Brookshear machine emulation code explained

Steps:

1. Load the file and read the text
2. Pre-process the code (still not running any assembly code). It has to be done in this order, otherwise it won’t work:
   1. Remove whitespace on either side
   2. Remove all comments
   3. Remove empty lines
   4. Get rid of label definitions (human text at certain points in the code to indicate areas in the code, such as the word startloop to indicate the start of a loop and endloop to indicate the end of a loop) and saving its position as its address in memory (e.g. if startloop is on line 5, we have a dictionary key-valid pair as “startloop”, 8 because of index value). We are saving it to a hex value
   5. Finally, we go through the assembly code once more, and check each line to see whether a label definition is in the code (e.g. if we find the word startloop within the code, then we replace startloop with 8).
3. Assemble the code
   1. Create a byte array with 256 values
   2. We get the pre-processed assembly text and save it to an internal string array (e.g. the first instruction of text is the first item in the string array, second instruction second item, etc. It should just naturally pass through because they’re both string arrays anyways)
   3. Write instructions/Convert it to assembly code. To pass what we’ve written to assembly code, we need to go through each line as a for loop and convert it into a hexadecimal notation (e.g. such as E10F). First of all, we create two variables called opcode (what the function is, such as MOV for move) and operand (data for the function), on the left- and right-hand side respectively. We can do this by creating a switch statement that checks if a line contains a certain value (e.g. contains AND/MOV). Most of the lines will need specific methods to grab the specific opcode and operand. We can check what each instruction is meant to be by checking the Brookshear Machine Emulation documentation.  
      A couple of examples:
      1. MOVR1->RA  
         The MOV uses the opcode of 4, and then a 0 afterwards. Then it takes a first position (r), a second position (s), and moves the content from r to s, in our example from 1 to A. So the whole instruction is: 401A
      2. ANDRA,R2->RA  
         This specific AND uses the opcode 8. The next letter is where it’s going into, which in our case is A. The next two letters are the inputs, which are A and 2. So the whole code here is 8AA2
      3. ROTRA,4  
         We rotate the bits of RA four bits to the right. The opcode for ROT is A, and then we give the register, which is A. This is followed by a 0, and then followed by the number of bits to shift by, which in this case is 4. So the whole instruction is AA04.

When we’re getting the opcode and the operand, we’re always writing it in the form 0xXY, with X and Y representing different characters between 0 and F. This forces the compiler to make the value hexadecimal.

* 1. When we’ve got both the opcode and operand, we pass them into an array, with each one representing a memory cell. E.g. [40, 1A, 8A, A2,… etc]. We do it like this instead of combined because the Brookshear machine emulator is meant to represent 8 bit machine, and each of the two characters together represent one byte. Because we’re adding to an array, we have to do code like this: machineCode[j] = opcode; machineCode[j + 1]. This also means that when we go through each instruction, we need to add it to the byte array which already exists, and we need to add 2 every time (e.g. j += 2) to force the instructions onto the next bit.

1. We create a machine object, pass through the fully assembled machine code, and now we go through and execute the code
   1. Firstly, we pass through our machine code into the machine object, saving it to an array variable.
   2. Then we go through each step of the instruction. Each step of Brookshear Machine Emulator, as well every other machine emulator, goes through the same three stage: Fetch(), Decode(), Execute().
   3. Fetch – The fetch method grabs a single piece of machine code from our array variable (in this case it would be the two letters), and puts it into the CIR (Current Instruction Register). It shifts it 8 bits (1 byte, or in human readable form it’s 2 characters) to the left and then it adds the next instructions onto the end. For example, [40, 1A,…], to do it properly with the correct data type, we would load the first part as 0040, we’d shift it 8 bits to become 4000, and then we’d add 1A to the CIR, making it 401A. Finalised the machine code once again
   4. Decode – We pass through our finalised machine code once to the decode, and we grab the first character (which is the opcode). We can do this very easily with a switch statement. When we grab the first character, we then set the variable instruction to whatever the specific function will be (e.g. if the first letter is C, then we set the variable instruction to HALT. That’s literally it. In c# (and other languages), you can save a function to a variable, and then call it later on.
   5. Execute – Once we have grabbed the correct function, we then just executed it. The way that each instruction is executed is going to be done differently depending on the number of variables that are being passed through.
      1. For example, the LoadDirect instruction has a first variable, register r, which is 1 character long (one Nibble), and another variable, the address, which is two characters long (one byte). Other instructions such as ADDI have three characters, which requires a different form of string manipulation.
      2. Once we grabbed all of our variables (bytes and nibbles), and we’ve manipulated them in the correct way, we can then either save them to a register address (for which we have 16) or to a memory address (for which we have 256).
      3. If we’re doing a jump, it’s slightly complicated. Every jump has a conditional statement that returns either true or false, but some conditionals guarantee a value of true because they compare register 0 to register 0 (comparing a value with itself), and other conditional compare the value of register 0 to the value of a given register.

There are two types of jumps within the Brookshear machine emulation: one that uses register addresses, and one that simply deals with addresses. The one that doesn’t deal with register addresses (starting with opcode B) only has an if equal comparator: if the content of register address r is equal to the content of register address 0, then immediately move to memory address xy (e.g. Brxy) B422 would check if register address 4 was equal to register address 0. If they were equal, then we’d move to memory address 22).  
The other type of jump, which begins with opcode F, also does a boolean comparison (but also adds options for greater than, more than, not equal to, etc). It compares register address r with register address 0, and if the boolean comparator returns true, then it goes to the memory address which is stored in register address t (e.g. Frxt. For F405, we are checking if register address 4 and register address 0 are the same value (equal to one another is signified by the 0 in the equation, which is the third character in the instruction). If this is true, then we go to the memory location of register address 5. So if register address 5 had the value 66, then we’d move to memory address 66)