Steven Bumbera

Project 1

**Part 1:** The project was successful. All tested inputs returned correct outputs.

**Part 2:**

* **printFormatted(stringstream &ss)**
  + The purpose of this function is to format the output so that the hex results are formatted properly.
  + Take in stringstream
  + Convert to string
  + Set length of string to be 8 long (for 8 bits)
  + Convert the string to upper
  + Cout the string padded with “0x”
* **Vector<string> getNumbers(string s)**
  + The purpose of this function is to parse each line, take all numbers (including negatives), and store them into a vector for later use.
  + Take in a string (this is the line of the file)
  + Declare a vector
  + Create a substring of the string to include all up to the null terminator
  + Using the boost library, create a character separator to ignore (“ ,rR()”)
  + Tokenize the substring with the character separator
  + Iterate through the token storing the un-skipped characters into each index of the vector.
  + Return the vector for later use
* **Map<string, int>::iterator searchByValue(map<string,int> &maps, string val)**
  + The purpose of this function is to search a map for a given value and return the iterator so that the second integer value can be accessed at a later date.
  + Take in a map and a string
  + Create an iterator that starts at the beginning of the map
  + Create a while loop that iterates through the map until it reaches the end
  + If the first value of the iterator (the string in the map) is equal to the value passed in as val, then return that iterator for later use
  + If it is not found, return end of map (the value was not found)
* **Void encodeI(int op, int rs1, int rs2, int imm)**
  + Pass in the opcode, rs1, rs2, and imm
  + Create a binary mask that is 32 digits long
  + Left shift the op 26 places to make it the first 6 bits of the mask
  + Bitwise OR the opcode with the mask
  + Left shift rs1 21 places to make it the next 5 bits after op
  + Bitwise OR rs1 with the mask
  + Left shift rs2 16 places to make it the next 5 bits after rs1
  + Bitwise OR rs2 with the mask
  + Check if imm is negative(this is not a separate function but will be explained after the encodings)
  + If it is not, or the imm with the mask (no need to shift since it is the remaining 16 places)
  + Create a stringstream, convert the now 32 long binary number to hex, send that hex result into the stringstream
  + Send the stringstream to printFormatted()
* **Void encodeJ(int op, int imm)**
  + Pass in opcode and imm
  + Create a binary mask that is 32 digits long
  + Left shift the op 26 places to make it the first 6 bits of the mask
  + Bitwise OR the opcode with the mask
  + Check if imm is negative(this is not a separate function but will be explained after the encodings)
  + If it is not, or the imm with the mask (no need to shift since it is the remaining 16 places)
  + Create a stringstream, convert the now 32 long binary number to hex, send that hex result into the stringstream
  + Send the stringstream to printFormatted()
* **Void encodeR(int op, int rs1, int rs2, int rd, int func)**
  + Pass in opcode, rs1, rs2, rd, and func
  + Create a binary mask that is 32 digits long
  + Left shift the op 26 places to make it the first 6 bits of the mask
  + Bitwise OR the opcode with the mask
  + Left shift rs1 21 places to make it the next 5 bits after op
  + Bitwise OR rs1 with the mask
  + Left shift rs2 16 places to make it the next 5 bits after rs1
  + Bitwise OR rs2 with the mask
  + Left shift rd 11 places to make it the next 5 bits after rs2
  + Bitwise OR rd with the mask
  + Bitwise OR func with the mask (no need to shift since it is the remaining 11 places)
  + Create a stringstream, convert the now 32 long binary number to hex, send that hex result into the stringstream
  + Send the stringstream to printFormatted()
* **Not a separate function but this is the process of checking for imm to be negative**
  + This all happens in an if(imm<0), else bitwise OR the imm with the mask
  + Create a new int variable
  + Convert imm to a string
  + Create a bitset 32 in length and pass imm into it
  + Iterate through the section you want to adjust. (In encodeI() the imm is 16 long so the for loop runs from 16 through 32, in encodeJ() the imm is 26 long so the for loop runs 26 through 32, in encodeR() there is no negative check because it is always dealing with registers)
  + Flip the bits that were iterated through
  + Convert the bitset to a new string
  + Take the int that was created earlier and set it equal to stoi(bitsetString, nullptr, 2). This converts the newly flipped bitset into an integer, the nullptr and 2 says to make that integer a binary value.
  + Bitwise OR this result with the mask

Step 3:

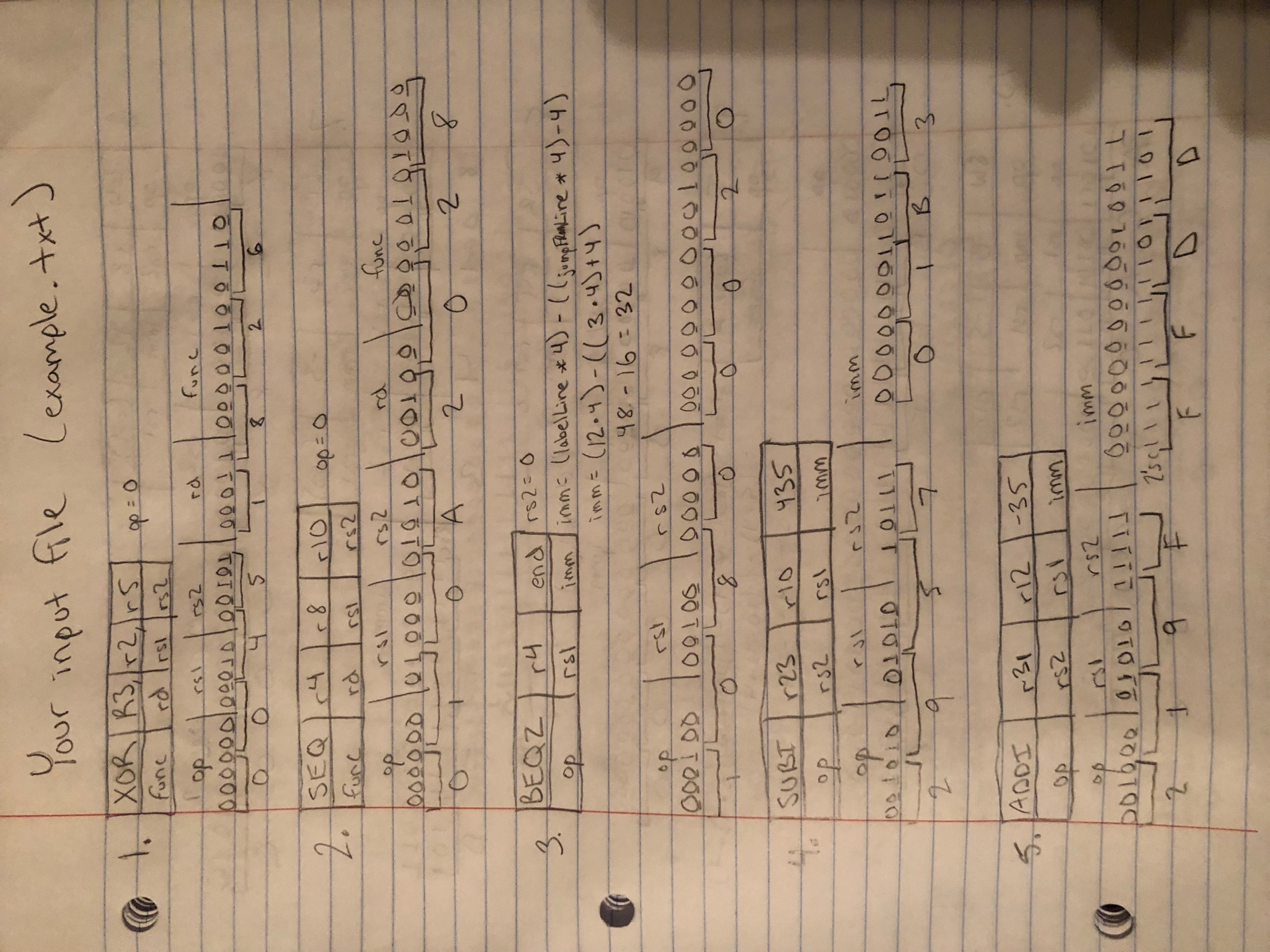
1. Created <string,int> maps and insert the name of the instruction with the OP code associated with it. Did this for jump, branch, ALU, load, store, and type R.
2. Created two empty <string, int> maps. One to hold <label, line label is on> and another to hold <Jump/Branch instructions, line they are called on>.
3. Create ifstream to read in the .txt file and declare all the variables that will be needed throughout main().
4. Check if file is open
5. **FIRST PASS**
   1. Loop through the file using getline until newline character
   2. Convert each line to upper case
   3. Get the first word of each line using substring
   4. Increment an int variable to count each line (increment first because the first line is 1, not 0. Could have just initialized the counter at 1)
   5. If the line contains a ‘:’, take the first word and insert it into the <label, line label is on> map like this <word before ‘:’, count>
   6. If the first word exists in the jump or branch map then insert it into the <Jump/Branch instructions, line they are called on> map like this <firstWord, count>
6. First pass is over. Before beginning the second pass call myfile.clear() to clear the flags from the previous iteration, and then call myfile.seekg(0, myfile.beg) to set the pointer back to the top of the file before beginning the second pass.
7. **SECOND PASS**
   1. Loop through the file using getline until newline character
   2. Convert each line to upper case
   3. Check if the line contains ‘:’, if it does cout a warning to say that this line is garbage. The reason for this is because the labels are on their own line and thus are processed like they are any other line, so a garbage answer is displayed. I’m sure there is a more clever way around this, but it doesn’t effect the functionality.
   4. In a series of if statements, search each map for the first word of that line, if it is found in a map then set a variable called type to the type of instruction. IE:
      1. If(mLoad.find(firstWord) != mLoad.end()){type = “load”;}
   5. The remainder is the logic behind each type found in section 7d, so a series of if statements check if(type == “whatever type was set to previously”). The logic is virtually the same for each type, the only difference is the values such as op, rs1, rs2, rd, func, imm.
   6. I will illustrate one of these methods here so that this readme isn’t ridiculously long. Let’s look at if(type == “R”) meaning the first word was found in the map that contains all of the R type instructions.
      1. NOTE: RType looks as such: op|rs1|rs2|rd|func
         1. Op always = 0
         2. Rd = the first encountered register
         3. Rs1 = the second encountered register
