Έκδοση RISC_V 2025

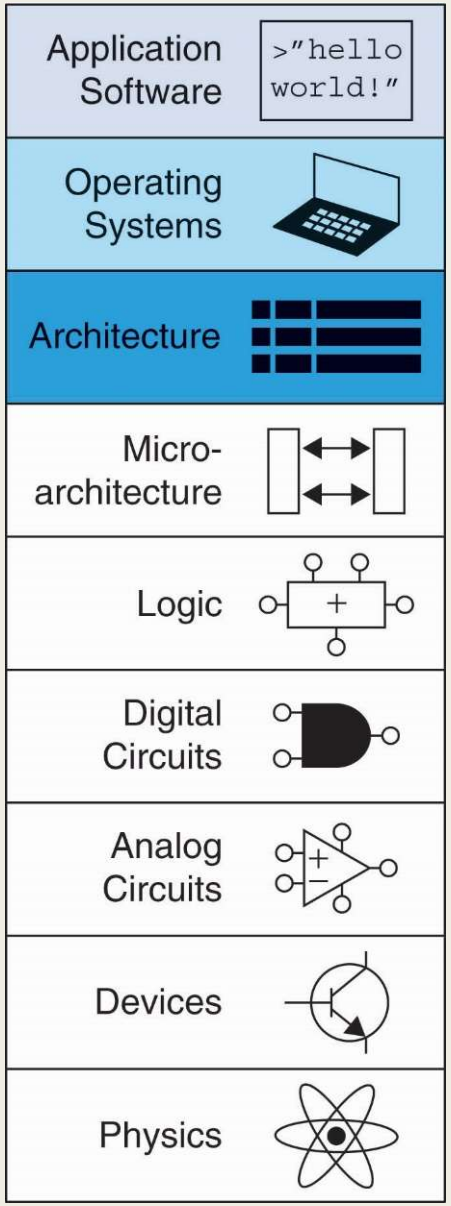
Κεφάλαιο 6

Αρχιτεκτονική

Καθηγητής Αντώνης Πασχάλης

Περιεχόμενα κεφαλαίου

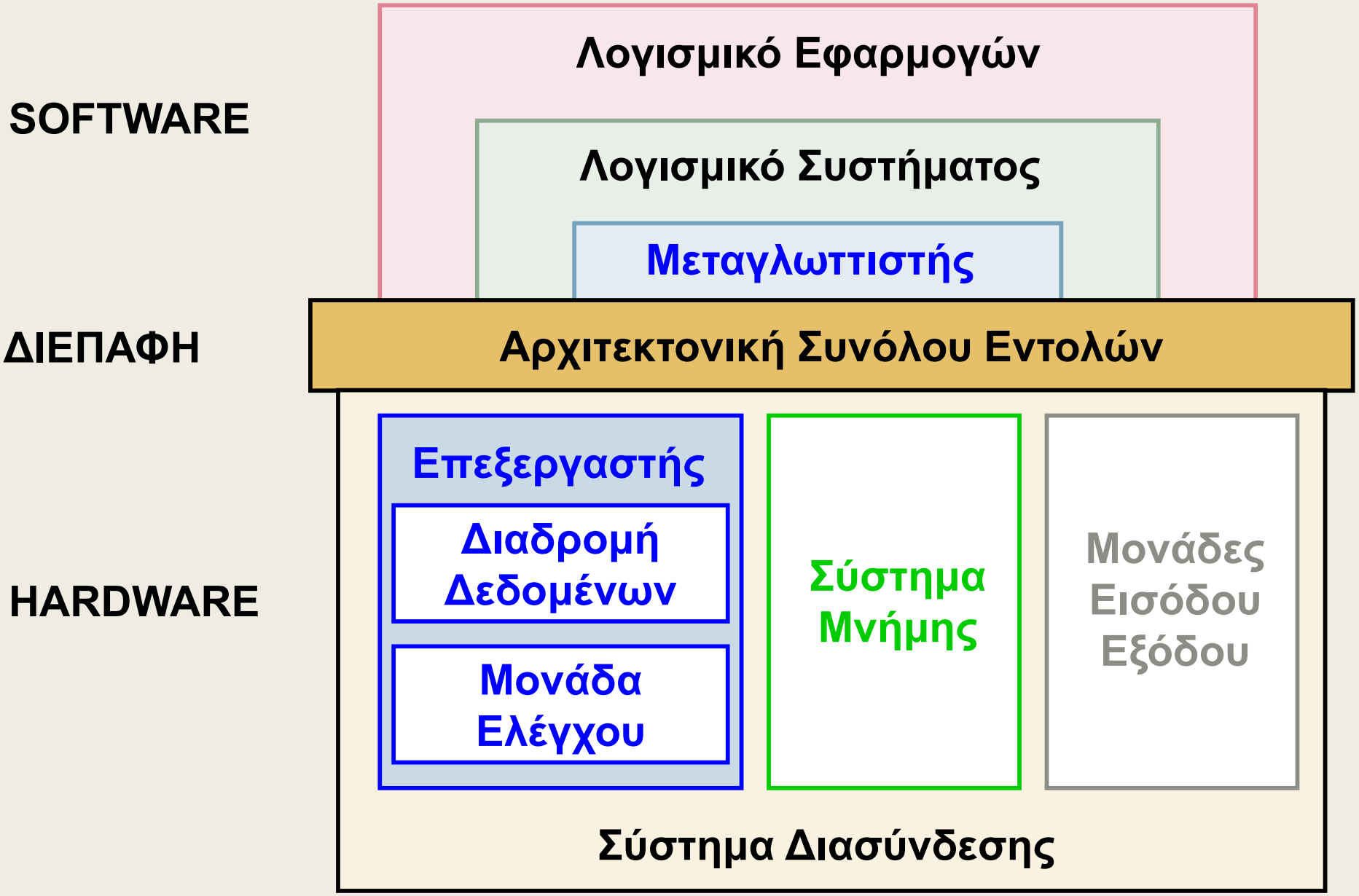
- Εισαγωγή
- Αρχιτεκτονική συνόλου εντολών
- Συμβολική γλώσσα
- Γλώσσα μηχανής
- Προγραμματισμός χαμηλού επιπέδου
- Τρόποι διευθυνσιοδότησης



Εισαγωγή

- Οι λέξεις της γλώσσας ενός υπολογιστή ονομάζονται **εντολές** (*instructions*) και ορίζουν ένα **σύνολο εντολών** (*instruction set*) σε μία συγκεκριμένη **αρχιτεκτονική συνόλου εντολών**
  - *Οποιοδήποτε πρόγραμμα που εκτελείται σε έναν υπολογιστή χρησιμοποιεί το ίδιο σύνολο εντολών*
- Οι εντολές κωδικοποιούνται ως δυαδικοί αριθμοί σε μια μορφή που ονομάζεται **γλώσσα μηχανής** (machine language).
  - *η αναπαράσταση εντολών σε γλώσσα μηχανής είναι δύσχρηστη*
- Προτιμούμε να αναπαριστούμε τις εντολές σε μια συμβολική μορφή που ονομάζεται **συμβολική γλώσσα** (assembly language)
  - *χρησιμοποιείται στον προγραμματισμό χαμηλού επιπέδου*
  - *εάν μάθουμε τη συμβολική γλώσσα ενός υπολογιστή μπορούμε να κατανοήσουμε τη συμβολική γλώσσα ενός άλλου υπολογιστή*
    - τουλάχιστον όσο αφορά στο βασικό σύνολο εντολών

Ο υπολογιστής



Το λογισμικό συστήματος

- Το απαιτούμενο λογισμικό για τη λειτουργία του υπολογιστή ως ένα ολοκληρωμένο σύστημα. Συμπεριλαμβάνει:
  - *το λειτουργικό σύστημα (operating system – OS)*
    - Πρόγραμμα επίβλεψης που διαχειρίζεται τους πόρους ενός υπολογιστή προς όφελος των εκτελούμενων προγραμμάτων
      - *Διαχείριση βασικών λειτουργιών εισόδου και εξόδου*
      - *Κατανομή χώρου αποθήκευσης και μνήμης*
      - *Παροχή δυνατοτήτων κοινής χρήσης του υπολογιστή μεταξύ πολλών εφαρμογών που τον χρησιμοποιούν ταυτόχρονα*
  - *τον μεταγλωττιστή (compiler)*
    - Πρόγραμμα που μεταφράζει εντολές μιας γλώσσας υψηλού επιπέδου (π.χ. C) σε εντολές συμβολικής γλώσσας (assembly language)
  - *τον συμβολομεταφραστή (assembler)*
    - Πρόγραμμα που μεταφράζει μια συμβολική έκδοση των εντολών στη δυαδική έκδοσή τους (σε γλώσσα μηχανής)

