ΗΡΥ 201– Ψηφιακοί Υπολογιστές

Παράδειγμα συμβάσεων χρήσης καταχωρητών MIPS και χρήσης στοίβας

ΔΙΟΝΥΣΗΣ ΠΝΕΥΜΑΤΙΚΑΤΟΣ

Παράδειγμα χρήσης καταχωρητών

Εστω οι παρακάτω δύο συναρτήσεις A και B:

```
int A(int x) {
   int y;

   y = B(x);
   y = y + x;

   return y;
}

int B(int arg) {
   return arg + 5;
}
```

Οι συμβάσεις χρήσης καταχωρητών του MIPS ορίζουν ότι το X στην A (και στην B) θα πρέπει να περαστεί στον $a0.

Επίσης ορίζουν ότι η τιμή επιστροφής θα πρέπει να μπει στον $v0.

Ακόμα η διεύθυνση επιστροφής της συνάρτησης βρίσκεται στον καταχωρητή $ra

Προσπάθεια Πρώτη

Η Β είναι απλή: το arg υπάρχει ήδη στον καταχωρητή $a0, οπότε το αυξάνουμε κατά 1, και γράφουμε το αποτέλεσμα στον $v0

Στην Α, η παράμετρος εισόδου x βρίσκεται στον $a0, οπότε για να περαστεί στην Β βάλαμε την γραμμή που αντιγράφει το $a0 στο $a0, το οποίο προφανώς είναι περιττό

Κατόπιν, καλούμε την Β η οποία επιστρέφει το αποτέλεσμα της στον $v0

Ακολούθως προσθέτουμε το αποτέλεσμα με το $a0 και βάζουμε το αποτέλεσμα στον $v0

Η κλήση Α(10) καλεί την Β(10) η οποία επιστρέφει το 15, η Α προσθέτει το 10 και επιστρέφει 25

Ο κώδικας είναι φαινομενικά σωστός

```
A:
        mov $a0, $a0 // περιττό!
        jal B
        add $v0, $v0, $a0
        jr $ra

B:
        addi $v0, $a0, 5
        jr $ra
```

Τι θα γίνει αν αλλάξω λίγο την Β;

Προσπάθεια Δεύτερη

Η Β άλλαξε λίγο: πρώτα γίνεται η αύξηση στον $a0 και μετά η αντιγραφή του αποτελέσματος στον $v0.

Ο κώδικας της Β είναι σωστός!

Αλλά υπάρχουν προβλήματα:

1) Η κλήση Α(10) καλεί την Β(10) η οποία επιστρέφει το 15, η Α προσθέτει το **15** και υπολογίζει το 30 το οποίο και επιστρέφει η Α.

2) Το ίδιο συμβαίνει και για τον $ra! Το $ra στην Α δείχνει π.χ. στην main από όπου καλέστηκε. Μετά την κλήση Β αλλάζει ώστε να δείχνει στην 3η εντολή της Α. Η Β επιστρέφει σωστά, αλλά το jr $ra της Α **δεν** επιστρέφει στην main αλλά στην ίδια την Α.

```
A:    mov $a0, $a0 // περιττό!
      jal B
      add $v0, $v0, $a0
      jr $ra

B:    addi $a0, $a0, 5
      mov   $v0, $a0
      jr $ra
```

Προσπάθεια Τρίτη

Η Α δημιουργεί ένα αντίγραφο του $a0 σε ένα **μπλε κουτάκι**. Κατόπιν καλεί την Β η οποία αλλάζει την τιμή του $a0 και επιστρέφει το αποτέλεσμα της (15 στην περίπτωση της κλήσης Α(10)) στον $v0 και επιστρέφει στην Α.

Η Α επαναφέρει την παλιά τιμή του $a0 (10) από το **μπλε κουτάκι**, και τώρα η πρόσθεση είναι σωστή και υπολογίζει 15 + 10 = 25 το οποίο και γράφεται στον $v0.

