

Mostafa Elfaggal - 900211128

Raef Hany - 900213194

Ali Elkhoully - 900212679

Description:

We chose javascript as our programming language, as it is similar to C++, so it is not so foreign to us. Moreover, because it'll be easier to implement a website-based GUI using this language. We started with the gui.html. The website based GUI. Then data.js was built. It was basically a file to store the current values of each register, as well as the memory (RAM) and Program Counter. We assumed that the program could read a byte or half a word from the memory, so we implemented functions that do so—functions that write and read into 1 byte and half a word. We have created basic Program Counter functions, such as setInitial (as was tasked), increment and decrement. Finally to this, we have a file dataConversion.js, which is responsible for converting values from a number system to another.

Next, we have the parser.js. This file is responsible for dividing the input from the user into proper data types to be able to use them as instructions. It first validates through the whole input, both the data insertion and assembly language insertion. It then breaks down the assembly code and data input into proper different data structures for easier use in the flow.

After that, we have the flow.js, which is the file responsible for executing the instructions in the assembly language. It runs through the divided data structures from the parser and executes them. It runs the code depending if we want them run all at once or a command by command.

Assumptions:

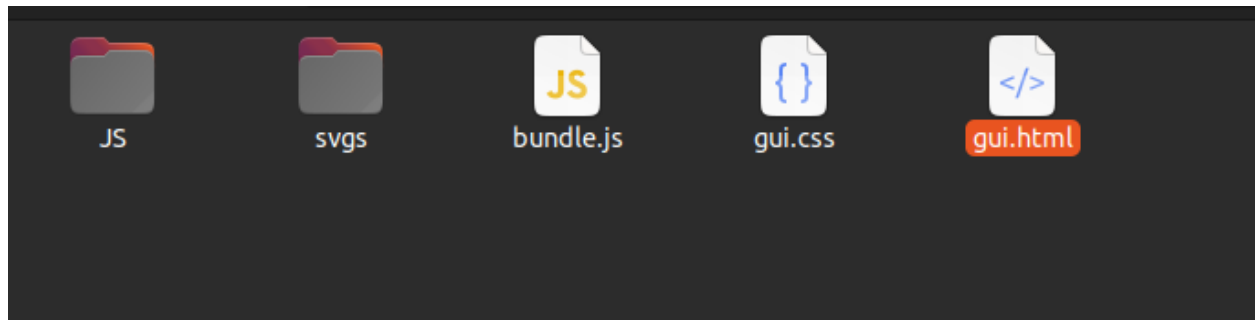
- Commands in input could be caps or lowercase.
- Offset could be inserted using brackets or as an input like registers. [sw t0,t1,0 **OR** sw t0,0(t1)]
- Users cannot add comments.
- Sp is initialized every run to -1 (the last address in the memory, since the stack grows upward)

Known Bugs:

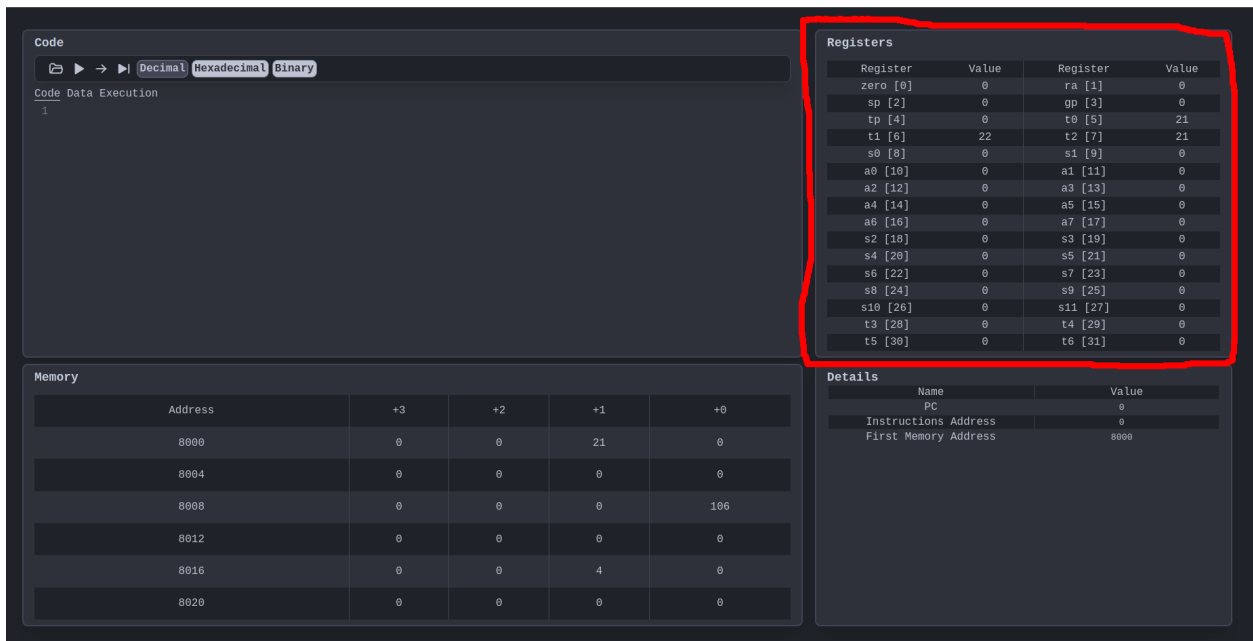
- We didn't put too much effort in validation, and so some cases (of unacceptable assembly instructions) may give errors

