Nicholas Bannister, Vinay Chaudhary, Aaron Hoy, Charles Norman, Jeremy Weagley

Team VerchuCom
PPU + Demo
Final Report

What We Built

Game Description

Overall Implementation

Our original plan of action was to implement the game of Worms that utilized a customized Physics Processing Unit (PPU) built on the FPGA. We had planned to run the game on the Workstation, send objects back and forth to a shared bank of memory on the board, and update the objects' state using the PPU. However, we came to realize that Worms would not fully show off our main feature of the project, which is the PPU. Therefore, we decided to create a "demo" which basically consisted of a bunch of squares interacting with each other, wind resistance and gravity within a confined area. We then took it to the next level and implemented a simple 2-player shooter game. The object of the game is for each player to destroy all of their squares, either blue or square, before their opponent. Each person has unlimited regular ammo along with 2 "nuclear bombs," which destroy all of the bombs within a certain radius.

Physics Processing Unit (PPU)

We designed a specialized CPU (A PPU) that operates on structs representing 2 dimensional objects (position, velocity, acceleration, orientation, etc.). It modeled various particle effects, such as

- Model collisions between particles and elastic/inelastic collisions
- Effects of Gravity
- Wind Resistance

The accelerator sits in a basic iteration loop that applies each of the various effects on all the particles in memory, updating their positions, velocities, etc. based on physics. The CPU creates and destroys objects in the PPU's loop, and reads the positions and orientations of the objects for use in the game. All updates are to the shared memory (BRAMs).

Hardware Description

PPU Pipeline / Overall Architecture

We recently realized that there are some duplicate signals going through the pipeline. Namely, the pending values and the operational values are both originally taken from the pending data out of memory, and they are the same for several stages. I have not changed it yet, but those will be run down the pipeline only once until the point where they divide. Also, Charles is working on the overlap amount unit and the acceleration calculator (its continuation) as one superpipelined unit, so there will be one fewer major stages, and the stage resulting from the merge will have more minor stages in it. Also in the current verilog code, the next address generator is not finished. Since the object array is null terminated, we know to loop A and B around to the base address when they get a null in their Object ID field. We want to know about this as soon as possible, and give the memory the correct wrapped around address in time, so the address generator must be a mealy machine. Its current implementation will be changed shortly. The rest of the pipeline still works like in the diagram.

In the first major stage, we generate an address to put on the memory read port and on the next cycle we get the entire object out and run it through the breakdown module to separate it into the various values associated with the object. Having this module makes it very easy to change our word size or the memory ordering without affecting the rest of the PPU. Here we also generate a NOP signal based on whether or not the memory data is ready, and A or B address is wrapping around, etc.

The second and third major stages are now being combined into one larger stage. Here we have a superpipelined acceleration calculator which takes in the positions and orientations of A and B along with other properties and outputs the resulting accelerations on A (a fixed number of cycles later). In parallel with this, we have the global acceleration calculator, which calculates accelerations resulting from gravity, wind, and wind viscosity. On the final iteration of A, once it has been compared to all over objects, the global accelerations will be applied instead.

In the next stage, we have out accumulation units. This consists of the acceleration accumulator and the overstress detector. The acceleration accumulator keeps a running sum of all the accelerations on A as a result of any collisions it has with any of the other objects. The overstress detector keeps track of whether or not any of the collisions have overstressed the object, causing it to eventually be destroyed. These units are tricky in stalls and changes in A because they are state full (using pipeline registers), so we had to run some additional signals to make sure they stay correct in these situations.

In the next stage we compute all the time dependent updates. Basically, we update the position based on the velocity and elapsed time, and update the velocity based on the acceleration and elapsed time. In order to obtain the elapsed time, we include a timestamp (measured in cycles) with each objects data. In this stages, we get the new cycles count and do a subtraction to get the elapsed number of cycles from the last time A was updated. We also keep this time to writeback as A's new timestamp.

In the final stage we put all the properties of A back into one word and write it back to the memory. Although it is not shown on the diagram, the pipeline must actually stall when there is a write, because there is normally a read on every cycle, and the PPU is only getting one port of the dual ported memory controller. This is ok, however, because writes will happen an insignificant percentage of the time (once about every n cycles, where n is the number of objects in the array). This is the issue that the memory can only have one port that has the ability to write. In the instance that the PPU is writing and the PPC (ethernet code) is trying to write at the same time, the PPU gets priority and a write failure signal is sent to the PPC. This is done because it is easier to handle a failed write in software than in the PPU architecture.

Overlap Amount Calculator

In this module of the pipeline, 2 objects will be tested for a collision. We have decided to treat all objects as squares. The objects have x and y coordinates along with the length of their sides (all are equal since they are squares). The overall goal of this module will be to output the total area of overlap between the two objects, horizontal/vertical distances between the centers of the two objects, and the projected direction of horizontal/vertical acceleration. This is done explicitly through several subtractors and adders with some extra compare and absolute value logic based on custom bit manipulation.

Collision Acceleration Calculator

The collision acceleration calculator is a bigger module that relies on the overlap amount calculator. This module takes in the outputs from the overlap amount calculator along with the object's mass (this is inverted to eliminate the need for a division) and positional/size properties. There are three outputs that this module creates: acceleration of the object in the x direction, acceleration of the object in the y direction, and angular or theta acceleration of the object. We are using Newton's F=m*a formula to calculate acceleration. Therefore, in calculating the 3 different accelerations, we use $a=F*m^{-1}$. We therefore need to determine the "force" for each component.