Ο κώδικας της Α υπολογίζει πλέον το σωστό αποτέλεσμα ανεξάρτητα πως θα γραφτεί η Β!

Το SAVE γίνεται πριν την κλήση της συνάρτησης που φοβόμαστε

Το RESTORE γίνεται μετά την κλήση και πριν την χρήση του καταχωρητή που χρειαζόμαστε

```
A:    SAVE $a0, LOCATION
      jal  B
      RESTORE $a0, LOCATION
      add $v0, $v0, $a0
      jr $ra

B:    addi $a0, $a0, 5
      mov   $v0, $a0
      jr $ra
```

Προσπάθεια Τρίτη++

Το ίδιο πρέπει να γίνει και για τον $ra!!!

Σε ένα άλλο **πορτοκαλί κουτάκι** η Α δημιουργεί ένα αντίγραφο του $ra

Η Α καλεί όποια συνάρτηση χρειάζεται (την Β στο συγκεκριμένο παράδειγμα)

Η Α επαναφέρει την παλιά τιμή του $ra από το **πορτοκαλί κουτάκι**, και τώρα η διεύθυνση επιστροφής είναι σωστή και δείχνει σε όποιον κάλεσε την Α.

Τώρα πια ο κώδικας της Α λειτουργεί σωστά υπολογίζει το σωστό αποτέλεσμα ανεξάρτητα πως θα γραφτεί η Β και επιστρέφει σωστά σε όποιον την κάλεσε!

```
A:    SAVE $ra, LOCATION-ra
      SAVE $a0, LOCATION
      jal  B
      RESTORE $a0, LOCATION
      add $v0, $v0, $a0
      RESTORE $ra, LOCATION-ra
      jr $ra

B:    addi $a0, $a0, 5
      mov    $v0, $a0
      jr $ra
```

Εξήγηση 3ης Προσπάθειας

Η Α δεν γνωρίζει αν η Β όπως έχει γραφτεί θα χρησιμοποιήσει ή όχι κάποιους καταχωρητές. Δεν υπάρχει καμμία εγγύηση για κάτι τέτοιο. Όπως και να γράψω την Β είναι σωστό.

Συνεπώς, η Α θεωρεί ότι οτιδήποτε χρήσιμο για την ίδια (ο $a0) μπορεί να καταστραφεί από την Β.

Ο $ra είναι άλλη περίπτωση: ανεξάρτητα πως θα γράψω την Β (ακόμα και για κενή Β), το $ra πανωγράφεται από την ίδια την Α όταν εκτελείται η εντολή jal Β!

```
A:  SAVE $ra, LOCATION-ra
    SAVE $a0, LOCATION
    jal  B
    RESTORE $a0, LOCATION
    add $v0, $v0, $a0
    RESTORE $ra, LOCATION-ra
    jr $ra

B:  addi $a0, $a0, 5
    mov    $v0, $a0
    jr $ra
```

Γενικός Τρόπος γραφής κώδικα

Πρόλογος

```
A:    SAVE $ra, LOCATION-ra
```

Σώμα

```
      SAVE $a0, LOCATION
      jal  B
      RESTORE $a0, LOCATION
      add $v0, $v0, $a0
```

Επίλογος

```
      RESTORE $ra, LOCATION-ra
      jr $ra
```

```
B:                                  Πρόλογος
      addi $a0, $a0, 5              Σώμα
      mov    $v0, $a0
      jr $ra                        Επίλογος
```

Κάθε συνάρτηση έχει Πρόλογο, Σώμα, Επίλογο.Ο πρόλογος μπορεί να είναι κενός (όπως στην περίπτωση της B).

Γράφουμε ***πρώτα*** το Σώμα, και ***μετά*** γεμίζουμε Πρόλογο και Επίλογο (όταν ξέρουμε τι χρειαζόμαστε όπως θα δούμε παρακάτω).

