Αυτό το άρθρο είναι συνέχεια του μέρους 1ου της σειράς Συναρτήσεις σε Assembly.

Μία συνάρτηση που ακολουθεί cdecl (C declaration) calling convention διακρίνεται για δύο λόγους. Πρώτον, οι παράμετροι και η διεύθυνση μνήμης του επόμενου instruction περνάνε στην συνάρτηση μέσω της στοίβας, από την συνάρτηση που την κάλεσε. Αυτό σημαίνει πως όλες οι παράμετροι θα πρέπει να έχουν ήδη τοποθετηθεί πριν κληθεί η συνάρτηση, πάνω στην στοίβα. Δεύτερον η συνάρτηση που κάνει την κλήση είναι υπεύθυνη για την εκκαθάριση της στοίβας από τις παραμέτρους. Η λογική πίσω από το cdecl θα μπορούσε να περιγραφεί με τα παρακάτω βήματα:

1. Σπρώχνουμε/τοποθετούμε τις παραμέτρους στην κορυφή της στοίβας με αντίθετη σειρά από αυτήν που θέλουμε να τις δηλώσουμε. Για παράδειγμα αν η συνάρτησή μας είχε πρωτότυπο στην C το

```
void my_function (int param1, int param2, int param3);
```

τότε θα κάναμε push τις παραμέτρους στην στοίβα, ξεκινώντας πρώτα από το param3 και ανεβαίνοντας μέχρι να φτάσουμε και στο param1. Αν θέλαμε να καλέσουμε την παραπάνω συνάρτηση με τιμές παραμέτρων `param1 = 14`, `param2 = 2222` και `param3 = 329` τότε θα κάναμε αυτό πρώτα:

```
pushl $329
pushl $2222
pushl $14
```

Το σύμβολο $ χρησιμοποιείται για immediate addressing. Αν δεν το βάζαμε τότε ο υπολογιστής αντί να έσπρωχνε την τιμή 329 στην στοίβα, θα προσπαθούσε να σπρώξει την τιμή που βρίσκεται **στην θέση μνήμης** υπ' αριθμόν 329. Αυτό το λάθος συχνά καταλήγει σε segmentation fault μιας και δεν έχουμε δικαιώματα να διαβάσουμε οποιαδήποτε θέση μνήμης μας καπνίσει, μέσα από το πρόγραμμά μας, πέρα από τις θέσεις μνήμης που ανήκουν στο ίδιο το πρόγραμμα.

*Η σειρά με την οποία βάζουμε τις παραμέτρους δεν παίζει τόσο μεγάλο ρόλο στην assembly (αρκεί να θυμόμαστε που βρίσκεται η κάθε παράμετρος στην στοίβα), ωστόσο είναι σημαντική για την C, καθώς οι συναρτήσεις που υλοποιούμε σε αυτήν πρέπει να μπορούν να διατηρούν ένα σταθερό ABI. Αυτό σημαίνει πως αν, κάθε φορά που κάνουμε compile μία βιβλιοθήκη που έχουμε γράψει στην C/C++ σε γλώσσα μηχανής, ο compiler έβαζε τις παραμέτρους στην στοίβα με διαφορετική σειρά τότε θα έσπαγε το ABI compatibility. Ως συνέπεια, θα έπρεπε να κάνουμε ξανά compile και linking από το 0 κάθε πρόγραμμα που χρησιμοποιούσε αυτήν την βιβλιοθήκη, αν κάναμε ξανά compile και την βιβλιοθήκη την ίδια (ακόμα και αν δεν αλλάζαμε τίποτα εμείς στον κώδικα, αρκεί ο compiler να μην διατηρούσε σταθερή την σειρά των παραμέτρων στην στοίβα).

