Visual Debugger for the i281 CPU

Iowa State University
Department of Electrical & Computer Engineering

Team 38 - Spring 2021
Client & Advisor: Alexander Stoytchev
Email: sdmay21-38@iastate.edu
Website: http://sdmay21-38.sd.ece.iastate.edu/

Eric Marcanio
Computer Engineering
marcanio@iastate.edu

Aiman Priester
Computer Engineering
priester@iastate.edu

Brady Kolosik
Computer Engineering
bkolosik@iastate.edu

Colby McKinley
Software Engineering
colbym@iastate.edu

Bryce Snell
Computer Engineering
bsnell@iastate.edu

Jacob Betsworth
Computer Engineering
jtbets12@iastate.edu

Executive Summary

Development Standards & Practices Used

Standard industry convention was used in all software designs. These include:

- ISO/IEC/IEEE 23026:2015
    - Systems and software engineering — Engineering and management of websites for systems, software, and services information
- ISO/IEC/IEEE 15026:2019
    - International Standard - Systems and software engineering
- IEEE 1008-1987
    - IEEE Standard for Software Unit Testing
- IEEE 1284-2000
    - IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers

Summary of Requirements

- Create a lightweight Javascript Web client which simulates the i281 Processor
    - Have a Visualiser that is able to propagate through generated machine code
    - GUI that simulates the switches and seven-segment display from an FPGA
- Develop an assembler to output machine code to the CPU
    - The ability to upload an assembly file or choose from a list of examples
    - A visual component that allows students to understand the conversion between assembly and binary
    - The ability to download generate verilog and machine code
- Create a Verilog model of the i281 processor
    - Have a one-to-one copy in Verilog to allow the students to understand the nuances of Quartus BDF designs and Verilog
    - Provides a framework to build on for their potential CprE281 projects.

Table of Contents

Introduction

Project Background

The i281 CPU is a processor conceptualized and designed by Dr. Stoytchev and former TA Kyung-Tae Kim. It was created, verified, and simulated using exclusively the DE2-115 FPGA. It is a 16 bit processor designed completely in Quartus using simple logic to teach students how the culmination of knowledge taught in CprE 281 (Iowa State's digital logic course) can be used to create a working processor that can run assembly code.

The course builds from boolean algebra into designing complex circuits. Since the course is often a student's first course in the Computer Engineering curriculum, it is challenging to understand at first. At the end of the semester the course introduces assembly code and the fundamentals of a processor, which are very abstract concepts to understand. This is why we are creating a simulation to explain the concept in an in depth and responsive manner. It also works to show students how the connection between assembly and machine code works, as in his original presentations Dr. Stoytchev focuses heavily on how the bits propagate through the processor from the machine code assembled by his custom assembler.

Problem Statement

Due to the requirement of having a DE2-115 FPGA, it would be difficult for students to experience and understand many of the concepts that are taught with this new material, and it is advantageous to do so as another course within the program, CprE 381(Computer Architecture), will require students to learn these ideas in depth to implement their own 32 bit processor that can run standard MIPS assembly language.

As a result of his desire for students to easily access and understand this knowledge, this project was created to help students learn how a CPU works without the strict requirements of the original design.

Problem Solution and Goals

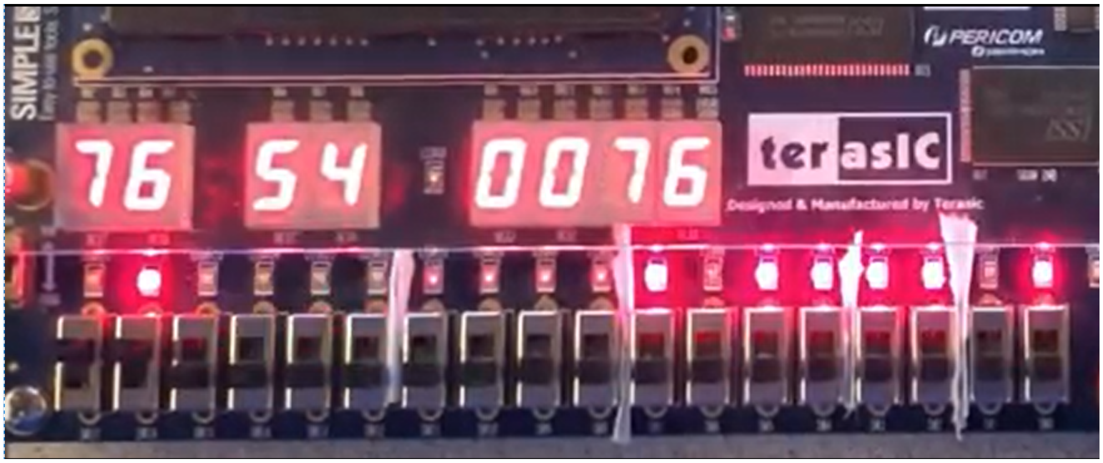
The project is a web app CPU simulation. The user will be able to input assembly language into the JavaScript assembler that we developed, which will in turn translate, or assemble, the inputted assembly into machine code. The assembler will offer a visual component to teach the conversion of assembly to machine code. The visualizer will have access to the values stored within the CPU simulation, and will be able to update certain components to reflect what is happening within the simulation.

In order to mimic the DE-115 FPGA as much as possible, certain aspects of the I/O are also replicated, namely the switches used for input, and the seven-segment displays that are used for outputting values in memory/registers. See Figure 2 below as reference.

Through this project we want to accomplish a working understandable CPU. The main goal is to have the CPU be easy to understand and also simple enough that any user can learn from the app as they work through it.

Aside from this, in order to make future testing and development easier it was also requested that a Verilog version of the project be developed as well, since the original project was exclusively made using block diagram files in Quartus. Along with this, general refactoring and cleaning up of the original BDF files took place to increase readability, and as a result understandability as well.

Figure 1: Altera I/O, Image provided by Dr. Stoytchev.



Conceptual Sketch

Our design for this project is exclusively based on the conceptual sketch that Dr. Stoytchev used in his lecture series, designed by himself to illustrate how a processor looks and is built in a way that is simple to understand. The figure below is the conceptual design.

Figure 2: The original design from Dr. Stoychev's lecture series on the i281 Processor

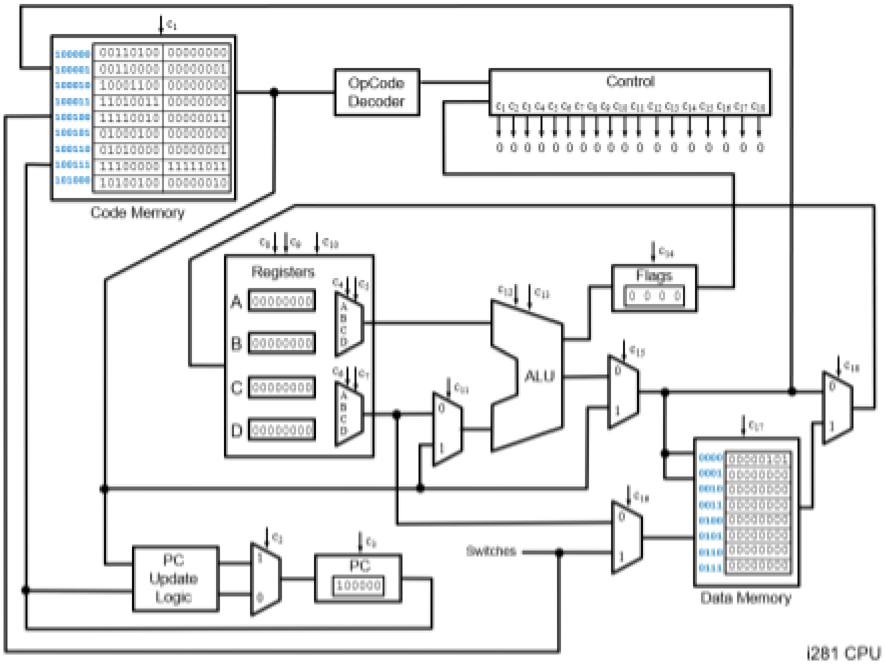


Figure 2: 281 CPU Presentation

Limitations

Looking back upon the expected deliverables from 491, our project manages to firmly meet and exceed these expectations. Additionally we were able to incorporate many of the "wish list" items from our client as well. However, not every "wish list" feature was implemented

- Ability to click inside a register box, change the value, then proceed to execute with the new value
- Ability to change assembly code in the browser after the program has been ran

There are some physical limitations to our project as well. We did not have time to address a lot of technical debt issues that are currently in the code base. This will hinder any future expansion as a lot of code is either not in use of no longer being used effectively.

Intended Users and Uses

This project will be used by Dr. Stoytchev as a teaching tool in his lectures in CPRE 281. It could also be used by students in that class to dive deeper into architecture.

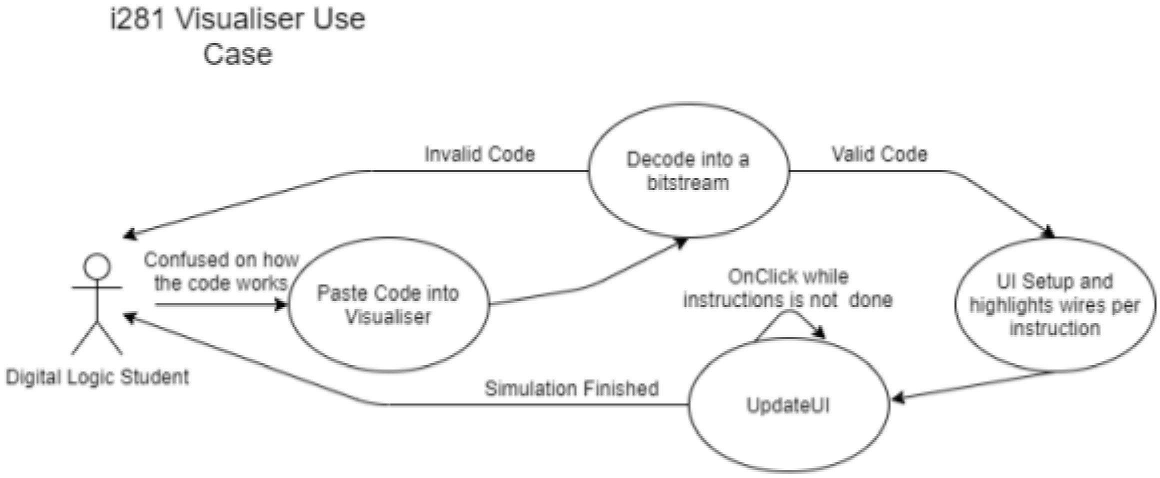

i281 Visualiser Use Case
Invalid Code
Decode into a bitstream
Valid Code
Confused on how the code works
Paste Code into Visualiser
OnClick while instructions is not done
UI Setup and highlights wires per instruction
Digital Logic Student
Simulation Finished
UpdateUI

Figure 3: Flowchart of CPRE 281 student using the visualizer

Related Work

In order to begin work on this project, we first needed to heavily reference Dr. Stoytchev's slides and in essence learn the material that has been added to his curriculum. Most of the group members had previously taken CprE 281 with Dr. Stoytchev, however at that time this material was not yet developed.

From there we split up with each person taking a specific role in project development. The assembler, visualizer, CPU, Altera GUI, and Verilog conversion were all developed separately from each other. There were some concerns with taking this approach, however this will be discussed later on in the report. Throughout Senior Design I, multiple proof of concept experiments were developed and demonstrated to Dr. Stoytchev.

Project Design

Functional Requirements

- Assembler
  - Able to import an assembly file provided by the user
  - Provide example assembly files for ease of use
  - Generate verilog files, machine code, and the assembly code and have it available for download
  - Output assembly code alongside machine code to show the conversion
  - Provide an option to syntax highlight assembly code and corresponding machine code
- Visualizer
  - Must display all critical components of i281 processor
  - Must highlight current executing instruction
    - Follows the highlighting scheme given by the original lecture set provided by Dr. Stoytchev
  - Must highlight relevant data path for current executing instruction
    - The most necessary path for understanding, or the critical path
    - Show syntax highlighted bits propagating through the data paths
  - Must display current data inside the processor
    - Control bits
    - Multiplexor select lines
    - Data Memory
- CPU Simulator
  - Must accurately compute the state of the i281 processor after a cycle
  - State must be accessible by the Visualizer
  - Must be able to initialize memory with data from Assembler
  - Must be able to run fast enough to allow the user to play game
- Verilog Conversion
  - Must be a direct 1-to-1 conversion of the Quartus BDF
  - Must be synthesizable and is able to run all the programs given by the client
  - Must not include any additional components that are not part of the initial student distribution
  - Strictly Verilog and no other additions

Non-functional Requirements

- Assembler
  - Output the machine code very fast
- Visualizer
  - Must be visually appealing

  - Must be intuitive for a 281 student to use and understand
- CPU
  - Process the machine code fast enough to see the output on the seven segment display
- Verilog Conversion
  - Naming conventions must be simple for CprE281 students to understand.
  - Adequate whitespace and proper comments are essential
- Entire project
  - Able to run on most operating systems and browsers

Engineering Constraints

The most important constraint was the restriction of no frameworks in our system. Since we wanted the project to be easily accessible and useful for years to come we created a web application. This requires a browser and access to the internet in order to run the application.

As far as the Verilog conversion is concerned, there should not be any additional libraries in use and must be restricted to Verilog and not any of its derivatives such as SystemVerilog. It also has to be compilable within Quartus Prime and not other compilers such as Icarus.

Previous Design

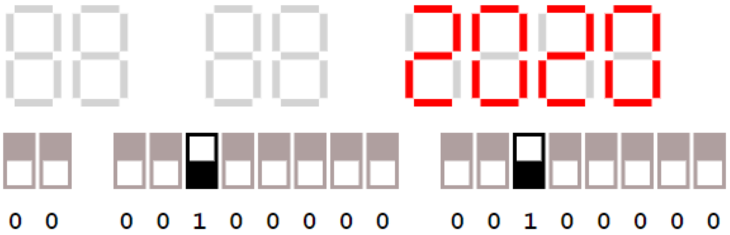
Much of our project was conceptualized and started before the end of Senior Design I, as a result the design prior to this semester and the current design are the same, albeit with many more features and functional requirements added in.

By the end of last semester, we had a working assembler that could error detect and assemble user inputted files, and a partial list of example assembly programs that could be loaded into the assembler as well via a drop-down menu.

On the CPU simulation side of the project, all of the major components of a processor were written and simulated in JavaScript, with the ability to run a single “NOOP” instruction tested and verified working.

For the visualizer, we had all major components and data paths drawn, however the design and appearance of this portion of the project has changed considerably. Figure 4 shows the state of the visualizer prior to the development that has taken place this semester. The Altera GUI was also not included in the main visualizer page as well, as it was still under development separately per the plan outlined in the ***Related Work*** section above. The Altera GUI as the end of last semester can be seen in figure 5.

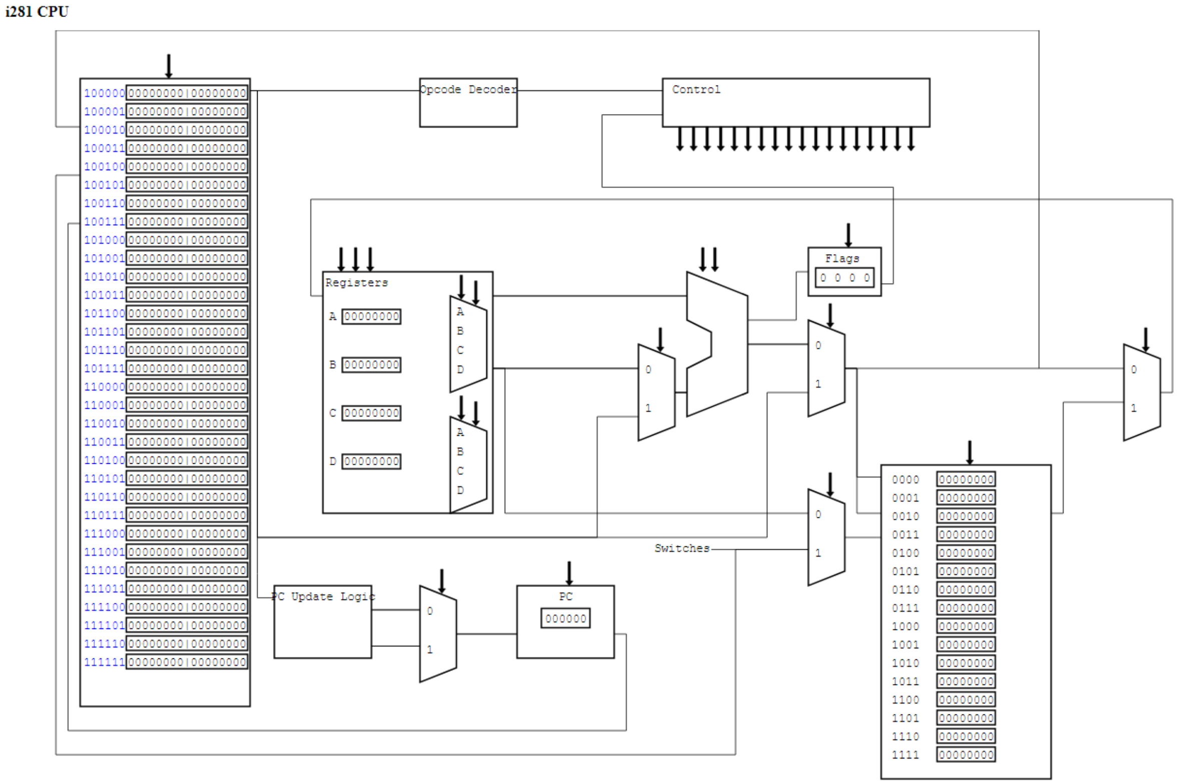
Figure 5: Working concept design for the Altera GUI



Our decisions surrounding testing the implementation of the visualizer has also changed significantly, as there is no real way to test components being drawn or initialized through the use of unit testing.

