Assembly Code

```
@ write asm function body here
        MOV R3, R1          @ initialize R3 with STARTING address of centroids10[CENTROID][2]
        LDR R4, =0x0        @ initialize distance between point to centroid0 as 0
        LDR R5, =0x0        @ initialize distance between point to centroid1 as 0
        MOV R6, R0          @ initialize R6 with STARTING address of points10[CENTROID][2]
        LDR R7, =0x0        @ counter for looping through all DATAPOINTS
        LDR R8, =0x0        @ counter for matching ONE point10 (x,y) to TWO centroid10 (x,y)
        MOV R12, R2         @ 0x01A0C002, save the STARTING address of class[DATAPOINT], need to use it later
```

We start off with initialising some registers that will be used later.

```
@ each loop iteration matches ONE point10 (x,y) to ONE centroid10 (x,y)
loop_p:
        LDR R9, [R6], #4        @ 0x04969004, POST-INDEX load contents of points10[DATAPOINT][2], x-coordinate of some point
        LDR R10, [R3], #4       @ POST-INDEX load contents of centroids10[CENTROID][2], x-coordinate of some centroid
        SUB R9, R9, R10         @ 0x0049900A, x-coordinate of points10 MINUS x-coordinate of centroids10, store in R9
        MUL R9, R9, R9          @ 0x00009919, squared difference of x-coordinates

        LDR R10, [R6]           @ 0x0516A000, NORMAL load contents of points10[DATAPOINT][2], y-coordinate of SAME point above
        LDR R11, [R3], #4       @ POST-INDEX load contents of centroids10[CENTROID][2], y-coordinate of SAME centroid above
        SUB R10, R10, R11       @ y-coordinate of points10 MINUS y-coordinate of centroids10, store in R10
        MUL R10, R10, R10       @ squared difference of y-coordinates

        ADD R9, R9, R10         @ 0x0089900A, squared Euclidean distance between some data point and centroid

        ADD R8, R8, #1          @ increment counter
        CMP R8, #2              @ have we found out TWO distances between point and centroid0/centroid0?
        ITTEE NE
        MOVNE R4, R9            @ if only found out ONE distance so far, then record this distance in R4
        MOVNE R6, R0            @ if only found out ONE distance so far, then points10[CENTROID][2] needs to be reset
        MOVEQ R5, R9            @ if found out TWO distances already, record this distance in R5
        LDREQ R8, =0x0          @ if found out TWO distances already, reset the counter

        BNE loop_p              @ 0x18000044, if only ONE distance found so far, proceed to match current point to 2nd centroid
        BEQ loop_d              @ 0x08800000, else, proceed to next point
```

For each iteration of *loop_p*, we are calculating the squared euclidean distance between ONE *point10(x,y)* and ONE *centroid10(x,y)*. Since each *point10(x,y)* has two squared euclidean distances with respect to BOTH *centroid10(x,y)*, we must repeat *loop_p* twice for some *point10(x,y)* corresponding to TWO *centroid10(x,y)*.

```
@ each loop iteration fills in an entry in class[DATAPOINTS]
loop_d:
        MOV R3, R1              @ restore R3 with STARTING address of centroids10[CENTROID][2]
        ADD R0, #8              @ go to next points10 point
        MOV R6, R0              @ go to next points10 point

        MOV R9, #0            @ point will belong to centroid0
        MOV R10, #1           @ point will belong to centroid1

        CMP R4, R5              @ point-centroid0 vs point-centroid1, does R4 - R5
        ITE GE                  @ condition: R4 >= R5 (SIGNED)
        STRGE R10, [R2], #4     @ 0x0482A004, R4 >= R5, so point must belong to centroid1
        STRLT R9, [R2], #4      @ 0x04829004, R4 < R5, so point must belong to centroid0

        ADD R7, R7, #1              @ increment "i" variable
        CMP R7, DATAPOINT           @ i == DATAPOINT in for loop?

        BNE loop_p                     @ if not, still have more points to classify
        BEQ whichCentroidMorePoints    @ else, can proceed to second part
```

From *loop_p*, each *point10(x,y)* has two distances associated with it (corresponding to *centroid0* and *centroid1*). For each iteration of *loop_d*, we will be calculating which *centroid10* the *point10(x,y)* currently being considered should belong to.

