Angela Seidel

5 November 2020

Controller Class Planning Stage & 30 New Instructions

**Schedule:**

* Nov. 7-13th (6 days): Try your best to get the classes completed by the end of the day
* Nov. 14-16th (3 days): Try to bring all the classes together to make the program functioning, make any extra media as necessary
* Nov. 17th: Final Turn-in due by midnight, try to only worry about any extra document organization for final turn-in

**The teacher would like the program to have at least 30 Pep/8 Instructions in PRJ#02.**

* In Pep/8, the masterlist of instruction can be found through “Help > Pep/8 Reference”

**Key:**

* Light Blue= Class Complete
* Red= In-progress of being finished. People that are tasked with completing this class are listed on the right
* ~~Strike through~~ = No longer required with Scope change posted on Thursday.
* Yellow = Needs to be assigned to someone with the Scope change posted on Thursday.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Thursday Reduced Project Scope Labels** | **Instruction Specifier** | **Mnemonic** | **Instruction** | **Addressing Modes** | **Status Bits** | **Assigned People** |
| STOP(1/1) | 0000 0000 | STOP | Stop execution | U |  | James (Done) |
|  | 0000 0001 | RETTR | Return from trap | U |  |  |
|  | ~~0000 0010~~ | ~~MOVSPA~~ | ~~Move SP to A~~ | ~~U~~ |  | ~~James~~ |
|  | ~~0000 0011~~ | ~~MOVFLGA~~ | ~~Move NZVC flags to A~~ | ~~U~~ |  | ~~James~~ |
| BR (1/9) | 0000 010a | BR | Branch unconditional | i, x |  | Caleb |
| BR (2/9) | 0000 011a | BRLE | Branch if less than or equal to | i, x |  | Caleb |
| BR (3/9) | 0000 100a | BRLT | Branch if less than | i, x |  | Caleb |
| BR (4/9) | 0000 101a | BREQ | Branch if equal to | i, x |  | Caleb |
| BR (5/9) | 0000 110a | BRNE | Branch if not equal to | i, x |  | Caleb |
| BR (6/9) | 0000 111a | BRGE | Branch if greater than or equal to | i, x |  | Caleb |
| BR (7/9) | 0001 000a | BRGT | Branch if greater than | i, x |  | James |
| BR (8/9) | 0001 001a | BRV | Branch if V | i, x |  | Angela |
| BR (9/9) | 0001 010a | BRC | Branch if C | i, x |  | Angela |
|  | 0001 011a | CALL | Call subroutine | i, x |  |  |
| Bit Operations (1/6) | 0001 100r | NOTr | Bitwise invert r | U | N Z | Angela |
| Bit Operations (2/6) | 0001 101r | NEGr | Negate r | U | N Z V | Kiet |
| Bit Operations (3/6) | 0001 110r | ASLr | Arithmetic shift left r | U | N Z V C | Kiet |
| Bit Operations (4/6) | 0001 111r | ASRr | Arithmetic shift right r | U | N Z C | Kiet |
| Bit Operations (5/6) | 0010 000r | ROLr | Rotate left r | U | C | Kiet |
| Bit Operations (6/6) | 0010 001r | RORr | Rotate right r | U | C | Kiet |
|  | 0010 01nn | NOPn | Unary no operation trap | U |  |  |
|  | 0010 1aaa | NOP | Nonunary no operation trap | i |  |  |
|  | 0011 0aaa | DECI | Decimal input trap | d, n, s, sf, x, sx, sxf | N Z V |  |
|  | 0011 1aaa | DECO | Decimal output trap | i, d, n, s, sf, x, sx, sxf |  |  |
|  | 0100 0aaa | STRO | String output trap | d, n, sf |  |  |
| CharIn Char(1/2) | 0100 1aaa | CHARI | Character input | d, n, s, sf, x, sx, sxf |  | Angela |
| CharOut Char(2/2) | 0101 0aaa | CHARO | Character output | i, d, n, s, sf, x, sx, sxf |  | James (Done) |
|  | 0101 1nnn | RETn | Return from call with n local bytes | U |  |  |
|  | ~~0110 0aaa~~ | ~~ADDSP~~ | ~~Add to stack pointer (SP)~~ | ~~i, d, n, s, sf, x, sx, sxf~~ | ~~N Z V C~~ | ~~Kiet~~ |
|  | ~~0110 1aaa~~ | ~~SUBSP~~ | ~~Subtract from stack pointer (SP)~~ | ~~i, d, n, s, sf, x, sx, sxf~~ | ~~N Z V C~~ | ~~Kiet~~ |
| ADDr Operation (1/5) | 0111 raaa | ADDr | Add to r | i, d, n, s, sf, x, sx, sxf | N Z V C | James (Done) |
| SUBr Operation (2/5) | 1000 raaa | SUBr | Subtract from r | i, d, n, s, sf, x, sx, sxf | N Z V C | James (Done) |
| ANDr Operation (3/5) | 1001 raaa | ANDr | Bitwise AND to r | i, d, n, s, sf, x, sx, sxf | N Z | Angela |
| ORr Operation (4/5) | 1010 raaa | ORr | Bitwise OR to r | i, d, n, s, sf, x, sx, sxf | N Z | Angela |
| CPr Operation (5/5) | 1011 raaa | CPr | Compare r | i, d, n, s, sf, x, sx, sxf | N Z V C |  |
| LD/STr Instructions (1/4) | 1100 raaa | LDr | Load r from memory | i, d, n, s, sf, x, sx, sxf | N Z | James (Done) |
| LD/STr Instructions (2/4) | 1101 raaa | LDBYTEr | Load byte r from memory | i, d, n, s, sf, x, sx, sxf | N Z | James |
| LD/STr Instructions (3/4) | 1110 raaa | STr | Store r to memory | d, n, s, sf, x, sx, sxf |  | James (Done) |
| LD/STr Instructions (4/4) | 1111 raaa | STBYTEr | Store byte r to memory | d, n, s, sf, x, sx, sxf |  | James |

