HY-225: Οργάνωση Υπολογιστών
Άνοιξη 2021

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Άσκηση 7:
Άσκηση Διαδικασιών και Λίστας, Επέκταση Προσήμου, Upper Immediates

Προθεσμία έως Τρίτη 30 Μαρτίου 2021, ώρα 23:59 (βδ. 7.2) (από βδ. 5.2)

7.1 Δομές Δεδομένων (Data Structures):

Σε αυτή την άσκηση θα χρησιμοποιήσουμε μία δομή δεδομένων (structure) που θα αποτελεί ένα κόμβο μιάς συνδεδεμένης λίστας (linked list). Κάθε κόμβος (δομή δεδομένων) μας θα αποτελείται από δύο λέξεις των 32 bits καθεμία (αφού ο RARS προσομοιώνει τον 32-μπιτο RISC-V, σε αντίθεση με το (Αγγλικό) βιβλίο που μιλά γιά τον 64-μπιτο): έναν ακέραιο **data** που θα περιέχει την "πληροφορία χρήστη", κι έναν δείκτη σύνδεσης (pointer) **nxtPtr** που θα περιέχει τη διεύθυνση του επόμενου κόμβου στη λίστα (στον τελευταίο κόμβο της λίστας, nxtPtr==0). Τα δύο στοιχεία (λέξεις) της δομής μας θα βρίσκονται σε διαδοχικές θέσεις (λέξεις) της μνήμης. Επομένως, κάθε δομή (κόμβος) μας θα έχει μέγεθος 2×4=8 Bytes. Διεύθυνση μιάς δομής είναι η διεύθυνση του πρώτου στοιχείου της, δηλαδή του στοιχείου με "μηδενικό offset", που γιά μας είναι το "data". Άρα, το δεύτερο στοιχείο της δομής μας, ο "nxtPtr", βρίσκεται στη διεύθυνση που προκύπτει προσθέτοντας 4×1=4 στη διεύθυνση της δομής (κόμβου).

7.2 Δυναμική Εκχώρηση Μνήμης (Dynamic Memory Allocation):

Το πρόγραμμά σας θα ζητάει και θα παίρνει δομές (κόμβους) από το "περιβάλλον" (λειτουργικό σύστημα) "δυναμικά", την ώρα που τρέχει (σε run-time). Γιά το σκοπό αυτό θα χρησιμοποιήστε το κάλεσμα περιβάλλοντος (environment call - ecall) **Sbrk** (Set Break). Το κάλεσμα αυτό "σπρώχνει" πιό πέρα (προς αύξουσες διευθύνσεις μνήμης) το σημείο "Break", το όριο δηλαδή πριν από το οποίο οι διευθύνσεις μνήμης που γεννά το πρόγραμμα είναι νόμιμες, ενώ μετά από το οποίο (και μέχρι την αρχή της στοίβας) οι διευθύνσεις είναι παράνομες και ενδεχόμενη χρήση τους προκαλεί το γνωστό από την C "segmentation violation - core dumped". Το κάλεσμα περιβάλλοντος "Sbrk" έχει κωδικό 9, και περιγράφεται στην καρτέλα Help→RISCV→Syscalls του RARS, και λειτουργεί κατ' αναλογία με τα άλλα καλέσματα περιβάλλοντος (εκτύπωσης και ανάγνωσης) που χρησιμοποιήσατε σε προηγούμενες ασκήσεις: Πριν το καλέσετε, βάλτε τον κωδικό της "υπηρεσίας", 9, στον καταχωρητή a7 (x17), και βάλτε στον γνωστό καταχωρητή a0 (x10) το όρισμα του καλέσματος, που εδώ είναι το πλήθος των νέων bytes που επιθυμείτε (ακέραιος αριθμός). Μετά την επιστροφή του, ο γνωστός καταχωρητής επιστρεφόμενης τιμής, a0 (x10) περιέχει τη διεύθυνση του νέου block μνήμης, του ζητηθέντος μεγέθους, που το σύστημα δίνει στο πρόγραμμά σας (έναν pointer). Η επιστρεφόμενη διεύθυνση μνήμης είναι πάντα διάφορη του μηδενός (εκτός –πιθανότατα– όταν γεμίσει όλη η μνήμη, αλλά δεν χρειάζεται εσείς εδώ να ελέγχετε κάτι τέτοιο), και είναι πάντα ευθυγραμμισμένη σε όρια λέξεων (πολλαπλάσιο του 4) (τουλάχιστο στη δική μας περίπτωση, που ζητάμε πάντα blocks μεγέθους πολλαπλάσιου του 4, αλλά –πιστεύω– και σε κάθε περίπτωση).

