MANUAL TO USE THE ISA

-AYUSH PAREEK

THE PROGRAMS

PROGRAM 1 :: Find frequency of a given number in an integer array

PROGRAM 2 :: Binary Search for an integer

PART 1 :: FROM ASSEMBLY CODE TO BINARY

How to use ?

**=>**Extract everything to the single folder

=>Put the assembly code in **‘input\_instructions.txt’**

=>Run main and give the base address as 200 (or any value much lesser than 1024)

=>Output machine code is found in **‘output\_machine\_code.txt’**

**NOTE :: Pre-run inputs and outputs can be found ‘Assembly CODE 1’(‘Assembly CODE 2’) and ‘Output Code1’(‘Output Code2’)**

Abstract Explanation of Algorithm

**STEP1::** Opcodes along with their assigned machine codes and expected operand formats

i.e.

/\*\*

rri = reg reg imm. addr.

rrr = reg reg reg

ri = reg imm.

rr = reg reg

a = add

z = zero

\*/

are read from **‘input\_opcode.txt’** by the program and are stored as a **HASH TABLE( to reduce time complexity while searching for those opcodes)**

**/\*We have used 2 pass assembler to remove forward addressing problem\*/**

**STEP2::** In pass 1, a symbol table of all labels is generated which can be read in **‘symbol\_table.txt’**

**STEP3::**In pass 2, the program in again read and symbol table is used to get the corresponding addresses associated with each label. Each instruction is converted to 32 bit machine code

Addressed associated with registers

R0-----> 00000

R1-----> 00001

R2-----> 00010

R3-----> 00011

R4-----> 00100

R5-----> 00101

R6-----> 00110

R7-----> 00111

R8-----> 01000

R9-----> 01001

R10-----> 01010

R11-----> 01011

R12-----> 01100

R13-----> 01101

R14-----> 01110

R15-----> 01111

/\*SPECIAL ARRAY REGISTERS FOR STORING THE SIZE OF THE ARRAY\*/

A0-----> 10000

A1-----> 10001

A2-----> 10010

A3-----> 10011

PART 2 :: FROM BINARY CODE TO EXECUTION

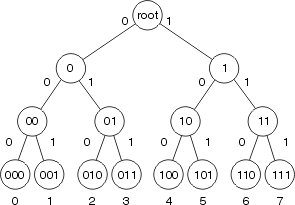
How to use ? (similar to previous file)

**/\*Use testbit.c with gcc -lm\*/**

Abstract Explanation of Algorithm

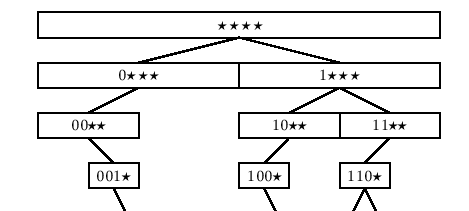
We are storing the opcodes in a Binary Trie .

Thus we have refrained from storing the machine codes corresponding to opcodes in additional space. This data structure saves huge amount of memory since no machine code is actually being stored.



The searching also takes place using this trie. Thus, we have implemented the most optimized scheme of

 Using trie, search complexities can be brought to most optimal limit (opcode length). This guarantees search in O(p) time where p is the length of the largest opcode. This scheme is much better than trial and error using masking in both time and space complexities.



The opcodes corresponding to the machine instruction is extracted and executed.