Το υλικό του υπολογιστή

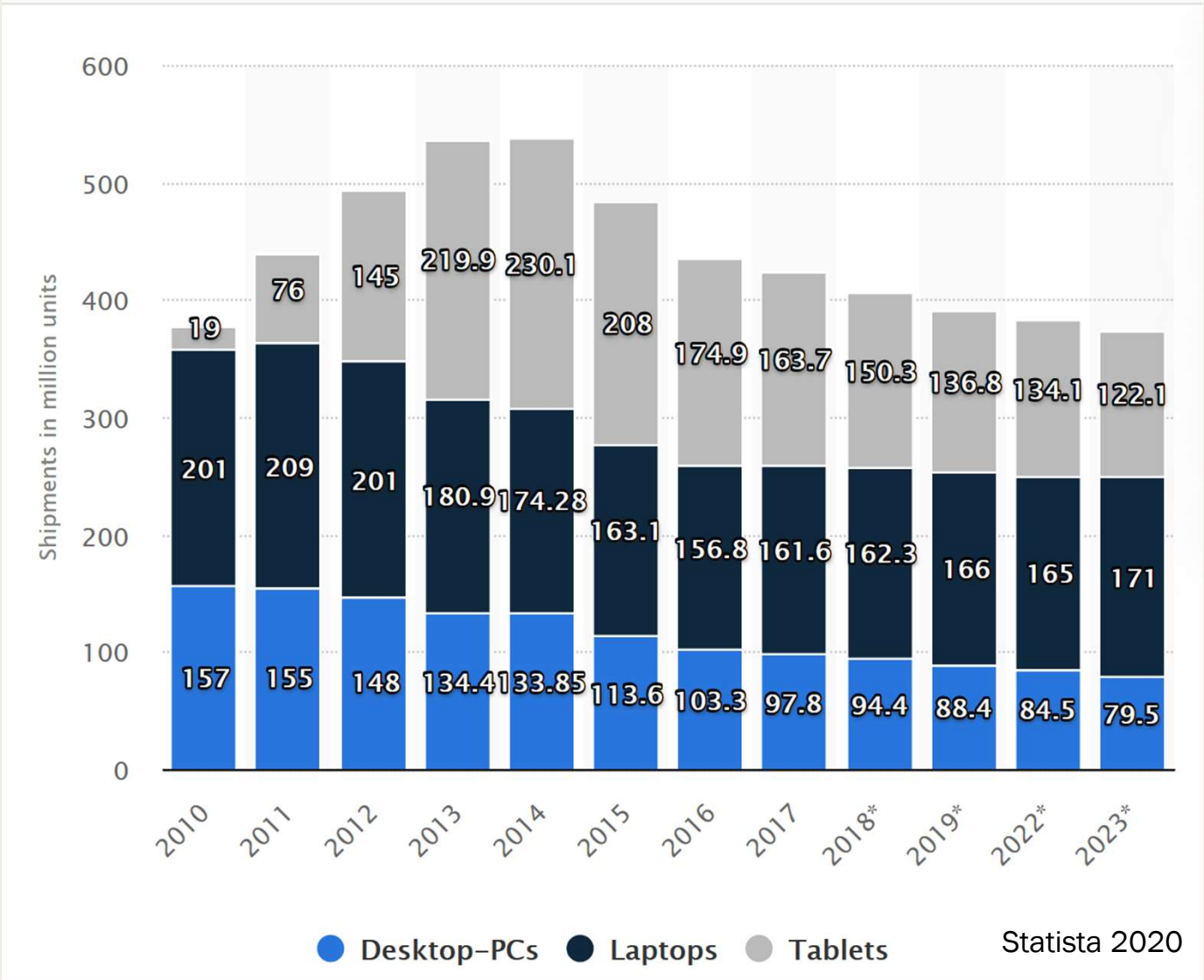
- Διαδρομή δεδομένων (datapath)
  - *το τμήμα τού επεξεργαστή που εκτελεί τις εντολές*
- Μονάδα ελέγχου (control unit)
  - *το τμήμα του επεξεργαστή που ελέγχει τη διαδρομή δεδομένων, τη μνήμη, και τις συσκευές εισόδου/εξόδου σύμφωνα με τις εντολές του προγράμματος*
- Μνήμη (memory)
  - *η περιοχή αποθήκευσης στην οποία διατηρούνται τα προγράμματα όταν εκτελούνται, και η οποία περιέχει τα δεδομένα που χρειάζονται τα προγράμματα αυτά*
- Συσκευή εισόδου (input device)
  - *ένας μηχανισμός μέσω του οποίου ο υπολογιστής τροφοδοτείται με πληροφορίες, όπως είναι το πληκτρολόγιο ή το ποντίκι*
- Συσκευή εξόδου (output device)
  - *Ένας μηχανισμός που μεταφέρει το αποτέλεσμα ενός υπολογισμού στο χρήστη (όπως η οθόνη ή ο εκτυπωτής) ή σε άλλον υπολογιστή*
- Σύστημα διασύνδεσης
  - *Ένας μηχανισμός που εξασφαλίζει την επικοινωνία του επεξεργαστή με τη μνήμη και τις μονάδες εισόδου - εξόδου.*

Βασικές κατηγορίες υπολογιστών

- Προσωπικός υπολογιστής (personal computer – PC)
  - *Ένας υπολογιστής σχεδιασμένος για χρήση από έναν ιδιώτη, που συνήθως περιλαμβάνει μια οθόνη, ένα πληκτρολόγιο και ένα ποντίκι*
  - *Σταθερός (desktop) ή φορητός (laptop)*
- Διακομιστής (server)
  - *Ένας υπολογιστής που χρησιμοποιείται για την εκτέλεση μεγαλύτερων προγραμμάτων για πολλούς χρήστες, συνήθως ταυτόχρονα, και ο οποίος τυπικά είναι προσπελάσιμος μόνο μέσω κάποιου δικτύου*
  - *Είναι η σύγχρονη μορφή αυτού που ήταν κάποτε τα μεγάλα υπολογιστικά συστήματα (mainframes)*
- Ενσωματωμένος υπολογιστής (embedded computer)
  - *Ένας υπολογιστής μέσα σε μια άλλη συσκευή, που χρησιμοποιείται για την εκτέλεση προκαθορισμένων εφαρμογών*
  - *Υπολογιστές σε ένα smartphone/tablet, σε ένα βιντεοπαιχνίδι, σε μια ψηφιακή τηλεόραση, ή σε μία οικιακή συσκευή*
  - *Δίκτυα επεξεργαστών που ελέγχουν ένα σύγχρονο αυτοκίνητο, ένα αεροπλάνο ή ένα εμπορικό πλοίο*
  - *Σήμερα, σχεδιάζονται κυρίως με τη χρήση πολλαπλών πυρήνων επεξεργαστών (processor cores) σε ένα τσιπ*

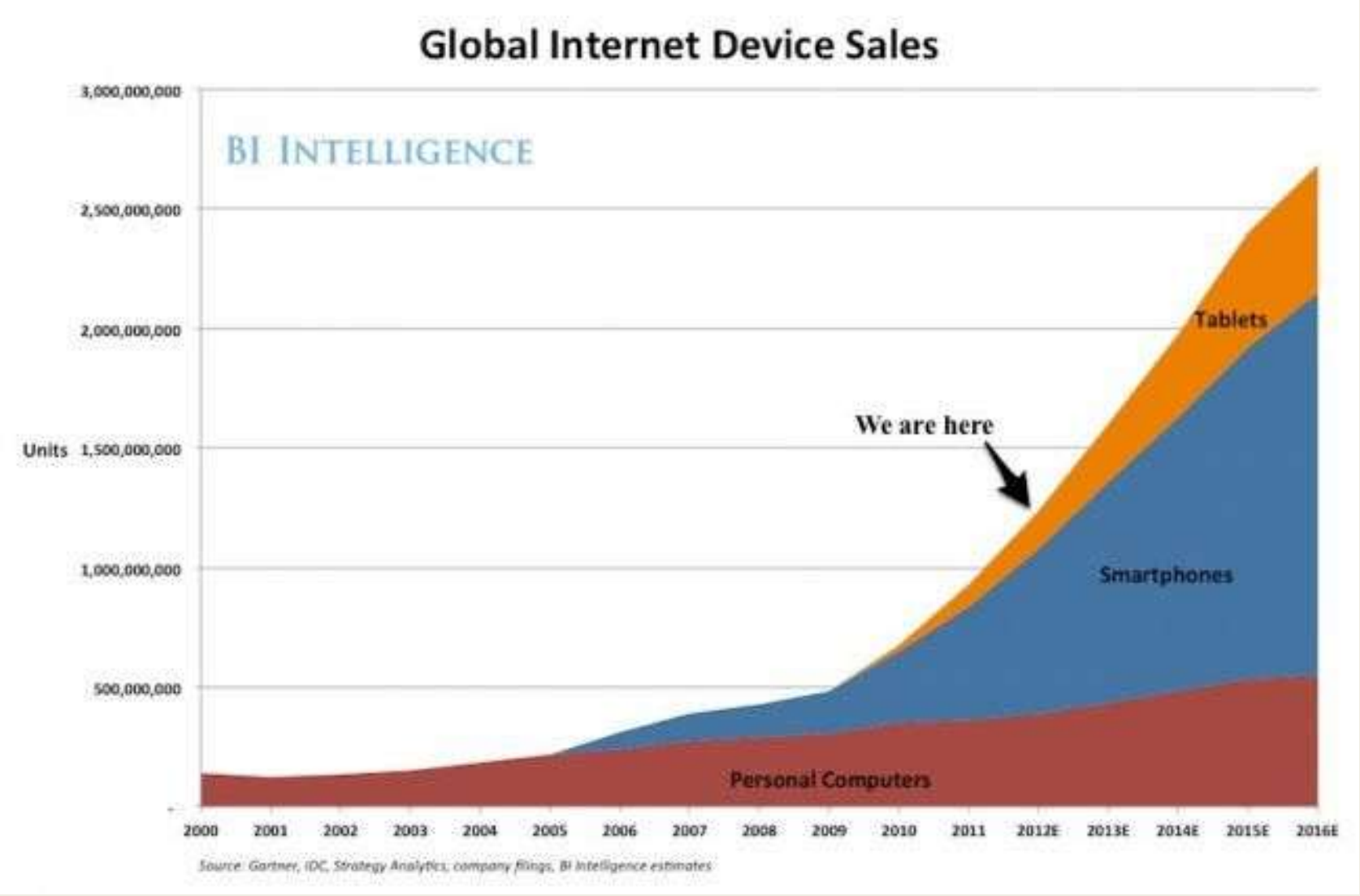
Πωλήσεις ανά κατηγορία υπολογιστών

- Στους προσωπικούς υπολογιστές υποχωρούν οι πωλήσεις των desktops έναντι των laptops και των tablets
  - *Πωλήσεις desktops (+ servers), laptops, tablets*



Πωλήσεις ανά κατηγορία υπολογιστών

- Κυριαρχούν οι ενσωματωμένοι υπολογιστές
  - *Πωλήσεις personal computers, smartphones, tablets*



Αρχιτεκτονική υπολογιστών (computer architecture)