Άσκηση
7.3 Κατασκευή και Σάρωση Συνδεδεμένης Λίστας

Γράψτε και τρέξτε στον RARS, σε Assembly του RISC-V, ένα πρόγραμμα που πρώτα θα κατασκευάζει και θα γεμίζει με θετικούς ακέραιους αριθμούς μία συνδεδεμένη λίστα (linked list), και στη συνέχεια θα την σαρώνει επαναληπτικά, τυπώνοντας κάθε φορά ένα διαφορετικό υποσύνολο των στοιχείων της – συγκεκριμένα: όσα στοιχεία της είναι μεγαλύτερα από δοθείσα τιμή. Το πρόγραμμά σας θα κρατάει στον καταχωρητή **s0** (x8) τον pointer στην αρχή (στον πρώτο κόμβο) της λίστας, και θα αποτελείται από δύο κομμάτια, 7.4 και 7.5 –βλ. αμέσως παρακάτω. Στη συνέχεια, βασικά κομμάτια του προγράμματός σας θα τα κάνετε διαδικασίες (procedures), όπως περιγράφει η §7.6. Μερικές από τις διαδικασίες θα είναι υπερβολικά μικρές, αλλά αυτό γίνεται γιά λόγους εξάσκησης, ούτως ώστε το βάθος καλεσμάτων να φτάνει 2 επίπεδα κάτω από την main. (Σημείωση γιά την επιστροφή από διαδικασία: την ψευδοεντολή **jr**, που γιά επιστροφή από διαδικασία κανονικά είναι "`jr ra`", ο RARS δεν την δέχεται έτσι, αλλά απαιτεί να εμφανίζεται και το Immediate Offset, οπότε πρέπει να την δίνετε ως: "**`jr ra, 0`**").

7.4 Κατασκευή της Λίστας

Χρησιμοποιήστε τον καταχωρητή **s1** (x9) σαν pointer στην ουρά (στον τελευταίο κόμβο) της λίστας. Γιά διευκόλυνση του βρόχου κατασκευής της λίστας (επειδή η εισαγωγή σε κενή λίστα διαφέρει από την εισαγωγή σε μη κενή λίστα), αρχικοποιήστε τη λίστα να περιέχει ένα αδιάφορο ("dummy") κόμβο: ένα κόμβο με data=0. Η αρχικοποίηση γίνεται ζητώντας και παίρνοντας ένα κόμβο από το περιβάλλον (λειτουργικό σύστημα), γράφοντας data=0 και nxtPtr=0 (τελευταίος κόμβος) σε αυτόν, και κάνοντας τους s0 και s1 να δείχνουν σε αυτόν τον κόμβο (να περιέχουν τη διεύθυνσή του). Μετά, μπείτε στο βρόχο ανάγνωσης στοιχείων και κατασκευής της λίστας. Σε κάθε ανακύκλωση αυτού του βρόχου:

1. Διαβάζουμε έναν ακέραιο αριθμό από την κονσόλα.
2. Εάν ο αριθμός αυτός είναι αρνητικός ή μηδέν, βγαίνουμε από το βρόχο, αλλιώς:
3. Ζητάμε έναν νέο κόμβο από το περιβάλλον (Sbrk - memory allocation).
4. Τοποθετούμε τον αριθμό που διαβάσαμε στο πεδίο "data" του κόμβου.
5. Συνδέουμε το νέο κόμβο στην ουρά της λίστας.

