Designing an Application-Specific Processor to Accelerate the Computation of Discreticized 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[1] in about 24 hours [2]. About 50% of the computation time in the model is occupied by dynamics[2] and tracer transport[3]. 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].

[1]"Low resolution" is defined as a grid size of 110 km by 110 km.
[2]Flow of water air, and other fluids.
[3]Particulate matter such as molecules and pollutants.

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

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 Δx 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 Δt 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].

$$h^*_{i,j} = h^n_{i,j} + \frac{\Delta t}{2} F_h(h^n, U^n, V^n)$$
$$U^*_{i,j} = U^n_{i,j} + \frac{\Delta t}{2} F_u(h^n, U^n, V^n)$$
$$V^*_{i,j} = V^n_{i,j} + \frac{\Delta t}{2} F_v(h^n, U^n, V^n)$$

Fig. 3. Half Time Step Calculation

$$h^{n+1}_{i,j} = h^n_{i,j} + \Delta t F_h(h^*, U^*, V^*)$$
$$U^{n+1}_{i,j} = U^n_{i,j} + \Delta t F_u(h^*, U^*, V^*)$$
$$V^{n+1}_{i,j} = V^n_{i,j} + \Delta t F_v(h^*, U^*, V^*)$$

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

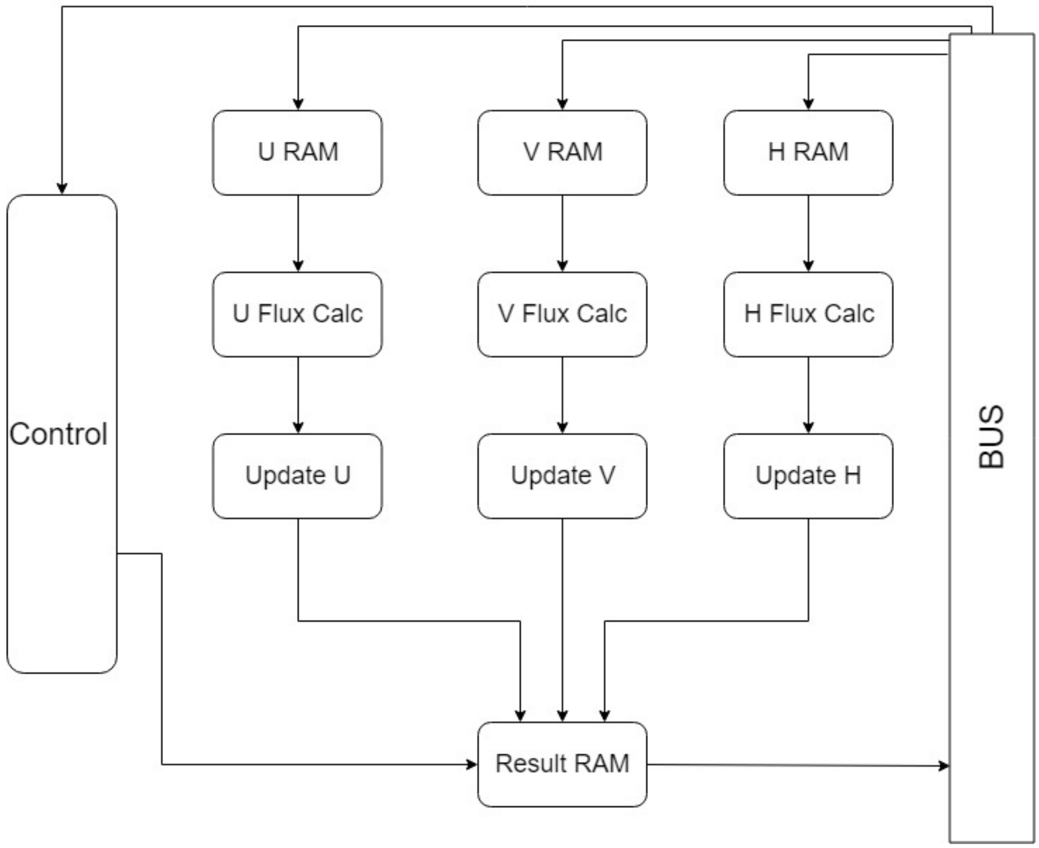

Fig. 5. Whole Time Step Calculation

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 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 %)

0 / 6 (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 compliment, 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

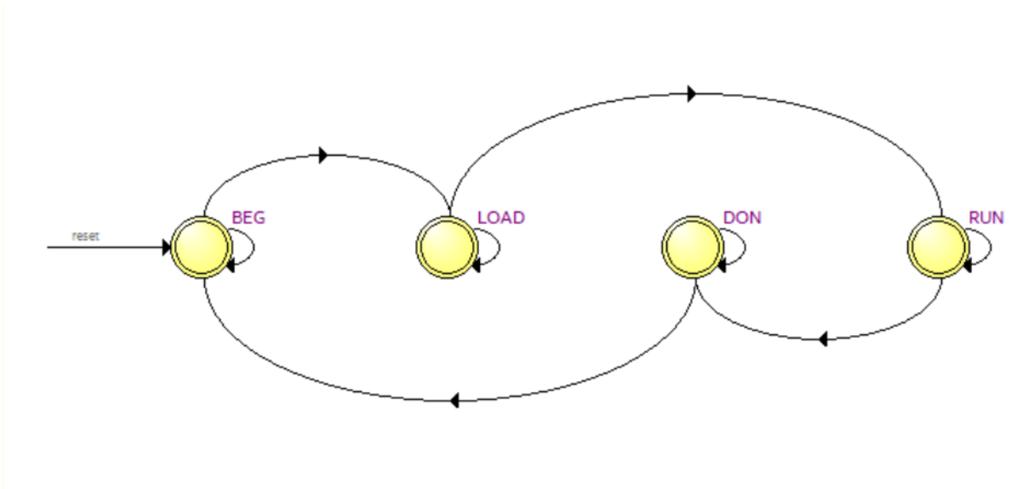

Fig. 7. Finite State Machine

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.

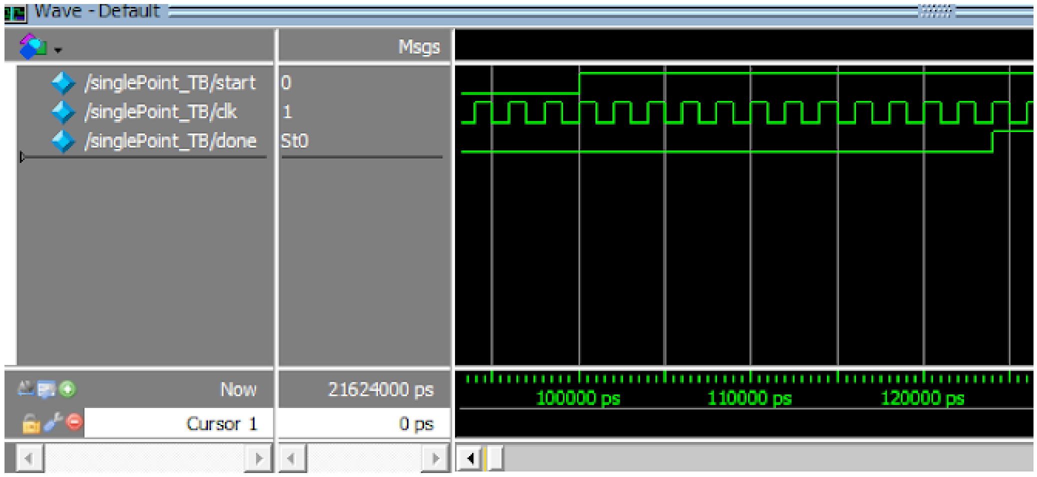

Fig. 8. ModelSim Simulation Showing Clock Cycles

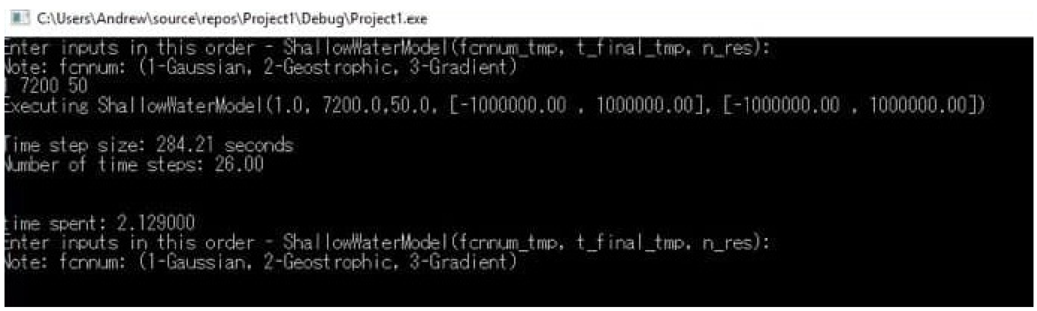

Fig. 9. Output of C Simulation

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 $128\,\mathrm{MHz}$. This is ignoring the fact that the bus clock speed maced at only $100\,\mathrm{MHz}$. 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 1.02e-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 $1.02 \times 10^{-4}\,\mathrm{ms}$ per point we would get $0.663\,\mathrm{s}$ which is 3.2 times faster than the $2.13\,\mathrm{s}$ 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

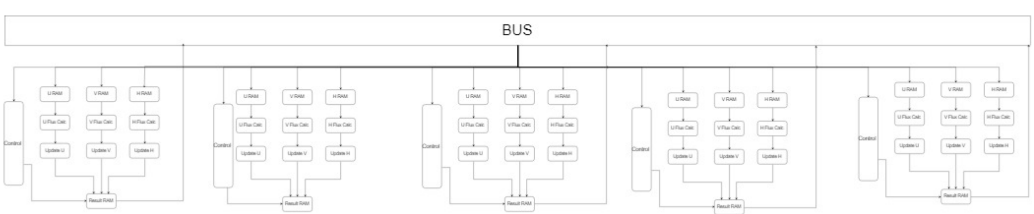

Fig. 12. Parallel Implementation

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 Joeseph 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. [Accessed: 06-Jun-2019].

