ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ (MICROLAB)

Bonus Εργασία στην HLS "**ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ VLSI "** του 8ου Εξαμήνου

Κωνσταντίνου Ιωάννου
ΑΜ: 03119840

(Ολόκληροι οι κώδικες και τα projects βρίσκονται στο zip αρχείο που θα παραδώσω)

Ζητούμενο-1

1) Αρχικά τροποποιούμε το δοθέν FIR φίλτρο στην HLS ώστε να λειτουργεί όπως αυτό που είχαμε φτιάξει σε VHDL στην εργαστηριακή άσκηση 4.

   Οι αλλαγές στον κώδικα τόσο στο header file όσο και στο κανονικό φαίνονται στην συνέχεια:

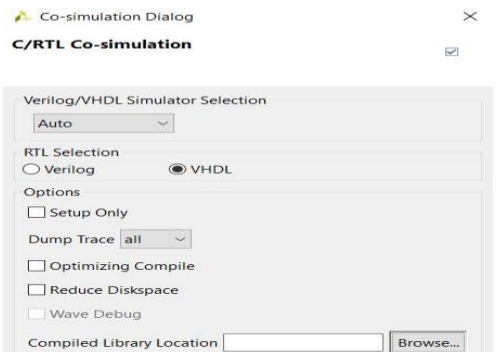
```c
#include <stdio.h>
#include <math.h>
#include "fir.h"

int main () {
  const int    SAMPLES=17;
  FILE         *fp;

  data_t signal, output;
 // coef_t taps[N] = {0,-10,-9,23,56,63,56,23,-9,-10,0,};
  coef_t taps[N] = {1,2,3,4,5,6,7,8};
  data_t signal_inputs[SAMPLES] = {40,248,245,124,204,36,107,234,202,245,0,0,0,0,0,0,0};
  int i, ramp_up;
  signal = 0;
  ramp_up = 1;

  fp=fopen("out.dat","w");
  for (i=0;i<SAMPLES;i++) {
    signal = signal_inputs[i];
    // Execute the function with latest input
    fir(&output,taps,signal);
    // Save the results.
    fprintf(fp,"%i %d %d\n",i,signal,output);
  }
  fclose(fp);
```

out.gold.dat | fir_test.c | fir.h | fir.c | fir_csim.log

```c
2 Vendor: Xilinx
46 #ifndef FIR_H_
47 #define FIR_H_
48 #define N   8

50 typedef int coef_t;
51 typedef int data_t;
52 typedef int acc_t;

54 void fir (
55   data_t *y,
56   coef_t c[N+1],
57   data_t x
58   );

60 #endif
```

Προφανώς αλλάζουμε και το αρχείο out.gold.dat σύμφωνα με τις τιμές εξόδου που περιμένουμε και τρέχοντας C simulation βλέπουμε ότι το φίλτρο μας λειτουργεί όπως περιμέναμε .

Για να μην φτιάχνω δικό μου testbench στο vivado (vhdl) κάνω το εξής και τρέχουμε RTL simulation .



out.gold.dat | fir_test.c | fir.h | fir.c | fir_csim.log

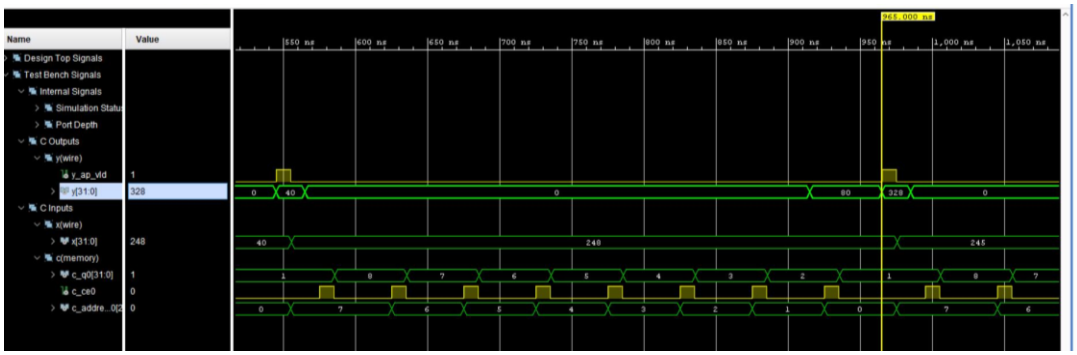
```
0 40 40
1 248 328
2 245 861
3 124 1518
4 204 2379
5 36 3276
6 107 4280
7 234 5518
8 202 6598
9 245 6011
10 0 5203
11 0 5239
12 0 4431
13 0 4931
14 0 4756
15 0 3331
16 0 1960
```

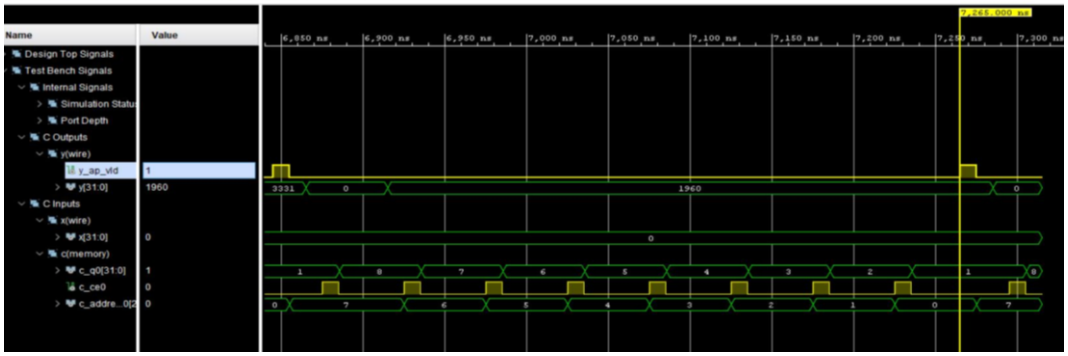
Console | Errors | Warnings | DRCs

Vivado HLS Console

```
   Generating csim.exe
Comparing against output data
*********************************************
PASS: The output matches the golden output!
*********************************************
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Στην συνέχεια με Open wave Viewer μπορούμε να δούμε την κυματομορφή του FIR φίλτρου για τιμές που δώσαμε στην HLS στην συνάρτηση main() . Σε screenshots παραθέτω την αρχή και το τέλος της κυματομορφής ενώ διαπίστωσα ότι οι ενδιάμεσες έξοδοι του φίλτρου είναι και αυτές σωστές.





Στην συγκεκριμένη περίπτωση στο RTL report έχουμε τα εξής αποτελέσματα

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
41	41	41	41	none

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	81
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	64	4
Multiplexer	-	-	-	116
Register	-	-	175	-
Total	0	3	239	201
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

2) Για να βελτιώσουμε τόσο το latency όσο και τα resources που χρησιμοποιούνται , θα αξιοποιήσουμε arbitrary precissiin types δηλαδή θα ορίσουμε συγκεκριμένο μέγεθος στους πίνακες εισόδου και εξόδου γιατί όπως γνωρίζουμε για τα συγκεκριμένα δεδομένα η έξοδος είναι πολύ μικρότερη από 32 bit που χρησιμοποιεί default το φίλτρο.