7.5 Σάρωση της Λίστας

Το δεύτερο μέρος του προγράμματος θα διαβάζει έναν μη αρνητικό αριθμό, και θα τυπώνει, με τη σειρά από την αρχή μέχρι το τέλος, όσα στοιχεία της λίστας είναι **μεγαλύτερα** από αυτόν τον αριθμό. (Αφού ο αριθμός είναι ≥0, και τυπώνουμε τα στοιχεία που είναι μεγαλύτερα από αυτόν, προκύπτει ότι ο κόμβος "dummy", που έχει data=0, δεν θα τυπώνεται ποτέ· παρατηρήστε ότι όταν δίδεται το 0 ως αριθμός σύγκρισης, θα τυπώνονται όλα τα στοιχεία της λίστας (που είναι πάντα όλα θετικά), εκτός του "dummy"). Μην χρησιμοποιήστε τον pointer στον τελευταίο κόμβο της λίστας (από την παλαιά τιμή του καταχωρητή s1 (x9)) γιά να βρίσκετε πού τελειώνει η λίστα –χρησιμοποιήστε τον nxtPtr κάθε κόμβου γιά να ξέρετε αν υπάρχει ή όχι επόμενος κόμβος στη λίστα. Το μέρος αυτό του προγράμματος κάνει τα εξής:

1. Διαβάζει έναν ακέραιο αριθμό από την κονσόλα και τον φυλάει στον καταχωρητή s1 (x9). Εάν ο αριθμός αυτός είναι αρνητικός, το πρόγραμμα τερματίζει μέσω της κλήσης περιβάλλοντος (ecall) "Exit" που έχει κωδικό 10

(δηλαδή βάζετε τον αριθμό 10 στον καταχωρητή a7 πριν το ecall).

2. Αρχικοποιεί τον καταχωρητή s2 (x18) σαν δείκτη (pointer) σάρωσης, να δείχνει στον πρώτο κόμβο της λίστας (τον ξέρουμε από τον s0 (x8)).
3. Μπαίνει σ' ένα βρόχο, σε κάθε ανακύκλωση του οποίου:
    i. ελέγχει αν τα "data" του κόμβου όπου δείχνει ο s2 (x18) είναι ή όχι μεγαλύτερα από τον s1 (x9),
    ii. αν είναι μεγαλύτερα τα τυπώνει,
    iii. ελέγχει αν υπάρχει ή όχι επόμενος κόμβος στη λίστα,
    iv. αν δεν υπάρχει βγαίνει από το βρόγχο,
    v. αν υπάρχει, προχωρεί τον s2 (x18) να δείξει σε αυτόν τον επόμενο κόμβο και επιστρέφει στην αρχή του βρόχου.
4. Μετά την έξοδο του βρόχου, επιστρέφει (πάντα) στην αρχή του δεύτερου μέρους του προγράμματος, γιά να ζητήσει μιά νέα τιμή και να ξανατυπώσει τα μεγαλύτερα από αυτήν στοιχεία (εάν η τιμή δεν είναι αρνητική).