The items below are pending implementations of non-existent classes or additions to existing classes.

|  |
| --- |
| Pep8GUIFrame \*Angela\* |
| - myController: Controller // The controller is a field inside the Pep8GUIFrame class. Keep this in mind with updating the GUI fields from what is given in the Controller states.  - myAssembler: Assembler // The assembler is only needed within the Pep8GUIFrame class. The Controller has nothing to do with code assembly. the Controller only reads object code when it is present in the GUI, whether it was written by the user or generated by the Assembler class. |
| - assembleSourceCodeToObjectCode(): void // simply takes the input from the mySourceCodeTextArea field, passes it as an input String to Assembler.assembleSourceCodeToObjectCode(String theSourceCode), then takes the output of that method and writes its contents to the myObjectCodeTextArea field. This extra processing method will only be run when the “Run Source” button is pressed.  - updateMemory(theControllerMemory:String[]): void // Takes the state of the Memory object from the Controller and formats it to look like the visual representation in the original Pep/8 Program for the user. The first column is the first hexadecimal address of the instruction line, and the second column is the next eight hexadecimal instructions for that line.  - updateCPUStates(): void // Use the getters from myController to update all of the internal states of the CPU, the Output, the Memory Dump representation, etc. |

|  |
| --- |
| Assembler \*Kiet\* (Can ask for help from Angela) |
|  |
| - assembleSourceCodeToObjectCode(theSourceCode:String): String -) Translates the Source Code from a String format into Object Code, then writes the results to an output String. This output string will be used by the GUI to write content to the Object Code text area. -) When reading theSourceCode String, it may include format specifiers like /n and /t that the user used to organize the Source code formatting. This may require extra calculations to read from the String in a literal sense while ignoring the format specifiers. -) The translation is from assembly language(written keywords) to machine language(hexadecimal). -) Machine Language (hexadecimal) should exist in groups of two with a space between them. It does not matter if the hex is represented in one continuous line or has multiple lines to its format.  -) If an error exists within the Source Code, you can throw an exception in the terminal and set the return string to Null   * The assembler will be called when the Pep8GUIFrame.java “Run Source” button is pressed. The GUI will call “assembleSourceCodeToObjectCode(String theSourceCode),” to which the Assembler object will translate the Source Code to Object Code, then the output of the method will be written to the Object Code text area by Pep8GUIFrame.java |

