ΕΣ 08 – Επεξεργαστές Ψηφιακών Σημάτων

Βελτιστοποίηση κώδικα σε επεξεργαστές ΨΕΣ

Τμήμα Επιστήμη και Τεχνολογίας Τηλεπικοινωνιών

Πανεπιστήμιο Πελοποννήσου

- ☐ Εισαγωγή
- ☐ Ανάπτυξη εφαρμογών σε Επεξεργαστές ΨΕΣ
- ☐ Βελτιστοποίηση κώδικα C
- ☐ Συνδυασμός C και Assembly
- ☐ Παράλληλη εκτέλεση εντολών
- ☐ Software Pipelining

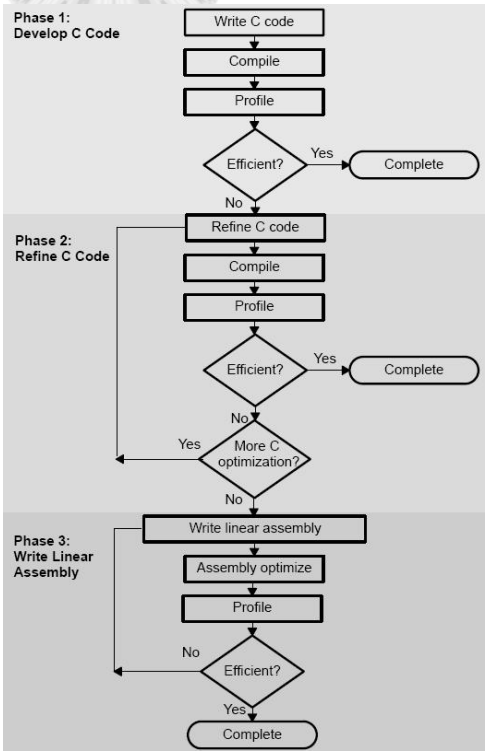
Βιβλιογραφία Ενότητας

- *Kehtarnavaz [2005]: Chapter 7*
- *Chassaing [2005]: Chapter 8*
- *Kuo [2005]: Chapter 4, Section 4.4, Chapter 5, Section 5.3*
- *TMS320C6000 Programmer's Guide*
- *Code Composer Studio Tutorial*
- *TMS320C6000 CPU and Instruction Set Reference Guide*

Εισαγωγή

- ◇ Η ιδιομορφία της αρχιτεκτονικής των επεξεργαστών Ψ.Ε.Σ συνεπάγεται ότι η βέλτιστη αποτελεσματικότητα (μεγαλύτερη ταχύτητα εκτέλεσης προγράμματος ώστε να μπορούν να υποστηριχθούν εφαρμογές επεξεργασίας σήματος σε πραγματικό χρόνο – real time signal processing) επιτυγχάνεται με προγραμματισμό σε ASSEMBLY.
- ◇ Δυστυχώς ο προγραμματισμός σε ASSEMBLY είναι και χρονοβόρος αλλά και επιρρεπής σε σφάλματα. Για το σκοπό αυτό έχουν αναπτυχθεί εργαλεία βελτιστοποίησης κώδικα (code optimization) γραμμένου είτε σε γλώσσα C είτε σε γραμμική ASSEMBLY (δηλαδή ASSEMBLY η οποία δεν λαμβάνει υπόψη τη συγκεκριμένη αρχιτεκτονική του επεξεργαστή τον οποίο θέλουμε να προγραμματίσουμε)
- ◇ Παρά τα ανωτέρω είναι πιθανόν σε κάποιες περιπτώσεις να χρειαστεί προγραμματισμός σε καθαρή ASSEMBLY ώστε να βελτιωθούν κάποια χρονοβόρα τμήματα κώδικα. Για το σκοπό αυτό πρέπει να υπάρχει αναλυτική γνώση τόσο των εντολών που υποστηρίζονται από το συγκεκριμένο επεξεργαστή όσο και της αρχιτεκτονικής του

