Εθνικο & Καποδιστριακο Πανεπιστημιο Αθηνων
Τμημα Πληροφορικης & Τηλεπικοινωνιων
Μαθημα: **Αρχιτεκτονικη Υπολογιστων ΙΙ**

Εργασία 1 (*υποχρεωτική*) – Διοχέτευση

Ακαδημαϊκο Ετος 2019 – 2020
(Εκφωνηση) Τεταρτη 20 Νοεμβριου 2019
(Παραδοση στο eclass μεχρι) **Τριτη 10 Δεκεμβριου 2019**

Επώνυμο	Όνομα	Αριθμός Μητρώου	Email
ΚΑΡΑΘΑΝΟΣ	ΑΛΕΞΑΝΔΡΟΣ	1115201400326	Sdi1400326@di.uoa.gr
ΑΛΕΞΙΟΥ	ΙΩΑΝΝΗΣ	1115200900254	Sdi0900254@di.uoa.gr

Πληροφορίες για τις Υποχρεωτικές Εργασίες του μαθήματος

- Οι υποχρεωτικές εργασίες του μαθήματος είναι **δύο**. Σκοπός τους είναι η κατανόηση των εννοιών του μαθήματος με χρήση αρχιτεκτονικών προσομοιωτών. Η πρώτη υποχρεωτική εργασία (αυτή) αφορά τη διοχέτευση (pipelining) και η δεύτερη θα αφορά τις κρυφές μνήμες (cache memories).
- Οι δύο αυτές εργασίες είναι υποχρεωτικές και η βαθμολογία του μαθήματος θα προκύπτει από το γραπτό (60%), την εργασία της διοχέτευσης (20%), και την εργασία των κρυφών μνημών (20%). Αυτός είναι ο τρόπος εξέτασης του μαθήματος για όσους φοιτητές έχουν αριθμό μητρώου 2009 και μεταγενέστερο (δηλαδή πήραν το μάθημα για πρώτη φορά το εαρινό εξάμηνο του 2012 και μετά). Καθένας από τους τρεις βαθμούς πρέπει να είναι προβιβάσιμος για να περαστεί προβιβάσιμος βαθμός στη γραμματεία.
- Για τους παλαιότερους φοιτητές (με αριθμό μητρώου 2008 και παλαιότερο) οι δύο εργασίες (pipeline, cache) είναι προαιρετικές. Αν κάποιος παλαιότερος φοιτητής δεν τις παραδώσει θα βαθμολογηθεί με ποσοστό 100% στο γραπτό. Αν τις παραδώσει, θα βαθμολογηθεί με τον παραπάνω τρόπο (γραπτό + 2 εργασίες).
- Κάθε ομάδα μπορεί να αποτελείται **από 1 έως και 3 φοιτητές**. Συμπληρώστε τα στοιχεία όλων των μελών της ομάδας στον παραπάνω πίνακα. Όλα τα μέλη της ομάδας πρέπει να έχουν ισότιμη συμμετοχή και να γνωρίζουν τις λεπτομέρειες της υλοποίησης της ομάδας.
- Για την εξεταστική Σεπτεμβρίου δε θα δοθούν άλλες εργασίες. Το Σεπτέμβριο εξετάζεται μόνο το γραπτό.
- Σε περίπτωση αντιγραφής θα μηδενίζονται όλες οι ομάδες που μετέχουν σε αυτή.
- Η παράδοση της **Εργασίας Διοχέτευσης** πρέπει να γίνει μέχρι τα **μεσάνυχτα της προθεσμίας ηλεκτρονικά** και μόνο στο eclass (να ανεβάσετε ένα μόνο αρχείο zip ή rar με την τεκμηρίωσή σας σε PDF και τον κώδικά σας). **Μην περιμένετε μέχρι την τελευταία στιγμή. Δεν θα υπάρξει παράταση στην προθεσμία παράδοσης ώστε να διατεθεί αρκετός χρόνος και για την εργασία των κρυφών μνημών της οποίας η εκφώνηση θα δοθεί αμέσως μετά**.

Ζητούμενο

Το ζητούμενο της εργασίας είναι να υλοποιήσετε ένα πρόγραμμα σε συμβολική γλώσσα για τον προσομοιωτή WinMIPS64 με σκοπό την εκτέλεσή του **στον μικρότερο δυνατό χρόνο** (μικρότερο αριθμό κύκλων ρολογιού).

Το πρόγραμμα πρέπει να εκτελεί την εξής απλή λειτουργία: υπολογίζει το άθροισμα των 500 στοιχείων ενός πίνακα απρόσημων (unsigned) ακεραίων αριθμών K[i] (i = 0, 1, … 499) που βρίσκονται στην μνήμη (στο τμήμα .data του πηγαίου αρχείου συμβολικής γλώσσας). Οι αριθμοί K[i] έχουν τιμές μεταξύ του 0 και του 127, ο πίνακας δεν είναι ταξινομημένος και οποιαδήποτε τιμή μπορεί να εμφανίζεται οποιονδήποτε αριθμό φορών. Στο τέλος της εκτέλεσης του προγράμματος να εκτυπώνεται στο Terminal το άθροισμα: “Sum=...”.

