Report

CS23BTECH11018 - Dondeti Samhitha
CS23BTECH11049 - Ramanolla Shravani

Design Section:

Storing Data :

- To store all information like instruction name and their corresponding opcode, funct3, funct7 etc.. map container from STL is used.
- A struct named ***InstructionDetails*** stores details like instruction format type (***FMT***), ***opcode, funct3*** and ***func7.***
- Then a map named ***Details*** that maps a string (instruction names like add ,sub etc..) to the struct ***InstructionDetail*** mentioned above.
- <bitset> STL container is used to store the data in binary form for variables like ***opcode, funct3 , funct7*** etc…
- And also we did not have any separate field for funct6 in the struct ***InstructionDetails***, we stored funct6 value in the funct7 field (bitset of size 7) itself , but while using this value the MSB (7th bit) is neglected in the implementation.
- If an instruction does not have funct7 or funct3 we store -1 in those places, since those values are not used later in our implementation.
- To store alias names of registers a map name ***regDetails*** is used. It maps a string (register name like t1 , sp , x2, etc ..) to an integer (register number 0 to 31).

String Parsing :

- To store all instructions an array of vectors (***`vector <string> arg[N]`***) is used where we define N as 100 using define keyword.Where 100 is maximum number of instructions that can be read from input.s. The value of N can be changed depending on how many maximum instructions we need to support.
- In the main function after opening input.s file, each line read and stored it to a string ***s*** and then parsed the string and divided it into arguments and stored each argument in a vector of ***arg[i]*** where i is the instruction number.
- And also this implementation supports empty lines and comments so the instruction number and the line number corresponding to that instruction in input.s file may not be the same, so a map named ***LineNumber*** is used which maps an integer(instruction number) to another integer (line number in input.s file). This map is useful while displaying errors to the user.

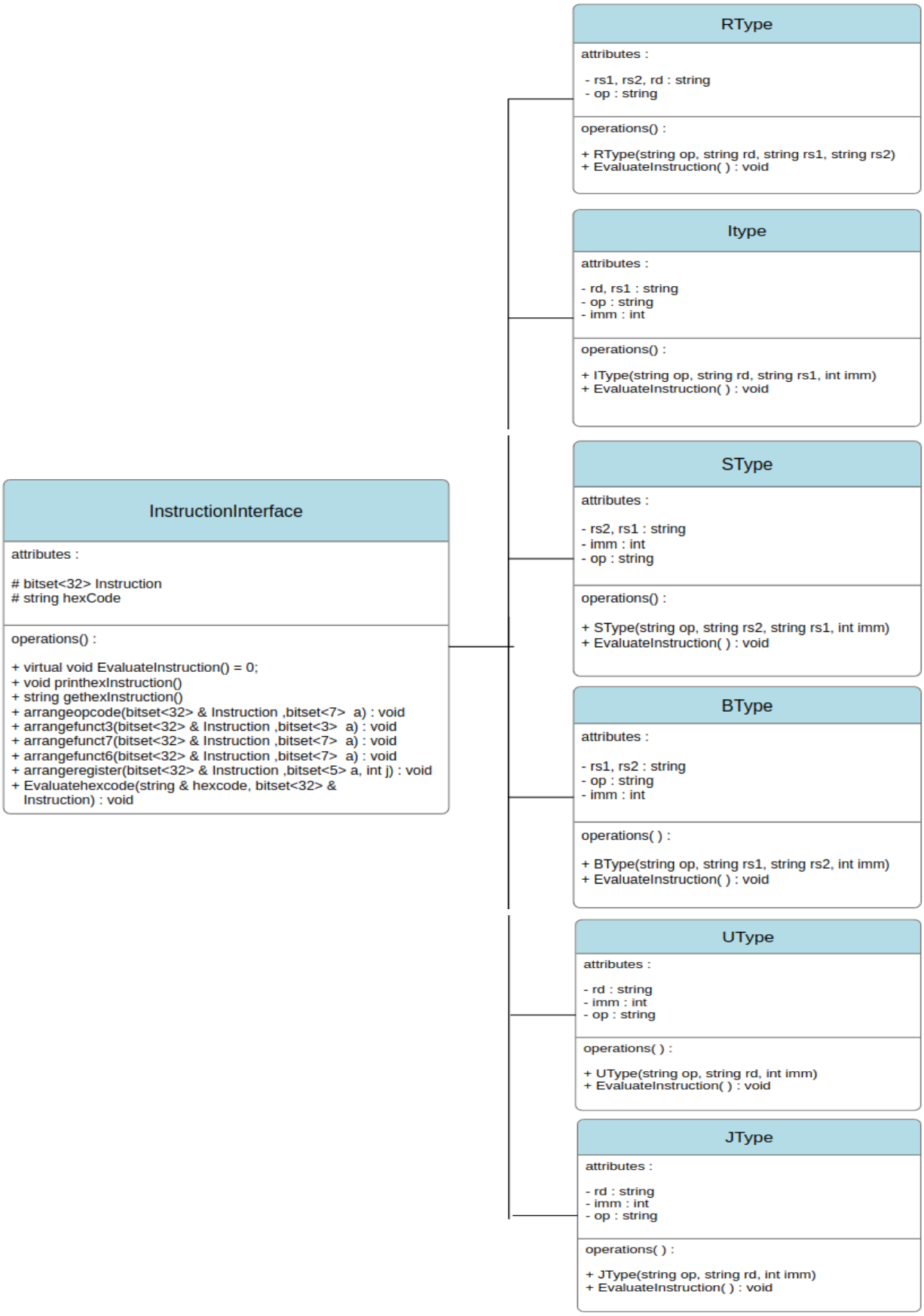
- During string parsing, if any Label is encountered, that label is stored in a map named ***Labels*** that maps a string (label name) to a pair of integers (instruction number and corresponding lineNumber). This map is useful while calculating offset for branch or jump instructions and to show errors to users.