This is done by taking the smaller of the two distances, and then updating the corresponding *class[DATAPOINTS]* entry to reflect the centroid that point should belong to.

Since there are *DATAPOINTS* number of *points10(x,y)*, *loop_d* has to run for *DATAPOINTS* number of times to successfully classify all *points10(x,y)* to their corresponding centroid.

```
whichCentroidMorePoints:
        MOV R2, R12         @ restore R2 to STARTING address of class[DATAPOINTS]
        LDR R3, =0x0        @ counter for number of points under centroid0
        LDR R4, =0x0        @ counter for number of points under centroid1
        LDR R5, =DATAPOINT  @ iterate through class[DATAPOINTS]

        B loop_c
```

From *loop_d*, we now have an array *class[DATAPOINTS]* containing the centroid that each *point10(x,y)* is associated with. Before iterating through *class[DATAPOINTS]*, we will need to do some initialization of registers, as seen in *whichCentroidMorePoints*.

```
@ iterate through class[DATAPOINT]
loop_c:
        CMP R5, #0          @ have we iterated through ENTIRE class[DATAPOINTS]?
        BEQ returnClass     @ if yes, return to C program

        LDR R6, [R2], #4    @ load element of class[DATAPOINTS] into R6

        CMP R6, #0          @ do R6 - 0
        ITE EQ              @ condition: R6 == 0?
        ADDEQ R3, #1        @ if yes, then increment counter for centroid0
        ADDNE R4, #1        @ if no, then increment counter for centroid1

        CMP R3, R4          @ points in centroid0 - points in centroid1
        ITE MI              @ condition: centroid0 has less points than centroid1
        MOVMI R7, #1        @ NEGATIVE: R7 stores centroid with larger amount of points (centroid1 here)
        MOVPL R7, #0        @ POSITIVE OR ZERO: R7 stores centroid with larger amount of points (centroid0 or centroid1 here)

        SUB R5, #1          @ decrement iteration counter through class[DATAPOINTS]
        B loop_c            @ loop back again


@ prepare value to return (class) to C program in R0
returnClass:
        MOV R0,R7     @ R7 contains the centroid number with the most points
```

In *loop_c*, we will be iterating through *class[DATAPOINTS]* to figure out which centroid has more *points10(x,y)* associated with it.

