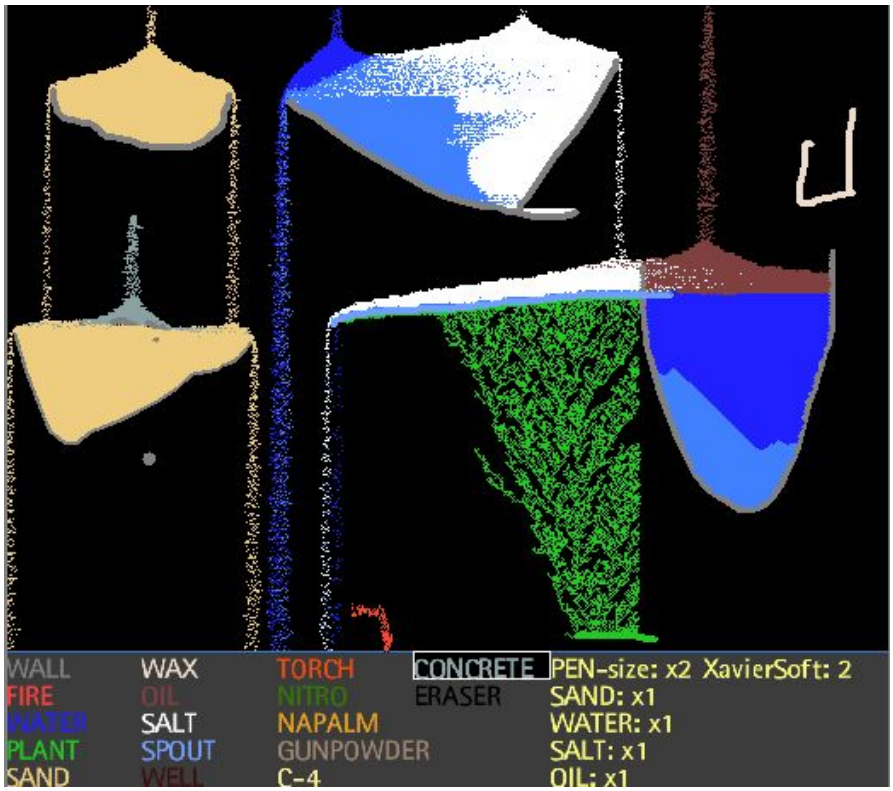
CSEE4180 - Falling Sand

Jeremy Adkins (JA3072)
James Kolsby (JRK2181)

