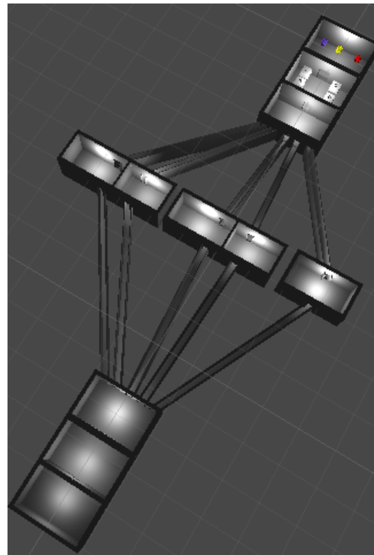
Designing a Superscalar Processor Simulation

Author:

William Callanan

Supervisor:

Chris Fernandes



March 18, 2014

Abstract

The purpose of this project is to create an effective simulation of a superscalar processor that could easily be understood by undergraduate students. Superscalar processors are difficult to understand, and I am hoping to use this simulation to help make them less frightening to students attempting to learn about computer hardware. The simulation I have been designing uses the Unity engine, which is a multiplatform game engine, to visualize the way superscalar processors deal with data. Overall while there is still a lot of work to be done on the project the simulation is mostly implemented and the results have been pleasing. Everything moves at a reasonable speed and while it hasn't been tested by other students yet it seems easy enough to understand.

Contents

1 Introduction **3**

1.1 A Scalar Processor . . . 3

1.2 From Scalar to Superscalar . . . 5

1.3 Improvements Provided by Superscalar . . . 7

2 Design **8**

2.1 Design Requirements . . . 8

2.2 Simulatior Design . . . 9

2.3 Instruction Design . . . 12

3 Final Design and Implementation **13**

3.1 Changes from mockup . . . 14

3.2 Student View . . . 14

3.3 Implementation . . . 16

3.3.1 Instruction Script . . . 16

3.3.2 Functional Unit Script . . . 20

3.3.3 Execution Unit scrpit . . . 21

3.3.4 Camera Script . . . 22

4 Evaluation **23**

4.1 270 Demonstration . . . 23

4.2 Post 270 Evaluation . . . 23

5 Future work **23**

6 Conclusion **24**

7 Glossary **25**

List of Figures

1 a model of a scalar processor 4
2 the stages in a superscalar processor 5
3 an existing picture model of a superscalar processor [1] 9
4 a mockup of what my simulation will look like to the student 10
5 register map at start 11
6 register map after renaming 11
7 2 integer instructions with their countdowns showing 12
8 the version of an instruction I currently plan to use 12
9 a previous version of an instruction, this is the more in depth section of it, the blue color to the left of the text would have been what represented the instruction as it moved 12
10 Instruction 1 without the wings 13
11 Overhead view of simulation un Unity 14
12 The view a student will see when they start a simulation 15
13 the settings for public variables for instrution I3 (code counting starts at 0, but viewed values start at 1) 16

List of Tables

1 scalar cycles 7
2 superscalar cycles 7
3 action comparasons 8

1 Introduction

Processors are the brains of computers, and much like the brain, modern processors are able to deal with multiple pieces of information at a time. However processors didn't always work like this. Initially there were scalar processors. These scalar processors could only execute a single instruction at a time. But developments over the past twenty years have allowed for the existence of a superscalar processor, a processor which can compute multiple pieces of information at once, and can even deal with these pieces of information out of order.

My goal for this project is to create an easy- to-understand animated simulation of how a superscalar processor functions. It will hopefully give students a clear understanding of both the differences between scalar and superscalar, and the benefits that superscalar provide.

There is at least one other superscalar simulator that I have already found known as SESC [2], however that simulator is flawed for my purposes for a few reasons. It is completely text based, it only shows the data inside each instruction and how much time it will take to execute. And even though that is useful information to someone who already understands processors, it is the wrong information to be showing in an undergraduate class, such CSC 270 Computer Organization. An undergraduate class will need a more visual approach, something that gives a visual representation of how data flows through the processor.

1.1 A Scalar Processor

In Figure 1 we see the stages of a scalar processor. First, there is **instruction fetch** and decode, where the instruction is collected from **Instruction Memory** then decoded into a language the processor can understand. The next step is **Register Read**, which is when the processor collects data from the **Registers**, units in the processor that store data that might need to be used later. After this we have the **Execute stage**. At this stage the instruction can now be run in the **Execution Units** otherwise known as ALUs or Arithmetic Logic Units, which are either integer, floating point, or memory units. One of these units will be used to compute the instruction and the result will move on to the final stage, **Register Write**. This final stage is where the result of the computation is written back to the proper register in the processor so later instructions can use that value.