[5] "Shallow water equations," *Wikipedia*, 20-Apr-2019. [Online]. Available: 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.

```c
// File: CShallowWater.c
// Created by Brian Dang and Andrew Lu
// EEC 181
// Ported MATLAB code to calculate Shalow Water Flow for different resolutions and time frames

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <math.h>
#define RES 50
#define HALO 2

double gravity =  9.80616;

void GaussianFcn(double x, double y, double *h, double *u, double *v)
{
  double a;

  /*  Radius from the center of the hill */
  a = sqrt(x * x + y * y);

  /*  Half-width of the hill */
  /*  Fluid height perturbation */
  /*  Height of the hill at this point */
  *h = 25.0 + 5.0 * exp(-(a * a) / 1.6E+11);

  /*  Zero velocity */
  *u = 0.0;
  *v = 0.0;
}

void GeostrophicFcn(double x, double y, double *h, double *u, double *v, double omega)
{
  double h_tmp;
  (void)x;

  /*  Background height */
  /*  Background velocity */
  /*  Latitude (override) */
  /*  Coriolis parameter (override) */
  /*  Wavelength */
  /*  Height of the hill at this point */
  h_tmp = 6.2831853071795862 * y / 1.0E+6;
  *h = 25.0 + 30.0 * (2.0 * omega * 0.70710678118654746) / gravity * 1.0E+6 /
    6.2831853071795862 * cos(h_tmp);

  /*  Fixed zonal velocity */
  *u = 30.0 * sin(h_tmp);

  /*  Zero meridional velocity */
  *v = 0.0;
}

void GradientFcn(double x, double y, double *h, double *u, double *v)
{
  double r;
  double theta;
  double uphi;
  double h_tmp;

  /*  Radius from the center of the hill */
  r = sqrt(x * x + y * y);

  /*  Angle from the center of the hill */
  theta = atan2(y, x);

  /*  Half-width of the hill */
```

```c
  /*  Fluid height perturbation */
  /*  Background fluid height */
  /*  Calculate fluid height */
  uphi = r * r;
  h_tmp = exp(-uphi / 4.0E+10);
  *h = 25.0 - 20.0 * h_tmp;

  /*  Rotational velocity */
  uphi = sqrt(gravity * 20.0 * (2.0 * uphi / 4.0E+10) * h_tmp);
  if (r == 0.0) {
    *u = 0.0;
    *v = 0.0;
  } else {
    *u = -sin(theta) * uphi;
    *v = cos(theta) * uphi;
  }
}

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]){
    int i;
    int j;
    for(j = 0; j < RES + 2*HALO; j++){
        h[0][j] = h[RES][j];
        h[1][j] = h[RES + 1][j];
        h[RES + HALO][j] = h[HALO][j];
        h[(RES + 2 * HALO) - 1][j] = h[2 * HALO - 1][j];

        hu[0][j] = hu[RES][j];
        hu[1][j] = hu[RES + 1][j];
        hu[RES + HALO][j] = hu[HALO][j];
        hu[(RES + 2 * HALO) - 1][j] = hu[2 * HALO - 1][j];

        hv[0][j] = hv[RES][j];
        hv[1][j] = hv[RES + 1][j];
        hv[RES + HALO][j] = hv[HALO][j];
        hv[(RES + 2 * HALO) - 1][j] = hv[2 * HALO - 1][j];
    }
    for(i = 0; i < RES + 2*HALO; i++){
        h[i][0] = h[0][RES];
        h[i][1] = h[1][RES + 1];
        h[i][RES + HALO] = h[i][HALO];
        h[i][(RES + 2 * HALO) - 1] = h[i][2 * HALO - 1];

        hu[i][0] = hu[0][RES];
        hu[i][1] = hu[1][RES + 1];
        hu[i][RES + HALO] = hu[i][HALO];
        hu[i][(RES + 2 * HALO) - 1] = hu[i][2 * HALO - 1];

        hv[i][0] = hv[0][RES];
        hv[i][1] = hv[1][RES + 1];
        hv[i][RES + HALO] = hv[i][HALO];
        hv[i][(RES + 2 * HALO) - 1] = hv[i][2 * HALO - 1];
    }

}

void CalculateXFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES
+ 2*HALO],
  double hv[RES + 2*HALO][RES + 2*HALO], double F[RES + 1][RES][3]){
    int i;
    int j;
    int io;
    int jo;
    double h_left;
    double hu_left;
    double hv_left;
    double h_right;
    double hu_right;
```

```
    double hv_right;
    double h_flux_left;
    double hu_flux_left;
    double hv_flux_left;
    double h_flux_right;
    double hu_flux_right;
    double hv_flux_right;
    double max_wave_speed;

    for(i = 0; i < RES + 1; i++){
        for(j = 0; j < RES; j++){
            io = i + HALO;
            jo = j + HALO;

        // Calculate left state (to 2nd order accuracy)
            h_left  = 0.25 * h[io][jo]  + h[io-1][jo]  - 0.25 * h[io-2][jo];
            hu_left = 0.25 * hu[io][jo] + hu[io-1][jo] - 0.25 * hu[io-2][jo];
            hv_left = 0.25 * hv[io][jo] + hv[io-1][jo] - 0.25 * hv[io-2][jo];

        // Calculate right state (to 2nd order accuracy)
            h_right  = 0.25 * h[io-1][jo]  + h[io][jo]  - 0.25 * h[io+1][jo];
            hu_right  = 0.25 * hu[io-1][jo]  + hu[io][jo]  - 0.25 * hu[io+1][jo];
            hv_right  = 0.25 * hv[io-1][jo]  + hv[io][jo]  - 0.25 * hv[io+1][jo];

        // Left flux
            h_flux_left = hu_left;
            hu_flux_left = hu_left * hu_left / h_left + 0.5 * gravity * h_left * h_left;
            hv_flux_left = hu_left * hv_left / h_left;

            h_flux_right = hu_right;
            hu_flux_right = hu_right * hu_right / h_right + 0.5 * gravity * h_right * h_right;
            hv_flux_right = hu_right * hv_right / h_right;

        // Max wave speed
            max_wave_speed = abs(0.5 * (hu_left / h_left + hu_right / h_right)) + sqrt(gravity * 0.5 * (h_left + h_right));

        // Calculate flux
            F[i][j][0] = 0.5 * ( h_flux_left +  h_flux_right) - 0.5 * max_wave_speed * (h_right - h_left);
            F[i][j][1] = 0.5 * (hu_flux_left + hu_flux_right) - 0.5 * max_wave_speed * (hu_right - hu_left);
            F[i][j][2] = 0.5 * (hv_flux_left + hv_flux_right) - 0.5 * max_wave_speed * (hv_right - hv_left);

        }
    }
}

void UpdateXFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES + 2*HALO],
  double hv[RES + 2*HALO][RES + 2*HALO], double XFlux[RES + 1][RES][3], double dx, double dt){
    int io;
    int jo;
    int i;
    int j;
    for(i = 0; i < RES + 1; i++){
        for(j = 0; j < RES; j++){
            io = i + HALO;
            jo = j + HALO;

            h[io][jo]   = h[io][jo]   + dt / dx * XFlux[i][j][0];
            h[io-1][jo] = h[io-1][jo] - dt / dx * XFlux[i][j][0];

            hu[io][jo]   = hu[io][jo]   + dt / dx * XFlux[i][j][1];
            hu[io-1][jo] = hu[io-1][jo] - dt / dx * XFlux[i][j][1];

