Frank Hu

Directed Study

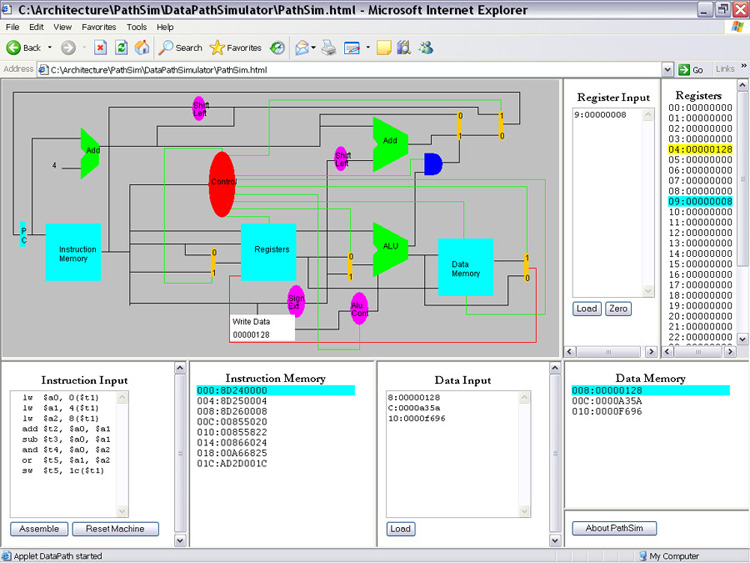
Dr. Barnes

December 10, 2015

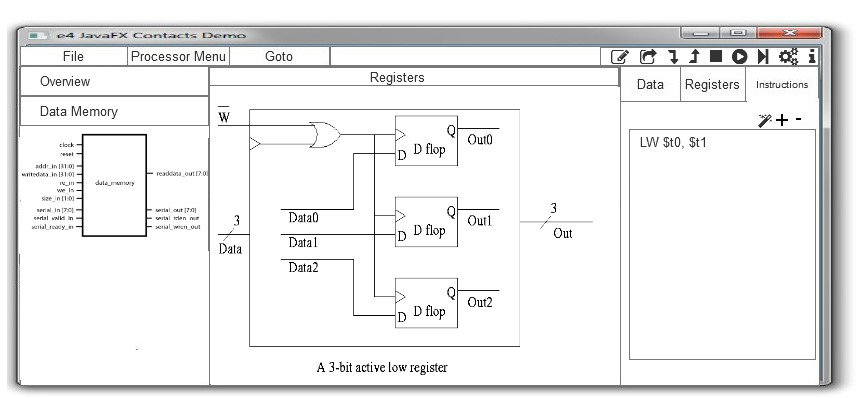
Directed Study Write Up

In the directed study with Brad Barnes and Michael Cotterell we developed a new user interface and a rework of the existing MIPS emulator program by Dr. Hunkins called Pathsim. We named this program “Prometheus”, inspired by the greek god who brought fire to humans. We created a UML of the entire program, and a sequence diagram of one of the components in the program. We decided to use JavaFX since the binding feature in JavaFX simulate the wires perfectly, and could create a real emulation as opposed to Pathsim which utilized more of a simulation approach.

The motivation behind recreating pathsim lies within the userinterface and difficulty using the program.



Above is a picture of what Pathsim looks like.