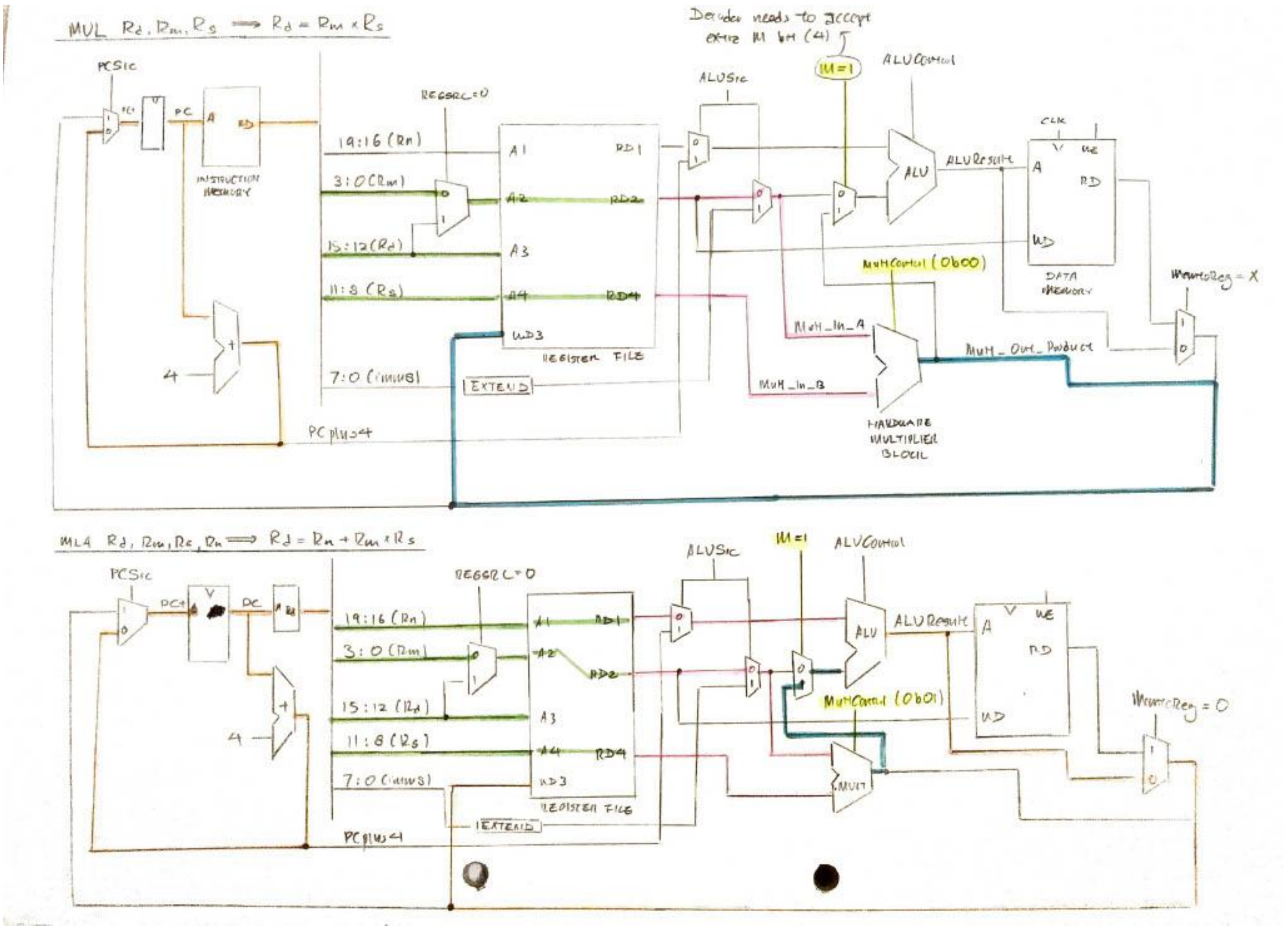
class[i] corresponds to the ith *points10(x,y)*, with it having either the value of 0 (point belongs to *centroid0*) or 1 (point belongs to *centroid1*).

Hence, for each *class[i]* element we read, we will increment the corresponding counter for centroid0 or centroid1, which is keeping track of how many points each centroid currently has associated with it.

Then, we will consider which counter is currently higher, and store the corresponding centroid into R7.

Once we are done iterating through *class[DATAPOINTS]*, R7 will contain the centroid number with the most points, which is returned to the C function.

Microarchitecture Modifications (for MUL/MLA)



In the Register File, we require an additional input *A4* which accepts *Rs* (bits 11:8 of **DP Register instruction**), and outputs *RD4* which contains the contents of register *Rs*.

Decoder takes in an additional input bit *M*, which is bit 4 of an **DP Register** instruction.

- *M* will be 1 for any **MUL**/**MLA** instruction
    - When *M* is 1, *RD2* will not flow to ALU. Instead, it will be redirected into MULT hardware.
- *M* will be 0 for any other **DP Register** instruction.

MultControl is a 4 bit control signal for a multiplexer, which comes from the *cmd* field of the **DP Register** instruction.

- if *M* is 1 and *cmd* is 0000, it is a **MUL** instruction
- if *M* is 1 and *cmd* is 0001, it is a **MLA** instruction
- hence *MultControl* is 0000 if (*op* == 0b0000) && (*I* == 0) && (*M* == 1) && (*cmd* == 0b0000)
- hence *MultControl* is 0001 if (*op* == 0b0000) && (*I* == 0) && (*M* == 1) && (*cmd* == 0b0001)

ALUControl will additionally need to account for the MLA scenario.

```
if (op == 0b0000) {
    // MUL/MLA or other DP Instructions

    if (M == 0b1 && I == 0b0) {
        // is either MUL or MLA instruction
        // ALUControl needs to do ADDITION for MLA
        // ALUControl is not used for MUL, so doesn't matter to set it for MUL
        ALUControl = 0b0100
    } else {
        // is other DP instructions, so ALUControl is whatever cmd is
        ALUControl = cmd
    }
} else {
    // is either Memory or Branch instructions
    // U bit determines if ALUControl is ADD or SUB

    if (U == 0b0) {
        ALUControl = 0b0100
    } else {
        ALUControl = 0b0010
    }
}

```