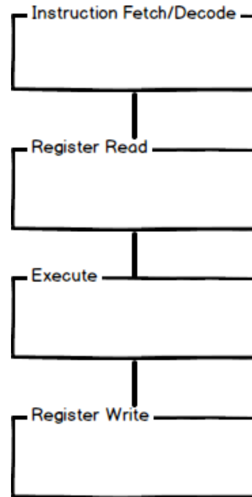


Figure 1: a model of a scalar processor

If we were to move an example instruction through a scalar processor it would end up being a little like this. Say we have three instructions that need to be executed, instructions, 1, 2, and 3 and they are running through a scalar processor with four registers, A, B, C, and D. *Instruction 1* adds the data from registers A and B and stores it in register B,*instruction 2* adds register B and the value 5 and stores it in A, and finally*instruction 3* adds C and D and stores it in B. In a processor instructions wouldnt actually be read off like this, but instead instruction 2, for example, would look like this addi, B, 5, A. the first job of the processor, as mentioned above, would be to fetch instruction 1 and decode it. After this it would perform register read and get the values that registers A and B are currently storing. Next is the execute stage where the values of registers A and B are added together , then register write would take place and that new value would be written into register B. After register write for instruction 1, the processor will fetch instruction 2 and decode it. However instruction 2 will not collect 2 values during its register read step because it only collects data from one register, the other data to be added is the provided 5. In the execute step the value of register B and 5 are added together, then in the write back step the result is stored in A. Now we will fetch our final instruction, instruction 3. It will read C and D to get the values and then add them during the execute step, then finally during the register write step it will store the result of its computation in register

B. Overall this set of instructions would take 12 cycles to execute assuming each stage takes 1 cycle.

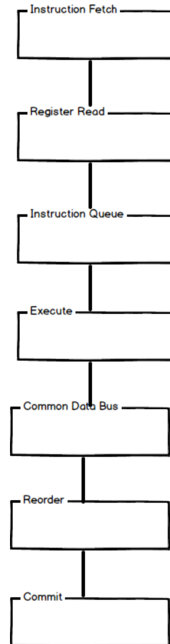


Figure 2: the stages in a superscalar processor

1.2 From Scalar to Superscalar

Figure 2 shows the stages of the superscalar processor. A few of them are the same as in the scalar processor, however there are a significant number of new or changed stages.

The first change is that **Instruction Fetch** now fetches four instructions at once instead of just one as it did in the scalar processor. Fetch also now occurs once every cycle instead of just having instructions be fetched once the previous instruction completes the register write stage.

The next step after fetch is **Register Read** which is a stage that has changed significantly from its scalar version. In superscalar processors Register Read has been updated to use a **Register Map** instead of just the registers. The reason for this is because if we are taking in multiple instructions at once there is the potential for us to have what is known as a **data hazard**. Data hazards are situations where a later instruction wants to use a register that an earlier instruction still needs to write data back to. If we allow

for out of order execution then these data hazards become very likely, and if left unchecked they can cause bad values to be used in computations, causing the data to become corrupted. The register map allows us to deal with these data hazards by making sure that instructions claim the register they are going to write back to when they go through the map to collect their data. This means that a later instruction looking for that data before it is ready will see a pointer to the instruction that claimed it, which will tell the later instruction that it must wait for a result from that instruction before it can compute.

After register reading we have a new stage known as the **Instruction Queue**. This stage is what makes sure that instructions who don't have all their data are not allowed to start computing. All instructions end up in the instruction queue at some point and the instruction queue allows them to enter into the execute stage based on a priority queue, where the instruction with the highest priority is the oldest instruction that is prepared to execute, meaning the processor has the hardware available to compute it and all of its data is ready. It is in this stage where things might get sent out of order if it will speed up the processing time of the program.

Once an instruction leaves the instruction queue it enters into the **Execute** stage, essentially this stage is the same as it was in the scalar version. Instructions still enter this stage, are computed, and then leave with their result. However the primary change to the execute stage in a superscalar processor is the hardware. In a superscalar processor there are multiple duplicates of the different types of ALUs, so there might be two Integer ALU's, two Floating Point ALU's and one Data unit. A setup like this would allow the processor to deal with 2 Integer instructions, 2 Floating Point instructions, and a single data instruction all at the same time.

After this we have another new component, the **Common Data Bus**. The purpose of this stage is to speed up the turnaround time on results. The common data bus does this by sending the results of finished computations back to the instruction queue so that any instructions that were waiting for that data can start going right away instead of having to wait for the instruction to commit.

Lastly, there are two new stages: the **reorder stage**, and the **commit stage**. The reorder stage has what is known as a **Reorder Buffer**. This keeps track of what the next instruction that can be committed is, and then holds all the others. This ensures that in order commit happens. This leads into the commit

stage which is when all of the registers are updated and any holds shown on the register map are removed. After commit an instruction is totally finished and has left the processor.