While the design is the same, the way our components interact with each other is somewhat different than we expected it to be by the end of last semester, see figure 6.

Figure 6: The i281 Visualizer as of 11/12/20

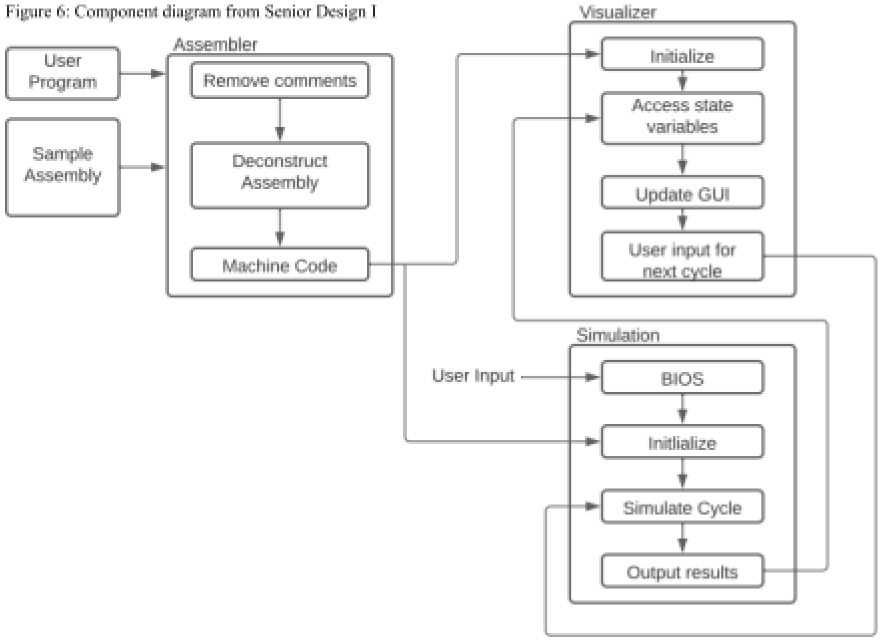


100000	00000000	00000000
100001	00000000	00000000
100010	00000000	00000000
100011	00000000	00000000
100100	00000000	00000000
100101	00000000	00000000
100110	00000000	00000000
100111	00000000	00000000
101000	00000000	00000000
101001	00000000	00000000
101010	00000000	00000000
101011	00000000	00000000
101100	00000000	00000000
101101	00000000	00000000
101110	00000000	00000000
101111	00000000	00000000
110000	00000000	00000000
110001	00000000	00000000
110010	00000000	00000000
110011	00000000	00000000
110100	00000000	00000000
110101	00000000	00000000
110110	00000000	00000000
110111	00000000	00000000
111000	00000000	00000000
111001	00000000	00000000
111010	00000000	00000000
111011	00000000	00000000
111100	00000000	00000000
111101	00000000	00000000
111110	00000000	00000000
111111	00000000	00000000

0000	00000000
0001	00000000
0010	00000000
0011	00000000
0100	00000000
0101	00000000
0110	00000000
0111	00000000
1000	00000000
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

A	00000000
B	00000000
C	00000000
D	00000000

Figure 6: Component diagram from Senior Design I



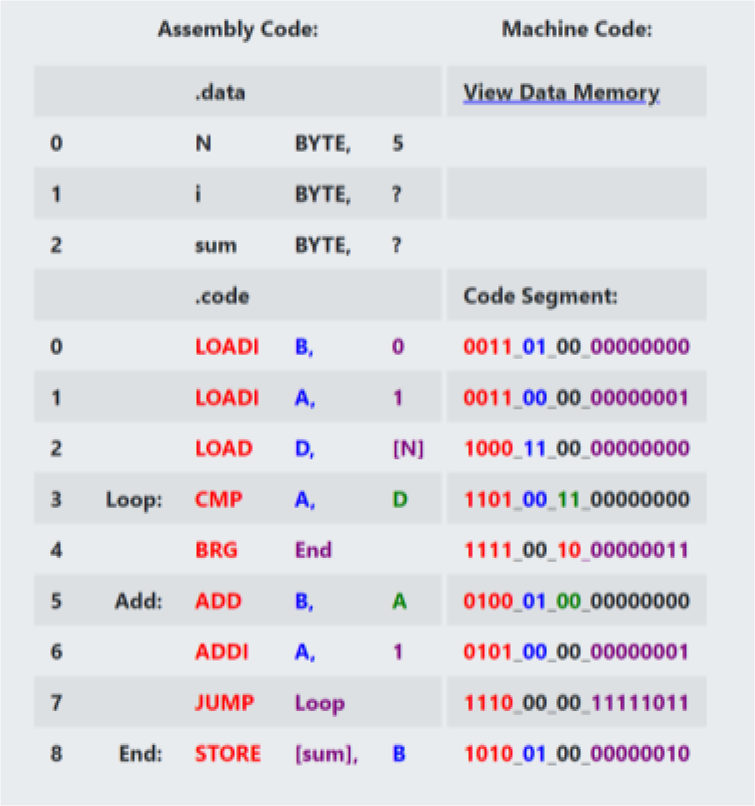
Current Design

Newly Implemented Features

From a Visualization perspective, there are very clear differences since the conclusion of 491.

- Functional Requirements
  - Instruction syntax highlighting
    - Bus highlighting
    - Control arrow highlighting
    - Mux switching
  - Integration of Altera board GUI
  - Addition of descriptions
  - Labeling bus widths
  - Show the current machine code instruction as assembly for understanding
  - Hovering over a line of machine code within the code memory will show the corresponding assembly in a tooltip.
  - Program/file name is shown at the top for user reference.
- Non functional Requirements
  - Changes to fonts to increase readability and visual appeal
  - Repositioning components to balance out GUI
  - Addition of intersections

Figure 7: Assembler Syntax Highlighting



		Assembly Code:			Machine Code:
		.data			View Data Memory
0		N	BYTE,	5	
1		i	BYTE,	?	
2		sum	BYTE,	?	
		.code			Code Segment:
0		LOADI	B,	0	0011_01_00_00000000
1		LOADI	A,	1	0011_00_00_00000001
2		LOAD	D,	[N]	1000_11_00_00000000
3	Loop:	CMP	A,	D	1101_00_11_00000000
4		BRG	End		1111_00_10_00000011
5	Add:	ADD	B,	A	0100_01_00_00000000
6		ADDI	A,	1	0101_00_00_00000001
7		JUMP	Loop		1110_00_00_11111011
8	End:	STORE	[sum],	B	1010_01_00_00000010

Assembler

The assembler is based off of a previously built java assembler. This new assembler features a user friendly interface that allows the user to learn the conversion between assembly and binary. The user can choose between uploading personal assembly files or choose from a list of examples provided. When the assembler is run it outputs the generated machine code along with the assembly. The user can turn on a syntax highlighting features that teaches the conversion between assembly and machine code.

On the left you can see an example of what the Syntax highlighting features looks like with a simple program. The assembler also generates 5 folders that the user can download including the machine code, verilog code, and the assembly code. This view also allows the user to view a pop up table of the data memory to understand how the data is structured in machine code. The assembler saves all of this data in the browser cache and passes it over to the CPU so it can run the generated code.

Visualizer

As alluded to earlier, the design of the visualizer was heavily influenced by the original design from the class slides. Modifications were made in order to fit all components with necessary functionality, one example can be seen with the Instruction memory module where we needed to fit 32 instructions and as a result took up much more vertical space than the original design. Other changes are made to increase the aesthetic, for example the internal wiring of the Register file is new to our design.

In addition to the visual appearance being finalized, the visualizer is also now integrated with the CPU simulation. This means that as the user presses the step button, cycles are simulated on the CPU and the visualizer is updated accordingly to show exactly what is going on inside of the CPU. The Altera GUI is also located to the right of the visualizer. The Altera GUI interacts exclusively with the CPU, it has no interaction with the visualizer as this is not required.

Also, as mentioned above machine code instructions are highlighted in a way that is almost identical to the provided presentations. This alone was not very meaningful, so it was decided that the busses should also be highlighted in the same way, to show how specific bits propagate through the CPU and to further reinforce the idea that the machine code alone determines how and what data flows through a processor, and showing exactly how this binary machine code can act as assembly language at the lowest level. Figures 8 and 9 show an example of the original slides compared to our implementation.

Figure 8: Original lecture material

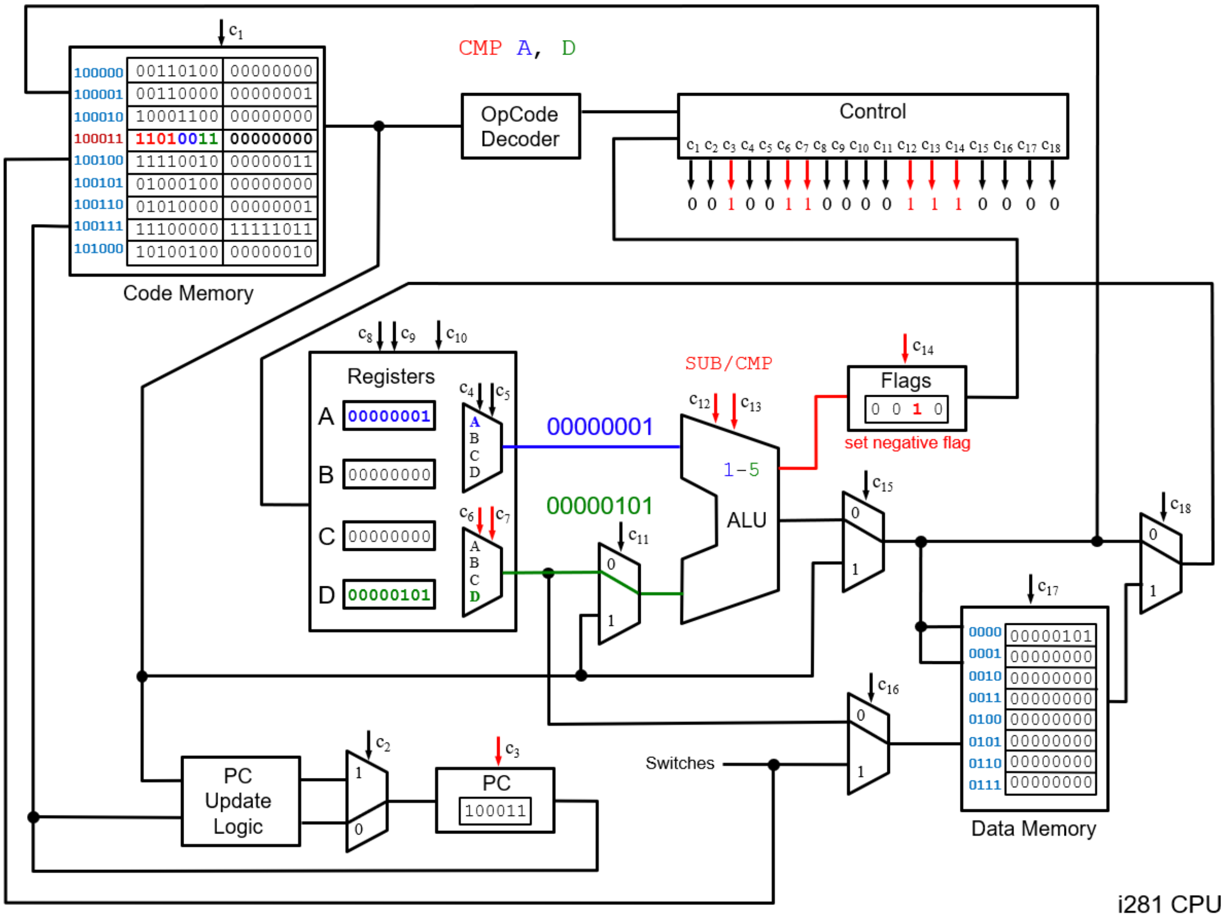
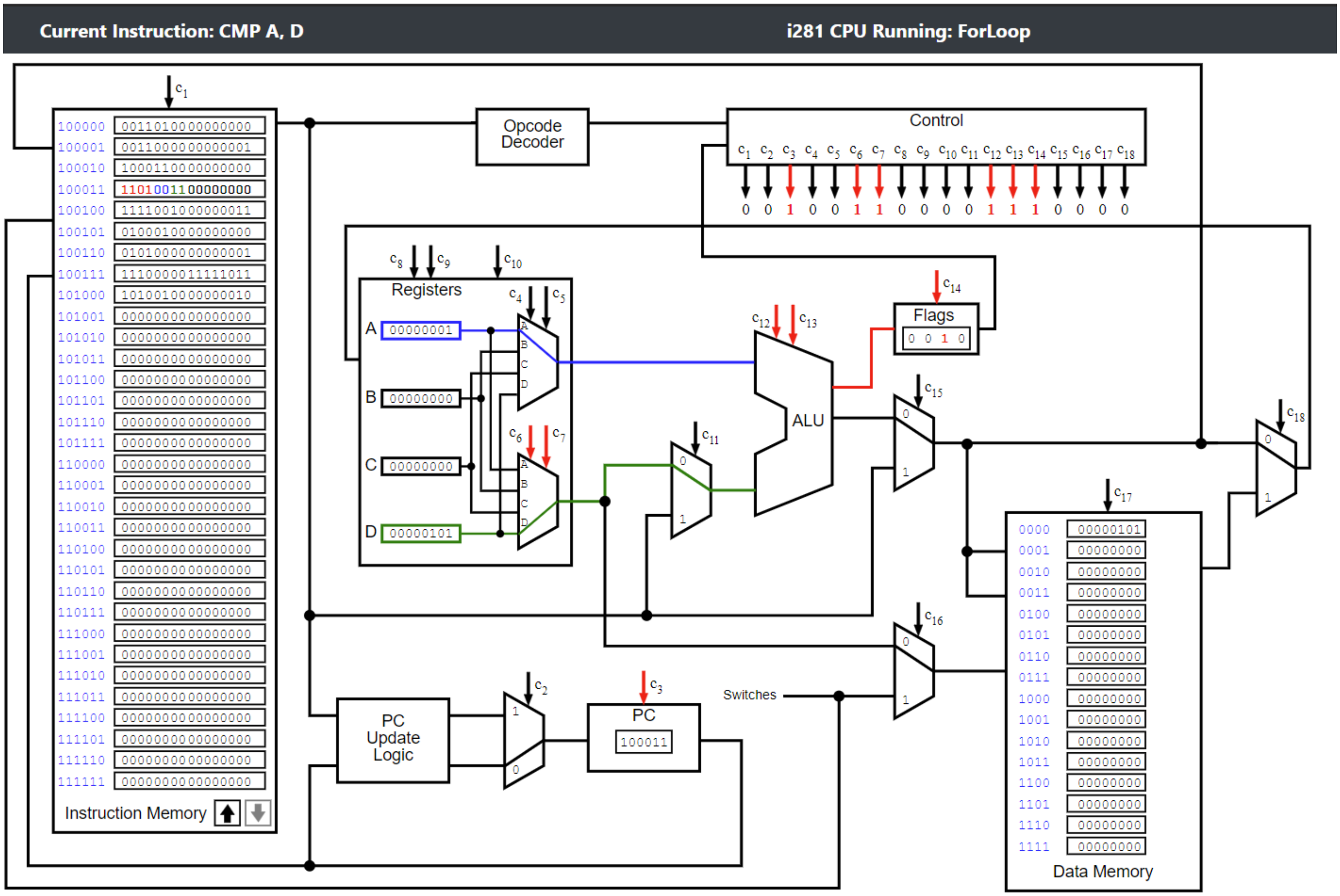


