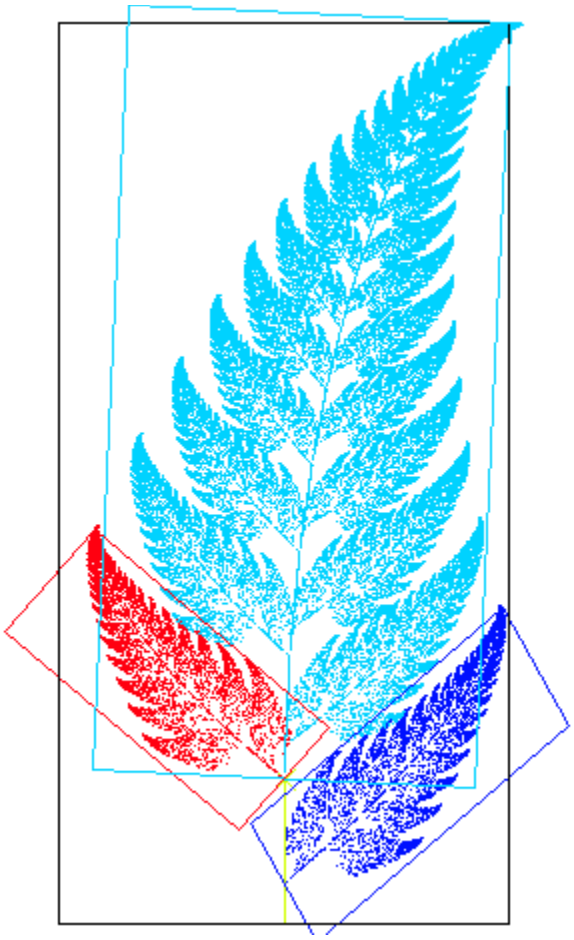
ΣΧΕΔΙΑΣΜΟΣ ΕΝΣΩΜΑΤΩΝ ΣΥΣΤΗΜΑΤΩΝ

Δεύτερο μέρος της εργασίας 2020 -2021

Εφαρμογή του συσχετισμένου μετασχηματισμού στην επεξεργασία εικόνας

affine transformation in image processing



Υπεύθυνος καθηγητής

Συρακούλης Γεώργιος

Ομάδα 34

Καλτάκης Αναστάσιος 57271

Λιάπης Σάββας 57403

Εισαγωγή

Σκοπός της εργασίας είναι, αφού υλοποιηθεί σε γλώσσα C ο αλγόριθμος σε c εφαρμογής του συσχετισμένου μετασχηματισμού στην επεξεργασία εικόνας και βελτιστοποιηθεί με τεχνικές βελτιστοποίησης βρόγχων, να εφαρμοστεί η μεθοδολογία βελτιστοποίησης μεταφοράς και αποθήκευσης δεδομένων στον αλγόριθμο αυτό.

Κατά την βελτιστοποίηση θα εξεταστεί μέσω του ARMulator του επεξεργαστή ARM τι αποτελέσματα έχει η εφαρμογή μετασχηματισμών βρόγχων για την κανονικοποίηση της δομής του αλγορίθμου στον πίνακα δεδομένων και στον αριθμό προσπελάσεων που γίνονται σε αυτόν, καθώς και στην ταχύτητα εκτέλεσης του αλγορίθμου.

Στην ουσία θα εξετάσουμε διάφορες ιεραρχίες μνήμης και θα ορίσουμε ποια από αυτές ταιριάζει καλύτερα για τον κώδικά μας.

Θεωρητική Βάση

Πριν δούμε πώς γίνεται να εκμεταλλευτούμε την ιεραρχία μνήμης είναι σκόπιμο να αναλυθούν τα διαφορετικά είδη μνημών που χρησιμοποιούνται κατά την κατασκευή συστημάτων παρόμοιων με αυτά που προσομοιώνονται.

- **Read Only Memory (ROM):** είναι προγραμματισμένες εξ αρχής με δεδομένα που δεν είναι δυνατόν να επεξεργαστούν. Η ROM διατηρεί τα δεδομένα της και χωρίς την ύπαρξη ρεύματος και για αυτό σε αυτή αποθηκεύονται δεδομένα που είναι απαραίτητα για την σωστή εκκίνηση του προγράμματος.
- **Random Access Memory (RAM):** σε αυτές υπάρχει δυνατότητα προσπέλασης αλλά και γραφής με χαρακτηριστικό την τυχαία σειρά προσπέλασης των στοιχείων της. Χωρίζονται σε 2 κατηγορίες τις SRAM (στατική) και DRAM (δυναμική) με κύριες διαφορές την κατανάλωση ισχύος και χρόνος προσπέλασης και δυνατότητες αποθήκευσης (μικρότερα στην SRAM) και το κόστος παραγωγής (μικρότερο στην DRAM).