The force for the x and y components are going to be modeled by Fx=a*b*dx*[elas_A+elas_B] and Fy=a*b*dy*[elas_A+elas_B], where a and b are dimensions of the overlap rectangle, elas_A and elas_B are constant properties of the objects that are given, and dx and dy are the horizontal and vertical distances between the center of the two objects. The angular force will be based on the same principles. However, now we will take Ftheta to be the greater of Fx and Fy. The only difference will be the direction. The angular acceleration must be negated due to convention. One last issue is saturation. Due to the multiplication of several numbers which are up to 16-bits long, the final values must be saturated. Therefore, in the end, the minimum acceleration will be -32,768 units and the maximum will be 32,767. These values will be interpreted and scaled appropriately by the game. At this point, the final outputs of this module should be ready to be passed on to the next stage of the pipeline.

Acceleration Accumulator

This module determines the new overall acceleration of the object based on it's "old" acceleration and the new acceleration based on a particular object/force. This is done through 3 parallel 16-bit adders and 3 parallel 16-bit muxes. Essentially the muxes are there so that junk data is not written into the accumulator when the next stage does not contain valid data.

Time-Dependent Update

The time independent update module takes the computations computed previously in the pipeline and actually applies it to the object. The "next state" of the object, most specifically the next position and velocity, will be updated. A simple subtraction is used to compute the total amount of cycles elapsed. This value is then multiplied by the velocity and acceleration components to achieve the next state position and velocity components, respectively. Once again, there is some custom bit manipulation that is done to assure the proper values are computed.

Global Effects Calculator

Along with the collision acceleration calculator, there is the global acceleration calculator which takes into account gravity, wind, and wind viscosity and sums them to create an overall acceleration due to natural elements. Linear acceleration (acc_x and acc_y) are created through simple computations with adding acceleration due to gravity (-9.8) and due to its current wind velocity. The acceleration due to the wind velocity will be scaled according to the wind viscosity and the height or width of the object (depending on which component of the acceleration is being calculated). All of these calculations will be done using several adders and 2 multipliers. The global angular acceleration will be calculated by summing all of the acceleration vectors and computing a very simple angle that will be one of 8 orientations. This will be done with 2 adders and a 8:1 demux.

Memory Controller

The memory module is implemented to allow communication between the Ethernet connection with the desktop workstation and the PPU. It is a fairly simple dual-port RAM with enables on each port to allow reading and writing. The word size is kept relatively large (256 bits) to allow all relevant data about an object to be read or written in a single clock cycle. Despite the large width, the memory is able to remain somewhat deep as well (~8000 words). Although this is a dual-port memory, due to technological limitations, only one port can write at a time, so an error is generated if the Ethernet and PPU both try to write at once. In these collision cases, the PPU is given priority. The memory is implemented in BRAMs; these block primitives allow very rapid, low latency memory accesses from within the FPGA fabric without having to deal with all of the issues involved in using the DRAM. As part of the PPU operation, the memory is traversed by the pipeline, with each item read, compared to every other item to determine collision interactions, and then updated in the memory. During the garbage collection phase implemented by the Power PC, the memory is compacted as objects that are flagged as destroyed are removed from it. This should allow the game to continually generate new objects without fear of overfilling the memory.