Μπορείτε να χρησιμοποιήσετε *οποιεσδήποτε* ρυθμίσεις του προσομοιωτή (Enable/Disable Forwarding, Enable/Disable Branch Target Buffer, Enable/Disable Delay Slot) και να γράψετε τον κώδικά σας και τις δομές δεδομένων σας με *όποιον* τρόπο θέλετε – μόνος στόχος είναι να ελαχιστοποιήσετε τον χρόνο εκτέλεσης του προγράμματος. Οι μόνοι περιορισμοί είναι αυτοί της προηγούμενης παραγράφου.

Μεταξύ των προγραμμάτων που εκτελούνται σωστά, τα ταχύτερα (που διαρκούν τον μικρότερο αριθμό κύκλων ρολογιού) θα βαθμολογηθούν με μεγαλύτερο βαθμό.

Εκτός από το πρόγραμμά σας σε συμβολική γλώσσα, το οποίο πρέπει να παραδώσετε σε ξεχωριστό αρχείο, να συμπληρώσετε τον ακόλουθο πίνακα για το πρόγραμμά σας.

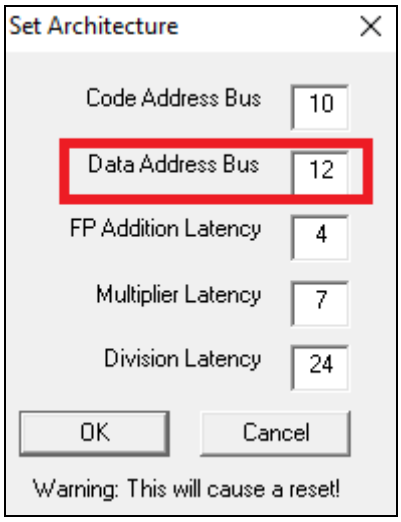
	Ενεργοποίηση Forwarding (Ναι / Όχι)	Ενεργοποίηση Branch Target Buffer (Ναι / Όχι)	Ενεργοποίηση Branch Delay Slot (Ναι / Όχι)	Κύκλοι ρολογιού εκτέλεσης
Απαντήσεις για το Πρόγραμμα μου	ΝΑΙ	ΝΑΙ	ΟΧΙ	1073

Τεκμηρίωση

[Σύντομη τεκμηρίωση της λύσης σας μέχρι **5 σελίδες ξεκινώντας από την επόμενη** – μην αλλάζετε τη μορφοποίηση του κειμένου (**και παραδώστε την τεκμηρίωση σε αρχείο PDF**). Η τεκμηρίωσή σας πρέπει να περιλαμβάνει παραδείγματα ορθής εκτέλεσης του προγράμματος και σχολιασμό για την επίλυση του προβλήματος και την επίτευξη μικρότερου χρόνου εκτέλεσης. Μπορείτε να χρησιμοποιήσετε εικόνες, διαγράμματα και ό,τι άλλο μπορεί να βοηθήσει στην εξήγηση της δουλειάς σας.]

Τεκμηρίωση Εργασίας 1

Αρχικές ρυθμίσεις: Πριν την εκκίνηση του προγράμματός μας, πρέπει να αλλάξουμε τις ρυθμίσεις του WinMIPS64, ώστε να μπορεί να τρέξει το πρόγραμμά μας, όπως στην ακόλουθη εικόνα:



Εικόνα 1:Ρύθμιση WinMIPS64 στο path Configure / Architecture / Set Architecture.

Δηλαδή από το 10 που είναι η default τιμή, την ορίζουμε στα 12, καθώς θέλουμε να μεγαλώσουμε τη μνήμη στο data segment, καθώς έχουμε ένα μεγάλο πίνακα (500 θέσεων να τον ορίσουμε στατικά στο κομμάτι data του προγράμματος) και να έχουμε δηλαδή διαθέσιμα 2^{12} bytes για data.
Αξίζει να σημειωθεί ότι ο πίνακας είναι ορισμένος στατικά στο πεδίο data του κώδικα, έχοντας επιλέξει για καθένα από τα 500 στοιχεία του πίνακα τιμές στο εύρος [0, 127].

Ζητούμενο: Το ζητούμενο της άσκησης ήταν η δημιουργία προγράμματος σε assembly MIPS για εκτέλεση στον προσομοιωτή WinMIPS64, το οποίο να μπορεί να αθροίζει τα στοιχεία ενός πίνακα 500 θέσεων, του οποίου οι τιμές κυμαίνονται στο εύρος [0, 127] και να εκτυπώνει το άθροισμα.