Ανάπτυξη εφαρμογών σε Επεξεργαστές ΨΕΣ



Η διαδικασία ανάπτυξης εφαρμογών σε επεξεργαστές Ψ.Ε.Σ περιλαμβάνει μια σειρά από βήματα τα οποία εμπλέκουν μια σειρά εργαλείων. Στην πρώτη φάση έχουμε επαλήθευση της πρωτότυπης εφαρμογής σε μια πλατφόρμα ταχείας προτυποποίησης (π.χ. Matlab). Στη συνέχεια ακολουθείται το διάγραμμα ροής του διπλανού σχήματος, δηλαδή:

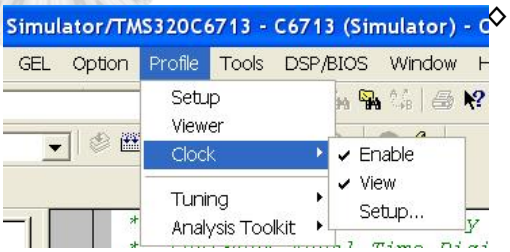
Ανάπτυξη κώδικα σε C χωρίς να λαμβάνεται υπόψη καμία ιδιομορφία του επεξεργαστή. Ο κώδικας C μπορεί να δημιουργηθεί χρησιμοποιώντας το SIMULINK και το Real Time Workshop

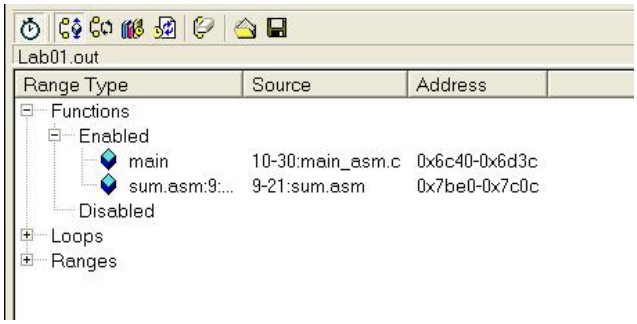
Βελτιστοποίηση κώδικα C χρησιμοποιώντας εργαλεία όπως το Profile του Code Composer Studio. Αν η φάση αυτή δεν οδηγήσει στο απαιτούμενο αποτέλεσμα τότε προχωράμε σε αναγνώριση των κρισίμων τμημάτων κώδικα τα οποία και κωδικοποιούμε με γραμμική ASSEMBLY.

Βελτιστοποίηση του κώδικα ASSEMBLY, δηλαδή μετατροπή της γραμμικής ASSEMBLY σε προσαρμοσμένη στο συγκεκριμένο επεξεργαστή ASSEMBLY.

Κωδικοποίηση με καθαρή ASSEMBLY, τμημάτων κώδικα που εξακολουθούν να εκτελούνται αργά.