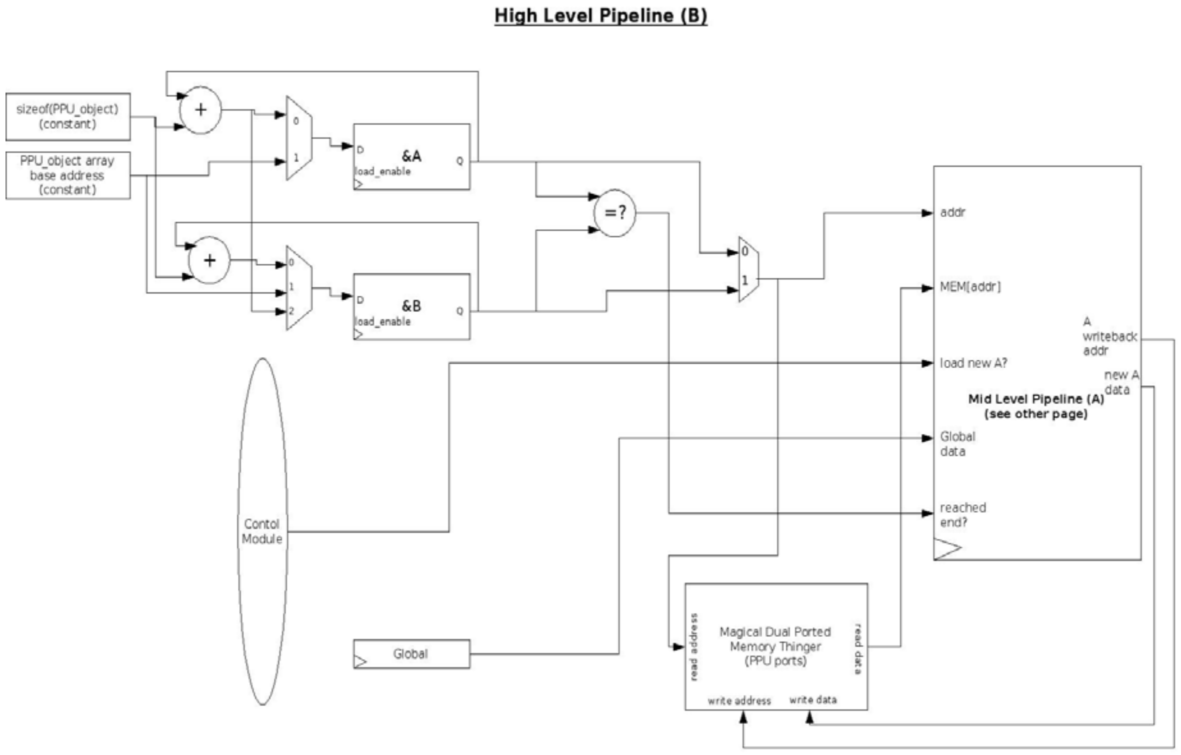
Variable Representation

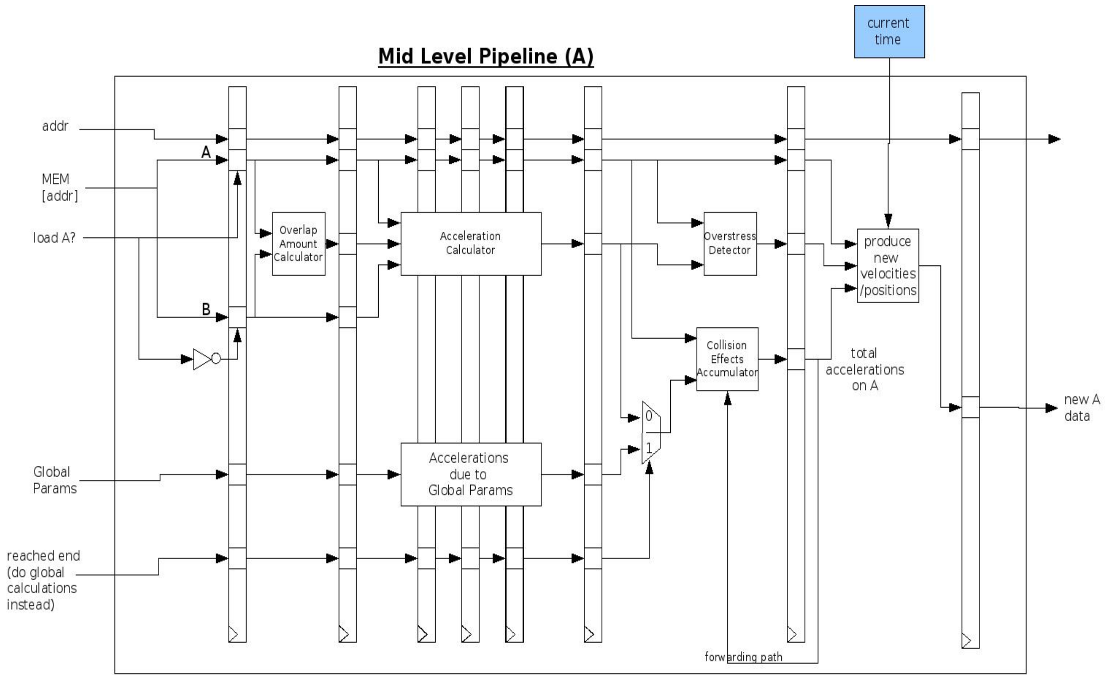
The following are the general fixed-point representations of the major variables in the physics computational portion of the pipeline:

	Fixed-point Representation	Format
x, y coordinates	8b.8b	unsigned
side length of square	5b	unsigned
a, b (amount overlap)	8b.8b	unsigned
dx, dy (separation distance)	8b.8b	unsigned
elas_A, elas_B, mass	4b.4b	unsigned

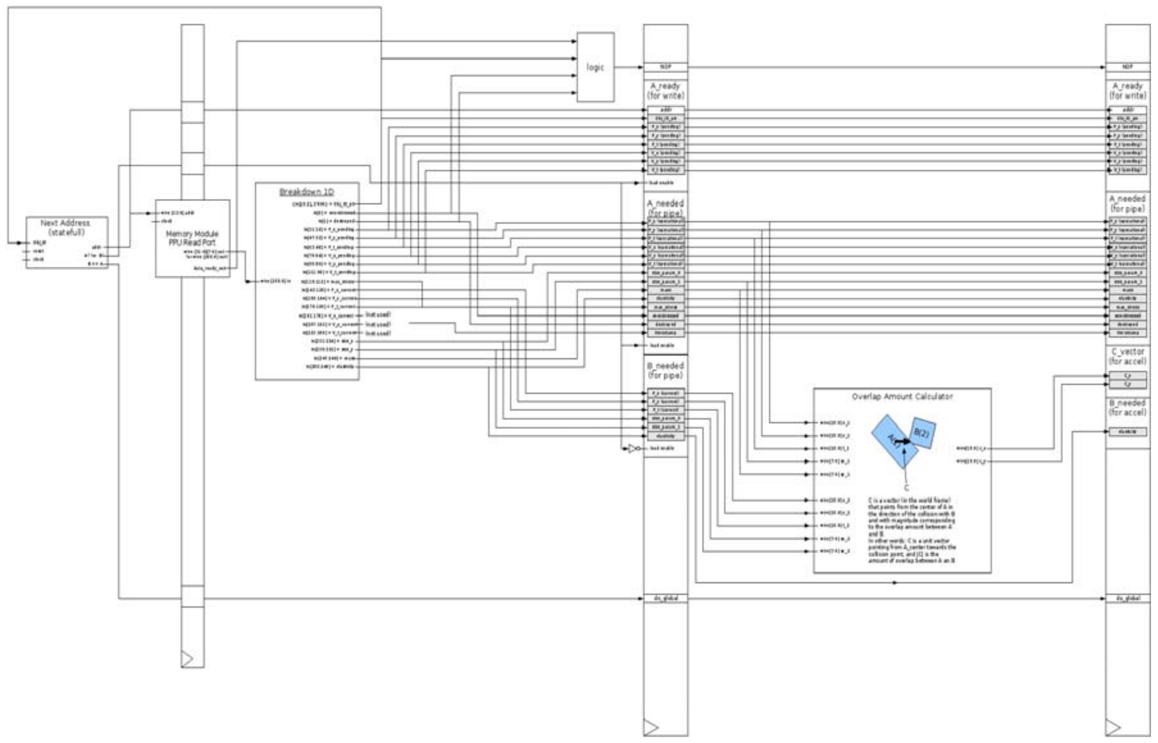
Time (cycles)	32b	unsigned
Velocity	8b.8b	2’s complement
Acceleration	16b	2’s complement

