**DIGITAL LOGIC DESIGN | CSE3015  
TERM PROJECT REPORT**

**Havva Karaçam – 150315029**

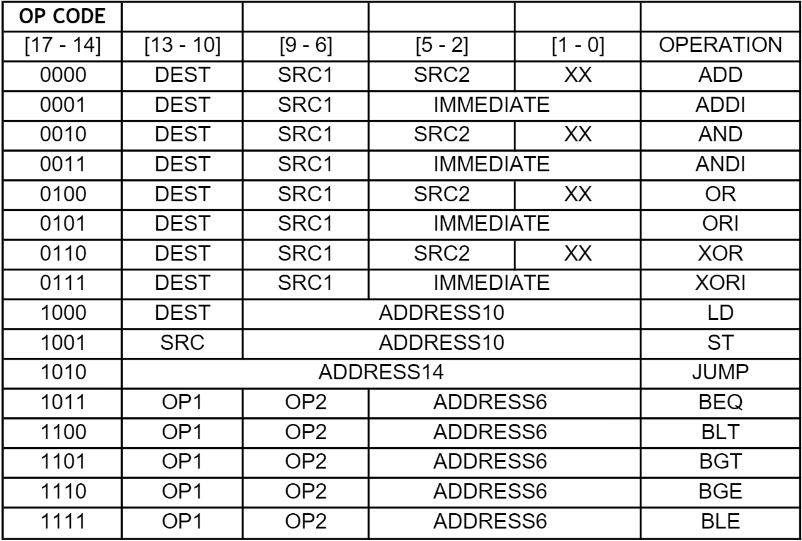
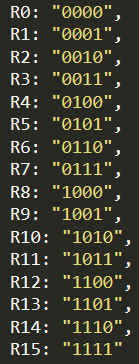
**Ömer Faruk Çakı – 150117821**

**Cem Güleç – 150117828**

# **Introduction**

To give a simplified version of aim of the project, we were expected to design and implement a processor which supports instruction set: (AND, ADD, OR, XOR, ANDI, ADDI, ORI, XORI, LD, ST, JUMP, BEQ, BGT, BLT, BGE, BLE).  
In order to do that we first needed to design an 18 bits of Instruction Set Architecture (ISA)- which declares our way of representing the operation code bits, register bits and remaining part of the instruction bits.

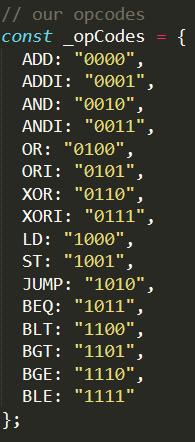
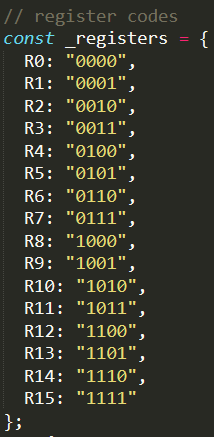
## **Instruction Set Architecture**

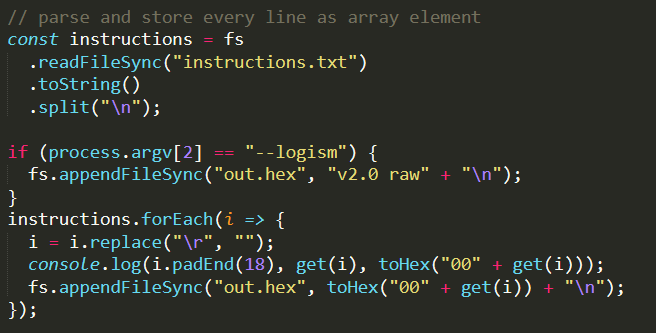
  
As in the above ISA, this is how we represented our instructions. In order to make Operation Code easier to read and make our job easier in Assembler (which is going to be told in the next part) we chose our Op-Code to be in most significant 4-bits. Then we put registers regarding of the operation requirements. For example, if the instruction’s Op-Code is ADD, then we will be needing 3 register space in the 18-bit instruction bits (each of the registers takes 4-bits of space). For the extra 2-bit space we decided to put into the least significant 2-bits [1-0] to avoid confliction.

