hu_2_in_y,
64      input [12:0] hu_f_in_y,
65      input [12:0] hv_0_in,
66      input [12:0] hv_1_in_x,
67      input [12:0] hv_2_in_x,
68      input [12:0] hv_f_in_x,
69      input [12:0] hv_1_in_y,

```

```

70     input [12:0] hv_2_in_y,
71     input [12:0] hv_f_in_y,
72     input [7:0] num_gridpoints, // max 256
73     input [9:0] num_timesteps, // max 1024
74     input [31:0] t_final_in,
75     input [12:0] dt_in,
76     input [12:0] dx_in,
77     output reg done,
78     output reg signed [12:0] test
79
80     // data matrices - MIGHT NEED TO GET REWORKED WITH BUS STUFF
81     reg signed [12:0] h [6:0]; // i-2, i-1, i+1, 0, j-2, j-2, j+1
82     reg signed [12:0] hu [6:0];
83     reg signed [12:0] hv [6:0];
84     reg signed [12:0] h_old [6:0];
85     reg signed [12:0] hu_old [6:0];
86     reg signed [12:0] hv_old [6:0];
87
88     // intermediate data
89     wire signed [25:0] h_flux_left, h_flux_right, hu_flux_left, hu_flux_right,
90     hv_flux_left, hv_flux_right;
91     wire signed [12:0] h_left, h_right, hu_left, hu_right, hv_left, hv_right;
92     wire signed [12:0] Fh_x, Fhu_x, Fhv_x, Fh_y, Fhu_y, Fhv_y;
93
94     // all the start control signals
95     reg start_boundCon;
96     reg start_calcXFlux, start_calcYFlux;
97     reg start_upXFlux, start_upYFlux;
98
99     // all the done control signals
100     wire done_boundCon_h, done_boundCon_hu, done_boundCon_hv;
101     wire done_flux_left;
102     wire done_fh_x, done_fhu_x, done_fhv_x, done_fh_y, done_fhu_y, done_fhv_y;
103     wire done_uph_x, done_uphu_x, done_uphv_x, done_uph_y, done_uphu_y, done_uphv_y;
104
105     // other control signals
106     reg update_half; // for UpdateFluxes | 0 = whole, 1 = half
107
108     // buffer signals
109     wire signed [12:0] foobar, h_0_new, h_1_new_x, h_1_new_y, hu_0_new, hu_1_new_x,
110     hu_1_new_y, hv_0_new, hv_1_new_x, hv_1_new_y;
111
112     reg [9:0] timestep;
113     reg [31:0] t_final, t_current;
114     reg [31:0] T [NT-1:0];
115     reg [12:0] dt; // not sure what the bounds are for this
116
117     // MIGHT NEED TO GET REWORKED WITH BUS STUFF
118     /*ApplyBoundaryConditions h_bound (.input_index(`H), .n_res(num_gridpoints),
119     .halo(`HALO), .start(start_boundCon), .clk(clk), .done(done_boundCon_h));
120     ApplyBoundaryConditions hu_bound (.input_index(`HU), .n_res(num_gridpoints),
121     .halo(`HALO), .start(start_boundCon), .clk(clk), .done(done_boundCon_hu));
122     ApplyBoundaryConditions hv_bound (.input_index(`HV), .n_res(num_gridpoints),
123     .halo(`HALO), .start(start_boundCon), .clk(clk), .done(done_boundCon_hv));
124     */
125     // MIGHT NEED TO GET REWORKED WITH BUS STUFF - not sure where the inputs should be
126     // coming from
127     // calculate fluxea
128
129     hRAM    matrixH(
130         .clock ( clk),
131         .data ( slave_writedata ),
132         .rdaddress (rd_addrA),
133         .wraddress (slave_address[3:0]),
134         .wren (matrixA_wren),
135         .q (matrixA_dout)
136     );
137
138     hRAM    matrixHu(

```

