All modifications should be done inside the main.js file

CONFIGURATIONS

```
// modify specs here
const config = {
    opcodeWidth: 5, // 32 is 2^5
    sup_neg_num: true, // support negative numbers (reserves the left most bit as sign and uses the 2's compliment system)
    default_mem: { // default memory
        size: 2 ** 8, // 2 ^ 8 = 256
        rom_size_per_address: 16,
        ram_size_per_address: 8
    }
};
```

`opcodeWidth`

How many bits does the opcode occupy

`sup_neg_num`

Set to `true` if negative numbers (2's compliment form) is needed else false

`default_mem`

Contains the default memory values, can always be changed on memory specs in the top right corner of the simulator

Beyond this point, all items are found inside the IIFE (Immediately Invoked Function Expression) after the creation of the config object.

REGISTERS

Creating/Modifying Registers

To create registers, you only need to create an instance of it and pass an object with the properties below. It is essential that the new instance is stored in a variable as this will become convenient in some cases when creating instructions.

- Name
  - *Name of the register*
- Address
  - *It's binary designation or representation*
- Size
  - *Number of bits it can hold*

Example:

```
const R = new Register({
    name: "R",
    address: "01",
    size: 8
})
```

NOTE: Registers with no binary representation must have an address of `NULL`

```
const Z = new Register({
    name: "Z",
    address: "NULL",
    size: 1
})
```

Change properties to modify a register

Removing Registers

Just delete the creation of instance of the register

IMPORTANT REGISTERS

Some registers are important and any modification or removal will cause the program to not work properly

```
// IMPORTANT REGISTERS. DO NOT DELETE
const PC = new Register({
    name: "PC",
    address: "NULL",
    size: 8
})

const AR = new Register({
    name: "AR",
    address: "NULL",
    size: 8
})
```

(Although you can change its size to config.default_mem.size)

INSTRUCTIONS

Creating/Modifying Instructions

When creating instructions, just create an instance of Instruction Class and it will automatically add on the simulator. There's no need to store the new created instance in a variable.

When creating a new instance of the Instruction Class, it requires 3 parameters:

- Instruction Name
- Instruction Logic (in a form of function)
- Instruction Machine Code Format (optional)

Example:

```
new Instruction("LDAC", function(){
    // set AC register value to value at memory address
    AC.val = MEM.getD(parseInt(this.op[0]));
}, `A-1[MA]`)
```

Instruction Name is straight forward, so let's proceed on logic

INSTRUCTION LOGIC

Just like in Arduino, however, much easier.

- Getting the operands
    - To access the operands, just use the this.op keyword, which is an array. The first index (0) contains the first operand, the second contains the second operand and so on…
    - NOTE: operands stored in the this.op are strings so if the inputted value is a number, be sure to use the parseInt function (if you intend it to be used as a number)

- Reading/Writing to a Register
    - If the register is already known (not specified in an operand) then just access the variable that it was stored on when creating it. This is why it is essential to store registers in a variable.

      Example:

      ```
      new Instruction("CLAC", function(){
          // AC = 0, Z = 1
          AC.valD = 0;
          Z.val = 1;
      })
      ```

    - If not then use the static get method in the Register class and pass the name of the register, in this case, its name is in the operand

      Example:

      ```
      new Instruction("PUTS", function () {
          Register.get(this.op[0]).valD = parseInt(this.op[1]);
      }, `A1-2[MS]`)
      ```

      Setting the value of a register specified in the first operand to whatever the value passed in the second operand

- Registers have 2 ways of accessing/writing its values. The valD and val properties
  - val
    val is the immediate value of the register, whatever you put/read in here is reflected to the simulator, and it is intended that if ever you set this property to a new value, you set it as binary in a form of a string

    Example:

```
new Instruction("COPY", function () {
    Register.get(this.op[0]).val = Register.get(this.op[1]).val;
}, `A12-`)
```

    Whatever the value of the register at second operand is copied to the register at first operand

  - valD
    valD is made to read/modify the value of a register as a decimal, in cases where you want to put a decimal value but since register uses binary system, you would want to use the valD property, or in cases where you want to read the value as decimal to perform operations since you can't do arithmetic on strings (because binary is represented here as strings)

    Example:

```
new Instruction("ADD", function () {
    const D = Register.get(this.op[0]),
        S = Register.get(this.op[1]);

    D.valD += S.valD;
}, `A12-`)
```