14 May 2019

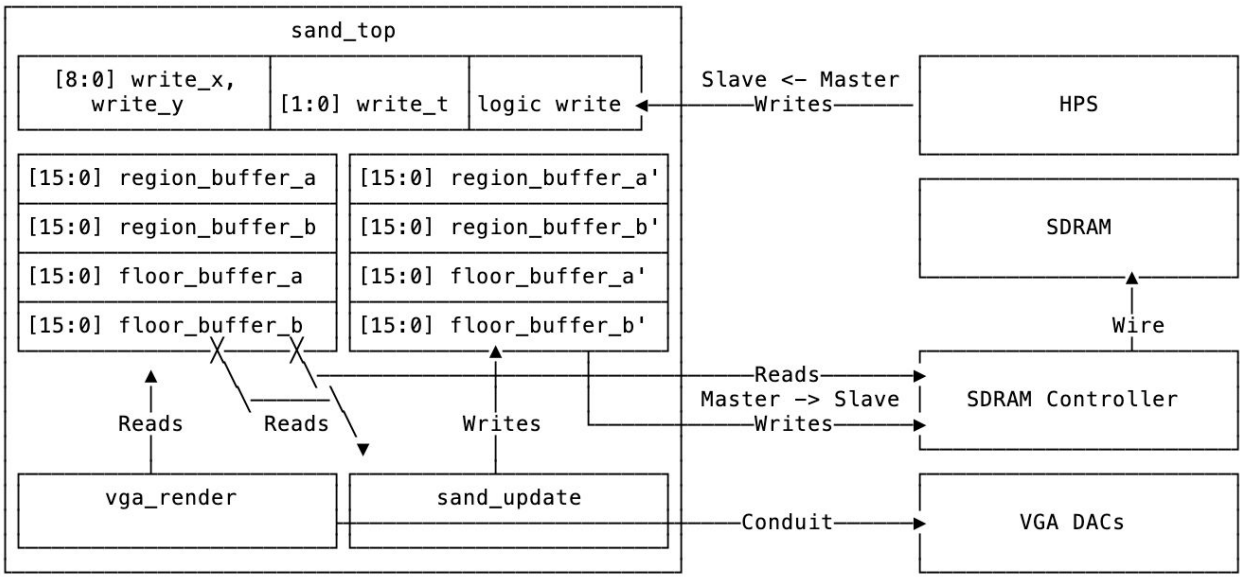
Table of Contents, p.1



Pyro Sand Game

Overview

We set out to design a game inspired by Pyro Sand Game, a java game from the late 2000's that centered around the interactions of small particles falling down the screen. Featuring numerous particle types, Pyro Sand enabled a user to create complicated structures and interactions by utilizing the properties of the particles.



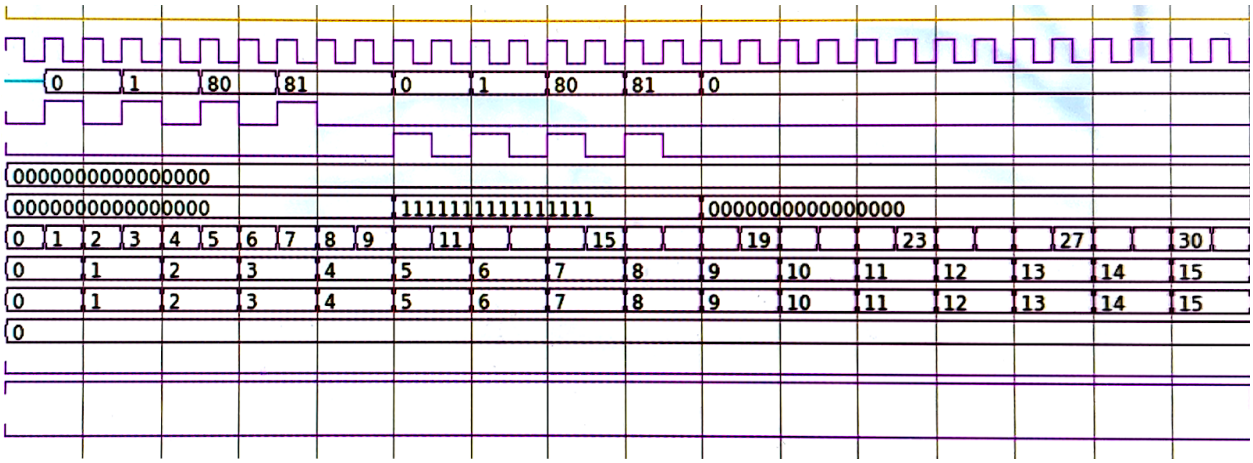
Seeking to make our implementation a fair bit simpler than this, we settled on 4 particle types, of which the usage of each will be elaborated upon within section 3a, the discussion of our physics engine. Our particles were represented as Sand, Air, Wall, and Sand(Already Moved). Sand(Already Moved) allowed for the saving of state to eliminate overlap in the physics calculations, and Wall remains static in place and cannot be moved except by user input. Sand falls down and piles up into conic mounds, simulating gravity in a 2D plane. As the screen updates and is saved as a structure in SDRAM, it is written to the screen by a VGA driver. The physics engine is structured as a cellular automaton, with each pixel being treated as an object acted upon by physics.