- Προσδιορίζει τη θεώρηση ενός υπολογιστή από τη σκοπιά του προγραμματιστή με βάση την αρχιτεκτονική συνόλου εντολών (instruction set architecture) που είναι η διεπαφή υλικού και λογισμικού
  - *κατηγορίες εντολών – λειτουργιών*
  - *τελεστέοι εντολών (instruction operands)*
    - τα απαιτούμενα δεδομένα που χρησιμοποιούνται κατά την εκτέλεση μίας πράξης καθώς και το αποτέλεσμα της πράξης
    - τύποι και μεγέθη τελεστέων
      - *ακέραιοι των 8, 16, 32, 64 bit*
      - *πραγματικοί κινητής υποδιαστολής (απλής ακρίβειας των 32 bit ή διπλής ακρίβειας των 64 bit)*
      - *χαρακτήρας ASCII ή 2 BCD σε ένα byte (8 bit)*
    - που βρίσκονται αποθηκευμένοι
      - *στους καταχωρητές, στη μνήμη ή σε πεδίο της εντολής*
      - *τρόποι διευθυνσιοδότησης*
  - *μορφή και κωδικοποίηση εντολών*
- Παραδείγματα γνωστών αρχιτεκτονικών (συνόλου εντολών)
  - *RISC-V, RISC-V, x86, MIPS, SPARC, PowerPC*

Μικροαρχιτεκτονική (microarchitecture)

- Προσδιορίζει πως υλοποιείται η αρχιτεκτονική στο υλικό
  - *Σχεδίαση της διαδρομής δεδομένων (datapath) του επεξεργαστή*
    - χωροταξική διάταξη των καταχωρητών, των μνημών, των μονάδων ALU και των άλλων δομικών στοιχείων που απαρτίζουν τη διαδρομή δεδομένων
  - *Σχεδίαση της μονάδας ελέγχου (control unit) του επεξεργαστή*
    - παραγωγή των κατάλληλων σημάτων ελέγχου για τον χρονισμό του επεξεργαστή και των υπολοίπων διατάξεων
- Συχνά υπάρχουν πολλές διαφορετικές μικροαρχιτεκτονικές για την ίδια αρχιτεκτονική
  - *Διαφορετικές μικροαρχιτεκτονικές από Intel και AMD για x86*

Βασικές αρχιτεκτονικές συνόλου εντολών
Ανάλογα με το που βρίσκονται αποθηκευμένοι οι τελεστέοι

- Έμμεση αποθήκευση σε στοίβα (stack)
  - *οι τελεστέοι αποθηκεύονται έμμεσα στη στοίβα, πριν και μετά την εκτέλεση της πράξης*
  - *οι εντολές φόρτωσης από τη μνήμη στη στοίβα (PUSH) και αποθήκευσης από τη στοίβα στη μνήμη (POP) προσδιορίζουν:*
    - έναν τελεστέο που αποθηκεύεται άμεσα στη μνήμη και έναν τελεστέο που αποθηκεύεται έμμεσα στη στοίβα
  - *οι εντολές εκτέλεσης πράξεων δεν προσδιορίζουν κανένα τελεστέο*
    - και οι 3 τελεστέοι προσδιορίζονται έμμεσα στη στοίβα
    - Η πράξη εκτελείται μεταξύ των δύο τελευταίων εισερχομένων δεδομένων στην στοίβα (που εξέρχονται από την στοίβα για να εκτελεσθεί η πράξη), ενώ το αποτέλεσμα αποθηκεύεται στην κορυφή της στοίβας (top of stack)
    - Χαρακτηρίζονται ως εντολές μηδενικής διεύθυνσης

Κώδικας γλώσσας υψηλού επιπέδου

```
b = mem[B];
c = mem[C];
a = b + c;
Mem[A] = a;
```

Κώδικας συμβολικής γλώσσας

```
PUSH B  ; το B είναι διεύθυνση
PUSH C  ; το C είναι διεύθυνση
ADD     ; μηδενικής διεύθυνσης
POP A   ; το C είναι διεύθυνση
```

Βασικές αρχιτεκτονικές συνόλου εντολών
Ανάλογα με το που βρίσκονται αποθηκευμένοι οι τελεστέοι

- Έμμεση αποθήκευση σε συσσωρευτή (accumulator)
  - *ο ένας από τους δύο τελεστέους πριν την εκτέλεση της πράξης καθώς και το αποτέλεσμα της πράξης αποθηκεύονται έμμεσα στο συσσωρευτή που είναι ένας καταχωρητής ειδικού σκοπού*
  - *οι εντολές φόρτωσης από τη μνήμη στον συσσωρευτή (LOAD) και αποθήκευσης από τον συσσωρευτή στη μνήμη (STORE) προσδιορίζουν:*
    - έναν τελεστέο που αποθηκεύεται άμεσα στη μνήμη και έναν τελεστέο που αποθηκεύεται έμμεσα στον συσσωρευτή
  - *οι εντολές εκτέλεσης πράξεων προσδιορίζουν:*
    - έναν τελεστέο που αποθηκεύεται άμεσα στη μνήμη και έναν τελεστέο που αποθηκεύεται έμμεσα στον συσσωρευτή

Κώδικας γλώσσας υψηλού επιπέδου

```
b = mem[B];
c = mem[C];
a = b + c;
Mem[A] = a;
```

Κώδικας συμβολικής γλώσσας

```
LOAD B   ; AC = mem[B]
ADD C    ; AC = AC + mem[C]
STORE A  ; mem[A] = AC
```

Βασικές αρχιτεκτονικές συνόλου εντολών
Ανάλογα με το που βρίσκονται αποθηκευμένοι οι τελεστέοι

- Άμεση αποθήκευση σε καταχωρητή και μνήμη
  - *ο ένας από τους δύο τελεστέους πριν την εκτέλεση της πράξης καθώς και το αποτέλεσμα της πράξης αποθηκεύονται άμεσα σε έναν καταχωρητή (έστω R1) του αρχείου καταχωρητών*
  - *οι εντολές φόρτωσης από τη μνήμη στο αρχείο καταχωρητών (LOAD) και αποθήκευσης από το αρχείο καταχωρητών στη μνήμη (STORE) προσδιορίζουν:*
    - έναν τελεστέο που αποθηκεύεται άμεσα στη μνήμη και έναν τελεστέο που αποθηκεύεται άμεσα σε έναν καταχωρητή
  - *οι εντολές εκτέλεσης πράξεων προσδιορίζουν:*
    - έναν τελεστέο που αποθηκεύεται άμεσα στη μνήμη και έναν τελεστέο που αποθηκεύεται άμεσα σε έναν καταχωρητή

Κώδικας γλώσσας υψηλού επιπέδου

`b = mem[B];`
`c = mem[C];`
`a = b + c;`
`Mem[A] = a;`

Κώδικας συμβολικής γλώσσας

`LOAD R1,B ; R1 = mem[B]`
`ADD R1,C ; R1 = R1 + mem[C]`
`STORE R1,A; mem[A] = R1`

Βασικές αρχιτεκτονικές συνόλου εντολών

Ανάλογα με το που βρίσκονται αποθηκευμένοι οι τελεστέοι

- Άμεση αποθήκευση σε πολλούς (2 ή 3) καταχωρητές
  - *οι εντολές φόρτωσης (LOAD) και αποθήκευσης (STORE) **διαχωρίζονται** από τις εντολές εκτέλεσης πράξεων που χρησιμοποιούν αποκλειστικά το αρχείο καταχωρητών*
  - *οι εντολές LOAD και STORE προσδιορίζουν:*
    - έναν τελεστέο που αποθηκεύεται άμεσα στη μνήμη και έναν τελεστέο που αποθηκεύεται άμεσα σε έναν καταχωρητή
  - *οι εντολές εκτέλεσης πράξεων προσδιορίζουν:*
    - τρεις τελεστέους που αποθηκεύονται άμεσα σε 2 (ή 3) καταχωρητές του αρχείου καταχωρητών
    - στην περίπτωση των 2 καταχωρητών ο ένας από τους δύο τελεστέους πριν την εκτέλεση της πράξης καθώς και το αποτέλεσμα της πράξης αποθηκεύονται άμεσα στον ίδιο καταχωρητή, έστω R1 (**`ADD R1,R2`**)

Κώδικας γλώσσας υψηλού επιπέδου

```
b = mem[B];
c = mem[C];
a = b + c;
Mem[A] = a;
```

Κώδικας συμβολικής γλώσσας

```
LOAD R1,B     ; R1 = mem[B]
LOAD R2,C     ; R2 = mem[C]
ADD R3,R1,R2; R3 = R1 + R2
STORE A,R3    ; mem[A] = R3
```

Βασικές αρχιτεκτονικές συνόλου εντολών
Ανάλογα με το που βρίσκονται αποθηκευμένοι οι τελεστέοι

- Συγκρίσεις
  - *Οι βασικές αρχιτεκτονικές συνόλου εντολών έμμεσης αποθήκευσης είτε σε στοίβα, είτε σε συσσωρευτή στόχευαν στη μείωση του μεγέθους ενός προγράμματος σε γλώσσα μηχανής*
    - εμφανίσθηκαν πριν την έλευση των μικροεπεξεργαστών και των διατάξεων μνήμης σε ολοκληρωμένα κυκλώματα (RAM, ROM), όταν οι ολοκληρώσεις ήταν μικρές (SSI και MSI κυκλώματα)
  - *Οι βασικές αρχιτεκτονικές συνόλου εντολών άμεσης αποθήκευσης είτε σε καταχωρητή και μνήμη, είτε σε πολλούς καταχωρητές στόχευαν στη χρήση του αρχείου καταχωρητών, που αυξάνει την απόδοση του υπολογιστή και παρέχει ευελιξία*
    - η άμεση αποθήκευση σε καταχωρητή και μνήμη εμφανίσθηκε με την έλευση των μικροεπεξεργαστών (ως υπολογιστές CISC),
    - η άμεση αποθήκευση σε πολλούς καταχωρητές εμφανίσθηκε στις αρχές του 1980, όταν το επέτρεψε η τεχνολογική εξέλιξη στις διατάξεις μνήμης (με την έλευση των υπολογιστών RISC)

Επιλεγμένες ασκήσεις

- Ποια είναι η ακολουθία των εντολών σε συμβολική γλώσσα που υπολογίζουν τις κάτωθι αριθμητικές παραστάσεις για όλες τις βασικές αρχιτεκτονικές συνόλου εντολών:
  - $Y = A + [10 \times (B - C)]$
  - $Y = (A+10) \times (B-C)^2$

  όπου τα Y, A, B, C είναι διευθύνσεις μνήμης

Επιλεγμένες ασκήσεις

- Y = A + [10 x (B - C)], όπου τα Y, A, B, C είναι διευθύνσεις μνήμης
- Έμμεση αποθήκευση σε στοίβα (stack)
  - ***8 εντολές*** *(στην αρχή και στο τέλος η στοίβα είναι μηδενισμένη)*

PUSH C	TOS = c (c = mem[C])
PUSH B	TOS = b / c (b = mem[B])
SUB	TOS = (b - c)
PUSH 10	TOS = 10 / (b - c)
MUL*	TOS = [10 x (b - c)]
PUSH A	TOS = a / [10 x (b - c)] (a = mem[A])
ADD	TOS = a + [10 x (b - c)]
POP Y	mem[Y] = TOS (TOS = 0)

TOS: top of stack. TOS / $TOS_{previous}$. TOS = TOS operation $TOS_{previous}$

*Στην εντολή MUL μόνο το δεξιό μισό του γινομένου αποθηκεύεται στο TOS.