```

133         .clock ( clk),
134         .data ( slave_writedata ),
135         .rdaddress (rd_addrB),
136         .waddress (slave_address[3:0]),
137         .wren (matrixB_wren),
138         .q (matrixB_dout)
139     );
140
141     hRAM    matrixHv(
142         .clock ( clk),
143         .data ( slave_writedata ),
144         .rdaddress (rd_addrC),
145         .waddress (slave_address[3:0]),
146         .wren (matrixB_wren),
147         .q (matrixB_dout)
148     );
149
150     hRAM    matrixResult(
151         .clock ( clk),
152         .data (vectorSum),
153         .rdaddress (slave_address[3:0]),
154         .waddress (wr_addr4_5),
155         .wren (wr_en4_5),
156         .q (matrixResult_dout)
157     );
158
159     CalcFluxes XFluxH (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
160         .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
161         .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
162         .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
163         .halo(`HALO), .gravity(`GRAVITY),
164         .F(Fh_x), .done(done_fh_x)
165     );
166     CalcFluxes XFluxHU (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
167         .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
168         .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
169         .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
170         .halo(`HALO), .gravity(`GRAVITY),
171         .F(Fhu_x), .done(done_fhu_x)
172     );
173     CalcFluxes XFluxHV (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
174         .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
175         .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
176         .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
177         .halo(`HALO), .gravity(`GRAVITY),
178         .F(Fhv_x), .done(done_fhv_x)
179     );
180
181     CalcFluxes YFluxH (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
182         .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[7]),
183         .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[7]),
184         .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[7]),
185         .halo(`HALO), .gravity(`GRAVITY),
186         .F(Fh_y), .done(done_fh_y)
187     );
188     CalcFluxes YFluxHU (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
189         .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[7]),
190         .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[7]),
191         .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[7]),
192         .halo(`HALO), .gravity(`GRAVITY),
193         .F(Fhu_y), .done(done_fhu_y)
194     );
195     CalcFluxes YFluxHV (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
196         .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[7]),
197         .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[7]),
198         .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[7]),
199         .halo(`HALO), .gravity(`GRAVITY),
200         .F(Fhv_y), .done(done_fhv_y)
201     );

```

```

202
203 // update fluxes
204 UpdateFluxes XUpH (.clk(clk), .start(start_upXFlux), .half(update_half),
205     .h_0 (h[3]), .h_1(h[1]),
206     .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
207     .h_0_new(h_0_new), .h_1_new(h_1_new_x),
208     .done(done_uph_x)
209 );
210 UpdateFluxes XUpHU (.clk(clk), .start(start_upXFlux), .half(update_half),
211     .h_0 (hu[3]), .h_1(hu[1]),
212     .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
213     .h_0_new(hu_0_new), .h_1_new(hu_1_new_x),
214     .done(done_uphu_x)
215 );
216 UpdateFluxes XUpHV (.clk(clk), .start(start_upXFlux), .half(update_half),
217     .h_0 (hv[3]), .h_1(hv[1]),
218     .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
219     .h_0_new(hv_0_new), .h_1_new(hv_1_new_x),
220     .done(done_uphv_x)
221 );
222
223 UpdateFluxes YUpH (.clk(clk), .start(start_upYFlux), .half(update_half),
224     .h_0 (h[3]), .h_1(h[5]),
225     .dt(dt_in), .dx(dx_in), .f_h(Fh_y),
226     .h_0_new(foobar), .h_1_new(h_1_new_y),
227     .done(done_uph_y)
228 );
229 UpdateFluxes YUpHU (.clk(clk), .start(start_upYFlux), .half(update_half),
230     .h_0 (hu[3]), .h_1(hu[5]),
231     .dt(dt_in), .dx(dx_in), .f_h(Fhu_y),
232     .h_0_new(foobar), .h_1_new(hu_1_new_y),
233     .done(done_uphu_y)
234 );
235 UpdateFluxes YUpHV (.clk(clk), .start(start_upYFlux), .half(update_half),
236     .h_0 (h[3]), .h_1(hv[5]),
237     .dt(dt_in), .dx(dx_in), .f_h(Fhv_y),
238     .h_0_new(foobar), .h_1_new(hv_1_new_y),
239     .done(done_uphv_y)
240 );
241 assign done_boundCon_h=start;
242 assign done_boundCon_hu=start;
243 assign done_boundCon_hv=start;
244 initial begin
245     // initialize start signals
246     start_boundCon = 0;
247     start_calcXFlux = 0;
248     start_calcYFlux = 0;
249     start_upXFlux = 0;
250     start_upYFlux = 0;
251
252     // set done to 0
253     done = 0;
254
255     // initialize data
256     h[0] = h_2_in_x;
257     h[1] = h_1_in_x;
258     h[2] = h_f_in_x;
259     h[3] = h_0_in;
260     h[4] = h_2_in_y;
261     h[5] = h_1_in_y;
262     h[6] = h_f_in_y;
263
264     hu[0] = hu_2_in_x;
265     hu[1] = hu_1_in_x;
266     hu[2] = hu_f_in_x;
267     hu[3] = hu_0_in;
268     hu[4] = hu_2_in_y;
269     hu[5] = hu_1_in_y;
270     hu[6] = hu_f_in_y;

```

