DIGITAL LOGIC DESIGN | CSE3015  
TERM PROJECT REPORT

Havva Karaçam – 150315029

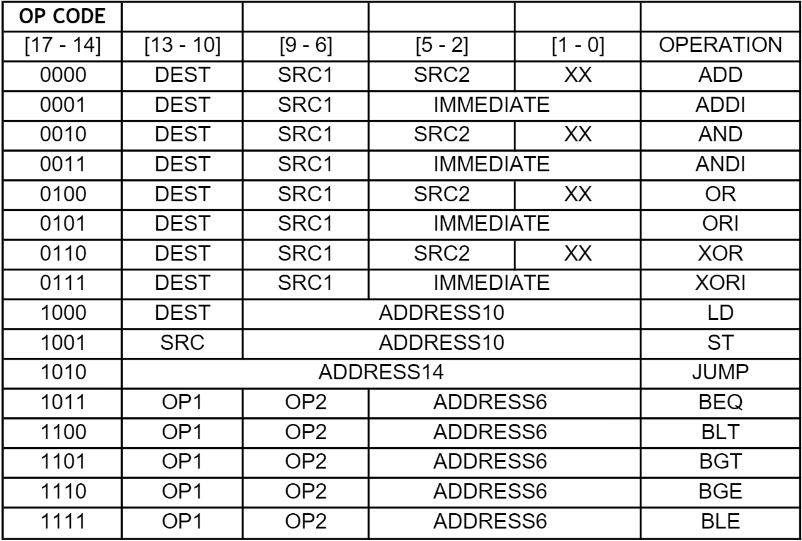
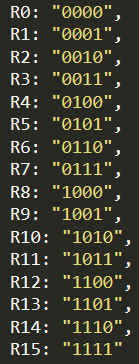
Ömer Faruk Çakı – 150117821

Cem Güleç – 150117828

# **Introduction**

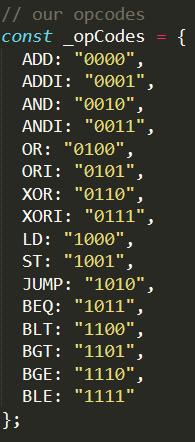
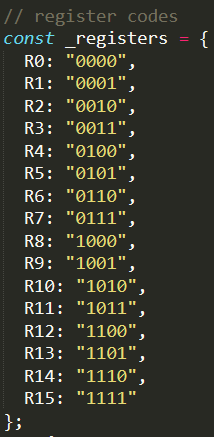
To give a simplified version of aim of the project, we were expected to design and implement a processor which supports instruction set: (AND, ADD, OR, XOR, ANDI, ADDI, ORI, XORI, LD, ST, JUMP, BEQ, BGT, BLT, BGE, BLE).  
In order to do that we first needed to design an 18 bits of Instruction Set Architecture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

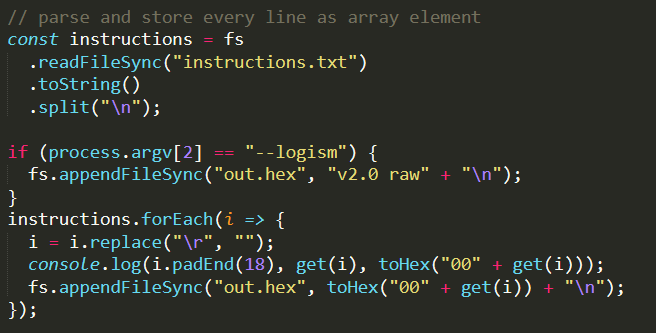
## **Instruction Set Architecture**

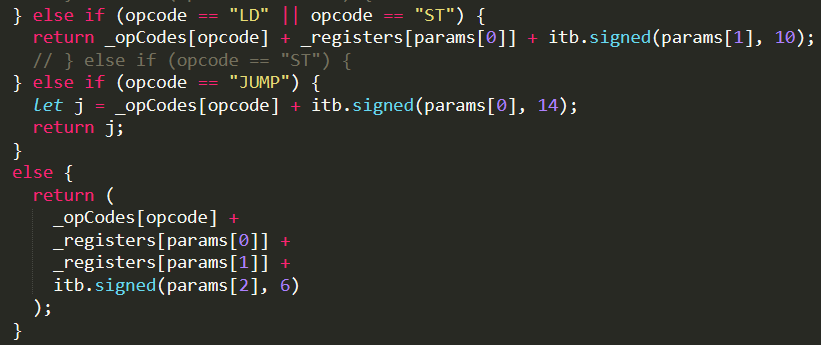
  
As in the above ISA, this is how we represented our instructions. In order to make Operation Code easier to read and make our job easier in Assembler (which is going to be told in the next part) we chose our Op-Code to be in most significant 4-bits. Then we put registers regarding of the operation requirements. 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.