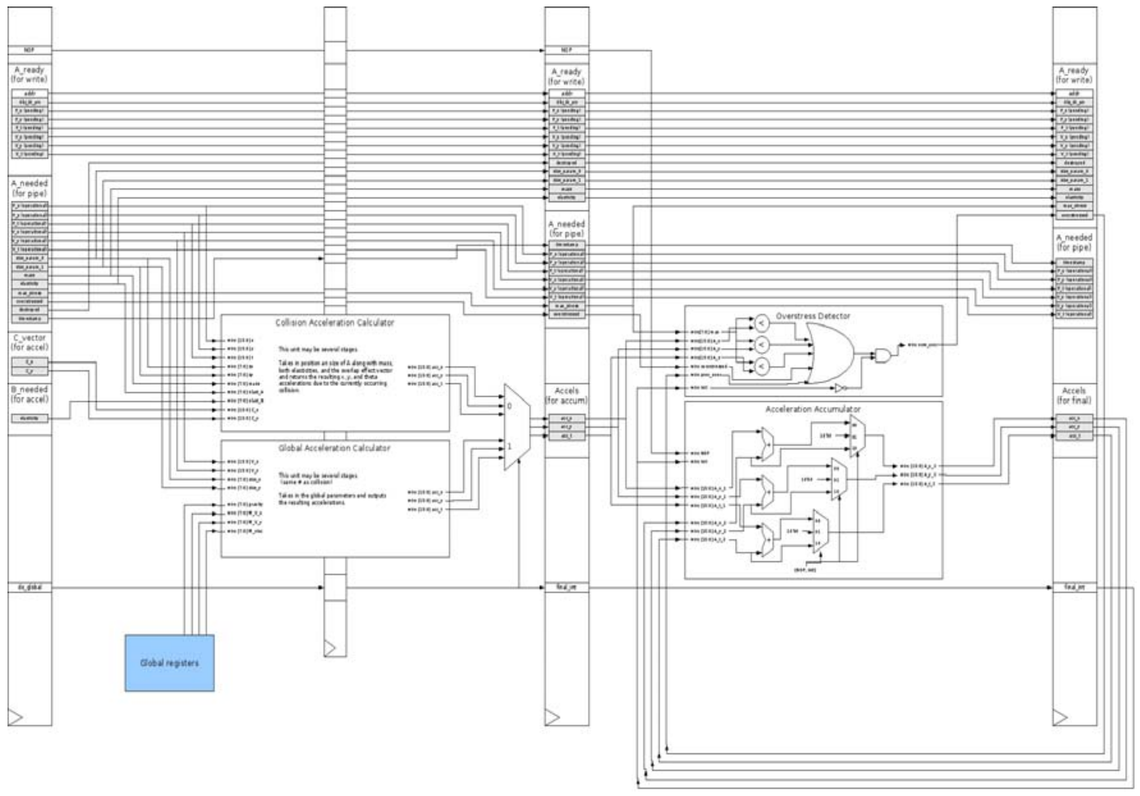
Architecture Diagrams

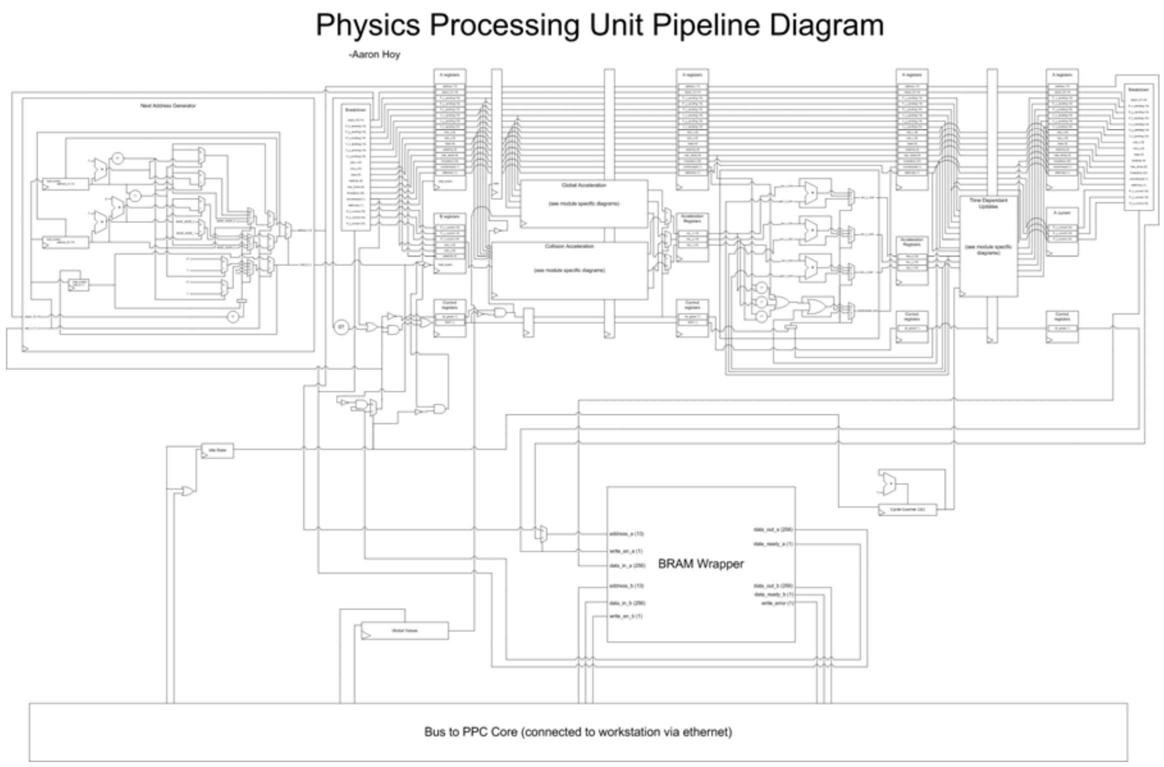



Mid Level Pipeline (A)
current time
addr
MEM [addr]
load A?
A
B
Global Params
reached end (do global calculations instead)
Overlap Amount Calculator
Acceleration Calculator
Accelerations due to Global Params
Overstress Detector
Collision Effects Accumulator
produce new velocities /positions
total accelerations on A
new A data
0
1
forwarding path

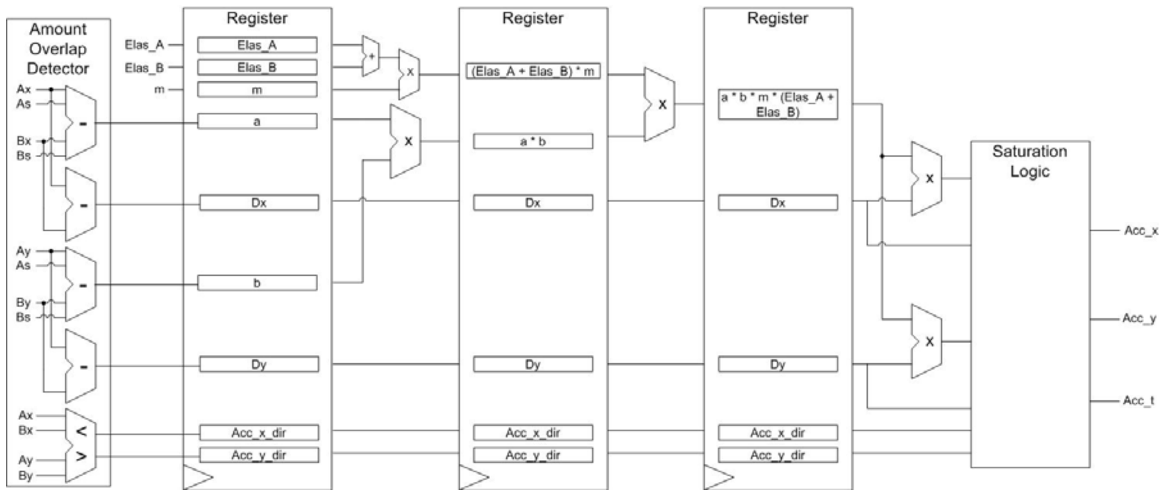
Architecture Diagrams – Stage by Stage






Physics Processing Unit Pipeline Diagram
-Aaron Hoy
Next Address Generator
Global Acceleration
(see module specific diagrams)
Collision Acceleration
(see module specific diagrams)
Time Dependant Updates
(see module specific diagrams)
Idle State
Cycle Counter (32)
BRAM Wrapper
Global Values
Bus to PPC Core (connected to workstation via ethernet)

Collision Acceleration Calculator



Software Description

Physics Demo

The demo starts with four rectangular bounding boxes. The idea of these is to force collisions of boxes by forcing them to bounce off 4 walls.

The idea of the demo was the show off everything the physics processors could do. The user had the ability to specify every aspect of the square "molecule", and add as many or as few of the objects as possible. The user could choose to add random objects with fixed size and mass (or not) as well as fine tune object creation. The user could control :

- Initial Position (X,Y, Theta)
- Initial Velocity (X,Y, Theta)
- Mass
- Elasticity (How "bouncy" the object was)
- Max stress (how much damage and object could take before it was destroyed)
- Size (square side size)

The user could also specify and change on the fly the following "global" params

- Gravity
- Wind viscosity (the "thickness" of the wind, how much it slow the objects down)
- Wind velocity (x,y)