```

271
272         hv[0] = hv_2_in_x;
273         hv[1] = hv_1_in_x;
274         hv[2] = hv_f_in_x;
275         hv[3] = hv_0_in;
276         hv[4] = hv_2_in_y;
277         hv[5] = hv_1_in_y;
278         hv[6] = hv_f_in_y;
279     end
280
281     // run the computation - MIGHT NEED TO GET REWORKED WITH BUS STUFF
282     always @(posedge clk) begin
283         // apply boundary conditions
284         start_boundCon = 1;
285         test=h[0];
286
287         if (start==0)begin
288             start_boundCon = 0;
289             start_calcXFlux = 0;
290             start_calcYFlux = 0;
291             start_upXFlux = 0;
292             start_upYFlux = 0;
293
294             // set done to 0
295             done = 0;
296
297             // initialize data
298             h[0] = h_2_in_x;
299             h[1] = h_1_in_x;
300             h[2] = h_f_in_x;
301             h[3] = h_0_in;
302             h[4] = h_2_in_y;
303             h[5] = h_1_in_y;
304             h[6] = h_f_in_y;
305
306             hu[0] = hu_2_in_x;
307             hu[1] = hu_1_in_x;
308             hu[2] = hu_f_in_x;
309             hu[3] = hu_0_in;
310             hu[4] = hu_2_in_y;
311             hu[5] = hu_1_in_y;
312             hu[6] = hu_f_in_y;
313
314             hv[0] = hv_2_in_x;
315             hv[1] = hv_1_in_x;
316             hv[2] = hv_f_in_x;
317             hv[3] = hv_0_in;
318             hv[4] = hv_2_in_y;
319             hv[5] = hv_1_in_y;
320             hv[6] = hv_f_in_y;
321         end
322
323
324         // store old variables
325         if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv) begin
326             start_boundCon = 0;
327             h_old[0] = h[0];
328             h_old[1] = h[1];
329             h_old[2] = h[2];
330             h_old[3] = h[3];
331             h_old[4] = h[4];
332             h_old[5] = h[5];
333             h_old[6] = h[6];
334
335             hu_old[0] = hu[0];
336             hu_old[1] = hu[1];
337             hu_old[2] = hu[2];
338             hu_old[3] = hu[3];
339             hu_old[4] = hu[4];

```