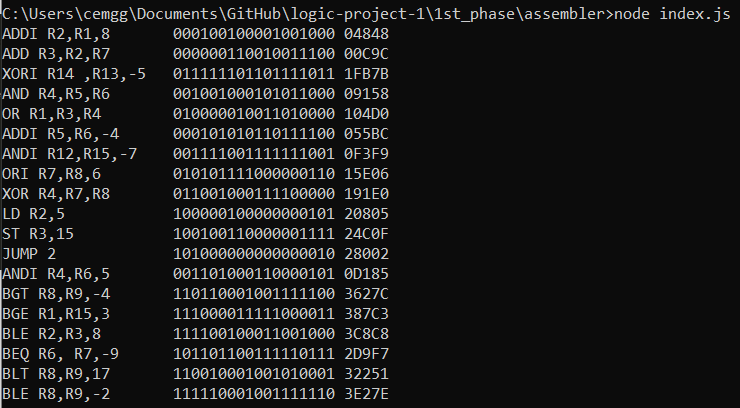
Apart from the arithmetic operations, we have Load (LD), Store (ST), Jump and Branch (BEQ, BLT, etc.) operations, which we need to consider.  
-> In case of Branch operations, we especially did not declare any n, z, p bits since we already know that their Op-Code’s are different. For this group of operations,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

# **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 instruction 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.

# **Main Circuit**

**Components Used:** Program Counter, Control Unit, Finite State Machine structures, RAM, ROM, Register File, ALU, Adders, Bit extenders, Comparators, Inverters, Registers, tunnels, AND gates, splitters, MUXes, DEMUXes and clock.

**Overview:** Instruction’s 18 bits gets split into parts and depending on the signal generated, particular ones gets put into register. Program counter is responsible for pointing to the instruction address on Instruction Memory (ROM). MUXes and DEMUXes used for the purpose of seperating behaviour of the components such as Register File’s both write and read functionalities.

# **a) Program Counter**

Our PC has 5 inputs. **Next** and **jump** signals determines whether PC is going to increment its value and process to next instruction by one or jump forward/backward by predetermined value. When next signal is 1, PC increments the current value by one with the next clock. We have an 18 bits **input** value, when **jump** signal is enabled (1), PC jump by that **input** value, input is a sign-extended value so if it’s a negative value PC jumps backward and forward for positive values. And we have a clock connected, when PC needs to update its value, clock signal also has to be 1, and we want Program Counter to jump next instruction when current instruction is completed; we have a **pc\_enable** signal connected to clock. Pc\_enable is only 1 at the end of each data path instruction and its set by FSM’s of instructions. They also set the pc\_enable=0 at the very beginning of the execution of the instruction. We prevent incrementing PC with every clock rising edge with this way. There is a counter inside PC. And lastly, we have a reset bit which is for development purposes.