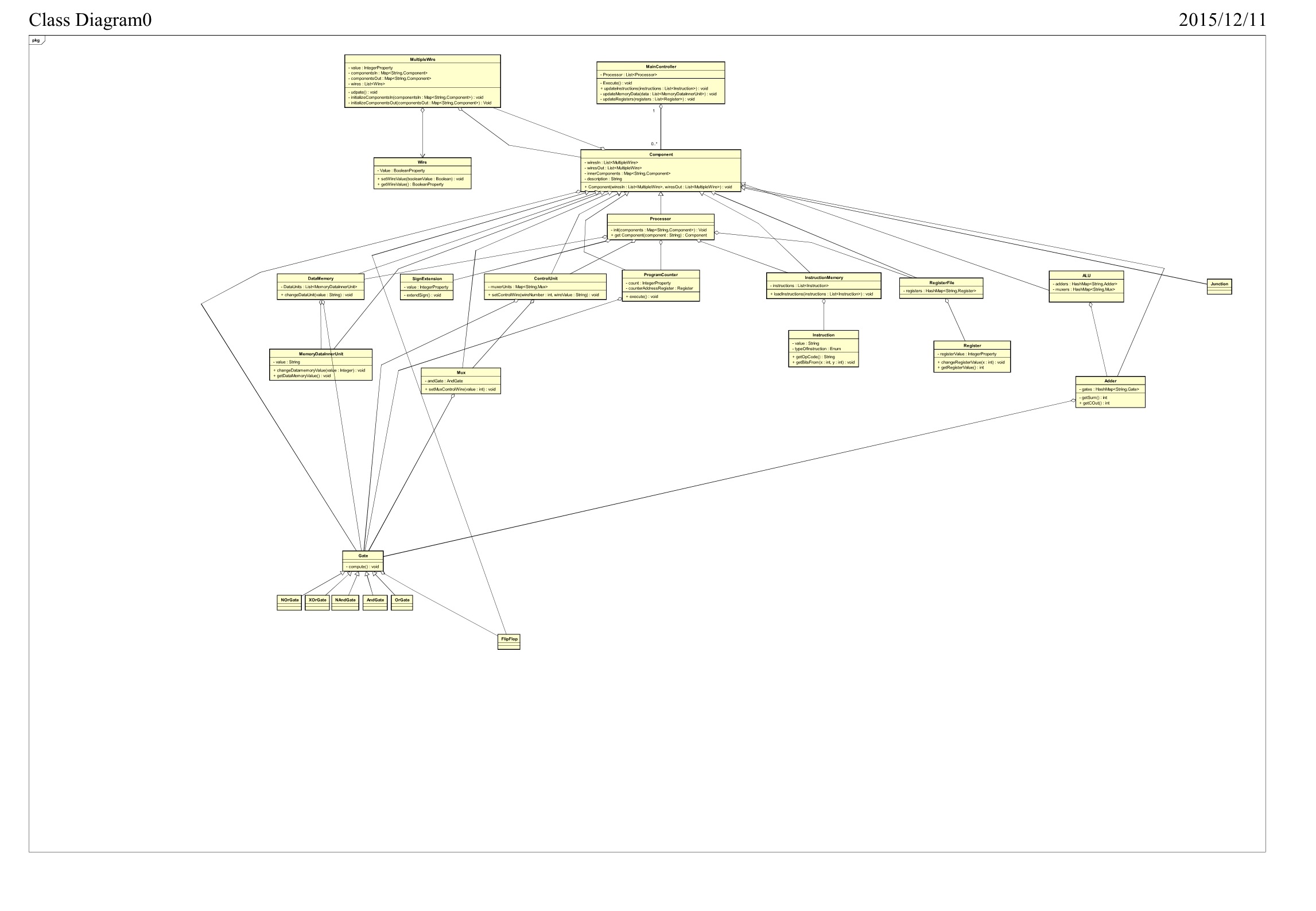
Below is a picture of what we hope to accomplish in Prometheus. 

We began the directed study by deciding on which technology to use. A main problem with Pathsim was that it was programming in a JApplet container medium. The JApplet is a difficult to use program and due to the many updates to Java, is not future proof. As of now, students have extreme difficulty using the JApplet Pathsim, it requires users to downgrade their JRE to the JApplet’s version. From there, they must use the correct browser or it will not work correctly. In order to remedy this, we decided to utilize an approach that doesn’t utilize browser technologies, such as Javax Swing, JavaFX, QT, or some other platform that could run standalone on a computer. We opted to utilize JavaFX since the bindings make it convenient to emulate the processor. This is because when a wire changes, we can bind an onChange function that when changed, will change another device, which will in turn emulate how an actual processor works.