```

340         hu_old[5] = hu[5];
341         hu_old[6] = hu[6];
342
343         hv_old[0] = hv[0];
344         hv_old[1] = hv[1];
345         hv_old[2] = hv[2];
346         hv_old[3] = hv[3];
347         hv_old[4] = hv[4];
348         hv_old[5] = hv[5];
349         hv_old[6] = hv[6];
350         // calculate fluxes (predictor step)
351         start_calcXFlux = 1;
352         start_calcYFlux = 1;
353     end
354
355     // update half a timestep
356     if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
done_fhv_y) begin
357         start_calcXFlux = 0;
358         start_calcYFlux = 0;
359         update_half = 1;
360         start_upXFlux = 1;
361         start_upYFlux = 1;
362     end
363
364     // apply boundary conditions
365     if (done_uph_x && done_uphu_x && done_uphv_x && done_uph_y && done_uphu_y &&
done_uphv_y) begin
366         start_upXFlux = 0;
367         start_upYFlux = 0;
368         h[1] = h_1_new_x;
369         h[3] = h_0_new;
370         h[5] = h_1_new_y;
371         hu[1] = hu_1_new_x;
372         hu[3] = hu_0_new;
373         hu[5] = hu_1_new_y;
374         hv[1] = hv_1_new_x;
375         hv[3] = hv_0_new;
376         hv[5] = hv_1_new_y;
377         start_boundCon = 1;
378     end
379
380     // calculate new fluxes
381     if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv && update_half) begin
382         start_calcXFlux = 1;
383         start_calcYFlux = 1;
384     end
385
386     // update a full timestep
387     if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
done_fhv_y && update_half) begin
388         start_calcXFlux = 0;
389         start_calcYFlux = 0;
390         h[0] = h_old[0];
391         h[1] = h_old[1];
392         h[2] = h_old[2];
393         h[3] = h_old[3];
394         h[4] = h_old[4];
395         h[5] = h_old[5];
396         h[6] = h_old[6];
397
398         hu[0] = hu_old[0];
399         hu[1] = hu_old[1];
400         hu[2] = hu_old[2];
401         hu[3] = hu_old[3];
402         hu[4] = hu_old[4];
403         hu[5] = hu_old[5];
404         hu[6] = hu_old[6];
405

```

```

406         hv[0] = hv_old[0];
407         hv[1] = hv_old[1];
408         hv[2] = hv_old[2];
409         hv[3] = hv_old[3];
410         hv[4] = hv_old[4];
411         hv[5] = hv_old[5];
412         hv[6] = hv_old[6];
413         update_half = 0;
414         start_upXFlux = 1;
415         start_upYFlux = 1;
416     end
417
418     if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv && !update_half)
419         begin
420             done = 1;
421         end
422     end
423     /*always @(h_old or hu_old or hv_old) begin
424         // calculate X fluxes (predictor step)
425         start_calcXFlux = 1;
426         start_calcYFlux = 1;
427     end*/
428
429 endmodule

```