```

```
            hv[io][jo]   = hv[io][jo]   + dt / dx * XFlux[i][j][2];
            hv[io-1][jo] = hv[io-1][jo] - dt / dx * XFlux[i][j][2];

        }
    }
}

void CalculateYFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES + 2*HALO],
  double hv[RES + 2*HALO][RES + 2*HALO], double F[RES][RES + 1][3]){
    int i;
    int j;
    int io;
    int jo;
    double h_left;
    double hu_left;
    double hv_left;
    double h_right;
    double hu_right;
    double hv_right;
    double h_flux_left;
    double hu_flux_left;
    double hv_flux_left;
    double h_flux_right;
    double hu_flux_right;
    double hv_flux_right;
    double max_wave_speed;

    for(i = 0; i < RES; i++){
        for(j = 0; j < RES + 1; j++){
            io = i + HALO;
            jo = j + HALO;

        // Calculate left state (to 2nd order accuracy)
            h_left  = 0.25 * h[io][jo]  + h[io][jo-1]  - 0.25 * h[io][jo-2];
            hu_left = 0.25 * hu[io][jo] + hu[io][jo-1] - 0.25 * hu[io][jo-2];
            hv_left = 0.25 * hv[io][jo] + hv[io][jo-1] - 0.25 * hv[io][jo-2];

        // Calculate right state (to 2nd order accuracy)
            h_right   = 0.25 * h[io][jo-1]   + h[io][jo]   - 0.25 * h[io][jo+1];
            hu_right  = 0.25 * hu[io][jo-1]  + hu[io][jo]  - 0.25 * hu[io][jo+1];
            hv_right  = 0.25 * hv[io][jo-1]  + hv[io][jo]  - 0.25 * hv[io][jo+1];

        // Left flux
            h_flux_left = hu_left;
            hv_flux_left = hu_left * hu_left / h_left + 0.5 * gravity * h_left * h_left;
            hu_flux_left = hu_left * hv_left / h_left;

            h_flux_right = hu_right;
            hv_flux_right = hu_right * hu_right / h_right + 0.5 * gravity * h_right * h_right;
            hu_flux_right = hu_right * hv_right / h_right;

        // Max wave speed
            max_wave_speed = abs(0.5 * (hu_left / h_left + hu_right / h_right)) + sqrt(gravity * 0.5 * (h_left + h_right));

        // Calculate flux
            F[i][j][0] = 0.5 * ( h_flux_left +  h_flux_right) - 0.5 * max_wave_speed * (h_right - h_left);
            F[i][j][1] = 0.5 * (hu_flux_left + hu_flux_right) - 0.5 * max_wave_speed * (hu_right - hu_left);
            F[i][j][2] = 0.5 * (hv_flux_left + hv_flux_right) - 0.5 * max_wave_speed * (hv_right - hv_left);

        }
    }
}
```

```
void UpdateYFluxes(double h[RES + 2*HALO][RES + 2*HALO], double hu[RES + 2*HALO][RES +
2*HALO],
  double hv[RES + 2*HALO][RES + 2*HALO], double YFlux[RES][RES + 1][3], double dx,
  double dt){
    int io;
    int jo;
    int i;
    int j;

    for(i = 0; i < RES; i++){
        for(j = 0; j < RES + 1; j++){
            io = i + HALO;
            jo = j + HALO;

            h[io][jo]   = h[io][jo]   + dt / dx * YFlux[i][j][0];
            h[io][jo-1] = h[io][jo-1] - dt / dx * YFlux[i][j][0];

            hu[io][jo]   = hu[io][jo]   + dt / dx * YFlux[i][j][1];
            hu[io][jo-1] = hu[io][jo-1] - dt / dx * YFlux[i][j][1];

            hv[io][jo]   = hv[io][jo]   + dt / dx * YFlux[i][j][2];
            hv[io][jo-1] = hv[io][jo-1] - dt / dx * YFlux[i][j][2];

        }
    }
}

int offset(int x, int y, int t){
    return (t * RES * RES) + (y * RES) + x;
}

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){
  double phi0 =  0.78539816339744828;//latitude
  double cfl = 0.5;
  double omega = pow(7.292, -5);//rotation rate

  double f = 2 * omega * sin(phi0);

  int halo = 2;

  double dx = (x_domain[1] - x_domain[0]) / n_res;
  double dy = (y_domain[1] - y_domain[0]) / n_res;

  int i;
  int j;
  double XEdge[n_res + 2];
  double YEdge[n_res + 2];
  double XFlux[RES + 1][RES][3];
  double YFlux[RES][RES + 1][3];


  for(i = 0; i <= n_res + 1; i++){
    XEdge[i] = i * dx + x_domain[0];
    YEdge[i] = i * dy + y_domain[0];
  }

  for(i = 0; i < n_res; i++){
    X[i] = 0.5 * (XEdge[i] + XEdge[i]);
    Y[i] = 0.5 * (YEdge[i] + YEdge[i]);
  }

  double h[RES + 2*halo][RES + 2*halo];//height and momentums
  double hu[RES + 2*halo][RES + 2*halo];
  double hv[RES + 2*halo][RES + 2*halo];
  double h_old[RES + 2*halo][RES + 2*halo];//height and momentums
  double hu_old[RES + 2*halo][RES + 2*halo];
  double hv_old[RES + 2*halo][RES + 2*halo];
```

```c
    int io;
    int jo;

    for(i = 0; i < n_res; i++){
      for(j = 0; j < n_res; j++){
        io = i + halo;
        jo = j + halo;

        GaussianFcn(X[i], Y[i], &(h[io][jo]), &(hu[io][jo]), &(hv[io][jo]));

        hu[io][jo] *= h[io][jo];
        hv[io][jo] *= h[io][jo];

        //doing Minit initial states with less loops
        if((io >= halo && io < n_res + halo) && (jo >= halo && jo < n_res + halo)){
          Minith[i][j] = h[io][jo];
          Minithu[i][j] = hu[io][jo] / Minith[i][j];
          Minithv[i][j] = hv[io][jo] / Minith[i][j];

        }

      }
    }

    //calculate initial timestep
    double max_wave_speed = 0;
    double wave_speed;
    for(i = 0; i < n_res; i++){
      for(j = 0; j < n_res; j++){
          wave_speed = sqrt(hu[i][j] * hu[i][j]) / h[i][j] + sqrt(gravity * h[i][j]);

          if(wave_speed > max_wave_speed)
              max_wave_speed = wave_speed;
      }
    }

    double dt = cfl * dx / max_wave_speed;

    double nt = ceil(t_final / dt);

    printf("Time step size: %f seconds\n", dt);
    printf("Number of time steps: %f\n", nt);

    *T = (double *)malloc(nt * sizeof(double));
    *Mh = (double *)malloc(nt * RES * RES * sizeof(double));
    *Mhu = (double *)malloc(nt * RES * RES * sizeof(double));
    *Mhv = (double *)malloc(nt * RES * RES * sizeof(double));

    if(*T)
      printf("T malloc success\n");
    if(*Mh)
      printf("Mh malloc success\n");
    if(*Mhu)
      printf("Mhu malloc success\n");
    if(*Mhv)
      printf("Mhv malloc success\n");


    double t_current = 0;
    int t;

    for(t = 0; t < (int)nt; t++){
      //special handling for final timestep
      if(t == nt)
          dt = t_final - t_current;

      ApplyBoundaryConditions(h, hu, hv);

      //store old values
```

```
      for(i = 0; i < RES + 2*halo; i++){
          for(j = 0; j < RES + 2*halo; j++){
              h_old[i][j] = h[i][j];
              hu_old[i][j] = hu[i][j];
              hv_old[i][j] = hv[i][j];
          }
      }

      //predictor step
      CalculateXFluxes(h, hu, hv, XFlux);
      CalculateYFluxes(h, hu, hv, YFlux);

      //update half a timestep
      UpdateXFluxes(h, hu, hv, XFlux, dx, 0.5 * dt);
      UpdateYFluxes(h, hu, hv, YFlux, dx, 0.5 * dt);

      // Apply source terms
      for(i = 0; i < RES; i++){
          for(j = 0; j < RES; j++){
              io = i + halo;
              jo = j + halo;

              hu[io][jo] = hu[io][jo] + 0.5 * dt * f * hv_old[io][jo];
              hv[io][jo] = hv[io][jo] - 0.5 * dt * f * hu_old[io][jo];
          }
      }

      ApplyBoundaryConditions(h, hu, hv);

      CalculateXFluxes(h, hu, hv, XFlux);
      CalculateYFluxes(h, hu, hv, YFlux);

      for(i = 0; i < RES + 2*halo; i++){
          for(j = 0; j < RES + 2*halo; j++){
              h[i][j] = h_old[i][j];
              hu[i][j] = hu_old[i][j];
              hv[i][j] = hv_old[i][j];
          }
      }

      UpdateXFluxes(h, hu, hv, XFlux, dx, dt);
      UpdateYFluxes(h, hu, hv, YFlux, dx, dt);

      t_current = t_current + dt;

      (*T)[t] = t_current;

      int off = 0;
      //store h, u, v for current timestep
      for(i = 0; i < RES; i++){
          for(j = 0; j < RES; j++){
              off = offset(i, j, t);
              (*Mh)[off] = h[HALO + i][HALO + j];
              //printf("h: %f\n", h[HALO + i][HALO + j]);
              (*Mhu)[off] = hu[HALO + i][HALO + j] / h[HALO + i][HALO + j];
              //printf("hu: %f\n", hu[HALO + i][HALO + j]);
              (*Mhv)[off] = hv[HALO + i][HALO + j] / h[HALO + i][HALO + j];
              //printf("hv: %f\n", hv[HALO + i][HALO + j]);
          }
      }

    }

    return nt;
  }

  int main(){
    int fcnnum = 0;
    int n_res = RES;
```

```c
  double t_final = 7200;
  double x_domain[2] = {-1000000, 1000000};
  double y_domain[2] = {-1000000, 1000000};
  double X[RES];
  double Y[RES];
  double *T;
  double *Mh;
  double *Mhu;
  double *Mhv;
  double Minith[n_res][n_res];
  double Minithu[n_res][n_res];
  double Minithv[n_res][n_res];
  int i;
  int j;
  double nt;
  clock_t t;

  t = clock();

  nt = ShallowWaterModel(fcnnum, t_final, n_res, x_domain, y_domain, X, Y, &T, Minith, Minithu, Minithv, &Mh, &Mhu, &Mhv);

  t = clock() - t;
  double time_taken = ((double)t)/CLOCKS_PER_SEC;

  //print inital values
  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      printf("Minith: ");
      printf("%f ", Minith[i][j]);
      if(j == n_res - 1)
        printf("\n");
    }
  }

  printf("\n");

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      printf("Minithu: ");
      printf("%f ", Minithu[i][j]);
        if(j == n_res - 1)
        printf("\n");
    }
  }

  printf("\n");

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      printf("Minithv: ");
      printf("%f ", Minithv[i][j]);
        if(j == n_res - 1)
        printf("\n");
    }
  }

  printf("\n");

  printf("nt: %f", nt);

  //print values for final timestep
  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      int off = offset(i, j, nt - 1);
      printf("Mh: ");
      printf("%f ", Mh[off]);
        if(j == n_res - 1)
        printf("\n");
```

```
      }
    }

    for(i = 0; i < n_res; i++){
      for(j = 0; j < n_res; j++){
        int off = offset(i, j, nt - 1);
        printf("Mhu: ");
        printf("%f ", Mhu[off]);
          if(j == n_res - 1)
          printf("\n");
      }
    }

    for(i = 0; i < n_res; i++){
      for(j = 0; j < n_res; j++){
        int off = offset(i, j, nt - 1);
        printf("Mhv: ");
        printf("%f ", Mhv[off]);
          if(j == n_res - 1)
          printf("\n");
      }
    }

    printf("Program took %f seconds to execute \n", time_taken);

    free(T);
    free(Mh);
    free(Mhv);
    free(Mhu);

    return 0;
  }
```

```c
// File: CShallowWater.c
// Created by Brian Dang
// EEC 181
// C code to run with hardware Verilog, doesn't use math.h libraries

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define RES 4
#define HALO 2
#define FIXED_POINT_FRACTIONAL_BITS 7

typedef uint16_t fixed_point_t;

double gravity =  9.80616;
volatile int *control   = (int*) 0xFF200000;
volatile int *matrixH   = (int*) 0xFF200020;
volatile int *matrixHu   = (int*) 0xFF200040;
volatile int *matrixHv    = (int*) 0xFF200060;
volatile int *matrixResult    = (int*) 0xFF200080;
volatile int *led       = (int*) 0xFF200140;
volatile int *switches  = (int*) 0xFF200130;
volatile int *buttons   = (int*) 0xFF200100;
volatile int *hex3_hex0 = (int*) 0xFF200120;
volatile int *hex5_hex4 = (int*) 0xFF200110;

inline fixed_point_t float_to_fixed(double input)
{
    return (fixed_point_t)(input * (1 << FIXED_POINT_FRACTIONAL_BITS));
}

double fixed16_to_double(uint16_t input, uint8_t fractional_bits)
{
    return ((double)input / (double)(1 << fractional_bits));
}

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]){
    int i;
    int j;
    for(j = 0; j < RES + 2*HALO; j++){
        h[0][j] = h[RES][j];
        h[1][j] = h[RES + 1][j];
        h[RES + HALO][j] = h[HALO][j];
        h[(RES + 2 * HALO) - 1][j] = h[2 * HALO - 1][j];

        hu[0][j] = hu[RES][j];
        hu[1][j] = hu[RES + 1][j];
        hu[RES + HALO][j] = hu[HALO][j];
        hu[(RES + 2 * HALO) - 1][j] = hu[2 * HALO - 1][j];

        hv[0][j] = hv[RES][j];
        hv[1][j] = hv[RES + 1][j];
        hv[RES + HALO][j] = hv[HALO][j];
        hv[(RES + 2 * HALO) - 1][j] = hv[2 * HALO - 1][j];
    }
    for(i = 0; i < RES + 2*HALO; i++){
        h[i][0] = h[0][RES];
        h[i][1] = h[1][RES + 1];
        h[i][RES + HALO] = h[i][HALO];
        h[i][(RES + 2 * HALO) - 1] = h[i][2 * HALO - 1];

        hu[i][0] = hu[0][RES];
        hu[i][1] = hu[1][RES + 1];
        hu[i][RES + HALO] = hu[i][HALO];
        hu[i][(RES + 2 * HALO) - 1] = hu[i][2 * HALO - 1];

        hv[i][0] = hv[0][RES];
```

```
            hv[i][1] = hv[1][RES + 1];
            hv[i][RES + HALO] = hv[i][HALO];
            hv[i][(RES + 2 * HALO) - 1] = hv[i][2 * HALO - 1];
        }

    }

    int offset(int x, int y, int t){
        return (t * RES * RES) + (y * RES) + x;
    }

    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){
      double phi0 =  0.78539816339744828;//latitude
      double cfl = 0.5;

      int halo = 2;

      double dx = (x_domain[1] - x_domain[0]) / n_res;
      double dy = (y_domain[1] - y_domain[0]) / n_res;

      int i;
      int j;
      double XEdge[n_res + 2];
      double YEdge[n_res + 2];
      double XFlux[RES + 1][RES][3];
      double YFlux[RES][RES + 1][3];


      for(i = 0; i <= n_res + 1; i++){
        XEdge[i] = i * dx + x_domain[0];
        YEdge[i] = i * dy + y_domain[0];
      }

      for(i = 0; i < n_res; i++){
        X[i] = 0.5 * (XEdge[i] + XEdge[i]);
        Y[i] = 0.5 * (YEdge[i] + YEdge[i]);
      }

      double h[RES + 2*halo][RES + 2*halo];//height and momentums
      double hu[RES + 2*halo][RES + 2*halo];
      double hv[RES + 2*halo][RES + 2*halo];
      double h2[RES + 2*halo][RES + 2*halo];//height and momentums
      double hu2[RES + 2*halo][RES + 2*halo];
      double hv2[RES + 2*halo][RES + 2*halo];
      int io;
      int jo;

      //loading up my own values because I can't run math.h from GaussianFcn
      for(i = 0; i < RES + 2*halo; i++){
         for(j = 0; j < RES + 2*halo; j++){
            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){
                h[i][j] = 0;
            }
            else{
                h[i][j] = 30;
            }
            hu[i][j] = 0;
            hv[i][j] = 0;
         }
      }
      for(i = 0; i < n_res; i++){
        for(j = 0; j < n_res; j++){
          io = i + halo;
          jo = j + halo;

          //doing Minit initial states with less loops
```

```
        if((io >= halo && io < n_res + halo) && (jo >= halo && jo < n_res + halo)){
          Minith[i][j] = h[io][jo];
          Minithu[i][j] = hu[io][jo] / Minith[i][j];
          Minithv[i][j] = hv[io][jo] / Minith[i][j];

        }

      }
    }

    for(i = 0; i < RES + 2*halo; i++){
       for(j = 0; j < RES + 2*halo; j++){
         printf("h: %f ", h[i][j]);
         if(j == (RES + 2*halo) - 1)
           printf("\n");
       }
     }

    double dt = 1.46;
    double nt = 7;

    printf("Time step size: %f seconds\n", dt);
    printf("Number of time steps: %f\n", nt);

    *T = (double *)malloc(nt * sizeof(double));
    *Mh = (double *)malloc(nt * RES * RES * sizeof(double));
    *Mhu = (double *)malloc(nt * RES * RES * sizeof(double));
    *Mhv = (double *)malloc(nt * RES * RES * sizeof(double));

    if(*T)
      printf("T malloc success\n");
    if(*Mh)
      printf("Mh malloc success\n");
    if(*Mhu)
      printf("Mhu malloc success\n");
    if(*Mhv)
      printf("Mhv malloc success\n");


    double t_current = 0;
    int t;

    for(t = 0; t < (int)nt; t++){
      //special handling for final timestep
      if(t == nt)
          dt = t_final - t_current;

      //starting verilog code, if statement to load from h or h2 array
      if((int)nt % 2 == 0){//first case
          ApplyBoundaryConditions(h, hu, hv);
      for(i = 0; i < RES + 1; i++){
          for(j = 0; j < RES + 1; j++){
              io = i + HALO;
              jo = j + HALO;

              //load values through bus
              *matrixH = float_to_fixed(h[io - 2][jo]);
              *(matrixH + 1) = float_to_fixed(h[io - 1][jo]);
              *(matrixH + 2) = float_to_fixed(h[io + 1][jo]);
              *(matrixH + 3) = float_to_fixed(h[io][jo]);
              *(matrixH + 4) = float_to_fixed(h[io][jo - 2]);
              *(matrixH + 5) = float_to_fixed(h[io][jo - 1]);
              *(matrixH + 6) = float_to_fixed(h[io][jo + 1]);
              *(matrixH + 7) = float_to_fixed(dt);

              *matrixHu = float_to_fixed(hu[io - 2][jo]);
              *(matrixHu + 1) = float_to_fixed(hu[io - 1][jo]);
              *(matrixHu + 2) = float_to_fixed(hu[io + 1][jo]);
              *(matrixHu + 3) = float_to_fixed(hu[io][jo]);
```

```
                *(matrixHu + 4) = float_to_fixed(hu[io][jo - 2]);
                *(matrixHu + 5) = float_to_fixed(hu[io][jo - 1]);
                *(matrixHu + 6) = float_to_fixed(hu[io][jo + 1]);
                *(matrixHu + 7) = float_to_fixed(dx);

                *matrixHv = float_to_fixed(hv[io - 2][jo]);
                *(matrixHv + 1) = float_to_fixed(hv[io - 1][jo]);
                *(matrixHv + 2) = float_to_fixed(hv[io + 1][jo]);
                *(matrixHv + 3) = float_to_fixed(hv[io][jo]);
                *(matrixHv + 4) = float_to_fixed(hv[io][jo - 2]);
                *(matrixHv + 5) = float_to_fixed(hv[io][jo - 1]);
                *(matrixHv + 6) = float_to_fixed(hv[io][jo + 1]);

                *control = 0x0002;//set start

                //wait for done to go high
                while(*control != 0xB555) {
                    *led = 0x0000AAAA;
                }

                *control = 0x00000000;//set start to 0

                // Verify that done goes low
                while (*control != 0xB554) {
                    *led = 0x0000000F;
                }

                //store output of verilog
                h2[io][jo] = fixed16_to_double(*(matrixResult), 7);
                h2[io-1][jo] = fixed16_to_double(*(matrixResult + 1), 7);
                hu2[io][jo] = fixed16_to_double(*(matrixResult + 2), 7);
                hu2[io-1][jo] = fixed16_to_double(*(matrixResult+3), 7);
                hv2[io][jo] = fixed16_to_double(*(matrixResult+4), 7);
                hv2[io-1][jo] = fixed16_to_double(*(matrixResult+5), 7);
                h2[io][jo-1] = fixed16_to_double(*(matrixResult+6), 7);
                hu2[io][jo-1] = fixed16_to_double(*(matrixResult+7), 7);
                hv2[io][jo-1] = fixed16_to_double(*(matrixResult+8), 7);

            }
        }
        int off = 0;
        //store outputs into M arrays by timestep
        for(i = 0; i < RES; i++){
            for(j = 0; j < RES; j++){
                off = offset(i, j, t);
                (*Mh)[off] = h2[HALO + i][HALO + j];
                //printf("h: %f\n", h[HALO + i][HALO + j]);
                (*Mhu)[off] = hu2[HALO + i][HALO + j] / h[HALO + i][HALO + j];
                //printf("hu: %f\n", hu[HALO + i][HALO + j]);
                (*Mhv)[off] = hv2[HALO + i][HALO + j] / h[HALO + i][HALO + j];
                //printf("hv: %f\n", hv[HALO + i][HALO + j]);
            }
        }

        }
        else{ //loading from 2nd array
        for(i = 0; i < RES + 1; i++){
            for(j = 0; j < RES + 1; j++){
                io = i + HALO;
                jo = j + HALO;

                //load values through bus
                *matrixH = float_to_fixed(h2[io - 2][jo]);
                *(matrixH + 1) = float_to_fixed(h2[io - 1][jo]);
                *(matrixH + 2) = float_to_fixed(h2[io + 1][jo]);
                *(matrixH + 3) = float_to_fixed(h2[io][jo]);
                *(matrixH + 4) = float_to_fixed(h2[io][jo - 2]);
                *(matrixH + 5) = float_to_fixed(h2[io][jo - 1]);
                *(matrixH + 6) = float_to_fixed(h2[io][jo + 1]);
```

```
          *(matrixH + 7) = float_to_fixed(dt);

          *matrixHu = float_to_fixed(hu2[io - 2][jo]);
          *(matrixHu + 1) = float_to_fixed(hu2[io - 1][jo]);
          *(matrixHu + 2) = float_to_fixed(hu2[io + 1][jo]);
          *(matrixHu + 3) = float_to_fixed(hu2[io][jo]);
          *(matrixHu + 4) = float_to_fixed(hu2[io][jo - 2]);
          *(matrixHu + 5) = float_to_fixed(hu2[io][jo - 1]);
          *(matrixHu + 6) = float_to_fixed(hu2[io][jo + 1]);
          *(matrixHu + 7) = float_to_fixed(dx);

          *matrixHv = float_to_fixed(hv2[io - 2][jo]);
          *(matrixHv + 1) = float_to_fixed(hv2[io - 1][jo]);
          *(matrixHv + 2) = float_to_fixed(hv2[io + 1][jo]);
          *(matrixHv + 3) = float_to_fixed(hv2[io][jo]);
          *(matrixHv + 4) = float_to_fixed(hv2[io][jo - 2]);
          *(matrixHv + 5) = float_to_fixed(hv2[io][jo - 1]);
          *(matrixHv + 6) = float_to_fixed(hv2[io][jo + 1]);

          *control = 0x0002;//set start

          //wait for done to go high
          while(*control != 0xb5555555) {
              *led = 0x0000AAAA;
          }

          *control = 0x00000000;//set start to 0

          // Verify that done goes low
          while (*control != 0xb5555554) {
              *led = 0x0000000F;
          }

          //store outputs
          h[io][jo] = fixed16_to_double(*(matrixResult), 7);
          h[io-1][jo] = fixed16_to_double(*(matrixResult + 1), 7);
          hu[io][jo] = fixed16_to_double(*(matrixResult + 2), 7);
          hu[io-1][jo] = fixed16_to_double(*(matrixResult+3), 7);
          hv[io][jo] = fixed16_to_double(*(matrixResult+4), 7);
          hv[io-1][jo] = fixed16_to_double(*(matrixResult+5), 7);
          h[io][jo-1] = fixed16_to_double(*(matrixResult+6), 7);
          hu[io][jo-1] = fixed16_to_double(*(matrixResult+7), 7);
          hv[io][jo-1] = fixed16_to_double(*(matrixResult+8), 7);

      }

  }
  int off = 0;
  //store h, u ,v for current timestep
  for(i = 0; i < RES; i++){
      for(j = 0; j < RES; j++){
          off = offset(i, j, t);
          (*Mh)[off] = h[HALO + i][HALO + j];
          //printf("h: %f\n", h[HALO + i][HALO + j]);
          (*Mhu)[off] = hu[HALO + i][HALO + j] / h[HALO + i][HALO + j];
          //printf("hu: %f\n", hu[HALO + i][HALO + j]);
          (*Mhv)[off] = hv[HALO + i][HALO + j] / h[HALO + i][HALO + j];
          //printf("hv: %f\n", hv[HALO + i][HALO + j]);
      }
  }

  }

  t_current = t_current + dt;
  //store current timestep
  (*T)[t] = t_current;

}