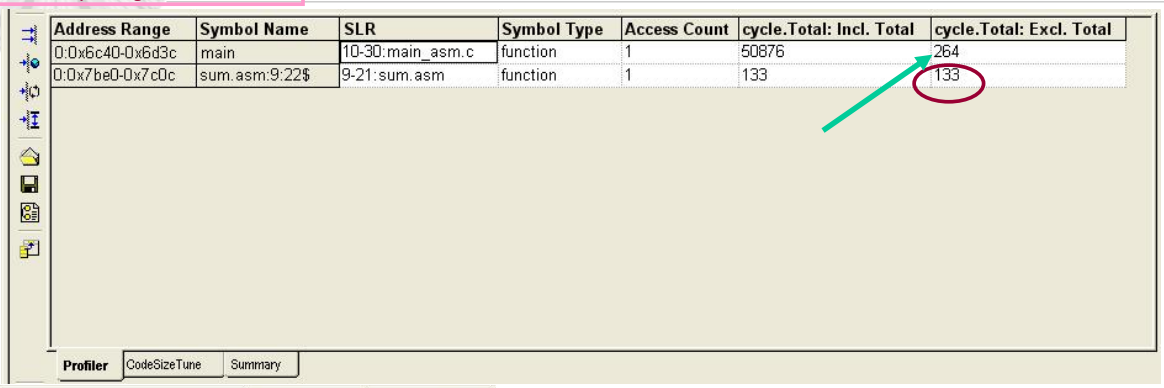
Βελτιστοποίηση κώδικα C





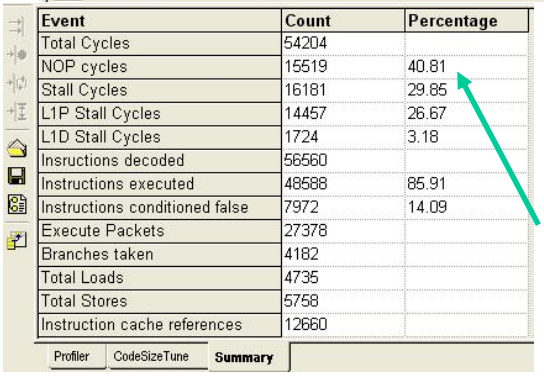
- Η βελτιστοποίηση κώδικα απαιτεί την πραγματοποίηση συγκρίσεων (benchmarking) για διάφορες / διάφορους:
  - Μορφές κώδικα (C, γραμμική ASSEMBLY, καθαρή ASSEMBLY)
  - Ρυθμίσεις του compiler
  - Τύπους δεδομένων προγράμματος (π.χ. short, double)
  - Κατηγορίες εντολών (fixed-point, floating-point)
- Το βασικό εργαλείο για την πραγματοποίηση συγκρίσεων είναι το εργαλείο Profile του Code Composer Studio
  - Επιλέγουμε τις συναρτήσεις που θέλουμε να παρακολουθήσουμε και να συγκρίνουμε την επίδοσή τους (κύκλοι ρολογιού) για τις διάφορες επιλογές, και εκτελούμε το πρόγραμμα μας

Βελτιστοποίηση κώδικα C (II)



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x6c40-0x6d3c	main	10-30:main_asm.c	function	1	50876	264
0:0x7be0-0x7c0c	sum.asm:9:22$	9-21:sum.asm	function	1	133	133

Profiler | CodeSizeTune | Summary



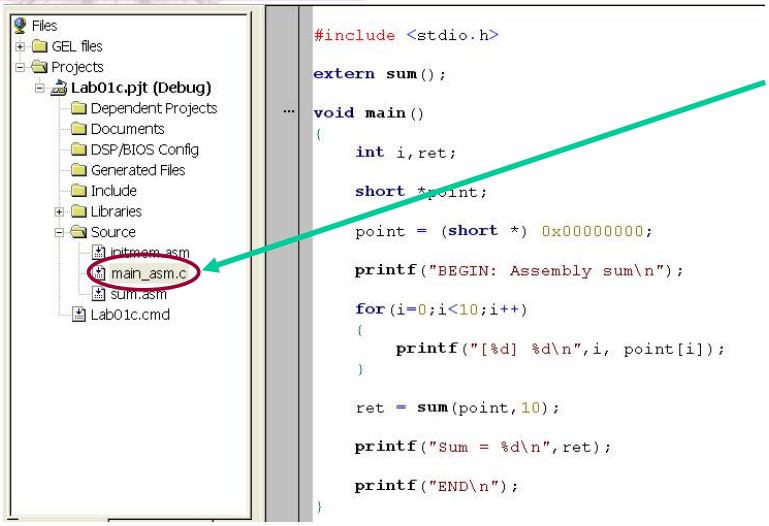
Event	Count	Percentage
Total Cycles	54204	
NOP cycles	15519	40.81
Stall Cycles	16181	29.85
L1P Stall Cycles	14457	26.67
L1D Stall Cycles	1724	3.18
Insructions decoded	56560	
Instructions executed	48588	85.91
Instructions conditioned false	7972	14.09
Execute Packets	27378	
Branches taken	4182	
Total Loads	4735	
Total Stores	5758	
Instruction cache references	12660	