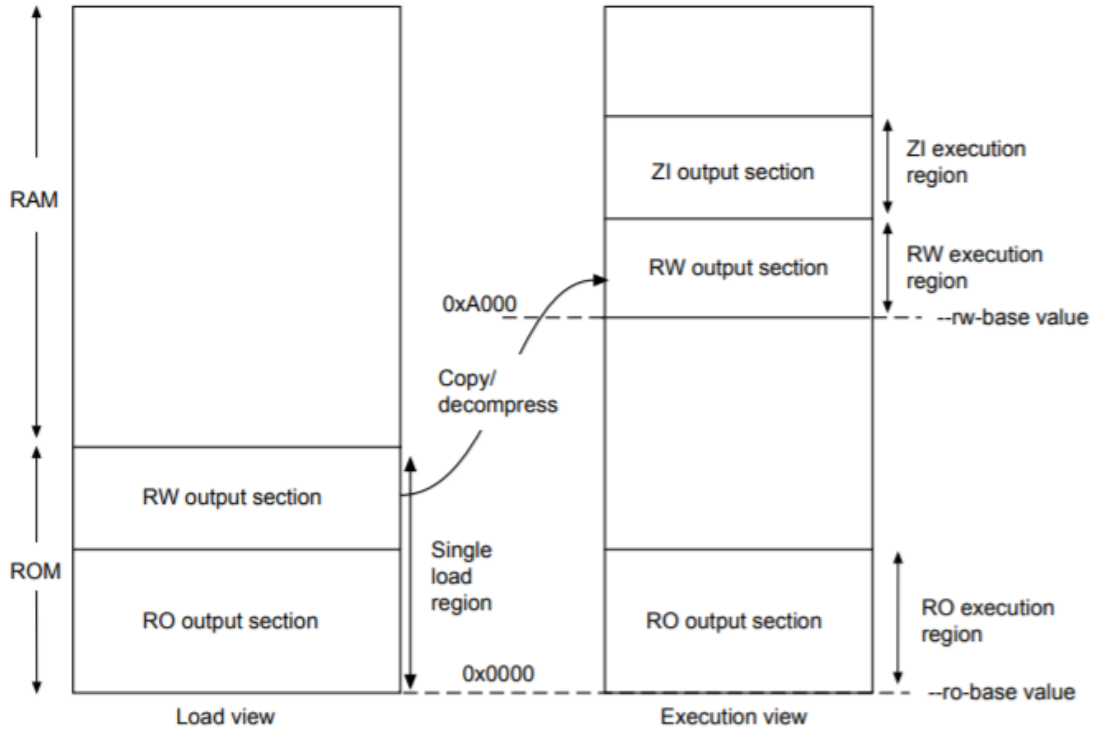
Εμείς ανάλογα με τους πόρους που χρειαζόμαστε και μπορούμε να διαθέσουμε ορίζουμε την ROM και RAM στην ιεραρχία μνήμης στο αρχείο memory map με τον εξής τρόπο :

<start address> <size> <name> <#bytes> <R or RW> <read times> <write times>, όπου:

- **<start address> :** η διεύθυνση που ξεκινάει η περιοχή μνήμης σε δεκαεξαδική μορφή.
- **<size>:** το μέγεθος της περιοχής μνήμης
- **<name>:** το όνομα της περιοχής μνήμης (συνήθως προσδιορίζει το είδος της μνήμης)
- **<#bytes>:** το εύρος του διαύλου σε bytes.
- **<R or RW>:** δικαιώματα (ανάγνωση μόνο ή και γραφή).
- **<read times> ,<write times> :** ο λόγος Ncycles/Scycles κατά την ανάγνωση και την γραφή αντίστοιχα. Για την εργαστηριακή άσκηση αυτοί θα είναι 1/1.

Πέρα από αυτό, είναι αναγκαίο να ορισθεί που θα τοποθετηθούν τα βασικά δεδομένα (ο κώδικας, οι σταθερές, και οδηγίες για το που θα αρχικοποιηθούν τα data στην μνήμη). Ειδικότερα, υπάρχουν 2 όψεις της μνήμης που πρέπει να ορισθούν.

- Load View: έχει να κάνει με το τι γίνεται στην μνήμη on reset
- Execution View: έχει να κάνει με το που βρίσκονται τα δεδομένα κατά την εκτέλεση του προγράμματος.