2. Καλούμε το `call` instruction με operand το όνομα της συνάρτησης. Η εντολή `call` κάνει 2 βασικές λειτουργίες. Αρχικά, σπρώχνει στην κορυφή της στοίβας την διεύθυνση του επόμενου instruction στον κώδικά μας. Με αυτόν τον τρόπο αποθηκεύουμε την θέση μνήμης στην οποία θα μπορεί να επιστρέψει η συνάρτησή μας όταν τελειώσει την εκτέλεσή της. Στην συνέχεια μεταφέρει την ροή του προγράμματος ώστε η επόμενη εντολή που θα εκτελεστεί να είναι η πρώτη εντολή μέσα στην συνάρτηση. Αυτό το κάνει αλλάζοντας την τιμή του eip register (instruction pointer) ο οποίος δείχνει κάθε φορά στην θέση μνήμης της επόμενης εντολής, που είναι να εκτελεστεί.

Μέχρι τώρα, συνεχίζοντας από το παραπάνω παράδειγμα κλήσης της my_function, έχουμε:

```
pushl $329
pushl $2222
pushl $14
call my_function
```

3. Όταν βρεθούμε μέσα στην συνάρτηση, αποθηκεύουμε την τωρινή τιμή του ebp register (base pointer) βάζοντάς την στην στοίβα και κάνουμε τον ebp να δείχνει στην κορυφή της στοίβας. Ο ebp είναι ένας ειδικός register που θα χρησιμοποιούμε σαν offset για να διαβάζουμε παραμέτρους και τοπικές μεταβλητές τις συνάρτησης. Μετά από αυτό, αντιγράφουμε την διεύθυνση που δείχνει ο esp (stack pointer) μέσα στον ebp. Έτσι δημιουργούμε και το stack frame μας. Το stack frame αποτελείται από τις παραμέτρους και τις τοπικές μεταβλητές της συνάρτησης, καθώς και την διεύθυνση της μνήμης του instruction στο οποίο θα επιστρέψει η συνάρτηση και θα συνεχίσει να εκτελείται το πρόγραμμα. Τις παραμέτρους και τις τοπικές μεταβλητές, τις ελέγχουμε μέσω offsets του ebp register. Δεν μπορούμε να χρησιμοποιήσουμε τον esp register γι’ αυτόν τον σκοπό, γιατί ο esp μπορεί να αλλάξει κατά την διάρκεια της συνάρτησης, αν προσθέτουμε ή αφαιρούμε δεδομένα από στοίβα, ενώ ο ebp θα δείχνει πάντα στην αρχική τιμή που έδειχνε ο esp όταν μπήκαμε στην συνάρτηση.

4. Το επόμενο βήμα είναι προαιρετικό διότι αναφέρεται στην δημιουργία τοπικών μεταβλητών (κάποιες συναρτήσεις μπορούν να μην χρησιμοποιούν τοπικές μεταβλητές). Για να φτιάξουμε τοπικές μεταβλητές, απλά κάνουμε χώρο στην στοίβα αφαιρώντας τον αριθμό των bytes που θέλουμε από τον esp. Ας μην ξεχνάμε, πως η στοίβα στις αρχιτεκτονικές x86 μεγαλώνει προς τα κάτω, δηλαδή όσο πιο πολλές τιμές έχει τόσο πιο πολύ μικραίνει η διεύθυνση μνήμης της. Άρα για να κάνουμε περισσότερο χώρο στην στοίβα αφαιρούμε από την κορυφή της στοίβας τον αριθμό των bytes που πιάνουν τα δεδομένα, που θέλουμε να αποθηκεύσουμε σε αυτήν, και με τα κατάλληλα offset από τον ebp έχουμε τις τοπικές μεταβλητές.