    Incrementing the value of register at operand 1 by the value of the register at operand 2

```
new Instruction("PUTS", function () {
    Register.get(this.op[0]).valD = parseInt(this.op[1]);
}, `A1-2[MS]`)
```

    Setting the value of the register at operand 1 to whatever immediate decimal value in the operand 2, however, if the operand 2 is intended to be binary, then parseInt is not needed and the val property should be used instead

- Reading/Writing to the Memory
  To access the Memory Object, the MEM keyword is used. Helper methods below:
  - Reading from memory
    Use the get method to get values in the memory. It requires an address parameter, which here is in binary form (that I do not suggest)

```
get: function(address){
    return this.data.get(address);
},
```

This is only the implementation. No examples because passing an address in binary form as an operand is unfriendly for mental health

In cases where the address is in decimal (type number), then use the getD method.

```
new Instruction("LDAC", function(){
    // set AC register value to value at memory address
    AC.val = MEM.getD(parseInt(this.op[0]));
}, `A-1[MA]`)
```

- Writing to memory
  Use the set method to set the values in the memory. It requires the value (in binary form this time) and the address, in this method, in binary form (that I do not suggest)

```
},
set: function(value, address){
    this.data.set(address, value);

    // configure dom
    const slotDOM = document.querySelector(`div.MEM_SLOT[data-address="${address}"]`);
    let dcVal = value;
    if(config.sup_neg_num && value[0] == "1" && value.length > 1)
        dcVal = -Utils.bin2Dec(Utils.comp(value));
    else
        dcVal = Utils.bin2Dec(value);
    slotDOM.children[1].innerHTML = `(${dcVal}) ${value}`;
},
```

This is the implementation. Do not mind the code after the configure dom comment as this is only used for updating the memory values in the GUI

In cases where the address is in decimal form, use the setD method instead

```
new Instruction("STAC", function(){
    // set value at memory address to ac value
    MEM.setD(AC.val, parseInt(this.op[0]));
}, `A-1[MA]`)
```

- Utility Functions

  Utility functions are found in the utils.js, these functions are used for easier processing of values when making complex instructions

Example

```
new Instruction("AND", function(){
    // AC = AC ^ R, then if AC == 0, Z = 1 else Z = 0
    AC.val = Utils.AND(AC.val, R.val);
    Z.val = (AC.valD == 0) ? 1 : 0;
})

new Instruction("OR", function(){
    // AC = AC V R, then if AC == 0, Z = 1 else Z = 0
    AC.val = Utils.OR(AC.val, R.val);
    Z.val = (AC.valD == 0) ? 1 : 0;
})

new Instruction("XOR", function(){
    // AC = AC xor R, then if AC == 0, Z = 1 else Z = 0
    AC.val = Utils.XOR(AC.val, R.val);
    Z.val = (AC.valD == 0) ? 1 : 0;
})

new Instruction("NOT", function(){
    // AC = AC`, then if AC == 0, Z = 1 else Z = 0
    AC.val = Utils.NOT(AC.val);
    Z.val = (AC.valD == 0) ? 1 : 0;
})
```

INSTRUCTION MACHINE CODE FORMAT

The third and an optional argument is the machine code format string. This format is used for proper arrangement of bits when “assembling” the assembly code into machine code

The default (if no value is passed) is

```
"A1-32";
```

The machine code format has 3 types of characters

- A

  The A character is replaced with the opcode of the instruction.

- Numbers (range from 1 – 3)
  The numbers correspond to the operands. 1 is the first operand, 2 is the second operand and so on. If ever your design has 9+ operands (2 digits+), please do not use the simulator
- “ _ “

  The excess character (-) is replaced if the number of bits used by the opcode and operands combined is less than the number of bits of a ROM memory. This is represented as “X” or don’t care in previous lectures

```
new Instruction("COPY", function () {
    Register.get(this.op[0]).val = Register.get(this.op[1]).val;
}, `A12-`)
```

In here, the first and second operands are register and is arrange that they are beside each other after the opcode, the rest of the excess bits are “X” or don’t cares

In cases of constant number of bits used by an operand in the machine code, you can put a “[number of bits]” right after the operand number. In cases of registers, there is no need since creating register already provides its address

Example:

```
new Instruction("STAC", function(){
    // set value at memory address to ac value
    MEM.setD(AC.val, parseInt(this.op[0]));
}, `A-1[8]`)
```

In the example above, the first operand has “[8]” right after it. This means that this operand will always contain 8 bits no matter how big or small the number is.