Πρέπει να συζητήσουμε:

(α) γιατί το Μπλέ και το Πορτοκαλί κουτάκι είναι διαφορετικά;

(β) που βρίσκονται τα κουτάκια αυτά;

(γ) Υπάρχουν άλλες επιλογές για να γράψω κώδικα;

Συζήτηση

Δυο τύποι SAVE/RESTORE

- Αυτά που αφορούν πρόλογο/επίλογο (μόνο το $ra στο προηγούμενο παράδειγμα)
- Τα άλλα που αφορούν καταχωρητές και τιμές που χρησιμοποιώ στο σώμα συνάρτησης

Που βρίσκονται τα κουτάκια αυτά; Πως γίνεται το SAVE/RESTORE;

- Ο χώρος θα πρέπει να είναι σε ιδιωτικό για την κάθε συνάρτηση μέρος
- Σε περίπτωση αναδρομικής συνάρτησης θα πρέπει να έχω πολλές τέτοιες θέσεις (μία για κάθε ενεργοποίηση της συνάρτησης)
- Συνεπώς θα πρέπει να είναι στην στοίβα. Οπότε SAVE = Push, Restore = Pop

Στοίβα συστήματος στον MIPS

- Υλοποιείται με τον $sp (η στοίβα μεγαλώνει προς τα κάτω)
- Push($a0) => ακολουθία: addui $sp, $sp, -4 ; sw $a0, 0($sp)
- Pop($a0) => ακολουθία: lw $a0, 0($sp) ; addui $sp, $sp, 4

Ο τελικός κώδικας της Α

Χρειαζόμαστε 2 λέξεις στην στοίβα

```
A:  addui  $sp, $sp, -8
Πρόλογος
    sw     $ra, 0($sp) //push
```

Στο offset 0 από τον $sp θα αποθηκευτεί ο $ra και σε offset 4 ο $a0

Η πρώτη εντολή του **Προλόγου** (addui) δημιουργεί τον χώρο κουνώντας τον $sp προς τα κάτω

```
Σώμα
    sw     $a0, 4($sp) //push
    jal    B
    lw     $a0, 4($sp) //pop
    add    $v0, $v0, $a0
```

Κατόπιν μπορούμε αν γράψουμε τον $ra υλοποιώντας το δεύτερο μισό της push

Ανάλογα αποθηκεύεται και ο $a0

Η επαναφορά του $a0 είναι απλά ένα lw

```
Επίλογος
    lw     $ra, 0($sp) //pop
    addui  $sp, $sp, 8
    jr     $ra
```

Η επαναφορά του $sp στην αρχική του θέση γίνεται στο τέλος του **Επιλόγου** αμέσως πριν το jr $ra.

Μια άλλη προσέγγιση;

Τι αν η Β εγγυόταν ότι **δεν θα άλλαζε** κανένα καταχωρητή;

Η Β σώζει και επαναφέρει τον $a0, οπότε η Α *δεν βλέπει αλλαγή*!

Ομοίως και η Α πρέπει να δώσει την ίδια εγγύηση σε όποιον την καλεί! Αλλά δεν χρησιμοποιεί άλλον καταχωρητή, οπότε είναι OK.

Η τεχνική **αυτή** λέγεται ***callee-save***, διότι η καλούμενη συνάρτηση έχει την υποχρέωση να κάνει όλη την δουλειά.

Η **προηγούμενη** τεχνική λέγεται ***caller-save*** διότι η συνάρτηση που καλεί πρέπει να σώσει τις ενδιαφέρουσες τιμές

Τα **πορτοκαλί** τί είναι; (Callee-save!)

```
B:  addui   $sp, $sp, -4
    sw      $a0,  0($sp) //push

    addi $a0, $a0, 5
    mov     $v0, $a0

    lw      $a0, 0($sp) //pop
    addui   $sp, $sp, 4
    jr      $ra
```

```
A:  addui   $sp, $sp, -4
    sw      $ra, 0($sp) //push

    jal     B
    add     $v0, $v0, $a0

    lw      $ra, 0($sp) //pop
    addui   $sp, $sp, 4
    jr      $ra
```