1.3 Improvements Provided by Superscalar

Table 1: scalar cycles

	Instruction Fetch	Register Read	Execute	Register Write
instruction 1	cycle 1	cycle 2	cycle 3	cycle 4
instruction 2	cycle 5	cycle 6	cycle 7	cycle 8
instruction 3	cycle 9	cycle 10	cycle 11	cycle 12

Table 2: superscalar cycles

	Instruction Fetch	Register Read	Instruction Queue	Execute	Common Data Bus	Reorder	Commit
instruction 1	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7
instruction 2	cycle 1	cycle 2	cycle 3	cycle 6	cycle 7	cycle 8	cycle 9
instruction 3	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 9

If we were to run our earlier example in a superscalar processor we would see a significant speed increase because superscalar would allow a few things to happen simultaneously instead of having to go one after the other. Table 1 shows that it will take 12 cycles for our example instruction to run through a scalar processor. Table 2 shows the same instruction running through a superscalar processor, and even though the superscalar processor has more steps for each instruction to go through it actually finishes faster, because multiple instructions are able to be in the same step at the same time. In Cycle 1 the scalar processor is only able to fetch instruction one, but in the superscalar processor all 3 instructions are fetched. In cycle 2 the scalar processor has instruction one perform register read, in superscalar all 3 instructions complete register read in the same cycle. In cycle 3 the scalar processor executes instruction 1, and in superscalar all 3 instructions enter the instruction queue. In cycle 4 the superscalar processor starts to execute instructions 1 and 3 only, this is because instruction 2 would have needed to use the register instruction 1 is writing back into, causing a data hazard. Meanwhile, the scalar processor has instruction 1 perform register write. in cycle 5 our instructions in superscalar have reached the common data bus, meaning their results have been sent back to the instruction queue allowing instruction 2 to finally execute in cycle 6 when it would just be

in the register read stage in the scalar processor. In cycle 7 we have instruction 1 commit from the reorder stage, however because instruction 2 has not arrived yet instruction 3 is held in the reorder buffer. in cycle 8 the scalar processor has just performed register write for instruction 2, but the superscalar processor has just had instruction 2 reach the reorder stage meaning that on cycle 9 both instruction 2 and instruction 3 will commit finishing the program.

Table 3: action comparasons

cycle number	scalar actions	superscalar actions
1	Fetch I1	Fetch I1 I2 I3
2	Reg Read I1	Reg Read I1 I2 I3
3	Execute I1	Enter Instruction Queue I1 I2 I3
4	Reg Write I1	Execute I1 I3
5	Fetch I2	Enter CDB and send result to IQ I1 I3
6	Reg Read I2	Execute I2 and Enter Reorder buffer I1 I3
7	Execute I2	Enter CDB and send result to IQ I2 commit I1
8	Register Write I2	Enter Reorder buffer I2
9	Fetch I3	commit I2 I3
10	Reg Read I3	
11	Execute I3	
12	Reg Write I3	

2 Design

As previously stated, my goal is to create an animated simulation that will allow a student to easily see the details of an example like the one I described earlier. In order to accomplish this I have needed to focus on creating a design that is simple enough to be easily understood, but complex enough that it gets all of the necessary info across to the student.

2.1 Design Requirements

There is a single real requirement from a design perspective for this project, and that is, that the end result be multiplatform. Most likely this is going to mean that it will be web based, but it could be a native application on a platform, so long as it works the same on windows and mac.

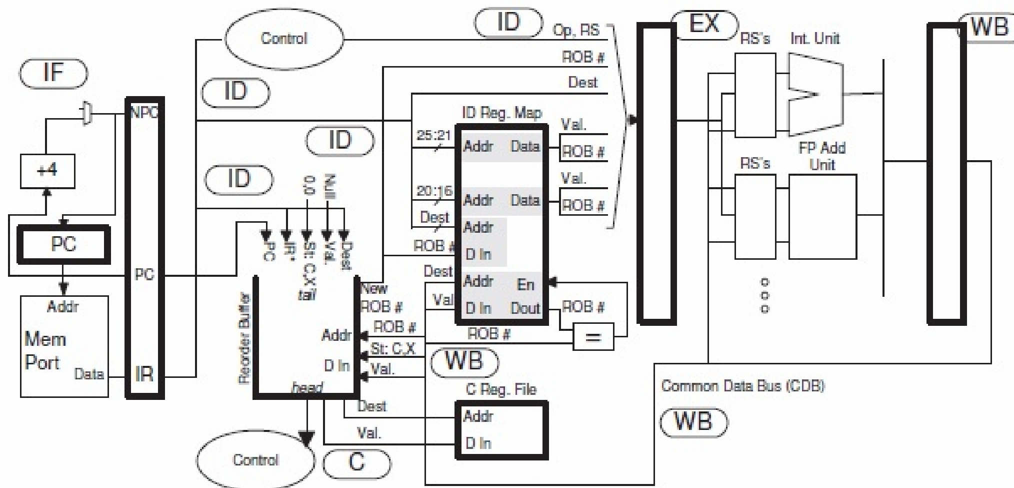


Figure 3: an existing picture model of a superscalar processor [1]

2.2 Simulatior Design

My design for the simulator is an attempt to improve drastically on existing picture images of superscalar processors as shown in figure 3. Which, although they convey a large amount of information, would be very difficult for a student to understand unless they already had a grasp of how superscalar processors worked. My design, shown in figure 4, deals with this problem by obscuring a lot of the unnecessary information, and focusing more on showing how data moves through the processor in a general sense.

