```
void DI(){
//DI(Decode Instruction): Determine the OP CODE and the OPERAND specifiers.
//WHAT structural mechanism in the JAVA language MAY be the best to decode an instruction?
    //Consider that ALL we have so far is the IS(OPCODE) and the OS(OPERAND)if trinary...
    //Like we did in the FI() method we need to 'bracket' the groups;
    //We have already determine if the instruction is Unary/Trinary
    //and set the logical indicator in the FI() method,so we know if an operand
    //is needed(Trinary);
        //
        // At at mimimum, we need to use the basic 'cascading' if/else if structure;
        // like we already did to identify the 'STOP' instruction, we do:
        // if(OP==0x??){ action}
        // else if(OP==0x??){ action }....
        // where the 0x?? is the HEX representation of the operation code
        // obviously this will work but will produce a VERY lengthy sequence...
        // so we take a look at each of the brackets and see that:
        //
        // for the UNARY(logical) bracket, the binary op code has one bit(low order),
        // encoded as an 'r', for ex: 00001 100r (NOT),
        // that represents one of the two user registers, 'A=0' and 'X=1',
        // since the logical instructions are relatively few(6), we could just identify
        // each one individually, like: IF(OP==0X18){//NOTA} else if(OP==19){//NOTX}
        // for a total of 12 else if'. For the register, we do not need to peserve its value
        // since each instruction will invoke a unique method which will implicitly handle
        // the different registers.
        //
        // likewise, for the BR's bracket, the binary op code has one bit(low order),
        // encoded as an 'a', for ex: 0000 010a (BR),
        // that represents one of the two addressing modes used, 'i=0' and 'x=1',
        // since the branching instructions are also relatively few(9), we could just identify
        // each one individually, like: IF(OP==0X04){//BR,i} else if(OP==0x05){//BR,x}
        // for a total of 18 else if'. Here, we need to preserve the addressing mode by
        // assigning it to the cpu.MODE variable, where it will be used later to calculate the
        // effective address.
        //
        // Now, for the last bracket, we realize that the binary op code has:
        // one bit(bit 3), encoded as an 'r', that represents the register(A/X) and
        // three bits(2/1/0), encoded as an 'a', that represents the 8 possible addressing modes.
```

```
        // Since there are 4 bits for each of the 9 instructions in this bracket
        // and if we tried the same technique used for the two prior brackets
        // we would wind up with 2^4=16 flavors for each, needing 9*16=144 else if's!!!
        // SO, we could do it by realizing that the instruction mnemonic only identifies the
        // user register, for ex: ADDX, and the addressing mode is separate and indicated
        // by specifiying a mnemonic letter appended via a comma, for ex: ADDX,d
        // So, if we do it this way we will only need 9*2=18 else if's to identify them...
        // BUT, in order to implement this technique, we need to make the three low order bits,
        // the 'aaa's, all zeros, so that ALL flavors of each instruction will be grouped into
        // one, for ex: ADDA,x, which has a binary encoding of '0111 1101' and
        // ADDA,d, which has a binary encoding of '0111 1001' both become '0111 1000' and thus
        // identified as ADDA's...
        // So we need to identify this last bracket, before we enter the if/else if structure
        // and zero out the 3 low order 'aaa' bits. NOW, we see the use of the global OP variable.
        // we can change it, just so that we can do this kind of trick...
        // So far we have the identified the operation and its mnemonic, which we will assign
        // to the cpu.DESCR variable, for ex: cpu.setDESCR("SUBX")
        // NOW, we need to figure out HOW to handle the addressing mode for this last bracket..
        // At the end of the selection structure, we need once again to identify the last bracket
        // and for it we the need to set the cpu.MODE variable, USING THE ORIGINAL OP CODE,
        // WHICH IS STORED IN THE cpu.IS variable...
        //
        // At this point we would have identified each of the instructions and set the cpu.DESCR
        // variable, with the instruction mnemonic, which also identifies which register it uses
        // with the exception of the BR's, which do not...
        // for the UNARY bracket, there is no addressing mode, but for the other two, we set the
        // cpu.MODE variable...
        // We would have also set the logical variables that control the step execution, accordingly
        // For the unary, the CO/FO/WO are set to false, since there are no operands to read or write
        // For the BR's, we need to do the CO, to produce the effective address for the branching
        // but FO/WO should be set to false..
        // For the last bracket, all instructions deal with an operand, so we need to do the CO
        // and the FO, except for the ST/STBYTE's and the WO will ONLY be executed by the ST/STBYTE...
        //
}// end of the DI() method
```