The demo store no data locally, and instead sent add, delete, and request all packets to the board. If the user was not adding anything, the demo would just be constantly requesting all the data over Ethernet. It would then display the data it got, and ask over and over again.

Finally, the user could delete all objects, and start and stop the PPU. See the website for a video of the demo in action.

Interesting Demo Cases

- Add a bunch of boxes on top of each other. Turn on gravity. Fire one box going very fast at the tower of boxes. The boxes will accurately tumble to the ground.
- Add once large box to the center, and add 100 random boxes at various velocities (a one click process). Turn on wind velocity and watch boxes get stuck on the center box and try to get pushed past it.
- Add three boxes that form a canyon. Add a bunch of random boxes and turn on gravity. Boxes will get stuck in the canyon and bounce around. Turn on wind velocity and watch boxes move back and forth in the canyon.
- Combine all three of these for more fun and interesting possibilities.

Game modification

At the end, we converted the demo into a basic game (that ran after clicking START GAME) so that our group met all the requirements of the project. The game was a 2d box shooter. The objective was to destroy all the boxes that corresponded to your crosshair's color. The red player controlled his crosshair with the wasd keys and the blue player controlled his crosshair with the arrow keys. Both players had separate fire and giant gun buttons. Collision detection between the crosshair bullets and the squares was done in software. On collision detection, a delete packet was sent to the board, and the Power PC deleted the object from its buffer so it didn't cause collisions anymore. Once all of one players squares had been destroyed, the game ended, and the software runs in demo mode again. The game had a laser sound effect that occurred when either player fired his weapon, as well as some killer background music.