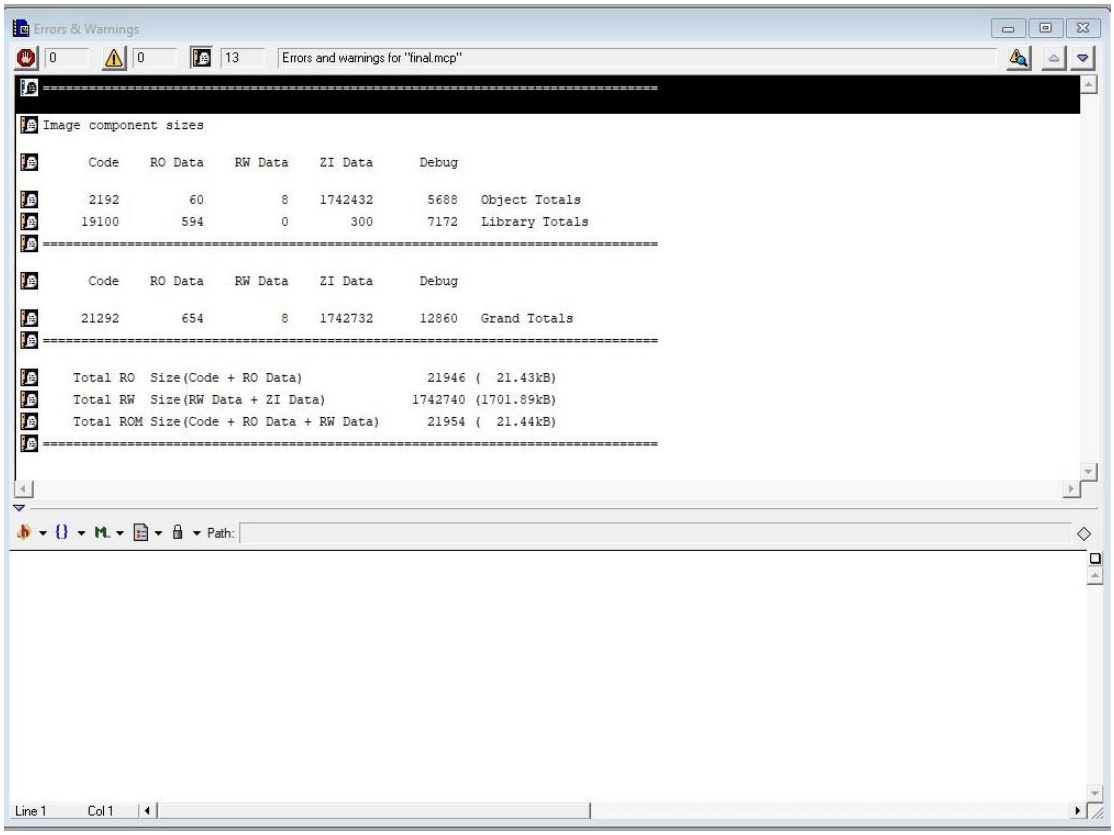
Όλα αυτά ορίζονται από το αρχείο scatter στο οποίο ορίζουμε το load view με την εξωτερική αναφορά (Συνήθως ROM) και execution view με τις εσωτερικές αναφορές.

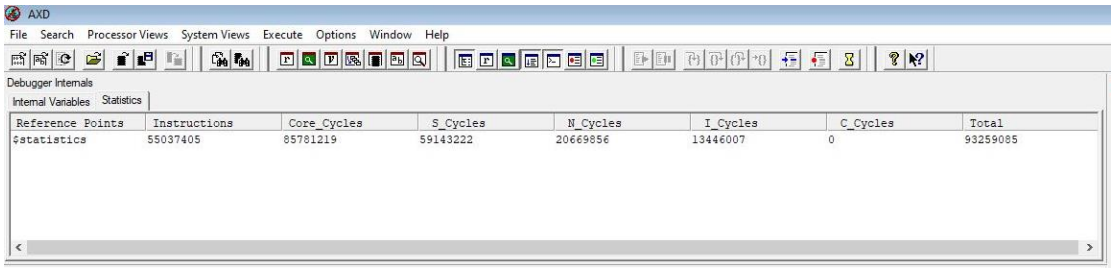
Επισημαίνεται ότι το αρχείο scatter πρέπει να είναι σε συμφωνία με το αρχείο memory map.

Ένα ακόμα αρχείο που θα χρησιμοποιηθεί είναι το αρχείο stack το οποίο αντιπροσωπεύει ουσιαστικά μία δομή stack-heap στην οποία «στοιβάζονται» όλες οι τοπικές μεταβλητές (αυτές που δηλώνονται στην main). Εμείς στην παρούσα εργασία θα χρησιμοπιήσουμε μόνο την δομή stack και όχι την heap, για την στοίβαξη των local μεταβλητών μας.

Υλοποίηση Πειράματος

Για την διεκπεραίωση της άσκησης θα χρησιμοποιηθεί ένας συνδυασμός των βελτιστοποιήσεων πάνω στον αλγόριθμο που συντάχθηκε στην προηγούμενη άσκηση. Με άλλα λόγια, είναι βελτιστοποιημένος με εφαρμογή loop unrolling, loop fussion, αρχικοποίηση κάποιων μεταβλητών σαν local αντί για global. Οπότε ξεκινάμε με σημείο αναφοράς τα εξής στοιχεία και προσπαθούμε να βελτιστοποιήσουμε τα παρακάτω :





Βελτιστοποίηση Μεταφοράς και Αποθήκευσης Δεδομένων

1. Έλεγχος με κατανομή δεδομένων σε ROM και RAM

ROM: 0x5800	RAM: 0x200000

Αρχικά πριν ξεκινήσουμε να παρουσιάζουμε τις διάφορες ιεραρχίες μνήμης του ενσωματομένου μας, πρέπει να εξηγήσουμε πως καταλήξαμε και με πια κριτήρια στην επιλογή των μεγεθών των μνημών.