```

```c
  return nt;
}

int main(){
  int fcnnum = 0;
  int n_res = RES;
  double t_final = 10;
  double x_domain[2] = {-100, 100};
  double y_domain[2] = {-100, 100};
  double X[RES];
  double Y[RES];
  double *T;
  double *Mh;
  double *Mhu;
  double *Mhv;
  double Minith[n_res][n_res];
  double Minithu[n_res][n_res];
  double Minithv[n_res][n_res];
  int i;
  int j;
  double nt;

  nt = ShallowWaterModel(fcnnum, t_final, n_res, x_domain, y_domain, X, Y, &T, Minith,
  Minithu, Minithv, &Mh, &Mhu, &Mhv);

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      printf("Minith: ");
      printf("%f ", Minith[i][j]);
      if(j == n_res - 1)
        printf("\n");
    }
  }

  printf("\n");

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      printf("Minithu: ");
      printf("%f ", Minithu[i][j]);
        if(j == n_res - 1)
        printf("\n");
    }
  }

  printf("\n");

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      printf("Minithv: ");
      printf("%f ", Minithv[i][j]);
        if(j == n_res - 1)
        printf("\n");
    }
  }

  printf("\n");

  printf("nt: %f", nt);

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      int off = offset(i, j, nt - 1);
      printf("Mh: ");
      printf("%f ", Mh[off]);
        if(j == n_res - 1)
        printf("\n");
    }
  }
```

```c

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      int off = offset(i, j, nt - 1);
      printf("Mhu: ");
      printf("%f ", Mhu[off]);
        if(j == n_res - 1)
        printf("\n");
    }
  }

  for(i = 0; i < n_res; i++){
    for(j = 0; j < n_res; j++){
      int off = offset(i, j, nt - 1);
      printf("Mhv: ");
      printf("%f ", Mhv[off]);
        if(j == n_res - 1)
        printf("\n");
    }
  }

  free(T);
  free(Mh);
  free(Mhv);
  free(Mhu);

  return 0;
}
```

```
/*
 * bussedAndFSMTop.v - the top level module to communicate with C code and run computation modules
 * Brian Dang - Shallow Water Simulation - EEC 181
 * May 2019
 */

`define H 2'b00
`define HU 2'b01
`define HV 2'b10

`define X 1'b0
`define Y 1'b1

// constants and computation parameters
`define HALO 2'd2
`define GRAVITY 13'b0010011100111// 9.80616 in decimal Q6.7 format - 13'b001001_1100111

`define CONTROL_OFFSET 3'b000
`define MATRIX_H_ADDRESS_OFFSET 3'b001
`define MATRIX_HU_ADDRESS_OFFSET 3'b010
`define MATRIX_HV_ADDRESS_OFFSET 3'b011
`define MATRIX_RESULT_ADDRESS_OFFSET 3'b100
`define CLEAR_INDEX 2'b0
`define MULTIPLY_INDEX 2'b01

module bussedAndFSMTop (
    // NUM_DATA is number of initial data points
    // NUM_GRID_SIMULT is number of grid points we'll be calculating at the same time
        // signals to connect to an Avalon clock source interface
    clk,
    reset,

    // signals to connect to an Avalon-MM slave interface
    slave_address,
    slave_read,
    slave_write,
    slave_readdata,
    slave_writedata,
    slave_byteenable
);
    parameter DATA_WIDTH = 16;
    parameter BEG = 2'b00;
    parameter LOAD = 2'b01;
    parameter RUN = 2'b11;
    parameter DON = 2'b10;

    input clk;
    input reset;

    input [6:0] slave_address;
    input slave_read;
    input slave_write;

    output reg [DATA_WIDTH-1:0] slave_readdata;
    input [DATA_WIDTH-1:0] slave_writedata;
    input [(DATA_WIDTH/8)-1:0] slave_byteenable;

    reg [7:0] dt_in;
    reg [12:0] dx_in;
    reg [7:0] dt_in_c;
    reg [12:0] dx_in_c;
    reg done;
    reg done_c;

    reg start;

    // data matrices - MIGHT NEED TO GET REWORKED WITH BUS STUFF
    reg signed [12:0] h [6:0]; // i-2, i-1, i+1, 0, j-2, j-2, j+1
```

```
reg signed [12:0] hu [6:0];
reg signed [12:0] hv [6:0];
reg signed [12:0] h_old [6:0];
reg signed [12:0] hu_old [6:0];
reg signed [12:0] hv_old [6:0];

reg signed [12:0] h_c [6:0]; // i-2, i-1, i+1, 0, j-2, j-2, j+1
reg signed [12:0] hu_c [6:0];
reg signed [12:0] hv_c [6:0];
reg signed [12:0] h_old_c [6:0];
reg signed [12:0] hu_old_c [6:0];
reg signed [12:0] hv_old_c [6:0];

// intermediate data
wire signed [25:0] h_flux_left, h_flux_right, hu_flux_left, hu_flux_right,
hv_flux_left, hv_flux_right;
wire signed [12:0] h_left, h_right, hu_left, hu_right, hv_left, hv_right;
wire signed [12:0] Fh_x, Fhu_x, Fhv_x, Fh_y, Fhu_y, Fhv_y;

// all the start control signals
//reg start_boundCon;
reg start_calcXFlux, start_calcYFlux;
reg start_upXFlux, start_upYFlux;
reg start_calcXFlux_c, start_calcYFlux_c;
reg start_upXFlux_c, start_upYFlux_c;

// all the done control signals
reg done_boundCon_h, done_boundCon_hu, done_boundCon_hv;
reg done_boundCon_h_c, done_boundCon_hu_c, done_boundCon_hv_c;
wire done_flux_left;
wire done_fh_x, done_fhu_x, done_fhv_x, done_fh_y, done_fhu_y, done_fhv_y;
wire done_uph_x, done_uphu_x, done_uphv_x, done_uph_y, done_uphu_y, done_uphv_y;

// other control signals
reg update_half; // for UpdateFluxes | 0 = whole, 1 = half
reg update_half_c; // for UpdateFluxes | 0 = whole, 1 = half

// buffer signals
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;

reg [3:0] rd_addressH;
reg [3:0] rd_addressHu;
reg [3:0] rd_addressHv;

reg [3:0] rd_addressH_c;
reg [3:0] rd_addressHu_c;
reg [3:0] rd_addressHv_c;

wire [DATA_WIDTH-1:0] matrixH_dout;
wire [DATA_WIDTH-1:0] matrixHu_dout;
wire [DATA_WIDTH-1:0] matrixHv_dout;
wire [DATA_WIDTH-1:0] matrixResult_dout;
reg [DATA_WIDTH-1:0] fluxData;//data to write to results vector
reg [DATA_WIDTH-1:0] fluxData_c;

wire matrixH_wren = slave_write & (slave_address[6:4] == `MATRIX_H_ADDRESS_OFFSET);
wire matrixHu_wren = slave_write & (slave_address[6:4] == `MATRIX_HU_ADDRESS_OFFSET);
wire matrixHv_wren = slave_write & (slave_address[6:4] == `MATRIX_HV_ADDRESS_OFFSET);

reg Rwr_en;//write enable for results
reg Rwr_en_c;
reg wr_addr;
reg wr_addr_c;

reg [1:0] state;
reg [1:0] state_c;

newhRam matrixH(
```

```
        .clock ( clk),
        .data ( slave_writedata ),
        .rdaddress (rd_addressH),
        .wraddress (slave_address[3:0]),
        .wren (matrixH_wren),
        .q (matrixH_dout)
    );

    newhRam matrixHu(
        .clock ( clk),
        .data ( slave_writedata ),
        .rdaddress (rd_addressHu),
        .wraddress (slave_address[3:0]),
        .wren (matrixHu_wren),
        .q (matrixHu_dout)
    );

    newhRam matrixHv(
        .clock ( clk),
        .data ( slave_writedata ),
        .rdaddress (rd_addressHv),
        .wraddress (slave_address[3:0]),
        .wren (matrixHv_wren),
        .q (matrixHv_dout)
    );

    newhRam matrixResult(
        .clock ( clk),
        .data (fluxData),
        .rdaddress (slave_address[3:0]),
        .wraddress (wr_addr),
        .wren (Rwr_en),
        .q (matrixResult_dout)
    );

    CalcFluxes XFluxH (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
        .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
        .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
        .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fh_x), .done(done_fh_x)
    );
    CalcFluxes XFluxHU (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
        .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
        .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
        .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhu_x), .done(done_fhu_x)
    );
    CalcFluxes XFluxHV (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
        .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
        .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
        .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhv_x), .done(done_fhv_x)
    );

    CalcFluxes YFluxH (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
        .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[6]),
        .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[6]),
        .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[6]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fh_y), .done(done_fh_y)
    );
    CalcFluxes YFluxHU (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
        .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[6]),
        .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[6]),
        .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[6]),
        .halo(`HALO), .gravity(`GRAVITY),
```

```
        .F(Fhu_y), .done(done_fhu_y)
    );
    CalcFluxes YFluxHV (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
        .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[6]),
        .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[6]),
        .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[6]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhv_y), .done(done_fhv_y)
    );

    // update fluxes
    UpdateFluxes XUpH (.clk(clk), .start(start_upXFlux), .half(update_half),
        .h_0 (h[3]), .h_1(h[1]),
        .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
        .h_0_new(h_0_new), .h_1_new(h_1_new_x),
        .done(done_uph_x)
    );
    UpdateFluxes XUpHU (.clk(clk), .start(start_upXFlux), .half(update_half),
        .h_0 (hu[3]), .h_1(hu[1]),
        .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
        .h_0_new(hu_0_new), .h_1_new(hu_1_new_x),
        .done(done_uphu_x)
    );
    UpdateFluxes XUpHV (.clk(clk), .start(start_upXFlux), .half(update_half),
        .h_0 (hv[3]), .h_1(hv[1]),
        .dt(dt_in), .dx(dx_in), .f_h(Fh_x),
        .h_0_new(hv_0_new), .h_1_new(hv_1_new_x),
        .done(done_uphv_x)
    );

    UpdateFluxes YUpH (.clk(clk), .start(start_upYFlux), .half(update_half),
        .h_0 (h[3]), .h_1(h[5]),
        .dt(dt_in), .dx(dx_in), .f_h(Fh_y),
        .h_0_new(foobar), .h_1_new(h_1_new_y),
        .done(done_uph_y)
    );
    UpdateFluxes YUpHU (.clk(clk), .start(start_upYFlux), .half(update_half),
        .h_0 (hu[3]), .h_1(hu[5]),
        .dt(dt_in), .dx(dx_in), .f_h(Fhu_y),
        .h_0_new(foobar), .h_1_new(hu_1_new_y),
        .done(done_uphu_y)
    );
    UpdateFluxes YUpHV (.clk(clk), .start(start_upYFlux), .half(update_half),
        .h_0 (h[3]), .h_1(hv[5]),
        .dt(dt_in), .dx(dx_in), .f_h(Fhv_y),
        .h_0_new(foobar), .h_1_new(hv_1_new_y),
        .done(done_uphv_y)
    );

    //FSM for controlling bus
    always @(*) begin
    state_c = state;
    rd_addressH_c = rd_addressH;
    rd_addressHu_c = rd_addressHu;
    rd_addressHv_c = rd_addressHv;
    wr_addr_c = wr_addr;
    Rwr_en_c = Rwr_en;
        case(state)
            BEG: begin//beginning state
                if(start == 1'b0) begin //if we are still loading
                    start_calcXFlux_c = 1'b0;
                    start_calcYFlux_c = 1'b0;
                    start_upXFlux_c = 1'b0;
                    start_upYFlux_c = 1'b0;
                    done_boundCon_h_c = 1'b0;
                    done_boundCon_hu_c = 1'b0;
                    done_boundCon_hv_c = 1'b0;
                    done_c = 1'b0;
                    update_half_c = 1'b0;
```

```
                    Rwr_en_c = 1'b0;
                end
                if(start == 1'b1) begin //done writing values and can load
                    rd_addressH_c = 1'b0;
                    rd_addressHu_c = 1'b0;
                    rd_addressHv_c = 1'b0;
                    state_c = LOAD;
                end
            end
            LOAD: begin
                if(rd_addressH < 3'd6 && rd_addressHu < 3'd6 && rd_addressHv < 3'd6)
                begin //get 7 h values
                    h_c[rd_addressH] = matrixH_dout;
                    hu_c[rd_addressH] = matrixHu_dout;
                    hv_c[rd_addressH] = matrixHv_dout;
                    rd_addressH_c = rd_addressH + 1'b1;
                    rd_addressHu_c = rd_addressHu + 1'b1;
                    rd_addressHv_c = rd_addressHv + 1'b1;
                end
                if(rd_addressH == 3'd7 && rd_addressHu == 3'd7 && rd_addressHv == 3'd7)
                begin //get dt and dx
                    dt_in_c = matrixH_dout;
                    dx_in_c = matrixHu_dout;
                    done_boundCon_h_c = 1'b1;
                    done_boundCon_hu_c = 1'b1;
                    done_boundCon_hv_c = 1'b1;
                    state_c = RUN;
                end
            end
            RUN: begin //run the algorithm
                // store old variables
                if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv) begin
                    h_old_c[0] = h[0];
                    h_old_c[1] = h[1];
                    h_old_c[2] = h[2];
                    h_old_c[3] = h[3];
                    h_old_c[4] = h[4];
                    h_old_c[5] = h[5];
                    h_old_c[6] = h[6];

                    hu_old_c[0] = hu[0];
                    hu_old_c[1] = hu[1];
                    hu_old_c[2] = hu[2];
                    hu_old_c[3] = hu[3];
                    hu_old_c[4] = hu[4];
                    hu_old_c[5] = hu[5];
                    hu_old_c[6] = hu[6];