5. Αφού η συνάρτησή μας έχει τελειώσει με την δουλεία της, επαναφέρει την στοίβα στην κατάσταση που βρισκόταν πριν κληθεί (η συνάρτηση), αποθηκεύει την τιμή επιστροφής στον eax register και επιστρέφει στο επόμενο instruction που βρίσκεται

μετά το `call`, στην συνάρτηση που την κάλεσε. Ο eax χρησιμοποιείται για την αποθήκευση της τιμής επιστροφής, όπου θα μπορούμε και να διαβάσουμε μόλις η συνάρτησή μας επιστρέψει. Η επαναφορά της στοίβας γίνεται απλά αν κάνουμε τον esp να δείχνει στην θέση μνήμης που βρισκόταν όταν άρχισε η συνάρτηση, δηλαδή στον ebp (στην αρχή του stack frame μας) και τον ebp να δείχνει στην τιμή που είχε όταν άρχισε η συνάρτηση, δηλαδή στην τιμή που βρίσκεται στην κορυφή της στοίβας (κάνοντας ένα `popl %ebp`). Η επιστροφή από την συνάρτηση γίνεται με την εντολή ret η οποία κάνει pop την τιμή που βρίσκεται στην κορυφή της στοίβας (η οποία αν θυμάστε καλά, ήταν η διεύθυνση του επόμενου instruction όπως την έκανε push το instruction call) και την αποθηκεύει στον eip register ο οποίος λέει στον επεξεργαστή ποια εντολή να εκτελέσει αμέσως μετά.

6. Τελευταίο βήμα είναι να καθαρίσουμε την στοίβα από την συνάρτηση που έγινε η κλήση. Αυτό πολύ απλά γίνεται αν κάνουμε τόσες φορές pop, όσες κάναμε push για να βάλουμε τις παραμέτρους μας μέσα στην στοίβα ώστε να περαστούν στην συνάρτηση.

Από το παραπάνω παράδειγμα δηλαδή, μία ολοκληρωμένη κλήση της my_function φαίνεται κάπως έτσι:

```
pushl $329
pushl $2222
pushl $14
call my_function
popl %edx
popl %edx
popl %edx
```

Εδώ χρησιμοποιούμε τυχαία τον register edx, απλά και μόνο επειδή δεν μπορούμε να καλέσουμε την `popl` χωρίς operand. Αντίστοιχα θα μπορούσαμε να κάνουμε το εξής:

```
pushl $329
pushl $2222
pushl $14
call my_function
addl $12, %esp
```

για να αφαιρέσουμε από την στοίβα τα 12 bytes (3 φορές που κάναμε `pushl` * 4 bytes που πιάνει κάθε αριθμός long) που προσθέσαμε. Αυτός ο κώδικας έχει το ίδιο αποτέλεσμα με τον παραπάνω, με την διαφορά πως δεν αποθηκεύονται πουθενά οι τιμές που βγάζουμε από την στοίβα.

Όλα αυτά ήταν σε πολύ θεωρητικό επίπεδο. Τώρα θα δούμε και θα αναλύσουμε ένα πολύ βασικό παράδειγμα προγράμματος assembly που υλοποιεί μία συνάρτηση η οποία υπολογίζει το μέγιστο μεταξύ δύο προσημασμένων ακεραίων και το επιστρέφει στην συνάρτηση που την κάλεσε.

```
.section .data
.section .text
.globl _start

_start:
    pushl $14
    pushl $29
    call get_max_number
    popl %edx
    popl %edx

    movl %eax, %ebx
    movl $1, %eax

    int $0x80

.type get_max_number, @function
get_max_number:
    pushl %ebp
    movl %esp, %ebp

    .equ first_number, 8
    .equ second_number, 12

    movl first_number(%ebp), %ebx

    cmpl %ebx, second_number(%ebp)
    jg second_number_is_max

    movl first_number(%ebp), %eax
    jmp return_from_function

    second_number_is_max:
    movl second_number(%ebp), %eax

    return_from_function:
    movl %ebp, %esp
    popl %ebp
    ret
```

Ξεκινάμε με την πρώτη πρώτη γραμμή, το `.section .data`. Δεν χρησιμοποιούμε αυτό το section στο πρόγραμμά μας, άρα το αφήνουμε κενό. Ακολουθεί το text segment/section στο οποίο βρίσκονται οι εντολές του προγράμματος:

```
.section .text
.globl _start
```

```
_start:
```