```
2⊕Vendor: Xilinx
46 #ifndef FIR_H_
47 #define FIR_H_
48 #define N   8
49 #include "ap_cint.h"
50 typedef uint8 coef_t;
51 typedef uint8 data_t;
52 typedef uint19 acc_t;
53
54
55
56 void fir (
57   acc_t *y,
58   coef_t c[N+1],
59   data_t x
60   );
61
62 #endif
```

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.702	1.25

Οπότε κάνοντας ξανά RTL synthesis report βλέπουμε ότι βελτιώνεται σημαντικά το latency:

Latency (clock cycles)

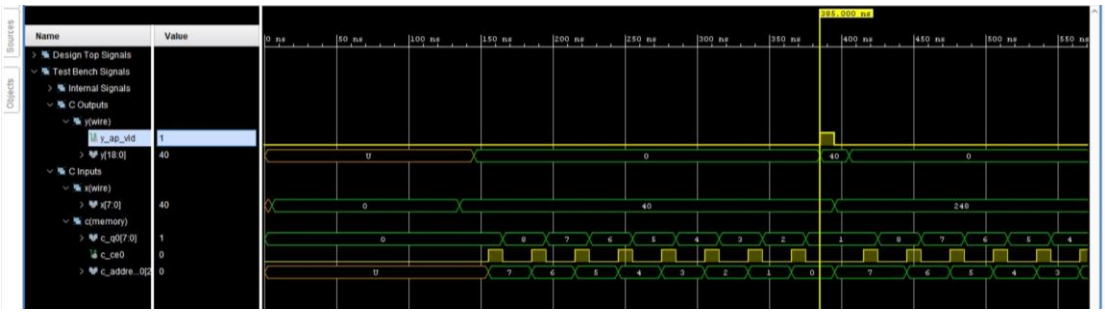
Summary

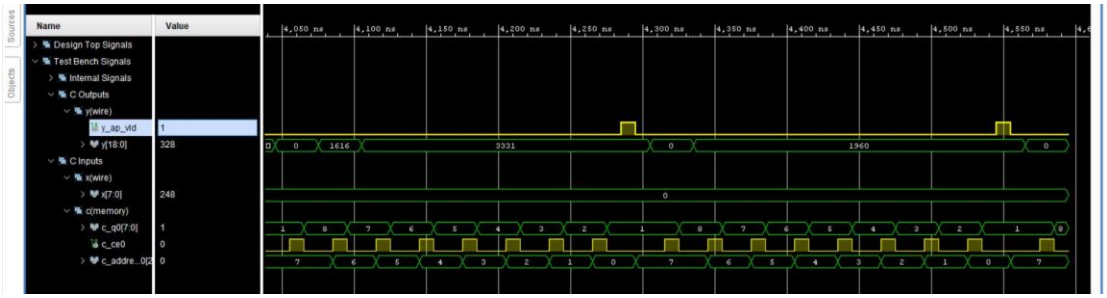
Latency		Interval		
min	max	min	max	Type
25	25	25	25	none

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	22
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	16	1
Multiplexer	-	-	-	105
Register	-	-	72	-
Total	0	1	88	128
Available	280	220	106400	53200
Utilization (%)	0	~0	~0	~0

Στην συνέχεια βλέπουμε πάλι την κυματομρφή του φίλτρου όμως τώρα η έξοδος και η είσοδος έχουν τον ελάχιστο αριθμό bits για να λειτουργήσει σωστά το φίλτρο.



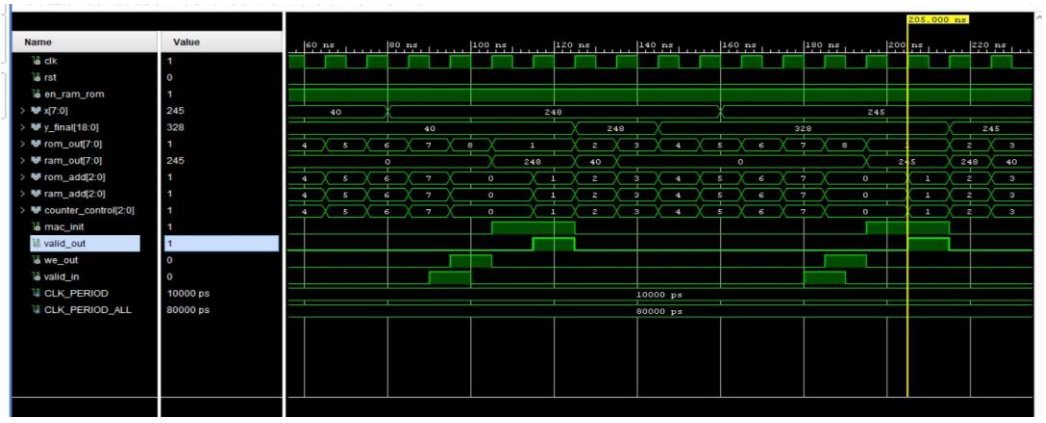


Χαρακτηριστικά	**1η Υλοποίηση**	**2η Υλοποίηση**
Estimated Clock	8.510	8.702
Latency(σε clocks)	41	25
Resource utilization	3 DSP 239FF 201 LUTs	1 DSP 88 FF 128 LUTs

Εξήγηση:Το estimated clock είναι μικρότερο στην πρώτη υλοποίηση καθώς χρησιμοποιεί περισσότερα FlipFlops και μειώνεται το critical path .Όμως το latency της πρώτης υλοποίησης είναι πολύ μεγαλύτερο (41*8.51 =348.9ns) από την δεύτερη υλοποίηση (25*8.70=217.55ns) το οποίο είναι λογικό καθώς μειώσαμε τα bit που χρησιμοποιούμε και οι πράξεις γίνονται πιο γρήγορα και προφανώς η 2η υλοποίηση καταναλώνει λιγότερους πόρους.

3) Τώρα θα συγκρίνουμε τα αποτελέσματα του FIR φίλτρου σε HLS σε σχέση με το FIR φίλτρο που σχεδιάσαμε σε vhdl στην εργαστηριακή άσκηση 4.

Προφανώς (βάζοντας ίδιο testbench σε vhdl) βλέπουμε ότι το φίλτρο μας έχει τις ίδιες εξόδου(valid_out =1) με το φίλτρο σε HLS.



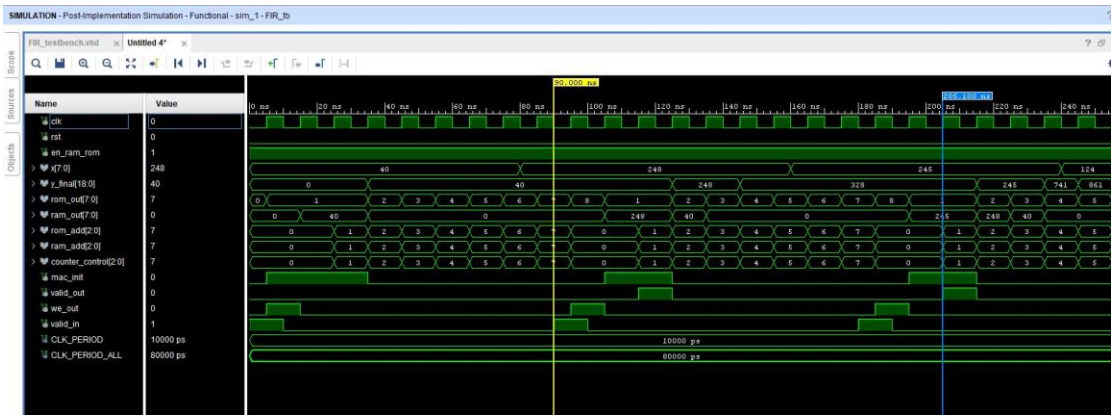
- Αυτοί είναι οι πόροι που καταναλώνει το FIR στη vhdl

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	88	17600	0.50
LUTRAM	1	6000	0.02
FF	168	35200	0.48
IO	59	100	59.00
BUFG	1	32	3.13

- Ενώ για το latency :



Latency = #clocks από την στιγμή που θα λάβει είσοδος μέχρι να υπολογίσει την έξοδο

Άρα Latency = 11 Clocks

- Για να βρούμε το Clock στο vivado κοιτάζουμε το μέγιστο critical path ,αρά

clock = 7.3ns

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source
Path 1	∞	10	7	18	rom/rom_out_reg[2]/C	macc/sum_reg[17]/D	7.319	3.134	4.185	∞	
Path 2	∞	10	7	18	rom/rom_out_reg[2]/C	macc/sum_reg[18]/D	7.235	3.050	4.185	∞	

Συμπεράσματα FIR σε HLS vs VHDL :

Χαρακτηριστικά	**VHDL**	**HLS**
Estimated Clock	7.3	8.702
Latency(σε clocks)	11	25
Resource utilization	1 BUFG 168 FF 88 LUTs 1 LUT RAM	1 DSP 88 FF 128 LUTs

Από το παραπάνω πίνακα συμπεραίνουμε ότι το κύκλωμα σε VHDL είναι πιο γρήγορο από αυτό σε HLS , συγκεκριμένα έχει και μικρότερο clock και πολύ μικρότερο latency = 11*7.3 =80.3ns ενώ σε HLS latency = 217ns .Αυτό συμβαίνει γιατί το FIR στην VHDL είναι πιο εξιδεικευμένο και σχεδιασμένο για αυτό το σκοπό ενώ στην HLS το φίλτρο λογικά παράγει πιο γενικό κώδικα καθώς βλέπουμε ότι χρησιμοποιεί και DSP οπότε θα μπορούμε αν θέλουμε να κάνουμε πράξεις ακόμη και με float.

Ζητούμενο-2

1) Αρχικά θα γράψουμε κώδικα ο οποίος υλοποιεί σε hls ένα svm classifier , θα χρειαστούμε testbench,header file και προφανώς source file για την top