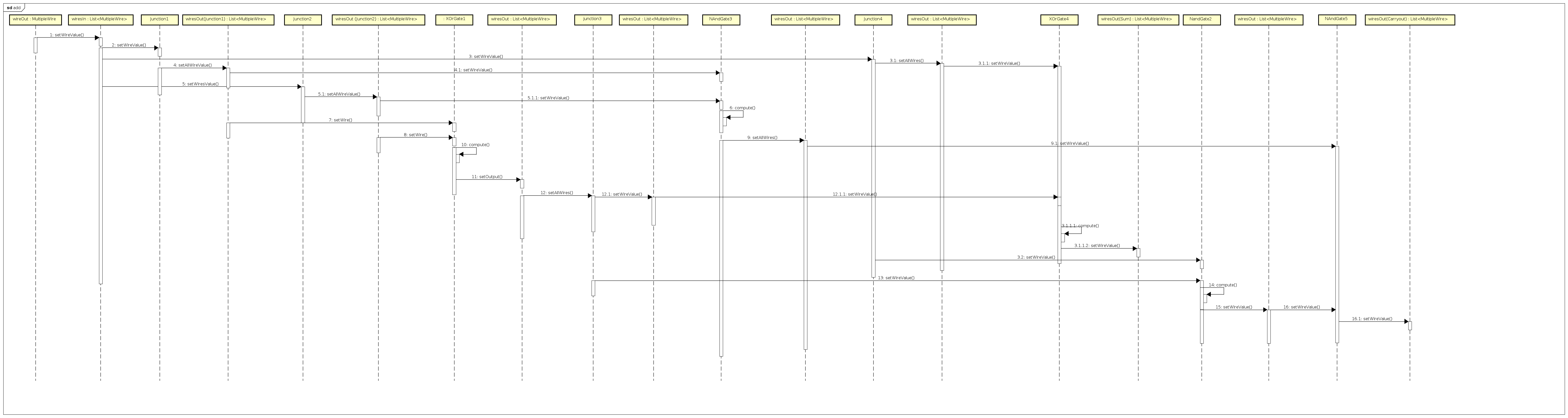
After choosing our technologies, we began creating different user interfaces for the program. We ran through three different sorts of user interfaces and ultimately decided on the interface located in the software design document. We talked about the many different situations and how the user experience should be when using the application. From there we began developing the software design specifications, creating many different scenarios that the user may wish to embark on. Along the way we found many different possible steps that the user may wish to take and talked about different steps, such as a settings option or perhaps a theming option. The most difficult decision we had argued over was how we would handle when a user “zooms” in on a component and wishes to see what is happening inside. How the software act, and what would should change in the user interface, ultimately we decided on putting the “previous views” on the left side of the software, letting users go back accordingly.

After creating the software design specification we began developing the UML of the entire program. I first utilized a poorly developed UML program, and was turned to Astah by Michael Cotterell. Astah was a much better UML creator and we also utilized this program to create our sequence diagrams. To begin with, we began modeling the abstractions of what the processor has. First the component class was created and then the component class broke up into the many different classes that each of the devices within the device should obtain. Many different aggregations and generalizations are shown in the UML and depict a fairly accurate design of what the software should be. These UML pictures, Software Design Specifications, and Sequence Diagrams should be included in the zip with this write up so that you can see the actual results of the directed study.

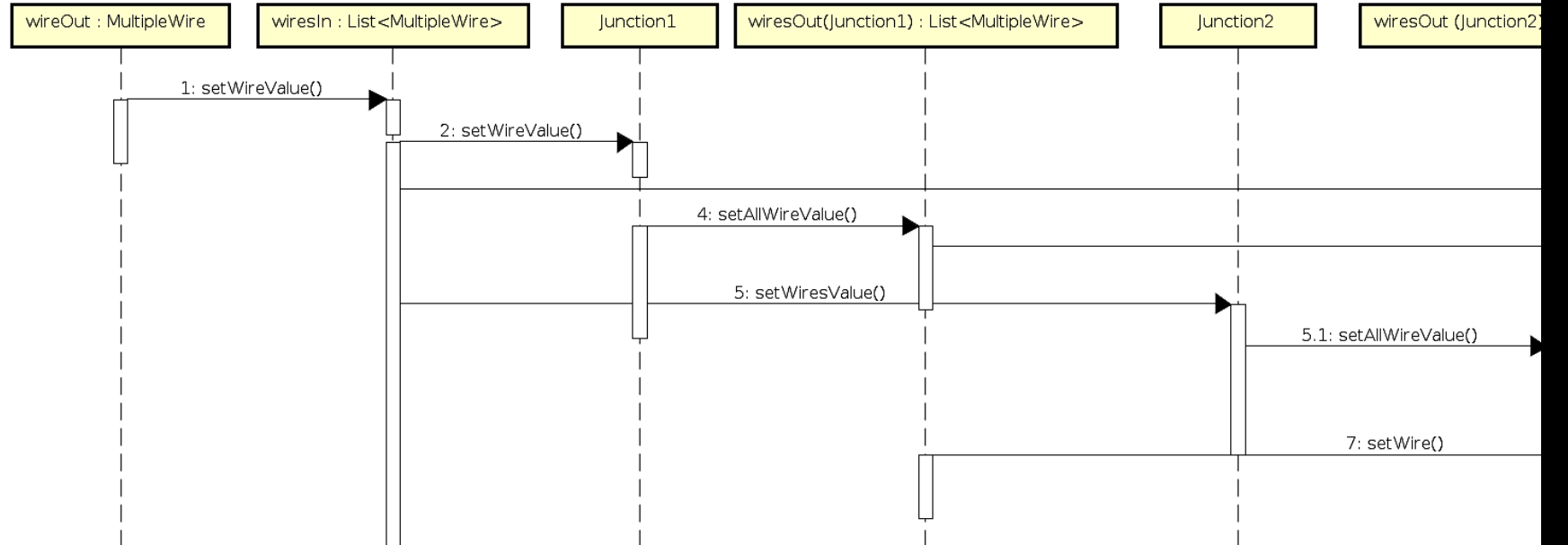
Below is a picture of the UML Diagram of the system architecture of Prometheus.