In my design I show each of the stages that we talked about for superscalar processors in the intro, however they are being shown as components instead of as stages. So we have instruction memory which will hold all of the instructions. Figure 5 shows a close up view of my simulated register map. Each register is shown as a square with a value inside, and there are two possibilities for what this value can be. It can either be the register name which is the default state of the register when no instruction has claimed it. Or it can be the renamed value, the register name is changed to this when an instruction intends to write back to this register at the end of its computation. Currently the actual contents of the register are never displayed, although it is possible that a future version might make displaying the register values an option

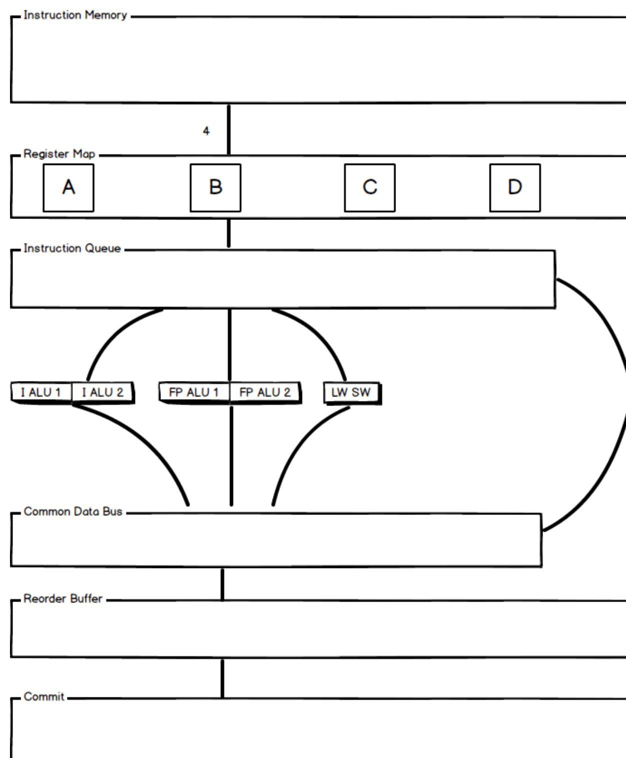


Figure 4: a mockup of what my simulation will look like to the student

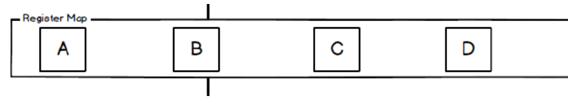


Figure 5: register map at start

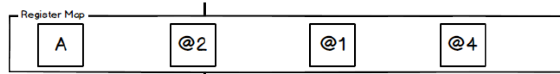


Figure 6: register map after renaming

if the students who review the program think it would be a useful feature. When an instruction claims a register, the syntax of the renaming value is, for example, @1. This will tell later instructions that the actual value they need to use will come from that instruction. So if instruction 1 is writing into register C then register C will show @1 inside it instead of C, like in figure 6. Then if instruction 2 wants to use register C instead of finding the value C meaning it can just use the value it will find the value @1 which will force instruction 2 to wait for the result of instruction 1 before it can compute.

The instruction queue will visibly hold the instructions and will inform the user of which instructions have been cleared leave the queue by checking a box above the instruction in question. Once they have been cleared to go they will move to one of the ALU's for its type and a countdown of the number of remaining cycles will be seen as in figure 7. With a number next to the ALU representing the remaining execution cycles. Each type of instruction has its own number of cycles that it takes to complete, with integer instructions taking one cycle, floating point instructions taking three cycles, and memory instructions taking two cycles. The ALU's for each of these types will be color coordinated with the instructions, with blue being used for the integer units, red being used for the floating point units, and green being used for the memory units.

When instructions leave the execution units they will enter into the common data bus. When they reach here their results are immediately sent back to the instruction queue so any waiting instructions will be able to use that data and leave on the next cycle if they are ready. And any instructions in the common data bus will enter into the reorder buffer on the next cycle. Instructions will be held in the reorder buffer unless all of the instructions before them have already committed or are already in the reorder buffer. Then finally

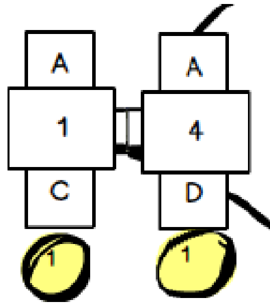


Figure 7: 2 integer instructions with their countdowns showing

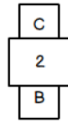


Figure 8: the version of an instruction I currently plan to use

the cycle after they enter the reorder buffer any instructions that are able to commit will, and if multiple instructions are ready to commit, meaning that, for example instructions 1, 2, and 3 are all in the reorder buffer at the same time, then they will all commit at the same time.