```
////////////////////////////////////////////////////////////////////////////////
// here for "svmClassification.h"
#define Dsv 18
#define Nsv 1222
#define g 8
#define b 2.8180

#define MAX_ROWS 18 // Maximum number of rows in the CSV file
#define MAX_COLS 1222  // Maximum number of columns in the CSV file

// Define data types and constants
typedef double  input_data_type;
typedef double support_vector_type;
typedef double coefficient_type;
typedef int output_type;

//int svmClassification_top(
//  input_data_type x[Dsv],
//  coefficient_type coefficients[Nsv],
//  support_vector_type support_vectors[Dsv*Nsv],
//  output_type* output
//  );

int svmClassification_top(
  double x[Dsv],
  double coefficients[Nsv],
  double support_vectors[Dsv*Nsv]
 // output_type* output
  );
```

Εδώ ορίζουμε τι τιμές θα έχουν οι σταθερές καθώς και κάνουμε typedf κάποια νέα είδη μεταβλητών , ενώ ορίζουμε και τι ορίσματα θα περιμένει η top level συνάρτηση μας.

```
////////////////////////////////////////////////////////////////////////////////
#include "Classifier.h"
#include <math.h>
#include <stdio.h>


// SVM classification function
output_type svmClassification(input_data_type x[Dsv],coefficient_type coefficients[Nsv],support_vector_type support_vectors[Dsv*Nsv]) {
    // Compute decision value
    double sum =0;
    double sum_new=0;
    int column ;
    for (int i = 0; i < Nsv; i++) {
        // Compute dot product between input features and support vector
        input_data_type metro_2 = 0.0;
        column=i;
        support_vectors[0] = 0.452850; // something wrong with the  the [0][0]
        for (int s = 0; s < Dsv; s++) {

              double diff = x[s] - support_vectors[column + s*Nsv];
              metro_2 += diff * diff;
              //if(i==0) printf("The metro_2 is %f\n", metro_2);
             // if(i==0) printf("The support vectors  is %f\n", support_vectors[column + s*Nsv]);
        }
        sum_new = (coefficients[i]*exp(-g*metro_2) ) ;
        sum = sum + sum_new;
        //printf("The step %d  the sum_new is: %f\n",i,  sum_new);
          }
      sum =sum-b;

    printf("Totale SUM  is %f\n", sum);
    output_type output;
    if (sum > 0) {
        output = 1;  // Positive class label
    } else if( sum < 0) {
        output = -1;  // Negative class label
    }

    return output;
}

// Top-level function for HLS synthesis
int svmClassification_top(input_data_type x[Dsv],coefficient_type coefficients[Nsv],support_vector_type support_vectors[Dsv*Nsv]) {
    int output;
    output = svmClassification( x,coefficients, support_vectors);
    return output;
}
```

Εδώ βρίσκεται η βασική συνάρτηση που υλοποιεί την ταξινόμηση και κάνει τις κατάλληλες πράξεις ,δέχεται ως ορίσματα ένα διάνυσμα εισόδου x[18] ,τα 1222 coefficients και έναν “2d-array” support_vector[1222][18].

Στην εξωτερική loop υπολογίσουμε το άθροισμα της συνάρτησης ενώ στο εσωτερικό loop υπολογίζουμε το μέτρο των διανυσμάτων που ουσιαστικά αποτελεί το feature_vector.

Τέλος όταν υπολογίσουμε το συνολικό sum εξετάζουμε το πρόσημο του και το ταξινομούμε ανάλογα.

Στην συνέχεια παρουσιάζουμε τον κώδικα που υλοποιεί το testbench:

Είδα στην συνέχεια ότι υπάρχουν έτοιμες συναρτήσεις #include scv_file ,αλλά τώρα είναι αργά.

Η συνάρτηση extractNumber ουσιαστικά χωρίζει το string που έχουμε πάρει από το csv μετατρέπει τους αριθμούς από string σε double και τα αποθηκεύει σε έναν πίνακα. Την μεταβλητή line την χρησιμοποιούμε αν θέλουμε να αποθηκεύσουμε κάποια συγκεκριμένη τιμή του csv.

```
13
14 void extractNumbers( char buffer[], double numbers[], int maxNumbers,int line) { // line =0 take all the lines.
15     char* token = strtok(buffer, ",");  // Tokenize the string using comma as the delimiter
16     int count = 0;
17    // printf("char token is: %s\n",token);
18
19     while (token != NULL && count < maxNumbers) {
20         numbers[count + maxNumbers*line] =   atof(token);  // Convert the token to a double and store it in the numbers array
21         count++;
22         token = strtok(NULL, ",");  // Move to the next token
23     }
24 }
25
```

```
26 void readCSVFile(const char* filePath,double values[],int maxValues,int line) {
27     FILE* file = fopen(filePath, "r");
28     if (file == NULL) {
29         printf("Failed to open the file.\n");
30         return;
31     }
32
33     int count = 0;
34     if(line ==0){ // take all the lines of csv file
35     char buffer[1222*270];
36
37     while (fgets(buffer, 100*sizeof(buffer), file) != NULL) {
38       //  printf("%s", buffer);
39         extractNumbers(  buffer,  values, 1222,count);
40         count++;
41      }
42     }
43     else if (line > 0){ // take the line you choose from csv file
44         char buffer[18*100];
45         while (fgets(buffer, 100*sizeof(buffer), file) != NULL) {
46               //  printf("%s", buffer);
47                 count++;
48                 if(line == count ) {
49                 extractNumbers(  buffer,  values, 1222,0);
50                 break;
51                 }
52     }
53    }
54     else { // for the annotation ONLY
55         char buffer[1000*2];
56         while (fgets(buffer, sizeof(buffer), file) && count < 1000) {
57                 double value = atof(buffer);
58                 values[count++] = value;
59             }
60    }
61    // printf("counter is %d\n",count);
62
63     fclose(file);
64 }
```

Μέσω της συνάρτησης readCSVFile διαβάζουμε το scv αρχείο που θέλουμε και σε συνεργασία με την προηγούμενη συνάρτηση αποθηκεύουμε τελικά τις τιμές σε έναν πίνακα.

Οι τιμή του πίνακα char buffer είναι τέτοια ώστε να χωρέσουν τελικά όλοι οι χαρακτήρες που θέλουμε ακόμη και στην χειρότερη περίπτωση.

Το line ορίζει την λειτουργία της συνάρτησης δηλαδή για

Line = 0 διαβάζουμε και αποθηκεύουμε όλο το scv file στον πίνακα

Για Line = x >0 διαβάζουμε και αποθηκεύουμε μόνο την x-οστή γραμμή του πίνακα.

Τελικά η main() στο testbench φαίνεται παρακάτω:

```
int main() {
const int SAMPLES = 1000; // 1000 input x vectors
input_data_type x[Dsv];
//coefficient_type coefficients[Nsv];
//support_vector_type support_vectors[MAX_ROWS][MAX_COLS];
//support_vector_type support_vectors[MAX_ROWS*MAX_COLS];
output_type* output;
int temp_array[SAMPLES+10];
double hit =0;
double miss =0;
FILE    *fp;


fp=fopen("out.dat","w");
//// function    coefficients <- line[0]
    const char* filePath1 = "C:/CSV_VLSI/sv_coef.csv";
    const int maxValues1 = 1222;  // Maximum number of values to store
    double coefficients[maxValues1];
   readCSVFile(filePath1,coefficients,maxValues1,0);
//        for (int s = 0; s < maxValues1; s++) {