Figure 9: Our implementation



Current Instruction: CMP A, D

i281 CPU Running: ForLoop

Instruction Memory

100000	0011010000000000
100001	0011000000000001
100010	1000110000000000
100011	1101001100000000
100100	1111001000000011
100101	0100010000000000
100110	0101000000000001
100111	1110000011111011
101000	1010010000000010
101001	0000000000000000
101010	0000000000000000
101011	0000000000000000
101100	0000000000000000
101101	0000000000000000
101110	0000000000000000
101111	0000000000000000
110000	0000000000000000
110001	0000000000000000
110010	0000000000000000
110011	0000000000000000
110100	0000000000000000
110101	0000000000000000
110110	0000000000000000
110111	0000000000000000
111000	0000000000000000
111001	0000000000000000
111010	0000000000000000
111011	0000000000000000
111100	0000000000000000
111101	0000000000000000
111110	0000000000000000
111111	0000000000000000

Control

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}
0	0	1	0	0	1	1	0	0	0	0	1	1	1	0	0	0	0

Registers

A	00000001
B	00000000
C	00000000
D	00000101

Flags: 0 0 1 0

PC: 100011

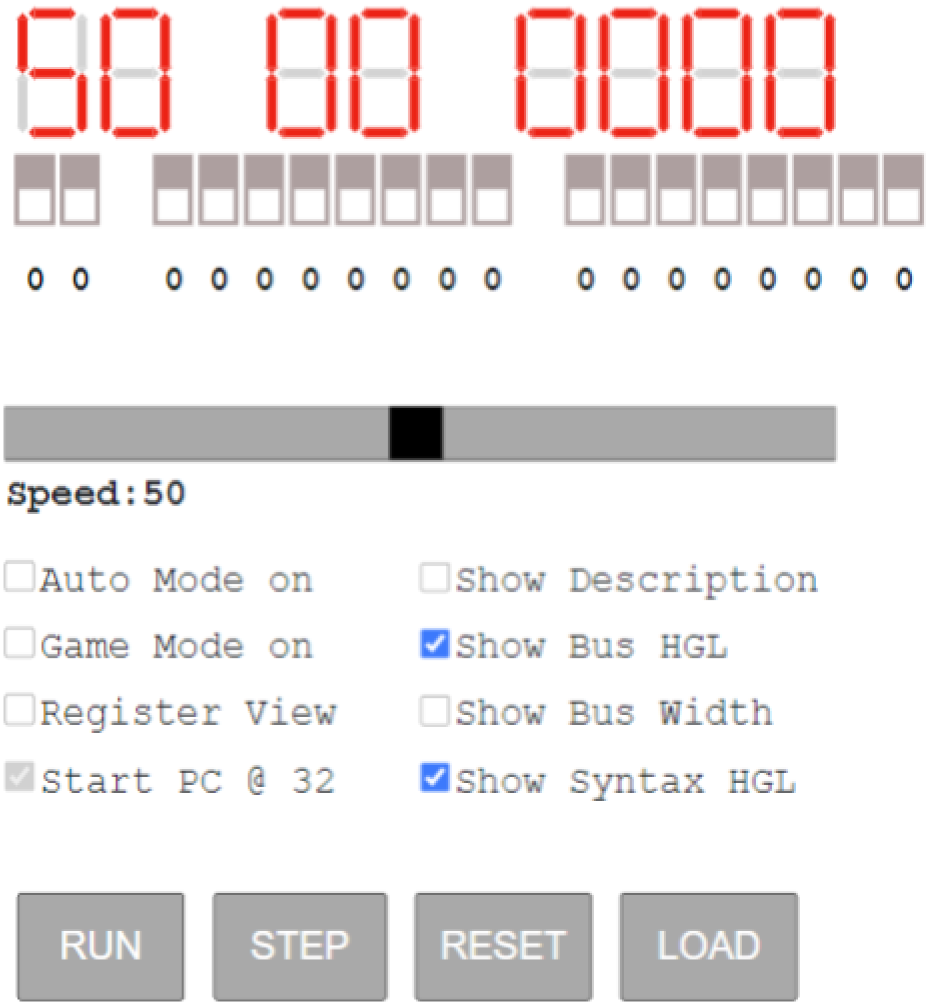
Data Memory

0000	00000101
0001	00000000
0010	00000000
0011	00000000
0100	00000000
0101	00000000
0110	00000000
0111	00000000
1000	00000000
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

As you can see, our implementation contains a little bit less information than the original slides, but the student is able to deduce what is happening because the values are present and easy to read, only the most necessary things are shown to avoid screen clutter and/or visual imbalances.

The Altera GUI has also changed considerably as well, with the seven-segment displays taking on a look more original to the actual DE-115 board, along with some extra checkboxes to control certain functionalities within the visualizer. There are also buttons and checkboxes to control how the simulation is run and displayed, which will be further explained in ***Appendix I*****.** Figure 10 shows a picture of the completed Altera GUI.

Figure 10: The finalized Altera GUI



Speed:50

☐ Auto Mode on
☐ Game Mode on
☐ Register View
☑ Start PC @ 32
☐ Show Description
☑ Show Bus HGL
☐ Show Bus Width
☑ Show Syntax HGL

As demonstrated, many changes have taken place in the development of this project, especially over the last 4 months. The result is an appealing, yet minimalistic design that is maximized for readability to help students understand to the best of their ability how this processor works. It also provides an easy way for Dr. Stoytchev to have an animated example without having to go through the trouble of creating a complex powerpoint presentation, allowing for an increased amount of flexibility in teaching these topics.

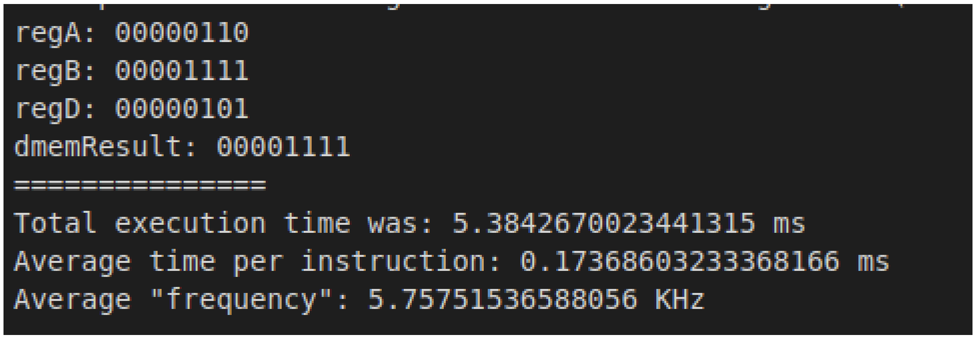
CPU

There were four phases of simulator testing and development this semester. Below are the highlights and milestones of this subproject.

On February 15th the simulator ran its first instruction correctly. During this phase the processor required manually loading instructions to the instruction memory. The only way to read values was through a debugger. However, this would soon be solved.

By February 20th the processor was running a single program that was manually loaded into the processor. This was the first major confirmation that the processor was working as intended. The program that was loaded was a simple for loop that added each number from 1 to 5 and stored it in a register. The screenshot below shows that program outputting the correct value of 15.

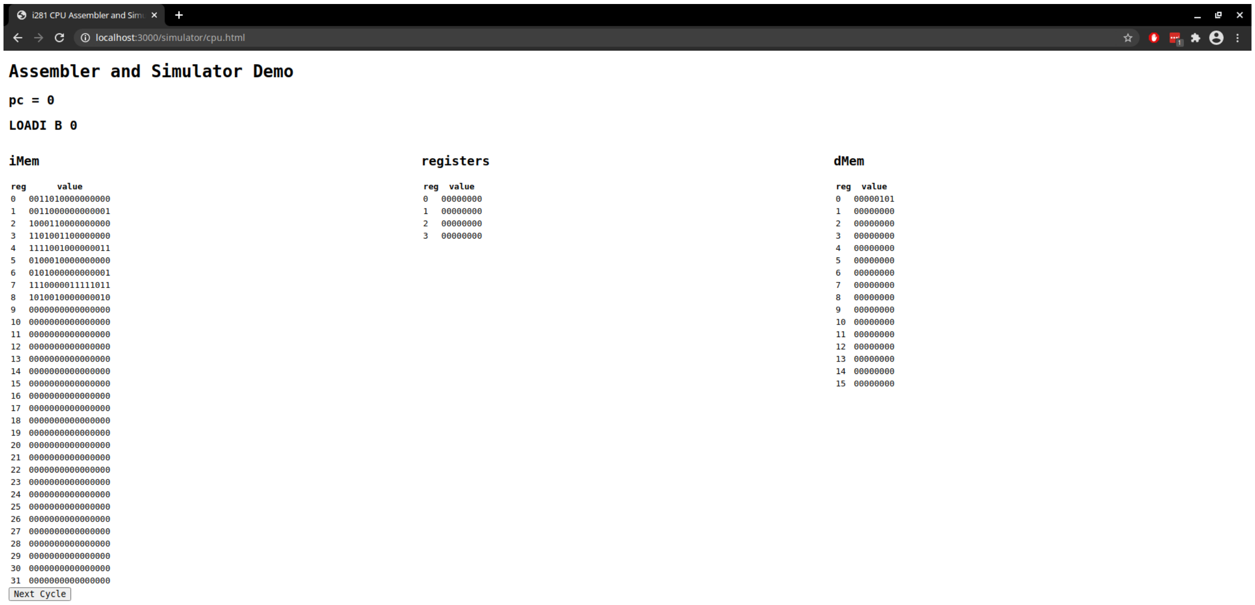
Figure 11: Results from running a program on the CPU simulaton



```
regA: 00000110
regB: 00001111
regD: 00000101
dmemResult: 00001111
===============
Total execution time was: 5.3842670023441315 ms
Average time per instruction: 0.17368603233368166 ms
Average "frequency": 5.75751536588056 KHz
```

The next major jump was the following week. That week the assembler was connected to the simulator. This allowed the simulator to be programmed much more quickly. A demo website was made for the simulator which can be seen below. It was extremely rudimentary (it was designed by the simulator team), but it worked. This allowed the user to step through the program and watch values be updated in the registers and data memory.

Figure 12: Initial proof of concept design showing values inside of processor updating in real time


i281 CPU Assembler and Simu
localhost:3000/simulator/cpu.html
Next Cycle

Assembler and Simulator Demo

pc = 0

LOADI B 0

iMem

reg	value
0	0011010000000000
1	0011000000000001
2	1000110000000000
3	1101001100000000
4	1111001000000011
5	0100010000000000
6	0101000000000001
7	1110000011111011
8	1010010000000010
9	0000000000000000
10	0000000000000000
11	0000000000000000
12	0000000000000000
13	0000000000000000
14	0000000000000000
15	0000000000000000
16	0000000000000000
17	0000000000000000
18	0000000000000000
19	0000000000000000
20	0000000000000000
21	0000000000000000
22	0000000000000000
23	0000000000000000
24	0000000000000000
25	0000000000000000
26	0000000000000000
27	0000000000000000
28	0000000000000000
29	0000000000000000
30	0000000000000000
31	0000000000000000

registers

reg	value
0	00000000
1	00000000
2	00000000
3	00000000

dMem

reg	value
0	00000101
1	00000000
2	00000000
3	00000000
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000000
15	00000000

After this step the simulator was connected to the visualizer (March 14th) and had a few small iterations during this time. The most major change that was added was the ability for the user to program the instruction memory with switches.

Once the simulator was connected to the visualizer it was much easier to look for errors. During this time some small errors were found and fixed. Further improvements were made to allow the user to play the game pong, which was the client's final goal.

As of today the CPU simulator is assumed complete with no known bugs.

Implementation

Our project was developed solely using JavaScript, with no libraries or extra dependencies being used. This approach was taken to maximize the number of years that this web app will be usable, as JavaScript is a powerful scripting language that is not likely to depreciate for a very long time. Libraries would hinder this goal, as at any point a library could be updated, or abandoned, causing dependency issues and/or breaking the program in ways that we cannot predict. Our project can be broken down into parts as mentioned above, these are the **CPU simulation, Visualizer and Altera GUI, assembler, and Verilog conversion.** The specifics of each implementation will be discussed in a list below.

CPU Simulation

The CPU simulator is the connecting piece between the assembler and visualizer. The simple description of the simulator is that it takes assembled instructions from the assembler, computes the state of the processor for each cycle of the processor, then allows the visualizer to access these computed values. This processor was written entirely in JavaScript without any libraries or frameworks. This allows the simulator to have a long life cycle.

The simulator is divided into logic parts based on a traditional processor. There are JavaScript implementations of multiplexers, decoders, ALUs, etc. The signals from each of these are connected to each other based on the i281 design provided by our client. The final CPU object has a function that runs a simulated cycle of the processor. This cycle calculates each state based on the instruction and data memory that are loaded at the beginning of the simulation.

Visualizer and Altera GUI

These specific components make heavy use of Scalable Vector Graphics(SVG), which is a standard that allows for the creation of drawn objects in an HTML file. These objects have properties that are easily accessed through JavaScript code, making them ideal as our visualization must update in real time. All objects drawn on the screen are an SVG object, created through the use of wrapper classes that allow us to construct objects at runtime and generate and draw the entire image when the web-page loads.

The visualizer page also makes use of Bootstrap, which in essence is an open-source and free to use utility that contains predefined CSS to make creating appealing web-pages much easier. This could be seen as a point of weakness in terms of dependencies, however bootstrap is widely used and is expected to be maintained as long as JavaScript is.

Everything else code-wise within the visualizer and Altera GUI are pure JavaScript, there are no other libraries or special dependencies required.

Assembler

The assembler is displayed with the HTML and bootstrap. The functionality is programmed using javascript and jquery. It takes files as input or preloaded examples. The files are parsed and formatted in such a way that the assembler can understand. The application runs through the assembly and stores all of the jump locations in a hashmap. Once it knows the jump location it goes through each instruction and calculates the corresponding binary. It supports over 30 different instructions and also supports multiple versions of each instruction. If a file is uploaded and contains errors then it will bring you to a screen that shows you what lines the errors occurred on and exactly what the error is. The assembler saves all of the data generated into the browser cache so it can be used by the CPU later down the line.

Verilog Conversion

Current design has appropriate naming conventions that were established early. This was followed throughout the entire project. This allows for the ease of comparing the Quartus BDF files directly to the Verilog files. In fact, it is entirely possible to replace an entire BDF module with the Verilog file and it will still work exactly the same. Some module names had to be different due to Verilog's compilation rule of not having a digit as the first character of the name. These modules had an underscore ("_") appended to the front of their name. For example, "2to1mux" was renamed to "_2to1mux". Due to the constraints imposed on language extensions, it was not possible to make the code even more concise. If SystemVerilog was allowed, it would have allowed files such as the 16x16 Register to be simplified by the use of a 2D array. Currently, it has a few more lines than it really needs to.

Operational Environment

It should also be noted that our project has been tested and verified using NodeJS to host a web server, listening on a specific port, however it was discovered that this is not necessarily required as the project will run on a simple web server by just pasting the code in, Dr. Stoytchev was able to host this on his public classes webpage.

A note on NodeJS

NodeJS is a commonly used JavaScript backend that is primarily used to manage file linking, and make it easier to link different JavaScript files together. It is expected to have a long lasting life, but as mentioned above it is not an absolute requirement for our finished project.

This project will also run on every modern browser, It has been tested and verified working on Google Chrome and other Chromium based browsers, as well as Firefox and Internet Explorer. Since Microsoft Edge recently migrated to a Chromium, it is also covered under this category. It is also confirmed working for Safari on MacOS as well.

Standards

Here are some specific parts of the referenced standards that we used, from www.iso.org and https://ieeexplore.ieee.org.

- *ISO/IEC/IEEE 23026:2015*
    - *4.12* Home Page - Page automatically loads the default program in the visualizer per the specifications of the original hardware design
    - *4.22* Responsive Web Design - Page automatically adjusts to user's screen size via the use of SVG objects
    - *4.27* Webmaster - Page will be run and administered by Dr. Stoytchev once the final project is relinquished to him. Multiple group members are currently enrolled or accepted into the masters program for the department of Computer and Electrical Engineering at Iowa State, they may also contribute in helping to maintain the project as well.
    - *4.28* Webpage - Each webpage of the project is indeed a coherent presentation of objects delivered to the user via a web browser.
    - *4.29* Website - All web pages are logically connected to each other and managed by the webmaster/s.
- *ISO/IEC/IEEE 15026:2019*
    - This standard refers specifically to assurance and risk assessment/management of a system of software. We took this into account when designing the project, however it will be running on a university managed server as a simple website. There is minimal risk, as it's scope is extremely limited to a small portion of the engineering college at Iowa State, which is an even smaller portion of the university as a whole.
- *IEEE 1008-1987*
    - This standard has to do with unit testing and testing as a whole, while unit testing was not widely used within our project as the majority of it could not be unit tested in any feasible way, validating the simulation of the CPU and all of its components utilized unit tests
- *IEEE 1284-2000*
    - This standard can be seen in the verilog conversion of the i281 CPU. The Altera DE-115 FPGA connects to computers through the use of a USB type A to mini USB connector. This is a form of serial communication, which has superseded parallel communication. This standard however specifically references "...or equivalent parallel port hardware with new software".