2.3 Instruction Design

Over the course of planning this I went through quite a few different designs for the instructions but figure 8 is what I currently plan to use. This is a very basic model that only deals with 2 registers, the top one is the register it collects data from, and the bottom one is the register it writes back to. There is another version of this instruction shown in figure 10 which obscures the register information that also might be used. Each particular version of this instruction has its advantages, in the winged version can get a clear understanding

Figure 9: a previous version of an instruction, this is the more in depth section of it, the blue color to the left of the text would have been what represented the instruction as it moved

Figure 10: Instruction 1 without the wings

of why things move, however due to the design we aren't getting an accurate view of an instruction since an instruction normally takes 2 inputs and leaves the results in an output. In the non-winged version we lose this understanding, however the details that the winged version provides would hopefully be obvious due to the actions the simulation takes even when it doesn't show the exact information to the user. There is a chance that I might make it so students can swap between these two views of instructions.

All of these instructions have one aspect to them that is not shown in my existing figures, and that is color. I mentioned earlier that the execution units will be colored, with integer ALUs being blue, floating point ALUs being red, and memory units being green. My above two examples share this same color scheme. Which makes it possible to show what operator an instruction is supposed to preform without needing to actually say the name of the operand.

There are also a few instruction views that I might add in later that are useful. Probably the most likely instruction model to be added would be figure 9. This is a design that I had scrapped previously for having too much detail. In this instruction we get all of the information about exactly what is moving and how, but the important part is the colored square to the left of the instruction, because this is what would represent the instruction as it moved through the simulation.

3 Final Design and Implementation

For my final design I built 3D models of each component in Maya and then combined them together in Unity, an overhead view of the design in unity can be seen in Figure 11. My final design ended up being very close to the design shown in Figure 4. I used Maya for building the models because it was the only tool that I could find that would let me put holes in my models to use as doors, and it had a free student version. Unity was the program of choice for building this simulation because it is a simple to use tool with robust scripting abilities.

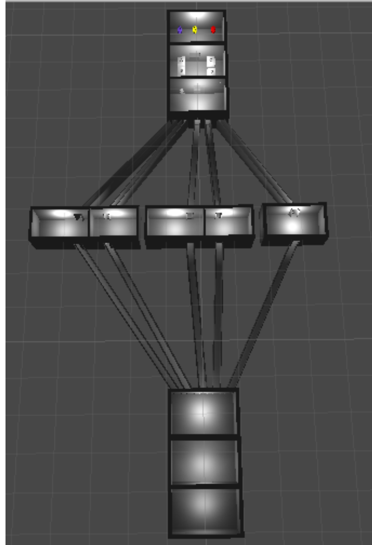


Figure 11: Overhead view of simulation un Unity

3.1 Changes from mockup

There are a few differences between my final prototype and my mockup. First, the sizes of some most of the units have changed, and second, the connection between the Common Data Bus and the Instruction Queue has been removed. The shrinking of the units was done so the camera could be up close to the units but still see everything inside them, as shown in Figure 12. While the removal of the connection between the Instruction Queue and the Common Data Bus was done to make the simulation less confusing, since students using the simulation for the first time may be confused about why a path from the instruction queue is never being used by data leaving the common data bus.

3.2 Student View

In figure 12 the starting view for the simulation is shown, from here the camera can be moved in any direction using W A S or D or the arrow keys. I decided to have student move the camera over having a script do it automatically for a few reasons. First, if the camera just moved automatically it would need to follow a particular path, and because the nature of superscalar processors is that things can happen in any order, this means that a path that looks good for one kind of instruction might not make sense for any

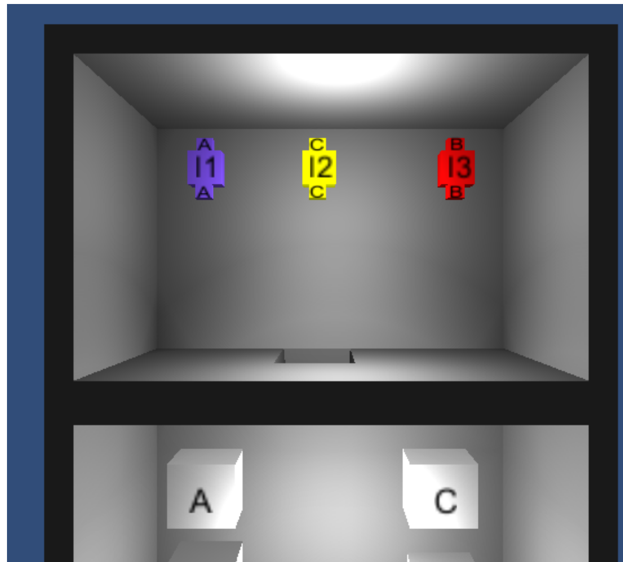


Figure 12: The view a student will see when they start a simulation

others, so each example built into the simulation would need to have a camera path to go along with it. I also could have done automatic camera movement by having the camera track a particular instruction, but while that is easy to do and would show the simulation running it would fail to convey what a superscalar processor does since superscalar processors are primarily for computing a lot of instructions at the same time, so just following a single one would miss a lot of important information.