//// function  full  support_vecotrs[] <-scv all lines
    const char* filePath2 = "C:/CSV_VLSI/support_vectors.csv";
    const int maxValues2 = 1222;  // Maximum number of values to store
    double support_vectors[maxValues2*19];
     readCSVFile(filePath2,support_vectors,maxValues2,0);
//                 for (int s = 0  ; s < maxValues2*18; s++) {


 for(int n = 0 ; n < SAMPLES;n++) {

/// function full  X <-  line[n]
      const char* filePath3 = "C:/CSV_VLSI/testing_set.csv";
      const int maxValues3 = 18;  // Maximum number of values to store
      double x[maxValues3];
      readCSVFile(filePath3,x,maxValues3,n+1);

// svmClassification(x[Dsv],coefficients[Nsv],support_vectors[Nsv][Dsv]);
//  Save the results.
    temp_array[n] = svmClassification_top(x,coefficients,support_vectors);//output;
        printf("My output  is %d\n", temp_array[n]);
       // fprintf(fp,"%d\n",output);
   }

   fclose(fp);
   printf ("Comparing against output data \n");

   const char* filePath4 = "C:/CSV_VLSI/annotation.csv";
            const int maxValues4 = 1000;  // Maximum number of values to store
            double given_values[maxValues4];
           readCSVFile(filePath4,given_values,maxValues4,-1);


    for (int i =0 ; i <SAMPLES; i++) {
    //function1(given_value, "C:/CSV_VLSI/annotation.csv", i);
    if(given_values[i] == temp_array[i]) hit++;
    else miss++;
    }

   double acc = (hit/SAMPLES)*100;
   printf("The accuracy is %.2f%%\n", acc);


}
```

Αρχικά αποθηκεύουμε σε πίνακες όλες τιμές του coefficient και του support_vector καθώς θα χρειαστούμε όλες τιμές τους για κάθε είσοδο. Στην συνέχεια έχουμε 1000 testcases ,οπότε σε ένα loop κάθε φορά αποθηκεύουμε σε έναν πίνακα το διάνυσμα εισόδου που θέλουμε (γραμμή-γραμμή στο csv) και καλούμε την βασική συνάρτηση που κάνει την ταξινόμηση .

Αποθηκεύουμε σε έναν πίνακα τι τιμή επέστρεψε η συνάρτηση {-1,1} και όταν τελειώσουμε όλα τα testcases συγκρίνουμε τα αποτελέσματα μας , με τα σωστά που μα δόθηκαν στο csv file annotation.Έτσι έχουμε το ποσοστό επιτυχίας για να γνωρίζουμε πόσο αξιόπιστος είναι ο ταξινομητής μας.

Συγκεκριμένα έχουμε ποσοστό επιτυχίας : **99.5%**

```
Comparing against output data
The accuracy is 99.50%
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Συνεχίζοντας εκτελούμε RTL synthesis ->Report και έχουμε τα εξής αποτελέσματα:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.232	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
439926	439926	439926	439926	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	142
FIFO	-	-	-	-
Instance	-	40	2441	4795
Memory	-	-	-	-
Multiplexer	-	-	-	422
Register	-	-	644	-
Total	0	40	3085	5359
Available	280	220	106400	53200
Utilization (%)	0	18	2	10

2) Για να βελτιστοποιήσουμε την υλοποίηση του κώδικα στο fpga εφαρμόζουμε loop unrolling ,τόσο auto loop unrolling με συγκεκριμένες εντολές όσο και manual loop unrolling ξαναγράφοντας κώδικα όπου χρειάζεται.

- **Auto unrolling**:Η βελτιστοποίηση του κώδικα στο fpga μπορεί να γίνει με χρήση της directive Unroll .Αυτή χρησιμοποιείται για να επιστρέψει την αποτελεσματικότερη εκτέλεση ενός εσωτερικού loop μέσω unrolling, που επιτρέπει την παράλληλη εκτέλεση των επαναλήψεων του βρόχου.

Στο συγκεκριμένο παράδειγμα με την εντολή #pragma HLS UNROLL factor = 9 , κάνουμε τον κώδικα να εκτελείται παράλληλα για 9 επαναλήψεις του βρόγχου.

Στο εσωτερικό loop απλώς προσθέτουμε την εντολή:

```
        for (int s = 0; s < Dsv; s++) {
#pragma HLS UNROLL factor =9 //auto unroll
                double diff = x[s] - support_vectors[column + s*Nsv];
                metro_2 += diff * diff;
                //if(i==0) printf("The metro_2 is %f\n", metro_2);
               // if(i==0) printf("The support vectors  is %f\n", support_vecto
        }
        sum_new = (coefficients[i]*exp(-g*metro_2) ) ;
        sum = sum + sum_new;
        //printf("The step %d  the sum_new is: %f\n",i,  sum_new);
           }
      sum =sum-b;
```

- Στο **manual unrolling** προσπαθούμε εμείς να επιδιώξουμε να πετύχουμε το εσωτερικό loop να εκτελεί 9 επαναλήψεις παράλληλα άρα γενικά να έχουμε 2 επαναλήψεις του εσωτερικού loop που το καθένα θα υπολογίζει 9 διαφορετικά feuture_vectors(το μέτρο)

```
for (int i = 0; i < Nsv; i++) {
    // Compute dot product between input features and support vector
    input_data_type metro_2[9] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
    input_data_type total_metro_2 =0;
    double diff[9];
    column=i;
    support_vectors[0] = 0.452850; // something wrong with the  the [0][0]
    for (int s = 0; s < Dsv; s+=9) {
          //   #pragma HLS UNROLL factor=9  // auto unroll !

        //!!!!Manual loop unroling!!!!
           diff[0] = x[s+0] - support_vectors[column + (s+0)*Nsv];
           diff[1] = x[s+1] - support_vectors[column + (s+1)*Nsv];
           diff[2] = x[s+2] - support_vectors[column + (s+2)*Nsv];
           diff[3] = x[s+3] - support_vectors[column + (s+3)*Nsv];
           diff[4] = x[s+4] - support_vectors[column + (s+4)*Nsv];
           diff[5] = x[s+5] - support_vectors[column + (s+5)*Nsv];
           diff[6] = x[s+6] - support_vectors[column + (s+6)*Nsv];
           diff[7] = x[s+7] - support_vectors[column + (s+7)*Nsv];
           diff[8] = x[s+8] - support_vectors[column + (s+8)*Nsv];


           metro_2[0] = diff[0] * diff[0];
           metro_2[1] = diff[1] * diff[1];
           metro_2[2] = diff[2] * diff[2];
           metro_2[3] = diff[3] * diff[3];
           metro_2[4] = diff[4] * diff[4];
           metro_2[5] = diff[5] * diff[5];
           metro_2[6] = diff[6] * diff[6];
           metro_2[7] = diff[7] * diff[7];
           metro_2[8] = diff[8] * diff[8];

 total_metro_2 +=   metro_2[0] + metro_2[1] + metro_2[2] + metro_2[3] + metro_2[4] + metro_2[5] + metro_2[6] + metro_2[7] + metro_2[8];


    }
    sum_new = (coefficients[i]*exp(-g*total_metro_2) ) ;
    sum_new =sum_new -  b;
    sum = sum + sum_new;
    //printf("The step %d  the sum_new is: %f\n",i,  sum_new);
      }
```

Εξήγηση: Αντί να περιμένουμε να ολοκληρωθεί σε κάθε επανάληψη ο υπολογισμός του μέτρου (metro_2) τώρα υπολογίζουμε ταυτόχρονα τόσο την διαφορά των X και των support_vectors όσο και μέτρο αυτής της διαφοράς , καθώς χρησιμοποιούμε διαφορετικά X[i] και Support_vectors[i] αυτό δεν δημιουργεί κάποιο θέμα αλλά αντιθέτως επιταχύνει την διαδικασία (εφόσον έχουμε πόρους επιπλέον) και πιθανόν να γλυτώνουμε τις εξαρτήσεις .(+++)

➢ Αποτελέσματα και συγκρίσεις auto και manual loop unrolling:

Auto loop unrolling

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.410	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
185751	185751	185751	185751	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	49	3274	5729
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	3	-
Total	0	49	3277	5744
Available	280	220	106400	53200
Utilization (%)	0	22	3	10

Manual loop unrolling

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.382	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
188195	188195	188195	188195	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	55	4272	7783
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	3	-
Total	0	55	4275	7798
Available	280	220	106400	53200
Utilization (%)	0	25	4	14

Detail

Αρχικά είναι προφανές ότι με το unrolling αν και αυξήθηκε το estimated clock (απο 8 ns) και χρησιμοποιούνται παραπάνω πόροι (κυρίως DSP) το latency μειώθηκε σημαντικά από τα 440000 clocks μόλις στα περίπου 20000 clocks.