User Guide

1. Open the “Code” Folder, and choose “gui.html”



2. Since the program is built with html, there is no need to compile. Opening gui.html, should automatically open in a browser
3. This is where the registers are shown:



The 'Registers' panel shows the following data:

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	0	gp [3]	0
tp [4]	0	t0 [5]	21
t1 [6]	22	t2 [7]	21
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

The 'Memory' panel shows the following data:

Address	+3	+2	+1	+0
8000	0	0	21	0
8004	0	0	0	0
8008	0	0	0	106
8012	0	0	0	0
8016	0	0	4	0
8020	0	0	0	0

The 'Details' panel shows the following data:

Name	Value
PC	0
Instructions Address	0
First Memory Address	8000

4. This is where the memory is shown:

The screenshot shows a debugger interface with four main panels: Code, Registers, Memory, and Details. The Memory panel is highlighted with a red box. It displays a table of memory addresses and their corresponding values in decimal, hexadecimal, and binary formats. The registers panel shows the current state of the CPU registers. The details panel shows the current instruction and its address.

Address	+3	+2	+1	+0
0000	0	0	21	0
0004	0	0	0	0
0008	0	0	0	106
0012	0	0	0	0
0016	0	0	4	0
0020	0	0	0	0

5. This is where some details are. Here you could adjust the initial instruction memory address (however they aren't actually placed in memory). You can also change the memory addresses viewed by changing the first memory address. You cannot change the PC manually, it will be automatically changed when running any code

The screenshot shows the same debugger interface as before, but with the Details panel highlighted by a red box. This panel contains information about the current instruction, including the PC (Program Counter), the instruction address, and the first memory address. The memory panel is also visible, showing the same data as in the previous screenshot.

Name	Value
PC	0
Instructions Address	0
First Memory Address	0000

6. This is the code section. You can open a previously written file or type in code instead. You can also edit the opened content (doesn't save in the opened file).

Code

Decimal Hexadecimal Binary

Code Data Execution

1

Registers

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	0	gp [3]	0
tp [4]	0	t0 [5]	21
t1 [6]	22	t2 [7]	21
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

Memory

Address	+3	+2	+1	+0
8000	0	0	21	0
8004	0	0	0	0
8008	0	0	0	106
8012	0	0	0	0
8016	0	0	4	0
8020	0	0	0	0

Details

Name	Value
PC	0
Instructions Address	0
First Memory Address	8000

7. Type some assembly code in the empty region as follows

Code

Decimal Hexadecimal Binary

Code Data Execution

1 addi t0,zero,15
2 addi t1,zero,23

Registers

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	0	gp [3]	0
tp [4]	0	t0 [5]	21
t1 [6]	22	t2 [7]	21
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

Memory

Address	+3	+2	+1	+0
8000	0	0	21	0
8004	0	0	0	0
8008	0	0	0	106
8012	0	0	0	0
8016	0	0	4	0
8020	0	0	0	0

Details

Name	Value
PC	0
Instructions Address	0
First Memory Address	8000

8. You can also provide some data to be loaded in the memory through the Data tab. It is placed by putting it line by line, where each line has the “address : value“, where the value is that to be put in said byte

Code

▶

→

▶

Decimal

Hexadecimal

Binary

Code Data Execution

1 8002:4
2 7003:64

Registers

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	0	gp [3]	0
tp [4]	0	t0 [5]	21
t1 [6]	22	t2 [7]	21
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

Memory

Address	+3	+2	+1	+0
8000	0	0	21	0
8004	0	0	0	0
8008	0	0	0	106
8012	0	0	0	0
8016	0	0	4	0
8020	0	0	0	0

Details

Name	Value
PC	0
Instructions Address	0
First Memory Address	8000

9. Press the play button to start

Code

▶

→

▶

Decimal

Hexadecimal

Binary

Code Data Execution

1 addi t0,zero,15
2 addi t1,zero,23

Registers

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	-1	gp [3]	0
tp [4]	0	t0 [5]	0
t1 [6]	0	t2 [7]	0
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

Memory

Address	+3	+2	+1	+0
8000	0	4	0	0
8004	0	0	0	0
8008	0	0	0	0
8012	0	0	0	0
8016	0	0	0	0
8020	0	0	0	0

Details

Name	Value
PC	0
Instructions Address	0
First Memory Address	8000

10. While running the program, the highlighted instruction is the one to be executed next.

Code

Decimal

Hexadecimal

Binary

Code

Data

Execution

```

1 addi t0,zero,15
2 addi t1,zero,23

```

Registers

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	-1	gp [3]	0
tp [4]	0	t0 [5]	0
t1 [6]	0	t2 [7]	0
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

Memory

Address	+3	+2	+1	+0
8000	0	4	0	0
8004	0	0	0	0
8008	0	0	0	0
8012	0	0	0	0
8016	0	0	0	0
8020	0	0	0	0

Details

Name	Value
PC	0
Instructions Address	0
First Memory Address	8000

11. Press the go to next, to execute the next instruction. You can also use the fall through button to execute all the commands till the end of the program (be aware that if the program enters an infinite loop, the computer will crash where the page will not be responding). All registers, memory, and details are updated after every iteration of execution.