```

1  /*
2  * CalcFluxes.v - calculates the flux of a point based off its current, h, U and V values
3  * Kyle Heien and Brian Kuhn - Shallow Water Simulations - EEC 181
4  * April 2019
5  */
6
7  module CalcFluxes(clk, gravity, start, xOrY, h_0, h_1, h_2, h_f, hu_0, hu_1, hu_2,
8  hu_f, hv_0, hv_1, hv_2, hv_f, halo, F, done);
9  input signed [12:0] h_0, h_1, h_2, h_f, hu_0, hu_1, hu_2, hu_f, hv_0, hv_1, hv_2, hv_f;
10 input signed [12:0] gravity;
11 input [1:0] halo;
12 input signed start;
13 input clk;
14 input xOrY; // 0 for x and 1 for y
15 output reg done;
16 output reg signed [12:0] F;
17
18 initial done =0;
19
20 wire signed [12:0] h_left, h_right, hu_left, hv_right; // might need to be 14 bits
21 wire signed [12:0] max_wave_speed;
22
23 wire signed [12:0] hu_right, hv_left;
24 reg signed [12:0] hx_left, hx_right;
25
26 wire done_l, done_r, done_mws;
27
28 wire signed [25:0] h_flux_left, h_flux_right, hu_flux_left, hu_flux_right,
29 hv_flux_left, hv_flux_right;
30
31 sideStates hLeft (.h_0(h_0), .h_1(h_1), .h_2(h_2), .sideState(h_left));
32 sideStates huLeft (.h_0(hu_0), .h_1(hu_1), .h_2(hu_2), .sideState(hu_left));
33 sideStates hvLeft (.h_0(hv_0), .h_1(hv_1), .h_2(hv_2), .sideState(hv_left));
34
35 sideStates hRight (.h_0(h_1), .h_1(h_0), .h_2(h_f), .sideState(h_right));
36 sideStates huRight (.h_0(hu_1), .h_1(hu_0), .h_2(hu_f), .sideState(hu_right));
37 sideStates hvRight (.h_0(hv_1), .h_1(hv_0), .h_2(hv_f), .sideState(hv_right));
38
39 CalcSideFluxes flux_left (.start(start), .h_side(h_left), .hu_side(hu_left),
40 .hv_side(hv_left), .clk(clk), .gravity(gravity),
41 .h_flux_side(h_flux_left), .hu_flux_side(hu_flux_left),
42 .hv_flux_side(hv_flux_right), .done(done_l)
43 );
44
45 CalcSideFluxes flux_right (.start(start), .h_side(h_right), .hu_side(hu_right),
46 .hv_side(hv_right), .clk(clk), .gravity(gravity),
47 .h_flux_side(h_flux_right), .hu_flux_side(hu_flux_right),
48 .hv_flux_side(hv_flux_right), .done(done_r)
49 );
50
51 MaxWaveSpeed mws (.clk(clk), .start(done_l && done_r), .gravity(gravity),
52 .h_left(h_left), .h_right(h_right), .hv_left(hx_left), .hv_right(hx_right),
53 .max_wave_speed(max_wave_speed),
54 .done(done_mws)
55 );
56
57 always @(*) begin
58     if (xOrY == 0) begin
59         hx_left = hu_left;
60         hx_right = hu_left;
61     end
62     if (xOrY == 1) begin
63         hx_left = hv_left;
64         hx_right = hv_right;
65     end
66 end
67
68 always @(done_l && done_r && done_mws) begin
69     F=(h_flux_left+h_flux_right)>>1-(h_right-h_left)*max_wave_speed>>1;

```

```
64 end
65
66 always @(*) begin
67     if (done_l==1&&done_mws==1&&done_r==1) begin
68         done =1;
69     end
70     else if(start==0 || done_l==0|| done_mws==0 || done_r==0) begin
71         done=0;
72     end
73
74
75 end
76 endmodule
```

```
1  /*
2  * sideStates.v - calculates h*_left and h*_right, depending on input
3  * Brian Kuhn - Shallow Water Simulations - EEC 181
4  * April 2019
5  */
6
7  // number of bits for input/output
8
9  module sideStates (
10     input signed [12:0] h_0,
11     input signed [12:0] h_1,
12     input signed [12:0] h_2,
13     output reg signed [12:0] sideState // might need to be [N:0] instead
14 );
15
16     wire [2:0] q = 3'b0_01; // q == quarter == 0.25
17
18     always @(*) begin
19         sideState = ((q*h_0) + h_1) - (q*h_2);
20     end
21 endmodule
```

```

1  /*
2  * CalcSideFluxes.v - calculates h*_flux_left and h*_flux_right, depending on input
3  * Brian Kuhn - Shallow Water Simulations - EEC 181
4  * April 2019
5  */
6
7  // number of bits for input/output
8
9  module CalcSideFluxes (
10     input signed [12:0] h_side,
11     input signed [12:0] hu_side,
12     input signed [12:0] hv_side,
13     input clk,
14     input signed [12:0] gravity,
15     input start,
16     output reg signed [25:0] h_flux_side,
17     output reg signed [25:0] hu_flux_side,
18     output reg signed [25:0] hv_flux_side,
19     output reg done
20 );
21
22     initial done =0;
23
24     wire [12:0] qu, qv;
25     wire done_u, done_v;
26
27     divide div_u(
28         .dividend(hu_side), .divisor(h_side),
29         .start(start), .quotient(qu),
30         .complete(done_u),
31         .clk(clk)
32     );
33
34     divide div_v(
35         .dividend(hv_side), .divisor(h_side),
36         .start(start), .quotient(qv),
37         .complete(done_v),
38         .clk(clk)
39     );
40
41     wire [1:0] half = 2'b0_1; // 0.5
42     //wire [10:0] gravity = 13'b001001_1100111 // 9.8046875
43
44     always @(posedge clk) begin
45         h_flux_side = hu_side;
46         hu_flux_side = hu_side*qu + half*gravity*h_side*h_side;
47         hv_flux_side = hu_side*qv;
48     end
49
50     always @(*) begin
51         if (done_v==1&&done_u==1) begin
52             done =1;
53         end
54         else if(start==0) begin
55             done=0;
56         end
57     end
58
59     endmodule
60

```