         4. Rs2 = the third encountered register
      2. First, create a <string, int> iterator and set it equal to searchByValue(mR, firstWord). Where mR is the map containing the R-type instructions.
      3. Set func = it->second. (Now func = the int associated with the first word in the line)
      4. A vector is created at the top of main, so before beginning this next step call vector.clear() so that you aren’t adding to values previously stored.
      5. Set the vector equal to getNumbers(line). This pulls the numbers out of the line, it stores the first encountered number into the 0th index of the vector, the second encountered number into the 1st index, and the third encountered number into the 2nd index.
      6. Set op = 0 (As mentioned before, R-Type has opcode at 0 by default)
      7. Stringstream the vector indices into the appropriate locations. Section f.i. tells us what needs to go where.
         1. RD = vector[0]
         2. Rs1 = vector[1]
         3. Rs2 = vector[2]
      8. Now that all the values have been captured they can be passed into encodeR(op, rs1, rs2, rd, func)
   7. The process remains the same for all other types. Jumps and branches need to be done a little differently because they receive their values from different maps and arithmetic needs to be done to ensure the correct locations of the lines. I will outline the differences using type = “branch” as an example.
      1. 3 iterators are created
         1. It = searchByValue(mBranch, firstWord). Op = it->second. So, the opcode is equal to the int value of the first word of the line.
         2. Create a temporary variable that will store the last word found in the line
         3. It2 = searchByValue(label, lastWord). labelLine = it2 ->second. So, labelLine is equal to the line number that the label actually exists on. (Or where to go, if the branch check returns 0)
         4. It3 = searchByValue(jLabel, firstWord). jLabelLine = it3->second. So, jLabelLine is equal to the line number you are currently on, ie: the label you are jumping from (or where the branch check is occurring)
      2. Set the imm = (labelLine\*4) – ((jLabelLine\*4)+4).
         1. That is the labelLine is being multiplied by 4 since each instruction is stored as an integer.
         2. jLabelLine is being multiplied by 4 for the same reason, 4 is then added to that result because the actual instruction is happening one line after the label.
         3. The difference is found between the two to illustrate the distance of the jump.
      3. Clear the vector
      4. Set the vector = getNumbers(line)
      5. Stringstream the first encountered number into rs1 (this is the case for branches)
      6. encodeI(op, rs1, 0, imm). Or you can set rs2 = 0 and call encodeI(op,rs1,rs2,imm).