Επιλεγμένες ασκήσεις

- Y = A + [10 x (B - C)], όπου τα Y, A, B, C είναι διευθύνσεις μνήμης
- Έμμεση αποθήκευση σε συσσωρευτή (accumulator)
  - *5 εντολές*

LOAD B	AC = b (b = mem[B])
SUB C	AC = AC - c = (b - c) (c = mem[C])
MULI 10*	AC = AC x 10 = [(b - c) x 10]
ADD A	AC = AC + mem[A] = [(b - c) x 10] + a (a = mem[A])
STORE Y	mem[Y] = AC

*Εντολή τύπου Immediate για πράξη με σταθερά αποθηκευμένη στην ίδια την εντολή. Στην εντολή MULI μόνο το δεξιό μισό του γινομένου αποθηκεύεται στο συσσωρευτή.

Επιλεγμένες ασκήσεις

- Y = A + [10 x (B - C)], όπου τα Y, A, B, C είναι διευθύνσεις μνήμης
- Άμεση αποθήκευση σε καταχωρητή και μνήμη
  - *5 εντολές*

LOAD R1, B	R1 = b (b = mem[B])
SUB R1, C	R1 = R1 - c = (b - c) (c = mem[C])
MULI R1, 10*	R1 = R1 x 10 = [(b - c) x 10]
ADD R1, A	R1 = R1 + a = [(b - c) x 10] + a (a = mem[A])
STORE R1, Y	mem[Y] = R1

*Εντολή τύπου Immediate για πράξη με σταθερά αποθηκευμένη στην ίδια την εντολή. Στην εντολή MULI μόνο το δεξιό μισό του γινομένου αποθηκεύεται στον καταχωρητή.

Επιλεγμένες ασκήσεις

- Y = A + [10 x (B - C)], όπου τα Y, A, B, C είναι διευθύνσεις μνήμης
- Άμεση αποθήκευση σε 3 καταχωρητές (για εκτέλεση πράξεων)
  - *7 εντολές*

LOAD R1, B	R1 = b (b = mem[B])
LOAD R2, C	R2 = c (c = mem[C])
LOAD R3, A	R3 = a (a =mem[A])
SUB R4, R1, R2	R4 = R1 - R2 = (b – c)
MULI R4, R4, 10*	R4 = R4 x 10 = [(b - c) x 10]
ADD R4, R4, R3	R4 = R4 + R3 = [(b - c) x 10] + a
STORE Y, R4	mem[Y] = R4

*Εντολή τύπου Immediate για πράξη με σταθερά αποθηκευμένη στην ίδια την εντολή. Στην εντολή MULI μόνο το δεξιό μισό του γινομένου αποθηκεύεται στον καταχωρητή.

Υπολογιστές CISC vs RISC

- Οι υπολογιστές CISC (complex instruction set computer) υλοποιούν απλές και σύνθετες εντολές (π.χ. x86)
  - *Το απαιτούμενο υλικό για την αποκωδικοποίηση και εκτέλεση των εντολών είναι ιδιαίτερα σύνθετο, μεγάλο και αργό ακόμα και για τις απλές εντολές*
- Οι υπολογιστές RISC (reduced instruction set computer) υλοποιούν μόνο απλές εντολές
  - *εμφανίσθηκαν στην αρχή της δεκαετίας του 1980 και βασίζονται στην παρατήρηση ότι η μεγάλη πλειοψηφία των εντολών που εμφανίζονται συχνά σε ένα πρόγραμμα είναι απλές εντολές*
  - *το απαιτούμενο υλικό για την αποκωδικοποίηση και εκτέλεση των εντολών διατηρείται απλό, μικρό και γρήγορο*
    - Μια εντολή σε έναν υπολογιστή CISC απαιτεί πολλές, ακόμα και εκατοντάδες, απλές εντολές σε έναν υπολογιστή RISC
    - Τα προγράμματα των υπολογιστών RISC έχουν περισσότερες εντολές και καταλαμβάνουν περισσότερο χώρο στη μνήμη

Υπολογιστές CISC vs RISC

- Η μεγάλη πλειοψηφία των εντολών (>90%) που εμφανίζονται συχνά σε ένα πρόγραμμα είναι απλές εντολές
  - *Οι 10 πιο συχνές απλές εντολές της αρχιτεκτονικής x86 που κυριαρχούν σε 5 προγράμματα SPECint92*

Σειρά	Εντολή	Ποσοστό % των συνολικά εκτελουμένων
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Άθροισμα	96%

Αρχιτεκτονική RISC-V

- Η αρχιτεκτονική RISC-V ορίστηκε αρχικά το 2010 στο Πανεπιστήμιο Μπέρκλεϊ στην Καλιφόρνια από τους Krste Asanović, Andrew Waterman, David Patterson κ.ά. (5η Αρχιτεκτονική RISC)
  - *Από τότε πολλοί έχουν συνεισφέρει στην περαιτέρω ανάπτυξή της.*
- Έχει συνεχώς αυξανόμενη και ραγδαία δημοφηλία επειδή:
  - *είναι μία αρχιτεκτονική* ***ανοικτού πηγαίου κώδικά*** *που την καθιστά συνεχώς επεκτάσιμη*
    - διαθέτει **εντολές μεγέθους 32 bit**, αλλά και **συμπιεσμένες εντολές μεγέθους 16 bit**
    - διαθέτει εύρος δεδομένων **32,64,128 bit**
  - *Επιτρέπεται η* ***ελεύθερη χρήση*** *της τα τελευταία χρόνια σε πληθώρα εφαρμογών, ακόμη και διαστημικών*
    - χωρίς ωστόσο να υπολείπεται σε δυνατότητες συγκριτικά με άλλες επιτυχημένες εμπορικές αρχιτεκτονικές, όπως ARM

Krste Asanovic

- Είναι καθηγητής της επιστήμης των υπολογιστών στο Πανεπιστήμιο Berkeley στην Καλιφόρνια
- Ξεκίνησε να δημιουργεί την αρχιτεκτονική RISC-V ως καλοκαιρινή ενασχόληση
- Από το 2023, είναι πρόεδρος του Διοικητικού Συμβουλίου του Ιδρύματος RISC-V.
- Είναι επίσης συνιδρυτής της εταιρείας SiFive, η οποία αναπτύσσει και διαθέτει στην αγορά τσιπ, πλακέτες και εργαλεία υποστήριξης της αρχιτεκτονικής RISC-V.

Andrew Waterman

- Σχεδιάζει μικροεπεξεργαστές στην εταιρία SiFive, την οποία ίδρυσε το 2015 μαζί με τον Krste Asanović με σκοπό να παρέχει πυρήνες και προσαρμοσμένα τσιπ RISC-V χαμηλού κόστους.
- Απηυδισμένος από τις ιδιοτροπίες των υφιστάμενων αρχιτεκτονικών συνόλου εντολών, σχεδίασε με άλλους την αρχιτεκτονική RISC-V ISA και τους πρώτους πυρήνες RISC-V
- Είναι κάτοχος διδακτορικού τίτλου σπουδών (Ph.D.) στην επιστήμη των υπολογιστών από το Πανεπιστήμιο Μπέρκλεϊ (2016)

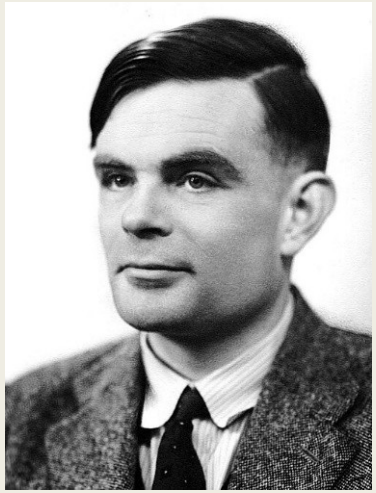
David Patterson

- Είναι καθηγητής της επιστήμης των υπολογιστών στο Πανεπιστήμιο Berkeley στην Καλιφόρνια από το 1976
- Μαζί με τον John Hennessy επινόησε το 1984 την υπολογιστική περιορισμένου συνόλου εντολών (Reduced Instruction Set Computing - RISC)
  - *αργότερα κυκλοφόρησε εμπορικά ως η αρχιτεκτονική SPARC*
- Συνέβαλε, και συνεχίζει να παίζει καθοριστικό ρόλο, στην ανάπτυξη της αρχιτεκτονικής RISC-V
- Το 2017 οι John Hennessy και David Patterson τιμήθηκαν με το βραβείο Turing (Turing Award) για το πρωτοποριακό έργο τους πάνω στη δημιουργία και υιοθέτηση μιας ποσοτικής προσέγγισης της σχεδίασης και αξιολόγησης αρχιτεκτονικών υπολογιστών