Profiler | CodeSizeTune | Summary

Χρησιμοποιώντας το profile tool βλέπουμε ότι η βασική συνάρτηση main_asm.c εκτελείται σε 264 κύκλους ρολογιού ενώ η συνάρτηση ASSEMBLY sum.asm εκτελείται σε 133 κύκλους ρολογιού

Από τα συνολικά στατιστικά βλέπουμε ότι έχουμε 40.81% του χρόνου του επεξεργαστή να δαπανάται σε NOP (No OPeration) εντολές

Βελτιστοποίηση κώδικα C (III)



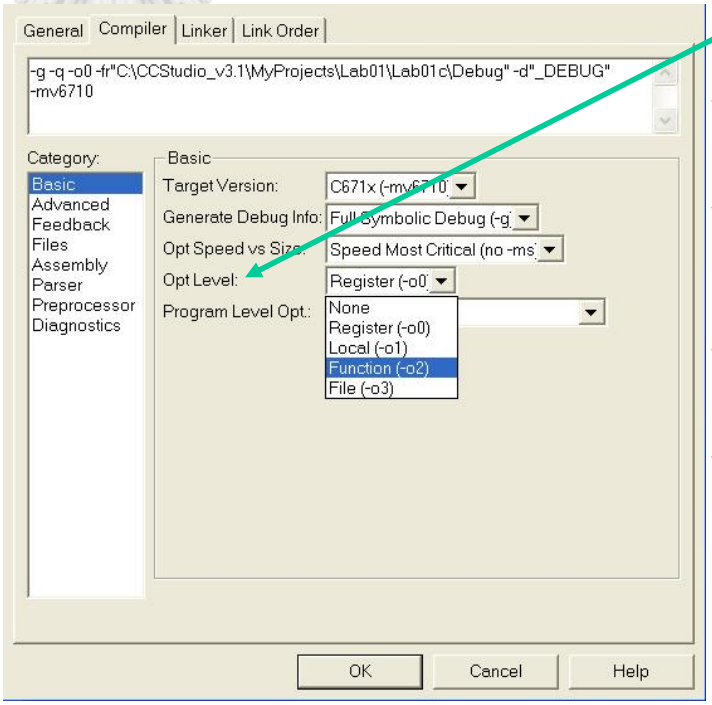
Στο παράδειγμα βλέπουμε πως μπορούμε να βελτιστοποιήσουμε την ταχύτητα εκτέλεσης του προγράμματος main_asm.c χρησιμοποιώντας το optimization tool του compiler

Με βελτιστοποίηση σε επίπεδο καταχωρητών (βλέπε επόμενη διαφάνεια) επιτυγχάνεται μείωση των κύκλων ρολογιού για την εκτέλεση της main_asm.c σε λιγότερο από το μισό.

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x6460-0x6564	main	10-30:main_asm.c	function	1	76047	541
0:0x7c00-0x7c2c	sum.asm:9:22$	9-21:sum.asm	function	1	215	215

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x6c40-0x6d30	main	10-30:main_asm.c	function	1	77107	247
0:0x7be0-0x7c0c	sum.asm:9:22$	9-21:sum.asm	function	1	211	211

Βελτιστοποίηση κώδικα C (IV)