Below is a sequence diagram of the adder device:



An explanation of the sequence diagram is as follows:



This is the beginning, where the we initialize the hashmap if wireout. We initialize the wiresIn and wiresOut outside of the class.

i.e.

HashMap**<**String**,** MultipleWire**>** inputs **=** **new** HashMap**<**String**,** MultipleWire**>();**

MultipleWire input1 **=** **new** MultipleWire**(**0**);** // A

MultipleWire input2 **=** **new** MultipleWire**(**0**);** // B

MultipleWire input3 **=** **new** MultipleWire**(**1**);** // C\_IN

inputs**.**put**(**"input1"**,** input1**);**

inputs**.**put**(**"input2"**,** input2**);**

inputs**.**put**(**"input3"**,** input3**);**

HashMap**<**String**,** MultipleWire**>** outputs **=** **new** HashMap**<**String**,** MultipleWire**>();**

HashMap**<**String**,** Component**>** noInnerComponents **=** **new** HashMap**<**String**,** Component**>();**

Adder adder **=** **new** Adder**(**inputs**,** outputs**,** noInnerComponents**,** "adder"**,** "Adds two bits"**);**

This is the main driver code in the 1bit adder example. We put these in, we simulate creating the adder and make our own wires that go into the adder. In the application we will have some other device going into it, but as of now we are letting these wires be set as turned on/off.

Then in the adder class. We have the four different junctions as represented in the

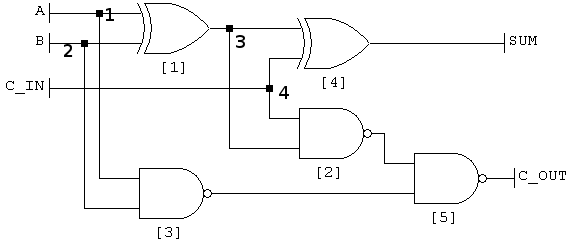


diagram.

First we get junction1, and set it’s input to A, and then we the code in the junction will then set it’s outputs to Nand3, and Xor1 accordingly.

public class Junction **extends** Component **{**

public Junction**(**HashMap**<**String**,** MultipleWire**>** in**,** HashMap**<**String**,** MultipleWire**>** out**,**

HashMap**<**String**,** Component**>** innerComponents**,** String name**,** String description**)** **{**