John Hennessy

- Είναι καθηγητής στο Τμήμα Ηλεκτρολόγων Μηχανικών και Επιστήμης των Υπολογιστών στο Πανεπιστήμιο Stanford, όπου θήτευσε και ως πρόεδρος του ιδρύματος από 2000 έως 2016.
- Μαζί με τον David Patterson επινόησε το 1984 την υπολογιστική περιορισμένου συνόλου εντολών (Reduced Instruction Set Computing - RISC)
  - *αργότερα κυκλοφόρησε εμπορικά ως η αρχιτεκτονική SPARC*
- Ανέπτυξε επίσης την αρχιτεκτονική MIPS και, μαζί με άλλους, ίδρυσε το 1984 την εταιρεία MIPS Computer Systems
  - *ο επεξεργαστής MIPS χρησιμοποιήθηκε σε πολλά εμπορικά συστήματα (προϊόντα των εταιρειών Silicon Graphics, Nintendo και Cisco*
- Το 2017 οι John Hennessy και David Patterson τιμήθηκαν με το βραβείο Turing (Turing Award) για το πρωτοποριακό έργο τους πάνω στη δημιουργία και υιοθέτηση μιας ποσοτικής προσέγγισης της σχεδίασης και αξιολόγησης αρχιτεκτονικών υπολογιστών

Alan Turing 1912–1954



- Βρετανός μαθηματικός ο οποίος θεωρείται πατέρας της θεωρητικής πληροφορικής και της τεχνητής νοημοσύνης.
- Εφηύρε τη **μηχανή Turing**, το μαθηματικό μοντέλο υπολογισμού ενός «αφηρημένου» επεξεργαστή
- Ανέπτυξε επίσης μια ηλεκτρομηχανική μηχανή για την αποκρυπτογράφηση κρυπτογραφημένων μηνυμάτων κατά τη διάρκεια του Β' Παγκόσμιου Πολέμου, εξαιτίας της οποίας το τέλος του πολέμου ήρθε πιο γρήγορα και σώθηκαν εκατομμύρια ζωές
- Το 1952 ο Turing διώχθηκε ποινικά για ομοφυλοφιλικές σεξουαλικές πράξεις και καταδικάστηκε σε χημικό ευνουχισμό αντί για φυλάκιση
- Δύο χρόνια αργότερα πέθανε από δηλητηρίαση με υδροκυάνιο
- Το ομώνυμο βραβείο (Turing Award), που αποτελεί την ύψιστη βράβευση στην επιστήμη των υπολογιστών, ονομάστηκε προς τιμήν του και απονέμεται κάθε χρόνο από το 1966
  - *Σήμερα συνοδεύεται από χρηματικό έπαθλο 1 εκατομμυρίου δολαρίων*

Αρχιτεκτονική RISC-V

- Η «*αρχιτεκτονική RISC-V*» που θα περιγράψουμε είναι η έκδοση 2.2 της αρχιτεκτονικής συνόλου εντολών RISC-V για πράξεις μεταξύ ακεραίων αριθμών μεγέθους 32 bit (RV32I).
  - *περιοριζόμαστε στις συνήθεις απλές εντολές της αρχιτεκτονικής συνόλου εντολών της RISC-V*
  - *διευθυνσιοδότηση της μνήμης στα 32 bit*
  - *λέξεις εντολών και δεδομένων μεγέθους 32 bit*
  - *αρχείο 32 καταχωρητών μεγέθους 32 bit*
    - ο program counter (PC) **δεν σχετίζεται με το αρχείο καταχωρητών**
- Οι λόγοι που επιλέξαμε να εστιάσουμε στην αρχιτεκτονική RISC-V είναι οι εξής:
  - *Η ραγδαία ανάπτυξή της δημοφιλίας της λόγω του ανοικτού κώδικα και των επεκτάσεών της (π.χ. εύρος δεδομένων 32, 64, 128 bit)*
  - *Είναι πιο εύκολη η υλοποίηση της μικροαρχιτεκτονικής της στο υλικό σε σχέση με άλλες μικροαρχιτεκτονικές RISC, όπως MIPS και ARM*

Το εγχειρίδιο του συνόλου εντολών της αρχιτεκτονικής RISC-V (RISC-V Instruction Set Manual) αποτελεί τον πιο έγκυρο ορισμό της αρχιτεκτονικής.

Αρχές σχεδίασης αρχιτεκτονικής

1. Η κανονικότητα ενισχύει την απλότητα
   - *Η σταθερή κωδικοποίηση εντολών που εμφανίζουν συνέπεια στο πλήθος των τελεστέων (δύο τελεστέοι προέλευσης και ένας τελεστής προορισμού) γίνεται πιο εύκολα στο υλικό.*
2. Οτιδήποτε είναι σύνηθες και χρησιμοποιείται συχνά πρέπει να είναι γρήγορο
   - *Το πλήθος των εντολών της αρχιτεκτονικής RISC-V διατηρείται εσκεμμένα μικρό (ορίζεται υπολογιστής RISC), έτσι ώστε:*
     - να καλύπτει τις συνήθεις και συχνά εμφανιζόμενες εντολές σε ένα πρόγραμμα, και
     - το απαιτούμενο υλικό να μπορεί να είναι απλό, μικρό και γρήγορο
3. Το μικρότερο μέγεθος του αποθηκευτικού μέσου συνεπάγεται πιο γρήγορη εκτέλεση της εντολής
   - *Η αρχιτεκτονική RISC-V χρησιμοποιεί ένα αρχείο καταχωρητών με 32 καταχωρητές μεγέθους 32 bit*
4. Η καλή σχεδίαση απαιτεί και καλούς συμβιβασμούς
   - *Η αρχιτεκτονική RISC-V ορίζει μόνο τέσσερεις κύριες μορφές εντολών: τύπου R, τύπου I, τύπου S/B, τύπου U/J*

Συμβολική γλώσσα – Η εντολή ADD

- Η πιο συνηθισμένη πράξη που εκτελούν οι υπολογιστές είναι η πρόσθεση (addition)

Κώδικας γλώσσας υψηλού επιπέδου
`a = b + c;`

Κώδικας συμβολικής γλώσσας της RISC-V
`ADD a, b, c`

- Η εντολή σε συμβολική γλώσσα γράφεται σε μία σειρά.
- Το πρώτο τμήμα της εντολής σε συμβολική γλώσσα, το ADD, ονομάζεται μνημονικό (*mnemonic*) και υποδεικνύει την πράξη που θα εκτελεστεί
- Η πράξη εκτελείται με τα b και c που ονομάζονται τελεστέοι προέλευσης (*source operands*) και αποθηκεύονται σε δύο καταχωρητές προέλευσης (source register 1 και source register 2)
- Το αποτέλεσμα εγγράφεται στο a, που ονομάζεται τελεστέος προορισμού (*destination operand*) και αποθηκεύεται σε έναν καταχωρητή προορισμού (destination register)

Συμβολική γλώσσα – Η εντολή SUB

- Η πράξη της αφαίρεσης (subtraction) είναι παρόμοια με την πράξη της πρόσθεσης

Κώδικας γλώσσας υψηλού επιπέδου
`a = b - c;`

Κώδικας συμβολικής γλώσσας της RISC-V
`SUB a, b, c`

- Στην αφαίρεση το μνημονικό είναι το SUB
- Το πλήθος των τελεστέων παραμένει ίδιο:
  - *δύο τελεστέοι προέλευσης (source operands) **b, c** που αποθηκεύονται σε δύο καταχωρητές προέλευσης (source register 1 και source register 2)*
  - *ένας τελεστέος προορισμού (destination operand) **a** που αποθηκεύεται σε έναν καταχωρητή προορισμού (destination register)*

Συμβολική γλώσσα – Σχόλια

- Στη συμβολική γλώσσα RISC-V χρησιμοποιούνται μόνο σχόλια μίας γραμμής
- Τα σχόλια ξεκινούν με το σύμβολο #
  και συνεχίζονται έως το τέλος της γραμμής

Κώδικας γλώσσας υψηλού επιπέδου

```
a = b + c;     // σχόλιο μίας γραμμής
/* σχόλιο πολλών γραμμών */
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
ADD a, b, c    # a = b + c
```

Συμβολική γλώσσα – Πιο σύνθετη εντολή

- Παράδειγμα εκτέλεσης σύνθετης εντολής υψηλού επιπέδου με πολλές εντολές της συμβολικής γλώσσας της αρχιτεκτονικής RISC-V
  - *Απαιτείται η χρήση της **προσωρινής μεταβλητής t**, στην οποία αποθηκεύεται το ενδιάμεσο αποτέλεσμα (b + c)*

Κώδικας γλώσσας υψηλού επιπέδου

```
a = b + c - d;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
ADD t, b, c          # t = b + c
SUB a, t, d          # a = t - d
```

Συμβολική γλώσσα – Τελεστέοι

- Στις εντολές προσδιορίζονται οι τελεστέοι ως τα απαιτούμενα δεδομένα που χρησιμοποιούνται κατά την εκτέλεση μίας πράξης καθώς και το αποτέλεσμα της πράξης
- Οι τελεστέοι μπορεί να είναι είτε μεταβλητές που αποθηκεύονται στο αρχείο των 32 καταχωρητών μεγέθους 32 bit ή στη μνήμη, είτε σταθερές που αποθηκεύονται σε κάποιο πεδίο της εντολής
  - *οι σταθερές στην ίδια την εντολή και οι μεταβλητές που αποθηκεύονται στο αρχείο καταχωρητών προσπελάζονται γρήγορα, αλλά διαθέτουν μικρή χωρητικότητα δεδομένων*
  - *Οι μεταβλητές που αποθηκεύονται στη μνήμη προσπελάζονται αργά, αλλά διαθέτουν μεγάλη χωρητικότητα δεδομένων*
  - *Ο χρόνος εκτέλεσης της εντολής εξαρτάται από τον χρόνο προσπέλασης και τη χωρητικότητα δεδομένων*
    - Το ιεραρχικό σύστημα μνήμης με αρχείο καταχωρητών και κρυφές μνήμες επιπέδων L1, L2 και ενδεχομένως L3, τεχνολογίας SRAM, καθώς και κύρια μνήμη, τεχνολογίας DRAM, στοχεύει στη βελτιστοποίηση του χρόνου προσπέλασης και της χωρητικότητας δεδομένων με κατάλληλους μηχανισμούς μεταφοράς δεδομένων