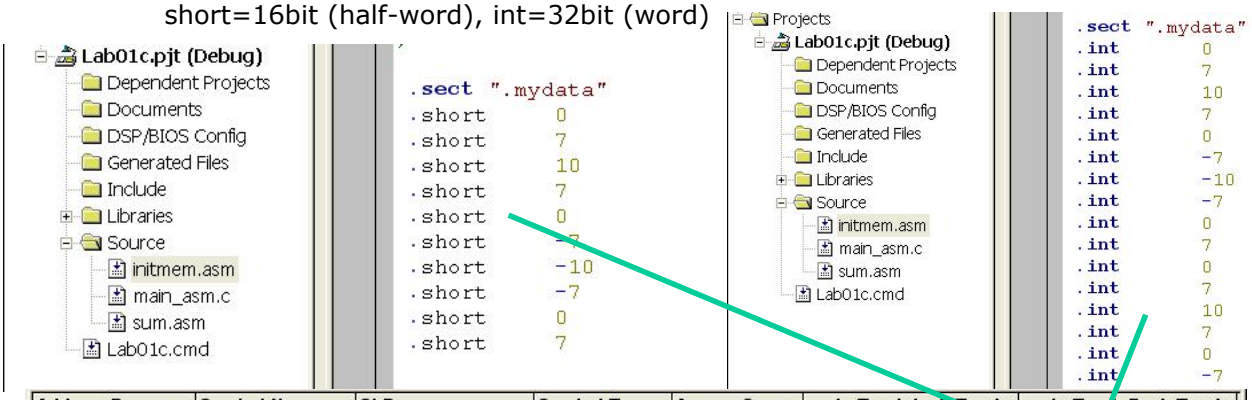


Υπάρχουν τέσσερα επίπεδα βελτιστοποίησης

- ◇ -o0, Register, δηλαδή βελτιστοποίηση χρήσης καταχωρητών
- ◇ -o1, Local, όπως παραπάνω συν προσπάθεια καλύτερης χρήσης των δομικών μονάδων (.L,.M,.D.,S.) των δύο ΚΜΕ που έχει ο επεξεργαστής
- ◇ -o2, Function, όπως παραπάνω συν προσπάθεια βελτιστοποίησης κώδικα ASSEMBLY με software pipelining
- ◇ -o3, File, όπως παραπάνω συν απαλοιφή μη χρησιμοποιούμενων συναρτήσεων

Η επίδραση του τύπου δεδομένων

- Η ταχύτητα εκτέλεσης ενός προγράμματος μειώνεται όταν ο τύπος των δεδομένων (ή των συντελεστών ενός φίλτρου) είναι κινητής υποδιαστολής.
  - Μικρό κέρδος εκτέλεσης επιτυγχάνεται και με διαφοροποίηση ανάμεσα στα bit αναπαράστασης σε αριθμούς σταθερής υποδιαστολής (βλέπε παράδειγμα: short=16bit (half-word), int=32bit (word)



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x7100-0x71e0	main	10-30:main_asm.c	function	1	76262	199
0:0x7bc0-0x7bec	sum.asm:9:22$	9-21:sum.asm	function	1	211	211

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x7100-0x71d0	main	10-30:main_asm.c	function	1	76539	254
0:0x7bc0-0x7bec	sum.asm:9:22$	9-21:sum.asm	function	1	211	211

Συνδυασμός C και Assembly

```c
#include <stdio.h>

extern sum();

void main()
{
    int i,ret;

    short *point;

    point = (short *) 0x00000000;

    printf("BEGIN: Assembly sum\n");

    for(i=0;i<10;i++)
    {
        printf("[%d] %d\n",i, point[i]);
    }

    ret =sum(point,10);

    printf("Sum = %d\n",ret);

    printf("END\n");
}
```

```
    .global _sum
_sum:
        ZERO    .L1 A9
        MV  .L1 B4,A2

loop:   LDH .D1 *A4++, A7
        NOP 4
        ADD .L1 A7,A9,A9
  [A2]  SUB .L1 A2,1,A2
  [A2]  B   .S1 loop
        NOP 5

        MV  .L1 A9,A4
        B   .S2 B3      ;
        NOP 5
```

- Σε πολλές περιπτώσεις οι βελτιστοποιήσεις στον κώδικα C δεν είναι επαρκείς και χρειάζεται να πάρουμε την 'κατάσταση στα χέρια' μας:
- Υπάρχουν τρεις δυνατές επιλογές:
  - Γραμμική ASSEMBLY (αρχεία .sa),
  - ASM εντολές (statements) σε κώδικα C: asm("assembly code")
  - Συναρτήσεις ASSEMBLY οι οποίες μπορούν να κληθούν από κώδικα C (C-callable assembly function (.asm)). Στην περίπτωση αυτή μπορούμε να βελτιστοποιήσουμε συγκεκριμένα κομμάτια κώδικα εφαρμόζοντας τεχνικές όπως παράλληλη εκτέλεση εντολών και software pipelining