Class Hierarchy :

- In class hierarchy, an interface named ***InstructionInterface*** that has its protected data members : ***Instruction*** (bitset<32> type holds binary form of instruction) and ***hexCode*** (string that holds hexadecimal form of instruction).
- The above 2 data members are common for all instruction format types so they are kept in parent class ***InstructionInterface.***
- ***RType, IType, SType, BType , UType and JType*** are inherited classes from ***InstructionInterface*** in this implementation***.***
- ***InstructionInterface*** has functions like ***arrangeopcode, arrangefunct3, arrangeregister*** etc … which takes binary form(***bitset***) of data like registers, opcode, funct3 etc.. and arrange them in ***Instruction*** data member mentioned above according to their positions.
- ***InstructionInterface*** also has a pure virtual function ***EvaluateInstruction*** which is defined in inherited class according to the format type class like ***RType, IType*** etc.. This function arranges registers, immediates , opcodes ,funct3 etc.. in ***Instruction*** data member by using the functions in ***InstructionInterface*** mentioned above.
- We also had a map named ***hexDigits*** that maps a string (4 bit binary like “1010”) to a char (hex digit). It is useful to convert binary form to hexadecimal form of instruction.
- In the main function, after storing all instruction arguments in an array of vectors ***arg*** we iterate through each instruction i.e each vector arg[i] and identify the Instruction format using the ***Details*** map mentioned above and checks whether the arguments provided are valid registers or labels etc.. and then create an Object of Class according to Instruction format type and call ***EvaluateInstrunction*** function. And then prints the hexadecimal form of instruction to the output.hex file.

UML Diagram :

The UML Diagram below shows the hierarchy of InstructionInterface class. This UML Diagram shows the attributes and operations of each class. Here the InstructionInterface is an abstract class.