**super(**in**,** out**,** innerComponents**,** name**,** description**);**

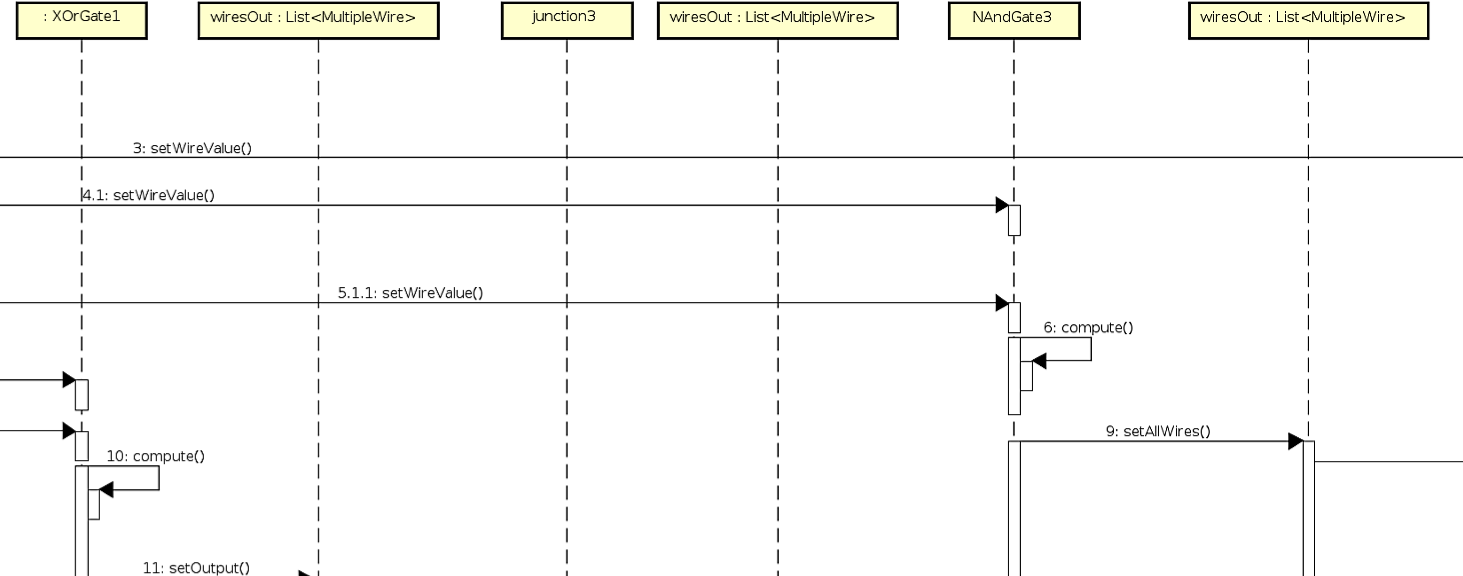
**this.**wiresOut**.**get**(**"output1"**).**getValue**().**bind**(this.**wiresIn**.**get**(**"input1"**).**getValue**());**

**this.**wiresOut**.**get**(**"output2"**).**getValue**().**bind**(this.**wiresIn**.**get**(**"input1"**).**getValue**());**

**}**

**}**

This is done by the code above.



Above is a continuation of the same exact thing happening in junction2, however now we have the XORgate1. The compute function is computed internally by the binding function which is :

public class XorGate **extends** Component **{**

public XorGate**(**HashMap**<**String**,** MultipleWire**>** in**,** HashMap**<**String**,** MultipleWire**>** out**,**

HashMap**<**String**,** Component**>** innerComponents**,** String name**,** String description**)** **{**

**super(**in**,** out**,** innerComponents**,** name**,** description**);**

**this.**wiresOut**.**get**(**"output1"**).**getValue**().**bind**(**Bindings**.**createIntegerBinding**(new** Callable**<**Integer**>()** **{**

@Override

public Integer call**()** **throws** Exception **{**

**return** wiresIn**.**get**(**"input1"**).**getValue**().**get**()** **^** wiresIn**.**get**(**"input2"**).**getValue**().**get**();**

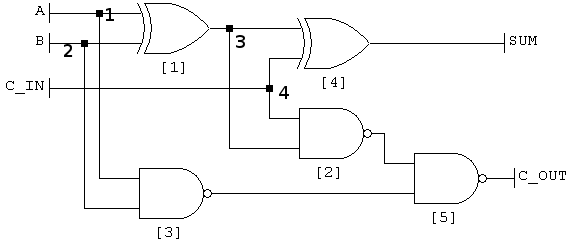
**}**

**},** **this.**wiresIn**.**get**(**"input1"**).**getValue**(),** **this.**wiresIn**.**get**(**"input2"**).**getValue**()));**