For convenience, the keyword MS and MA can be used to indicate MemorySize and MemoryAddress

MemorySize means the amount of bits that a single memory of RAM can contain

MemoryAddress means the amount of bits of an address in RAM or ROM

```
new Instruction("LDAC", function(){
    // set AC register value to value at memory address
    AC.val = MEM.getD(parseInt(this.op[0]));
}, `A-1[MA]`)
```

```
new Instruction("JUMP", function(){
    // set pc value to the memory address provided
    PC.valD = parseInt(this.op[0]);
}, `A-1[MA]`)

new Instruction("JMPZ", function(){
    // if Z = 1 THEN GOTO memory address
    if(Z.valD == 1) PC.valD = parseInt(this.op[0]);
}, `A-1[MA]`)

new Instruction("JPNZ", function(){
    // if Z = 0 THEN GOTO memory address
    if(Z.valD == 0) PC.valD = parseInt(this.op[0]);
}, `A-1[MA]`)
```

In here, the first operands are expected to be the same number of bits with MemoryAddress as the operand are indeed a memory address in the RAM (LDAC) and ROM (Jump Operations)

```
new Instruction("PUTS", function () {
    Register.get(this.op[0]).valD = parseInt(this.op[1]);
}, `A1-2[MS]`)
```

In here, the 2nd operand is an immediate value, that is to say that it has the same number of bits of each memory slot in the RAM.

In cases where there are no operands, you can omit the argument

```
// create instruction sets here
new Instruction("NOP", function () {
    // do nothing
})
```

```
new Instruction("ADD", function(){
    // AC = AC + R, then if AC == 0, Z = 1 else Z = 0
    AC.valD += R.valD;
    Z.val = (AC.valD == 0) ? 1 : 0;
})

new Instruction("SUB", function(){
    // AC = AC - R, then if AC == 0, Z = 1 else Z = 0
    AC.valD -= R.valD;
    Z.val = (AC.valD == 0) ? 1 : 0;
})

new Instruction("INAC", function(){
    // AC = AC + 1, then if AC == 0, Z = 1 else Z = 0
    AC.valD++;
    Z.val = (AC.valD == 0) ? 1 : 0;
})

new Instruction("CLAC", function(){
    // AC = 0, Z = 1
    AC.valD = 0;
    Z.val = 1;
})
```

Machine code format is optional if the machine code feature has no use.

“COMPILER”

- The use of variables that signifies memory address in RAM can be used. Each variable are assigned its own address automatically

```
PUTS X 1
SAVE A X
PUTS X 7
SAVE B X
PUTS X 12
SAVE C X
```

Converted to:

```
0: PUTS X 1
1: SAVE 0 X
2: PUTS X 7
3: SAVE 1 X
4: PUTS X 12
5: SAVE 2 X
```

A ⟶ 0
B ⟶ 1
C ⟶ 2

- The use of markers to indicate parts of the program can be used, but they require the colon ":" character right after to access/create them

```
SQUARE:
INCR X
ADD R1 R2
TEST R2 0
FJMP SQUARE:
```

```
009
010
011
012
```

Converted to:

```
9: INCR X       1000100000000000
10: ADD R1 R2   0101101100000000
11: TEST R2 0   0111010000000000
12: FJMP 9      0011100000001001
```

DEBUGGING

When the simulator is not working properly look for these signs:

- A register is saved as a memory address with a comma in it

```
000 - 00000000 (X,)
```