7.6 Χρήση Διαδικασιών

- Μετατρέψτε το βήμα 7.4(1) σε μια υπορουτίνα **`read_int()`** η οποία δεν παίρνει καμία παράμετρο (όρισμα) εισόδου, διαβάζει έναν ακέραιο αριθμό, και επιστρέφει τον αριθμό που διάβασε.
- Μετατρέψτε το βήμα 7.4(3) σε μια υπορουτίνα **`node_alloc()`** η οποία δεν παίρνει καμία παράμετρο εισόδου, ζητάει από το περιβάλλον να δεσμεύσει ένα κόμβο για τη λίστα, και επιστρέφει τη διεύθυνση της μνήμης που έχει δεσμευθεί, ώστε το πρόγραμμα που καλεί τη node_alloc() να εισάγει τον κόμβο στη λίστα.
- Αλλάξτε το πρόγραμμα του 7.4 ώστε να χρησιμοποιεί τις ρουτίνες read_int() και node_alloc(). Στη χρήση καταχωρητών από τις διαδικασίες που γράφετε –εδώ από τις read_int() και node_alloc()– όπως και από το πρόγραμμα σας που τις καλεί, πρέπει να τηρήσετε τις συμβάσεις χρήσης καταχωτητών (αν και, ειδικά οι read_int() και node_alloc() είναι αρκετά απλές και δεν είναι απαραίτητο να έχουν τοπικές μεταβλητές).
- Γράψτε μια υπορουτίνα **`print_node()`** που κάνει ό,τι περιγράφουν τα βήματα 7.5(3i) και 7.5(3ii). Η υπορουτίνα θα παίρνει ως εισόδους (α) τη διεύθυνση ενός κόμβου, και (β) τον ακέραιο προς τον οποίο συγκρίνουμε, και δεν θα επιστρέφει τίποτε.
- Γράψτε μια υπορουτίνα **`search_list()`** που υλοποιεί όλο το βήμα 7.5(3). Η ρουτίνα θα παίρνει ως πρώτη είσοδο τη διεύθυνση του πρώτου κόμβου τις λίστας (δείκτης σάρωσης) και ως δεύτερη είσοδο την τιμή για την οποία πρέπει να ψάξει. Στη συνέχεια θα υλοποιεί τα βήματα (i,ii,iii,iv,v), καλώντας τη συνάρτηση print_node() για τα βήματα (i,ii). Όπως είπαμε, πρέπει να τηρείτε τις συμβάσεις χρήσης καταχωτητών· κάθε υπορουτίνα που χρειάζεται αποθήκευση κάποιου καταχωρητή ή/και της διεύθυνσης επιστροφής, θα πρέπει να το κάνει στη στοίβα, με τον τρόπο που ορίζουν οι συμβάσεις.
- Αλλάξτε το πρόγραμμα σας του μέρους 7.5 ώστε να καλεί τις read_int() και search_list(), όπου η τελευταία θα καλεί την print_node(). Και πάλι πρέπει να χρησιμοποιήσετε τις συμβάσεις χρήσης των καταχωρητών μεταξύ καλούσας και καλούμενης διαδικασίας. Επίσης η κάθε ρουτίνα που χρειάζεται τη στοίβα θα πρέπει να το κάνει με τον τρόπο που ορίζουν οι συμβάσεις.
- Τελικά, το συνολικό πρόγραμμα σας θα πρέπει να έχει μια **`main()`** που αποτελείται απο δύο μέρη. Το πρώτο μέρος της θα υλοποιεί το 7.4 με χρήση των node_alloc() και read_int(), ενώ το δεύτερο θα υλοποιεί το 7.5 με χρήση των read_int(), search_list(), και print_node().

Τρόπος Παράδοσης: Παραδώστε ηλεκτρονικά τον κώδικά σας, "**`ex07.asm`**", κι ένα στιγμιότυπο της εκτέλεσής του στον RARS, "**`ex07.jpg`**". Παραδώστε τα αυτά μέσω: **`turnin ex07@hy225 [directoryName]`**
Θα εξεταστείτε **και προφορικά** για την Άσκηση 7, με διαδικασία γιά την οποία θα ενημερωθείτε μέσω ηλτά (email) στη λίστα του μαθήματος.

7.7 Προσημασμένοι Αριθμοί, Επέκταση Προσήμου

Οι σημερινοί επεξεργαστές παριστάνουν τους προσημασμένους (signed) αριθμούς σε κωδικοποίηση συμπληρώματος ως-προς-2, όπως είχαμε δεί στο μάθημα της Ψηφιακής Σχεδίασης (HY-120, ενότητα 6.3). Σε αυτή την αναπαράσταση, η μετατροπή προσημασμένου (signed) αριθμού από λιγότερα σε περισσότερα bits γίνεται με την εξής τεχνική, που ονομάζεται "*επέκταση προσήμου*" *(sign extension)* και αποδεικνύεται μαθηματικά ως εξής:

Έστω ο προσημασμένος ακέραιος $A_{s,k}$ με *k bits*, τον οποίο θέλουμε να μετατρέψουμε στον ίδιο αριθμό $A_{s,n}$ με *n bits*: $A_{s,n} = A_{s,k}$ όπου n>k. Εάν τα bits του $A_{s,k}$ τα ερμηνεύσουμε σαν μη προσημασμένο (unsigned) ακέραιο, τότε θα μοιάζουν με (θα δηλώνουν) έναν αριθμό που ας το ονομάσουμε $A_{u,k}$ και ομοίως εάν τα bits του $A_{s,n}$ τα ερμηνεύσουμε σαν unsigned τότε θα μοιάζουν με τον $A_{u,n}$. Εάν ο $A_{s,k} = A_{s,n}$ είναι μη αρνητικός (δηλ. θετικός ή μηδέν), τότε, κατά τον ορισμό της κωδικοποίησης συμπληρώματος ως-προς-2, (α) το αριστερό bit τους θα είναι μηδέν, και (β) οι απρόσημες ερμηνίες τους θα είναι: $A_{u,k} = A_{s,k}$ και $A_{u,n} = A_{s,n}$, και αφού $A_{s,n} = A_{s,k}$ τότε θα είναι και: $A_{u,n} = A_{u,k}$. Αυτοί οι δύο τελευταίοι, αφού είνα unsigned ακέραιοι, με n και k bits αντίστοιχα, και είναι ίσοι μεταξύ τους, προκύπτει ότι ο $A_{u,n}$ που έχει περισσότερα bits θα είναι ίδιος με τον $A_{u,k}$ αλλά με n-k μηδενικά προστεθημένα αριστερά από τον $A_{u,k}$.

Αλλιώς, εάν οι $A_{s,n} = A_{s,k}$ είναι αρνητικοί, τότε, κατά τον ορισμό μας, (α) το αριστερό bit τους θα είναι ένα, και (β) οι απρόσημες ερμηνίες τους θα είναι: $A_{u,k} = A_{s,k} + 2^k$ και $A_{u,n} = A_{s,n} + 2^n$. Δεδομένου ότι: $A_{s,n} = A_{s,k}$ προκύπτει ότι θα είναι και: $A_{u,n} - 2^n = A_{u,k} - 2^k$. Επομένως, η αναπαράσταση που ψάχνουμε είναι η: $A_{u,n} = A_{u,k} - 2^k + 2^n = A_{u,k} + (2^n - 2^k) = (2^{n-k} - 1) \cdot 2^k + A_{u,k}$. Σε αυτήν την τελευταία έκφραση, ο μεν αριστερός προσθετέος, $(2^{n-k} - 1) \cdot 2^k$, αποτελείται από (n-k) το πλήθος άσσους (που είναι ο αριθμός $2^{n-k} - 1$) ολισθημένους αριστερά κατά k θέσεις bits (που είναι ο πολλαπλασιασμός επί 2^k), ο δε δεξιός προσθετέος, $A_{u,k}$, είναι τα αρχικά k bits του αρχικού αριθμού που μας δόθηκε. Δεδομένου ότι ο πρώτος προσθετέος έχει όλο μηδενικά στις k δεξιές θέσεις, ο δε δεύτερος προσθετέος έχει όλο μηδενικά στις (n-k) αριστερές θέσεις, το άθροισμά τους θα είναι προφανώς απλώς η "συγκόλληση" (concatenation) των δύο αυτών ποσοτήτων. Επομένως, η αναπαράσταση με n bits του αρχικού αριθμού που μας δόθηκε θα αποτελείται από τα αρχικά k bits, δεξιά, μαζί με (n-k) άσσους κολλημένους ακριβώς αριστερά τους.

Συνολικά λοιπόν, γιά να μετατρέψουμε ένα προσημασμένο (signed) ακέραιο από k (λιγότερα) bits σε n (περισσότερα) bits, δεν έχουμε παρά να κάνουμε το εξής: Τοποθετούμε αριστερά από τα bits που μας δόθηκαν (n-k) επιπλέον bits τα οποία είναι όλα τους *αντίγραφα του αριστερού* (most significant) bit του αριθμού που μας δόθηκε, δηλαδή αντίγραφα του bit που υποδεικνύει το πρόσημο του δοθέντος αριθμού (0 γιά θετικούς ή το μηδεν, 1 γιά αρνητικούς). Η πράξη αυτή ονομάζεται επομένως, προφανώς, *επέκταση προσήμου* (sign extension).