                    hv_old_c[0] = hv[0];
                    hv_old_c[1] = hv[1];
                    hv_old_c[2] = hv[2];
                    hv_old_c[3] = hv[3];
                    hv_old_c[4] = hv[4];
                    hv_old_c[5] = hv[5];
                    hv_old_c[6] = hv[6];
                    // calculate fluxes (predictor step)
                    start_calcXFlux_c = 1'b1;
                    start_calcYFlux_c = 1'b1;
                    done_boundCon_h_c = 1'b0;
                    done_boundCon_hu_c = 1'b0;
                    done_boundCon_hv_c = 1'b0;
                end

                // update half a timestep
                if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
                done_fhv_y) begin
                    start_calcXFlux_c = 1'b0;
                    start_calcYFlux_c = 1'b0;
                    update_half_c = 1'b1;
```

```
                start_upXFlux_c = 1'b1;
                start_upYFlux_c = 1'b1;
            end

            // apply boundary conditions
            if (done_uph_x && done_uphu_x && done_uphv_x && done_uph_y && done_uphu_y && done_uphv_y) begin
                start_upXFlux_c = 0;
                start_upYFlux_c = 0;
                h_c[1] = h_1_new_x;
                h_c[3] = h_0_new;
                h_c[5] = h_1_new_y;
                hu_c[1] = hu_1_new_x;
                hu_c[3] = hu_0_new;
                hu_c[5] = hu_1_new_y;
                hv_c[1] = hv_1_new_x;
                hv_c[3] = hv_0_new;
                hv_c[5] = hv_1_new_y;
                done_boundCon_h_c = 1'b1;
                done_boundCon_hu_c = 1'b1;
                done_boundCon_hv_c = 1'b1;
            end

            // calculate new fluxes
            if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv && update_half) begin
                start_calcXFlux_c = 1'b1;
                start_calcYFlux_c = 1'b1;
            end

            // update a full timestep
            if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y && done_fhv_y && update_half) begin
                start_calcXFlux_c = 1'b0;
                start_calcYFlux_c = 1'b0;
                h_c[0] = h_old[0];
                h_c[1] = h_old[1];
                h_c[2] = h_old[2];
                h_c[3] = h_old[3];
                h_c[4] = h_old[4];
                h_c[5] = h_old[5];
                h_c[6] = h_old[6];

                hu_c[0] = hu_old[0];
                hu_c[1] = hu_old[1];
                hu_c[2] = hu_old[2];
                hu_c[3] = hu_old[3];
                hu_c[4] = hu_old[4];
                hu_c[5] = hu_old[5];
                hu_c[6] = hu_old[6];

                hv_c[0] = hv_old[0];
                hv_c[1] = hv_old[1];
                hv_c[2] = hv_old[2];
                hv_c[3] = hv_old[3];
                hv_c[4] = hv_old[4];
                hv_c[5] = hv_old[5];
                hv_c[6] = hv_old[6];
                update_half_c = 1'b0;
                start_upXFlux_c = 1'b1;
                start_upYFlux_c = 1'b1;
            end

            if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv && !update_half) begin
                start_upXFlux_c = 1'b0;
                start_upYFlux_c = 1'b0;
                state_c = DON;//change to finish and load final values
                wr_addr_c = 1'b0;
```

```
                    Rwr_en_c = 1'b1;
                    fluxData_c = h_0_new;
                end
            end
            DON: begin
                if(start == 1'b1) begin
                    Rwr_en_c = 1'b1;
                    wr_addr_c = wr_addr + 1'b1;
                    case(wr_addr)
                        1'd1: begin
                            fluxData_c = h_1_new_x;
                        end
                        2'd2: begin
                            fluxData_c = hu_0_new;
                        end
                        2'd3: begin
                            fluxData_c = hu_1_new_x;
                        end
                        3'd4: begin
                            fluxData_c = hv_0_new;
                        end
                        3'd5: begin
                            fluxData_c = hv_1_new_x;
                        end
                        3'd6: begin
                            fluxData_c = h_1_new_y;
                        end
                        3'd7: begin
                            fluxData_c = hu_1_new_y;
                        end
                        4'd8: begin
                            fluxData_c = hv_1_new_y;
                            done_c = 1'b1;
                        end
                    endcase
                end
                if(start == 1'b0) begin
                    Rwr_en_c = 1'b0;
                    done_c = 1'b0;
                    state_c = BEG;//back to beginning
                end
            end
        endcase
    end

        //for reading values
    always @(slave_address or matrixH_dout or matrixHu_dout or matrixHv_dout or matrixResult_dout or done) begin
      case(slave_address[6:4])
        `MATRIX_H_ADDRESS_OFFSET: slave_readdata = matrixH_dout;
        `MATRIX_HU_ADDRESS_OFFSET: slave_readdata = matrixHu_dout;
        `MATRIX_HV_ADDRESS_OFFSET: slave_readdata = matrixHv_dout;
        `MATRIX_RESULT_ADDRESS_OFFSET: slave_readdata = matrixResult_dout;
        `CONTROL_OFFSET: slave_readdata = {16'b101_1010_1010_1010, done};
      endcase // case (slave_address[9:8])
    end

    always @(posedge clk) begin
        state <= state_c;
        wr_addr <= wr_addr_c;
        rd_addressH <= rd_addressH_c;
        rd_addressHu <= rd_addressHu_c;
        rd_addressHv <= rd_addressHv_c;
        done_boundCon_h <= done_boundCon_h_c;
        done_boundCon_hu <= done_boundCon_hu_c;
        done_boundCon_hv <= done_boundCon_hv_c;
        done <= done_c;
        Rwr_en <= Rwr_en_c;
```

```
        fluxData <= fluxData_c;
        h[0] <= h_c[0];
        h[1] <= h_c[1];
        h[2] <= h_c[2];
        h[3] <= h_c[3];
        h[4] <= h_c[4];
        h[5] <= h_c[5];
        h[6] <= h_c[6];
        hu[0] <= hu_c[0];
        hu[1] <= hu_c[1];
        hu[2] <= hu_c[2];
        hu[3] <= hu_c[3];
        hu[4] <= hu_c[4];
        hu[5] <= hu_c[5];
        hu[6] <= hu_c[6];
        hv[0] <= hv_c[0];
        hv[1] <= hv_c[1];
        hv[2] <= hv_c[2];
        hv[3] <= hv_c[3];
        hv[4] <= hv_c[4];
        hv[5] <= hv_c[5];
        hv[6] <= hv_c[6];
        h_old[0] <= h_old_c[0];
        h_old[1] <= h_old_c[1];
        h_old[2] <= h_old_c[2];
        h_old[3] <= h_old_c[3];
        h_old[4] <= h_old_c[4];
        h_old[5] <= h_old_c[5];
        h_old[6] <= h_old_c[6];
        hu_old[0] <= hu_old_c[0];
        hu_old[1] <= hu_old_c[1];
        hu_old[2] <= hu_old_c[2];
        hu_old[3] <= hu_old_c[3];
        hu_old[4] <= hu_old_c[4];
        hu_old[5] <= hu_old_c[5];
        hu_old[6] <= hu_old_c[6];
        hv_old[0] <= hv_old_c[0];
        hv_old[1] <= hv_old_c[1];
        hv_old[2] <= hv_old_c[2];
        hv_old[3] <= hv_old_c[3];
        hv_old[4] <= hv_old_c[4];
        hv_old[5] <= hv_old_c[5];
        hv_old[6] <= hv_old_c[6];
        start_calcXFlux <= start_calcXFlux_c;
        start_calcYFlux <= start_calcYFlux_c;
        start_upXFlux <= start_upXFlux_c;
        start_upYFlux <= start_upYFlux_c;
        update_half <= update_half_c;
        if ((slave_write == 1) && (slave_address[6:4] == `CONTROL_OFFSET)) begin
            case (slave_writedata[1])
                1'b0: begin
                    start <= 1'b0;
                end

                1'b1: begin
                    start <= 1'b1;
                end
            endcase // case (slave_address[1:0])
        end
    end

endmodule
```

```
// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: newhRam.v
// Megafunction Name(s):
//          altsyncram
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 17.0.0 Build 595 04/25/2017 SJ Lite Edition
// ************************************************************


//Copyright (C) 2017  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Intel and sold by Intel or its
//authorized distributors.  Please refer to the applicable
//agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module newhRam (
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input     clock;
    input   [15:0]  data;
    input   [3:0]  rdaddress;
    input   [3:0]  wraddress;
    input     wren;
    output  [15:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
    tri0      wren;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [15:0] sub_wire0;
    wire [15:0] q = sub_wire0[15:0];

    altsyncram  altsyncram_component (
                .address_a (wraddress),
                .address_b (rdaddress),
                .clock0 (clock),
```

```
				.data_a (data),
				.wren_a (wren),
				.q_b (sub_wire0),
				.aclr0 (1'b0),
				.aclr1 (1'b0),
				.addressstall_a (1'b0),
				.addressstall_b (1'b0),
				.byteena_a (1'b1),
				.byteena_b (1'b1),
				.clock1 (1'b1),
				.clocken0 (1'b1),
				.clocken1 (1'b1),
				.clocken2 (1'b1),
				.clocken3 (1'b1),
				.data_b ({16{1'b1}}),
				.eccstatus (),
				.q_a (),
				.rden_a (1'b1),
				.rden_b (1'b1),
				.wren_b (1'b0));
	defparam
		altsyncram_component.address_aclr_b = "NONE",
		altsyncram_component.address_reg_b = "CLOCK0",
		altsyncram_component.clock_enable_input_a = "BYPASS",
		altsyncram_component.clock_enable_input_b = "BYPASS",
		altsyncram_component.clock_enable_output_b = "BYPASS",
		altsyncram_component.intended_device_family = "Cyclone V",
		altsyncram_component.lpm_type = "altsyncram",
		altsyncram_component.numwords_a = 16,
		altsyncram_component.numwords_b = 16,
		altsyncram_component.operation_mode = "DUAL_PORT",
		altsyncram_component.outdata_aclr_b = "NONE",
		altsyncram_component.outdata_reg_b = "CLOCK0",
		altsyncram_component.power_up_uninitialized = "FALSE",
		altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
		altsyncram_component.widthad_a = 4,
		altsyncram_component.widthad_b = 4,
		altsyncram_component.width_a = 16,
		altsyncram_component.width_b = 16,
		altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC "0"
// Retrieval info: PRIVATE: CLRq NUMERIC "0"
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRrren NUMERIC "0"
// Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRwren NUMERIC "0"
// Retrieval info: PRIVATE: Clock NUMERIC "0"
// Retrieval info: PRIVATE: Clock_A NUMERIC "0"
// Retrieval info: PRIVATE: Clock_B NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
```

```
// Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_B"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MEMSIZE NUMERIC "256"
// Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING ""
// Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "2"
// Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "2"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "1"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
// Retrieval info: PRIVATE: REGrren NUMERIC "1"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
// Retrieval info: PRIVATE: VarWidth NUMERIC "0"
// Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "16"
// Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "16"
// Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "16"
// Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "16"
// Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "0"
// Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: ADDRESS_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16"
// Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "16"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "DUAL_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS STRING "DONT_CARE"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "4"
// Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "4"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "16"
// Retrieval info: CONSTANT: WIDTH_B NUMERIC "16"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data 0 0 16 0 INPUT NODEFVAL "data[15..0]"
// Retrieval info: USED_PORT: q 0 0 16 0 OUTPUT NODEFVAL "q[15..0]"
// Retrieval info: USED_PORT: rdaddress 0 0 4 0 INPUT NODEFVAL "rdaddress[3..0]"
// Retrieval info: USED_PORT: wraddress 0 0 4 0 INPUT NODEFVAL "wraddress[3..0]"
// Retrieval info: USED_PORT: wren 0 0 0 0 INPUT GND "wren"
// Retrieval info: CONNECT: @address_a 0 0 4 0 wraddress 0 0 4 0
// Retrieval info: CONNECT: @address_b 0 0 4 0 rdaddress 0 0 4 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 16 0 data 0 0 16 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
// Retrieval info: CONNECT: q 0 0 16 0 @q_b 0 0 16 0
```

```
// Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL newhRam.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL newhRam_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL newhRam_bb.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