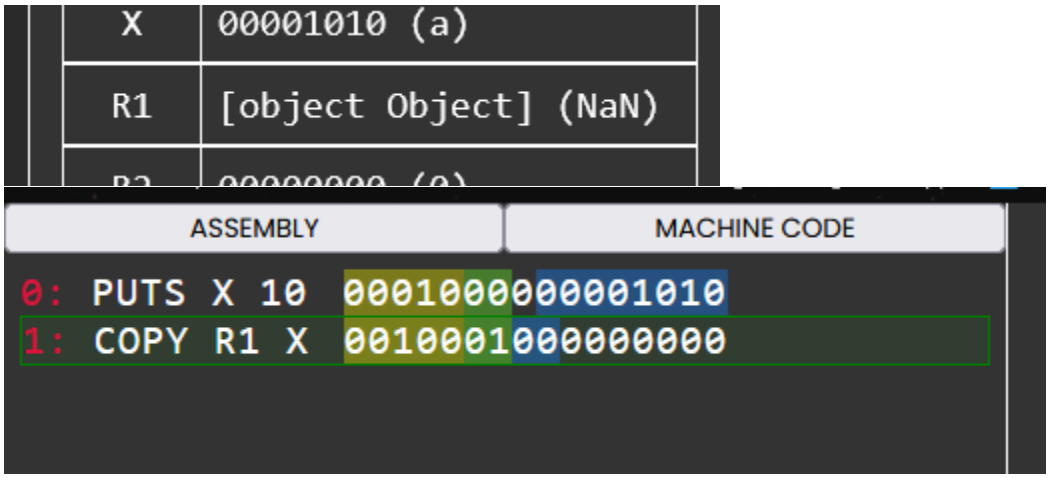
The simulator prohibits the use of comma to separate operands, use only 1 space

- A “[number]” appeared in the machine code

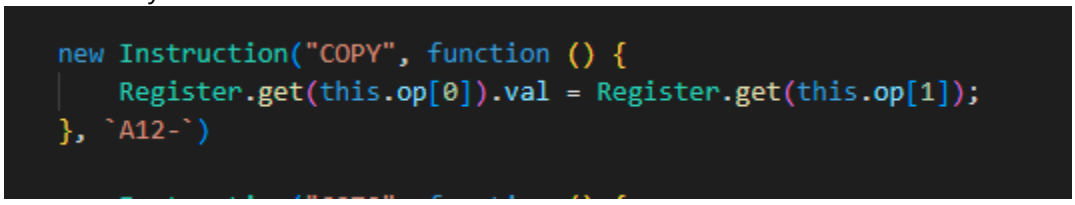
Caused by disordered operands, in the example above, first operand should be a register and the second operand should be an immediate value, not the other way around. If the symptom shows even if the order is correct, then please check the instruction logic in main.js

- A wild “[object Object]” has appeared

When assigning values in the instruction logic, make sure you are assigning it to a value, not the object



Is caused by



```
new Instruction("COPY", function () {
    Register.get(this.op[0]).val = Register.get(this.op[1]);
}, `A12-`)
```

Setting the value of register at 1st operand to the register object of the register at 2nd operand

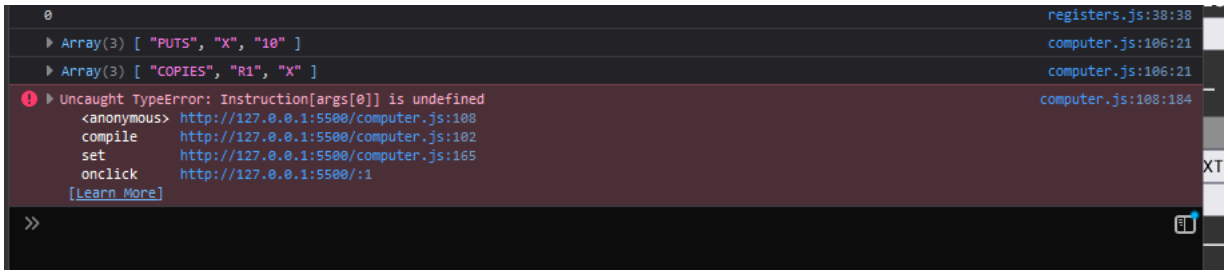
Fix:

```
new Instruction("COPY", function () {
    Register.get(this.op[0]).val = Register.get(this.op[1]).val;
}, `A12-`)
```

Added the val property to access the value of the register

- After all of that, code is still not working properly

  Check the console in the dev tools (Press CTRL+SHIFT+I for Mozilla and Chrome browsers) and check for this error:



  Instruction[args[0]] is undefined, this usually means that the logged information right before this error contains an instruction that is not created, thus it being undefined. As you can see above, the logged info right after the error contains an instruction with a name of “COPIES” with operands of R1 and X. There is no such thing as “COPIES” in my ISA, thus, the reason of the error.

- Still not working properly

  Email me jkdimpas@gmail.com or just message me in my facebook https://www.facebook.com/josh.dimpas.9/