7.8 Εντολές *Upper Immediate*: lui, auipc

Η εντολή **`lui rd, Imm20`** (load upper immediate), με format "**U**" (βλ. §4.3), στον 32-μπιτο RISC-V, γράφει τα 20 bits του πεδίου Imm20 της στα αριστερά 20 bits του καταχωρητή rd, και μηδενίζει τα δεξιά 12 bits του rd. Στον 64-μπιτο RISC-V, βάζει τα παραπάνω 32 bits στο δεξιό (LS) ήμισυ του rd, και τα κάνει sign-extend στο αριστερό (MS) ήμισυ του rd. Παρ' ότι ονομάζεται "load" όμως **δεν** είναι εντολή προσπέλασης της μνήμης δεδομένων –δεν ανήκει στην κατηγορία των εντολών load.

Στη συνήθη χρήση της ακολουθείται από μιάν εντολή `addi` (add immediate), η οποία προσθέτει στον ίδιο καταχωρητή το 12-μπιτο Immediate της. Δεδομένου ότι η `lui` έβαλε 12 μηδενικά δεξιά στον καταχωρητή, η πρόσθεση καταλήγει στο να αφήσει στα 12 δεξιά bits τη σταθερή ποσότητα από την εντολή `addi`. Στον 32-μπιτο RISC-V, στα 20 αριστερά bits του καταχωρητή, η μεν εντολή `lui` έβαλε την 20-μπιτη σταθερά της, η δε εντολή `addi` προσθέτει σε αυτήν είτε (α) 20 μηδενικά εάν η (12-μπιτη) σταθερά της έχει 0 στο αριστερό bit της, είτε (β) 20 άσους εάν η (12-μπιτη) σταθερά της έχει 1 στο αριστερό bit της. Στη μεν περίπτωση (α) τα 20 μηδενικά αφήνουν αναλοίωτα τα 20 αριστερά bits, στη δε περίπτωση (β) η πρόσθεση 20 άσων ισοδυναμεί με την πρόσθεση του αριθμού -1 (μείον 1) σε αυτά τα 20 bits. Έτσι τελικά μπορούμε να συνθέσουμε την οιαδήποτε αυθαίρετη 32-μπιτη σταθερά, βάζοντας τα μεν 12 δεξιά bits της στο Immediate της `addi`, τα δε 20 αριστερά bits της, είτε αυτούσια είτε αυξημένα κατά +1, στο Immediate της `lui` – όπου η επιλογή "αυτούσια" ή "αυξημένα κατά +1" γίνεται όταν τα 12ο από δεξιά bit είναι 0 ή 1, αντίστοιχα. Στο βιβλίο η εντολή αυτή, καθώς και η τυπική χρήση της, περιγράφονται στην αρχή της §2.10.