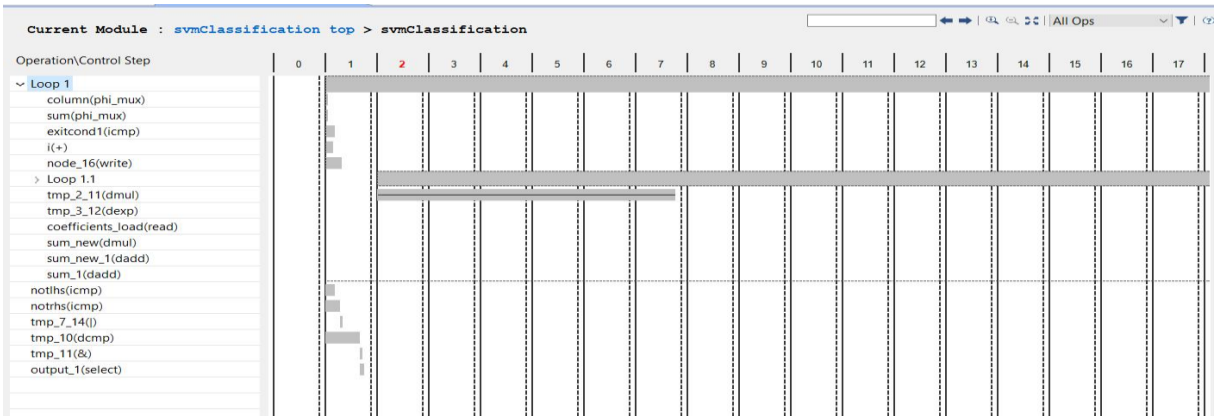
Τώρα παρατηρώντας τις δυο μεθόδους unrolling βλέπουμε ότι έχουν παρόμοια reports με λίγες διαφορές όπως

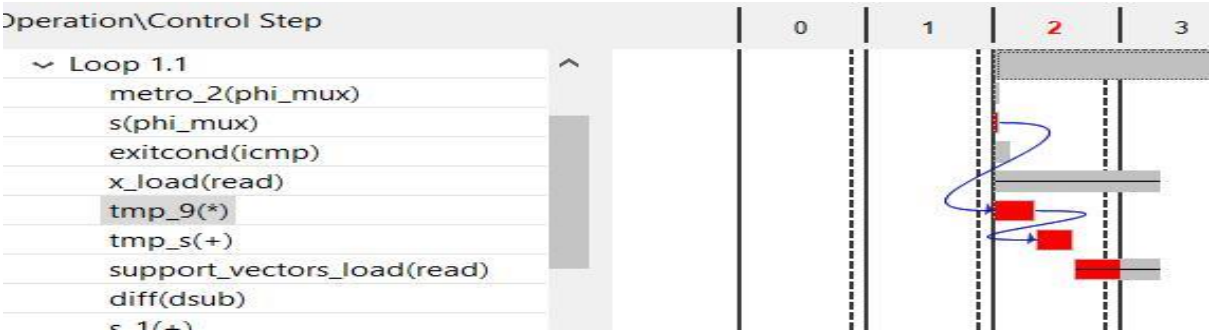
- Το estimated clock είναι ελάχιστα μεγαλύτερο στο manual unrolling
- Ενώ το latency είναι ελάχιστα μεγαλύτερο στο manual unrolling
- Επίσης το manual unrolling χρησιμοποιεί 6 επιπλέον DSP ,998 FF και 2000 LUTs.

Γενικά το auto unrolling είναι πολύ πιο γρήγορο και εύκολο αλλά με το manual unrolling ο προγραμματιστής καταλαβαίνει καλύτερα τι γίνεται στην σχεδίαση του και ενδεχομένως αναλόγως τον αλγόριθμο του προγράμματος και την εμπειρία του προγραμματιστή να προτιμάται το manual unrolling .Βέβαια το auto unrolling είναι σαφές ότι θα παρέχει μια βελτιστοποιημένη λύση.

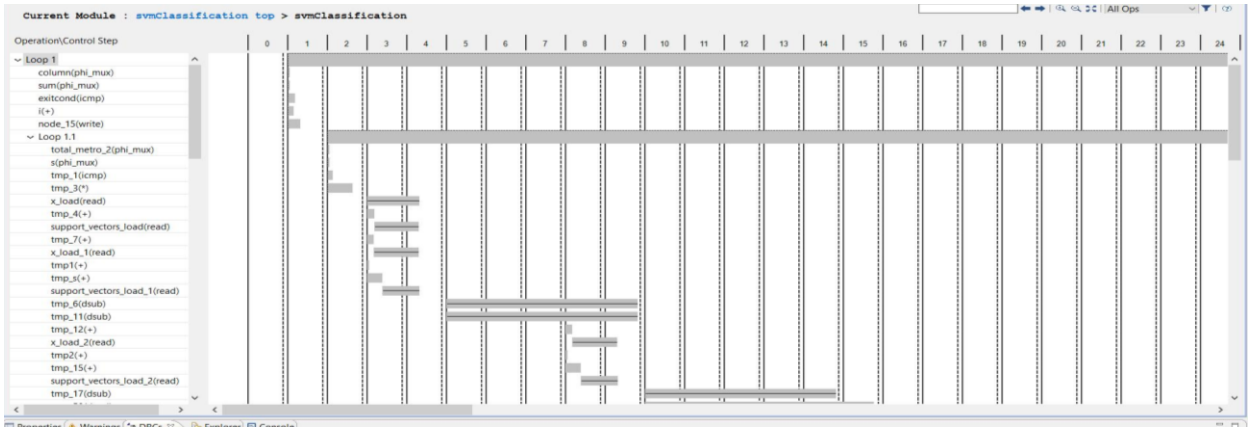
Παρατηρούμε ότι το Estimated Clock είναι μεγαλύτερο από το Target clock το οποίο δημιουργεί πρόβλημα στην υλοποίηση της σχεδίασης μας ,αλλά θα το διορθώσουμε στην συνέχεια με το ερώτημα 3 το array partition. Ιδανικά θέλουμε **estimated clock = targer clock -Ucertainty** και σίγουρα πρέπει **estimated clock < target clock**

Auto loop unrolling(Sheduling)





Manual loop unrolling(Sheduling)



Dependency: displays information related to iterations which have a loop carried dependency. For example, a read transaction could have a dependency on a prior write value.

Όπως βλέπουμε για να κάνουμε read το support_vector έχουμε εξάρτηση από το column (το i στο εξωτερικό loop) και το s στο εσωτερικό loop ,πιο συγκεκριμένα βλέπουμε με κόκκινα να έχουμε εξαρτήσεις(depentecies) στα (*),(+),read των support vectors στο Auto loop unrolling.

Ενώ στο manual unrolling δεν έχουμε εξαρτήσεις στο scheduling!

3)Τώρα θα προσθέσουμε στο πρόγραμμα μας (τόσο με auto όσο και με manual loop unroling) array partition ,δηλαδή να χωρίσουμε τους πίνακες μας σε μικρότερους πίνακες ή ακόμη και σε ξεχωριστά στοιχεία.

- Αυτό θα οδηγήσει το RTL να χρησιμοποιεί πολλαπλές μικρές μνήμες αντί μιας μεγάλης μνήμης.
- Θα αυξηθεί λοιπόν η απόδοση του πλήθους των read and writes και λοιπόν και η αποδοτικότητα.
- Αλλά θα προφανώς η σχεδίαση μας θα χρειάζεται περισσότερη μνήμη και καταχωρητές για να αξιοποιήσει.