This implementation results in a simple game, in which a stream of sand from the top of the screen and a user cursor functioning as a sand spout/wall pen/eraser allows the user to create structures and watch the sand pile up.

Timing Design

One of the most important roles of the sand_top module is to synchronize the VGA outputs with the reads and writes between SDRAM and block RAM. Our SDRAM was configured to input 24-bit addresses and output 16-bit words. While our proof-of-concept was configured with two screens, one to store the current state of the screen, and the other to write individual pixel updates to. When we began to configure the SDRAM, we decided that storing two screens would be an unnecessary overhead when we could instead keep track of which pixels had already been updated during the scan of the screen. We settled on using two-bit pixels with the following parameters:

```
00 -> EMPTY_T
01 -> SAND_T
10 -> SAND_MOVED_T
11 -> WALL
```

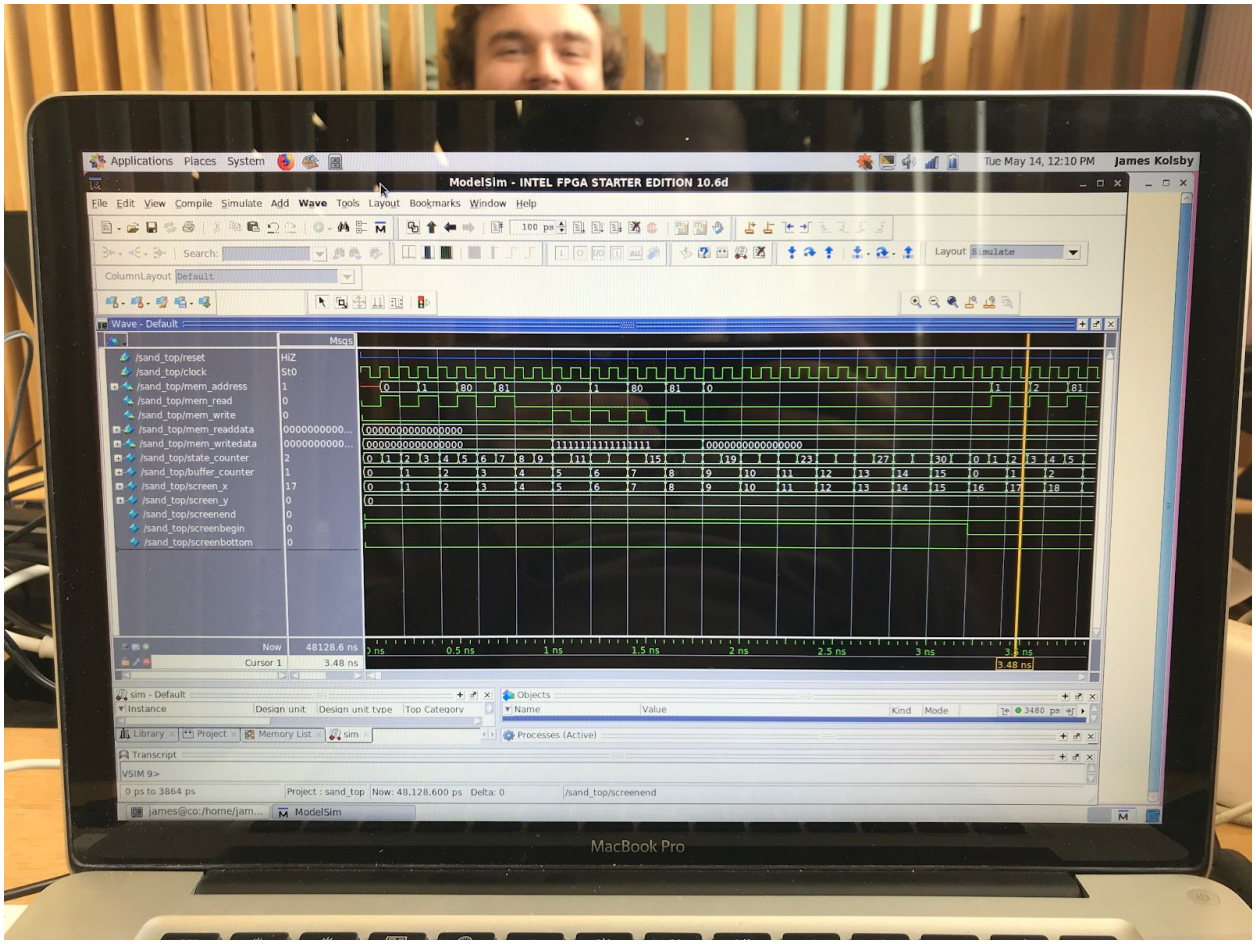