```

1  /*
2  * MaxWaveSpeed.v - calculates maximum wave speed for a height with velocity
3  * Kyle Heien - Shallow Water Simulations - EEC 181
4  * April 2019
5  */
6
7  module MaxWaveSpeed(clk, hv_left, hv_right, h_left, h_right, gravity,
max_wave_speed, done, start);
8  input signed [12:0] hv_left;
9  input signed [12:0] h_left;
10 input signed [12:0] hv_right;
11 input signed [12:0] h_right;
12 input signed [12:0] gravity;
13 input start;
14 output reg signed [12:0] max_wave_speed;
15 output reg done;
16 input clk;
17
18 wire signed [12:0] quotient_left;
19 wire signed [12:0] quotient_right;
20 wire signed [12:0] sqrt_out;
21 wire done1;
22 wire done2;
23 wire done3;
24 reg signed [12:0] sqrtinput;
25 reg signed [12:0] sqrtinput1;
26 reg signed [12:0] quotient_out;
27 reg signed [12:0] quotient_out1;
28
29 initial done =0;
30 divide lut (
31     .start(start),
32     .quotient(quotient_left),
33     .dividend(hv_left),
34     .divisor(h_left),
35     .clk(clk),
36     .complete(done1)
37 );
38 divide rut (
39     .start(start),
40     .quotient(quotient_right),
41     .dividend(hv_right),
42     .divisor(h_right),
43     .clk(clk),
44     .complete(done2)
45 );
46
47 sqrt sut(
48     .clk(clk),
49     .start(done2&&done1),
50     .data(sqrtinput),
51     .answer(sqrt_out),
52     .done(done3)
53 );
54
55 initial begin
56     done=0;
57 end
58
59
60
61 always @(*)begin
62     sqrtinput1=gravity*(h_left+h_right)>>1;
63     quotient_out1=(quotient_right+quotient_left)>>1;
64     max_wave_speed=quotient_out+sqrt_out;
65     if (sqrtinput1[12]==1) begin
66         sqrtinput=~sqrtinput1+1;
67     end
68     else begin

```

```
69         sqrtinput=sqrtinput1;
70     end
71     if (quotient_out1[12]==1) begin
72         quotient_out=~quotient_out1+1;
73     end
74     else begin
75         quotient_out=quotient_out1;
76     end
77
78 end
79
80
81 always @(*) begin
82     if (done3==1) begin
83         done =1;
84     end
85     else if(start==0) begin
86         done=0;
87     end
88
89 end
90
91
92
93
94
95 endmodule
96
```