*Σημειώνουμε ότι μετατρέπουμε τον support_vector σε 2D πίνακα ώστε να είναι πιο εύκολος στην διαχείριση του αλλά και για να είναι πιο κατανοητό το array partition. (τα specs απο το synthesisi report δεν αλλάζουν σε σχέση με τον μονοδιάστατο πίνακα παρά μόνο ότι με 2D χρησιμοποιούμε ένα λιγότερο DSP και ένα περισσότερο LUT)

Αλλαγές για να γίνει 2D array:

```
//make the 1D array to 2D array
          support_vectors[0] = 0.452850; // something wrong with the  the [0][0]
          support_vector_type array[Nsv][Dsv];
          for(int q = 0;q <Nsv; q++) {
              for(int m =0 ; m < Dsv;m++) {

                array[q][m]= support_vectors[Nsv*m +q];
              }
             }
```

```
for (int i = 0; i < Nsv; i++) {
    input_data_type metro_2 = 0.0;
    column=i;

    support_vectors[0][0] = 0.452850 ;

    for (int s = 0; s < Dsv; s++) {
          #pragma HLS UNROLL factor=9  // auto unroll !
          //double diff = x[s] - support_vectors[column + s*Nsv]; //for 1D
          double diff = x[s] - (support_vectors[i][s]);          // for 2D
           metro_2 += diff * diff;

    }
```

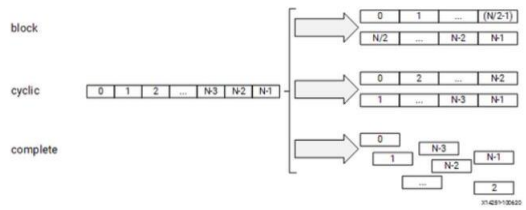
- Τώρα ας προσπαθήσουμε να κάνουμε array partition στο auto-unrolling.

Προσθέτουμε λοιπόν δύο επιπλέον εντολές στον κώδικα μας, με το σκεπτικό ότι θα θέλαμε να έχουμε άμεσα διαθέσιμα και τα 18 στοιχεία του x(input) αλλά και για κάθε εξωτερικό loop να έχουμε ξεχωριστά τα στοιχεία support_vecros[i][0-18].

```
// Array partions
 #pragma HLS array_partition variable = x  complete  dim =1
 #pragma HLS array_partition variable =support_vectors   complete  dim =2

for (int i = 0; i < Nsv; i++) {
    input_data_type metro_2 = 0.0;
    column=i;

    support_vectors[0][0] = 0.452850 ;

    for (int s = 0; s < Dsv; s++) {
          #pragma HLS UNROLL factor=9  // auto unroll !
          //double diff = x[s] - support_vectors[column + s*Nsv]; //for 1D
          double diff = x[s] - (support_vectors[i][s]);          // for 2D
           metro_2 += diff * diff;

    }
```

Θυμίζουμε το εξής:

Figure: Array Partitioning



- Τώρα για manual unrolling αρχικά το κάνουμε και αυτό ώστε suppoer_vector να είναι 2D για τους ίδιους λόγους με πριν

```
output_type svmClassification(input_data_type x[Dsv],coefficient_type coefficients[Nsv],s
    // Compute decision value
    double sum =0;
    double sum_new=0;
    int column ;
    // Array partions
    #pragma HLS array_partition variable = x  complete dim =1
    #pragma HLS array_partition variable =support_vectors   block factor = 18 dim =2

    for (int i = 0; i < Nsv; i++) {
        // Compute dot product between input features and support vector
        input_data_type metro_2[9] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
        input_data_type total_metro_2 =0;
        double diff[9];
        column=i;
        support_vectors[0][0] = 0.452850; // something wrong with the  the [0][0]
        for (int s = 0; s < Dsv; s+=9) {

            //!!!!Manual loop unroling!!!!
               diff[0] = x[s+0] - support_vectors[column][s];
               diff[1] = x[s+1] - support_vectors[column][s+1];
               diff[2] = x[s+2] - support_vectors[column][s+2];
               diff[3] = x[s+3] - support_vectors[column][s+3];
               diff[4] = x[s+4] - support_vectors[column][s+4];
               diff[5] = x[s+5] - support_vectors[column][s+5];
               diff[6] = x[s+6] - support_vectors[column][s+6];
               diff[7] = x[s+7] - support_vectors[column][s+7];
               diff[8] = x[s+8] - support_vectors[column][s+8];
```

Auto loop unrolling

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.348	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
191856	191856	191856	191856	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	64	9346	15936
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	3	-
Total	0	64	9349	15951
Available	280	220	106400	53200
Utilization (%)	0	29	8	29

Detail

Manual loop unrolling

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.348	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
191856	191856	191856	191856	none

Detail

Instance

Loop

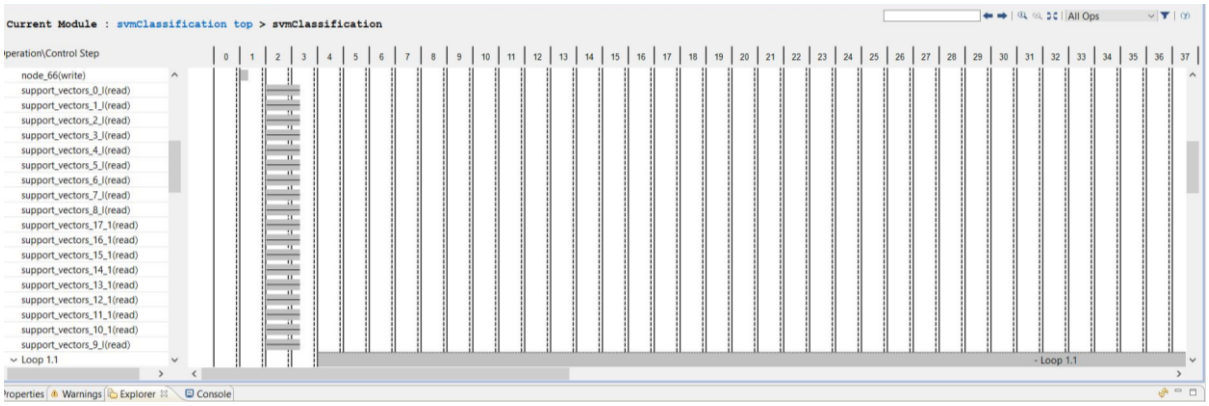
Utilization Estimates

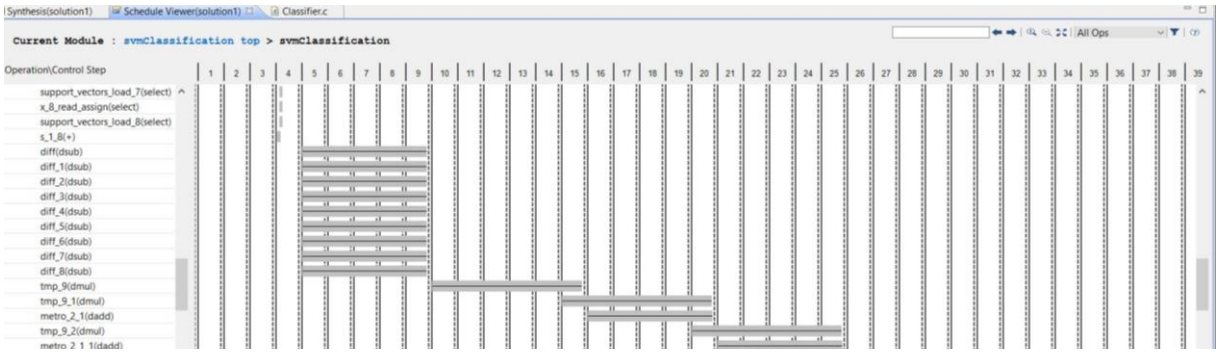
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	75	9727	16514
Memory	-	-	-	-
Multiplexer	-	-	-	15
Register	-	-	3	-
Total	0	75	9730	16529
Available	280	220	106400	53200
Utilization (%)	0	34	9	31