Caller/Callee Save

Callee-Save

- Η κάθε συνάρτηση *εγγυάται ότι δεν θα αλλάξει κανένα καταχωρητή* (εκτός π.χ. του $v0)
- Απλούστερο:
  - Στον πρόλογο σώζω όλους τους καταχωρητές που χρησιμοποιώ
  - Στον επίλογο επαναφέρω την παλιά τιμή τους

Caller-Save

- Οι συναρτήσεις *δεν εγγυόνται τίποτα* σε αυτόν που τις καλεί
- Πλήρης ελευθερία χρήσης καταχωρητων
- Είναι πιο αποτελεσματικό (;)
- Πρέπει να σώζω μόνο ότι επηρρεάζεται από κλήσεις συναρτήσεων => όχι όλους τους καταχωρητές.
  - Εντός του σώματος της συνάρτησης

Caller/Callee Save

Πότε συμφέρει το ένα και πότε το άλλο;

- Το να εγγυόμαι στους άλλους ότι δεν θα αλλάξω τίποτα είναι σημαντική ευθύνη
- Το να μην έχω εγγυήσεις κάνει την ζωή μου πιο δύσκολη.

Ιδανικά;

- Υπάρχουν περιπτώσεις που σε προσέγγιση caller-save δεν χρειάζεται να σώσω τους καταχωρητές που χρησιμοποιώ
- Όταν η «ζωή» της τιμής που βρίσκεται στον καταχωρητή δεν διασταυρώνεται με κλήση συνάρτησης:

```
addi   $t3, $a2, 4
xor    $a0, $t3, $a1
jal    B
…
addi   $t3, $v0, 5
```

Η τιμή στον $t3 δεν χρειάζεται μετά την κλήση της B, οπότε **δεν χρειάζεται save/ restore**

Το καλύτερο και από τις δύο προσεγγίσεις;;;

Οι συμβάσεις του MIPS χωρίζουν τους καταχωρητές σε δύο ομάδες

- **Callee-save** ($s0-$s7)
- **Caller-save** ($t0-$t9)

Η κάθε συνάρτηση δίνει εγγύηση ότι **δεν θα αλλάξει** τους $s0-$s7

Η κάθε συνάρτηση **μπορεί να αλλάξει** τους υπόλοιπους καταχωρητές: $t0-$t7, $a0-$a3, $v0, $v1, $at, $ra

Απλή στρατηγική χρήσης:

- Για καταχωρητή που η ζωή του «διασταυρώνεται» με κλήση συνάρτησης ➔ χρήση $s_ (με save/restore σε πρόλογο/επίλογο)
- Για καταχωρητή που χρησιμοποιείται χωρίς να «διασταυρώνεται» με κλήση συνάρτησης ➔ χρήση $t_ (με save/restore σε πρόλογο/επίλογο)

Η Α με χρήση Calee-save καταχωρητών

Χρειαζόμαστε 2 λέξεις στην στοίβα

```
A:  addui  $sp, $sp, -8
    sw     $ra, 0($sp) //push
    sw     $s0, 4($sp) //push
```

Στο offset 0 από τον $sp θα αποθηκευτεί ο $ra και σε offset 4 ο $s0

Στον **Πρόλογο** δημιουργώ χώρο και αποθηκεύω $ra και $s0

```
    move   $s0, $a0
    jal    B
    add    $v0, $v0, $s0
```

Αρχικά στο σώμα δημιουργώ αντίγραφο του $a0 στον $s0 και στην συνέχεια χρησιμοποιώ τον $s0 για να αναφερθώ στο όρισμα X της Α

```
    lw     $s0, 4($sp) //pop
    lw     $ra, 0($sp) //pop
    addui  $sp, $sp, 8
    jr     $ra
```

Στον **Επίλογο** γίνεται η επαναφορά των $s0 και $ra και η επαναφορά του $sp στην αρχική του θέση

Πρόλογος

Σώμα

Επίλογος

Η Aloop

```
int Aloop(int X, int N) {
   int y = 0, i;

   for (i=N; i>0; i--) {
      y = B(X) + y;
   }
   return y;
}
```

Η Aloop καλεί Ν φορές την Β(Χ) και αθροίζει το αποτέλεσμα.

