Arm

1.) File → New → όνομα → ok

(σε άσπρο)
2.) δεξί κλικ → add files → προσθέτω .c αρχείο

3.) DebugRel Settings κουμπί
(-remove (dlog)) -info totals
-libpath C:\Lib -armlib ↙ γράφω αυτό

4.) Uncheck στα options τα 3 πρώτα κουτάκια

5.) δεξί κλικ → compile (σε .c)

6.) Πατάω κουμπί make

Object totals ⇝ οφείλονται σε δικό μας κώδικα
Library totals ⇝ οφείλονται στις βιβλιοθήκες που γίνονται include

- **Code** → πόσα byte πιάνει ο κώδικας στη μνήμη
- **RO data** → Read only → Είναι στο στο αρχικό, αν βάλω const μεταβλητή προστίθεται εδώ
- **RW data** → Read write, Επανεγγράψιμα, αρχικοποιημένα (π.χ. int a=5;)
- **ZI data** (zero initialized) → Read write, μη αρχικοποιημένα, επανεγγράψιμα (π.χ. int a;) → 4 bytes

Debug → δεν μας αφορούν, δεδομένα που οφείλονται στο debugging

Grand Totals → άθροισμα object + library totals

(Μνήμη ROM λειτουργεί και χωρίς ρεύμα γι' αυτό μπαίνουν τα RW data εκεί)

Scanned with CamScanner

7.) Τρέχω AXD Debugger
(σε ανοιχτά windows)

- File → Load image
- Βρίσκω φάκελο project (π.χ. Label)
- Μπαίνω σε φάκελο DebugRel
- Επιλέγω .axf αρχείο
- Open

Εμφανίζεται ο κώδικας σε assembly

- Πατάμε κουμπί Go (μέχρι να σβήσει)

8.) System Views → Debugger Internals
- Statistics

Instructions → εντολές σε assembly

Core Cycles → συνολικοί κύκλοι στον επεξεργαστή

S Cycles (sequencial) → program counter ↑ κάθε φορά κατά 1, διαδοχικές κλήσεις μνήμης

N Cycles (non sequencial) → κάνει <u>branch</u>, μεταπηδά διευθύνσεις, δεν μπορεί να γίνει σωστό pipeline. (όχι διαδοχικές κλήσεις μνήμης)

I Cycles (internal) → κύκλοι στους οποίους δεν κάνει ο επεξεργαστής call στη μνήμη (π.χ. 3 κύκλοι για μια πράξη)

C Cycles → δεν μας αφορούν (αν είχαμε βοηθητικό έξτρα επεξεργαστή)

Τρόποι βελτιστοποίησης

σε Run ανοίγει .axf μετά compile → make → run

1.) Loop unrolling

```
for (i=0; i<1000; i=i++) {
    a[i]=5;
}
```

↓

```
for (i=0; i<1000; i=i+2) {
    a[i] = 5;
    a[i+1] = 5;
}
```

βλέπω core cycles για σύγκριση

εντολή branch εκτελείται τις μισές φορές

Trade off → αυξάνεται ROM μνήμη (αυξάνεται το code)

→ συνήθως 2-4 χρησιμοποιώ

2.) Loop Fusion

- ένωση δύο for σε μία →

```c
for(i=0; i<1000; i++)
    a[i]=5;
for(i=0; i<1000; i++)
    b[i]=5;
        ↓
for(i=0; i<1000; i++)
    a[i]=5;
    b[i]=5;
```

3.) Loop interchange

γραμμή-γραμμή αποθηκεύεται ένας δισδιάστατος πίνακας στη μνήμη στη γλώσσα C (row major)

Άρα

```
for (      ) ← γραμμές
    for (      ) ← στήλες
```

} => sequencial

Αν τα βάλω ανάποδα => non sequencial

Να τα θυμάσαι και τα δύο

4.) Loop collapsing

δύο for (for for)

τη κάνω μονή με

χρήση pointer

π.χ.

```
int a[100][300];

for (i=0; i<300; i++)
    for (j=0; j<100; j++)
        a[j][i]=0;
```

```
int *p = &a[0][0];
for (i=0; i= 30000; i++) {
    *p++ =0
}
```

5.) Loop inversion {βάζω στο τέλος τη while τον έλεγχο}

```
int i, a[100];
 i = 0;
while (i<100){
 a[i]=0;
 i++;
 }
```

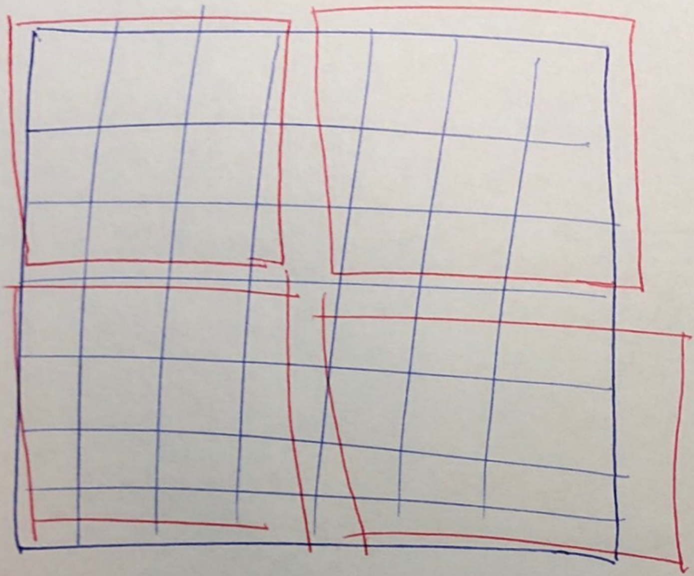
{πολύ μικρή βελτιστοποίηση

αφορά pipeline

δηλ. περίπτωση υπάρχει branch στο τέλος και πάει ξανά πάνω

```
if (i<100) {
   do {
      a[i]=0;
      i++;
   }while (i<100)
```

6.7 Loop tiling



```
for
  for
    for
      for
```

Χωράνε στη μνήμη cache
και ο υπολογισμός γίνεται πιο
γρήγορα

Στην πρώτη εργασία δεν βοηθάει
(χειρότερο αποτέλεσμα)