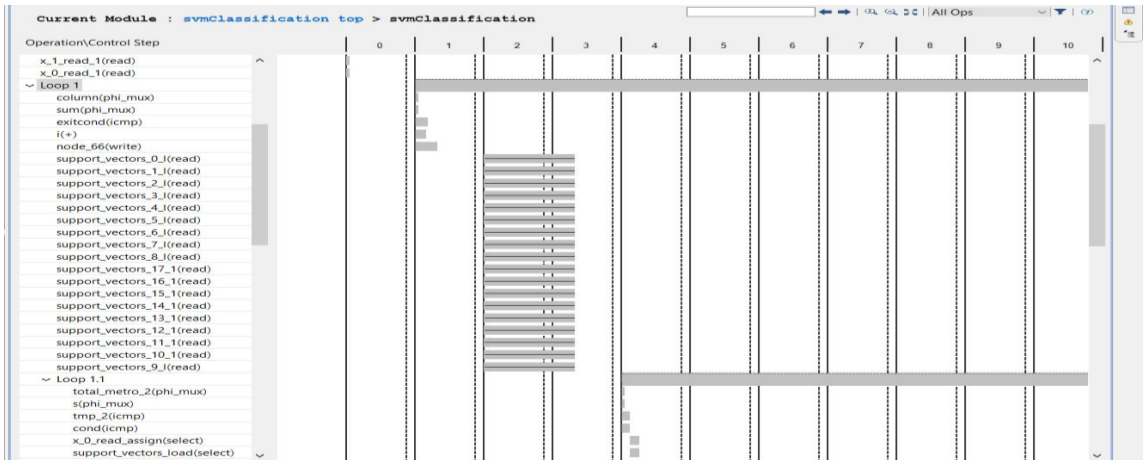
Παρατηρούμε λοιπόν ότι μειώνεται με partion array το estimated clock σε 9.348 ns ,αλλά το latency παραμένει σταθερό. Επίσης αυξάνονται οι πόροι(DSP,FF,LUTs) που χρησιμοποιεί η σχεδίαση μας καθώς όταν σπάμε τον πίνακα σε elements πολλαπλασιάζουμε την λογική .Το ρολόι είναι λογικό να μειώνεται καθώς με το «σπάσιμο» του πίνακα μπορούν να γίνουν περισσότερες πράξεις και υπολογισμοί παράλληλα. Βέβαια η σχεδίαση γίνεται πιο sensitive στο critical path , καθώς το εργαλείο κάθε φορά αποφασίσει τι θα γίνει παράλληλα με τι. Γενικά το array partition μείωσε αρκετό το clock estimated ώστε να είναι στα αποδεκτά όρια δηλαδή μικρότερο από το target clock.

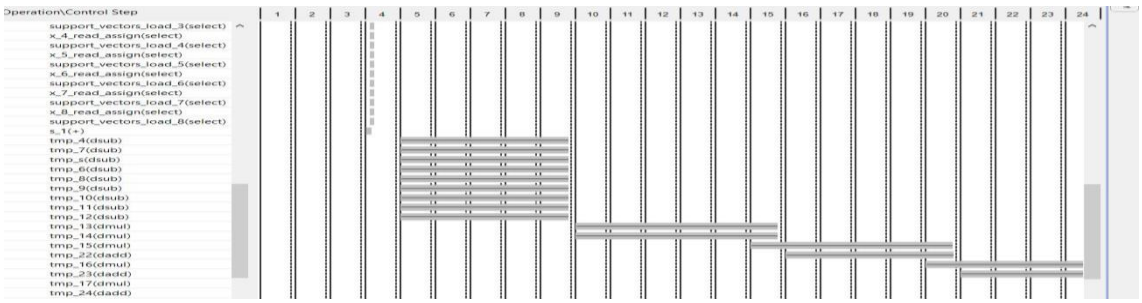
Auto loop unrolling(Sheduling)





Manual loop unrolling(Sheduling)





Παρατηρούμε λοιπόν ότι έχουμε πολλές εντολές(assemply) που γίνονται παράλληλα πλέον, όπως το read στα support vectors καθώς και υπολογισμός τις διαφοράς των διανυσμάτων (x-support_vector).Αυτό λοιπόν μειώνει το critical path και αυξάνει τους πόρους ώστε να γίνονται αυτές οι διαδικασίες παράλληλα.

Διαφορά των δύο Sheduling : (+++)

4) Στην συνέχεια θα χρησιμοποιήσουμε 2 instances για optimized Svm_classifier που θα εκτελούνται παράλληλα.

   Κώδικας για 2 instances:

   Τροποποιούμε την βασική συνάρτηση ώστε να κάνει τα μισές επαναλήψεις πλέον στο εξωτερικό loop ώστε ώστε να σπάσουμε δηλαδή το άθροισμα σε δύο επιμέρους αθροίσματα που εκτελούνται παράλληλα. Επίσης να επιστρέφει πλέον το μέχρι εκείνη την στιγμή άθροισμα η συνάρτηση και έχει το πρόσημο του αθροίσματος καθώς δεν έχει ολοκληρωθεί παρά το μισό άθροισμα.

```
// SVM classification function
double svmClassification1(input_data_type x[Dsv],coefficient_type coefficients[Nsv],support_vector_type support_vectors[Nsv][Dsv]) { //support_vectors[Dsv*Nsv]
    // Compute decision value
    double sum =0;
    double sum_new=0;
    int column ;



    // Array partions
     #pragma HLS array_partition variable = x  complete  dim =1
     #pragma HLS array_partition variable =support_vectors   complete  dim =2

    for (int i = 0; i < Nsv/2; i++) {
        input_data_type metro_2 = 0.0;
        column=i;

       // support_vectors[0][0] = 0.452850 ;

        for (int s = 0; s < Dsv; s++) {
             #pragma HLS UNROLL factor=9  // auto unroll !
             //double diff = x[s] - support_vectors[column + s*Nsv]; //for 1D
             double diff = x[s] - (support_vectors[i][s]);          // for 2D
              metro_2 += diff * diff;

        }
        sum_new = (coefficients[i]*exp(-g*metro_2) ) ; // printf("function 1: cof[%d] = %f\n",i,coefficients[i] );
        sum = sum + sum_new;

           }
        sum =sum -  b;


    return sum;
}
```

Επίσης τροποποιούμε την Top_Hls function ώστε αρχικά να χωρίζουμε τους πίνακες coeff[] και support_vector[][] στην μέση έτσι ώστε να χρησιμοποιήσουμε τον καθένα ξεχωριστά στα επιμέρους αθροίσματα.

Στην συνέχεια με την εντολή #pragma HlS dataflow ,επιτρέπουμε στην top level function να εκτελεί τις συναρτήσεις της (επιμέρους αθροίσματα) παράλληλα. Προφανώς καλούμε δυο φορές την συνάρτηση που υλοποιεί το άθροισμα αλλά με διαφορετικούς πίνακες για ορίσματα εκτός από αυτά τις εισόδου , προσθέτουμε λοιπόν αυτά τα επιμέρους αθροίσματα και τότε κοιτάζουμε το πρόσημο του συνολικού αθροίσματος.

```
// Top-level function for HLS synthesis
int svmClassification_top(input_data_type x[Dsv],coefficient_type coefficients[Nsv],support_vector_type support_vectors[Nsv][Dsv]) { //support_vectors[Dsv*Nsv]

    double sum1,sum2,sum;
    int i;
    coefficient_type coefficients1[Nsv/2],coefficients2[Nsv/2];
    support_vector_type support_vectors1[Nsv/2][Dsv],support_vectors2[Nsv/2][Dsv];

    for( i=0; i< Nsv/2; i++ )  { // split  coefficients[Nsv]
        coefficients1[i] = coefficients[i]; // printf("cof1[%d] = %f\n",i,coefficients1[i] );
        coefficients2[i] = coefficients[i +Nsv/2]; // printf("cof2[%d] = %f\n",i,coefficients2[i] );
        }


    for( i =0; i< Nsv/2;i++) { //// split  support_vectors[Nsv][Dsv]
        for(int j=0;j<Dsv;j++){
            support_vectors1[i][j] = support_vectors[i][j]; // if(i== (Nsv/2)-1)  printf("sv1[%d][%d] = %f\n",i,j,support_vectors1[i][j] );
            support_vectors2[i][j] = support_vectors[i+Nsv/2][j];  //if(i==0)  printf("sv2[%d][%d] = %f\n",i,j,support_vectors2[i][j] );
        }
    }

    #pragma HLS dataflow  // Add dataflow directive here for parallel execution
    sum1 = svmClassification1( x,coefficients1, support_vectors1);
    sum2 = svmClassification2( x,coefficients2, support_vectors2);
    sum =sum1+sum2;

        output_type output;
        if (sum > 0) {
            output = 1;  // Positive class label
        } else if( sum < 0) {
            output = -1;  // Negative class label
        }
    return output;
}
```

Για 1 instance:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.348	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
190634	190634	190634	190634	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	10	92
Instance	-	64	9352	15959
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	64	9362	16053
Available	280	220	106400	53200
Utilization (%)	0	29	8	30

Για 2 instance:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.348	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
118543	118543	95318	95318	dataflow