--δεν εξετάζουμε αν η Aloop έχει έννοια ή μπορεί να γραφτεί αλλιώς—

Η «ζωή» της y ***εκτείνεται*** πέρα της κλήσης της Β.

Η «ζωή» της Χ ***εκτείνεται*** πέρα της κλήσης της Β.

Η «ζωή» της i ***εκτείνεται*** πέρα της κλήσης της Β.

Η Aloop με Callee-save καταχωρητές

```
Aloop:addui  $sp, $sp, -16
      sw     $ra, 0($sp) //push
      sw     $s0, 4($sp) //push
      sw     $s1, 8($sp) //push
      sw     $s2, 12($sp) //push

      move   $s0, $a0   // X
      move   $s1, $a1   // I = N
      move   $s2, $zero // y
Loop: move   $a0, $s0 // B's arg
      jal    B
      add    $s2, $v0, $s2
      addi   $s1, $s1, -1
      bnez   $s1, Loop
      move   $v0, $s2

      lw     $s2, 12($sp) //pop
      lw     $s1,  8($sp) //pop
      lw     $s0,  4($sp) //pop
      lw     $ra,  0($sp) //pop
      addui  $sp, $sp, 16
      jr     $ra
```

Εδώ η Aloop υλοποιείται με χρήση saved καταχωρητών ($s0, $s1, $s2).

Τρεις τιμές πρέπει να «επιβιώσουν» μετά την κλήση της B, το X (στον $s0), το i (στον $s1) και το y (στον $s2).

Για να γίνει η κλήση B(X) πρέπει να αντιγράψουμε τον $s0 στον $a0

ΣΥΝΟΛΟ SAVE/RESTORE

= 4 / 4

Η Aloop με Caller-save καταχωρητές

```
Aloop:addui  $sp, $sp, -16
      sw     $ra, 0($sp) //push
```

```
Loop:  sw     $a0, 4($sp) //push
       sw     $a1, 8($sp) //push
       sw     $t0, 12($sp) //push
       jal    B
       lw     $a0, 4($sp) //pop
       lw     $t0, 12($sp) //pop
       add    $t0, $v0, $t0
       lw     $a1, 8($sp) //pop
       addi   $a1, $a1, -1
       bnez   $a1, Loop
       move   $v0, $t0
```

```
       lw     $ra,  0($sp) //pop
       addui  $sp, $sp, 16
       jr     $ra
```

Εδώ η Aloop υλοποιείται με χρήση temporary καταχωρητών ($t0, $a0, $a1).

Τρεις τιμές πρέπει να «επιβιώσουν» μετά την κλήση της B, το X (στον $a0), το i (στον $a1) και το y (στον $t0).

ΣΥΝΟΛΟ SAVE/RESTORE

= 3*N+1 / 3*N+1

Η Aloop2

```
int Aloop2(int X, int N) {
   int y;

   for (i=N, i>0; i--) {
      y = X + i;
      X = B(y);
   }
   return i;
}
```

Η Aloop2 καλεί N φορές την B(X) με παράμετρο το (X + i), και βάζει το αποτέλεσμα στην X

Η «ζωή» της y ***δεν*** εκτείνεται πέρα της κλήσης της B.

Η «ζωή» της X ***δεν*** εκτείνεται πέρα της κλήσης της B (ξεκινάει μετά την B και χρησιμοποιείται πριν την B!)

Η «ζωή» της i ***εκτείνεται*** πέρα της κλήσης της B.