Testing

- Simulator
  - Unit tests for components
  - End to End tests verified manually
- Visualizer
  - Testing done manually
- Assembler
  - Created a program to validate if the machine code output was correct
  - Uploading files of multiple incorrect formats to see if the assembler would detect the errors
  - JS unit testing to validate paths of buttons

Interface Testing

Most testing done on the Visualizer GUI was through manual testing. This process included the tedious loading of the project then verifying that the interactions concluded a desired result. Additionally we tested pathways that were not intended to see the results and patched these as needed after discussing it with our client. As many (unintentionally) stumble upon this path and get lost in an erroneous result.

Results

After countless hours of development and regular meetings with our faculty advisor, we are extremely satisfied with the finished product we have delivered.

The Visual Debugger for the i281 CPU is a packaged web application that includes not only a custom JavaScript assembler, but also contains an all encompassing i281 CPU simulation exclusively written in the same language, tested and verified to follow the design specifications of the original CPU that was developed by Dr. Stoytchev in tandem with Kyung-Tae Kim. Along with the simulation comes a visualizer that is able to animate what is happening within the processor, highlighting specific syntax and showing how data moves through a processor, making the concepts taught in class as easy to understand as they can be. It works as intended and it is expected to be deployed on Dr. Stoytchev's class webpage for students to use as early as next semester when he will be teaching the class.

Conclusions

The entirety of our group poured a lot of time and effort into this project and we feel as though we have delivered a product surpassing original requirements given to us at the start of the year. Since all of us are either computer, or software engineers, this project was very satisfying to work on as it allowed us to take a step back and reflect on our time within the program, especially with the end result being used to help teach a flagship class in the computer engineering department.

Appendix I: Assembler and Visualizer Operation Manual

Getting Started

Typical Use

This project is not intended to be run locally on a user's personal computer, however you are free to attempt to run it locally but we cannot guarantee that it will work as intended. In it's given use, Dr. Stoytchev will provide the user with a link to a page hosted on his class webpage. From there the visualizer and assembler portion of the project will be accessible for use.

Running locally

If for some reason you wish to run the project locally, please follow the following steps to get the project up and running on the loopback or localhost address of your computer. This project is only tested and validating working on either native Linux, or the Microsoft Windows Subsystem for Linux. This means that Linux is a solid requirement to run the entire project.

Installation

This project is designed to run on a NodeJS web server locally, as a result you must first make sure that you have NodeJS 14.x installed on your linux computer.

Use the following link to install the correct version of NodeJS:
https://computingforgeeks.com/install-node-js-14-on-ubuntu-debian-linux/

After this is finished, there are a few extra packages that need to be installed via npm. Run these commands entering 'y' if necessary:

```
npm install esm
npm install commonjs
npm install http
npm install httpserver
```

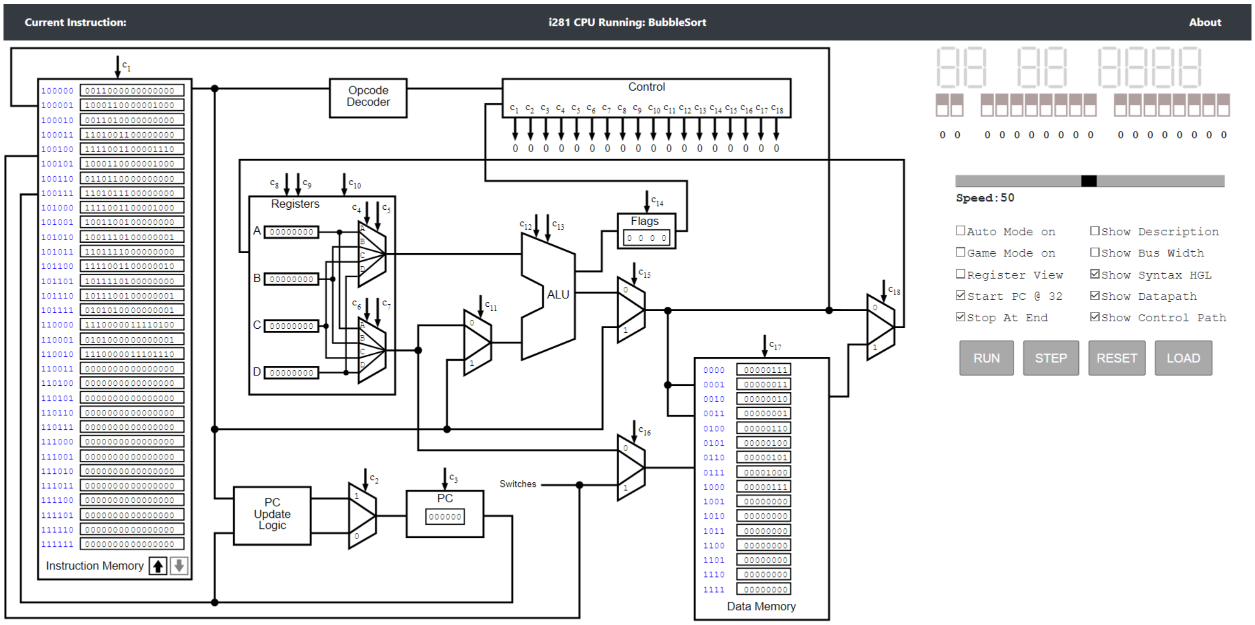
From here, download the provided code base and extract it to a file. Change into the i281 directory and execute the following command:

```
sudo node src/index.js
```

By default this will open a webserver on the loopback address(127.0.0.1) listening on port 3000. You can now access the project by opening up a browser and typing either 127.0.0.1:3000 or localhost:3000 into the address bar.

Visualizer

After navigating to either the link provided by Dr. Stoytchev (or the address specified when running the project locally), you will be greeted by the i281 CPU Simulation and Visualizer.



100000	0011000000000000
100001	1000110000001000
100010	0011010000000000
100011	1101001100000000
100100	1111001100001110
100101	1000110000001000
100110	0110110000000000
100111	1101011100000000
101000	1111001100001000
101001	1001100100000000
101010	1001110100000001
101011	1101111000000000
101100	1111001100000010
101101	1011110100000000
101110	1011100100000001
101111	0101010000000001
110000	1110000011110100
110001	0101000000000001
110010	1110000011101110
110011	0000000000000000
110100	0000000000000000
110101	0000000000000000
110110	0000000000000000
110111	0000000000000000
111000	0000000000000000
111001	0000000000000000
111010	0000000000000000
111011	0000000000000000
111100	0000000000000000
111101	0000000000000000
111110	0000000000000000
111111	0000000000000000

Instruction Memory

0000	00000111
0001	00000011
0010	00000010
0011	00000001
0100	00000110
0101	00000100
0110	00000101
0111	00001000
1000	00000111
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

Data Memory

Speed:50

□Auto Mode on
□Game Mode on
□Register View
☑Start PC @ 32
☑Stop At End
□Show Description
□Show Bus Width
☑Show Syntax HGL
☑Show Datapath
☑Show Control Path

By default the simulation loads the example program "Bubblesort".

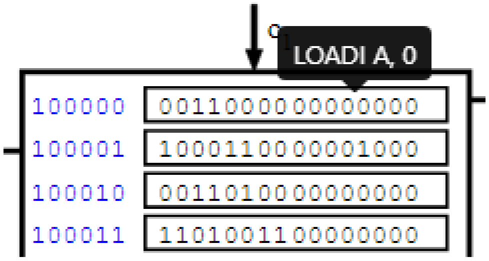
You can see at the top, there is a gray navigation bar. This bar shows the name of the currently running program, the currently running assembly instruction, and also has a button that pops a window up which contains some information about the developers of the project, and the developers of the i281 CPU.

From here you are free to run the simulation and watch the visualizer update values inside the processor in real time. Next we will explain certain functionalities going from the left side of the screen, to the right.

Instruction Memory

The instruction memory shows the machine code that is stored within the CPU. It can show up to 32 lines, and the arrow buttons in the bottom right corner can be used to switch between the lower 32 instructions (0-31), and the upper 32 instructions (32-63).

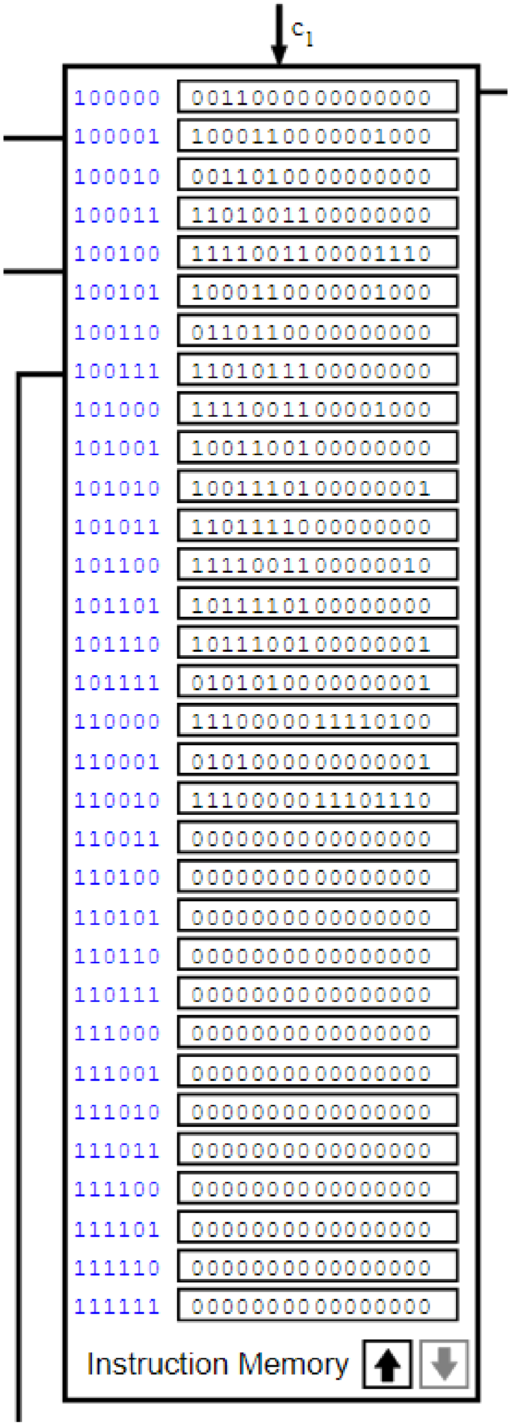
You also have the ability to mouse over any text that is a part of the machine code string to see the corresponding assembly version of the instruction. See below:



```
LOADI A, 0
100000  00110000 00000000
100001  10001100 00001000
100010  00110100 00000000
100011  11010011 00000000
```

The mouse cursor was not preserved in the screenshot, however it is placed over the corresponding text causing the pop-up tooltip to appear.

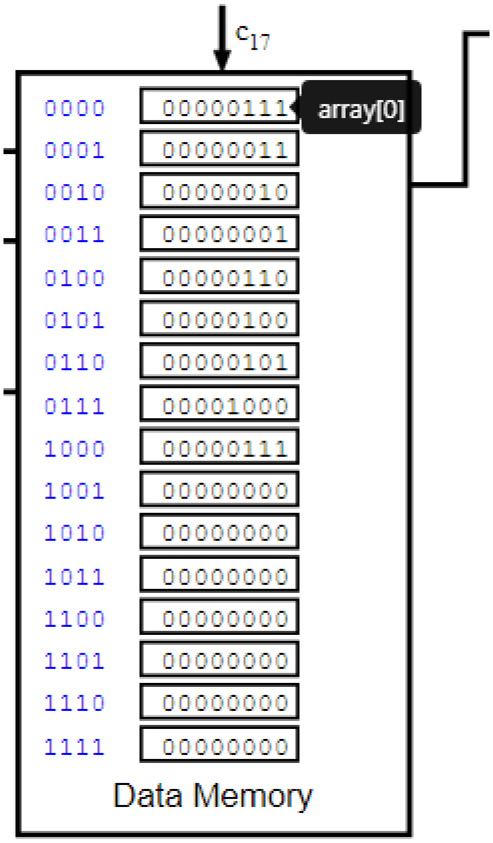
The Instruction memory will also flip between the lower and upper 32 instructions depending on the value of the program counter, and by default every program starts at the 32nd instruction bypassing the first 32. This is because the lower 32 only perform a single "NOOP" instruction followed by a jump to instruction 32. If you wish to know why this is the case, ask Dr. Stoytchev for the specifics of the design of the processor.



```
100000  00110000 00000000
100001  10001100 00001000
100010  00110100 00000000
100011  11010011 00000000
100100  11110011 00001110
100101  10001100 00001000
100110  01101100 00000000
100111  11010111 00000000
101000  11110011 00001000
101001  10011001 00000000
101010  10011101 00000001
101011  11011110 00000000
101100  11110011 00000010
101101  10111101 00000000
101110  10111001 00000001
101111  01010100 00000001
110000  11100000 11110100
110001  01010000 00000001
110010  11100000 11101110
110011  00000000 00000000
110100  00000000 00000000
110101  00000000 00000000
110110  00000000 00000000
110111  00000000 00000000
111000  00000000 00000000
111001  00000000 00000000
111010  00000000 00000000
111011  00000000 00000000
111100  00000000 00000000
111101  00000000 00000000
111110  00000000 00000000
111111  00000000 00000000
```

Data Memory

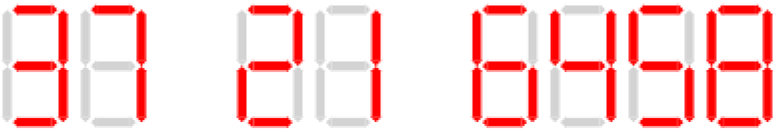
Similar to the instruction memory, the data memory stores eight 16-bit cells of memory. Hovering over the data cell will reveal the comments as parsed by the assembler. See the image on the right for an example of this from the default bubble sort program.

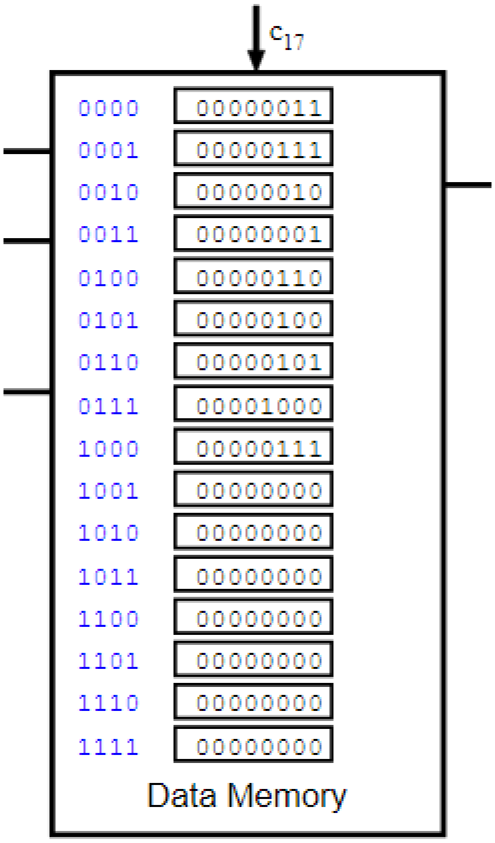


0000	00000111
0001	00000011
0010	00000010
0011	00000001
0100	00000110
0101	00000100
0110	00000101
0111	00001000
1000	00000111
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

Seven Segment Displays

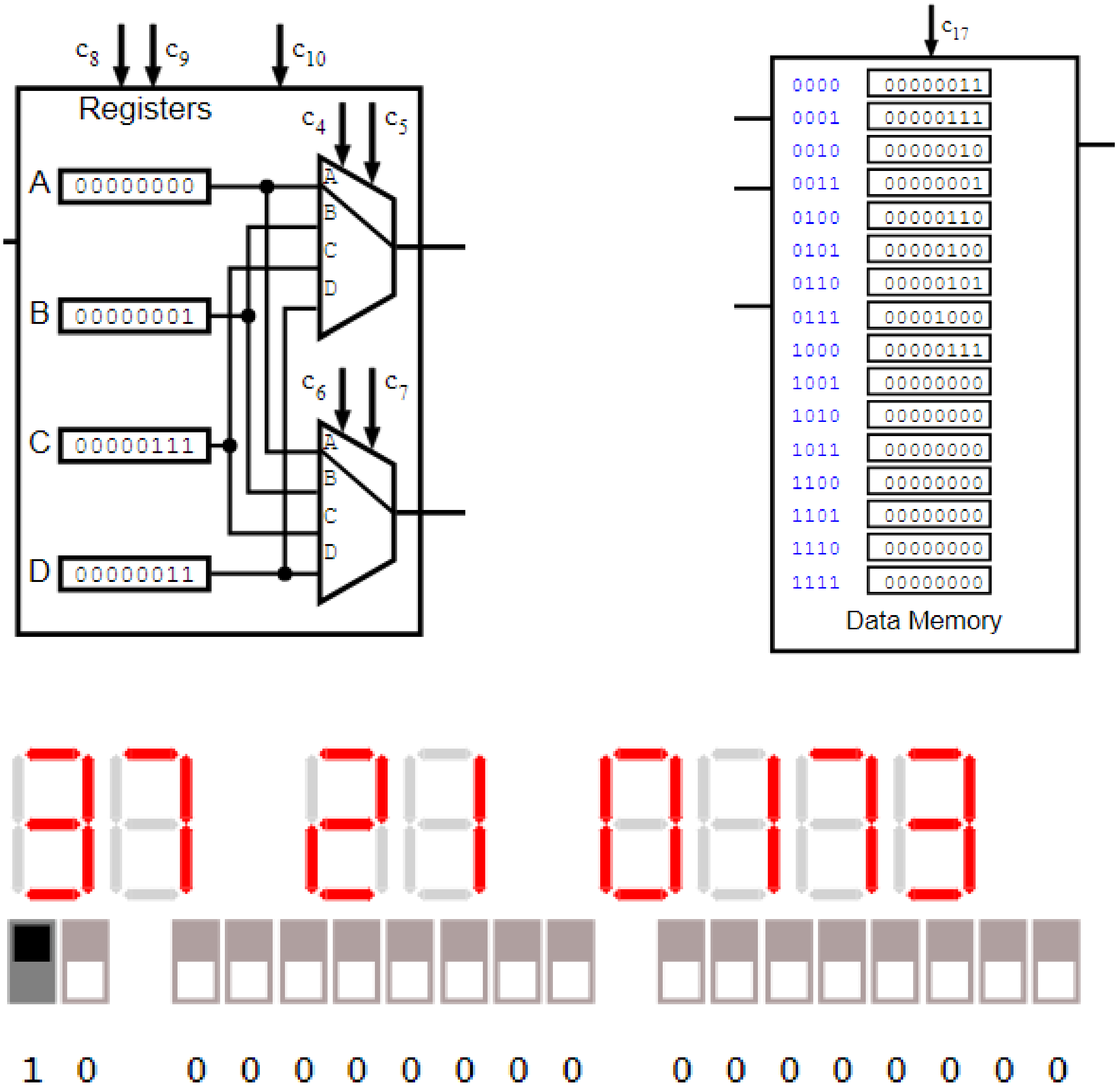
By default the seven segment displays correspond to the first eight blocks of data memory, with the value at 0000 on the far left and the value at 1000 on the far right. The images below will show this clearly.