Screenshot: 32-clock timing for one region read / write cycle: `(from top) ->CLK, mem_addr->, mem_read->, mem_write->, ->mem_readdata, mem_writedata->, state_counter->, buffer_counter->, row_end->, row_begin->, screen_bottom->`

For a single, two clock cycle read from RAM, we can cache 8 pixels. At our desired VGA resolution of 640x480, we only need to output one pixel every other clock cycle, which meant that our toplevel had to force the computation cycle to wait exactly half of the time for the VGA to burn through its cache before continuing to the next blocks of RAM.

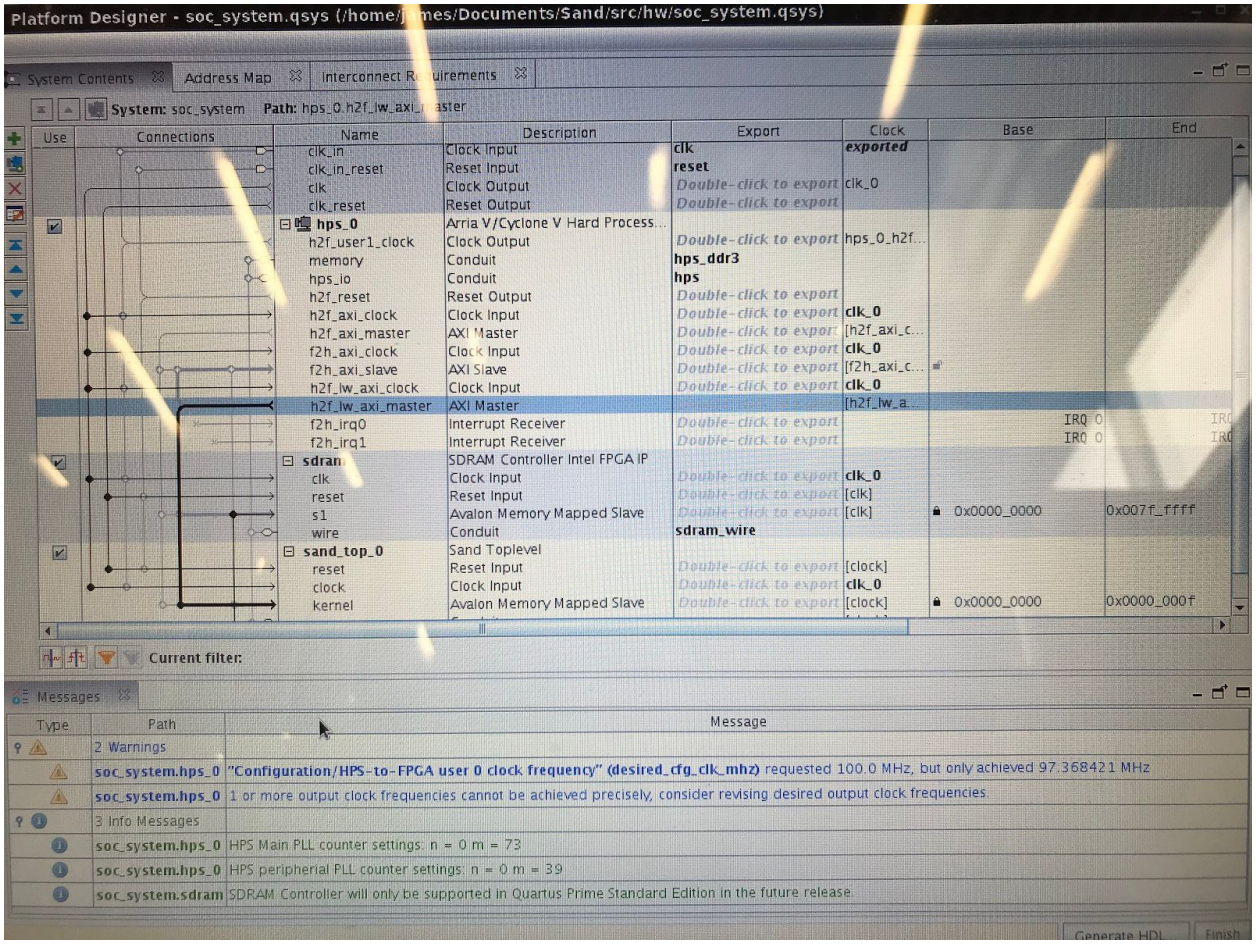
Our read pattern allowed us to store only 64-bits in block memory at a time, 16 bits from the current region, 16 bits from the region immediately to the right, and 16 bits from each of the "floors" beneath each of these regions. Because our sand game only implemented falling downwards, it was not necessary to load the "ceilings" above the current region. These four read cycles are followed by a single clock cycle of inactivity during which four updated buffers