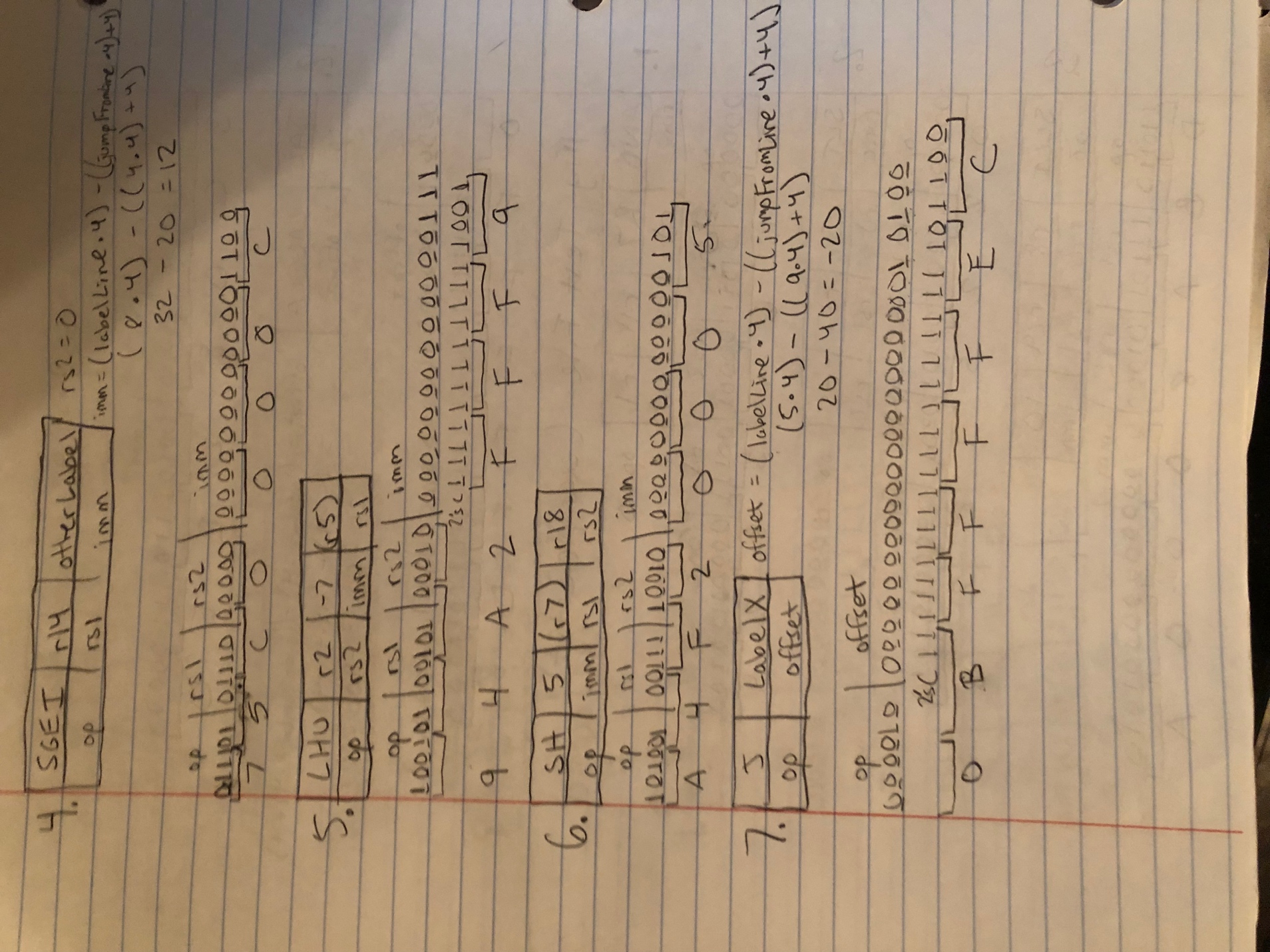
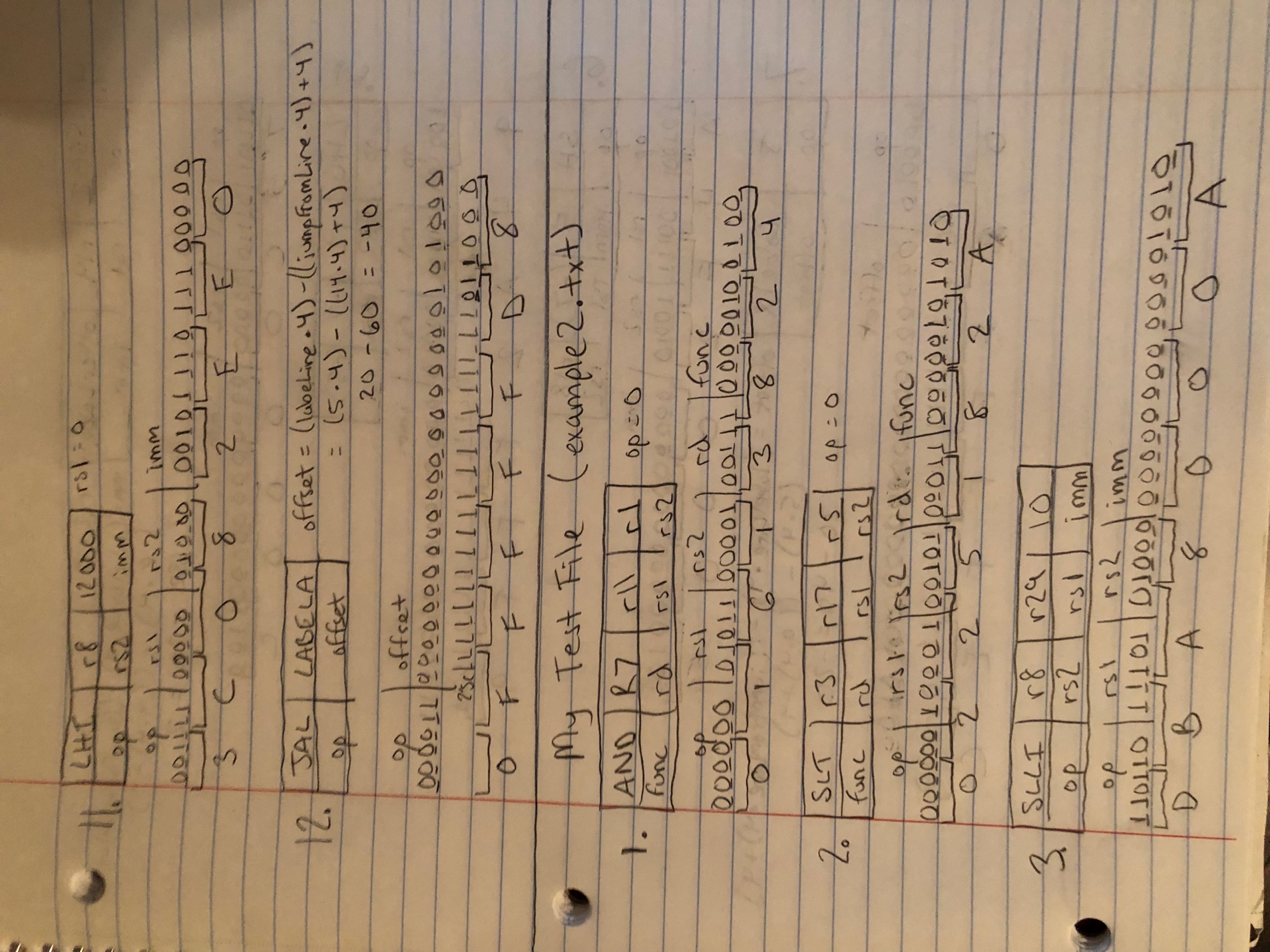