- ☑ Εισαγωγή
- ☑ Ανάπτυξη εφαρμογών σε Επεξεργαστές ΨΕΣ
- ☑ Βελτιστοποίηση κώδικα C
- ★ Συνδυασμός C και Assembly
- ☐ Παράλληλη εκτέλεση εντολών
- ☐ Software Pipelining

Assembly συναρτήσεις καλούμενες από C

```
#include <stdio.h>

extern sum();

void main()
{
    int i,ret;

    short *point;

    point = (short *) 0x00000000;

    printf("BEGIN: Assembly sum\n");

    for(i=0;i<10;i++)
    {
        printf("[%d] %d\n",i, point[i]);
    }

    ret = sum(point,10);

    printf("Sum = %d\n",ret);

    printf("END\n");
}
```

```
    .global _sum
_sum:
        ZERO    .L1 A9
        MV      .L1 B4,A2
loop:   LDH     .D1 *A4++,A7
        NOP 4
        ADD     .L1 A7,A9,A9
[A2]    SUB     .L1 A2,1,A2
[A2]    B       .S1 loop
        NOP 5

        MV      .L1 A9,A4
        B       .S2 B3
        NOP 5
```

- ◇ Τα ορίσματα των συναρτήσεων μεταφέρονται μέσω των καταχωρητών A4, B4, A6, B6, ... με τη σειρά που δίνεται
- ◇ Το αποτέλεσμα της κλήσης της συνάρτησης επιστρέφεται μέσω του καταχωρητή A4.
- ◇ Η διεύθυνση επιστροφής από τον κώδικα στον οποίο έχει γίνει η κλήση της συνάρτησης δίνεται στον καταχωρητή B3. Οπότε πρέπει να δίνεται μεγάλη προσοχή να μην χρησιμοποιείται για άλλο σκοπό ο B3 εντός της συνάρτησης ASSEMBLY
- Ονοματολογία (βλέπε προηγ. διαφάνεια):
  - Στον κώδικα C: label (η συνάρτηση δηλώνεται ως external)
  - Στον κώδικα ASSEMBLY: _label (η συνάρτηση δηλώνεται ως global)
  - Πρόσβαση σε global μεταβλητές εντός του κώδικα ASSEMBLY: .ref _variablename

ΕΣ 08: Επεξεργαστές Ψηφιακών Σημάτων

- ☑ Εισαγωγή
- ☑ Ανάπτυξη εφαρμογών σε Επεξεργαστές ΨΕΣ
- ☑ Βελτιστοποίηση κώδικα C
- ★ Συνδυασμός C και Assembly
- ☐ Παράλληλη εκτέλεση εντολών
- ☐ Software Pipelining

Σκελετός assembly συναρτήσεων καλούμενων από C

```
; header comments
; passed in parameters in 32-bit registers A4, B4, A6, ... in that order

            .def _myfunc            ; allow calls from external
ACONSTANT   .equ 100                ; declare constants
            .ref _aglobalvariable   ; refer to a global variable

_myfunc:    NOP                     ; instructions go here
            B       B3              ; return (branch to addr B3)
                                    ; function output will be in A4
            NOP     5               ; pipeline flush

            .end
```