0000	00000011
0001	00000111
0010	00000010
0011	00000001
0100	00000110
0101	00000100
0110	00000101
0111	00001000
1000	00000111
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

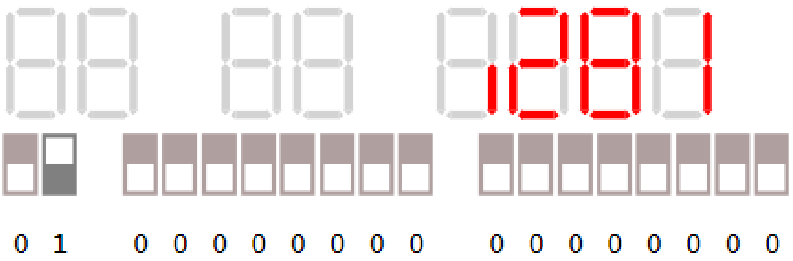
There is also another display mode, this can be accessed by flipping switch 17 up, or checking the register view box below as well. In this mode, the leftmost four displays will show the first values in memory, while the rightmost four will show the values stored in the registers from A to D.

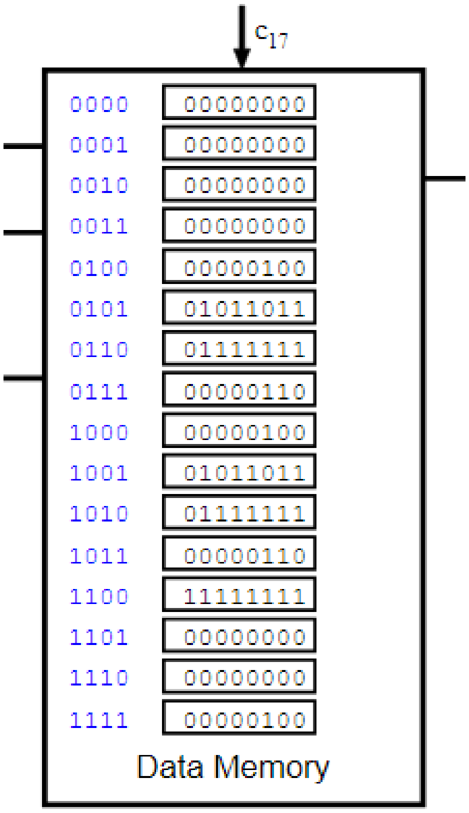


0000	00000011
0001	00000111
0010	00000010
0011	00000001
0100	00000110
0101	00000100
0110	00000101
0111	00001000
1000	00000111
1001	00000000
1010	00000000
1011	00000000
1100	00000000
1101	00000000
1110	00000000
1111	00000000

Above you can see the relation, as well as switch number 17 in the up position.

There is one more additional display mode that is necessary to run the “Pong” example program and play the game. It switches the registers to use a binary encoding, and sets the speed of the simulation to be fast enough to play the game or animate the program. This can be turned on by flipping switch 16 to the up position, or checking the “Game Mode On” box. If you would like to know the specifics of this binary encoding, please ask your instructor.

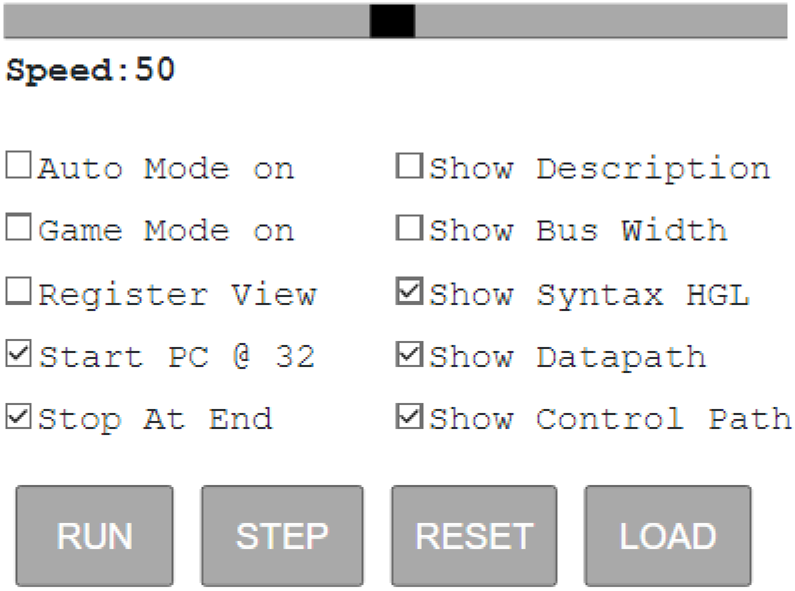




0000	00000000
0001	00000000
0010	00000000
0011	00000000
0100	00000100
0101	01011011
0110	01111111
0111	00000110
1000	00000100
1001	01011011
1010	01111111
1011	00000110
1100	11111111
1101	00000000
1110	00000000
1111	00000100

Controls

Below the seven segment display is an area that can be used to control how the simulation is run, and what the visualizer looks like while it is run.



Speed:50

□Auto Mode on
□Game Mode on
□Register View
☑Start PC @ 32
☑Stop At End

□Show Description
□Show Bus Width
☑Show Syntax HGL
☑Show Datapath
☑Show Control Path

Speed Slider, Auto Mode on checkbox, and run button

These three elements all work together with each other. The Auto Mode on checkbox will force the simulation to run cycles at a regular interval, and this interval is controlled by the speed slider. The run button will cause the Auto Mode on box to be checked, thus turning on the aforementioned mode.

Game Mode on

As mentioned above in the display section, this box will turn on game mode and force the 16th switch into the upper position. This flips the display encoding mode to the binary mode, and causes the simulation to run at a very fast rate. It should be noted that checking this box also disables the datapath being displayed as well to prevent flashing colors on the screen.

Register View

Also as mentioned above, this flips the 17th switch and causes the display to switch to the register view mode.

Start PC @ 32

This box controls where the simulation starts the program. It is checked by default, meaning that the program will always start at 32(unless in some specific cases). Unchecking this box will force the simulation to start at instruction 0. Once a program has started, the checkbox becomes grayed out. This is because the checkbox directly changes the value of the program counter in the simulation, and this is not something that should be changed externally once a program is running.

In some special cases, this box is unchecked by default. This happens when a program is longer than 32 instructions, such that the bios space is used to store actual code instead of the default bios used for simpler programs. In this case the simulation must start at 0, as the code in the instruction memory is not neglectable. For more information about this, ask your instructor about the details of the lower 32 instructions in the instruction memory.

Show Description

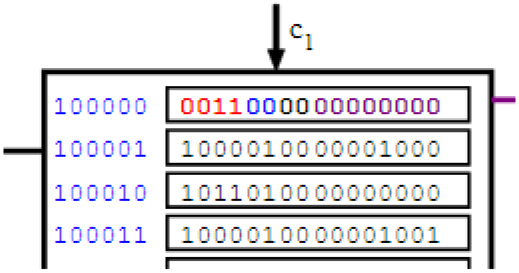
This checkbox toggles text on the screen that displays extra information about the various connections and signals displayed on the visualizer. It closely follows Dr. Stoytchev's lecture slides. No image will be added here, as you can simply check the box to see what it does.

Show Bus Width

This checkbox toggles tic marks on the various busses and connecting wires within the processor, showing how many bits are propagated on each line. This once again closely follows the lecture slides for the i281 CPU, and once again an image will not be provided here.

Show Syntax HGL

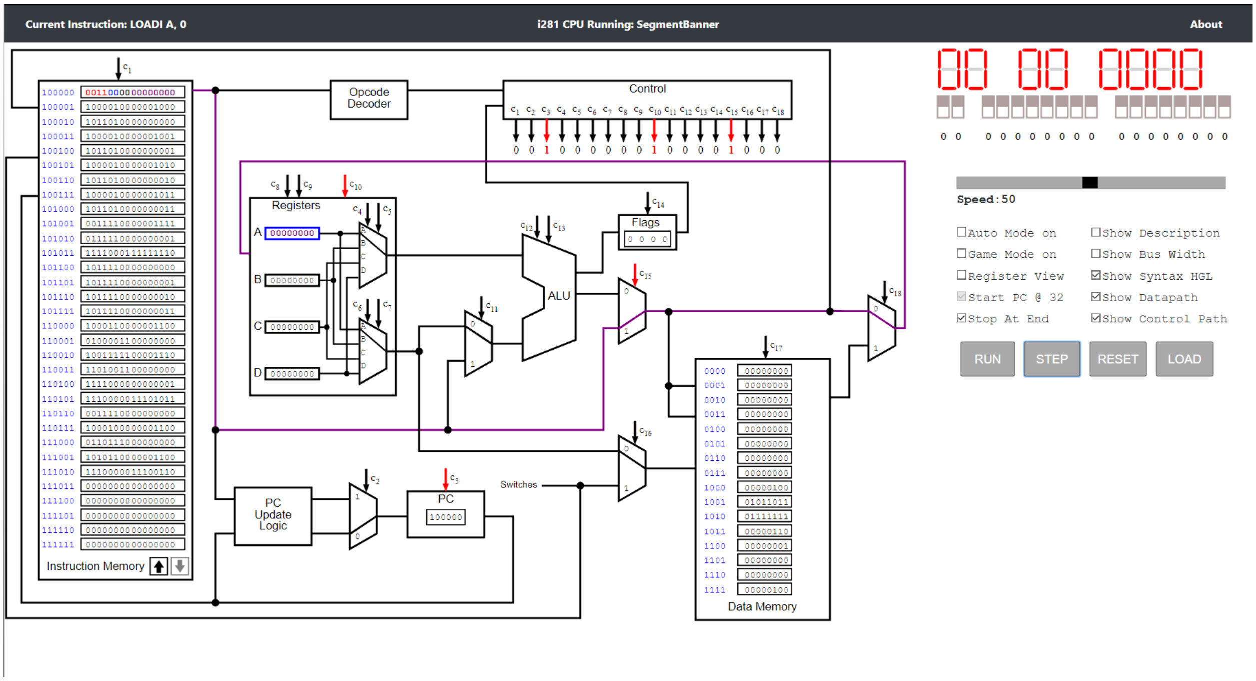
This checkbox toggles syntax highlighting for the bit-strings in the instruction memory. It will highlight the current instruction that is running in the simulation. This syntax highlighting follows the same format as the highlighting in the assembler that you can see below. It is checked by default, and when unchecked the visualizer will not highlight the current instruction anymore. Below is what a LOADI instruction looks like with this highlighting enabled.



```
100000  0011000000000000
100001  1000010000001000
100010  1011010000000000
100011  1000010000001001
```

Show Datapath

This checkbox determines whether or not the data path is highlighted to show how the bits in the machine code propagate through the processor. It is checked by default, and when disabled it will not display any colors along the busses and connections in the processor. It works in tandem with the syntax highlighting, using the same colors to show how parts of the machine code go through the processor. Below shows a LOADI instruction with the datapath being shown.



Current Instruction: LOADI A, 0

i281 CPU Running: SegmentBanner

```
100000 0011000000000000
100001 1000010000001000
100010 1011010000000000
100011 1000010000001001
100100 1011010000000001
100101 1000010000001010
100110 1011010000000010
100111 1000010000001011
101000 1011010000000011
101001 0011110000001111
101010 0111110000000001
101011 1111000111111110
101100 1011110000000000
101101 1011110000000001
101110 1011110000000010
101111 1011110000000011
110000 1000110000001100
110001 0100001100000000
110010 1001111100001110
110011 1101001100000000
110100 1111000000000001
110101 1110000011101011
110110 0011110000000000
110111 1000100000001100
111000 0110111000000000
111001 1010110000001100
111010 1110000011100110
111011 0000000000000000
111100 0000000000000000
111101 0000000000000000
111110 0000000000000000
111111 0000000000000000
```

Instruction Memory

```
0000 00000000
0001 00000000
0010 00000000
0011 00000000
0100 00000000
0101 00000000
0110 00000000
0111 00000000
1000 00000100
1001 01011011
1010 01111111
1011 00000110
1100 00000001
1101 00000000
1110 00000000
1111 00000100
```

Data Memory

Control: 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0

Registers: A 00000000, B 00000000, C 00000000, D 00000000

Flags: 0 0 0 0

PC: 100000

Switches: 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0

Speed:50

□Auto Mode on
□Game Mode on
□Register View
☑Start PC @ 32
☑Stop At End
□Show Description
□Show Bus Width
☑Show Syntax HGL
☑Show Datapath
☑Show Control Path

Show Control Path

This box will cause the control bits, and as well as the control arrows to be colored red when the corresponding text value is 1. This is enabled by default, and you can see what this looks like in the image above from the *Show Datapath* section.

Step

This button, when clicked, will cause the processor to run a single cycle and update the visualizer according to the checkboxes that have been discussed above.

Reset

This button will reload the page, resetting the simulation to its original state before running any instructions.

Load (Subject to name change)

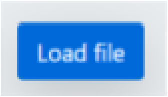
This button redirects the user to the assembler landing page, where they are able to load a new example program or their own custom assembly file and run it through the cpu as discussed below.

Assembler

Uploading a file

When you first open the assembler you can choose between uploading an assembly file or choosing from a list of examples. The "load file" button can be found in the center of the screen and prompts the user to upload a file. The user can also click the "Examples" button in the top right corner and choose from the list of examples.