InstructionInterface
attributes :
bitset<32> Instruction
string hexCode
operations() :
+ virtual void EvaluateInstruction() = 0;
+ void printhexInstruction()
+ string gethexInstruction()
+ arrangeopcode(bitset<32> & Instruction ,bitset<7> a) : void
+ arrangefunct3(bitset<32> & Instruction ,bitset<3> a) : void
+ arrangefunct7(bitset<32> & Instruction ,bitset<7> a) : void
+ arrangefunct6(bitset<32> & Instruction ,bitset<7> a) : void
+ arrangeregister(bitset<32> & Instruction ,bitset<5> a, int j) : void
+ Evaluatehexcode(string & hexcode, bitset<32> & Instruction) : void
RType
attributes :
- rs1, rs2, rd : string
- op : string
operations() :
+ RType(string op, string rd, string rs1, string rs2)
+ EvaluateInstruction( ) : void
Itype
attributes :
- rd, rs1 : string
- op : string
- imm : int
operations() :
+ IType(string op, string rd, string rs1, int imm)
+ EvaluateInstruction( ) : void
SType
attributes :
- rs2, rs1 : string
- imm : int
- op : string
operations() :
+ SType(string op, string rs2, string rs1, int imm)
+ EvaluateInstruction( ) : void
BType
attributes :
- rs1, rs2 : string
- op : string
- imm : int
operations( ) :
+ BType(string op, string rs1, string rs2, int imm)
+ EvaluateInstruction( ) : void
UType
attributes :
- rd : string
- imm : int
- op : string
operations( ) :
+ UType(string op, string rd, int imm)
+ EvaluateInstruction( ) : void
JType
attributes :
- rd : string
- imm : int
- op : string
operations( ) :
+ JType(string op, string rd, int imm)
+ EvaluateInstruction( ) : void

Error handling:

- To handle errors there are some functions like ***IsValidImmediate , IsvalidLabel, IsValidRegister , IsvalidInstruction*** etc …
- ***IsValidRegister*** function takes a string as a parameter and checks whether it is a valid register or not by checking whether it is present in a map ***regDetails*** mentioned above.
- Similarly ***IsValidOperation*** function takes a string as a parameter and checks whether it is a valid operation or not by checking whether it is present in a map ***Details*** mentioned above.
- ***IsValidLabel*** function takes a string as a parameter and checks whether it is a valid label or not by checking whether it is present in a map ***Labels*** mentioned above.
- ***IsValidImmediate*** function takes a string as a parameter and checks whether it is a valid immediate or not either in the decimal number system or hexadecimal number.
- ***seperateImmediate*** function takes a reference of a string(like offset(register)) as parameter and separates offset and register and returns offset as integer and updates the parameter string as register string.
- And also there is a function named ***convertToInt*** which converts string to Integer according to whether it is in decimal number system or Hexadecimal number system.

Test cases :

- In a filename GenerateTestCases.cpp , there are functions like ***GenerateRType, GenerateUType etc ..*** which randomly generates instructions of that type by choosing registers and operations randomly.
- By using those functions we can generate test cases of different type and store it in filename ***testcases.s*** . We can take this file as an input file and run our program and also get the hexadecimal code from ripes or any other stimulator . And compare them by using ***diff*** command in the terminal.

  Command:

  diff -w output.hex riscv.hex
- The correctness of the code is also checked by taking the edge cases of immediate values manually and checked the output hex code with the hex code provided by ripes or any other simulator.

Facilities Available:

- It supports all types of instructions Rtype, IType , BType ,SType , JType, UType.
- It also supports comments. (The content written after ; in that line is treated as comment)
- It supports empty lines .
- It supports instructions either with or without comma between arguments (just a space character is enough between arguments)
- It supports both Decimal number system and hexaDecimal number system for immediates.

Limitations:

- In the input file there should be no space character between the same argument (like 32(rs1) there is space between (and 3, then it will treat it as 2 separate arguments similarly arguments like rs 1 or su b are treated as 2 arguments.)
- It does not support any pseudo instructions.