Το `_start` label ορίζει την θέση μνήμης εισόδου του προγράμματος ώστε να γνωρίζει το λειτουργικό από ποια εντολή να ξεκινήσει να εκτελεί το πρόγραμμά μας. Στην C έχουμε την main() που κάνει την αντίστοιχη δουλειά. Σε άλλες γλώσσες έχουμε παρόμοιες συναρτήσεις που συμβολίζουν το entry point του προγράμματος. Αν συνδέαμε το πρόγραμμά μας με την C standard library (στάνταρ βιβλιοθήκη της C) τότε θα έπρεπε να βάζαμε το main label στην θέση του `_start`. Ωστόσο σε αυτό το πρόγραμμά μας δεν χρησιμοποιούμε συναρτήσεις από την βιβλιοθήκη της C άρα βάζουμε το προεπιλεγμένο label για entry point προγράμματος assembly, που είναι το `_start`.

Στις επόμενες γραμμές γίνεται κλήση της συνάρτησης `get_max_number` που έχουμε δηλώσει πιο κάτω. Για να καταλάβουμε καλύτερα πως λειτουργεί το cdecl θα ακολουθήσουμε την στοίβα όσο προχωράει το πρόγραμμα. Αρχικά η στοίβα μας είναι κενή και ο esp απλά δείχνει σε μία θέση μνήμης που δεν περιέχει τίποτα μέσα.

```
__ <= (%esp)
```

Οι παρενθέσεις είναι σύνταξη AT&T για indirect memory addressing. Αφού ο esp έχει αποθηκευμένη μέσα του μία θέση μνήμης (την κορυφή της στοίβας κάθε φορά) εμάς μας ενδιαφέρουν τα δεδομένα που βρίσκονται σε αυτήν την θέση, και όχι η θέση η ίδια. Ας μην ξεχνάμε πως ο esp είναι stack **pointer** και άρα δρα σαν δείχτης που δείχνει στην θέση μνήμης της κορυφή της στοίβας, δεν δρα σαν απλή μεταβλητή στην οποία αποθηκεύονται τα δεδομένα της κορυφής κάθε φορά. Χωρίς τις παρενθέσεις αναφερόμαστε απευθείας στην τιμή που έχει αποθηκευμένος ο οποιοσδήποτε register και στην προκειμένη περίπτωση απλά θα βλέπαμε μία θέση μνήμης που δεν μας ενδιαφέρει.

Η συνάρτησή μας, παίρνει ως ορίσματα 2 ακέραιους αριθμούς και επιστρέφει τον μεγαλύτερο από τους δύο. Άρα θα πρέπει να βάλουμε δύο τιμές στην στοίβα πριν καλέσουμε την συνάρτηση, αλλιώς θα έχουμε πρόβλημα όταν η συνάρτησή μας προσπαθήσει να διαβάσει τις παραμέτρους. Οι τιμές που χρησιμοποιούμε στο συγκεκριμένο πρόγραμμα είναι οι αριθμοί 14 και 29.

```
pushl $14
pushl $29
```

Η στοίβα μας μετά από αυτά τα δύο instructions μοιάζει κάπως έτσι:

```
παράμετρος 29  <= (%esp)
παράμετρος 14
__
```

Βλέπουμε πως το 29 είναι στην κορυφή, αφού το βάλαμε τελευταίο. Θυμηθείτε το παράδειγμα με την στοίβα πιάτων κατά την οποία μπορούμε είτε να αφαιρούμε πιάτα είτε να τα προσθέτουμε στην κορυφή της. Τα «πιάτα» σε αυτήν την αναλογία είναι οι αριθμοί μας.