Διαδικασία που ακολουθήθηκε: Πριν περάσουμε στο τελικό αλγόριθμο που επιλέξαμε για μια σχετικά γρήγορη εκτέλεση σύμφωνα με αυτά που συζητήθηκαν εντός της ομάδας μας, κρίνουμε σκόπιμο να παραθέσουμε εν συντομία τη συλλογιστική πορεία μας προς την επιλογή αυτή.
Καταρχάς η πρώτη συζήτηση και προβληματισμός ήταν για το αν θα χρησιμοποιηθεί επαναληπτική διαδικασία ή αναδρομική. Σαν αλγόριθμος η επαναληπτική διαδικασία θα ήταν σε μορφή κώδικα c:

```
int sum(int arr[], int n)
{
    int sum = 0; // initialize sum

    // Iterate through all elements
    // and add them to sum
    for (int i = 0; i < n; i++)
    sum += arr[i];

    return sum;
}
```

Και θα είχε γραμμική πολυπλοκότητα τάξης O(n).
Σαν αλγόριθμος η αναδρομική διαδικασία θα ήταν σε μορφή κώδικα c:

```
int findSum(int A[], int N)
{
    if (N <= 0)
        return 0;
    return (findSum(A, N - 1) + A[N - 1]);
}
```

Και θα είχε και αυτός ο αλγόριθμος γραμμική πολυπλοκότητα τάξης O(n).
Συνεπώς εστιάζουμε στην ελαχιστοποίηση των εξαρτήσεων που προκύπτουν κατά την εκτέλεση του προγράμματος. Τέτοια μπορεί να είναι RAW stalls, αλλά και αρκετά Branch stalls, δεδομένου ότι πρέπει να προσπελαστούν όλα τα στοιχεία του πίνακα (500 σύνολο) για να αθροιστούν. Το πλήθος εντολών του τελικού προγράμματος, στο οποίο καταλήξαμε, μπορεί μεν να είναι μεγάλο, ωστόσο πιο μεγάλη σημασία έχει να καταφέρουμε να μειώσουμε την πολυπλοκότητα των προαναφερθέντων και να μη χρειαστεί να τις κάνουμε όλες τις πράξεις σε όλες τις επαναλήψεις.
Άρα εστιάσαμε στα εξής:

- Να βρούμε έναν πιο αποδοτικό αλγόριθμο ώστε να γίνονται οι λιγότεροι δυνατοί έλεγχοι.
- Να μειωθεί το CPI, με το να μειωθούν οι RAW εξαρτήσεις με χρήση forwarding και η ελαχιστοποίηση των branches που πρέπει να υπολογιστούν με τη λύση της τεχνικής loop unrolling, εκτός από μείωσή τους από αλγοριθμικής απόψεως.

Να σημειωθεί ότι το σωστό αποτέλεσμα, όπου και εκτυπώνεται στο terminal με το πέρας κάθε προσπάθειας, για αυτόν τον πίνακα είναι 18575.

Στατιστικά προσπαθειών πριν το επιλεχθέν: Για να έχει νόημα η σύγκριση των προσπαθειών μας και πως αυτές οδήγησαν στην τελική έκδοση, κρίναμε σκόπιμο να παρουσιάσουμε την πρώτη προσπάθεια, ξεκινώντας από την επιλογή της επαναληπτικής μεθόδου, μέχρι την τελευταία προσπάθεια που ήταν η επιλογή της αναδρομικής μεθόδου.