```

1  /*
2  * divide.v - based off the qdiv.v file, however converts the inputs and outputs from
3  * 2's compliment to signed magnitude
4  * Kyle Heien - Shallow Water Simulations - EEC 181
5  * April 2019
6  */
7  module divide (
8  input signed [12:0] dividend,
9  input signed [12:0] divisor,
10 input start,
11 input clk,
12 output reg signed [12:0] quotient,
13 output reg complete);
14
15
16 reg [13:0] i_dividend;
17 reg [13:0] i_divisor;
18 wire [13:0] o_quotient_out;
19 wire done;
20 initial complete =0;
21
22 qdiv but (
23     .i_dividend(i_dividend),
24     .i_divisor(i_divisor),
25     .i_start(start),
26     .i_clk(clk),
27     .o_quotient_out(o_quotient_out),
28     .o_complete(done),
29     .o_overflow(o_overflow));
30
31 always @(*) begin
32     if (dividend[12]==1)begin
33         i_dividend[12:0]= ~dividend+1;
34         i_dividend[13]=1;
35     end
36
37     else begin
38         i_dividend=dividend;
39     end
40
41     if (divisor[12]==1)begin
42         i_divisor[12:0]= ~divisor+1;
43         i_divisor[13]=1;
44     end
45
46     else begin
47         i_divisor=divisor;
48     end
49     if (o_quotient_out[13]==1)begin
50         quotient=~o_quotient_out[12]-1;
51     end
52
53     else begin
54         quotient=o_quotient_out;
55     end
56 end
57
58 always @(*)begin
59     if (start==1&&done==0)begin
60         complete=1;
61     end
62     else begin
63         complete=0;
64     end
65
66 end
67 endmodule

```

```

1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////
3  // Company:          Burke
4  // Engineer:         Tom Burke
5  //
6  // Create Date:      19:39:14 08/24/2011
7  // Design Name:
8  // Module Name:      qdiv.v
9  // Project Name:      Fixed-point Math Library (Verilog)
10 // Target Devices:
11 // Tool versions:     Xilinx ISE WebPack v14.7
12 // Description:       Fixed-point division in (Q,N) format
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Revision 0.02 - 25 May 2014
19 //                   Updated to fix an error
20 //
21 // Additional Comments: Based on my description on youtube:
22 //                   http://youtu.be/TEnaPMYiuR8
23 //
24 //////////////////////////////////////////////////
25
26 module qdiv #(
27     //Parameterized values
28     parameter Q = 7,
29     parameter N = 14
30 )
31 (
32     input  [N-1:0] i_dividend,
33     input  [N-1:0] i_divisor,
34     input      i_start,
35     input      i_clk,
36     output [N-1:0] o_quotient_out,
37     output      o_complete,
38     output      o_overflow
39 );
40
41     reg [2*N+Q-3:0] reg_working_quotient; // Our working copy of the quotient
42     reg [N-1:0]      reg_quotient;        // Final quotient
43     reg [N-2+Q:0]    reg_working_dividend; // Working copy of the dividend
44     reg [2*N+Q-3:0] reg_working_divisor;  // Working copy of the divisor
45
46     reg [N-1:0]      reg_count;           // This is obviously a lot bigger than it
47                                           // needs to be, as we only need
48                                           // count to N-1+Q but,
49                                           // computing that number of bits
50                                           // requires a
51                                           // logarithm (base 2), and I
52                                           // don't know how to do that in a
53                                           // way that will work for
54                                           // everyone
55
56     reg      reg_done;                    // Computation completed flag
57     reg      reg_sign;                    // The quotient's sign bit
58     reg      reg_overflow;                // Overflow flag
59
60     initial reg_done = 1'b1;              // Initial state is to not be doing anything
61     initial reg_overflow = 1'b0;          // And there should be no woverflow
62     present
63     initial reg_sign = 1'b0;              // And the sign should be positive
64
65     initial reg_working_quotient = 0;
66     initial reg_quotient = 0;
67     initial reg_working_dividend = 0;
68     initial reg_working_divisor = 0;
69     initial reg_count = 0;

```