Η μνήμη ROM επιλέχθηκε με κριτήριο την read only λειτουργία της, με σκοπό να υπάρχουν σε αυτή όλα τα απαραίτητα δεδομένα για την εκκίνηση του ενσωματομένου μας αλλά και την αποθήκευση του πηγαίου κώδικα.

Έτσι απο το section Total RO Data του CodeWarrior βλέπουμε ότι αυτά τα δεδομένα καταλαμβάνουν χώρο ίσο με 21.44 kb. Άρα μια μνήμη ROM των 22 kb θα μας είναι αρκετή.Για να μετατρέψουμε αυτήν την δεκαδική (decimal) χωρητικότητα σε δεκαεξαδική (hex) ακολουθούμε την εξής διαδικασία:

$$22 * 1024 = 22528(decimal) \xrightarrow{hex} 0x5800$$

Αντίστοιχα για την επιλογή της RAM (SRAM) πρέπει να δούμε πόσο χώρο καταλαμβάνουν τα RW και τα ZI Data μας. Αυτό φαίνεται απο το section Total RW Size του CodeWarrior, που είναι ίσα με 1701.89 kb.Για να ήμαστε σίγουροι ότι η μνήμη μας δεν θα γεμίσει στο 100% κατά τη λειτουργία του ενσωματομένου θα δώσουμε ένα 20% παραπάνω χωρητικότητα στην μνήμη μας.Έτσι θα έχουμε :

$$1701.89 + 20\% * 1701.89 = 1701.89 + 340.378 = 2042.268\ kb$$

Με μια στρογγυλοποίηση θα χρησιμοποιήσουμε μια μνήμη ram μεγέθος 2 mb δηλαδή 2048 kb.Για την μετατροπή στο δεκαεξαδικό:

$$2048 * 1024 = 2097152(decimal) \xrightarrow{hex} 0x200000$$

Και έτσι προκύψαν οι αναγραφόμενες χωρητικότητες.

Στην συνέχεια θα αναλύσουμε το σκεπτικό αλλά και θα συγκρίνουμε τα αποτέλεσματα πίσω από την κάθε μια ιεραρχία που χρησιμοποιήσαμε, και στο τέλος με βάση το performance αλλά και το πιθανό κόστος που μπορεί να έχει η κάθε μνήμη θα αποφασίσουμε για το ποια είναι η καλύτερη για την δουλειά μας.

Αρχεία

stack.c:

```
#include <rt_misc.h>

 __value_in_regs struct __initial_stackheap __user_initial_stackheap(
 unsigned R0, unsigned SP, unsigned R2, unsigned SL){

 struct __initial_stackheap config;

 //config.heap_limit  = 0x00724B40;

 //config.stack_limit = 0x00004000;
 //config.stack_limit = 0x00100000;

/* works */
config.heap_base   = 0x5900;
config.stack_base  = 0x200000;


/* factory defaults */
 //config.heap_base   = 0x00060000;
 //config.stack_base  = 0x00080000;
 return config;}
```

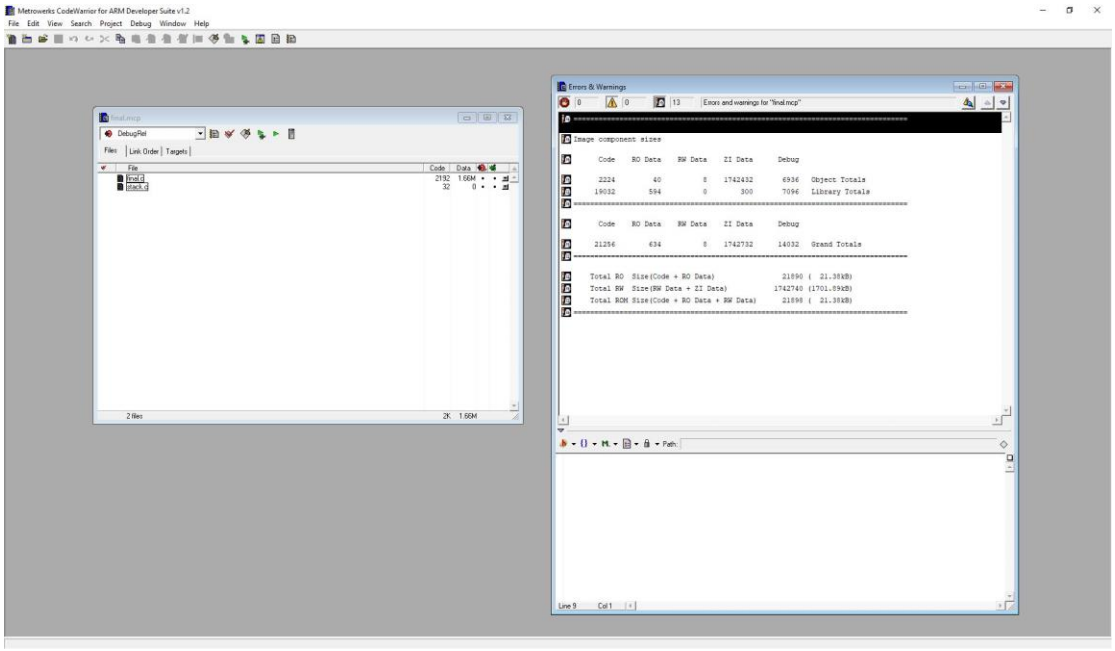
memory.map:

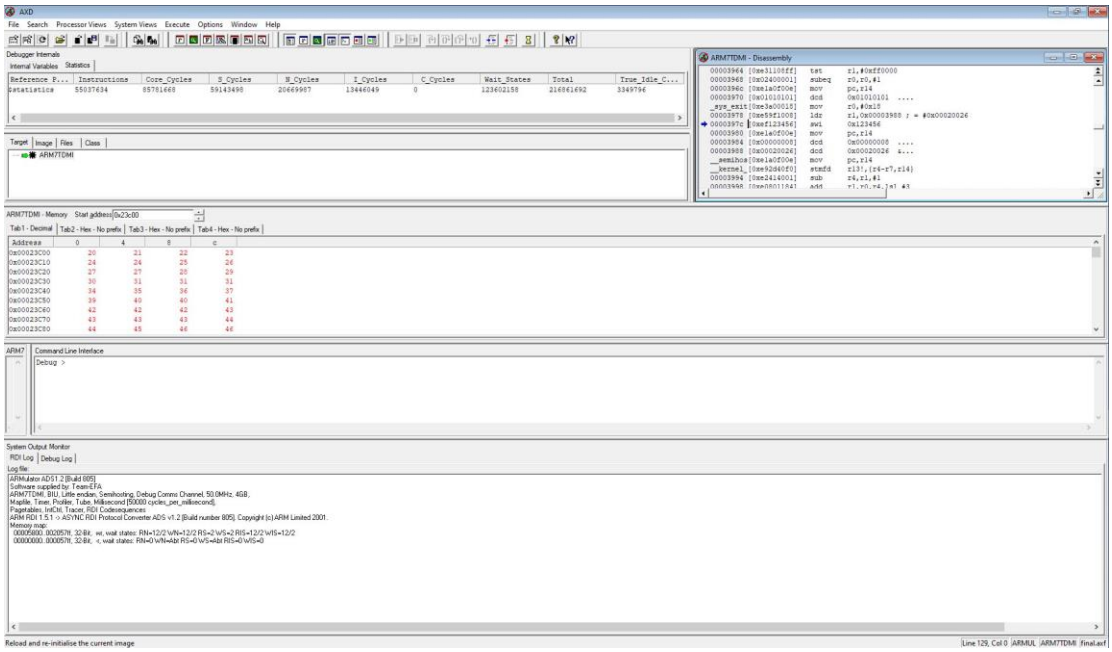
```
00000000 00005800 ROM 4 R 1/1 1/1
00005800 00200000 RAM 4 RW 250/50 250/50
```

scatter.txt:

```
ROM 0x0 0x05800
{
        ROM 0x0 0x5800
        {
                *.o ( +RO )
        }
        RAM 0x5800 0x200000
        {
                * ( +RW )
                * ( +ZI )
        }

}
```

Αποτελέσματα





Παρατηρήσεις

Αυτό που πρέπει να κρατήσουμε εμείς και να έχουμε υπ'όψην μας είναι τα λεγόμενα Wait_states που μας δείχνουν το πόσο περιμένει η cpu την μνήμη μας , αλλά και τα Idle Cycles.

2. Έλεγχος με κατανομή δεδομένων σε ROM SRAM και DRAM

ROM: 0x5800	DRAM: 0x200000	SRAM: 0x200000

Η δυσκολία σε αυτή την τριπλή ιεραρχία μνημών ήταν να μπορέσουμε να βρούμε την βέλτιστη δομή για 3 μνήμες με το σκεπτικό ότι, η μια όπως και πριν θα είναι η ROM μας, στις άλλες δυο έπρεπε να χωρίσουμε τα δεδομένα μας έτσι ώστε στην μια να υπάρχουν μεταβλητές με λίγες προσπελάσεις και στην άλλη μεταβλητές με πολλές προσπελάσεις, για να μπορούμε εν τέλη να αποφασίσουμε πια θα είναι η «γρήγορη» και πια η «αργή» μας μνήμη.