Σκελετός assembly συναρτήσεων καλούμενων από C

Παράδειγμα assembly συνάρτησης καλούμενης από C

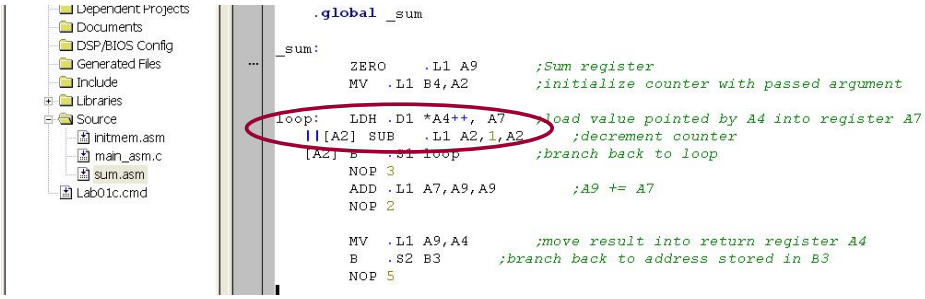
```
;FIRCASMfunc.asm ASM function called from C to implement FIR
;A4 = Samples address, B4 = coeff address, A6 = filter order
;Delays organized as:x(n-(N-1))...x(n);coeff as h[0]...h[N-1

        .def    _fircasmfunc
_fircasmfunc:               ;ASM function called from C
        MV      A6,A1       ;setup loop count
        MPY     A6,2,A6     ;since dly buffer data as byte
        ZERO    A8          ;init A8 for accumulation
        ADD     A6,B4,B4    ;since coeff buffer data as byte
        SUB     B4,1,B4     ;B4=bottom coeff array h[N-1]
loop:                       ;start of FIR loop
        LDH     *A4++,A2    ;A2=x[n-(N-1)+i] i=0,1,...,N-1
        LDH     *B4--,B2    ;B2=h[N-1-i] i=0,1,...,N-1
        NOP     4
        MPY     A2,B2,A6    ;A6=x[n-(N-1)+i]*h[N-1-i]
        NOP
        ADD     A6,A8,A8    ;accumlate in A8
        LDH     *A4,A7      ;A7=x[(n-(N-1)+i+1]update delays
        NOP     4           ;using data move "up"
        STH     A7,*-A4[1]  ;-->x[(n-(N-1)+i] update sample
        SUB     A1,1,A1     ;decrement loop count
    [A1] B      loop        ;branch to loop if count # 0
        NOP     5

        MV      A8,A4       ;result returned in A4
        B       B3          ;return addr to calling routine
        NOP     4
```

Παράλληλη εκτέλεση στις δομικές μονάδες

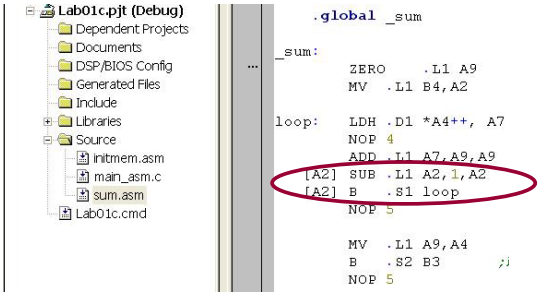
- Εκμεταλλευόμενοι το γεγονός ότι έχουμε 8 ανεξάρτητες μονάδες στον επεξεργαστή μπορούμε να εκτελέσουμε παράλληλα μέχρι και 8 εντολές.
- Στο επόμενο παράδειγμα οι εντολές LDH και SUB μπορούν να εκτελεστούν παράλληλα (βλέπε ||) με αποτέλεσμα την επιπλέον βελτίωση της ταχύτητας εκτέλεσης σε 116 κύκλους ρολογιού

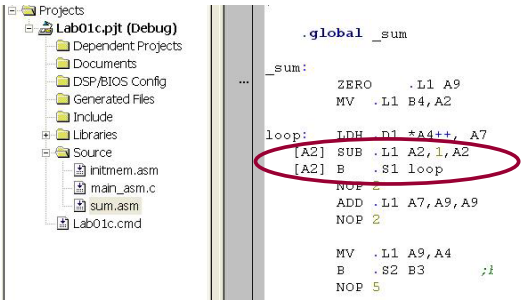


Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x7100-0x71d0	main	10-30:main_asm.c	function	1	76444	254
0:0x7bc0-0x7bec	sum.asm:9:22$	9-21:sum.asm	function	1	116	116