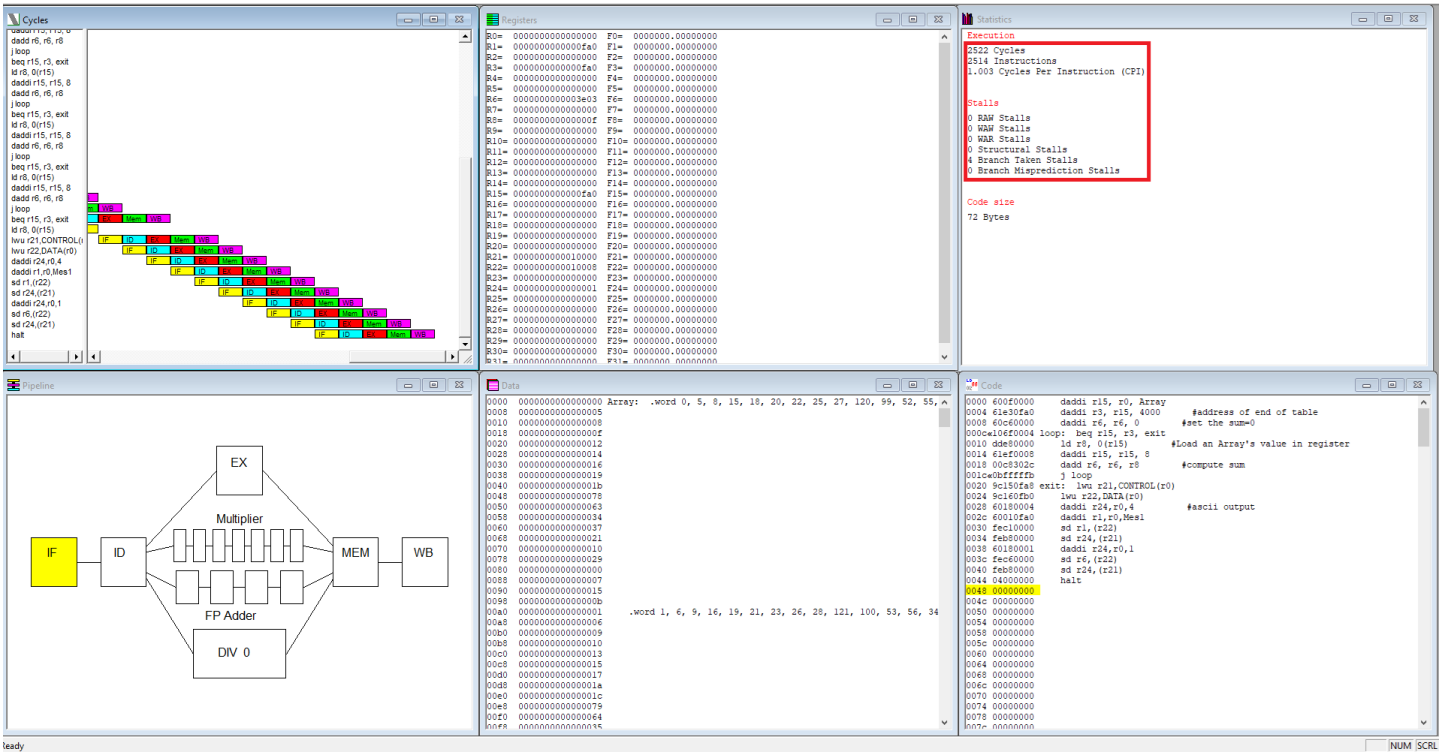
1. **Απλή επαναληπτική μέθοδος:**

Στην ακόλουθη εικόνα φαίνεται η πρώτη προσπάθεια εκπόνησης του προγράμματος, με τη loop να "τρώει" την περισσότερη πολυπλοκότητα. Το loop θα εκτελεστεί 500 φορές και με δοκιμές ελέγξαμε ότι ο πιο αποδοτικός τρόπος είναι σε συνδυασμό με το Forwarding και το Branch Target Buffer.

```
main:

        daddi r15, r0, Array
        daddi r3, r15, 4000          #address of end of table
        daddi r6, r6, 0              #set the sum=0

loop:   beq r15, r3, exit
        ld r8, 0(r15)                #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8              #compute sum
        j loop

exit:   lwu r21,CONTROL(r0)
        lwu r22,DATA(r0)
        daddi r24,r0,4               #ascii output
        daddi r1,r0,Mes1
        sd r1,(r22)
        sd r24,(r21)
        daddi r24,r0,1
        sd r6,(r22)
        sd r24,(r21)
        halt
```

Εικόνα 2: Πρώτη προσπάθεια κώδικα.

Τα αντίστοιχα αποτελέσματα φαίνονται και στην ακόλουθη εικόνα:



Εικόνα 3: Εικόνα τερματισμού από την 1η προσπάθεια υλοποίησης.

Σε αυτή την περίπτωση τα αποτελέσματα όπως βλέπουμε είναι τα εξής:

```
Forwarding and Branch Target Buffer active:
Execution:
2522 Cycles
2514 Instructions
1.003 Cycles Per Instruction (CPI)

Stalls:
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
4 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
72 bytes
```

Επιτυγχάνουμε να αποφύγουμε τα RAW stalls, καθώς δεν έχουμε καθυστερήσεις μετά από εντολές τύπου ld. Επίσης, αποφεύγουμε τα delay slots, καθώς λόγω του μεγάλου πλήθους από branches, προστίθενται αρκετές καθυστερήσεις που μας αυξάνουν το κόστος σε κύκλους, και εντέλει έχουμε 3019 κύκλους με CPI 1.003.