are written to by the physics controller. Once these buffers have been updated, they are written back to RAM at the same addresses from which they were read.



Screenshot: ModelSim demonstrates the functionality of the toplevel read-write cycle

To keep the timing intact, the top-level counts incoming clock edges and sends the appropriate signals to the SDRAM controller via an Avalon Memory-Mapped Interface. Earlier revisions of our design intended to implement the VGA and physics modules as masters to the SDRAM slave, however the scheduling of multiple masters proved more difficult than giving all control to the toplevel. This allowed us to move data to and from modules much more easily, also it made compilation far easier without the need to frequently re-analyze interface changes in qsys.



Screenshot: sand_top instance in Qsys Platform Designer. Just beneath the kernel slave is the Avalon Memory Mapped Master connected to sdram slave s1.

All user input in our design is communicated from a kernel module which reads keyboard input and writes to the sand_top via another Avalon Memory Mapped Interface. It is capable of changing the pen position via the arrow keys, the particle type to draw via the number keys, and the spacebar to put the pen down/up.

Project Sub Modules

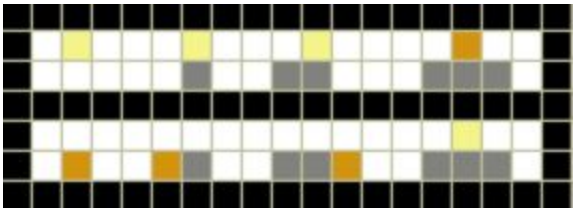
Physics Engine

Our physics engine is modeled as a cellular automaton, in which each particle is a discrete unit that is subject to the physics engine's rules. The screen is modeled as a full 640x480 array of pixels, each of which is 2 bits due to our 4-particle implementation. This provides us with a screen structure that is 614,400 bits or 76,800 bytes. Because this is too large to store in block memory, it necessitated the storage of the screen struct in SDRAM. The algorithm iterates down each row of blocks, requiring 4 SDRAM words (16 bits each) per calculation cycle. It divides the data in RAM into 2 structures; region and floor. Region is the current block in question as well as the next one in the row; floor is the block beneath the one in question as well as the next block in that row. This allows an iteration of a single block at a time across the row, providing the ability for particles to be aware of those beside and beneath themselves.

Once the relation is established between adjoining blocks, each 2-bit pixel in Region is calculated, and its position is updated through combinatorial logic. The 32 bit input for Region and Floor returns a 32 bit output consisting of New_Region and New_Floor.