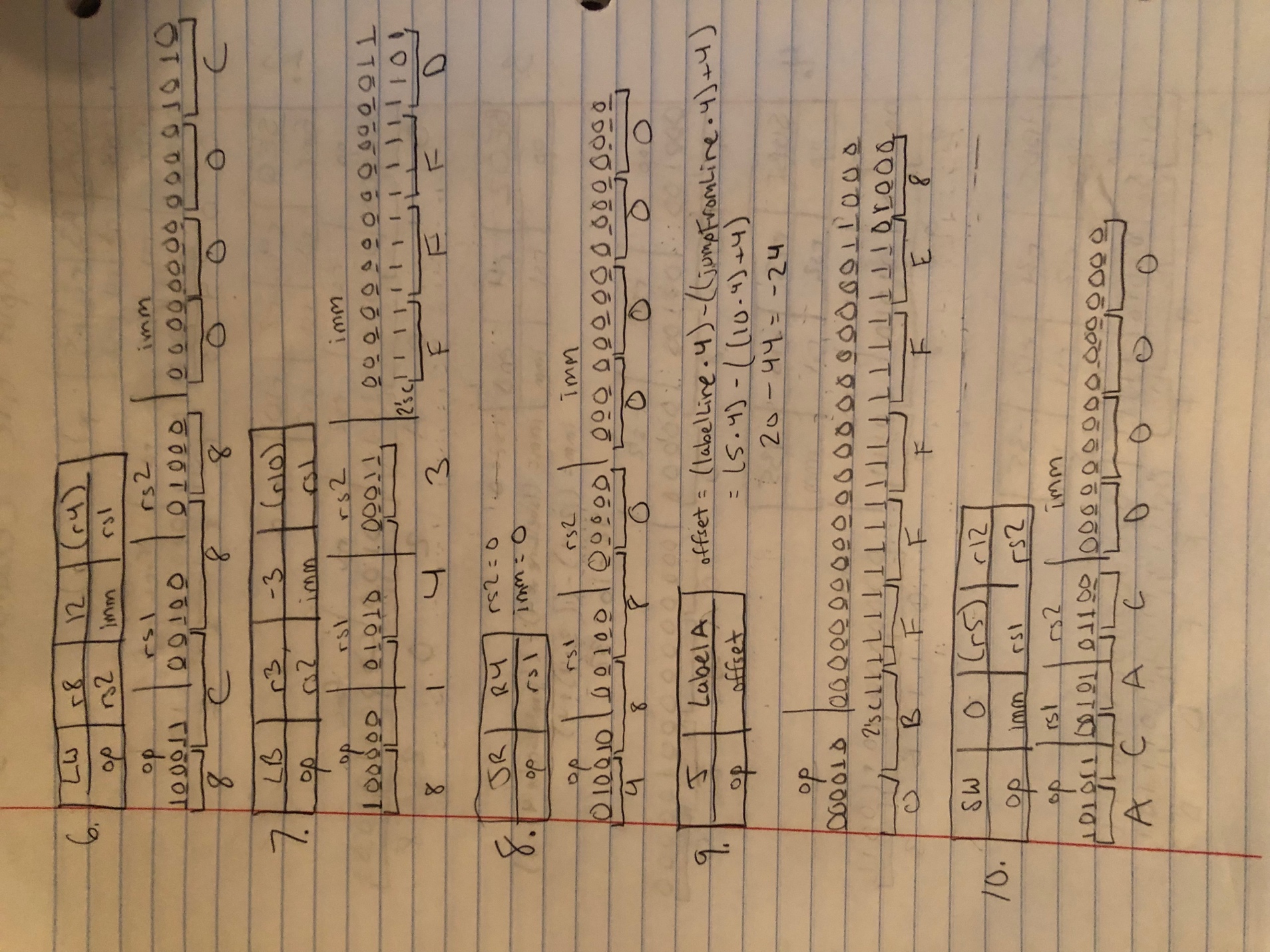
**PART 4:**