```
/*
 * singlePoint.v - the top level module to get a proof the hw works in serial
 * Brian Kuhn, Brian Dang and Kyle Heien - Shallow Water Simulation - EEC 181
 * May 2019
 */

`define H 2'b00
`define HU 2'b01
`define HV 2'b10

`define X 1'b0
`define Y 1'b1

// constants and computation parameters
`define HALO 2'd2
`define GRAVITY 13'b0010011100111// 9.80616 in decimal Q6.7 format - 13'b001001_1100111

`define CONTROL_OFFSET 2'b00
`define MATRIX_A_ADDRESS_OFFSET 2'b01
`define MATRIX_B_ADDRESS_OFFSET 2'b10
`define MATRIX_RESULT_ADDRESS_OFFSET 2'b11

module singlePoint #()(
    // signals to connect to an Avalon clock source interface
    clk,
    reset,

    // signals to connect to an Avalon-MM slave interface
    slave_address,
    slave_read,
    slave_write,
    slave_readdata,
    slave_writedata,
    slave_byteenable
);

    parameter DATA_WIDTH = 16;

    input clk,
    reg start,
    input reset,

        // slave interface
    input [9:0] slave_address;
    input slave_read;
    input slave_write;
    output reg [DATA_WIDTH-1:0] slave_readdata;
    input [DATA_WIDTH-1:0] slave_writedata;
    input [(DATA_WIDTH/8)-1:0] slave_byteenable;

    input [12:0] h_0_in,//hio,jo)
    input [12:0] h_1_in_x,//(i0-1)
    input [12:0] h_2_in_x,//(io-2)
    input [12:0] h_f_in_x,//(io+1)
    input [12:0] h_1_in_y,
    input [12:0] h_2_in_y,
    input [12:0] h_f_in_y,
    input [12:0] hu_0_in,
    input [12:0] hu_1_in_x,
    input [12:0] hu_2_in_x,
    input [12:0] hu_f_in_x,
    input [12:0] hu_1_in_y,
    input [12:0] hu_2_in_y,
    input [12:0] hu_f_in_y,
    input [12:0] hv_0_in,
    input [12:0] hv_1_in_x,
    input [12:0] hv_2_in_x,
    input [12:0] hv_f_in_x,
    input [12:0] hv_1_in_y,
```

```
    input [12:0] hv_2_in_y,
    input [12:0] hv_f_in_y,
    input [7:0] num_gridpoints, // max 256
    input [9:0] num_timesteps, // max 1024
    input [31:0] t_final_in,
    input [12:0] dt_in,
    input [12:0] dx_in,
    output reg done,
    output reg signed [12:0]    test

    // data matrices - MIGHT NEED TO GET REWORKED WITH BUS STUFF
    reg signed [12:0] h [6:0]; // i-2, i-1, i+1, 0, j-2, j-2, j+1
    reg signed [12:0] hu [6:0];
    reg signed [12:0] hv [6:0];
    reg signed [12:0] h_old [6:0];
    reg signed [12:0] hu_old [6:0];
    reg signed [12:0] hv_old [6:0];

    // intermediate data
    wire signed [25:0] h_flux_left, h_flux_right, hu_flux_left, hu_flux_right,
    hv_flux_left, hv_flux_right;
    wire signed [12:0] h_left, h_right, hu_left, hu_right, hv_left, hv_right;
    wire signed [12:0] Fh_x, Fhu_x, Fhv_x, Fh_y, Fhu_y, Fhv_y;

    // all the start control signals
    reg start_boundCon;
    reg start_calcXFlux, start_calcYFlux;
    reg start_upXFlux, start_upYFlux;

    // all the done control signals
    wire done_boundCon_h, done_boundCon_hu, done_boundCon_hv;
    wire done_flux_left;
    wire done_fh_x, done_fhu_x, done_fhv_x, done_fh_y, done_fhu_y, done_fhv_y;
    wire done_uph_x, done_uphu_x, done_uphv_x, done_uph_y, done_uphu_y, done_uphv_y;

    // other control signals
    reg update_half; // for UpdateFluxes | 0 = whole, 1 = half

    // buffer signals
    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;

    reg [9:0] timestep;
    reg [31:0] t_final, t_current;
    reg [31:0] T [NT-1:0];
    reg [12:0] dt; // not sure what the bounds are for this

    // MIGHT NEED TO GET REWORKED WITH BUS STUFF
    /*ApplyBoundaryConditions h_bound (.input_index(`H), .n_res(num_gridpoints),
    .halo(`HALO), .start(start_boundCon), .clk(clk), .done(done_boundCon_h));
    ApplyBoundaryConditions hu_bound (.input_index(`HU), .n_res(num_gridpoints),
    .halo(`HALO), .start(start_boundCon), .clk(clk), .done(done_boundCon_hu));
    ApplyBoundaryConditions hv_bound (.input_index(`HV), .n_res(num_gridpoints),
    .halo(`HALO), .start(start_boundCon), .clk(clk), .done(done_boundCon_hv));
    */
    // MIGHT NEED TO GET REWORKED WITH BUS STUFF - not sure where the inputs should be
    coming from
    // calculate fluxea

    hRAM    matrixH(
        .clock ( clk),
        .data ( slave_writedata ),
        .rdaddress (rd_addrA),
        .wraddress (slave_address[3:0]),
        .wren (matrixA_wren),
        .q (matrixA_dout)
    );

    hRAM    matrixHu(
```

```
        .clock ( clk),
        .data ( slave_writedata ),
        .rdaddress (rd_addrB),
        .wraddress (slave_address[3:0]),
        .wren (matrixB_wren),
        .q (matrixB_dout)
    );

    hRAM    matrixHv(
        .clock ( clk),
        .data ( slave_writedata ),
        .rdaddress (rd_addrC),
        .wraddress (slave_address[3:0]),
        .wren (matrixB_wren),
        .q (matrixB_dout)
    );

    hRAM    matrixResult(
        .clock ( clk),
        .data (vectorSum),
        .rdaddress (slave_address[3:0]),
        .wraddress (wr_addr4_5),
        .wren (wr_en4_5),
        .q (matrixResult_dout)
    );