If a pixel in Region (the row being iterated through in RAM) is found to be sand, its surroundings will be checked. If the space beneath it is open, it will simply fall down. If it sits atop another particle, it will travel downwards diagonally if possible. This allows sand to stack up and flow down the side of the "dune". It is placed into Floor as Sand(Already Moved) so that the engine knows not to remove it when Floor eventually becomes Region for the next row of calculations. When Sand(Already Moved) is found in Region, it is converted into regular Sand so it will be able to move again the next time it is iterated over. These transform checks run on each pixel.

The engine produces a single spout of sand at a given coordinate, as well as creating a block of a given particle type at a user-input specified cursor. It can be set to Sand to act as a spout, as Wall to draw structures, and Air to act as an eraser. As sand falls and stacks up at the bottom of the screen, the screen will eventually fill. Therefore there is also a Reset to empty the screen of sand.

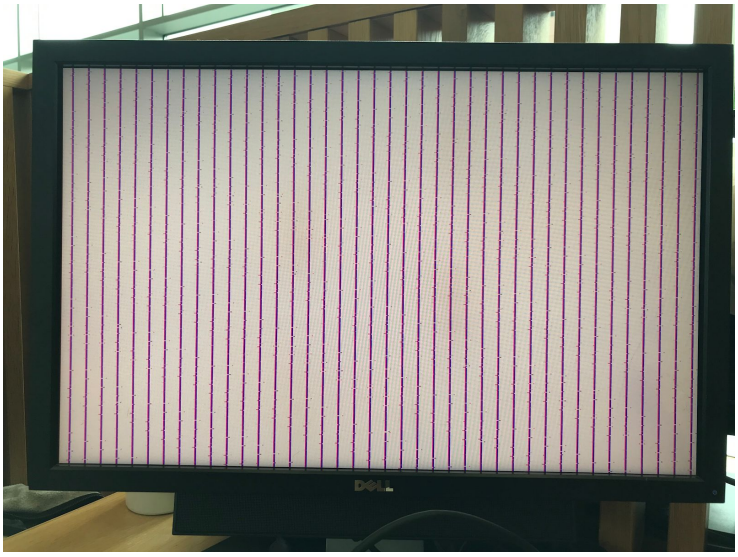


Above is an example of some transforms that would be made to the pixels in an example set of rows. Sand (yellow) falls to the Floor layer, being marked as Sand(Already Moved) (orange). Any Sand(Already Moved) remaining in the upper row from previous moves is converted to regular sand to ensure it only moves once per screen update. Wall (gray) is immobile and Air (white) is functionally empty space. If a sand rests atop a lone particle, it will shift diagonally to the left or right depending on horizontal position of the particle.

VGA Render

The rendering module takes the concatenation of both region buffers as its input, and a pointer within that double-sized buffer for the current pixel which it should be displaying. Because the VGA timing requires one pixel be output every other clock edge, this pointer could be calculated by dividing the state_counter by 2. Within a combinatorial logic block, the { VGA_R, VGA_G, VGA_B } values were assigned to colors mapped to the slice of current buffer at the pointer. Because the VGA read from these buffers at half the rate that they were computed, this required that computations and memory operations sleep for half of the possible states of the toplevel.

After simulation, the VGA module was our only way of getting feedback from the actual board to get an idea of what is happening during runtime. When we first encountered problems with data being written to RAM and then disappearing, we began to investigate by outputting specific colors to VGA on the condition of a high mem_readdatavalid or mem_writeresponsevalid signal. In the case of the former, we would output a red pixel, and a blue pixel for the latter. We believed that the issue was with writing, as our background was white, indicative of successfully reading all zeroes from RAM. The output of this test is seen to the right. Notice both red and blue pixels appearing in regular intervals along the screen. This proved that these signals were at times going high, however the data which we were attempting to write to SDRAM was not



being read back. In the end, this means of debugging was not sufficient to diagnose the problem.