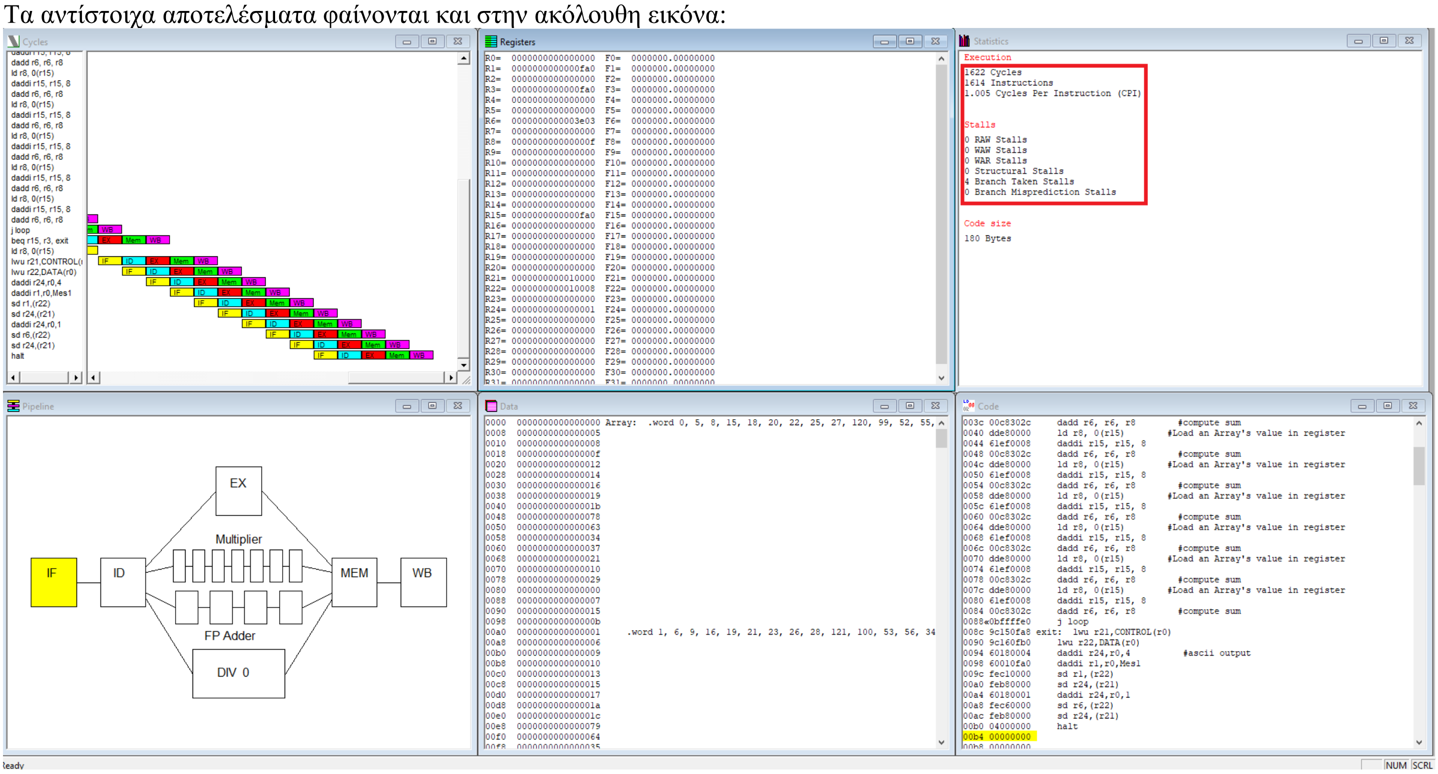
2. Βελτιωμένη επαναληπτική μέθοδος:

Στην επόμενη εικόνα φαίνεται η δεύτερη προσπάθεια εκπόνησης του προγράμματος, με τη loop να "τρώει" πάλι την περισσότερη πολυπλοκότητα. Ωστόσο μέσα στο loop κάναμε χειροκίνητα 8 επαναλήψεις και εντέλει θα εκτελεστεί 50 φορές και με δοκιμές ελέγξαμε ότι ο πιο αποδοτικός τρόπος είναι σε συνδυασμό με το Forwarding και το Branch Target Buffer.

```
loop:   beq r15, r3, exit
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        ld r8, 0(r15)           #Load an Array's value in register
        daddi r15, r15, 8
        dadd r6, r6, r8         #compute sum
        j loop
```

Εικόνα 4: Δεύτερη προσπάθεια κώδικα.

Τα αντίστοιχα αποτελέσματα φαίνονται και στην ακόλουθη εικόνα:



Εικόνα 5: Εικόνα τερματισμού από την 2η προσπάθεια υλοποίησης.

Σε αυτή την περίπτωση τα αποτελέσματα όπως βλέπουμε είναι τα εξής:

```
Forwarding and Branch Target Buffer active:
Execution:
1622 Cycles
1614 Instructions
```

```
1.005 Cycles Per Instruction (CPI)

Stalls:
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
4 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
180 bytes
```

Παρατηρούμε κατ' αντιστοιχία με την προηγούμενη περίπτωση, οι κύκλοι μειώθηκαν κατά 36% περίπου και φτάνουμε τους 1622 κύκλους με CPI 1.005.