Code

Decimal

Hexadecimal

Binary

Code

Data

Execution

```

1 addi t0,zero,15
2 addi t1,zero,23

```

Registers

Register	Value	Register	Value
zero [0]	0	ra [1]	0
sp [2]	-1	gp [3]	0
tp [4]	0	t0 [5]	15
t1 [6]	0	t2 [7]	0
s0 [8]	0	s1 [9]	0
a0 [10]	0	a1 [11]	0
a2 [12]	0	a3 [13]	0
a4 [14]	0	a5 [15]	0
a6 [16]	0	a7 [17]	0
s2 [18]	0	s3 [19]	0
s4 [20]	0	s5 [21]	0
s6 [22]	0	s7 [23]	0
s8 [24]	0	s9 [25]	0
s10 [26]	0	s11 [27]	0
t3 [28]	0	t4 [29]	0
t5 [30]	0	t6 [31]	0

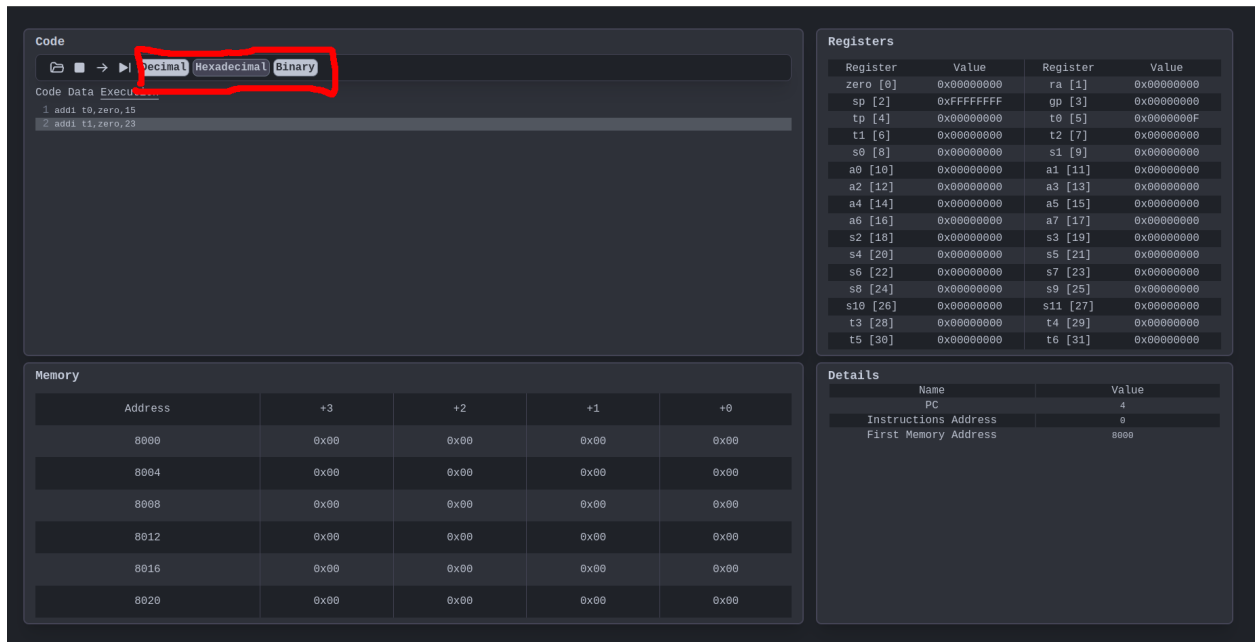
Memory

Address	+3	+2	+1	+0
8000	0	4	0	0
8004	0	0	0	0
8008	0	0	0	0
8012	0	0	0	0
8016	0	0	0	0
8020	0	0	0	0

Details

Name	Value
PC	4
Instructions Address	0
First Memory Address	8000

12. You can also change the viewed data format from decimal, hexadecimal, or binary using the options at the top



Test Cases:

1. randomTest.s
Runs all possible instructions to show that they operate. It doesn't actually do anything.
2. summ.s
Runs a function which operates recursively to sum all integers from the initial parameter a0, down to 0. So $5+4+3+2+1 = 15$. Returns the value in a0
3. mulLoop.s
Operates along with the mulLoop_data.txt file, which includes the data to be loaded in the memory. Multiplies two numbers from the memory by each other. The values in the memory are loaded from the data file.