Figure 13: Load file Button



Assembly Format

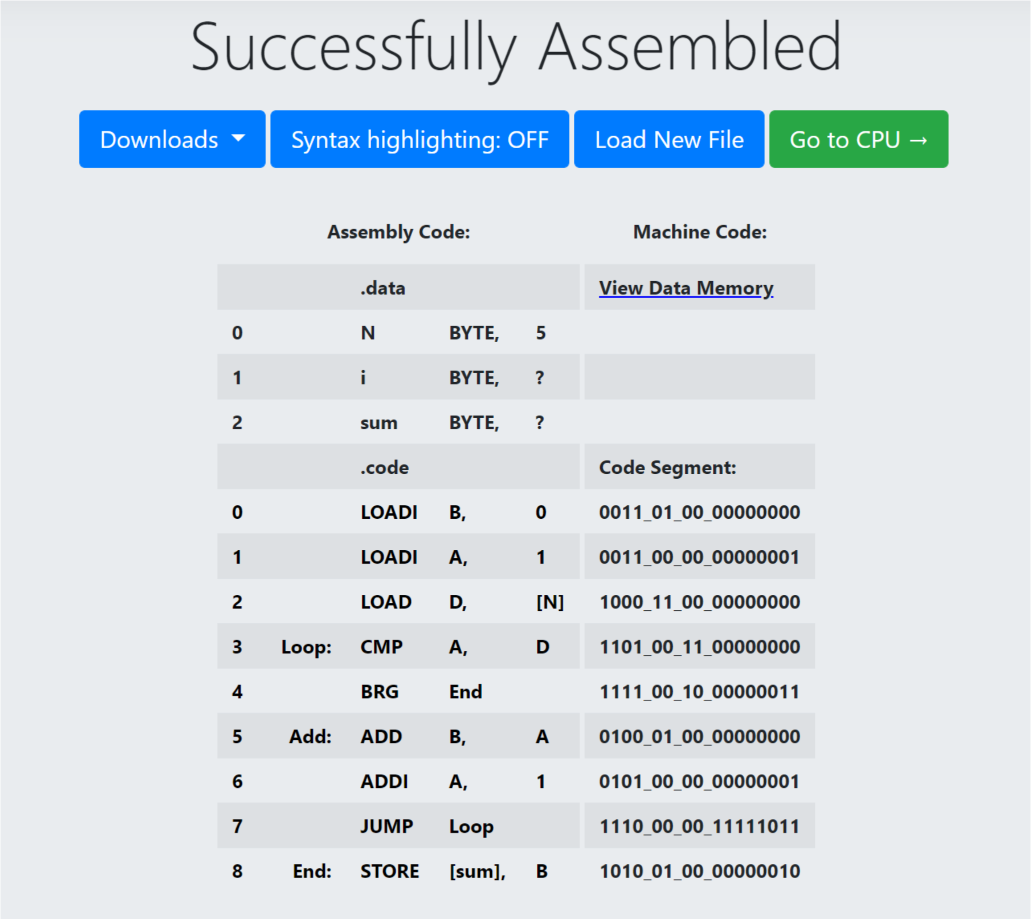
Below is a table of all the instructions supported and their format (The instructions are not case sensitive)

Opcode	**Description**
NOOP	No Operation
InputC	Input into Code memory
InputCF	Input into Code memory with Offset
InputD	Input into Data memory
InputDF	Input into Data memory with Offset
Move	Move the contents of one register to another
LoadI	Load Immediate values
LoadP	Load Pointer values
Add	Add two registers
AddI	Add an Immediate value to a register
Sub	Subtract two registers
SubI	Subtract an Immediate value from a register
Load	Load from a data memory address into a register
LoadF	Load with an Offset specified by another register

Store	Store a register into a data memory address
StoreF	Store with an offset specified by another register
ShiftL	Shift Left all bits in a register
ShiftR	Shift Right all bits in a register
CMP	Compare the values in a register
Jump	Jump unconditionally to a specified address
Bre	Branch if equal
Brne	Branch if not equal
Brg	Branch if greater than
Brge	Branch if greater than or equal

Viewing Assembly Output

Below is an example of what the output looks like when a user runs a simple for loop in the assembler.



Successfully Assembled

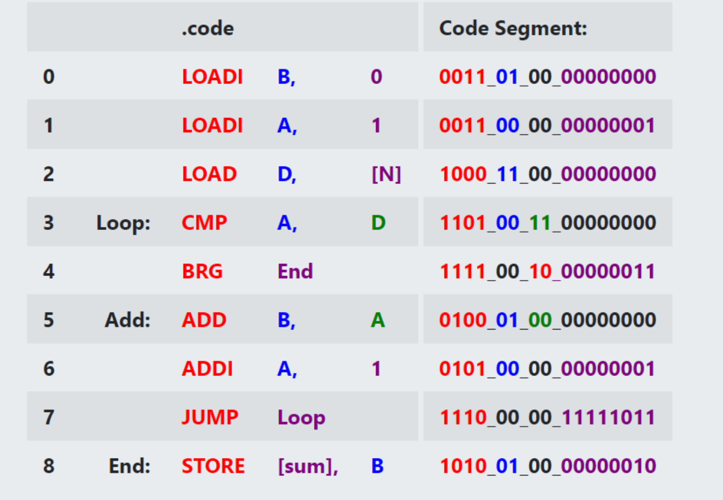
		Assembly Code:			Machine Code:
		.data			View Data Memory
0		N	BYTE,	5	
1		i	BYTE,	?	
2		sum	BYTE,	?	
		.code			Code Segment:
0		LOADI	B,	0	0011_01_00_00000000
1		LOADI	A,	1	0011_00_00_00000001
2		LOAD	D,	[N]	1000_11_00_00000000
3	Loop:	CMP	A,	D	1101_00_11_00000000
4		BRG	End		1111_00_10_00000011
5	Add:	ADD	B,	A	0100_01_00_00000000
6		ADDI	A,	1	0101_00_00_00000001
7		JUMP	Loop		1110_00_00_11111011
8	End:	STORE	[sum],	B	1010_01_00_00000010

Looking at the buttons at the top of the screen:

- Downloads- This button drops down a list of files that the assembler created that are available for download. The user can download Machine code, verilog user code low, verilog user code high, verilog user data, and the assembly code. All of these files are generated through the assembly code run.

- Syntax Highlighting- When turned on this enables the user to learn the conversion between assembly code and machine code. On the right is an example when the button is clicked. From the picture you can see that the instructions match up with the machine code and allow the user to understand the conversion.

- Load New File- This brings the user back to the main page to reupload a file.

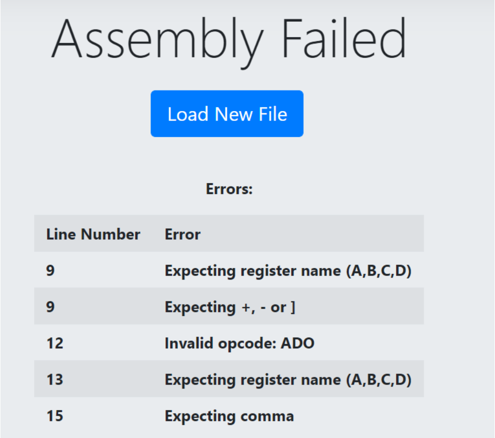


		.code			Code Segment:
0		LOADI	B,	0	0011_01_00_00000000
1		LOADI	A,	1	0011_00_00_00000001
2		LOAD	D,	[N]	1000_11_00_00000000
3	Loop:	CMP	A,	D	1101_00_11_00000000
4		BRG	End		1111_00_10_00000011
5	Add:	ADD	B,	A	0100_01_00_00000000
6		ADDI	A,	1	0101_00_00_00000001
7		JUMP	Loop		1110_00_00_11111011
8	End:	STORE	[sum],	B	1010_01_00_00000010

- Go to CPU- When clicking this button the machine code that was generated in the assembler is sent over to the CPU visualiser and then the user is able to run their program.

Receiving an Error

If the user uploads an assembly file with errors in the program the assembler will alert you that the assembly has failed and show you where the errors occurred. On the right you will see that the assembly file loaded in has a few errors. The error detection will notice incorrect register names, incorrect characters, invalid opcodes, and also if commas are missing.



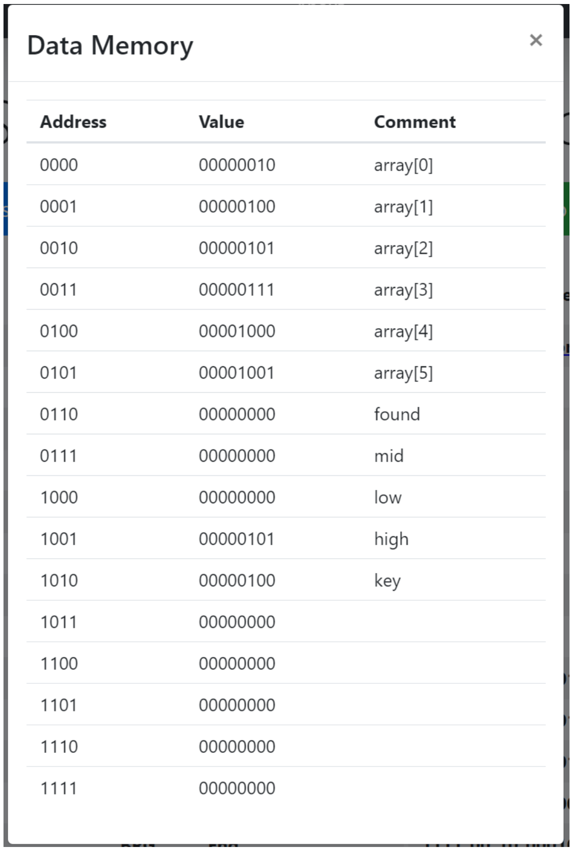
Assembly Failed

Errors:

Line Number	Error
9	Expecting register name (A,B,C,D)
9	Expecting +, - or]
12	Invalid opcode: ADO
13	Expecting register name (A,B,C,D)
15	Expecting comma

Viewing Data memory

If the user would like to view the contents of the data memory they can click “View Data Memory” in the machine code table. This will display a pop up showing the contents of the data memory. On the right is an example of the data memory when Binary search is assembled. It displays the address, value, and variable name of all the contents inside data memory. If an array is supplied like shown in the figure you are able to view each index of the array.



Data Memory

Address	Value	Comment
0000	00000010	array[0]
0001	00000100	array[1]
0010	00000101	array[2]
0011	00000111	array[3]
0100	00001000	array[4]
0101	00001001	array[5]
0110	00000000	found
0111	00000000	mid
1000	00000000	low
1001	00000101	high
1010	00000100	key
1011	00000000	
1100	00000000	
1101	00000000	
1110	00000000	
1111	00000000	

Appendix II: Other Considerations

During the initial brainstorming process for this project, the team had multiple different ideas before settling on using exclusively JavaScript and HTML for the project. As a result, we had no other concrete implementations and only experiments that have not persisted, as they were just testing. They will be discussed briefly below.

Web Assembly

Web assembly(WASM) is a utility that allows for the compilation of in our case C or C++ that would allow us to run the processor simulation in C or C++ rather than Javascript. This offers somewhat of an advantage, as Javascript does not have pointers or anything that can be used to replicate them, instead requiring callback functions and making “linking” different parts of the processor simulation more difficult. This however was not a good solution to our problem, as our client wished to have a project that would be long lasting and have very few dependencies or points of failure, as mentioned above. It was ultimately decided that we would exclusively use Javascript instead, and there was only one iteration of the project outside of the initial “Hello World” tests that we implemented using WASM.

HTML Canvas

Before we decided to use the SVG standard for creating and displaying objects on an SVG canvas, there were experiments done using a simple HTML canvas to draw figures and shapes, as well as change the colors of these objects and potentially text as well. There was nothing wrong with this approach, however HTML canvas did not have the same support for changing attributes as SVG, and SVG tended to look much nicer and scaled much better as the screen would zoom in and out. We decided to use SVG for this reason, and there was no other implementation aside from what we iterated on to produce the end product.

Sources

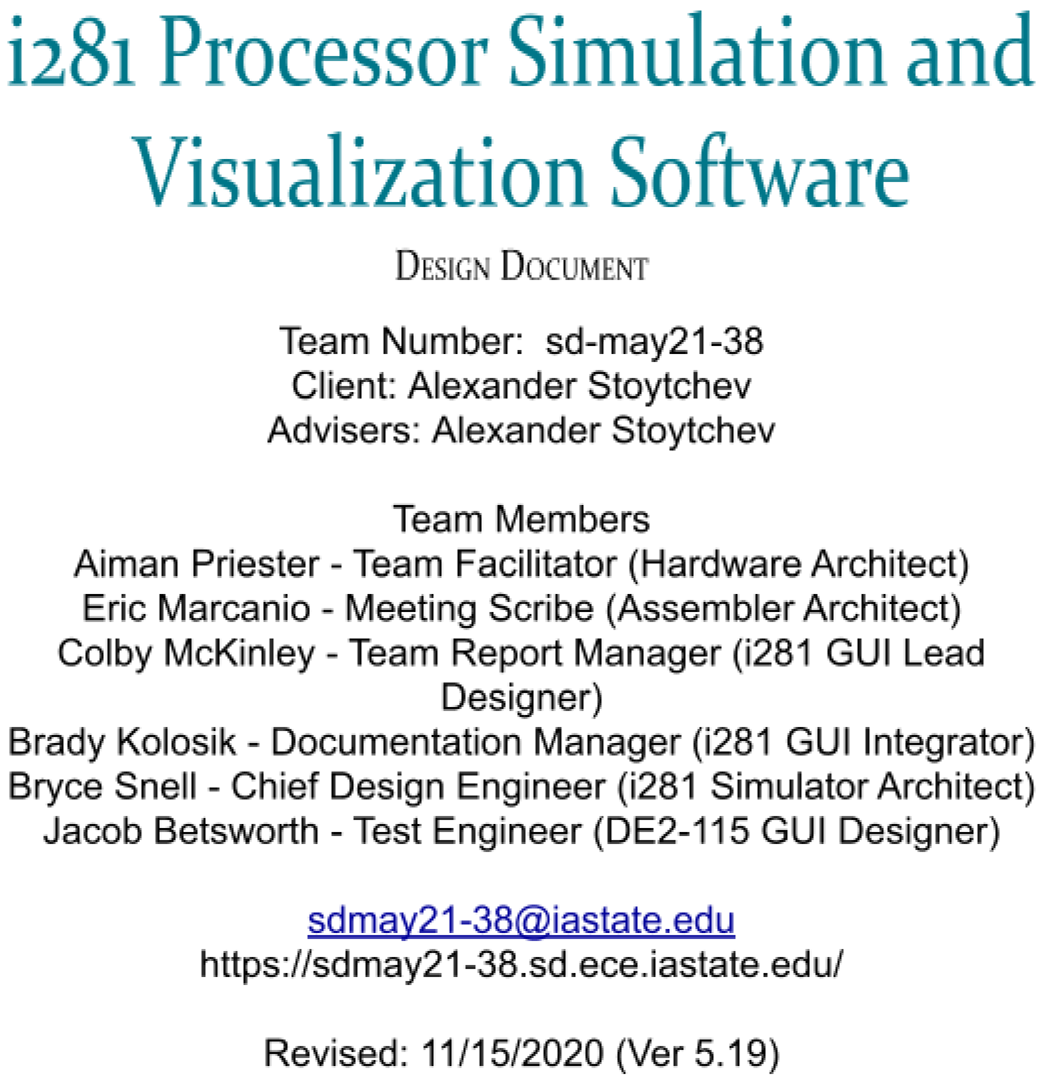
All images used in this presentation were taken from in house created lectures provided by Dr. Stoytchev, or screenshots taken as the project was in use on our own personal computers

ISO, International Organization for Standardization, 2015, www.iso.org/obp/ui/#iso:std:iso-iec-ieee:23026:ed-1:v1:en.

ISO, International Organization for Standardization, 2015, https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:15026:-1:ed-1:v1:en

"IEEE Standard for Software Unit Testing," in *ANSI/IEEE Std 1008-1987* , vol., no., pp.1-28, 30 Nov. 1986, doi: 10.1109/IEEESTD.1986.81001.

"IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers," in *IEEE Std 1284-2000* , vol., no., pp.i-100, 2000, doi: 10.1109/IEEESTD.2000.91947.



i281 Processor Simulation and Visualization Software

Design Document

Team Number: sd-may21-38
Client: Alexander Stoytchev
Advisers: Alexander Stoytchev

Team Members
Aiman Priester - Team Facilitator (Hardware Architect)
Eric Marcanio - Meeting Scribe (Assembler Architect)
Colby McKinley - Team Report Manager (i281 GUI Lead Designer)
Brady Kolosik - Documentation Manager (i281 GUI Integrator)
Bryce Snell - Chief Design Engineer (i281 Simulator Architect)
Jacob Betsworth - Test Engineer (DE2-115 GUI Designer)

sdmay21-38@iastate.edu
https://sdmay21-38.sd.ece.iastate.edu/

Revised: 11/15/2020 (Ver 5.19)

Executive Summary

Development Standards & Practices Used

ISO/IEC/IEEE 23026:2015

Systems and software engineering — Engineering and management of websites for systems, software, and services information

ISO/IEC/IEEE 15026:2019

International Standard - Systems and software engineering

IEEE 1008-1987

IEEE Standard for Software Unit Testing

IEEE 1284-2000

IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers

Summary of Requirements

- Create a lightweight Javascript Web client which simulates the i281 Processor
- Develop an assembler to output machine code to the CPU
- Create a Verilog model of the i281 processor

Applicable Courses from Iowa State University Curriculum

- CPR E 281: Digital Logic
  - Understanding of logic gates
- CPR E 381: Computer Organization and Assembly Level Programming
  - Understanding of processors and assembly language
- CPR E 581: Computer Systems Architecture
  - Understanding of processor simulators
- S E 319: Construction of User Interfaces
  - Understanding of front end technologies and implementation
  - Miniature Javascript project
- S E 329: Software Project Management
  - Understanding of creating software project proposals and the study of software development in industry
- COM S 309: Software Development Practices
  - Collaboration on a semester long software project
  - Understanding of front end technologies and implementation

- MATH 201: Introduction to Proofs
  - Aids understanding of the correctness of the program, beyond enumeration of all cases
- MATH 317: Theory of Linear Algebra
  - Understanding vectors and matrices, important to binary operations and the implementation of the register file and ALU.
- MATH 314: Graph Theory
  - Understanding pathways through a systems (enhances testing by creating meaningful and concise tests)

New Skills/Knowledge acquired that was not taught in courses

- Design and Implementation of practical graphical user interfaces and libraries
- Developing large pure Javascript projects
- Fundamentals of Compiling and Assemblers
-

Table of Contents

List of figures/tables/symbols/definitions

As of the current version of the document,

1 Introduction

1.1 Acknowledgement

We would like to state our appreciation towards Dr. Alexander Stoytchev for guiding us through this project. Dr. Stoytchev has committed himself to lending a helping hand and advising us along the way. We would also like to extend our appreciation towards the Electrical and Computer Engineering department for allowing us to perform hands-on work to further our knowledge in this subject area. We are certain that with the knowledge gained from this project, we are able to advance our careers and become innovators in our respective fields.