Σύνολο καταχωρητών RISC-V

- Αποθηκεύονται στο αρχείο 32 καταχωρητών των 32 bit

Όνομασία	Αρ. καταχωρητή	Χρήση
zero	x0	Σταθερή τιμή 0 (Δεν εγγράφεται)
ra	x1	Επιστρεφόμενη τιμή (return address)
sp	x2	Δείκτης (κορυφής) στοίβας (stack pointer)
gp	x3	Καθολικός δείκτης (global pointer)
tp	x4	Δείκτης νήματος (thread pointer)
t0–t2	x5-7	Προσωρινοί (temporary) καταχωρητές t0-t2
s0/fp	x8	Αποθηκευμένος (saved) καταχωρητής s0 ή Δείκτης πλαισίου (στοίβας) (frame pointer)
s1	x9	Αποθηκευμένος καταχωρητής s1
a0–a1	x10-11	Ορίσματα συναρτήσεων/Επιστρεφόμενες τιμές
a2–a7	x12-17	Ορίσματα συναρτήσεων
s2–s11	x18-27	Αποθηκευμένοι καταχωρητές s2-s11
t3–t6	x28-31	Προσωρινοί καταχωρητές t3-t6

Στους αποθηκευμένους καταχωρητές (s) κρατάμε τις μεταβλητές, ενώ στους προσωρινούς καταχωρητές (t) κρατάμε τα ενδιάμεσα αποτελέσματα των πράξεων

Συμβολική γλώσσα – Καταχωρητές ως τελεστέοι

- Παράδειγμα της εντολής ADD με καταχωρητές ως τελεστέους:
  - *Χρησιμοποιούμε τα ονόματα από το σύνολο καταχωρητών της αρχιτεκτονικής RISC-V*
  - *Οι μεταβλητές a, b και c τοποθετούνται αυθαίρετα στους αποθηκευμένους καταχωρητές s0, s1 και s2 του αρχείου καταχωρητών*
  - *Οι εντολές που χρησιμοποιούν δύο καταχωρητές προέλευσης (για αποθήκευση μεταβλητών) είναι **τύπου R (Register)***

Κώδικας γλώσσας υψηλού επιπέδου

```
a = b + c;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a, s1 = b, s2 = c
ADD s0, s1, s2 # a = b + c
```

Σχόλιο σε ολόκληρη γραμμή με #

Συμβολική γλώσσα – Προσωρινοί καταχωρητές

- Παράδειγμα εκτέλεσης εντολών με αποθήκευση αποτελέσματος ενδιάμεσου υπολογισμού
  - *Το ενδιάμεσο αποτέλεσμα της πράξης b + c αποθηκεύεται στον προσωρινό καταχωρητή t0 του αρχείου καταχωρητών*

Κώδικας γλώσσας υψηλού επιπέδου

```
a = b + c - d;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a, s1 = b, s2 = c, s3 = d, t0 = t
ADD t0, s1, s2  # t = b + c
SUB s0, t0, s3  # a = t - d
```

Συμβολική γλώσσα – Καταχωρητές ως τελεστέοι

- Παράδειγμα μετάφρασης κώδικα υψηλού επιπέδου σε συμβολική γλώσσα RISC-V
  - *Χρησιμοποιούνται 8 αποθηκευμένοι καταχωρητές s0-s7 για τις μεταβλητές και 2 προσωρινοί καταχωρητές t0 και t1 για αποθήκευση αποτελεσμάτων ενδιάμεσων υπολογισμών*

Κώδικας γλώσσας υψηλού επιπέδου

```
a = b – c;
f = (g + h) – (i + j);
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a, s1 = b, s2 = c, s3 = f
# s4 = g, s5 = h, s6 = i, s7 = j
SUB s0, s1, s2   # a = b – c
ADD t0, s4, s5   # t0 = g + h
ADD t1, s6, s7   # t1 = i + j
SUB s3, t0, t1   # f = (g + h) – (i + j)
```

Συμβολική γλώσσα – Άμεσοι τελεστέοι

- Εκτός από πράξεις με καταχωρητές, οι εντολές RISC-V μπορούν να χρησιμοποιούν σταθερές ή *άμεσους* τελεστέους
- Αυτές οι σταθερές ονομάζονται άμεσοι τελεστέοι επειδή οι τιμές τους είναι άμεσα αποθηκευμένες στην ίδια την εντολή και δεν απαιτούν την προσπέλαση κάποιου καταχωρητή ή μνήμης
  - *Επιτρέπεται η χρήση ενός μόνο άμεσου τελεστέου στην εντολή*
- Στην αρχιτεκτονική RISC-V, οι άμεσοι τελεστέοι είναι προσημασμένοι αριθμοί των 12 bit σε απεικόνιση συμπληρώματος ως προς 2 και το μέγεθός τους αυξάνεται στα 32 bit μέσω **επέκτασης προσήμου**
- Το αποτέλεσμα της πράξης αποθηκεύεται πάντα σε καταχωρητή προορισμού
- Οι εντολές που χρησιμοποιούν έναν καταχωρητή προέλευσης (για αποθήκευση μίας μεταβλητής) και έναν άμεσο τελεστέο (για αποθήκευση μίας σταθεράς) είναι **τύπου I (Immediate)**
- Στη συμβολική γλώσσα RISC-V, ένας άμεσος τελεστέος
  - *γράφεται ως έχει εάν είναι δεκαδικός αριθμός (default)*
    - **είτε θετικός, είτε αρνητικός με (-) μπροστά**
  - *ξεκινάει με πρόθεμα 0x, εάν είναι δεκαεξαδικός αριθμός*
  - *ξεκινάει με πρόθεμα 0b, εάν είναι δυαδικός αριθμός*

Συμβολική γλώσσα – Άμεσοι τελεστέοι

- Παραδείγματα της εντολής **ADDI** (δεν απαιτείται η ύπαρξη εντολής **SUBI**) με **προσημασμένους** άμεσους τελεστέους
  - *Οι μεταβλητές a και b τοποθετούνται αυθαίρετα στους αποθηκευμένους καταχωρητές s0 και s1 του αρχείου καταχωρητών*

Κώδικας γλώσσας υψηλού επιπέδου

```
a = a + 4;
b = a - 12;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a, s1 = b
ADDI s0, s0, 4          # a = a + 4
ADDI s1, s0, -12        # b = a - 12
```

Συμβολική γλώσσα – Σταθερές 12 bit

- Η εντολή πρόσθεσης τύπου I (ADDI) αποτελεί έναν χρήσιμο τρόπο για τον καθορισμό αρχικών τιμών μέχρι 12 bit [-2048, 2047] στους καταχωρητές με τη χρήση άμεσων τελεστέων

Κώδικας γλώσσας υψηλού επιπέδου

```
i = 0;
x = 2032;
y = -78;
Z = 109;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s4 = i, s5 = x, s6 = y, s7 = z
ADDI s4, zero, 0            # i = 0
ADDI s5, zero, 2032         # x = 2032
ADDI s6, zero, -78          # y = -78
ADDI s7, zero, 109          # z = 109
ADDI s7, zero, 0x6D         # z = 109
ADDI s7, zero, 0b01101101   # z = 109
```

Η ίδια εντολή με τρεις διαφορετικούς τρόπους

Συμβολική γλώσσα – Σταθερές 32 bit (Εντολή LUI)

- Η εντολές φόρτωσης αριστερού άμεσου τελεστέου (load upper immediate, LUI) και πρόσθεσης τύπου I (ADDI) χρησιμοποιούνται συνδυαστικά για τον καθορισμό αρχικών τιμών μέχρι 32 bit [-2^{32}, $2^{32}-1$] στους καταχωρητές με τη χρήση άμεσων τελεστέων
- Η εντολή LUI (τύπου U/J) φορτώνει έναν άμεσο τελεστέο μεγέθους 20 bit στα 20 περισσότερο σημαντικά bit ενός καταχωρητή προορισμού και συμπληρώνει τα λιγότερο σημαντικά bit με την τιμή 0

Κώδικας γλώσσας υψηλού επιπέδου

```
int a = 0xABCDE123
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a
LUI  s0, 0xABCDE        # s0 = 0xABCDE000
ADDI s0, s0, 0x123      # s0 = 0xABCDE123
```

```
  0xABCDE000
+ 0x00000123   (επέκταση προσήμου ενός θετικού)
  ----------
  0xABCDE123   (σωστό αποτέλεσμα)
```

Συμβολική γλώσσα – Σταθερές 32 bit (Εντολή LUI)

Κώδικας γλώσσας υψηλού επιπέδου

```
int a = 0xABCDE987
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a
LUI  s0, 0xABCDE        # s0 = 0xABCDE000
ADDI s0, s0, 0x987      # s0 = 0xABCDD987
```

```
   0xABCDE000
 + 0xFFFFF987 (επέκταση προσήμου ενός αρνητικού)
  0x1ABCDD987  (λάθος αποτέλεσμα)
```

0xFFFFF987 = -1657

- Απαιτείται διόρθωση, όταν η σταθερά στην εντολή ADDI είναι αρνητικός αριθμός!
  - *Αύξηση της σταθεράς των 20 bit στην LUI κατά 1*

Συμβολική γλώσσα – Σταθερές 32 bit (Εντολή LUI)

Κώδικας γλώσσας υψηλού επιπέδου

```
int a = 0xABCDE987
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = a
LUI  s0, 0xABCDF          # s0 = 0xABCDF000
ADDI s0, s0, 0x987        # s0 = 0xABCDE987
```

```
   0xABCDF000
 + 0xFFFFF987 (επέκταση προσήμου ενός αρνητικού)
  0x1ABCDE987  (σωστό αποτέλεσμα)
```