I selected this view for the camera because it is close enough that all the values on the instructions and registers should be easy to read, but is also far enough away that the user can get a feel for what is around the unit they are looking at. The camera in the simulation pans slightly faster than the instructions move allowing the camera to catch up as things move. However this does have an unintended side effect of letting the camera outrun the instructions, but this is better than the user not being able to catch up with the instructions. As it stands it is not a huge problem that the camera can outrun the instruction since the user can just stop panning and wait for the instruction to catch up, but it would be nice to have a solution to this problem in a future version of the simulation.

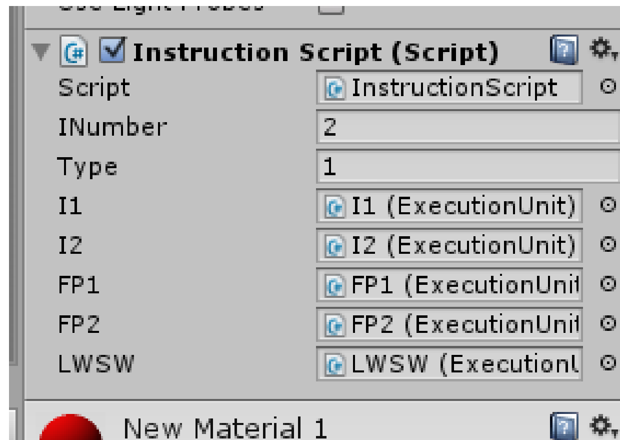


Figure 13: the settings for public variables for instrution I3 (code counting starts at 0, but viewed values start at 1)

3.3 Implementation

My design was built using 4 separate classes of code, however at this point only 2 of these are totally functional. These sections are, Instruction, Functional Unit (Instruction Queue, Common Data bus, reorder buffer), Execution Unit (Integer ALU, Floating point ALU, memory unit), and camera. At this point only the camera and the Instruction scripts are fully written and functional. Unity scripts are very interesting in that the only way scripts are run is through an update function inside them which is called once every frame for each object with a script attached. However they are not all called at the same time, and if one update takes a long time then the ones behind it get held up as well. This meant that in order to keep things running smoothly I had to make sure that if nothing changed for an instruction on a particular frame it could get in and out of update very quickly.

3.3.1 Instruction Script

```
using UnityEngine;
using System.Collections;
```

```
public class InstructionScript : MonoBehaviour {

bool move = false;
string Waypoint = "Waypoint 1";
Vector3 myPosition = new Vector3();
string currUnit = "InstructionMemory";//the current functional unit
public int INumber;//the number of this instruction
float speed = .2f; //default speed
public int type; //int instruction
Vector3 FinalDestination = new Vector3 ();
Vector3 MidPoint = new Vector3 ();
Vector3 StartPoint = new Vector3();
public ExecutionUnit I1;
public ExecutionUnit I2;
public ExecutionUnit FP1;
public ExecutionUnit FP2;
public ExecutionUnit LWSW;
```

Above is how all instruction are initialized, they always have move as false because they won't be moving right when they are created, instructions have a helper method for the purpose of setting move to true for when they are supposed to move. Because all instructions always start in the instruction queue they can all have waypoint as Waypoint 1 and currUnit as Instruction Memory all of the public values get changed and set from the main unity window and can be initialized here and then given different values for different instructions in the main unity window. So we can set the type of the instruction, and its instruction number outside. Then there are the execution units, these are referencing the execution unit script attached to those units. By identifying and naming them here I can get those to reference particular execution units and call functions inside their scripts, Figure 13 shows the settings for one particular version of the instruction script.

The instruction script uses a waypoint system to navigate instructions from one unit to the next. Luckily Unity includes a move towards function that takes the start position, the final position and the distance of motion as a float. This was useful because the built in function made sure that if the distance to the destination was less than the distance it should move on that step it would shorten the step to make sure it didn't overshoot. This meant that all my update function did was set what should be the next waypoint and inside the nextwaypoint function there was a check right away for if the current position of the instruction matched the position of the waypoint we are trying to move towards. This keeps the script from getting caught up in a potentially long and unnecessary check. Below is the code for my update function along with the start of my newWaypoint function.

```
void Update ()
{
myPosition = transform.position;
Vector3 NextWaypoint = (GameObject.Find(Waypoint).transform.position);
if (move==true)
{
transform.position = Vector3.MoveTowards(myPosition,NextWaypoint, speed);
newWaypoint(myPosition,NextWaypoint);
}
}
private void newWaypoint(Vector3 curPosition, Vector3 NextWaypoint)
{
if (NextWaypoint == myPosition)
```