Βλέποντας προσεκτικά τον κώδικα μας, είδαμε ότι οι πίνακες εγγραφής και ανάγνωσης της εικόνας μας, είναι αυτοί που έχουν τις περισσότερες προσπελάσεις.

Έτσι αποφασίστηκε στην γρήγορή μας μνήμη (DRAM) να μπουν οι πίνακες εγγραφής και ανάγνωσης.

Έπειτα από δοκιμές είδαμε ότι στην πραγματικότητα η DRAM μας πρέπει να έχει το ίδιο μέγεθος με την SRAM μας, μιας και οι πίνακες που θέλουμε να αποθηκεύσουμε εκεί, καταλαμβάνουν μεγάλο μέγεθος.

Αρχεία

memory.map:

```
00000000 00005800 ROM 4 R 1/1 1/1
00005800 00001000 DRAM 4 RW 250/50 250/50
00006800 00200000 SRAM 4 RW 10/1 10/1
```

scatter.txt:

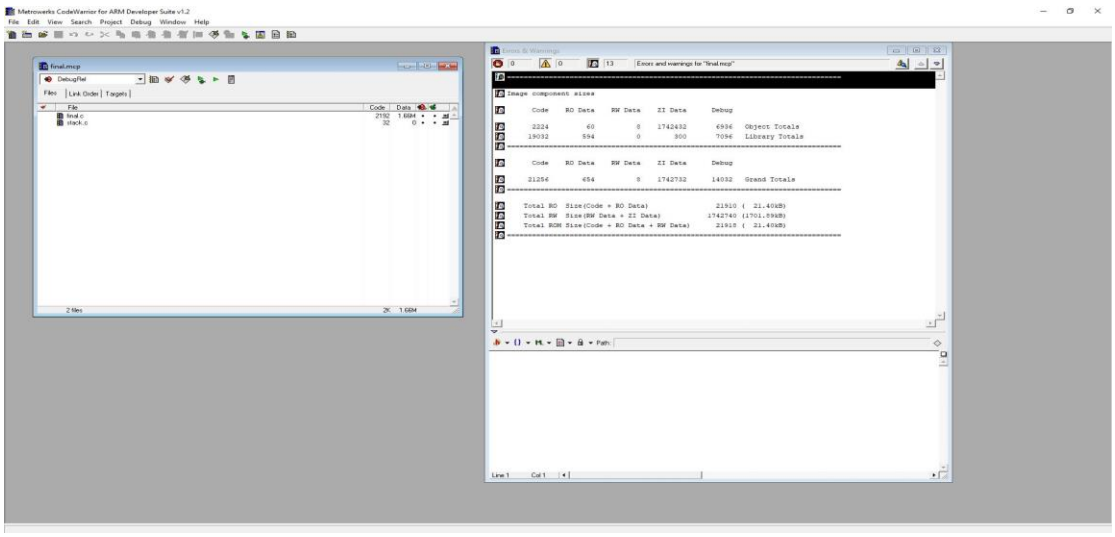
```
ROM 0x0 0x05800
{
        ROM 0x0 0x5800
        {
                *.o ( +RO )
        }
        DRAM 0x5800 0x1000
        {
                * ( +RW )
                * ( +ZI )
        }
        SRAM 0x6800 0x200000
        {
                * (ram)
        }

}
```

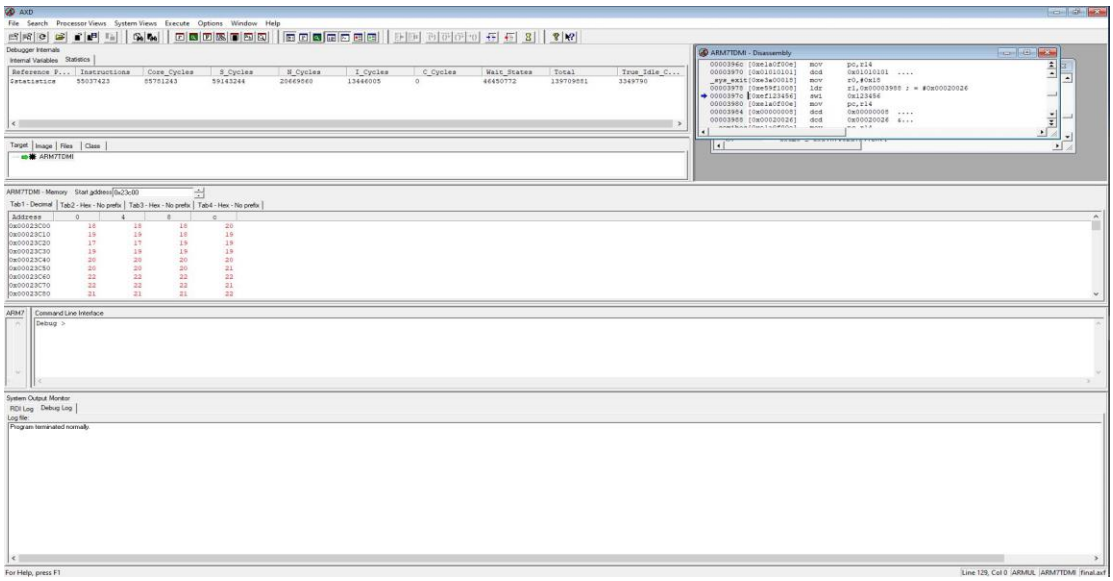
final (σημείο αλλαγών) :

```
#pragma arm section zidata="ram"
int current_y[N][M];
int current_u[N/2][M/2];
int current_v[N/2][M/2];
int new_y[N][M];
int new_u[N/2][M/2];
int new_v[N/2][M/2];
#pragma arm section
```

