You may work in teams of 1 to 4 on this assignment. EXACTLY ONE MEMBER OF YOUR TEAM SHOULD SUBMIT THE ASSIGNMENT IN CANVAS! THE TAs WILL ASSESS A PENALTY IF MORE THAN ONE MEMBER OF YOUR GROUP SUBMITS THE ASSIGNMENT, AS THIS CREATES A SIGNIFICANT OVERHEAD FOR THEM WHILE GRADING. Your submission should include a file, "authors.txt" giving the names and NetIDs of each member of your team.

For this assignment, you will be implementing a disassembler for the binaries that run on our LEGv8 emulator in binary mode. Your disassembler will handle input files containing any number of contiguous, binary LEGv8 instructions encoded in big-endian byte order. The input file name will be gi rinted to the terminal, should be- LEGv8 assembly code that generated the binary.

Except that it ignores the PC and flow control, a disassembler essentially implements the first two stages (fetch and decode) of the five stage pipeline described in lecture and the textbook. A working disassembler requires perhaps half of the total work of building a binary emulator.

Your disassembl uctions:

- ADD
- ADDI
- AND
- ANDI
- B
- B.cond: This is a CB instruction in which the Rt field is not a register, but a code that indicates the condition extension. These have the values (base 16):
    - 0: EQ
    - 1: NE
    - 2: HS
    - 3: LO
    - 4: MI
    - 5: PL
    - 6: VS
    - 7: VC

    - 8: HI
    - 9: LS
    - a: GE
    - b: LT
    - c: GT
    - d: LE
- BL
- BR: The branch target is encoded in the Rn field.
- CBNZ
- CBZ
- EOR
- EORI
- LDUR
- LSL: This instruction uses the shamt field to encode the shift amount, while Rm is unused.
- LS
- OR
- ORRI
- STUR
- SUB
- SUBI
- SUBIS
- SUBS
- MUL
- MOVK: This is an IW instruction; however, it has a nine-bit opcode--not 11 bits!--follow by a 2-bit special op field. The 2 bit field has the value 0, 1, 2, or 3 corresponding with shifts of 0, 16, 32, or 48 bits, respectively. The data sheet in the textbook gives ambiguous and self-contradictory information about this instruction.
- MOVZ: Same as MOVK
- SDIV
- PRNT: This is an added instruction (part of our emulator, but not part of LEG or ARM) that prints a register name and its contents in hex and decimal. This is

an R instruction. The opcode is 11111111101. The register is given in the Rd field.

- PRNL: This is an added instruction that prints a blank line. This is an R instruction. The opcode is 11111111100.
- DUMP: This is an added instruction that displays the contents of all registers and memory, as well as the disassembled program. This is an R instruction. The opcode is 11111111110.
- HALT: This is an R instruction that triggers a DUMP and terminates the emulator. This is an R instruction. The opcode is 11111111111

You may implement you lds if a compiled language) and runs on pyrite. This restriction is necessary in order to give the TAs a common platform to test and evaluate your solution. Most important, popular, or trendy languages are already installed on pyrite, including C, C++, Java, and Python. If your language of choice is not installed, you are welcome to contact the system administrators <coms-ssg@iast e no guarantees about whether or how timely their service will be.

In order to minimize the burden on the TAs, your s e bourne shell scripts in the top level directory: build.sh and run.s the executable command(s) necessary to build your program; if your program is in an interpreted language, this final may be empty. run.sh should take one parameter, the name of a LEGv8 binary file, and pass it to your disassembler. For instance, if I were implementing my disassembler in C and had the program in a source file named disasm.c, my build.sh would contain exactly:

```
gcc disasm.c -o disasm
```

And my run.sh would contain exactly:

```
./disasm $1
```

Bourne shell script are simply text files, and *$1* is interpreted as the first positional parameter passed to the script on the command line. To run your scripts, execute `sh build.sh` or `sh run.sh <legv8 assembly file>`.

The data lost in converting from assembly to machine code are comments and label names. Both of these are completely irretrievable, but new label names can be generated, even if these are devoid of the semantic meanings imparted by the original program

author. For example, you can simply number the labels: "label1", "label2", etc. Your disassembled output should generate new label names such that our emulator can execute or assemble your generated code. Your "reassembled disassembly" should be byte-for-byte identical to the input.

To use the emulator as an assembler (the output file will have the same name is the input with ".machine" concatenated onto the end): `./legv8emul <legv8 assembly file> -a`

To run the emulator in binary emulation mode: `./legv8emul <legv8 binary file> -b`

Here is example C code to load the binary program from disc:

```
//This is in main()
    if (binary) {
      fd = open(argv
      fstat(fd, &buf);
      program = mmap(NULL, buf.st_size, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE, fd, 0);
      bprogram = calloc(buf.st_size / 4, sizeof (*bprogram));
      for (i =
        progra
        decode(program[i], bprogram + i);
      }
      emulate(bprogram, buf.st_size / 4, &m);
      return 0;
    }
```