Ethernet Protocol and Board Server

Our original plan was to boot Linux onto the FPGA board and have the game completely run from the Linux OS on the board. In the end, we ended up having too many problems with linux, we could not get X11 to work because we could not get "make" and "gcc" installed on the board, plus the game uses many external libraries that we would also have to install from scratch.. The entire process of sitting and reading documentation on how to install compilers and all the required software that we needed proved to be too time consuming. To solve these problems, we decided to run the game on the workstation and have our game communicate through the Ethernet to our PPU.

We code a server onto the board using Xilinx sockets API and pass information to and from our game on a given socket. Our software team has had reasonable experience with network programming and has written servers before, so the transition to this new plan was fairly easy.

The basis for our network protocol is very simple. We arrange all the data that we needed into one of four types of packets:
REQUESTALL, ADD,DELETE and GC garbage collect). The game will assemble the data into a packet, tack a header that we specify onto it and send it off to the server. On the server side, depending on what the client requests, will perform the desired operation on our memory module, which in turn will update our BRAM.

Type	ADD	DELETE	REQUEST-ALL	GC
Size (bytes)	18	2	0 (Req) 8 (Resp)	0
Contents	Object ID Pos (X, Y, Angle) Vel (X, Y, Angle) Max Stress Elasticity	Object ID	Object ID Pos (X, Y, Angle)	Nothing

Another key design of the server is that we need it to utilize only the DRAM, because our memory module takes up about 90% of the BRAM and the 10% left for our server is not enough given that we request information every screen refresh/redraw.

How We Built It

Initially our design methodology was to completely diagram the design so that it could be easily partitioned and implemented. We divided the design into software and hardware portions, and assigned group members to each part. We created a huge top-level diagram of the pipeline and then slowly filled in detail to the point where each part of the design could be easily implemented. The software team looked through the open source Worms game and began to pull apart the physics computations in order to properly adapt it to the physics hardware. We made the decision to run the game on the work station and communicate with the hardware via ethernet, so then the software team began to work on the software for that. At this point the hardware team began to implement various parts of the physics pipeline, with different parts being assigned to each person so that work could proceed in parallel.

Once the individual modules in the pipeline were finished, it was time to implement them on the Xilinx board. A simple project had already been created in the Xilinx Platform Studio to implement the ethernet communication, and this was working well. The bus interface was then added along with the memory so that the software running on the PowerPC could communicate with it. We though that once this was working we would have only to attach the physics pipeline to the memory module and everything would work. However, we soon found that the Xilinx synthesizer would not properly or completely implement our design modules. At this point our careful design methodology fell to pieces and we began to try to debug the design. Since we were unaware where the problem was located, members of both the hardware and software teams worked on the debugging almost non-stop. A lot of the debugging became a sort of guess-and-check black box testing, since we couldn't see what was happening inside the board and we didn't know what was implemented and what wasn't. When we learned how to use the Chipscope tool this helped us greatly. We found that when certain nets were connected to the Chipscope, we would not only be able to see what was happening with the hardware, but we could also force the synthesis tool to implement the connected parts of the design. We began to add more and more things to the Chipscope in order to be sure our design was being implemented the way we wanted.

Even with the help of Chipscope, we soon found that the synthesizer was unable to deal with much of our design. A wise decision from the software team had given us a simple physics demo with which we could easily debug the design, and it showed us that our computations were still incorrect. We began to redesign the pipeline, using a much simpler design than we had originally used. Although much of the pipeline remained unchanged, the modules with the heaviest and most important computations had to be replaced with combinational modules that used simpler heuristics rather than the complete physical equations we intially planned to implement. Finally after much hacking we were able to get the pipeline working pretty well.

What We Learned

Most of what we learned in this project revolved around troubleshooting and debugging. Initially, we planned to have the entire design planned and diagrammed so that we would only have to implement it in Verilog, test it, and then connect it and load it onto the board. However, we had a lot of trouble with the Xilinx synthesis tools, so a very large part of the project became figuring out how to work with the less than optimal Xilinx tools. We even ended up having to greatly reduce the functionality of our design and abandon the idea of adapting our open source Worms game to work with our physics processor. So instead of learning a lot about digital design theory, we instead learned the vast limitations of FPGAs and their software tools. If we had know the difficulties that we would face in implementing our design on the FPGA at the beginning of the semester, our design would have been very different, and we would perhaps have been a little less ambitious in what we hoped to accomplish.

Despite the difficulties in implementation, we feel that our design flow was a good choice. Even though we ended up abandoning a lot of the initial design, the time we spent designing and diagramming helped us to better understand the problem of designing the physics processor and understand which parts of the design would require the most work. In this way we were able to segment the design fairly well and assign different group members to work on different parts of the project. Although at the end it came down to all of us working to debug the design, initially we were able to accomplish a lot in parallel, which was a good decision. Another good design choice which we made was to run the game/demo on the workstation instead of on the board. There is no telling how far we would have gotten if we had to debug the game on the PowerPC instead of on the workstation. Choosing to just send the data via Ethernet was a good decision. A bad decision of ours was probably the decision to try to adapt an existing game to our physics pipeline. While the game Worms utilized basic software physics that were easily changed to take advantage of our hardware, the very nature of the game itself did not fully utilize our physics engine. Perhaps if the Xilinx tools had not given us as much trouble as they did, we would have been able to modify the game to properly show off our physics hardware, but since debugging placed such a time constraint on us, we had to abandon the game. At this point, we made another of our good decisions: implementing a simple physics demo with a user interface so that we could add or remove boxes, adjust global parameters, etc. This demo proved invaluable in debugging and testing the design since we could easily see what effect the changes we were making in the hardware had. We were even able to adapt this demo into a simple game by the end of the semester.