The check for next waypoint and my position matching is inside the new waypoint function because it gives it a slightly higher chance of being true, meaning that it could potentially save us a frame. If that check is false then nothing happens and update ends for that frame, however if that check is true then we need to find what the next waypoint should be. This is done with a string comparison of a variable called currUnit which is initialized to the value InstructionMemory and whenever the instruction moves to a new

unit the value of currUnit is changed to the unit it has entered. Once we know the unit we are in then each unit has a set of at least 2 waypoints associated with it. The first of these is going to be the waypoint the instruction needs to move to start. This waypoint is always in the center of a unit and is used to line up the instruction with the door so it can go through. Then there is the final destination, this is the center waypoint in the next unit and reaching that lets us know that we have entered a new unit. Here the version of that code used for the Instruction memory unit.

```
if (currUnit.Equals("InstructionMemory"))
{
FinalDestination = GameObject.Find("Waypoint 2").transform.position;
StartPoint = GameObject.Find("Waypoint 1").transform.position;
if (StartPoint == myPosition)
{
Waypoint = "Waypoint 2";
}
else if (FinalDestination == myPosition)
{
currUnit = "RegisterMap";
move = false;
}
}
```

The one exception to this is in the setup in the ALU units, this is because they need to move to a start unit and then to a middle unit at the end of the long path to the Common Data Bus, then they move to the center of the Common Data bus for the final destination. One example of this version of the code is shown below.

```
else if (currUnit.Equals("I1"))
{
```

```
FinalDestination = GameObject.Find("Waypoint 4").transform.position;
StartPoint = GameObject.Find("Waypoint I1").transform.position;
MidPoint = GameObject.Find ("Waypoint I1E").transform.position;
if (StartPoint == myPosition)
{
Waypoint = "Waypoint I1E";
}
else if (MidPoint == myPosition)
{
Waypoint = "Waypoint 4";
}
else if (FinalDestination == myPosition)
{
currUnit = "Common Data Bus";
I1.setUseage(false);
move = false;
}
}
```

A final part that the instruction script doesn't have yet is a way to send back a finished value to the rest of the simulation so that it knows that it can start a new cycle, but will be implemented later.

3.3.2 Functional Unit Script

The functional unit scrip is unwritten as of this prototype although it is mostly designed, it is the algorithm for determining what instructions are allowed to move during a given cycle. In the same way that instructions are able to know what functional unit they are in, units are also able to know what instructions are inside them using an invisible box that is the size of the unit. Any instructions inside the unit will be colliding with it allowing the unit to identify instructions it controls and what information they have. The basic algorithm for the instruction queue section of this might be

check for instructions inside unit

if Instrution in unit AND register name matches instruction name AND instruction hardware is avalible **then**

Set instruction move to true

end if

Because of how different all these units are in terms of what is allowed to move from them it might make sense to have a separate script for each unit, however there are many benefits to having one script for every unit with different functions for each type of unit. That would allow me to create a simple update method that jumps to each type of unit. However the downsides of this are that a unit with a lot of instructions to deal with might slow things down.

3.3.3 Execution Unit scrpit

The script for the execution unit is a simplified version of the functional unit script and is slightly written. The functional units have a few important parameters, the first is their execution time, which is how many cycles it takes them to finish an instruction. Then they have a Boolean in use value that lets them tell the rest of the simulation whether they are currently running an instruction or not. This is a method that is called by the instruction script when an instruction is about to move to the unit so that other instructions know it is claimed and don't try to move to that unit themselves. However at this point only one part of this code is written and that is the usage part. It is a simple setter that allows the instruction script to check and set if the unit is in use or not

```
//returns true if the unit is in use, false otherwise
public bool getUseage()
{
return inUse;
}
//tells us if the unit is in use, true means it is in use
public void setUseage(bool use)
```

```
{
inUse = use;
}
```

The cycle check will come when I implement the finished check in the instructions and a way to keep track of cycles.

3.3.4 Camera Script

```
// Update is called once per frame
void Update ()
{
//Move left and right
if (Input.GetKey (KeyCode.A) || Input.GetKey (KeyCode.LeftArrow)) {
this.transform.position += new Vector3 (.3f, 0, 0);
} else if (Input.GetKey (KeyCode.D) || Input.GetKey (KeyCode.RightArrow)) {
this.transform.position += new Vector3 (-.3f, 0, 0);
} else if (Input.GetKey (KeyCode.W) || Input.GetKey (KeyCode.UpArrow)) {
this.transform.position += new Vector3 (0, 0, -.3f);
} else if (Input.GetKey (KeyCode.S) || Input.GetKey (KeyCode.DownArrow)) {
this.transform.position += new Vector3 (0, 0, .3f);
} else {
this.rigidbody.velocity = new Vector3 (0, 0, 0);
}
}
```

The camera script is uses its update function to check for key codes and transforms the camera position based on them. It looks for the key codes for either the arrow keys or W A S D to move the camera. In the future the camera will also be where the cycles are being kept track of, and what will check if we can move

on to a new cycle.