0xFFFFF987 = -1657

0xE + 1 = 14 +1 = 15 = 0xF

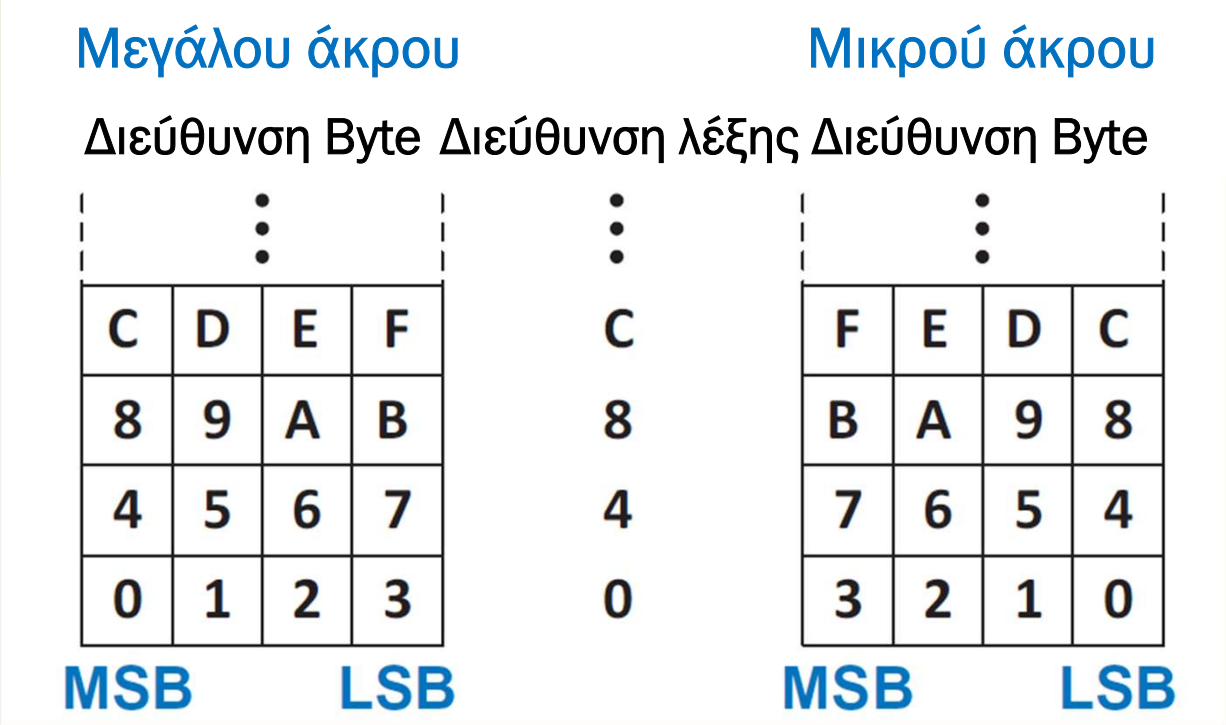
- Απαιτείται διόρθωση, όταν η σταθερά στην εντολή ADDI είναι αρνητικός αριθμός!
  - *Αύξηση της σταθεράς των 20 bit στην LUI κατά 1*

Διαχείριση μνήμης στην αρχιτεκτονική RISC-V

- Εκτός από τους καταχωρητές, τα δεδομένα μπορούν επίσης να αποθηκεύονται στη μνήμη δεδομένων
  - *η μνήμη είναι πολύ πιο μεγάλη, αλλά και πιο αργή*
- Στην αρχιτεκτονική RISC-V, οι πράξεις εκτελούνται αποκλειστικά με καταχωρητές
  - *τα δεδομένα που είναι αποθηκευμένα στη μνήμη πρέπει να μετακινηθούν πρώτα σε έναν καταχωρητή, ώστε να είναι εφικτή η επεξεργασία τους*
- Η αρχιτεκτονική RISC-V χρησιμοποιεί το μέγεθος των 32 bit τόσο για τις διευθύνσεις μνήμης όσο και για τις λέξεις δεδομένων
- Στην αρχιτεκτονική RISC-V η μνήμη δεδομένων είναι προσπελάσιμη κατά byte (byte addressable), με ευθυγραμμισμένες διευθύνσεις, όπου διευθυνσιοδοτείται το μικρό άκρο της λέξης δεδομένων (little endian)
  - *Μια λέξη δεδομένων μεγέθους 32 bit αποτελείται από 4 byte, άρα η διεύθυνση κάθε λέξης είναι ευθυγραμμισμένη σε πολλαπλάσια του 4*
  - *Η διεύθυνση της λέξης δεδομένων ταυτίζεται με τη διεύθυνση του λιγότερου σημαντικού byte (least significant byte – LSB) της λέξης δεδομένων (διευθυνσιοδότηση μικρού άκρου)*

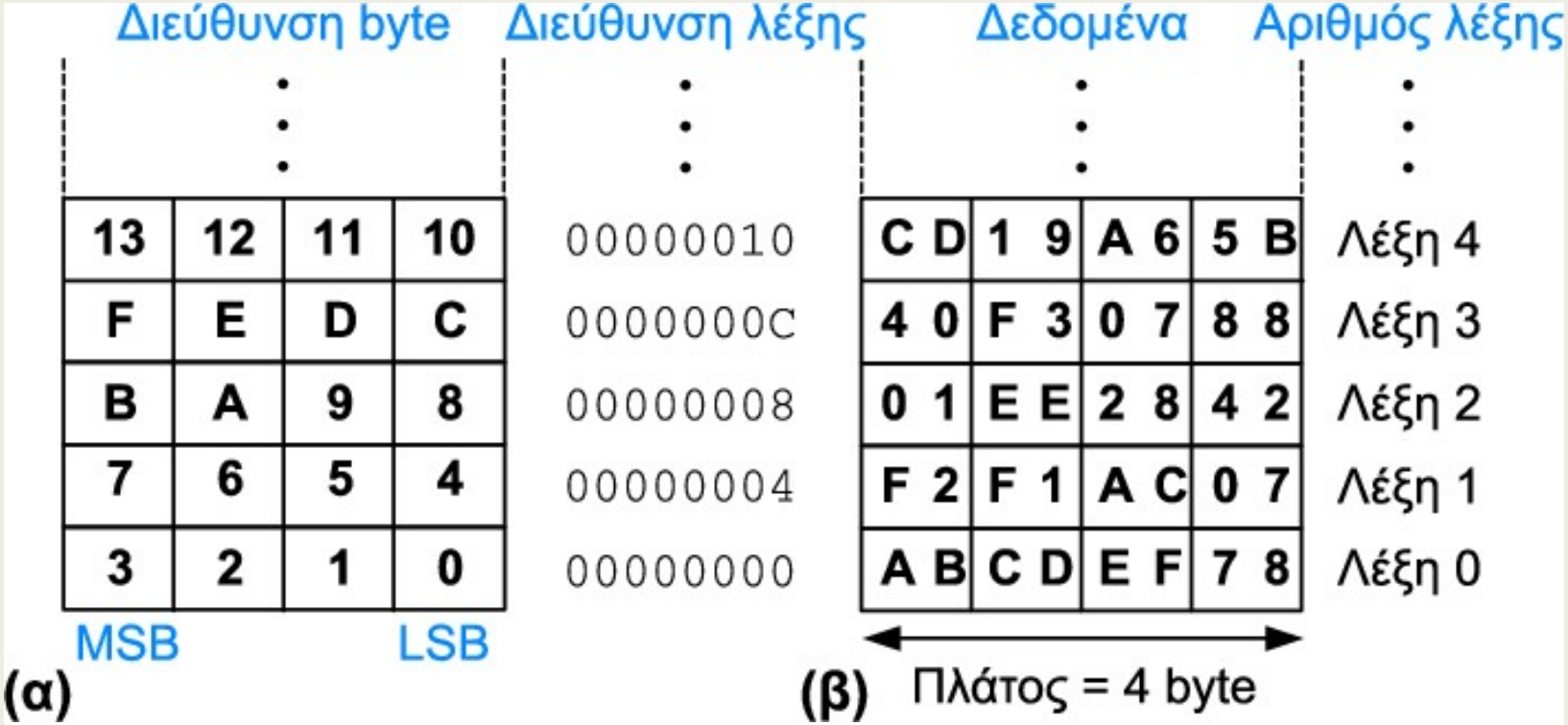
Διαχείριση μνήμης στην αρχιτεκτονική RISC-V

- Διευθυνσιοδότηση μικρού άκρου (little-endian) της μνήμης δεδομένων
  - *Η διεύθυνση της λέξης δεδομένων ταυτίζεται με τη διεύθυνση του λιγότερου σημαντικού byte (least significant byte – LSB) της λέξης δεδομένων*
- Διευθυνσιοδότηση μεγάλου άκρου (big-endian) της μνήμης δεδομένων
  - *Η διεύθυνση της λέξης δεδομένων ταυτίζεται με τη διεύθυνση του περισσοτερου σημαντικού byte (most significant byte – MSB) της λέξης δεδομένων*
- Η αρχιτεκτονική RISC-V χρησιμοποιεί συνήθως την διευθυνσιοδότηση μικρού άκρου, αν και υπάρχει παραλλαγή που χρησιμοποιεί την διευθυνσιοδότηση μεγάλου άκρου



Διαχείριση μνήμης στην αρχιτεκτονική RISC-V

- Θα χρησιμοποιήσουμε διευθυνσιοδότηση μικρού άκρου και προσπέλαση κατά byte
  - *Η διεύθυνση της λέξης ταυτίζεται με τη διεύθυνση του LSB της λέξης και είναι ευθυγραμμισμένη (σε πολλαπλάσια του 4)*



(α) διευθύνσεις byte και λέξης και (β) δεδομένα μνήμης

Διαχείριση μνήμης στην αρχιτεκτονική RISC-V

- Όπως και τα δεδομένα, έτσι και οι εντολές αποθηκεύονται στη **μνήμη εντολών**
- Η αρχιτεκτονική RISC-V διαθέτει **εντολές μεγέθους 32 bit** (με τις οποίες θα ασχοληθούμε στο πλαίσιο αυτού του μαθήματος), αλλά και **συμπιεσμένες εντολές μεγέθους 16 bit**
- Στην αρχιτεκτονική RISC-V η μνήμη εντολών είναι **προσπελάσιμη κατά byte** (byte addressable), με **ευθυγραμμισμένες διευθύνσεις**, όπου διευθυνσιοδοτείται το **μικρό άκρο** της λέξης δεδομένων (little endian)
  - *Μια εντολή μεγέθους 32 bit αποτελείται από 4 byte, άρα η διεύθυνση κάθε εντολής είναι **ευθυγραμμισμένη** σε πολλαπλάσια του 4*
  - *Η διεύθυνση της εντολής ταυτίζεται με τη διεύθυνση του **λιγότερου σημαντικού byte (least significant byte – LSB)** της εντολής (διευθυνσιοδότηση μικρού άκρου)*
- Ο **μετρητής προγράμματος (program counter, PC)** παρακολουθεί την **τρέχουσα εντολή**
  - *Σε αυτόν αποθηκεύεται η διεύθυνση μνήμης της τρέχουσας εντολής, η οποία αυξάνεται κατά 4/2 μετά την εκτέλεση κάθε εντολής των 32/16 bit, ώστε ο επεξεργαστής να μπορεί να προσκομίσει (fetch) την επόμενη εντολή από τη μνήμη εντολών*
  - *Το λιγότερο σημαντικό ψηφίο του μετρητή PC είναι **πάντα '0'** λόγω της ευθυγράμμισης (είτε πρόκειται για εντολές των 32 bit, είτε των 16 bit)*