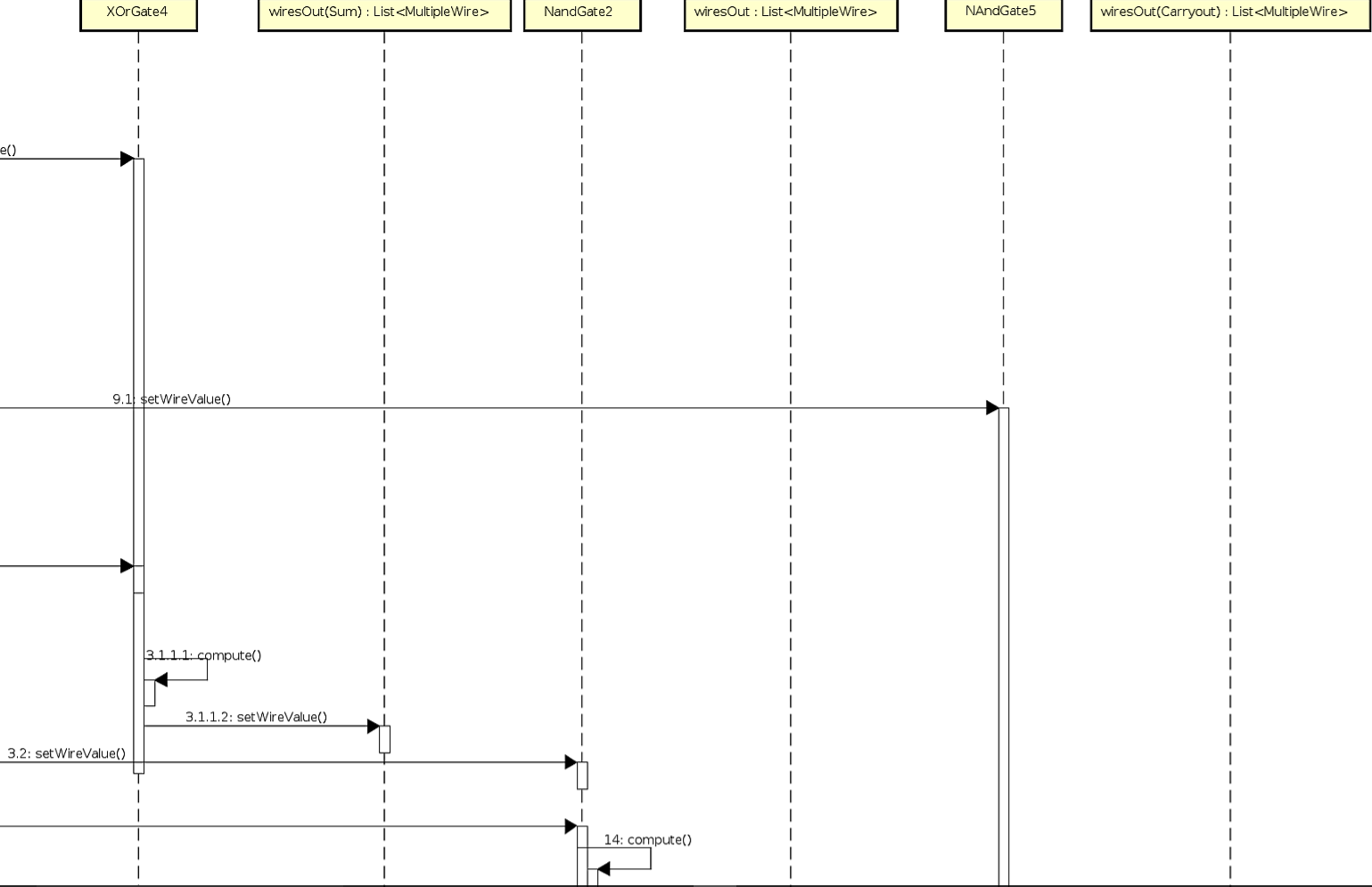
**}**

**}**

The binding will then turn the output into the xor of the inputs received from the xor item. So in actuality the compute function is never called, but for clarity it is included.  (The binding function internally performs the compute() and the setOutput().

This then leads to junction3. Performs the same thing as junctions 1 and 2, however the values go out to XOR 4 and Nand2.

In this part it is similar to the xor gate above. The binding function, for NandGate3 is compute() and setOutput(). The code is as shown :



/\*\*

\* Class NandGate, this class is a little bit different since max/min doesn't work here. I had utilize the create integer binding function to create a new binding

\* to turn the integer input values into boolean values. From there, it utilizes a true nand operation, and then returns 1/0 accordingly.