Top-Level User Input

The project takes in a coordinate from a kernel module, which turns the given coordinate into a spout, similar to the one at the top of the screen. This spout functions differently depending on the user's selection, allowing the user to drop in more sand, draw walls, or erase particles using a USB keyboard.

Two-way communication could have been a very useful feature to implement for the purposes of debugging. If our kernel module could read data from the hardware device, we could possibly print out the values being read from SDRAM or even timing-specific logs to illustrate discrepancies with our timing diagram.

Milestones

I. Prototype

The first step of this project was to decide upon an algorithm which we would implement in hardware. We chose to do this in python initially for its ease of use and focus on pure arithmetic. Our algorithm was described as follows:

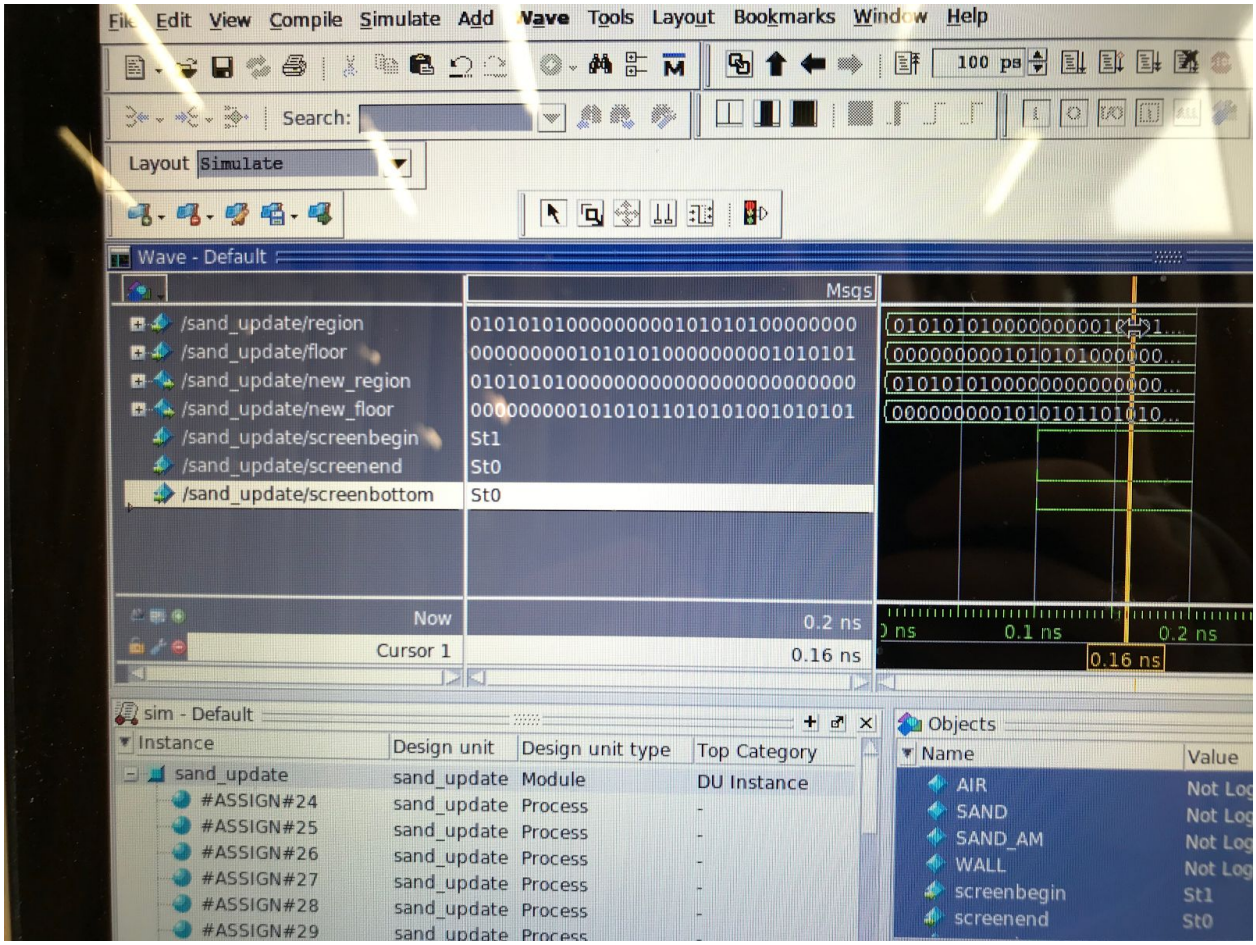
```
def check(screen, i, j, d):
    for k in d:

        if k == 0:
            return (i,j)

        elif (k in range(1,4)):
            (_i,_j) = (i-1,j-2+k)

        elif (k in range(4,7)):
            (_i,_j) = (i+1,j-5+k)

        elif (k in range(7,9)):
            (_i,_j) = (i,j-8+k)

        else:
            continue;

        if (_j in range(WIDTH) and \
            _i in range(HEIGHT) and \
            screen[_i][_j] == EMPTY):
            return (_i, _j)

    return (i,j)
```

```
def update(screen):
    output = ""
    nextscreen = [
            [EMPTY for y in
range(WIDTH)]
            for x in range(HEIGHT)]
    for i in range(HEIGHT):
        for j in range(WIDTH):
            output +=
DISPLAY[screen[i][j]]
            output += " "
            # IF NOT EMPTY
            if screen[i][j] > EMPTY:
                p = screen[i][j]
                screen[i][j] = EMPTY
                d = VECTOR[p]
                (_i,_j) = check(screen,
i, j, d)
                nextscreen[_i][_j] = p
        output += "\n"
    print(output)
    return nextscreen
```