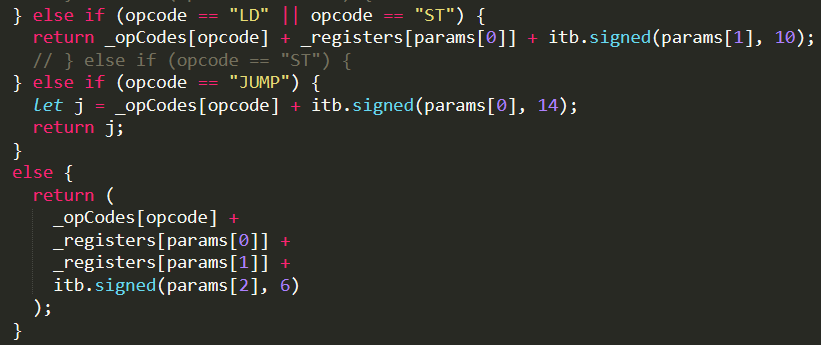
Apart from the arithmetic operations, we have Load (LD), Store (ST), Jump and Branch (BEQ, BLT, etc.) operations, which we need to consider.  
-> In case of Branch operations, we especially did not declare any n, z, p bits since we already know that their Op-Code’s are different. For this group of operations, we declared 2 register space right after the Op-Code bits and rest of the bits are reserved for the address bits.  
 **1.2. Registers**

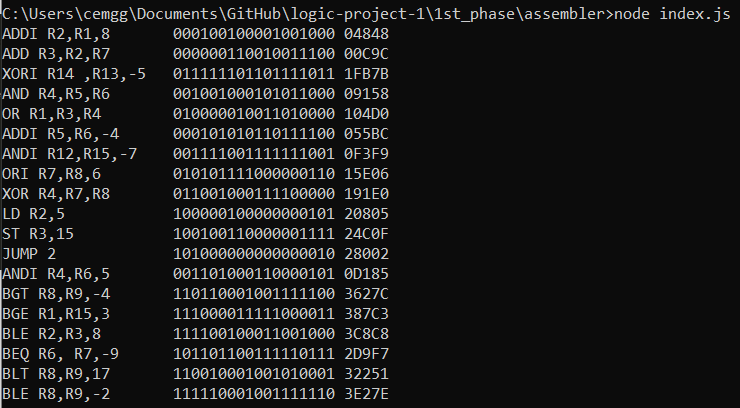
Next up, as it was specified, we were to be using 16 registers.   
We declared them as it is shown in the above image.   
Since there are 16 registers, we devoted 4 bits to represent them.

# **Assembler**

  
We picked JavaScript(node.js) for the purpose of coding our assembler structure. Assembler’s job is to be parsing the given instruction set bits as it was declared in the Instruction Set Architecture. Then, output every converted instructions as 5-bits in a “.hex” file.  
  
We included a 3rd part library “fs”, for the file operations. And another library which we created, named “int-to-binary”, which is responsible for parsing the instruction set and outputting 5-bits of output format.   
As in the right we declared our Operation Codes as well as registers:

  
Then, for the next part we simply read all the instructions from the .txt file. Then, making them ready to use after parsing and storing all of them as array element.

In our main parsing function, in order to return 18 bits binary number, we first split and declare opcode and parameters. Then, if it is an arithmetic operation as in the right image, we return corresponding ISA configuration values. For example: In case of ADDI R3, R2, 7 we first return the opcode bits, then registers (R3, R2), then 7 as the immediate value by calculating its value from our library.

  
  
  
  
  
  
  
  
  
  
Finally, after giving all kind of instructions and parsing them into 18 bits we gather the information we need to use on Instruction Memory as 5-bits hexadecimals.