1.2 Problem and Project Statement

This project is being created to help students learn how a CPU works in CPRE 281, Iowa State's Digital Logic course. The course builds from boolean algebra into designing complex circuits. Since the course is often a student's first course in the Computer Engineering curriculum, it is challenging to understand at first. At the end of the semester the course introduces assembly code and the fundamentals of a processor, which are very abstract concepts to understand. This is why we are creating a simulation to explain the concept in an in depth and responsive manner.

The project is a web app CPU simulation. The user will be able to input assembly language and have it compile on the mock CPU. Each component of the CPU will have functionality and show the user the information being stored inside. Each line of assembly will propagate through the CPU with colored and detailed lines.

Through this project we want to accomplish a working understandable CPU. The main goal is to have the CPU be easy to understand and also simple enough that any user can learn from the app as they work through it.

1.3 Operational Environment

Our project will be used in the classroom setting as a teaching tool. Thus there are some considerations that must be taken into account.

- Ease of use. Our client wants this to be simple for him and students to use.
- Clarity. This should clearly show how the processor functions.
- Responsiveness. A slow application does not benefit our client or his students.

1.4 Requirements

Our project needs the following elements:

- A dynamic model of the i281 processor.
- An assembler using the i281 ISA.
- A Verilog version of the i281 processor.

More specifically we are trying to accomplish the following:

- Run the application on a website.
- Show inner workings of processor components.
- Show the conversion from assembly to machine code.
- Align the graphical modules of the processor to Verilog.

1.5 Intended Users and Uses

This project will be used by Dr. Stoytchev as a teaching tool in his lectures in CPRE 281. It could also be used by students in that class to dive deeper into architecture.

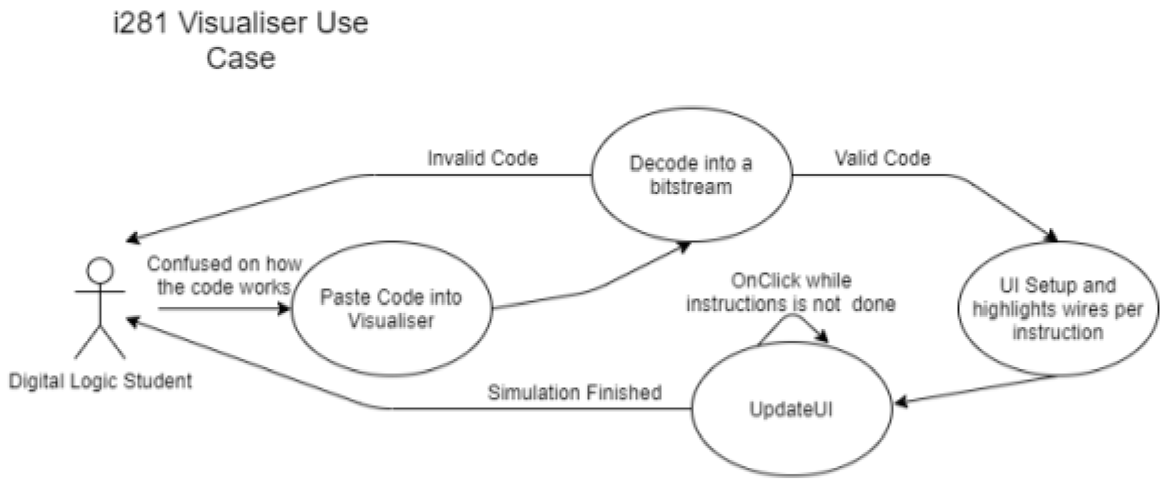

i281 Visualiser Use Case
Invalid Code
Decode into a bitstream
Valid Code
Confused on how the code works
Paste Code into Visualiser
OnClick while instructions is not done
UI Setup and highlights wires per instruction
Digital Logic Student
Simulation Finished
UpdateUI

Figure 1: Flowchart of CPRE 281 student using the visualizer

1.6 Assumptions and Limitations

Assumptions

- Can only be used if there is an internet connection
- Students have a preliminary understanding of the i281 CPU
- Signal lines will be colored to show the dataflow of the bits
- The visualiser should somewhat scale to the screen size of the user
- User will be able to either type in the program or upload a text file to run programs

Limitations

- End product will not include timing analysis for the CPU
- Cost to produce the project should be close to zero
- Visualiser must be lightweight to be able to run on a browser

1.7 Expected End Product and Deliverables

There shall be three deliverables by the end of the work period. Those are, the i281 Visualiser, Verilog conversion of the i281 and an Assembler for the i281.

HTML and Javascript code that when run on a website will show the different elements of the CPU architecture as well as the data that will run through them. There will also be elements for user input, such as switches, that will control input bits and allow for the user to input a text file of assembly code to simulate. The i281 Visualiser will be hosted by ISU on the clients' website. The Visualiser will be kept there for use in the Fall of 2021 in CPRE281. Students shall be able to access it at any time. Non-students, inclusive students that are not from ISU are also permitted to use the Visualiser to enhance their Digital Logic learning.

A Verilog version of the i281 CPU that will simulate the specifics of the i281 CPU design. It will be debugged using an FPGA to make sure it is as similar in functionality as possible to the CPU. The Verilog version of the i281 shall also be available on the clients' website. This is also to be used in the Fall of 2021. Students will not necessarily need this for the class, however, the client has expressed preference towards having it in case of a class overhaul.

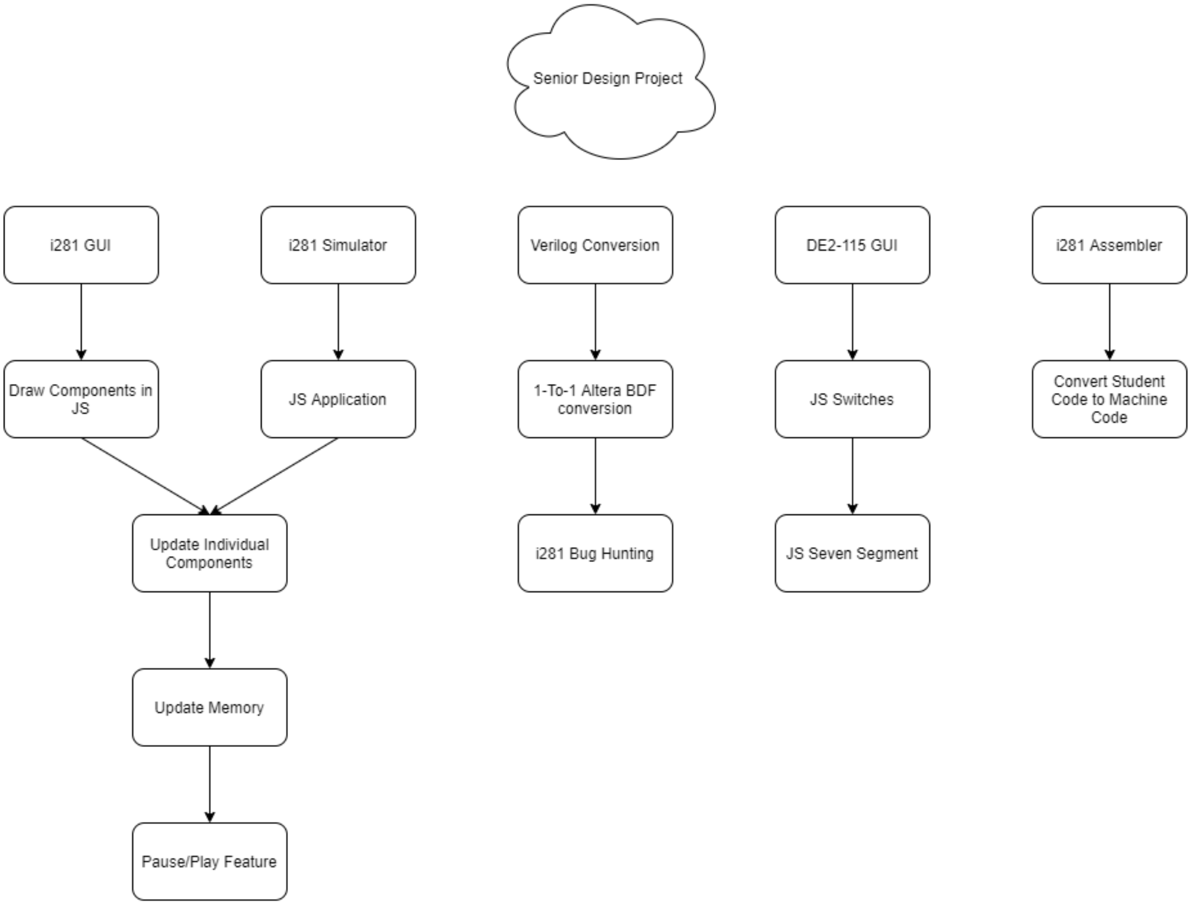
Lastly, the final deliverable is the Assembler. The Assembler will allow students to confirm their programs are exactly what they expect to write. This should allow for error checking and shall be available on the clients' website.

All three deliverables are expected to be ready by the end of the work term in May 2021.

2 Project Plan

2.1 Task Decomposition

- Assembler in Javascript to output machine code
- i281 Visualiser
  - Learn and compile information about drawing in Javascript
  - Learn to change individual components within the drawn CPU
  - Each wire lights up when data is being transferred through it
  - Memory is updated when it is being ran
  - Pause and play feature to understand what is happening
- Conversion to Verilog
  - 1 to 1 conversion from current version of the i281
  - Bug hunting and patching



2.2 Risks And Risk Management/Mitigation

Developing verilog version of i281 CPU. The FPGA board may overheat and subsequently start a fire. A comparable event is unlikely to happen (0.01). This task cannot be eliminated since it is deliverable in the final product. We can buy a fan for additional cooling support.

When developing the software for other parts there may be data loss in the project. This is a very high risk (.9) and we are going to use Gitlab to minimize the loss of data in the project.

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

Key milestones:

- Completion of Verilog i281 CPU
- JS assembler outputting files
- JS assembler with acceptable UI
- Visualizer UI completed
- Completed i281 CPU similar
- Ability to play pong

2.4 Project Timeline/Schedule

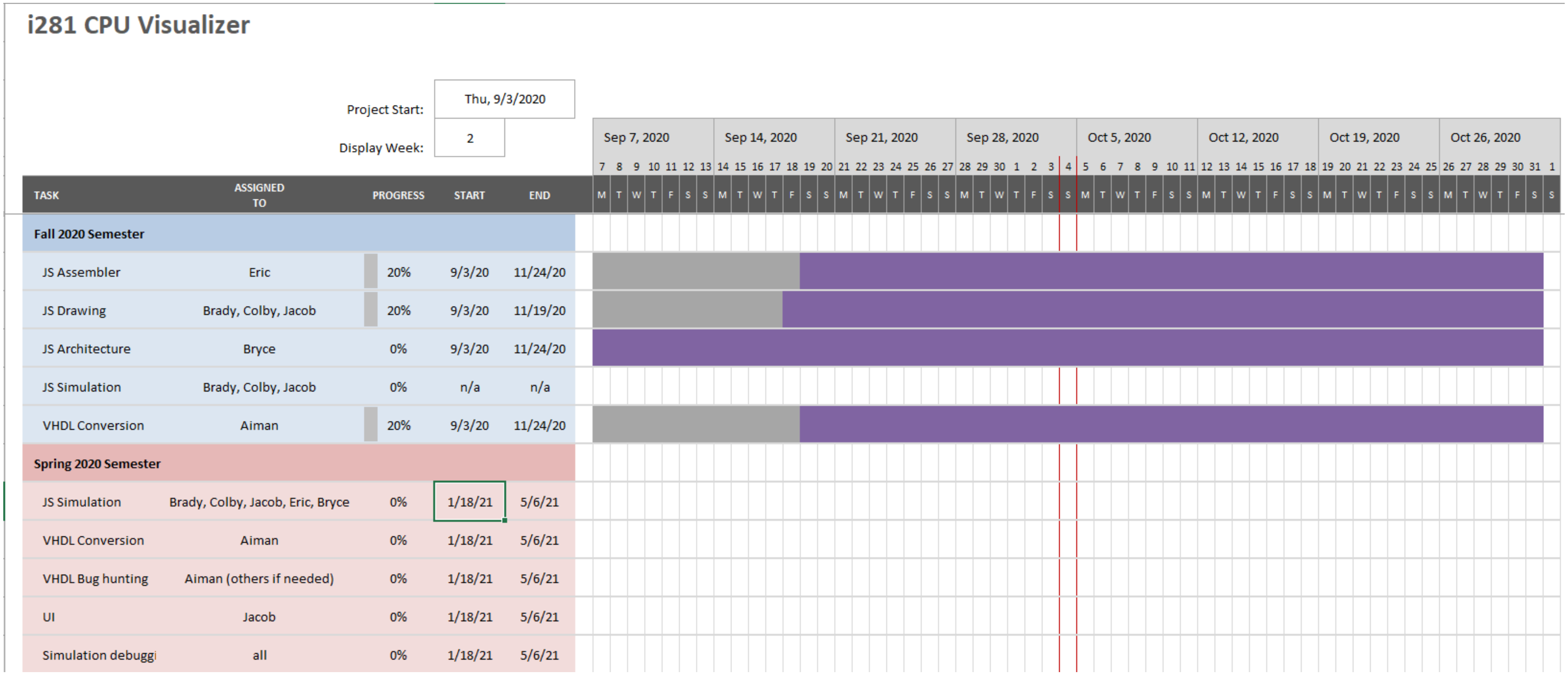
As it is still very early in the project, the provided gantt chart offers the goals that have been set for this semester and the next as well. As previously stated, our end goal is to have all three deliverables completed by the end of the spring semester in 2021, and that is reflected in the spring 2020 semester area of the chart below.

In order to successfully complete the task at hand, the group has divided up tasks to achieve parallelism in the development of the project, this semester will be spent further solidifying the division of work, and we have already split up the work as you can see in the chart below.

The fall semester will be spent completing the assembler such that a student will be able to translate from assembly to machine code, with the front end developers mainly exploring javascript and the various frameworks that can be used to draw, and animate the simulation that is required by the deliverable defined as “i281 visualizer”.

The spring semester will likely be a continuation of the fall semester, with some tasks having a lot of carry-over. At that point, the documentation will be out of the way leaving the team to fully focus on the development of the project. It’s expected that around half-way or so through the semester a semi-working prototype will be developed so that the phase of debugging and polishing can begin.

Note that this is also a rough schedule, and hard deadlines have the potential to be moved around.



i281 CPU Visualizer

Project Start:	Thu, 9/3/2020
Display Week:	2

TASK	ASSIGNED TO	PROGRESS	START	END
Fall 2020 Semester				
JS Assembler	Eric	20%	9/3/20	11/24/20
JS Drawing	Brady, Colby, Jacob	20%	9/3/20	11/19/20
JS Architecture	Bryce	0%	9/3/20	11/24/20
JS Simulation	Brady, Colby, Jacob	0%	n/a	n/a
VHDL Conversion	Aiman	20%	9/3/20	11/24/20
Spring 2020 Semester				
JS Simulation	Brady, Colby, Jacob, Eric, Bryce	0%	1/18/21	5/6/21
VHDL Conversion	Aiman	0%	1/18/21	5/6/21
VHDL Bug hunting	Aiman (others if needed)	0%	1/18/21	5/6/21
UI	Jacob	0%	1/18/21	5/6/21
Simulation debuggi	all	0%	1/18/21	5/6/21

2.5 Project Tracking Procedures

We are using multiple tools to keep our project moving forward. First, we are using Discord for inner team communications and discussions. We are keeping a list of weekly minutes for the discussion with Dr. Stoychev on our senior design website. Additionally we are using Gitlab as our code repository. Gitlab also allows us to create and organize tasks for our sprints.