As far as advice we can offer to future generations, we would mostly advise them to avoid the Xilinx tools altogether if possible. They are full of bugs, they are difficult to use, and they admittedly have important features missing or broken. It seemed in the class that those projects that had more software portions in their design had a better time of it, while designs like ours that were more hardware-based had a worse time simply because of the poor documentation and poor functionality of the Xilinx FPGAs and their software tools. In the future, more of the class should be devoted to familiarizing students with the limitations of the equipment so that less time will be spent having to figure it out, or perhaps better equipment should be acquired.

Team Contributions

Nicholas Bannister

Individual Contribution: Software Team

Planning: 10 weeks

Worms

Throughout this phase of the project I was primarily responsible in assisting the planning of the game and researching open source games that we could potentially use for our project. This task was not particularly easy because there are many open source games but only a handful that would even remotely demonstrate what we wanted to do with the physics processor. I the end, going with worms seemed like the best way to go.

Demo

When we finally came to the realization that worms was not going to work as we had planned we stepped into the planning phase of the physics demo. During this phase I was primarily responsible for the OpenGL design and coding of the demo, and helping the second software engineer (Vinay) in testing and debugging OpenGL problems.

PPU

For the PPU I assisted in the original design and architecture of the data structure that eventually held all of the physics objects. I.E. the array of objects that we traverse and compare with each other object, then update each object physics properties per cycle.

Changes

If I had to do this phase over again I would have not chosen to do and open source game and instead written a demo from the beginning and then turned the demon into a fully interactive game. I.E. simply have two modes that we had in the end.

Design/Testing: Till the end

Worms

I was not really involved in the design of worms. It was open source. I looked over a small fraction of the worm's source code but it was a pretty large program. I would say I saw and understood about 15% of the entire code base.

Demo

For the demo I was responsible for the graphics and in the end the testing of the demo. I assisted in the making of the final "game" by coding the functionality to target and destroy objects.

Lastly I helped in the coding of the starter code for the PPU server on the Xilinx board.

Changes

Once again I would scrap open source and go directly to demo. Secondly I would have like to design a much better game.

Final Demo: 1 week

Demo

For this I was responsible for the putting the slides together for the poster and talking about to anyone who wanted to know. Not many people asked about it however. It was more of a show them the demo, then walk away.

Over All Impressions: A Lifetime

I enjoyed the class and feel that interaction with a large/long group project will better prepare me for further projects now that I know what not to do. For improving the class: I don't know if there is anything, other than going to the 2.0 version, that can be really done to improve the class, everything is self defined and self paced so each group is responsible for their own little project world.

I also created the final website for the course.

Vinay Chaudhary

Overall my role for this project was that of a software engineer. I was primarily responsible for all software related issues. This included both software running on the power pc on the board, and demo code running on the workstation. I also worked on modifying worms to work with our physics processor, but abandoned it after I decided it would not be a good demo for the physics processor.

Besides the software component, I worked with Aaron in the initial design of the PPU. Together we came up with the "molecule" data structure and the array structure in which to store it. I also decided that we should just use an array for data storage, and once the array was filled, delete previously lazily deleted items. This is analogous to some managed languages "heap compaction" algorithms. The whole design process took about 10 hours a week for the first 4/5 weeks.

After this my primary role was design and implementation of the game and all the software on the board. For a few weeks, I worked on modifying worms to better show off our physics processor. The task proved more difficult than previously imagined. I actually did manage to factor out some of the physics, but some of the stuff was too hard-coded in software to work. There was another major problem in that there were very very few collisions occurring in the game, and our physics processor was primarily based on accurately modeling collisions. Worms would have served only to show off the global effects. Additionally, I realized that debugging the PPU on the board with only worms would be extremely difficult.

At this point I abandoned worms (after putting about 50 hours working on it :( ), and started working on our physics demo, called "Square molecules in a box". I was responsible for all aspects of the demo except for the graphics. The demo (excluding debugging) took about 20-40 hours to make and test.

After I had tested the demo without the PPU (with a test server that would take the place of the the Power PC), I worked extensively on debugging the Power PC board software. The software running on the Power PC was fairly basic. It was just an infinite loop that read packets from the ethernet hardware (using the wonderfully buggy xilnet software package) and responded. There is a section of this report that explains this in more detail. All in all this took about 20 hours.

I also worked on converting the demo into a basic game so that our group met all the requirements of the project. The game was to be a 2d box shooting demo. Nick did the graphics and the user input, and I wrote all the game logic. The objective was to destroy all the boxes that corresponded to your crosshair's color. I spent about 24 hours doing this. I also added some background music and laser sounds. It was a real hit with Professors Marculescu's children, who recommended our group receive an A :).

Finally I spent the last two weeks in lab working with Aaron debugging the hardware implementation on the board. I basically looked over the Verilog to make sure that changes he made made some sense. I also modified the Power PC /demo in order to better test changes to the PPU. All in a heavily sleep deprived and caffeinated two weeks. During this period of time I probably spent 10 hours a day in lab for 5/7 days in each week.

