Report

CS23BTECH11018 - Dondeti Samhitha
CS23BTECH11049 - Ramanolla Shravani

Design Section:

Class Hierarchy :

Memory :

To design Memory Class we had 3 other classes named ***textSectionClass, dataSectionClass, stackSectionClass.***

textSectionClass:

- In textSectionClass we have two private members : ***vector<bitset<8>> mem*** , ***unsigned int top.***
- We stored each byte of a memory as ***bitset<8>***. So ***mem*** is an array of bytes in memory.
- The Constructor textSectionClass initializes the ***mem*** to size 4*S i.e 40 . It means initially it can store up to S(i.e 10) number of risc v instructions or 4*S i(i.e 40) bytes of memory.
- ***unsigned int top*** points to the top of ***mem*** vector i.e least index of ***mem*** vector such that all the positions above this index in ***mem*** vector is unoccupied.
- Initially the ***top*** is initialized to 0, when we write an instruction to textSectionClass during load command the ***top*** is incremented by 4(since each risc v instruction is 4 bytes).
- Other than constructor textSectionClass has public functions like : ***writeInstruction , readInstruction , writeData , ReadData , printMemory.***
- ***writeInstruction*** function takes a string ***s*** that contains hexadecimal format of encoded instruction containing 8 characters and store this instruction in textSectionClass in form of binary in 4 bytes (i.e 4 bitset<8> ‘s) and increment ***top*** by 4.
- ***readInstruction*** function takes an ***unsigned int pc*** as input and returns an instruction (i.e 4 bytes) located at the address ***pc*** i.e it stores 4 bytes starting from address ***pc*** and stores it in bitset<32> and returns it.
- ***writeData*** function takes an ***int n, unsigned int address, bitset<64> value***. It stores ***n*** bytes data from ***value*** (starting from LSB i.e from ***value[0]***) in ***mem*** vector starting from index value ***address*** .
- ***readData*** function takes an ***int n, unsigned int address*** . It reads ***n*** bytes data from ***mem*** vector starting from index value ***address*** and stores it in ***bitset<64>*** starting from index 0 and returns it. Note: If n < 8 then remaining bits in bitset<64> are filled with zeroes.

- printMemory function has 2 variants one does not take any arguments it prints complete memory from 0 to 0x10000 , another variant takes ***int n , unsigned int address*** as arguments and prints only ***n*** bytes of memory starting from index ***address***.

dataSectionClass:

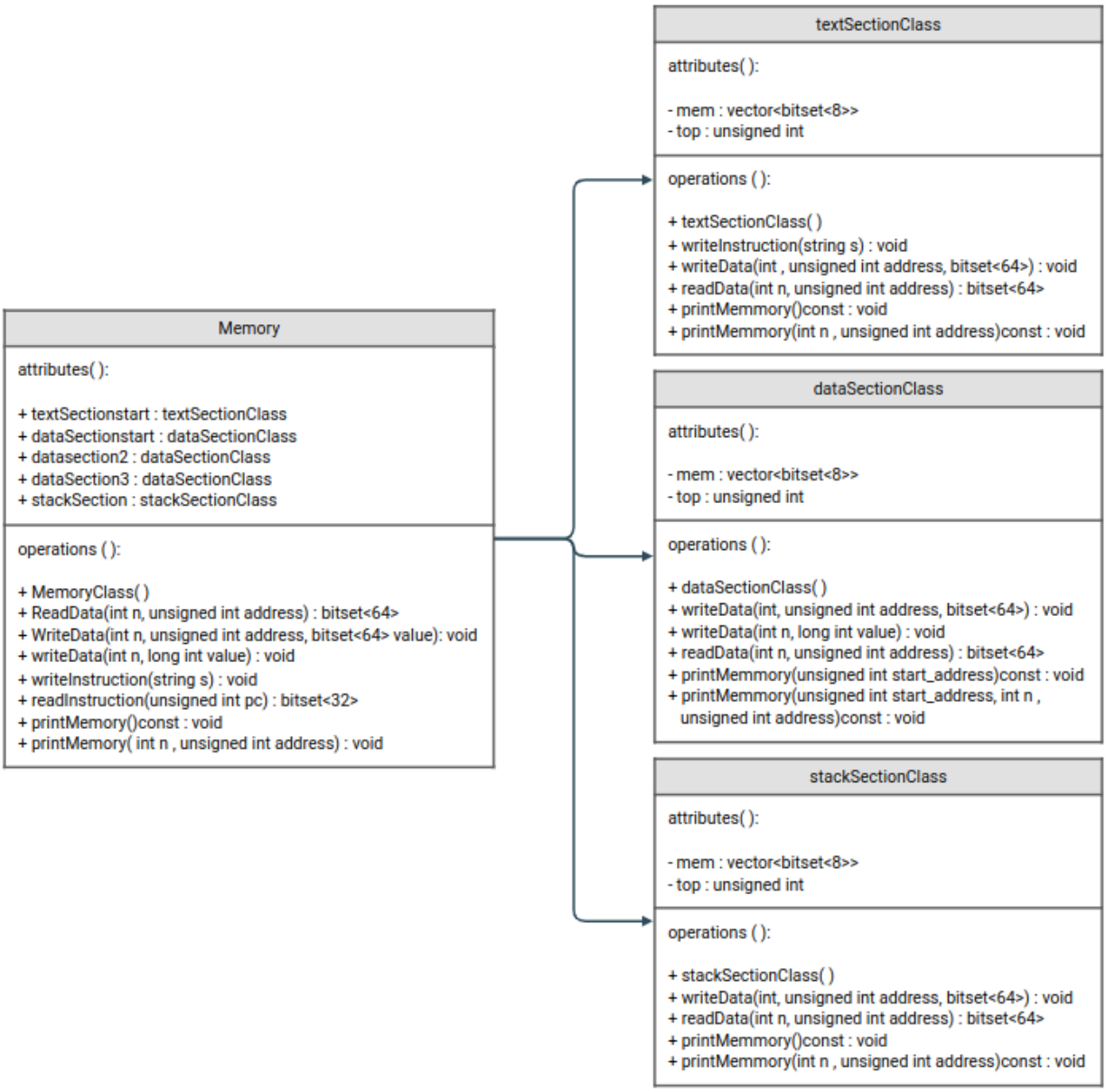
- ***dataSectionClass*** is similar to ***textSectionClass*** but this class does not ***writeInstruction and readInstruction*** functions, since we are assuming the number of instructions in the input file is not very large. If you want you can just add these functions to ***dataSegmentClass*** also.
- For any object of this class index 0 of ***mem*** vector refers to the starting address of that object. i.e ***mem[0]*** in ***dataSectionstart*** in ***Memory*** refers to address ox10000 , whereas it refers to address 0x20000 and 0x30000 in ***dataSection2*** and ***dataSection3*** respectively.
- So ***readData*** or ***writeData*** functions receive the index for that corresponding object (i.e actual address - 0x10000 for ***dataSectionStart***) , not the actual address of the memory.

stackSectionClass:

- ***stackSectionClass*** is similar to ***dataSectionClass. The only*** difference is that the ***stackSectionClass*** index 0 in ***mem*** vector (i.e ***mem[0]***) corresponds to address 0x4ffff not starting address 0x40000.
- So index ***i*** in the ***mem*** vector (i.e ***mem[i]***) corresponds to address ***0x50000 - i - 1 .***

MemoryClass:

- For this Implementation we considered memory addresses from 0 to 0x50000.
- In MemoryClass we used different classes like ***textSectionClass, dataSectionClass, stackSectionClass.***
- In ***MemoryClass*** we have one ***textSectionClass*** object named ***textSection*** that handles the memory from address 0 to 0x10000 , three ***dataSectionClass*** objects names ***dataSectionStart , dataSection2 , dataSection3*** which handles memory from address 0x10000 to 0x20000 , 0x20000 to 0x30000 , 0x30000 to 0x40000 respectively and one ***stackSectionClass*** object named ***stackSection*** which handles memory from address 0x40000 to 0x50000..
- All three classes :***textSectionClass, dataSectionClass, stackSectionClass.*** has functions like ***readData , writeData and printMemory functions***. The only difference is in stackSectionClass that 0 index in mem vector corresponds to address 0x4ffff not 0x40000.
- ***readData, writeData and printMemory functions*** checks address and calls any of ***readData, writeData and printMemory*** functions of ***textSectionClass, dataSectionClass, stackSectionClass*** according to the address. I.e when the address is between 0x20000 to 0x30000 it calls from ***dataSection2*** or if the address is between 0x40000 and 0x50000 it calls from ***stackSection.***