2.6 Personnel Effort Requirements

Name	Task	Description	Reference
Eric Marcanio	Javascript assembler	Create an assembler that will take in i281 Assembly language and output machine language to the CPU simulator	https://simple.wikipedia.org/wiki/Assembler
Brady Kolosik	i281 Visualizer	Explore frameworks and methods to create a visualized version of the i281 CPU executing assembly code written for its architecture	
Colby	i281 Visualizer	Create a visualization of a simulated CPU while executing arbitrary assembly code.	https://developer.mozilla.org/en-US/docs/Web/JavaScript
Jacob Betsworth	User Interface of the Website	Creating the elements the user will interact with for input and to see output.	
Bryce Snell	i281 Visualizer	Develop a tool to simulate the i281 processor's output in a webpage	
Aiman Priester	Verilog Conversion	Converting each module into Verilog with the intention of squashing bugs in the i281 design	https://www.ece.iastate.edu/~alexs/classes/2020_Fall_281/

Table 2.6.1: Personnel Effort Requirements

2.7 Other Resource Requirements

Most of the project is a web application so we will not need any physical resources aside from a server to host on.

2.8 Financial Requirements

Funding for the FPGA Board was provided by the team member who required its use. ISU was not able to provide funding for this due to unspecified purchasing rules by ISU Purchasing Office.

3 Design

3.1 Previous Work And Literature

Initially the framework to this project was created by Dr. Stoytchev and a former student, Kyung-Tae Kim. The completed processor was implemented with basic logic gates from the ground up. This was to ensure that the Digital Logic students could understand the underlying design of a processor.

Since this processor was made in-house at ISU, this processor is unique. However, it similarly models the MIPS processor that is introduced in CprE 381. However, introducing the MIPS processor to CprE281 students will be overwhelming. Hence the i281 processor was born.

To be specific to this project, the team did find a project that has been done with a similar idea. This external project was a JQuery assembler like project [1]. We ended up using this to validate the achievability of creating a minimalistic Javascript project.

3.2 Design Thinking

The initial project objective was to create a CPU simulator for students to interact with. The motivation for the project is to help Digital Logic students grasp a high level understanding of computer organization and the motivation behind the class. Due to the many moving parts of a CPU, the normal power point slides would not be sufficient material. In hopes of having a long product lifespan, we chose to develop a Web application.

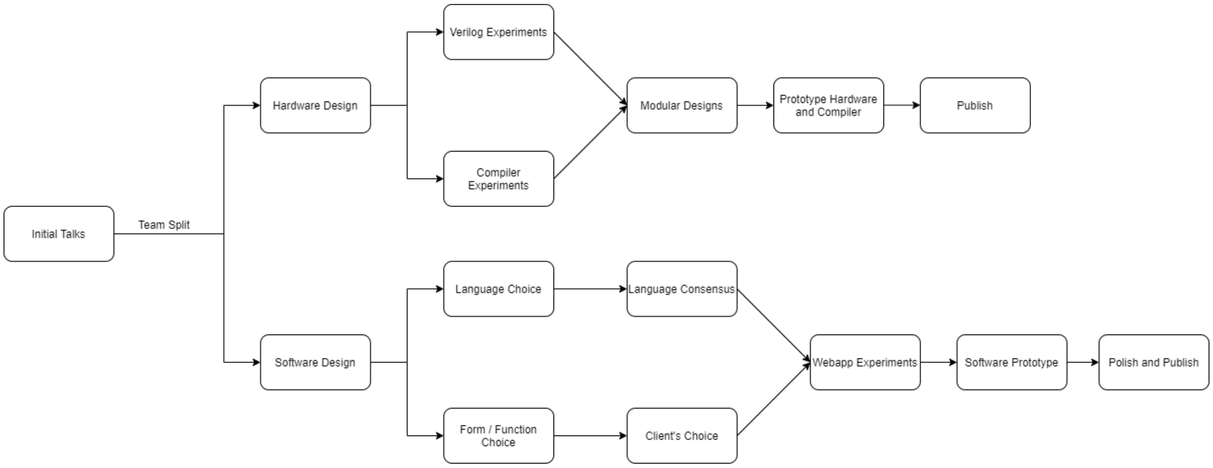


Table 3.2: Design Thinking Diagram

3.3 Proposed Design

There were several approaches to the solution we researched, using a particular framework (AngularJS or React), the use of leveraging languages together (Javascript and HTML, integrating Web Assembly C/C++), and several graphical design approaches. Based on parliamentary discussions with our advisor, we are creating a minimalistic Javascript application. There are no frameworks integrated, and will only be integrating HTML alongside Javascript.

So far we have created a NodeJS application which can properly send back HTML code with any tags. There are no third party libraries in use in the project; this may change depending on the graphics route. The project currently correctly simulates a functioning ALU and has basic graphics to go along with it. The project will eventually have each component being a separate class.

After several iterations of development, we have successfully created an SVG image which can be manipulated by Javascript interactions. We have created all major components in our display of the i281 processor and are able to dynamically interact with them

.

Additionally we have started trying to create code which simulates a CPU. The majority of the processor components have been made and a full simulator has been started. Since Javascript is a weakly typed language, we will be using arrays to do bit manipulations. There are ongoing discussions on how to handle Endianness and the differences between hardware representations and software representations.

Our design will be similar, but yet very different, from gem5 [2]. gem5 is a CPU simulator used in research. It allows designers to make processors then see performance. We will be using a similar modular design, but focusing on higher performance. A gem5 simulation will often take hours to run, but we need to be able to run our simple programs quickly.

The assembler has a simple interface. It can take a given use file, either text input or file upload, and completes the first of through iterations of completing. It is successfully compiling some assembly files.

The project will adhere to ECMAScript standards. It is meant to ensure the interoperability of Web pages across different Web browsers, and is common industry practice to use in NodeJS projects.

3.4 Technology Considerations

There are two main programming used in web development today, PHP and Javascript. Historically PHP was more common than in today's web market, however has been seen increasing popularity due to new changes. Javascript is currently more popular in web development, and has a vast amount of third party frameworks and libraries. Since PHP is more oriented towards server side development rather than Javascript, which is used more for client side development, we choose to go with Javascript.

We choose NodeJS since it uses non-blocking, event-driven I/O to remain lightweight and efficient real-time applications that run across distributed devices. NodeJS is based on the open web stack (HTML, CSS and JS), which perfectly aligns with our other design choices. It is difficult to compare NodeJS to another technology in its classification, however a comparable technology is ASP.Net. ASP.Net provides support for web applications, complex APIs and recently microservices. In our project most of these support features would go used.

3.5 Design Analysis

This design has a large amount of potential. We are using very common technologies for web applications that perform similar functions. Javascript seems well suited for the task of controlling dynamic web content. Our client also seems happy with this approach

Our current minimalistic approach to the project adds overhead into the implementation of the ideas. Since we have no framework to handle common actions, such as serving html pages, it is left to use to build the underlying infrastructure. There is little help offered online beyond library documentation and intuitive debugging. Building a runnable project has been time consuming and frustrating at points, however it hopefully increases our understanding of Javascript and will help us troubleshoot bugs later on in development.

The minimalist approach aligns with some current industry professionals opinions on the future of web development.

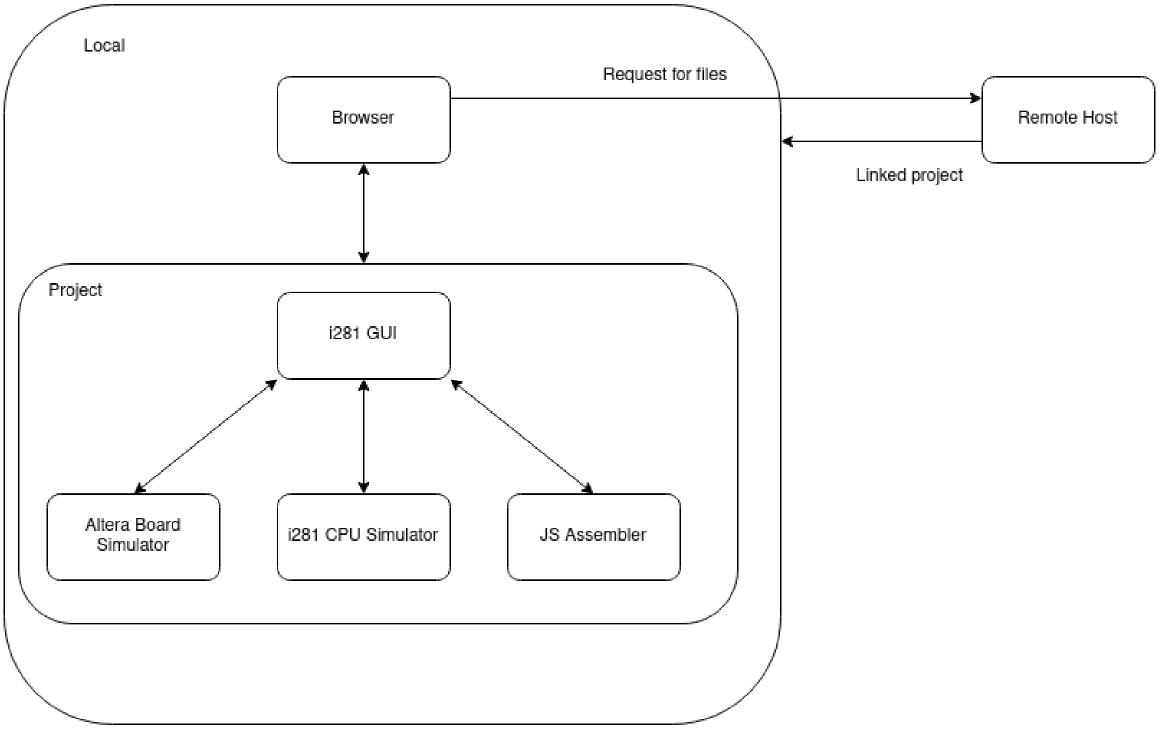
3.6 Development Process

We are developing this project with a modified version of Agile, called extreme programming. Extreme uses the core ideas from Agile but places extreme emphasis on client feedback.

3.7 Design Plan

The user will make a request to a remote host for the project via their browser. The browser will be the interface between the user and the project. Within the project the i281 GUI will interface with each of the modules.

Modules: i281 GUI, i281 CPU simulator, Altera board simulator, JS assembler



4 Testing

Verilog Conversion

The modules created on a per component basis must be rigorously tested. In the early stages, testbenches via ModelSim will be made to verify the designs of the components. However, later on in the development, hardware testing must be performed to ensure that the conversion is in fact accurate. This is a concern due to the variability of hardware performance.

i281 Visualiser

In large part testing a GUI is simplistic. We need only to check the elements we created on the document i.e. checking if the fetched id returns null or not. A further step is the check to see if the desired listeners on these objects are defined. In testing the wrapper classes we will have more traditional unit tests to verify the functionality. Currently in industry testing the GUI is often handwaved due to the scaling complexity in verification and ease of verification. That is trusting the object is created correctly if it is created.   This also falls into the testing philosophy of not testing code that is not yours.

i281 Simulator

The simulator testing is likely to be the most straightforward. We understand how the processor should work (through both the diagrams provided as well as the work done in the Verilog Conversion), we simply need to test that the simulated portions match the expected results. Pieces will be tested individually, then as larger functional units.

4.1 Unit Testing

We are using a common Javascript testing framework, Jest. We have created some simple test cases to demonstrate how a test file is created. The tests are unit tests for the CPU simulator.

4.2 Interface Testing

Interfaces will be tested with drivers that will be developed alongside the modules. Some of the needed drivers are:

- Memory driver to test memory and register units
- ALU driver
- Control driver
- OpCode driver

These drivers will send commands through the final interface between modules and check for proper responses from the output interface. This will allow us to test edge cases and confirm that the final control works as expected.

Small, incomplete versions of these unit tests have been started for testing the simulator logic. These will likely adapt and grow over time as we focus more on verification of our project.

4.3 Acceptance Testing

Our goal is to include our client after each cycle of iteration. This will give him a good view of progress and allow him to make changes as he sees fit. We plan to show that our tool meets his requirements by doing live demonstrations as well as reports of unit testing. This will show him the overall functionality as well as confirm the quality of the underlying code.

4.4 Results

Currently we have developed some small snippets of code to test languages and frameworks. Since this testing phase was intentionally open minded and less structured (to allow exploration and experimentation) we did not develop any formalized tests for the online tools.

One of our goals for the next phase of development is to develop a testing procedure and tests for it. Specifically, we will be implementing unit testing and hopefully a pipeline of known good tests to prevent project regression.

5 Implementation

- i281 GUI and Simulation
    - Linking the JS version of the i281 to the GUI
    - Linking the Assembler
- i281 Assembler
    - Allow users to write assembly in a predefined text entry region
- Verilog Conversion
    - Integrate Verilog to the DE2-115 FPGA and continue bug hunting

6 Closing Material

6.1 Conclusion

The team has successfully performed experiments and has learned a whole lot about the nuances of JavaScript, Verilog and the CPU architectures in general. The goal of this project is essentially to create a learning aid for future CprE 281 students. The best way forward is to create something that is accessible to all students regardless of technological limitations and across all OSes. Hence, we chose to create an online web app that will work with most if not all modern browsers. The lack of specific frameworks will also allow the longevity of this project to educate many semesters worth of CprE281 students.

6.2 References

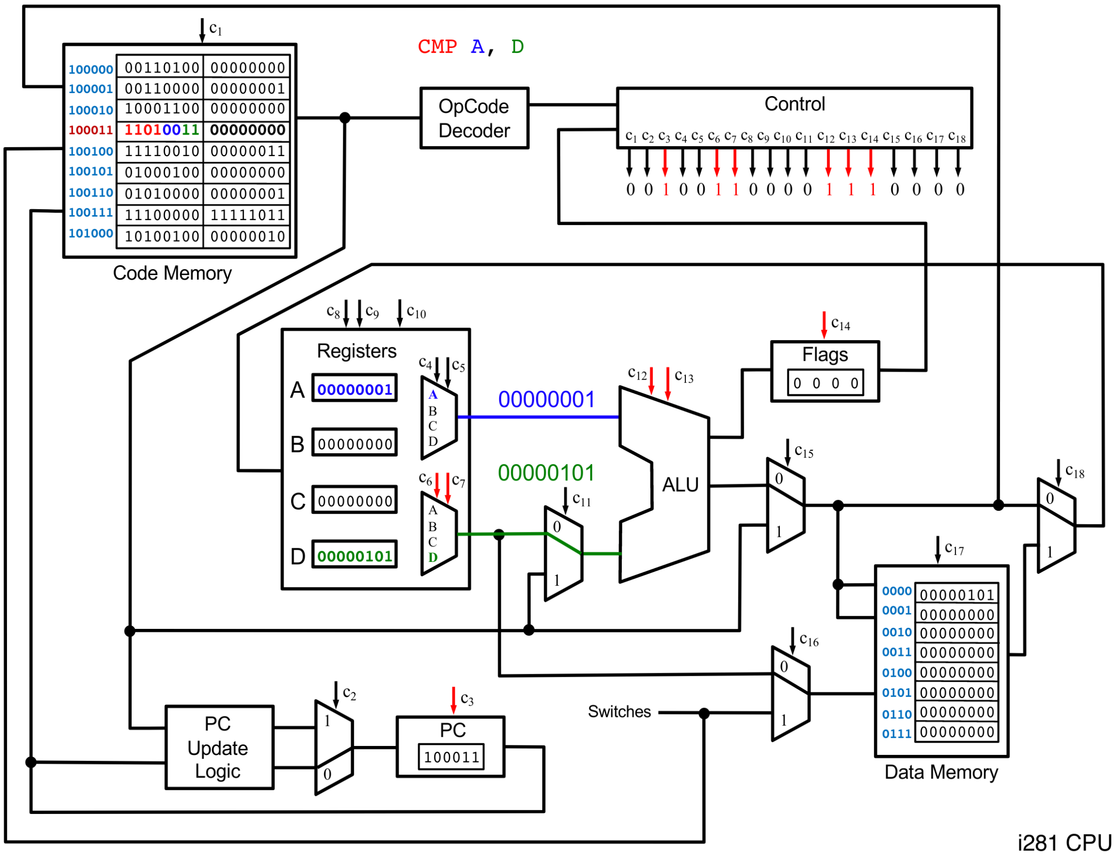
RodRego

[1] D. Dennett and B. Tanen, *RodRego*, 2014. [Online]. Available: http://proto.atech.tufts.edu/RodRego/. [Accessed: 11-Nov-2020].

Gem5

[2] gem5. (Version 20). [Online]. Available: http://www.gem5.org

6.3 Appendices



100000	00110100	00000000
100001	00110000	00000001
100010	10001100	00000000
100011	11010011	**00000000**
100100	11110010	00000011
100101	01000100	00000000
100110	01010000	00000001
100111	11100000	11111011
101000	10100100	00000010

0000	00000101
0001	00000000
0010	00000000
0011	00000000
0100	00000000
0101	00000000
0110	00000000
0111	00000000