Java_CentralPartsUnit Function Oriented Edition Documentation

Intro:

I'll keep this brief, but there is a little bit of important information that will be helpful when using the Java_CentralPartsUnit Function Oriented Edition library. This library represents bits as **char** type number being either **1** or **0**. Bits are often grouped into **char[]** arrays, with index 0 being your 1 bit, index 1 being your 2 bit, index 2 being your 4 bit, etc. Also if you couldn't tell by the name, this library is Function Oriented, so if you like OOP, check to see if I have released the object oriented version of this library on my GitHub. Feel free to make a pull request if you see any possible improvements or bugs and have fun!

Section 1: Logic

- It is important to note that a whole computer can be build with nothing but connections and NOT gates (*I double dog dare you to give it a try*). Knowing this, if you really wanna get hardcore about building your CPU from the ground up, only importing the Logic.java file into your code is a perfectly fine option.
- The Logic.java file contains 7 different logical operations: OR, NOR, XOR, XNOR, AND, NAND, NOT.
- All of the functions representing the logical operations are public, static, and output chars of either 1 or 0 representing a binary bit.
- Below are the truth tables for each of the fuctions

- OR(char BIT_A, char BIT_B)

BIT_A	BIT_B	OUT
0	0	0
1	0	1
0	1	1
1	1	1

- NOR(char BIT_A, char BIT_B)

BIT_A	BIT_B	OUT

0	0	1
1	0	0
0	1	0
1	1	0

- XOR(char BIT_A, char BIT_B)

BIT_A	BIT_B	OUT
0	0	0
1	0	1
0	1	1
1	1	0

- XNOR(char BIT_A, char BIT_B)

BIT_A	BIT_B	OUT
0	0	1
1	0	0
0	1	0
1	1	1

- AND(char BIT_A, char BIT_B)

BIT_A	BIT_B	OUT
0	0	0
1	0	0
0	1	0
1	1	1

- NAND(char BIT_A, char BIT_B)

BIT_A	BIT_B	OUT
0	0	1
1	0	1
0	1	1
1	1	0

- NOT(char BIT)

BIT	OUT

1	**0**
0	**1**

Section 2: Bitwise Logic

- The functions inside BitwiseLogic.java do logic functions on two variable-sized bytes (meaning a byte can be however many bits you want (like I said, total control)).
- If you do not understand how the bitwise system works, its simple: we have BYTE_A, BYTE_B, and OUT, and lets say our logic operation is OR. We take the first bit if BYTE_A and the first bit of BYTE_B and do the OR operation on the to get the first bit of OUT. Then we repeat for the rest of the bits in the bytes.
- You can have variable sized bytes, but they must have equal length or you'll receive an error and the function will return **null**.
- All of the functions are public static char[]s, and take the parameters (char[] BYTE_A, char[] BYTE_B) with the exception of NOT which only takes (char[] BYTE).
- All of the logic functions in BitwiseLogic.java are bitwise versions of Logic.java functions.

Section 3: Complex Logic

- ComplexLogic.java is exactly what it sounds like. It's simply functions to ease some of the pressure on your back as far as designing everything goes, and using them will probably save you some RAM as well. You wont understand these functions unless you understand how machines do binary addition and subtraction.
- There are 3 functions in the ComplexLogic.java file: ADD, HALF_ADD, and TWO_COMP. All of these functions are public static char[]s.
- ADD takes in three parameters: (char BIT_A, char BIT_B, char CARRY_IN). ADD puts out a char[] containing two chars inside: {OUT, CARRY_OUT}. The carry out is necessary to be able to link ADD functions, and the OUT is the output.
- HALF_ADD works the exact same as ADD, but only has two parameters: (char BIT_A, char BIT_B).
- TWO_COMP finds the two's compliment of any number by flipping its bits and then adding one. It takes one parameter: (char BYTE[]). The output is simply the two's compliment of that byte. This function might save you a lot of time on your CPU design.

Section 4: Register

- The register is the only object you can create with this version of CentralPartsUnit. The initializer of Register.java takes two parameters: (int size, String name). The size is the size in bits of the register, and the name is the name of the register.
- Input and output functions can be enabled or disabled, but are enabled by default
- All functions in the register are public, all but two are void. Here's the list:

1. erase() (makes the register all zeros)
2. setValue(char[] new_value) (sets the value of the register to whatever value was inputted)
3. setBit(int bit, char value) (sets the bit at index bit – 1 to value)
4. outputEnable() (enables output functions)
5. outputDisable() (disables output functions)
6. inputEnable() (enables input functions)
7. inputDisable() (disables input functions)
8. char[] getValue() (gets the value stored in the register)
9. char getBit(int bit) (gets the value of the bit stored at index bit - 1)