|  |
| --- |
| Controller \*James makes Base\* (Can ask for help from Kiet, Angela) |
| - ~~myConverter: Converter~~---- static methods  - myDecode----Decode  - myMemory-----MemoryUnit  - myAR -----AccumulatorRegister: String (binary representation)  - myIndexRegister: String (binary representation)  - mySP------StackPointer: int // Where the Stack Pointer is currently located (in decimal, for the MemoryUnit’s array format)  - myPC------ProgramCounter: int // Where the Program Counter is currently located (in decimal, for the MemoryUnit’s array format)  - myIR------InstructionRegister:String (24bits binary)  - myNFlag: int (1 or 0 value)  - myZFlag: int (1 or 0 value)  - myVFlag: int (1 or 0 value)  - myCFlag: int (1 or 0 value)  - myInput: String (what the user wrote to the Input text area in the GUI)  - myOutput: String (what will be written in the Output text area in the GUI) |
| + Controller(): void  + loadObjectCodeIntoMemory(theObjectCode:String): void  -) Takes a String representation of what the user or assembler typed into the “Object Code” text area from the GUI and stores it in the myMemoryUnit object. -) Doesn’t have to check for errors with running yet (unless binary bits are too long or something)  + setInput(theInput:String): void // Takes a String representation of what the user typed in the “Input” text area from the GUI and assigns it to myInput. (This is useful because if we decide to implement a Step button, this prevents the user from modifying the contents of the Input while the program is running step by step).  + run(): void -) loadObjectCodeIntoMemory() is called first -) PC set to 0 -) Then the method goes into a loop until it reads a stop instruction from myMemoryUnit. **Enter Loop, reading a single instruction from memory each iteration:**  -) First, PC increment by 3 to set up for next instruction -) Then, Decoder.decodeInstruction(final String hex) is called -) The Decoder class will have a switch statement that returns a new Instruction-inherited object based on the correct opcode of the binary instruction. -) Then, the Decoder goes back to this run() method, and we call \*newInstructionObject\*.execute(this), which uses getters and setters within this Controller class to write to memory, update NZVC flags, update Instruction Specifiers, etc. ~~-)~~ *~~Execute method should increment the PC where it should go next. This is because if we increment the PC here automatically, then if we Jump to a new location, then the jump will be wrongly incremented.~~*  Angela (original idea-haver) has now decided that this is a bad idea. To match the Fetch, Decode, Execute, etc. cycle, PC will be incremented by 3 automatically at the start, then any sort of PC jumps can be calculated and override PC after if jumps are necessary. **Once the loop ends by the current looped instruction reading an exit condition, the Run method stops.**  // Getters and Setters   * getMemoryUnit() //the toString of MemoryUnit that returns the whole memory and is meant to be used for GUI formatting. Do not worry about this for .execute() methods. * getMyAccumulatorRegister() * setMyAccumulatorRegister() * getMyIndexRegister() * setMyIndexRegister() * getMyStackPointer() * setMyStackPointer() * getMyProgramCounter() * setMyProgramCounter() * getMyInstructionSpecifier() * setMyInstructionSpecifier() * getMyOperandSpecifier() * setMyOperandSpecifier() * getMyOperand() * setMyOperand() * getMyNFlag() * setMyNFlag() * getMyZFlag() * setMyZFlag() * getMyVFlag() * setMyVFlag() * getMyCFlag() * setMyCFlag() * getMyInput() * getMyOutput() * appendToMyOutput()   // Memory Manipulation (inherits from MemoryUnit methods)   * getMemoryDataAt(int index) * storeMemoryAt(int Index, String theData)   + clearMemory(): void // Clears the contents stored in the Memory object myMemory, and also resets the values of the CPU states. |
|  |

**Notes: How to use the myOutput field when using .execute() methods:**

When using an instruction to print to output:

* When reading the instruction for “Write H to Output” which is represented by the machine code of “50 00 48”, reading “50” triggers the Decoder to create a CharOut instruction, then the CharOut.execute() method realizes that “48” is an ASCII “H” and appends that ASCII “H” to the myOutput field in the Constructor. This is done by calling the controller’s appendToMyOutput() method. The MyOutput field will then be written to the GUI so that the user can see that an H was printed to the Output box.