\* @author Frank Hu

\*

\*/

public class NandGate **extends** Component **{**

NumberBinding output **=** **null;**

public NandGate**(**HashMap**<**String**,** MultipleWire**>** in**,** HashMap**<**String**,** MultipleWire**>** out**,**

HashMap**<**String**,** Component**>** innerComponents**,** String name**,** String description**)** **{**

**super(**in**,** out**,** innerComponents**,** name**,** description**);**

Bindings**.**min**(this.**wiresIn**.**get**(**"input1"**).**getValue**(),** **this.**wiresIn**.**get**(**"input2"**).**getValue**());**

**this.**wiresOut**.**get**(**"output1"**).**getValue**().**bind**(**Bindings**.**createIntegerBinding**(new** Callable**<**Integer**>()** **{**

@Override

public Integer call**()** **throws** Exception **{**

boolean wiresInOne **=** **false;**

boolean wiresInTwo **=** **false;**

wiresInOne **=** wiresIn**.**get**(**"input1"**).**getValue**().**get**()** **==** 0 **?** **false** **:** **true;**

wiresInTwo **=** wiresIn**.**get**(**"input2"**).**getValue**().**get**()** **==** 0 **?** **false** **:** **true;**

**if** **(!(**wiresInOne **&&** wiresInTwo**))** **{**

**return** 1**;**

**}** **else** **{**

**return** 0**;**

**}**

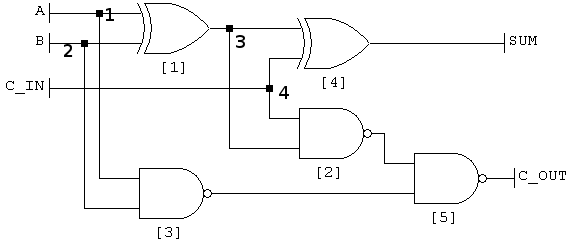
**}**

**},** **this.**wiresIn**.**get**(**"input1"**).**getValue**(),** **this.**wiresIn**.**get**(**"input2"**).**getValue**()));**

**}**

**}**

Xorgate4, is similar to xorgate1. It does the same exact thing. We do not wire this output up to anything, however the hashmap set value of “output1” of the adder is the output of xorgate1.

As we continue through the sequence diagram. We get to Nandgate5 and nandgate2, which in the diagram below creates the output of C\_out. By utilizing the code above, and by simply setting the hashmaps of each of the components of “input1” and “input2” for each of the gates correctly it should output the correct sum and C\_out accordingly.

This adder is the only component that has been built all the way. The rest of the components still need to be built, however this is a good example to start from. The JavaDocs will also be included in this zip file. Please refer to the java doc if you require any more information.

Nick was also included in this directed study and helped model the different components by utilizing sum of products and product of sums rules to generate the different gates that each of the components uses to produce their respective outputs. These components in the file “Components and Sub-Components” should also be included in the zip file, and helps show what each device will have within the software. By modeling each of the and/or/nand gates and showing how the gates are arranged in paper, we are able to model them in the software more accurately with less error. Through the bindings we can change the input and the output from the program counter and from there the program would actually emulate a single cycle process by utilizing JavaFX’s bindings. Now near the end of the program, all we have to do is write the code for the program and model the user interface in a simple UI editor such as SceneFX for javafx, though it is not necessary to utilize SceneFX.

After creating the user interface and the underlying “engine” of the MIPS emulator all that is left to do is to wire up the buttons to the engine. From there, the zoom in functions will need to be implemented on click of a device. This can be done by utilizing JavaFX’s animations, to zoom in on the device’s inner components. Whether or not the components are drawn before or after the animation is not certain yet, and is up to future developers to choose what to do. A large majority of the sequencing for the User Interface has not been created or written about, so that will be the next step for creating the Prometheus. An icon has been provided as the title page for the software design specification, but it is open to new icons, perhaps a picture of the god Prometheus or some other icon. From there assuming the program is finally in its finished state and the basic single cycle process works correctly, one can add pipelining to the emulator by simply adding pipes in between the processor where they belong and emulate memory with the pipes. This should be easily done by simply adding the pipes and changing the bindings of some of the outputs of the devices and some of the inputs of other devices. By utilizing JavaFX’s bindings we are able to easily add, replace, and modify each of the devices since the design is completely modular so that you can remove a device and replace it with another easily. Something that could be improved upon is the abstraction of the devices, we utilize few abstractions and they could perhaps be improved upon. One could possibly add an interface or other abstract class to the program that we did not see previously added.

In order to get a better view of the scope of what was accomplished during this Directed Study, please refer to the software design document at:

<https://docs.google.com/document/d/14JVHQP0aJPjKRfwtySUIZ_icgbrD68uEEqJk_RQV2eE/edit?usp=sharing>

Here you will be able to access higher resolution pictures and be able to zoom in on them with the correct picture viewing software.

Although all the files are included in the zip file, the software design document is much more easily accessible through the google doc form, because the indexing is not working in the pdf.