```

64
65
66 assign o_quotient_out[N-2:0] = reg_quotient[N-2:0]; // The division results
67 assign o_quotient_out[N-1] = reg_sign; // The sign of the
quotient
68 assign o_complete = reg_done;
69 assign o_overflow = reg_overflow;
70
71 always @( posedge i_clk ) begin
72     if( reg_done && i_start ) begin // This is
our startup condition
73         // Need to check for a divide by zero right here, I think....
74         reg_done <= 1'b0; // We're
not done
75         reg_count <= N+Q-1; // Set the count
76         reg_working_quotient <= 0; // Clear out
the quotient register
77         reg_working_dividend <= 0; // Clear out
the dividend register
78         reg_working_divisor <= 0; // Clear out
the divisor register
79         reg_overflow <= 1'b0; // Clear the
overflow register
80
81         reg_working_dividend[N+Q-2:Q] <= i_dividend[N-2:0]; //
Left-align the dividend in its working register
82         reg_working_divisor[2*N+Q-3:N+Q-1] <= i_divisor[N-2:0]; // Left-align
the divisor into its working register
83
84         reg_sign <= i_dividend[N-1] ^ i_divisor[N-1]; // Set the sign bit
85     end
86     else if(!reg_done) begin
87         reg_working_divisor <= reg_working_divisor >> 1; // Right shift the
divisor (that is, divide it by two - aka reduce the divisor)
88         reg_count <= reg_count - 1; // Decrement the
count
89
90         // If the dividend is greater than the divisor
91         if(reg_working_dividend >= reg_working_divisor) begin
92             reg_working_quotient[reg_count] <=
1'b1; // Set the quotient bit
93             reg_working_dividend <= reg_working_dividend - reg_working_divisor;
// and subtract the divisor from the dividend
94         end
95
96         //stop condition
97         if(reg_count == 0) begin
98             reg_done <= 1'b1; // If we're
done, it's time to tell the calling process
99             reg_quotient <= reg_working_quotient; // Move in our working
copy to the outside world
100             if (reg_working_quotient[2*N+Q-3:N]>0)
101                 reg_overflow <= 1'b1;
102             end
103         else
104             reg_count <= reg_count - 1;
105         end
106     end
107 endmodule

```

```

1  /*
2  * sqrt.v
3  * Brian Kuhn - Shallow Water Simulations - EEC 181
4  * May 2019
5  */
6
7  module sqrt #(parameter DATA_WIDTH = 13, ANSWER_WIDTH = 13, TRIAL_WIDTH = 8) (
8      input clk,
9      input start,
10     input wire [DATA_WIDTH-1:0] data,
11     output reg [ANSWER_WIDTH-1:0] answer,
12     output done
13 );
14
15     reg busy;
16     reg [TRIAL_WIDTH-1:0] bit;
17     wire [TRIAL_WIDTH-1:0] trial;
18     assign trial = answer | (1 << bit);
19
20     always @(posedge clk) begin
21         if (busy) begin
22             if (bit == 0)
23                 busy <= 0;
24             else
25                 bit <= bit - 1;
26                 if (trial*trial <= data)
27                     answer <= trial;
28             end else if (start) begin
29                 busy <= 1;
30                 answer <= 0;
31                 bit <= TRIAL_WIDTH - 1;
32             end
33         end
34
35         assign done = ~busy;
36     endmodule

```

```

1  /*
2  * UpdateFluxes.v - calculates a new value(h,V or U) based of the old value and the flux
3  * Kyle Heien - Shallow Water Simulations - EEC 181
4  * April 2019
5  */
6  module UpdateFluxes(
7  input clk,
8  input start,
9  input half,
10 input signed [12:0] h_0,
11 input signed [12:0] h_1,
12 input signed [7:0] dt,
13 input signed [12:0] dx,
14 input signed [12:0] f_h,
15 output reg signed [12:0] h_0_new,
16 output reg signed [12:0] h_1_new,
17 output reg done);
18 wire signed [12:0] quotient;
19 wire done1;
20 reg signed [12:0] dt1;
21 divide tut (
22     .start(start),
23     .quotient(quotient),
24     .dividend(dt1),
25     .divisor(dx),
26     .complete(done1),
27     .clk(clk)
28 );
29 initial begin
30     done=0;
31     if (half==1) begin
32         dt1=dt>>1;
33     end
34     if (half==0) begin
35         dt1=dt;
36     end
37 end
38
39
40 always @(*) begin
41     h_0_new=h_0+quotient*f_h;
42     h_1_new=h_1-quotient*f_h;
43 end
44
45 always @(*) begin
46     if (start==1 && done1==1) begin
47         done =1;
48     end
49     else begin
50         done=0;
51     end
52
53 end
54 endmodule

```