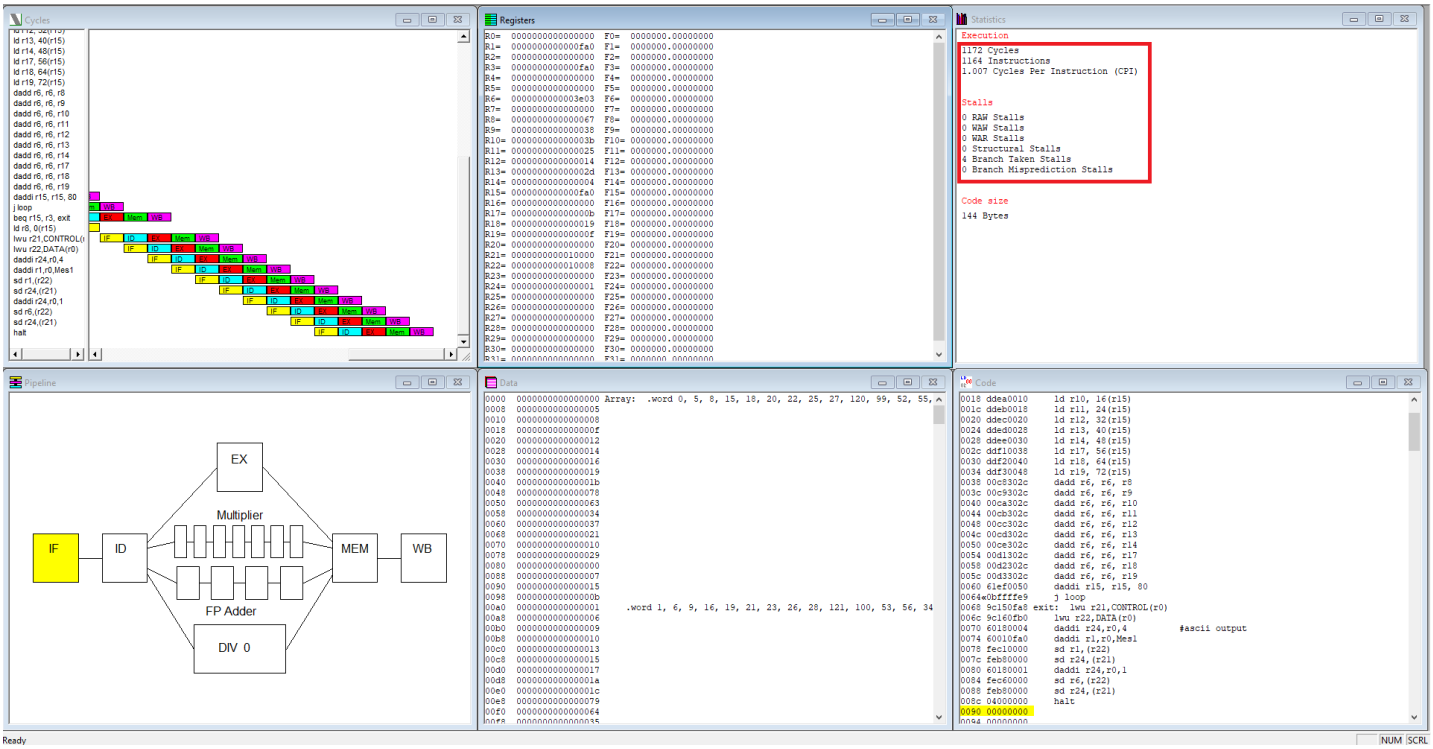
3. Βελτιωμένη επαναληπτική μέθοδος – 10 Loop Unrolling

Στην επόμενη εικόνα φαίνεται η τρίτη προσπάθεια εκπόνησης του προγράμματος, με τη loop να "τρώει" πάλι την περισσότερη πολυπλοκότητα. Ωστόσο μέσα στο loop κάναμε 10 Loop Unrolling (10 επαναλήψεις) και εντέλει θα εκτελεστεί 50 φορές και με δοκιμές ελέγξαμε ότι ο πιο αποδοτικός τρόπος είναι σε συνδυασμό με το Forwarding και το Branch Target Buffer.

```
loop:   beq r15, r3, exit
        ld r8, 0(r15)              #Load an Array's value in register
        ld r9, 8(r15)
        ld r10, 16(r15)
        ld r11, 24(r15)
        ld r12, 32(r15)
        ld r13, 40(r15)
        ld r14, 48(r15)
        ld r17, 56(r15)
        ld r18, 64(r15)
        ld r19, 72(r15)
        dadd r6, r6, r8
        dadd r6, r6, r9
        dadd r6, r6, r10
        dadd r6, r6, r11
        dadd r6, r6, r12
        dadd r6, r6, r13
        dadd r6, r6, r14
        dadd r6, r6, r17
        dadd r6, r6, r18
        dadd r6, r6, r19
        daddi r15, r15, 80
        j loop
```

Εικόνα 6: Τρίτη προσπάθεια κώδικα.

Τα αντίστοιχα αποτελέσματα φαίνονται και στην ακόλουθη εικόνα:



Εικόνα 7: Εικόνα τερματισμού από την 3η προσπάθεια υλοποίησης.