Detail

Instance

Loop

Utilization Estimates

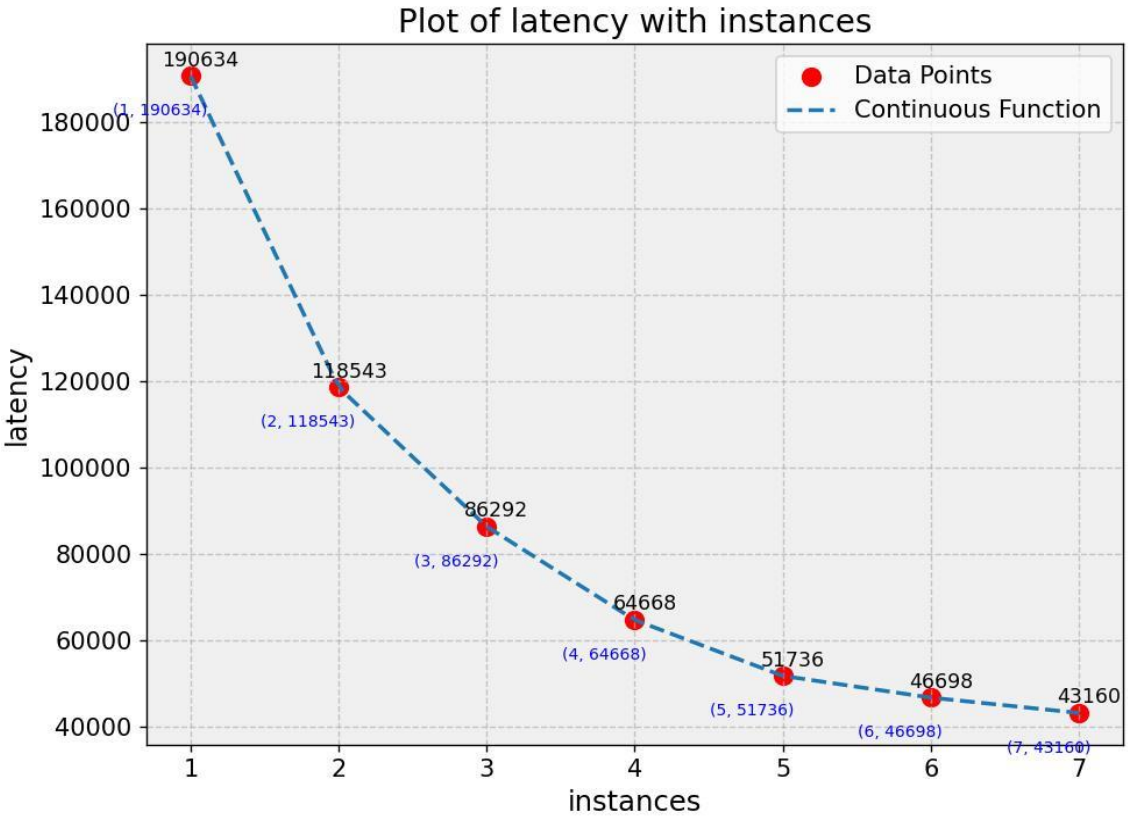
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	208
FIFO	0	-	285	4272
Instance	-	131	21839	33835
Memory	152	-	0	0
Multiplexer	-	-	-	396
Register	-	-	47	-
Total	152	131	22171	38711
Available	280	220	106400	53200
Utilization (%)	54	59	20	72

Είναι προφανές ότι το latency μειώνεται σημαντικά σχεδόν υποδιπλασιάζεται για instance 2 καθώς πλέον εκτελούμε παράλληλα , ξεχωριστά και ανεξάρτητα τα δύο αθροίσματα δηλαδή σαν να υπολογίσαμε εξαρχής ένα άθροισμα με 611 επαναλήψεις αντί για 1222.
Για να το πετύχουμε βέβαια αυτό χρησιμοποιούμε αρκετούς παραπάνω πόρους και Block_rams ,τα διπλάσια DSP ,FFs και Luts.Επομένως αν έχουμε

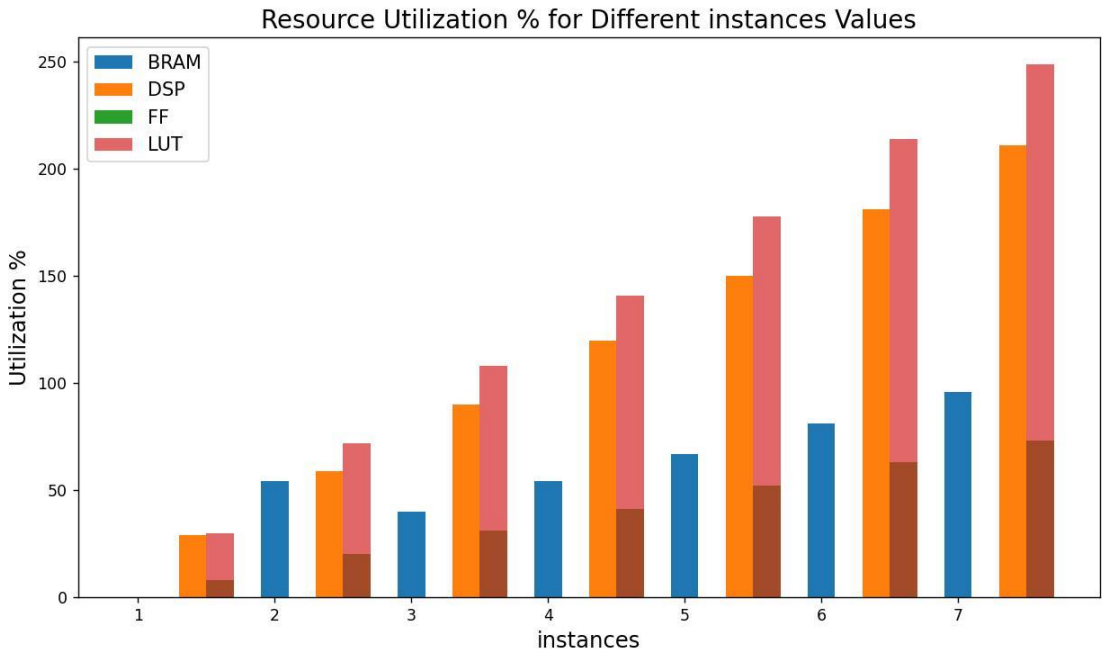
έναν επεξεργαστή αρκετά δυνατό που μένει αχρησιμοποίητος αξίζει να αυξήσουμε τα instance και να χωρίσουμε το πρόβλημα μας σε μικρότερα επιμέρους προβλήματα που εκτελούνται παράλληλα.

Με την ίδια λογική δοκιμάζουμε και για περισσότερα instances και σημειώνοντας τις τιμές latency αλλά και τους πόρους που χρησιμοποιεί η κάθε σχεδίαση έχουμε τα παρακάτω διαγράμματα.(με pyhton)



Παρατηρούμε ότι καθώς αυξάνονται τα instances και γίνονται περισσότερα μικρότερα αθροίσματα παράλληλα μειώνεται όπως είναι λογικό το latency , όμως αρχικά ο ρυθμός μείωσης πέφτει απότομα , επομένως θα πρέπει να σκεφτούμε το trade-off ,δηλαδή αν αξίζει να καταναλώσουμε επιπλέον πόρους ώστε να μειώσουμε το latency.

Ας δούμε λοιπόν ένα bar plot των πόρων που καταναλώνονται καθώς αυξάνονται τα instances:



Προκύπτει λοιπόν ότι έχουμε με την αύξηση των instances σημαντική αύξηση των DSP ώστε να γίνονται οι πράξεις στα επιμέρους αθροίσματα , των LUTs αλλά και έως ένα βαθμό των Block ram.

Είναι λοιπόν σαφές ότι δεν αξίζει να έχουμε πολλά instances γιατί ενώ η κατανάλωση πόρων αυξάνεται σχεδόν γραμμικά η μείωση του latency μειώνεται σχεδόν αντιστρόφως εκθετικά. Αξίζει λοιπόν στην συγκεκριμένη σχεδίαση να χρησιμοποιήσουμε το πολύ εως 4 instances με το μεγαλύτερο κέρδος να την έχουμε στα δυο instances.