    CalcFluxes XFluxH (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
        .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
        .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
        .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fh_x), .done(done_fh_x)
    );
    CalcFluxes XFluxHU (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
        .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
        .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
        .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhu_x), .done(done_fhu_x)
    );
    CalcFluxes XFluxHV (.clk(clk), .start(start_calcXFlux), .xOrY(`X),
        .h_0(h[3]), .h_1(h[1]), .h_2(h[0]), .h_f(h[2]),
        .hu_0(hu[3]), .hu_1(hu[1]), .hu_2(hu[0]), .hu_f(hu[2]),
        .hv_0(hv[3]), .hv_1(hv[1]), .hv_2(hv[0]), .hv_f(hv[2]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhv_x), .done(done_fhv_x)
    );

    CalcFluxes YFluxH (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
        .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[7]),
        .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[7]),
        .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[7]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fh_y), .done(done_fh_y)
    );
    CalcFluxes YFluxHU (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
        .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[7]),
        .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[7]),
        .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[7]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhu_y), .done(done_fhu_y)
    );
    CalcFluxes YFluxHV (.clk(clk), .start(start_calcYFlux), .xOrY(`Y),
        .h_0(h[3]), .h_1(h[5]), .h_2(h[4]), .h_f(h[7]),
        .hu_0(hu[3]), .hu_1(hu[5]), .hu_2(hu[4]), .hu_f(hu[7]),
        .hv_0(hv[3]), .hv_1(hv[5]), .hv_2(hv[4]), .hv_f(hv[7]),
        .halo(`HALO), .gravity(`GRAVITY),
        .F(Fhv_y), .done(done_fhv_y)
    );
```

```
	// update fluxes
	UpdateFluxes XUpH (.clk(clk), .start(start_upXFlux), .half(update_half),
		.h_0 (h[3]), .h_1(h[1]),
		.dt(dt_in), .dx(dx_in), .f_h(Fh_x),
		.h_0_new(h_0_new), .h_1_new(h_1_new_x),
		.done(done_uph_x)
	);
	UpdateFluxes XUpHU (.clk(clk), .start(start_upXFlux), .half(update_half),
		.h_0 (hu[3]), .h_1(hu[1]),
		.dt(dt_in), .dx(dx_in), .f_h(Fh_x),
		.h_0_new(hu_0_new), .h_1_new(hu_1_new_x),
		.done(done_uphu_x)
	);
	UpdateFluxes XUpHV (.clk(clk), .start(start_upXFlux), .half(update_half),
		.h_0 (hv[3]), .h_1(hv[1]),
		.dt(dt_in), .dx(dx_in), .f_h(Fh_x),
		.h_0_new(hv_0_new), .h_1_new(hv_1_new_x),
		.done(done_uphv_x)
	);

	UpdateFluxes YUpH (.clk(clk), .start(start_upYFlux), .half(update_half),
		.h_0 (h[3]), .h_1(h[5]),
		.dt(dt_in), .dx(dx_in), .f_h(Fh_y),
		.h_0_new(foobar), .h_1_new(h_1_new_y),
		.done(done_uph_y)
	);
	UpdateFluxes YUpHU (.clk(clk), .start(start_upYFlux), .half(update_half),
		.h_0 (hu[3]), .h_1(hu[5]),
		.dt(dt_in), .dx(dx_in), .f_h(Fhu_y),
		.h_0_new(foobar), .h_1_new(hu_1_new_y),
		.done(done_uphu_y)
	);
	UpdateFluxes YUpHV (.clk(clk), .start(start_upYFlux), .half(update_half),
		.h_0 (h[3]), .h_1(hv[5]),
		.dt(dt_in), .dx(dx_in), .f_h(Fhv_y),
		.h_0_new(foobar), .h_1_new(hv_1_new_y),
		.done(done_uphv_y)
	);
	assign done_boundCon_h=start;
	assign done_boundCon_hu=start;
	assign done_boundCon_hv=start;
	initial begin
		// initialize start signals
			start_boundCon = 0;
			start_calcXFlux = 0;
			start_calcYFlux = 0;
			start_upXFlux = 0;
			start_upYFlux = 0;

		// set done to 0
			done = 0;

		// initialize data
			h[0] = h_2_in_x;
			h[1] = h_1_in_x;
			h[2] = h_f_in_x;
			h[3] = h_0_in;
			h[4] = h_2_in_y;
			h[5] = h_1_in_y;
			h[6] = h_f_in_y;

			hu[0] = hu_2_in_x;
			hu[1] = hu_1_in_x;
			hu[2] = hu_f_in_x;
			hu[3] = hu_0_in;
			hu[4] = hu_2_in_y;
			hu[5] = hu_1_in_y;
			hu[6] = hu_f_in_y;
```

```

                hv[0] = hv_2_in_x;
                hv[1] = hv_1_in_x;
                hv[2] = hv_f_in_x;
                hv[3] = hv_0_in;
                hv[4] = hv_2_in_y;
                hv[5] = hv_1_in_y;
                hv[6] = hv_f_in_y;
        end

        // run the computation - MIGHT NEED TO GET REWORKED WITH BUS STUFF
        always @(posedge clk) begin
            // apply boundary conditions
            start_boundCon = 1;
            test=h[0];

            if (start==0)begin
                start_boundCon = 0;
                start_calcXFlux = 0;
                start_calcYFlux = 0;
                start_upXFlux = 0;
                start_upYFlux = 0;

            // set done to 0
                done = 0;

            // initialize data
                h[0] = h_2_in_x;
                h[1] = h_1_in_x;
                h[2] = h_f_in_x;
                h[3] = h_0_in;
                h[4] = h_2_in_y;
                h[5] = h_1_in_y;
                h[6] = h_f_in_y;

                hu[0] = hu_2_in_x;
                hu[1] = hu_1_in_x;
                hu[2] = hu_f_in_x;
                hu[3] = hu_0_in;
                hu[4] = hu_2_in_y;
                hu[5] = hu_1_in_y;
                hu[6] = hu_f_in_y;

                hv[0] = hv_2_in_x;
                hv[1] = hv_1_in_x;
                hv[2] = hv_f_in_x;
                hv[3] = hv_0_in;
                hv[4] = hv_2_in_y;
                hv[5] = hv_1_in_y;
                hv[6] = hv_f_in_y;
            end


            // store old variables
            if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv) begin
                start_boundCon = 0;
                h_old[0] = h[0];
                h_old[1] = h[1];
                h_old[2] = h[2];
                h_old[3] = h[3];
                h_old[4] = h[4];
                h_old[5] = h[5];
                h_old[6] = h[6];

                hu_old[0] = hu[0];
                hu_old[1] = hu[1];
                hu_old[2] = hu[2];
                hu_old[3] = hu[3];
                hu_old[4] = hu[4];
```

```
                hu_old[5] = hu[5];
                hu_old[6] = hu[6];

                hv_old[0] = hv[0];
                hv_old[1] = hv[1];
                hv_old[2] = hv[2];
                hv_old[3] = hv[3];
                hv_old[4] = hv[4];
                hv_old[5] = hv[5];
                hv_old[6] = hv[6];
                // calculate fluxes (predictor step)
                start_calcXFlux = 1;
                start_calcYFlux = 1;
            end

            // update half a timestep
            if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
            done_fhv_y) begin
                start_calcXFlux = 0;
                start_calcYFlux = 0;
                update_half = 1;
                start_upXFlux = 1;
                start_upYFlux = 1;
            end

            // apply boundary conditions
            if (done_uph_x && done_uphu_x && done_uphv_x && done_uph_y && done_uphu_y &&
            done_uphv_y) begin
                start_upXFlux = 0;
                start_upYFlux = 0;
                h[1] = h_1_new_x;
                h[3] = h_0_new;
                h[5] = h_1_new_y;
                hu[1] = hu_1_new_x;
                hu[3] = hu_0_new;
                hu[5] = hu_1_new_y;
                hv[1] = hv_1_new_x;
                hv[3] = hv_0_new;
                hv[5] = hv_1_new_y;
                start_boundCon = 1;
            end

            // calculate new fluxes
            if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv && update_half) begin
                start_calcXFlux = 1;
                start_calcYFlux = 1;
            end

            // update a full timestep
            if (done_fh_x && done_fhu_x && done_fhv_x && done_fh_y && done_fhu_y &&
            done_fhv_y && update_half) begin
                start_calcXFlux = 0;
                start_calcYFlux = 0;
                h[0] = h_old[0];
                h[1] = h_old[1];
                h[2] = h_old[2];
                h[3] = h_old[3];
                h[4] = h_old[4];
                h[5] = h_old[5];
                h[6] = h_old[6];

                hu[0] = hu_old[0];
                hu[1] = hu_old[1];
                hu[2] = hu_old[2];
                hu[3] = hu_old[3];
                hu[4] = hu_old[4];
                hu[5] = hu_old[5];
                hu[6] = hu_old[6];

```

```
            hv[0] = hv_old[0];
            hv[1] = hv_old[1];
            hv[2] = hv_old[2];
            hv[3] = hv_old[3];
            hv[4] = hv_old[4];
            hv[5] = hv_old[5];
            hv[6] = hv_old[6];
            update_half = 0;
            start_upXFlux = 1;
            start_upYFlux = 1;
        end

        if (done_boundCon_h && done_boundCon_hu && done_boundCon_hv && !update_half)
        begin
            done = 1;
        end
    end

    /*always @(h_old or hu_old or hv_old) begin
        // calculate X fluxes (predictor step)
        start_calcXFlux = 1;
        start_calcYFlux = 1;
    end*/

endmodule
```

```
/*
 * CalcFluxes.v - calculates the flux of a point based off its current, h, U and V values
 * Kyle Heien and Brian Kuhn - Shallow Water Simulations - EEC 181
 * April 2019
 */

module CalcFluxes(clk, gravity, start, xOrY, h_0, h_1, h_2, h_f, hu_0, hu_1, hu_2,
hu_f, hv_0, hv_1, hv_2, hv_f, halo, F, done);
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;
input signed [12:0] gravity;
input [1:0] halo;
input signed start;
input clk;
input xOrY; // 0 for x and 1 for y
output reg  done;
output reg signed [12:0] F;

initial done =0;

wire signed [12:0] h_left, h_right, hu_left, hv_right; // might need to be 14 bits
wire signed [12:0] max_wave_speed;

wire signed [12:0] hu_right, hv_left;
reg signed [12:0] hx_left, hx_right;

wire done_l, done_r, done_mws;

wire signed [25:0] h_flux_left, h_flux_right, hu_flux_left, hu_flux_right,
hv_flux_left, hv_flux_right;

sideStates hLeft (.h_0(h_0), .h_1(h_1), .h_2(h_2), .sideState(h_left));
sideStates huLeft (.h_0(hu_0), .h_1(hu_1), .h_2(hu_2), .sideState(hu_left));
sideStates hvLeft (.h_0(hv_0), .h_1(hv_1), .h_2(hv_2), .sideState(hv_left));

sideStates hRight (.h_0(h_1), .h_1(h_0), .h_2(h_f), .sideState(h_right));
sideStates huRight (.h_0(hu_1), .h_1(hu_0), .h_2(hu_f), .sideState(hu_right));
sideStates hvRight (.h_0(hv_1), .h_1(hv_0), .h_2(hv_f), .sideState(hv_right));

CalcSideFluxes flux_left (.start(start),.h_side(h_left), .hu_side(hu_left),
.hv_side(hv_left), .clk(clk), .gravity(gravity),
    .h_flux_side(h_flux_left), .hu_flux_side(hu_flux_left),
    .hv_flux_side(hv_flux_right), .done(done_l)
);

CalcSideFluxes flux_right (.start(start),.h_side(h_right), .hu_side(hu_right),
.hv_side(hv_right), .clk(clk), .gravity(gravity),
    .h_flux_side(h_flux_right), .hu_flux_side(hu_flux_right),
    .hv_flux_side(hv_flux_right), .done(done_r)
);

MaxWaveSpeed mws (.clk(clk), .start(done_l && done_r), .gravity(gravity),
    .h_left(h_left), .h_right(h_right), .hv_left(hx_left), .hv_right(hx_right),
    .max_wave_speed(max_wave_speed),
    .done(done_mws)
);

always @(*) begin
    if (xOrY == 0) begin
        hx_left = hu_left;
        hx_right = hu_left;
    end
    if (xOrY == 1) begin
        hx_left = hv_left;
        hx_right = hv_right;
    end
end

always @(done_l && done_r && done_mws) begin
    F=(h_flux_left+h_flux_right)>>1-(h_right-h_left)*max_wave_speed>>1;
```