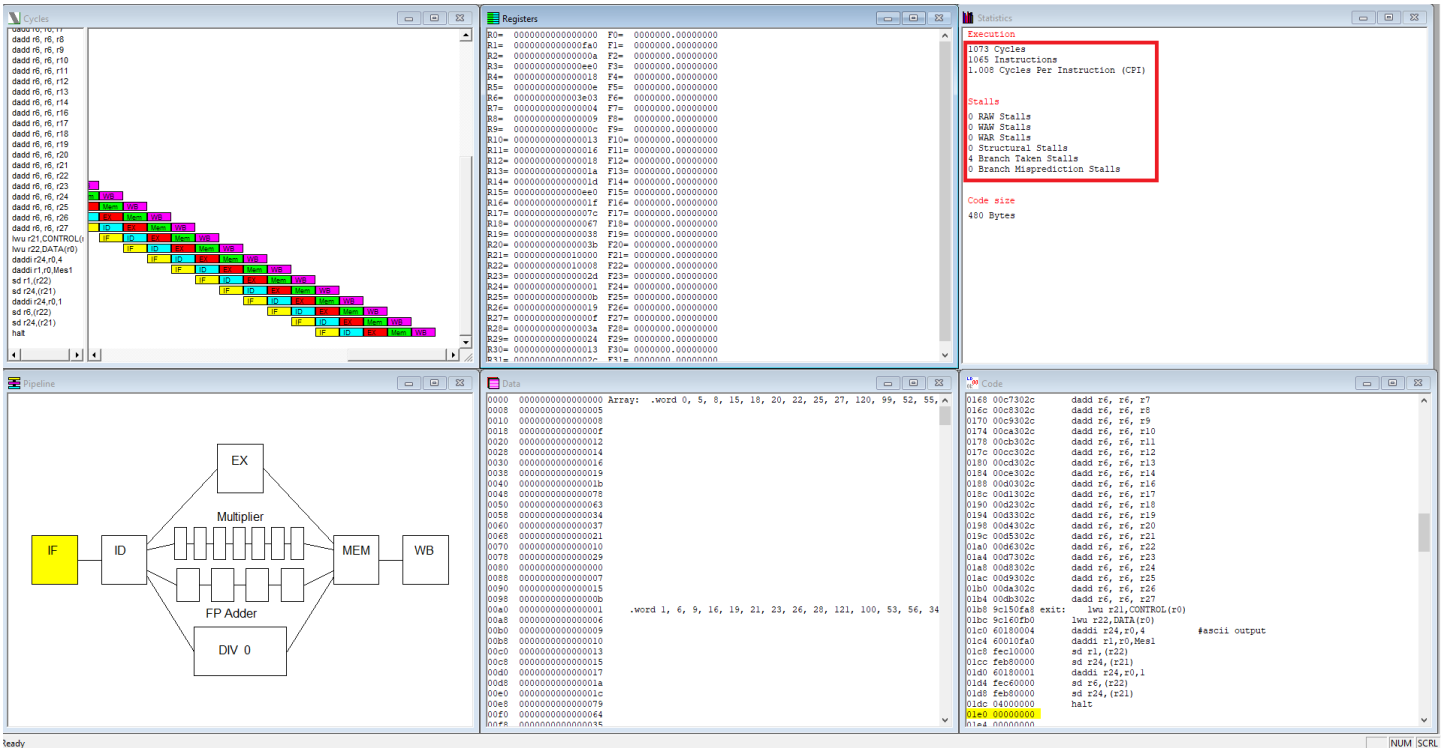
Σε αυτή την περίπτωση τα αποτελέσματα όπως βλέπουμε είναι τα εξής:

```
Forwarding and Branch Target Buffer active:
Execution:
1172 Cycles
1164 Instructions
1.007 Cycles Per Instruction (CPI)

Stalls:
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
4 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
144 bytes
```

Παρατηρούμε κατ' αντιστοιχία με την προηγούμενη περίπτωση, οι κύκλοι μειώθηκαν κατά 28% περίπου και φτάνουμε τους 1172 κύκλους με CPI 1.007.

4. Βελτιωμένη επαναληπτική μέθοδος – 24 Loop Unrolling

Στην τέταρτη προσπάθεια εκπόνησης του προγράμματος, η loop "τρώει" πάλι την περισσότερη πολυπλοκότητα. Ωστόσο μέσα στο loop κάναμε 28 Loop Unrolling (28 επαναλήψεις, χρησιμοποιώντας όλους τους διαθέσιμους registers) και μια χειροκίνητη επανάληψη εκτός του βρόχου, για να αθροίσουμε και τα 500 στοιχεία. Εντέλει θα εκτελεστεί 17 φορές και με δοκιμές ελέγξαμε ότι ο πιο αποδοτικός τρόπος είναι σε συνδυασμό με το Forwarding και το Branch Target Buffer. Ο κώδικας παρατίθεται στο αρχείο .s που είναι συνημμένο και είναι η τελική έκδοση.
Τα αντίστοιχα αποτελέσματα φαίνονται και στην ακόλουθη εικόνα:



Εικόνα 8: Εικόνα τερματισμού από την 4η προσπάθεια υλοποίησης.