Επίσης ο RISC-V προσφέρει υποστήριξη γιά **Relocatable Code**, δηλαδή κώδικα Assermbly/Object τον οποίο μπορεί εύκολα ο Linker να αλλάξει (ή ακόμα και να μην χρειάζεται καμία αλλαγή) προκειμένου να τον συνενώνει (link) με άλλον κώδικα, τοποθετώντας (φορτώνοντάς) τον σε *αυθαίρετη* θέση στη μνήμη (δηλαδή να τον κάνει rellocate), όπως π.χ. χρειάζεται γιά τη δυναμική συνένωση διαδικασιών βιβλιοθήκης (dynamically linked libraries - §2.12, pp. 130-132 Αγγλικού βιβλίου). Το βασικό addressing mode (τρόπος δημιουργίας διευθύνσεων μνήμης) γιά σκοπούς εύκολου relocation είναι το PC-relative: όταν μιά διεύθυνση μνήμης εκφράζεται σαν *σχετική απόσταση* από την τιμή του PC της εντολής που την γεννά, τότε αυτή η απόσταση δεν αλλάζει κατά το relocation. Γιά τις διακλαδώσεις υπό συνθήκη, η διεύθυνση προορισμού δίδεται πάντα σαν PC-relative όπως έχουμε δεί, με βελησνεκές ±4 KBytes. Το ίδιο ισχύει και γιά το κάλεσμα διαδικασίας (jal), καθώς και γιά το απλό άλμα (jump) του συντίθεται ως ψευδοεντολή μέσω αυτού, με βελησνεκές ±1 MByte. Ο RISC-V προσφέρει *μία ακόμα εντολή*, την **`auipc`** (add upper immediate to PC), προκειμένου (α) να επεκτείνει το κάλεσμα/άλμα σε οιαδήποτε αυθαίρετη 32-μπιτη απόσταση, και (β) να προσφέρει επίσης PC-relative addressing, και μάλιστα με αυθαίρετη 32-μπιτη απόσταση, γιά εντολές *load και store δεδομένων*, ως εξής.

Η εντολή **`auipc rd, Imm20`** (add upper immediate to PC), με format επίσης "**U**" όπως και η lui, πρώτα κατασκευάζει την ίδια σταθερή ποσότητα όπως και η lui, και στη συνέχεια προσθέτει αυτή τη σταθερή ποσότητα στην τιμή του PC της και γράφει το αποτέλεσμα της πρόσθεσης στον καταχωρητή rd· με άλλα λόγια, είναι σαν να κάνει: `lui rd, Imm20` και αμέσως μετά να κάνει: `rd ← PC+rd`. Ή αλλιώς μπορούμε να πούμε ότι η `auipc rd, Imm20` κάνει: *rd ← PC + (sign-extended)Imm20×2^{12}*. Όπως και η lui, η auipc μπορεί να χρησιμοποιηθεί σαν η πρώτη εντολή σε ένα ζευγάρι εντολών με δεύτερη εντολή είτε ένα κάλεσμα/άλμα είτε μία load/store, γιά να προκαλέσει κάλεσμα/άλμα ή προσπέλαση δεδομένων PC-

relative στη διεύθυνση `PC+Const32`, όπου `Const32` είναι μιά αυθαίρετη 32-μπιτη σταθερά αποτελούμενη από δύο κομάτια, `HI` τα 20 αριστερά bits, και `LO` τα 12 δεξιά bits, ως εξής. Η πρώτη εντολή του ζεύγους κατασκευάζει την τιμή $PC+HI\times 2^{12}$ και την τοποθετεί σ' έναν προσωρινό καταχωρητή, π.χ. τον t0: **`auipc t0, HI`** (ή `auipc t0, HI+1` εάν το κομάτι `LO` αρχίζει με 1, όπως σχολιάζαμε παραπάνω και γιά την lui). Η δεύτερη εντολή του ζεύγους είναι είτε jalr γιά κάλεσμα/άλμα είτε load/store, όπου και στις δύο περιπτώσεις χρησιμοποιείται σαν pointer ο προηγούμενος καταχωρητής t0, και σαν Offset τα δεξιά 12 bits της Const32, το LO, άρα τελικά η διεύθυνση είναι η $PC+HI\times 2^{12}+LO = PC+Const32$:

- Κάλεσμα PC-relative σε αυθαίρετη απόστ.:
  `auipc t0, HI` (ή `HI+1`); **`jalr ra, LO(t0)`**
- Άλμα PC-relative σε αυθαίρετη απόσταση:
  `auipc t0, HI` (ή `HI+1`); **`jalr x0, LO(t0)`**
- Load PC-relative σε αυθαίρετη απόσταση:
  `auipc t0, HI` (ή `HI+1`); **`ld rd, LO(t0)`**
- Store PC-relative σε αυθαίρετη απόσταση:
  `auipc t0, HI` (ή `HI+1`); **`sd rs2, LO(t0)`**