Με Callee-save καταχωρητές

```
Aloop:addui  $sp, $sp, -16
      sw     $ra, 0($sp) //push
      sw     $s0, 4($sp) //push
      sw     $s1, 8($sp) //push
      sw     $s2, 12($sp) //push
```

```
      move   $s0, $a0    // x
      move   $s1, $a1    // i = N
Loop: add    $a0, $s0, $s1 //
      jal    B
      move   $s0, $v0
      addi   $s1, $s1, -1
      bnez   $s1, Loop
      move   $v0, $s1
```

```
      lw     $s2, 12($sp) //pop
      lw     $s1,  8($sp) //pop
      lw     $s0,  4($sp) //pop
      lw     $ra,  0($sp) //pop
      addui  $sp, $sp, 16
      jr     $ra
```

Εδώ η Aloop υλοποιείται με χρήση saved καταχωρητών ($s0, $s1, $s2)

Τρεις τιμές πρέπει να «επιβιώσουν» μετά την κλήση της B, το X (στον $s0), το N (στον $s1) και το y (στον $s2)

Για να γίνει η κλήση B(X) πρέπει να αντιγράψουμε τον $s0 στον $a0

ΣΥΝΟΛΟ SAVE/RESTORE

= 4 / 4

Με Caller-save καταχωρητές

```
Aloop:addui  $sp, $sp, -8
      sw     $ra, 0($sp) //push
```

```
      move   $t0, $a1 // i=N
Loop: sw     $t0, 4($sp) //push i
      add    $a0, $t1, $t0 // X+i
      jal    B
      lw     $t0, 4($sp) //pop
      move   $t1, $v0
      addi   $t0, $t0, -1
      bnez   $t0, Loop
      move   $v0, $t0
```

```
      lw     $ra,  0($sp) //pop
      addui  $sp, $sp, 8
      jr     $ra
```

Εδώ η Aloop2 υλοποιείται με χρήση temporary καταχωρητών ($t0, $a0, $a1)

Η μία τιμή που πρέπει να «επιβιώσει» μετά την κλήση της B, το i, βρίσκεται στον $t0 ο οποίος πρέπει να σωθεί και να επαναφερθεί. Το X αποθηκεύθεται στον $t1 το οποίο *δεν χρειάζεται* να σωθεί.

ΣΥΝΟΛΟ SAVE/RESTORE

= N+1 / N+1

Με Caller- **και** Callee-save καταχωρητές

```
Aloop:addui  $sp, $sp, -8
      sw     $ra, 0($sp) //push
      sw     $s0, 4($sp) //push
```

```
      move   $s0, $a1 // i=N
Loop: add    $a0, $t1, $s0
      jal    B
      move   $t1, $v0
      addi   $s0, $s0, -1
      bnez   $s0, Loop
      move   $v0, $s0
```

```
      lw     $s0, 4($sp) //pop
      lw     $ra, 0($sp) //pop
      addui  $sp, $sp, 8
      jr     $ra
```

Εδώ η Aloop2 υλοποιείται με χρήση και temporary ($a0 για το y) και saved καταχωρητών ($s0 για το i)

Η μία τιμή που πρέπει να «επιβιώσει» μετά την κλήση της B, το i, βρίσκεται στον $s0 οπότε δεν χρειάζεται να κάνουμε κάτι. Το X αποθηκεύθεται στον $t1 το οποίο *δεν χρειάζεται* να σωθεί

ΣΥΝΟΛΟ SAVE/RESTORE = 2 / 2

Ακόμα καλύτερα!

```
A loop:addui  $sp, $sp, -8
       sw     $ra, 0($sp) //push
       sw     $s0, 4($sp) //push
```

```
       move   $s0, $a1 // i=N
Loop:  add    $a0, $v0, $s0
       jal    B
       addi   $s0, $s0, -1
       bnez   $s0, Loop
       move   $v0, $s0
```

```
       lw     $s0, 4($sp) //pop
       lw     $ra, 0($sp) //pop
       addui  $sp, $sp, 8
       jr     $ra
```

Το X αποθηκεύθεται στον $v0 και δεν χρειάζεται η move

ΣΥΝΟΛΟ SAVE/RESTORE
= 2 / 2

Σύνολο εντολών:

Στατικές (πρόγραμμα) 13

Δυναμικές: 4*N+9

Ο μικρότερος και αποδοτικότερος κώδικας!