Software Pipelining

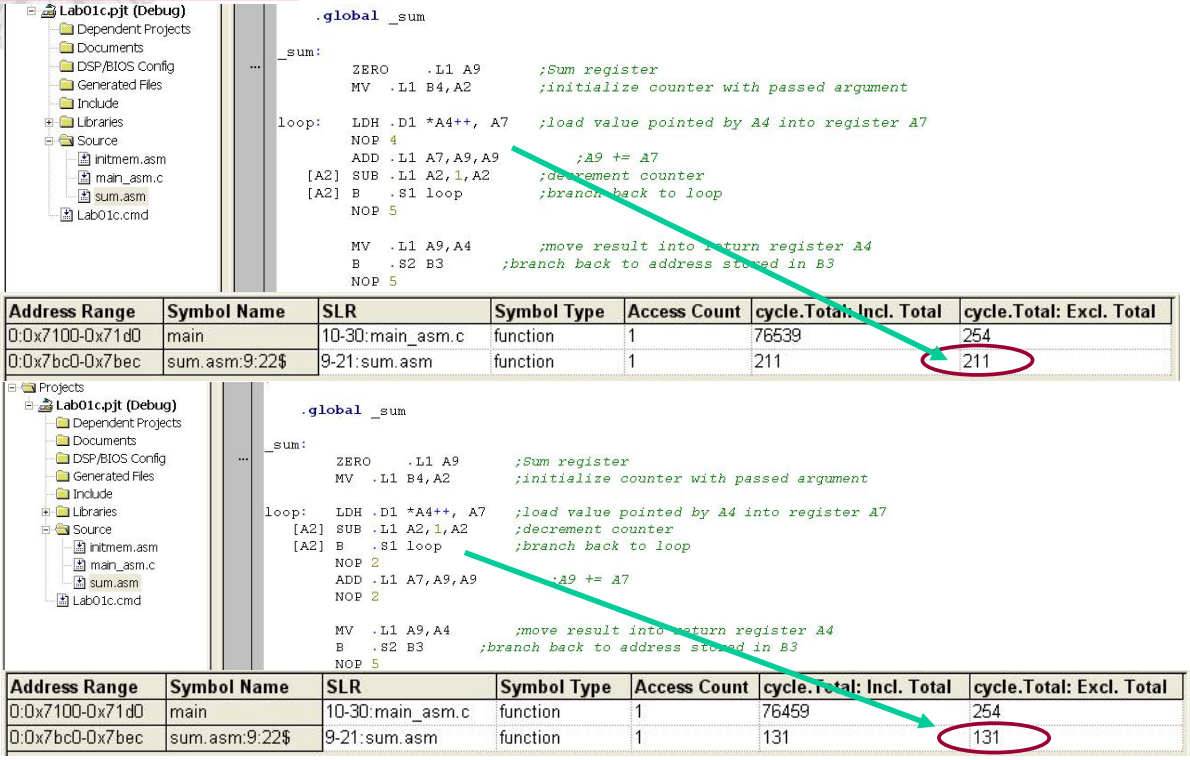
- Η τεχνική software pipelining εφαρμόζεται σε συναρτήσεις ASSEMBLY και προσπαθεί να αναδιατάξει τη σειρά εκτέλεσης των εντολών ώστε να ελαχιστοποιούνται οι εντολές NOP.
- Στο επόμενο παράδειγμα οι εντολές SUB και B δεν περιμένουν αποτέλεσμα από την εντολή LDH (σε αντίθεση με την εντολή ADD) ούτε επηρεάζουν άμεσα την εντολή ADD, άρα μπορούν να αναδιαταχθούν πριν από αυτή με χρήση της τεχνικής delayed branch





- ☑ Εισαγωγή
- ☑ Ανάπτυξη εφαρμογών σε Επεξεργαστές ΨΕΣ
- ☑ Βελτιστοποίηση κώδικα C
- ☑ Συνδυασμός C και Assembly
- ☑ Παράλληλη εκτέλεση εντολών
- ★ Software Pipelining

Software Pipelining (II)



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x7100-0x71d0	main	10-30:main_asm.c	function	1	76539	254
0:0x7bc0-0x7bec	sum.asm:9:22$	9-21:sum.asm	function	1	211	211

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl. Total	cycle.Total: Excl. Total
0:0x7100-0x71d0	main	10-30:main_asm.c	function	1	76459	254
0:0x7bc0-0x7bec	sum.asm:9:22$	9-21:sum.asm	function	1	131	131