ουσιαστικά, οτιδήποτε περικλύεται μέσα στο pragma arm section έχει σαν λεζάντα την λέξη ram. Όπως είδαμε το όνομα αυτό της λεζάντας το βάζουμε στο scatter file μεσα στην περιοχή για την SRAM.

Αποτελέσματα





Παρατηρησεις

Όπως μπορούμε να δούμε, σε σύγκριση με την ιεραρχία 2 μνημών, στην ιεραρχία 3ων μνημών έχουμε πετύχει μια πολύ καλή μείωση στα Wait States αλλα και στα Idle Cycles.Απο τα 123.602.158 Wait states έχουμε πέσει στα 46.450.772. Αυτό σημαίνει ότι πετυχαίνουμε βελτίωση της τάξης άνω του 50%.

3. Έλεγχος με κατανομή δεδομένων σε ROM και SRAM

Όπως είδαμε και πριν, μια τριπλή ιεραρχία μνήμης μας βοήθησε να πετύχουμε μια πολύ καλή μείωση στα wait states και κατα συνέπεια στην απόδοση του ενσωματομένου μας που είναι και ένα απο τα ζητούμενα της μελέτης αυτής. Αλλά όπως είπαμε και στην αρχή της εργασίας αυτής, δεν ενδιαφερόμαστε καθαρά για την απόδοση αυτή καθε αυτή αλλά για τον ζύγισμα απόδοσης και κόστους.

Είδαμε ότι για να πετύχουμε την μείωση των Wait States χρειάστηκε να βάλουμε τους πίνακες των δεδομένων μας σε μια σχετικά μεγάλη και γρήγορη μνήμη SRAM, και όλα τα υπόλοιπα να μείνουν στην σχετικά φθηνή και αργή DRAM. Τώρα σκεπτόμενοι ότι επειδή τα δεδομένα που μένουν στην μνήμη DRAM δεν έχουν μεγάλο μέγεθος, μήπως θα ήταν το ίδιο αν όχι καλύτερο να χρησιμοποιήσουμε δυο μόνο μνήμες ; Μια γρήγορη για τα RW Data και τα ZI Data και μια ROM για τον κώδικα και τα RO Data μας.

Αρχεία

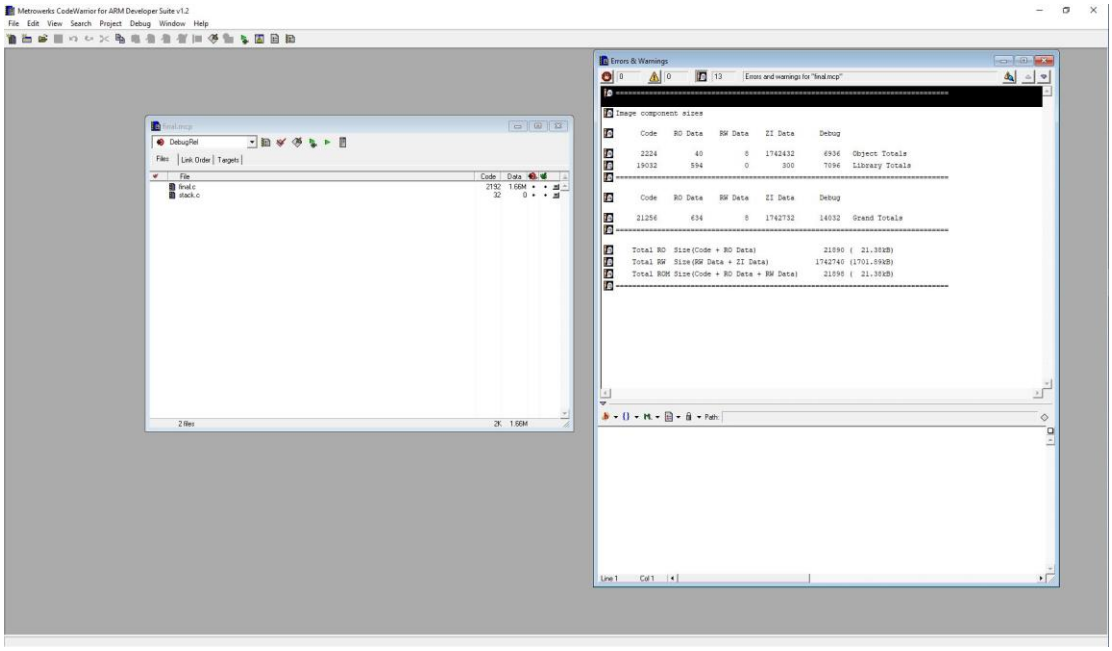
memory.map:

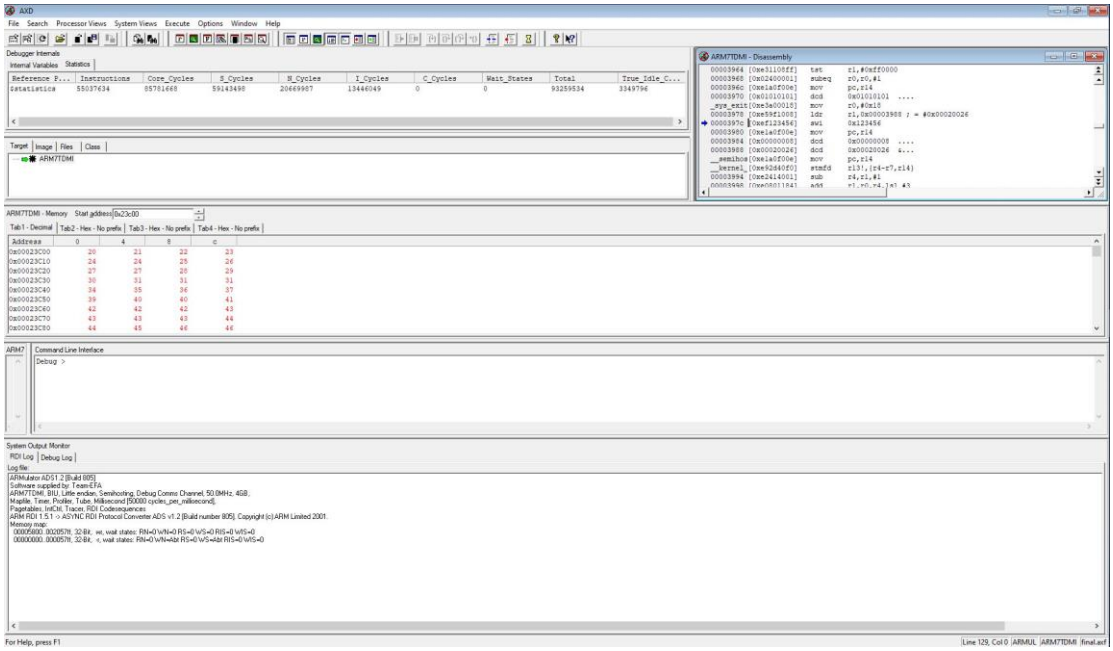
```
00000000 00005800 ROM 4 R 1/1 1/1
00005800 00200000 SRAM 4 RW 1/1 1/1
```

scatter.txt:

```
ROM 0x0 0x05800
{
        ROM 0x0 0x5800
        {
                *.o ( +RO )
        }
        SRAM 0x5800 0x200000
        {
                * ( +RW )
                * ( +ZI )
        }

}
```

Αποτελέσματα





Παρατηρήσεις

Όπως μπορούμε να δούμε τα wait states είναι μηδενικά. Αυτό μάλλον οφείλεται στο ότι ο επεξεργαστής μας συγχρονίζεται με τις μνήμες και δεν τις περιμένει. Κλείνοντας με την 3η ιεραρχία που εξετάσαμε, πετύχαμε και χαμηλότερα Wait States αλλά και έχουμε και μια πιο οικονομική συσκευή έναντι της ιεραρχίας 3ων μνημών, γιατί αφαιρέσαμε την περιττή εν τέλη DRAM.

Συμπεράσματα

Από τα παραπάνω καταλήξαμε να επιλέξουμε και να εφαρμώσουμε την 3η ιεραρχία που εξετάσαμε δηλαδή, ιεραρχία 2 μνημών, μιας ROM που να περιέχει όλα τα RO Data και τον κώδικα του προγράμματος και μια γρήγορη SRAM που να περιέχει όλα τα RW Data και ZI Data.

Αν οι περιορισμοί κόστους και ισχύος ήταν συγκεκριμένοι και είχαμε τεκμηριωμένες πληροφορίες για αυτά τα μεγέθη για το κάθε είδος ram τότε η απόφαση μας πιθανόν να διέφερε, ή να χρειαζόταν κάποια μικρή επεξεργασία τουλάχιστον. Ωστόσο αυτή η εφαρμογή μας δίνει πάρα πολύ καλά αποτελέσματα και τα δεχόμαστε χωρίς να ξέρουμε στην ουσία το κόστος υλοποίησης της.