Aaron Hoy

For this project, I designed and implemented the architecture for the physics processing unit. The first challenge was the address generator for the memory read stage of the pipeline. Figuring out which address to read next may seem easy, but it was actually one of the more difficult parts of the PPU implementation. There needed to be two registers in the unit, one to store the most recent object A address, and another to hold the current object B address. In a normal case, the object A address stays the same, and the object B address is incremented. However there are special cases, such as when one of the addresses wraps around from the end to the beginning of the array, or when object B had made it all the way around the array and it is time to change A. For each normal special case there are also abnormal special cases such as more than one happening at once, or a stall in the read stage due to a write back happening at the same time. Once I debugged the address generator I implemented the rest of the pipeline architecture, putting dummy modules in place of the computational units. Once I got all the timing and special cases in the pipeline debugged, I improved the dummy modules to actually do something so I could test that objects bounced off of each other. After getting all that done I began meeting in the lab with the rest of the group so we could put all our parts of the project together. The Xilinx software turned out to be a nightmare and I spent hours and hours helping to debug the bus interface between the PPC core and the PPU verilog modules. I also added data ready signals to the memory controller (previously it had no way of letting the other modules know if they had a successful read). The unit accurate computational units that we had unfortunately could not be synthesized properly by Xilinx. It kept trying to optimize things out of the design and it ended up breaking the calculations. To get around this I gradually improved my dummy computational units until they actually did legitimate physics calculations. Getting the masses and elasticities normalized to reasonable numbers was quite tricky because of the Xilinx "optimizations" so I was only able to shift the fixed points on those number so be barely in range of normal masses and elasticities, so the objects all behave a little more like beach balls on a rubber sheet than solid blocks on the ground. However for the given values, the physics are still realistic. I also eventually implemented wind viscosity, wind velocities, and got conservation of energy to work properly in the other computational units. To show off these features I made some cool demos where a moving block is sent in from the side to knock over a vertical set of blocks that have settled out in a stack. The stack topples over and then the objects settle upright on the ground. During most of the semester I spend a reasonable amount of time simulating the architecture. I would say I spent about 12 to 15 hours a week at that time. Once we started working with the board and trying to combine all our parts of the project, I ended up spending a lot more time. For the last 3 weeks of the project, I literally spend every waking our in the lab dealing with the Xilinx synthesizer aside from 3 hours of sleep some nights and a few exams and lab demos for other classes that I had to go to.

Charles Norman

I started off, like most of the other members in the group, by helping brainstorm ideas on what project we should do and how to implement it. I helped Aaron and Jeremy in creating the overall architecture for the hardware design. Most of my work was with the hardware design and implementation of the amount overlap and collision acceleration calculator along with the time independent update. I spent a lot of time trying to take realistic physics calculations and turn them into logic in hardware while still keeping as much accuracy as possible. In the end, most of my modules were not used due to complications with the Xilinx board, mainly because a lot of the computations were synthesized out. An example of this was when I needed to negate something twice for precision, but Xilinx insisted on leaving it positive the whole time. Despite these setbacks, we were able to use the general calculations that I computed earlier on and create simpler computations that were able to synthesize correctly on the Xilinx board. As far as time spent, in the beginning, I did not have a lot of work to do (generally less than 10 hours a week) due to the fact that I had to wait to see exactly how the software was going to interact with the board. Also, since I did not have a lot of architecture experience, my input with the overall hardware architecture was limited. Most of my contribution came in the last weeks when our hardware architecture was complete and the software implementation was close to complete as well. This is when I finally coded the necessary Verilog for the physics processor. In addition to coding the Verilog modules, I worked on a floating point → fixed-point → floating point converter in C. However, we saw that this was unnecessary and did not implement it. Instead, we used the built-in casting function in C since edge cases were not present. I would say that my weekly contribution during this period was greater than 15 hours per week.

As far as my opinion for the course, I believe it is a more laid back course compared to other capstones. Compared to my first capstone, 18-525, there was a lot less structure. In my opinion, this is good and bad. It allows for more flexibility in the type of projects created. However, there is the possibility of less work getting accomplished. Also, in any given project, there is a high probability that at least one person's expertise will probably not be used. In our project, I was not fluent enough in software design for work on the game. Also, I have not taken any networking courses, so I was no help in communicating with the board. Most of my expertise came in digital hardware design. However, that was not used really at all in this project since only basic hardware design was needed. By the time my hardware modules were complete, there wasn't much I could do with helping debug the project on the board due to the fact that Aaron and Jeremy were so far along with that portion.

Jeremy Weagley

Initially, most of my work on the project was in the design. I worked with Aaron and Charles to plan out how the hardware would function. Then I began working on the memory module for the design. I initially planned to use the Core Generator to create the memory out of BRAMs, but apparently the version of Core Generator that we had was unable to create “large” memories. After finally discovering this and also finding that I was unable to update the Core Generator, I began to implement the memory module by hand. Once it was completed, I began to work on a way to interface our hardware with the Xilinx PowerPC. Initially, I was the one with the most experience with Xilinx boards in the group, given that I had worked with them the previous summer. I eventually managed to create a bus interface to allow the PowerPC to access the physics hardware via the Processor Logic Bus. Data could be sent or received through software addressable registers, so it looked like everything would be working fine once the physics hardware was finished.

It was at this point that most of my work on the project began. Even though everything had been simulated and looked like it would work correctly, nothing seemed to work. After we had spent many hours debugging, we learned that the Xilinx synthesis tools were not synthesizing vast portions of the design. I probably spent twice as much time in the last two weeks of the project as I did in the entire rest of the project. For an entire week I spent about 6-8 hours a night in the lab with other members of the group trying to debug the design and figure out ways to get things to work. Eventually we figured out ways to force the synthesizer to implement our design and after re-tooling the physics pipeline, we were finally able to get the physics processor working reasonably well.

As far as my impressions of the course, I thought it went pretty well. As I mentioned before, it seemed like the project started slowly but then picked up way too much toward the end. While this is partly due to the fact that I was putting off some of the project work, it was also due to the fact that we did not know the difficulty we would face in using the Xilinx tools. My recommendation for the course would be to include more instruction about FPGA design at the beginning of the course, and also to introduce Chipscope earlier, since that seemed to be the only useful debugging tool.

Project Website

Our project website can be found at: www.andrew.cmu.edu/user/nbannist/index.htm