4 Evaluation

In CS 270 students are taught about how computer hardware functions with a large focus on processors. My evaluation is based around this course, and there are two parts to it. This first is an evaluation of how well CS majors who have not taken 270 understand the design and information it is trying to convey. And the second part is to look at students who are taking CS 270, or have taken CS 270 and seeing how well they understand superscalar processors after playing with the simulation.

4.1 270 Demonstration

I would like to demonstrate this program to the students taking CS 270 to get their opinions on the quality of the simulation and see if they have any recommendations for features that might be useful for them. Ideally later down the line, this demonstration could be turned into a lab where at the end the students could make feature requests. Hopefully these requests would result in a more robust and easy to understand program.

4.2 Post 270 Evaluation

The real test for conveying information comes from testing it on people who have already taken 270. These students will get the chance to play with the program hopefully in more than just a demo setting. At the end they will also answer a few questions to evaluate their understanding of superscalar processors, they will also leave behind their own evaluation of the simulation and potental features they would find useful, the requests from the people who have taken 270 and who are in 270 will be compared, along with the answers to the questions to see if any trends in understanding appear or if there are features that are only requested by people in 270 over people who have finished it.

5 Future work

There is a lot of future work for this project. Outside of things like keeping track of and displaying the clock cycles and implementing the functional unit script I would also like to do a few much bigger things

with this project. One of the biggest features that I would like to add would be the ability to run student programs through the simulation, this feature would let students see their own work moving and that might be more useful to them than a few generic examples. Another big feature would be a scalar mode, this would take the simulation and make it so that instead of multiple instructions entering at once all instructions entered one at a time. This would be very useful in helping students understand the differences between scalar and superscalar since they would be much more obvious. Another fun possibility would be to gameify the simulation, this would involve removing the automatic movement of the instructions and make it so the student would select the instructions that could move and where they should go. This could become a fun puzzle and would be a useful way to gauge their understanding of how the processors function.

These big features are still a long way off however, before those there are many small things that should be done first, such as adjustable speed of camera and instructions. And support for an adjustable number of instructions per example, since I am currently capped at 10 due to space limitations, so I would either need a way to shrink the instructions when there are more of them or increase the size of the units to accommodate larger examples.

6 Conclusion

Overall I feel that although there is still a lot that can be done the project of designing an easy to understand simulation of a superscalar processor has been mostly finished. Some of it may not have been implemented yet, but I believe that with the designs that have already been finished implementing the last few things to get the simulator working as intended shouldn't be too much work. This project has many different paths it can go down all of which can be turned into projects of their own based off this core. I mentioned 3 of them (gameifying, scalar, and student programs) in my future work section, but I am sure there are many more possibilities. My hope is that in the future I am able to finish my core implementation for this project and at least start work on one of my larger goals, and we'll see where it goes from there.

7 Glossary

Instruction - A command that tells the processor what type of computation to do, and what data to use for the computation, it also includes information about what register to use to store the result if there is one to be stored.

Registers - Components where data used in computations is stored.

Instruction Memory - Where instructions are stored before entering the processor.

Instruction Fetch - Act of getting instructions from instruction memory.

Register Read - When the processor collects information from the registers for the instruction it is preforming.

Execute Stage - using the execution units to compute the result of an instruction.

Execution Units - The physical units in the processor that preform the computation of an instruction, there are multiple different types including Integer Arithmetic logic units, Floating point Arithmetic logic units and Memory units. These units deal with integer, or whole number instructions, floating point or decimal instructions, and memory or load and store operations respectively.

Register Write - Writing the value of the computation back to the register files

Register Map - A map of the registers where a register entry will show either the current value of the register or a pointer to the instruction that will provide the latest value for a register.

data hazards - A situation where a later instruction is dependent on information from an earlier instruction

Instruction Queue - A queue where instructions are held until their data is ready and they can start to execute. It is a priority queue where the next instruction to leave is the oldest instruction that is ready to execute, meaning it has all of its data, and the processor has the proper hardware available to compute it.

Common Data Bus - A unit in the superscalar processor that comes after the execution units. Sends the results of completed instructions back to the Instruction Queue so that instructions waiting for those results can start executing.

Reorder Stage - ensures that instructions commit in order using the reorder buffer

Reorder buffer - keeps track of what the next instruction that can commit is, then holds all instructions that are not that one to ensure that everything commits in order.

commit stage - shows that an instruction is totally finished, it in this stage when the registers are written to and any holds on registers are removed.

References

[1] David M. Koppelman. Dynamic schedueling, April 2013. http://www.ece.lsu.edu/ee4720/2013/lsli10.pdf.

[2] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.