Στην συνέχεια γίνεται η κλήση της συνάρτησης:

```
call get_max_number
```

Θυμηθείτε από προηγουμένως που είπαμε πως το instruction call κάνει δύο πράματα. Αρχικά κάνει `pushl` την θέση μνήμης της επόμενης εντολής στην στοίβα, που στο παράδειγμά μας είναι η

```
popl %edx
```

Αυτό το κάνει για να γνωρίζει η συνάρτηση που θα καλέσουμε, σε ποια θέση μνήμης να επιστρέψει και να συνεχίσει η εκτέλεση του προγράμματος. Άρα μετά την call η στοίβα μας έχει την εξής αναπαράσταση:

```
return address  <= (%esp)
παράμετρος 29
παράμετρος 14
__
```

όπου return address είναι η θέση μνήμης του instruction που ακολουθεί την `call`. Δεύτερη λειτουργία που εκτελεί η `call` είναι να αλλάξει την τιμή του eip register ώστε να δείχνει στην αρχή της συνάρτησης. Με άλλα λόγια μεταφέρει την ροή του προγράμματός μας στην αρχή του `get_max_number label`. Σημαντικό είναι να θυμόμαστε πως τα labels στην assembly είναι στην ουσία θέσεις μνήμης στις οποίες δίνουμε συμβολικά ονόματα για να τις θυμόμαστε πιο εύκολα. Άρα και τα ονόματα συναρτήσεων είναι στην ουσία οι θέσεις μνήμης της πρώτης εντολής της συνάρτησης. Περισσότερα γι’ αυτό λέω στο άρθρο μου για τους function pointers.

Η εκτέλεση του προγράμματός μας συνεχίζει από το πρώτο instruction που ακολουθεί το label της συνάρτησης:

```
pushl %ebp
movl %esp, %ebp
```

Αυτή η ακολουθία από instructions είναι στάνταρ στο cdecl, όπως αναφέραμε πιο πάνω. Στην ουσία σπρώχνουμε τον base pointer register πάνω στην στοίβα για να κατασκευάσουμε το stack frame μας:

```
old %ebp value  <= (%esp)
return address
```

```
παράμετρος 29
παράμετρος 14
__
```

Και αντιγράφουμε την θέση μνήμης της κορυφής της στοίβας στον ebp ώστε να έχουμε το εξής αποτέλεσμα:

```
old %ebp value  <= (%esp) και (%ebp)
return address
παράμετρος 29
παράμετρος 14
__
```

Από εδώ και πέρα μπορούμε να αναφερόμαστε σε οποιεσδήποτε μεταβλητές της συνάρτησής μας, ως offset του ebp. Δηλαδή:

```
old %ebp value      <= (%esp) και (%ebp)
return address      <= 4(%esp) και 4(%ebp)
παράμετρος 29       <= 8(%esp) και 8(%ebp)
παράμετρος 14       <= 12(%esp) και 12(%ebp)
__                  <= 16(%esp) και 16(%ebp)
```

Βλέπουμε πως θα μπορούσαμε να χρησιμοποιήσουμε και τον esp register για αυτόν τον σκοπό. Ωστόσο, αν θελήσουμε να προσθέσουμε τοπικές μεταβλητές στην συνάρτηση (πχ αφαιρώντας bytes από την κορυφή της στοίβας, τον esp δηλαδή) τότε η τιμή του esp θα αλλάξει και άρα θα αλλάξουν και τα offsets, ενώ η τιμή του ebp εγγυημένα παραμένει σταθερή καθ’ όλη τη διάρκεια που εκτελείται η συνάρτησή μας. Για παράδειγμα αν είχαμε δύο τοπικές μεταβλητές τότε η στοίβα μας θα έμοιαζε κάπως έτσι

```
τοπική μεταβλητή 1  <= (%esp) και -8(%ebp)
τοπική μεταβλητή 2  <= 4(%esp) και -4(%ebp)
old %ebp value      <= 8(%esp) και (%ebp)
return address      <= 12(%esp) και 4(%ebp)
παράμετρος 29       <= 16(%esp) και 8(%ebp)
παράμετρος 14       <= 20(%esp) και 12(%ebp)
__                  <= 24(%esp) και 16(%ebp)
```

Βλέπουμε πως τα offsets του esp αλλάζουν, καθώς αλλάζει και η τιμή του, αλλά ο ebp παραμένει σταθερός.

Η συνέχεια στο επόμενο μέρος.... stay tuned