This algorithm creates a rudimentary sandbox displaying particle interactions in a small array that simulates a screen. This algorithm would provide the foundation of our physics engine, the completion of which was part of our Milestone II.

II. Working Simulation / Physics Engine

This milestone was centered on planning and verifying clock sequencing in ModelSim, as well as verifying that the physics engine was working properly. In the simulation below, two input rows, region and floor, output the expected new_region and new_floor. The reason the sand only falls in the right end of the input is to ensure that each calculation is only done once on a particle, despite that half of the row chunk being subjected to calculations twice. The leftmost pixels in the chunk only update if the chunk is at the left end of the screen; otherwise, the bits will eventually be in the right half of a chunk to fall properly. We confirmed that VGA would sync properly with calculation to prevent runaway screens, and confirmed that Milestone II was met.



III. Working VGA, RAM, Peripheral Controller

Our VGA driver is able to display the default empty screen properly, and is satisfactory for displaying the game. However, our pixel values kept being lost by the SDRAM controller, which gave every indication of proper storage and retrieval yet didn't actually update its values. While the VGA output was useful for debugging, it did not prove informative enough to diagnose the root cause of our SDRAM memory access failure. We verified that signals were being raised signifying successful reads and writes, which we displayed as red and blue ends of a chunk. This properly generated expected red and blue stripes on the screen that would seem to indicate SDRAM was being accessed, yet no particle types were being held. Our inability to get the SDRAM register to function was our greatest challenge and was our shortcoming in Milestone III.

Challenges

As discussed in Milestone III, our functioning infrastructure was unable to properly interface with the SDRAM controller. Despite experimentation and research into proper memory usage in SDRAM, the device would not respond correctly. This challenge's resolution would result in a completely functional simulation/animation, as the other components are verified to behave correctly (except for hardware, the adaptation from Lab 2 and 3 was put last in the workflow after SDRAM access). We are disappointed by this shortcoming but feel pleased with the complexity of our 32 bit to 32 bit physics function, the logic simulation of which is a perfect example of the usage and utility of an FPGA.