For these tables I removed the exact output for the GARBAGE lines simply so that it is more readable. To see exact output please see the .txt files

|  |  |
| --- | --- |
| INPUT (your file example.txt) | OUTPUT (EXAMPLEoutput.txt) |
| XOR R3, r2, r5  seq r4, r8, r10  beqz r4, end  subi r23, R10, 435  LabelA:  addi r31, R12, -35  LW r8, 12(r4)  Lb r3, -3(r10)  jr r4  J labelA  sw 0(r5), r12  end:  LHI r8, 12000  JAL labelA | XOR: 0x00451826  SEQ: 0x010A2028  BEQZ: 0x10800020  SUBI: 0x295701B3  GARBAGE  ADDI: 0x219FFFDD  LW: 0x8C88000C  LB: 0x8143FFFD  JR: 0x48800000  J: 0x0BFFFFE8  GARBAGE  SW: 0xACAC0000  LHI: 0x3C082EE0  JAL: 0x0FFFFFD8 |

|  |  |
| --- | --- |
| INPUT (my file example2.txt) | OUTPUT (EXAMPLE2output.txt |
| AND R7, r11, r1  slt r3, r17, r5  slli r8, r29, 10  sgei r14, otherlabel  labelX:  lhu r2, -7(r5)  sh 5(r7), r18  otherLabel:  J labelx | AND: 0x01613824  SLT: 0x0225182A  SLLI: 0xDBA8000A  SGEI: 0x75C0000C  GARBAGE  LHU: 0x94A2FFF9  SH: 0xA4F20005  GARBAGE  J: 0x0BFFFFEC |

****

****