Σε αυτή την περίπτωση τα αποτελέσματα όπως βλέπουμε είναι τα εξής:

```
Forwarding and Branch Target Buffer active:
Execution:
1073 Cycles
1065 Instructions
1.008 Cycles Per Instruction (CPI)

Stalls:
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
4 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
480 bytes
```

Παρατηρούμε κατ' αντιστοιχία με την προηγούμενη περίπτωση, οι κύκλοι μειώθηκαν κατά 9% περίπου και φτάνουμε τους 1073 κύκλους με CPI 1.008, παρά το γεγονός ότι αυξήθηκε το μέγεθος του προγράμματος, αυτή είναι η βέλτιστη επιλογή μέχρι στιγμής.

5. Διερεύνηση για βελτίωση κύκλων μέσω αναδρομής

Στην πέμπτη και τελευταία προσπάθεια εκπόνησης του προγράμματος, κατασκευάσαμε εκ νέου το πρόγραμμα με αναδρομή, δοκιμάσαμε να το τρέξουμε για μικρό μέγεθος πίνακα (τάξης μεγέθους 52 στοιχείων) και έτρεχε σωστά. Ωστόσο στην περίπτωση του δικού μας πίνακα, δηλαδή 500 στοιχείων, επειδή φορτώνουμε στο stack τα 500 αυτά στοιχεία, πέρα από τη μνήμη που είναι ορισμένος στατικά, δε χωράει στην υπάρχουσα μνήμη (με χωρητικότητα 2^{12} bytes), οπότε δεν μπορούμε να εξάγουμε τα αντίστοιχα συμπεράσματα. Ωστόσο, παραθέτουμε τον κώδικα, καθώς σαν φιλοσοφία είναι σωστός:

```
    .data

Array:  .word 0, 5, 8, 15, 18, 20, 22, 25, 27, 120, 99, 52, 55, 33, 16, 41, 0, 7, 21, 11
        .word 1, 6, 9, 16, 19, 21, 23, 26, 28, 121, 100, 53, 56, 34, 17, 42, 1, 8, 22, 12
        .word 2, 7, 10, 17, 20, 22, 24, 27, 29, 122, 101, 54

title:  .asciiz "Sum = "

CONTROL: .word32 0x10000
DATA:    .word32 0x10008

        .text

        daddi r15, r0, Array
        dadd r10,r0,r0
        lwu r21,CONTROL(r0)
        lwu r22,DATA(r0)
        daddi r24,r0,4      ; ascii output
        daddi r1,r0,title
        sd r1,(r22)
        sd r24,(r21)
        daddi r1, r15, 416

start:  daddi r29,r0,0x3c0   ; position a stack in data memory, use r29 as stack pointer
        jal sum
        daddi r24,r0,1      ; integer output
        sd r10,(r22)
        sd r24,(r21)
        halt

;
; parameter passed in r15 and r1, return value in r10
;

sum:    beq r15, r1,out      ; if reached end of table
        sd r31,(r29)
        daddi r29,r29,8   ; push return address onto stack


        ld r2, 0(r15)     ;
        daddi r15, r15, 8
        sd r2, (r29)
        daddi r29,r29,8   ; push r2 on stack

        jal sum     ; recurse...
        nop
        daddi r29,r29,-8
        ld r3,(r29)       ; pop n off the stack

        dadd r10,r10,r3    ; move sum r3 to r10

        daddi r29,r29,-8  ; pop return address
        ld r31,0(r29)
out:    jr r31
```

Εικόνα 9: Αναδρομικός κώδικας προγράμματος.

Συγκρίναμε τα αποτελέσματα με το αν τρέχαμε τον κώδικα αυτόν με την επιλογή Forwarding και ενεργό το Delay Slot και παρατηρήθηκαν 1064 κύκλοι με CPI 1.177. Τροποποιώντας τον αντίστοιχο κώδικα της 4ης προσπάθειας για να ελέγξουμε τυχόν χειρότερη ή καλύτερη πολυπλοκότητα, με ενεργό το Forwarding και το Branch Target Buffer, είδαμε ότι τρέχει μόνο σε 256 κύκλους με CPI 1.016. Άρα, με μικρότερα μεγέθη πίνακα όμως, παρατηρήθηκε και πάλι η καλύτερη πολυπλοκότητα σε κύκλους της 4ης προσπάθειας.

Σημείωση: Όπως έχει αναφερθεί και στο μάθημα, με τη χρήση αναδρομικού προγράμματος λόγω bug στο WinMIPS64 ΔΕΝ πρέπει να βάλουμε Branch Target Buffer, καθώς εκτελείται λανθασμένα το αναδρομικό πρόγραμμα και μπαίνει σε ατέρμονο loop. Το nop, μετά την εντολή "jal sum" έχει τοποθετηθεί για να αποθηκεύονται σωστά οι τιμές του πίνακα στη μνήμη (αλλιώς έχουμε overwrite).