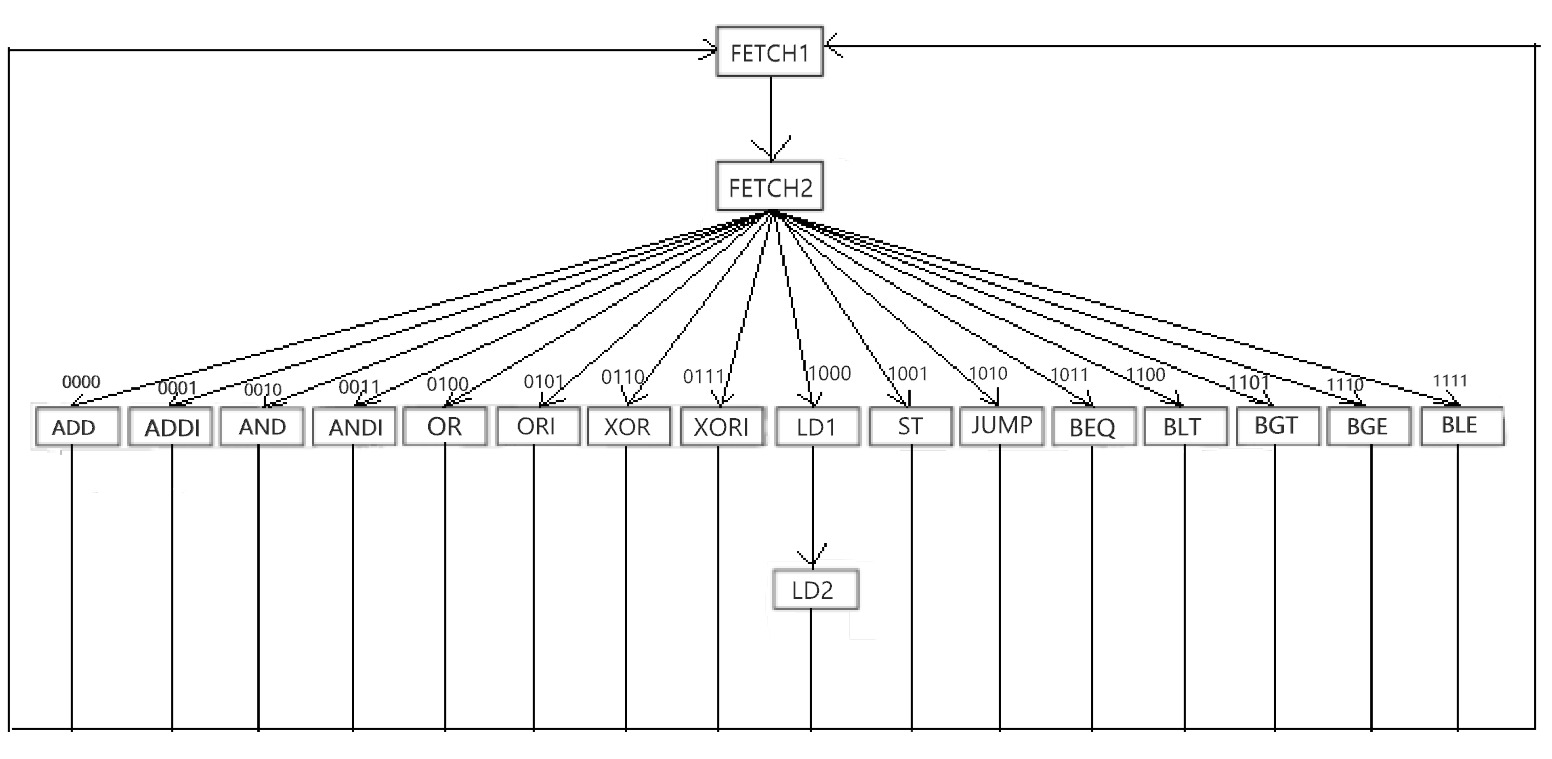
# **b) Control Unit**

Control Unit has 3 inputs; 18 bits input coming directly from Program Counter value, register\_1\_value and register\_2\_value which are values read from registers. CU splits our 18 bits input into bits and parses all values determined by our ISA by grouping appropriate ones. We have 4x16 Decoder inside CU, it decodes the opcode (leftmost 4 bits) and produces unique signals for each operation. After that we generate **is\_alu** signal if the operation is one of the ALU operation. We also generate 2 bits **ALU\_Operation** output for later use in ALU component. **Register\_write\_enabled** signal is generated if the operation is either LD or any of the ALU ops. Also, **memRead** signal for ST, **memWrite** for LD, **is\_jump** signal for JMP.

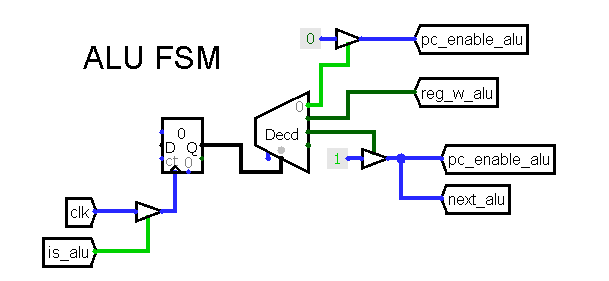
We connected all BXX operations with an or gate to produce an **is\_comparison** signal to determine that that instruction is a BXX instruction.

**18 Bits Comperator** inside the CU is responsible for generating **gt**, **lt**, **eq** signals. register\_1\_value and register\_2\_value is connected to this comparator. Based on the result, we check if the current BXX operation is correct and we produce a **comparison\_successor** signal on success.

# **c) Finite State Machine**



All of our identical instructions have their own execution structure since they are doing different operations. Let's take a look at how we do that for ALU operations



We have a structure like the image on the right for ALU operations. A counter (limited with 0-2) is increases at every clock and the value of the counter directly connected to decoder. These decoded values are actually representing our states. In the first state we disable the pc\_enable bit because we don’t want to program counter to increase its value while we are working on ALU operation. This works like a lock. By that time our 18 bits instruction is already parsed in Control Unit and most of the necessary signals are already produced. In the next state we set register enable bit to 1 and with the next clock signal, the produced value will be written into register. Lastly, in the last state we again set the pc\_enable=1 so that program counter can increase its value and continue with the next instruction.

Other FSMs are similar to this one but with different amount of states.

# **d) Data Memory**

We used RAM component as data memory. We have connected 10 bits address input, decides which address to write/read. Reg2\_value is representing the value read from register, and data\_out is the value of the register which pointed by address\_10. When mem\_write=1 it writes the register value into RAM; if mem\_read=1 it reads from the RAM and store its value into an appropriate register.  
  
  
  
  
  
**e) Register File**  
Register file includes 16 registers with inputs for read register 1, read register 2 and write register. Write data value is the input value of register. All registers’ data bits are 18-bit because of the input data size. RegWrite signal controls the register file. RegWrite signal will store an incoming result in the register indicated by write register. Multiplexers and decoders are used for the selection of registers.

**f) ALU**  
Alu is responsible for arithmetic operations. It includes two 18-bit inputs, 18-bit adder, and, or, xor and 18-bit result. Sum (with 18-bit adder), and, or and xor operations are determined thanks to mux which has select bit (2 bit) named as ALU operation. ALU operation select bits are determined in CU based on the operation code.

**8.1 Adder:** Adder used in ALU consists of nine 2-bit adder in order to sum 18-bit input. Also, 2-bit adder is created with full adder structures.