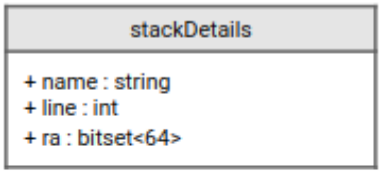
RegisterFileClass:

- To store registers we had a class named ***RegisterFileClass***.
- ***RegisterFileClass*** contains one private data member: ***bitset<64> reg[32]*** .
- Since all 32 registers are of 64 bits, they are stored in an array of bitset<64> of size 32.
- It has three public functions: ***bitset<64> readReg(int n) const***,
  void writeReg(int n, bitset<64> value), void printRegs()const .
- ***readReg*** function takes an ***integer n*** as a parameter and returns the value of ***n***th register i.e reg[n].

- ***writeReg*** function takes an ***integer n*** and a ***bitset<64> value*** as parameter and stores the ***biset<64> value*** in ***n***th register i.e ***reg[n]=value*** if ***n*** is not equal to 0 since x0 is always zero.
- ***printRegs*** function prints all register values in hexadecimal format.

StackDetails:

- To maintain the function stack we used a variable named ***functionStack*** which is a vector of a struct named ***StackDetails.***
- StackDetails is a struct that has 3 fields : string ***name***; int ***line***; bitset<64> ***ra .***where ***name*** stores the label name or function name of that stack. And ***line*** stores the line number of last executed instruction in that function and ***ra*** stores the return address of this function .
- We use ***ra*** while popping from ***functionStack*** , we will verify whether we are popping the correct stack pointer by checking the return address.



Execution Flow:

- First we will take the input command from the user and according to the input command we execute different pieces of code.
- If the input command is in the form ***load filename*** . We will read the file with a filename as ***filename*** that contains an assembly code and encode those instructions and store them in textSection by using the ***writeInstruction*** function in Memory
- If the input command is in the.form ***break n .***It will put the breakpoint at line number n if there exists an instruction at line n else it will keep the breakpoint at the line containing the nearest next instruction.
- If the input command is in the.form ***del break n .*** It will delete the breakpoint at line number n if there exists a breakpoint at line n else it says that there is no breakpoint at line n to delete.
- If the input command is in the.form ***run.*** It will execute the upcoming instruction until it reaches the break point i.e before executing any instruction it checks whether there is a breakpoint at that instruction . If yes then it will stop executing .Otherwise it executes that instruction.

- If the input command is in the.form ***step.*** It will execute only one upcoming instruction and stop.

Error handling:

- When a user asks to keep a breakpoint at an empty line or a line that does not correspond to instructions like inside .data section or comment etc .. we will keep the break point to the next nearest instruction line number.
- If the user asks to delete breakpoint at a line number where there is no breakpoint then it shows an error message.
- If a user enters commands like run ,step without loading file it shows an error message saying file is not loaded and also if there are any errors in loaded file it shows where errors are present and does not allow you to execute instructions , so the user has to fix the errors and load it again.
- If code encounters any runtime errors like accessing invalid memory (like <0 or >0x50000 or when pc value changes to -ve or greater than line number in file) it shows an error message saying runtime error and stops executing further instructions.

Test cases :

- We tested our code with provided test cases and also checked with SanityChecker provided in moodle.
- We checked our code with the GCD code of Lab2.
- The correctness of the code is also checked by taking the edge cases of immediate , jump instructions and shift instructions etc .. and checked with ripes simulator.

Facilities Available:

- It also supports comments. (The content written after ; in that line is treated as comment)
- It supports empty lines .
- It supports the .data section multiple times if you mention .data and .text properly.
- 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.
- It supports all types of instructions Rtype, IType , BType ,SType , JType, UType.
- We added extra commands like ***show memory , show break points*** where show memory shows complete memory and show break points shows line numbers where breakpoints are available.

- And also while popping a function from stack when we use jalr we also compare the return address to check whether it's popping the correct stack frame.

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.
- Number of instructions in input.s should not be very large such that the instruction should fit in text section memory.