```
end

always @(*) begin
    if (done_l==1&&done_mws==1&&done_r==1) begin
        done =1;
    end
    else if(start==0 || done_l==0|| done_mws==0 || done_r==0) begin
        done=0;
    end


end
endmodule
```

```
/*
 * sideStates.v - calculates h*_left and h*_right, depending on input
 * Brian Kuhn - Shallow Water Simulations - EEC 181
 * April 2019
 */

 // number of bits for input/output

module sideStates (
    input signed [12:0] h_0,
    input signed [12:0] h_1,
    input signed [12:0] h_2,
    output reg signed [12:0] sideState // might need to be [N:0] instead
    );

    wire [2:0] q = 3'b0_01; // q  == quarter == 0.25

    always @(*) begin
        sideState = ((q*h_0) + h_1) - (q*h_2);
    end
endmodule
```

```
/*
 * CalcSideFluxes.v - calculates h*_flux_left and h*_flux_right, depending on input
 * Brian Kuhn - Shallow Water Simulations - EEC 181
 * April 2019
 */

 // number of bits for input/output

 module CalcSideFluxes (
    input signed [12:0] h_side,
    input signed [12:0] hu_side,
    input signed [12:0] hv_side,
    input clk,
    input signed [12:0] gravity,
    input start,
    output reg signed [25:0] h_flux_side,
    output reg signed [25:0] hu_flux_side,
    output reg signed [25:0] hv_flux_side,
    output reg done
    );

    initial done =0;

    wire [12:0] qu, qv;
    wire done_u, done_v;

    divide div_u(
        .dividend(hu_side), .divisor(h_side),
        .start(start), .quotient(qu),
        .complete(done_u),
        .clk(clk)
    );

    divide div_v(
        .dividend(hv_side), .divisor(h_side),
        .start(start), .quotient(qv),
        .complete(done_v),
        .clk(clk)
    );

    wire [1:0] half = 2'b0_1; // 0.5
    //wire [10:0] gravity = 13'b001001_1100111 // 9.8046875

    always @(posedge clk) begin
        h_flux_side = hu_side;
        hu_flux_side = hu_side*qu + half*gravity*h_side*h_side;
        hv_flux_side = hu_side*qv;
    end

    always @(*) begin
        if (done_v==1&&done_u==1) begin
            done =1;
        end
        else if(start==0) begin
            done=0;
        end

    end

endmodule
```

```verilog
/*
 * MaxWaveSpeed.v - calculates maximum wave speed for a height with velocity
 * Kyle Heien - Shallow Water Simulations - EEC 181
 * April 2019
 */

module MaxWaveSpeed(clk, hv_left, hv_right, h_left, h_right, gravity,
max_wave_speed,done, start);
input signed [12:0] hv_left;
input signed [12:0] h_left;
input signed [12:0] hv_right;
input signed [12:0] h_right;
input signed [12:0] gravity;
input start;
output reg signed [12:0] max_wave_speed;
output reg done;
input clk;

wire signed [12:0] quotient_left;
wire signed [12:0] quotient_right;
wire signed [12:0] sqrt_out;
wire  done1;
wire  done2;
wire done3;
reg signed [12:0] sqrtinput;
reg signed [12:0] sqrtinput1;
reg signed [12:0] quotient_out;
reg signed [12:0]  quotient_out1;

initial done =0;
divide lut (
        .start(start),
        .quotient(quotient_left),
        .dividend(hv_left),
        .divisor(h_left),
        .clk(clk),
        .complete(done1)
);
divide rut (
        .start(start),
        .quotient(quotient_right),
        .dividend(hv_right),
        .divisor(h_right),
        .clk(clk),
        .complete(done2)
);

sqrt sut(
    .clk(clk),
    .start(done2&&done1),
    .data(sqrtinput),
    .answer(sqrt_out),
    .done(done3)
);

initial begin
done=0;
end



always @(*)begin
    sqrtinput1=gravity*(h_left+h_right)>>1;
    quotient_out1=(quotient_right+quotient_left)>>1;
    max_wave_speed=quotient_out+sqrt_out;
    if (sqrtinput1[12]==1) begin
        sqrtinput=~sqrtinput1+1;
    end
    else begin
```

```
            sqrtinput=sqrtinput1;
        end
        if (quotient_out1[12]==1) begin
            quotient_out=~quotient_out1+1;
        end
        else begin
            quotient_out=quotient_out1;
        end

    end


    always @(*) begin
        if (done3==1) begin
            done =1;
        end
        else if(start==0) begin
            done=0;
        end

    end




    endmodule
```

```
/*
 * divide.v - based off the qdiv.v file, however converts the inputs and outputs from 2's copliment to signed magnitude
 * Kyle Heien - Shallow Water Simulations - EEC 181
 * April 2019
 */

module divide (
input signed [12:0] dividend,
input signed [12:0] divisor,
input  start,
input  clk,
output reg signed [12:0] quotient,
output reg complete);


reg [13:0] i_dividend;
reg [13:0] i_divisor;
wire [13:0] o_quotient_out;
wire done;
initial complete =0;

qdiv but (
        .i_dividend(i_dividend),
        .i_divisor(i_divisor),
        .i_start(start),
        .i_clk(clk),
        .o_quotient_out(o_quotient_out),
        .o_complete(done),
        .o_overflow(o_overflow));

always @(*) begin
    if (dividend[12]==1)begin
        i_dividend[12:0]= ~dividend+1;
        i_dividend[13]=1;
    end

    else begin
        i_dividend=dividend;
    end

    if (divisor[12]==1)begin
        i_divisor[12:0]= ~divisor+1;
        i_divisor[13]=1;
    end

    else begin
        i_divisor=divisor;
    end
    if (o_quotient_out[13]==1)begin
        quotient=~o_quotient_out[12]-1;
    end

    else begin
        quotient=o_quotient_out;
    end
end

always @(*)begin
    if (start==1&&done==0)begin
        complete=1;
    end
    else begin
        complete=0;
    end

end
endmodule
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:             Burke
// Engineer:            Tom Burke
//
// Create Date:     19:39:14 08/24/2011
// Design Name:
// Module Name:     qdiv.v
// Project Name:        Fixed-point Math Library (Verilog)
// Target Devices:
// Tool versions:       Xilinx ISE WebPack v14.7
// Description:         Fixed-point division in (Q,N) format
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
//  Revision 0.02 - 25 May 2014
//                              Updated to fix an error
//
// Additional Comments: Based on my description on youtube:
//          http://youtu.be/TEnaPMYiuR8
//
//////////////////////////////////////////////////////////////////////////////////

module qdiv #(
    //Parameterized values
    parameter Q = 7,
    parameter N = 14
    )
    (
    input   [N-1:0] i_dividend,
    input   [N-1:0] i_divisor,
    input   i_start,
    input   i_clk,
    output  [N-1:0] o_quotient_out,
    output  o_complete,
    output  o_overflow
    );

    reg [2*N+Q-3:0] reg_working_quotient;   //  Our working copy of the quotient
    reg [N-1:0]         reg_quotient;               //  Final quotient
    reg [N-2+Q:0]       reg_working_dividend;   //  Working copy of the dividend
    reg [2*N+Q-3:0] reg_working_divisor;        // Working copy of the divisor

    reg [N-1:0]             reg_count;      //  This is obviously a lot bigger than it needs to be, as we only need
                                                //      count to N-1+Q but, computing that number of bits requires a
                                                //      logarithm (base 2), and I don't know how to do that in a
                                                //      way that will work for everyone

    reg                 reg_done;           //  Computation completed flag
    reg                 reg_sign;           //  The quotient's sign bit
    reg                 reg_overflow;       //  Overflow flag

    initial reg_done = 1'b1;                //  Initial state is to not be doing anything
    initial reg_overflow = 1'b0;            //      And there should be no woverflow present
    initial reg_sign = 1'b0;                //      And the sign should be positive

    initial reg_working_quotient = 0;
    initial reg_quotient = 0;
    initial reg_working_dividend = 0;
    initial reg_working_divisor = 0;
    initial reg_count = 0;
```

```
    assign o_quotient_out[N-2:0] = reg_quotient[N-2:0]; //  The division results
    assign o_quotient_out[N-1] = reg_sign;                  //  The sign of the quotient
    assign o_complete = reg_done;
    assign o_overflow = reg_overflow;

    always @( posedge i_clk ) begin
        if( reg_done && i_start ) begin                             //  This is our startup condition
            //  Need to check for a divide by zero right here, I think....
            reg_done <= 1'b0;                                       //  We're not done
            reg_count <= N+Q-1;                                     //  Set the count
            reg_working_quotient <= 0;                              //  Clear out the quotient register
            reg_working_dividend <= 0;                              //  Clear out the dividend register
            reg_working_divisor <= 0;                               //  Clear out the divisor register
            reg_overflow <= 1'b0;                                   //  Clear the overflow register

            reg_working_dividend[N+Q-2:Q] <= i_dividend[N-2:0];             //  Left-align the dividend in its working register
            reg_working_divisor[2*N+Q-3:N+Q-1] <= i_divisor[N-2:0];     //  Left-align the divisor into its working register

            reg_sign <= i_dividend[N-1] ^ i_divisor[N-1];       //  Set the sign bit
            end
        else if(!reg_done) begin
            reg_working_divisor <= reg_working_divisor >> 1;    //  Right shift the divisor (that is, divide it by two - aka reduce the divisor)
            reg_count <= reg_count - 1;                             //  Decrement the count

            //  If the dividend is greater than the divisor
            if(reg_working_dividend >= reg_working_divisor) begin
                reg_working_quotient[reg_count] <= 1'b1;                        //  Set the quotient bit
                reg_working_dividend <= reg_working_dividend - reg_working_divisor;  //      and subtract the divisor from the dividend
                end

            //stop condition
            if(reg_count == 0) begin
                reg_done <= 1'b1;                                   //  If we're done, it's time to tell the calling process
                reg_quotient <= reg_working_quotient;           //  Move in our working copy to the outside world
                if (reg_working_quotient[2*N+Q-3:N]>0)
                    reg_overflow <= 1'b1;
                    end
            else
                reg_count <= reg_count - 1;
            end
        end
endmodule
```

```verilog
/*
 * sqrt.v
 * Brian Kuhn - Shallow Water Simulations - EEC 181
 * May 2019
*/

module sqrt #(parameter DATA_WIDTH = 13, ANSWER_WIDTH = 13, TRIAL_WIDTH = 8)(
    input clk,
    input start,
    input wire [DATA_WIDTH-1:0] data,
    output reg [ANSWER_WIDTH-1:0] answer,
    output done
);

    reg busy;
    reg [TRIAL_WIDTH-1:0] bit;
    wire [TRIAL_WIDTH-1:0] trial;
    assign trial = answer | (1 << bit);

    always @(posedge clk) begin
        if (busy) begin
            if (bit == 0)
                busy <= 0;
            else
                bit <= bit - 1;
            if (trial*trial <= data)
                answer <= trial;
        end else if (start) begin
            busy <= 1;
            answer <= 0;
            bit <= TRIAL_WIDTH - 1;
        end
    end

    assign done = ~busy;
endmodule
```

```verilog
/*
 * UpdateFluxes.v - calculates a new value(h,V or U) based of the old value and the flux
 * Kyle Heien - Shallow Water Simulations - EEC 181
 * April 2019
 */
module UpdateFluxes(
input clk,
input start,
input half,
input signed [12:0] h_0,
input signed [12:0] h_1,
input signed [7:0] dt,
input signed [12:0] dx,
input signed [12:0] f_h,
output reg signed [12:0] h_0_new,
output reg signed [12:0] h_1_new,
output reg done);
wire signed [12:0] quotient;
wire done1;
reg signed [12:0] dt1;
divide tut (
        .start(start),
        .quotient(quotient),
        .dividend(dt1),
        .divisor(dx),
        .complete(done1),
        .clk(clk)
    );
initial begin
    done=0;
    if (half==1) begin
        dt1=dt>>1;
    end
    if (half==0) begin
        dt1=dt;
    end
end


always @(*) begin
h_0_new=h_0+quotient*f_h;
h_1_new=h_1-quotient*f_h;
end

always @(*) begin
    if (start==1 && done1==1) begin
        done =1;
    end
    else begin
        done=0;
    end

end
endmodule
```