Συμβολική γλώσσα – Εντολές μνήμης

- Για τον προσδιορισμό της διεύθυνσης της λέξης δεδομένων στη μνήμη δεδομένων χρησιμοποιείται ο τρόπος διευθυνσιοδότησης βάσης (με σχετική απόσταση) (base addressing with offset) που χρησιμοποιεί:
  - *έναν καταχωρητή βάσης (base register) του αρχείου καταχωρητών που περιέχει τη διεύθυνση βάσης*
    - χρησιμοποιείται ο καταχωρητής προέλευσης 1 (source register 1, Rs1)
  - *μια σχετική απόσταση (offset) σε bytes ως προσημασμένος ακέραιος των 12 bit που είναι άμεσα αποθηκευμένος στην ίδια την εντολή*
    - Προστίθεται στην τιμή του καταχωρητή βάσης μετά από επέκταση προσήμου από τα 12 bit στα 32 bit για να προσδιορισθεί η διεύθυνση της λέξης δεδομένων
    - Η λέξη δεδομένων στη συγκεκριμένη διεύθυνση της μνήμης συμβολίζεται ως **`mem[Rs1+S_Ext(imm)]`**
  - *Η διεύθυνση της λέξης δεδομένων των 32 bit είναι ευθυγραμμισμένη (πολλαπλάσια του 4 – λήγει σε 0, 4, 8, C), ώστε να υλοποιείται εύκολα στο υλικό η μεταφορά δεδομένων από και προς τη μνήμη*
    - Την ευθυγράμμιση ακολουθεί και το offset

Το offset γράφεται ως δεκαδικός αριθμός (default) ή ως δεκαεξαδικός αριθμός

Συμβολική γλώσσα – Η εντολή LW

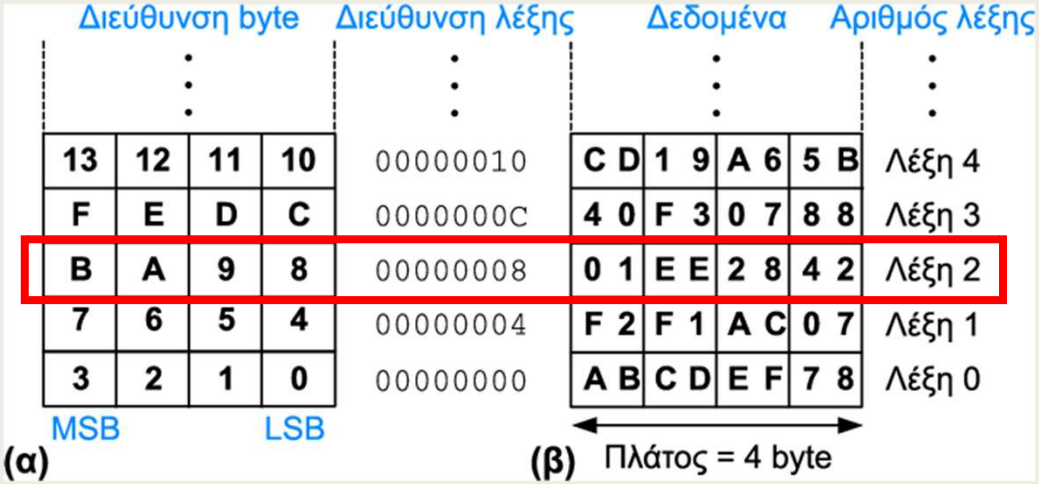
- Η αρχιτεκτονική RISC-V ορίζει την εντολή φόρτωσης λέξης δεδομένων (*load word, LW*) (τύπου I) με την οποία μια λέξη δεδομένων διαβάζεται από τη μνήμη και φορτώνεται σε έναν καταχωρητή του αρχείου καταχωρητών
  - *Μορφή εντολής: LW destination register Rd, offset(source register Rs1)*
  - *Πράξη εντολής:* **`Rd = mem[Rs1+S_Ext(imm)]`**
  - *Παράδειγμα: Μετά την εκτέλεση της εντολής LW, στον καταχωρητή s7 υπάρχει η τιμή που είναι αποθηκευμένη στη διεύθυνση 8 της μνήμης*

Κώδικας γλώσσας υψηλού επιπέδου

```
a = mem[2];
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s7 = a
LW s7, 8(zero)      # s7 = 0x01EE2842
```



Δεδομένα από την διεύθυνση 0+8=8 σε bytes της μνήμης ή την διεύθυνση 2 σε λέξεις

Συμβολική γλώσσα – Η εντολή SW

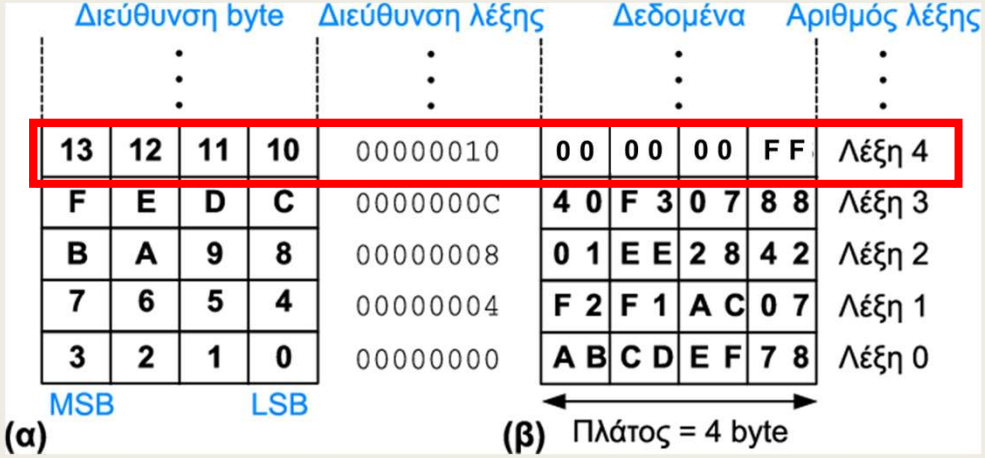
- Η αρχιτεκτονική RISC-V ορίζει την εντολή αποθήκευσης λέξης δεδομένων (*store word, SW*) (τύπου S/B) με την οποία εγγράφει στην μνήμη δεδομένων μια λέξη δεδομένων που είναι αποθηκευμένη στο αρχείο καταχωρητών
  - *Μορφή εντολής: SW source register Rs2, offset(source register Rs1)*
  - *Πράξη εντολής:* **`mem[Rs1+S_Ext(imm)] = Rs2`**
  - *Παράδειγμα: Μετά την εκτέλεση της εντολής SW, το περιεχόμενο (255) του t3 αποθηκεύεται στη διεύθυνση 16 = 0x10 (hex) της μνήμης*

Κώδικας γλώσσας υψηλού επιπέδου

```
mem[4] = 255;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
ADDI t3, zero, 255   # t3 = 0x000000FF = 255
SW   t3, 0x10(zero)  # mem[4] = 0x000000FF = 255
```



Το περιεχόμενο του καταχωρητή t3 (255) αποθηκεύεται στην διεύθυνση 0+16=16 σε bytes της μνήμης ή στην διεύθυνση 4 σε λέξεις

Συμβολική γλώσσα – Λογικές εντολές

- Η αρχιτεκτονική RISC-V ορίζει τις ακόλουθες λογικές εντολές για την εκτέλεση *λογικών πράξεων* σε επίπεδο bit (*bitwise*):
  - *AND*, *OR*, *XOR* *(τύπου R)*
  - *ANDI*, *ORI*, *XORI* *(τύπου I)*
- Για την εκτέλεση της λογικής πράξης χρησιμοποιούνται δύο τελεστέοι προέλευσης:
  - *ο πρώτος τελεστέος προέλευσης είναι πάντα αποθηκευμένος σε έναν καταχωρητή προέλευσης*
  - *ο δεύτερος τελεστέος προέλευσης μπορεί να είναι είτε άμεσος τελεστέος μεγέθους 12 bit* ***με επέκταση προσήμου*** *(στις εντολές τύπου I), είτε τελεστέος αποθηκευμένος σε έναν καταχωρητή προέλευσης (στις εντολές τύπου R)*
- Μετά την εκτέλεση της λογικής πράξης προκύπτει ένας τελεστέος προορισμού, που αποθηκεύεται σε έναν καταχωρητή προορισμού

Προσοχή! Οι λογικοί άμεσοι τελεστέοι με επέκταση προσήμου είναι κάπως ασυνήθιστοι.

Πολλές άλλες αρχιτεκτονικές, όπως η MIPS και η ARM, επεκτείνουν με μηδενικά τον άμεσο τελεστέο για λογικές πράξεις.

Συμβολική γλώσσα – Λογικές εντολές

- Πρακτική χρήση των λογικών εντολών
- AND/ANDI
  - *Εκκαθάριση ψηφίων ή εφαρμογή μάσκας (με ανάθεση της τιμής 0 σε συγκεκριμένα bit)*
  - *Παράδειγμα: 0xF234012F AND 0x000000FF = 0x0000002F*
- OR/ORI
  - *Ανάμειξη πεδίων δύο καταχωρητών*
  - *Παράδειγμα: 0x347A0000 OR 0x000072FC = 0x347A72FC*
- XOR/XORI
  - *Αντιστροφή ψηφίων (συμπλήρωμα ως προς 1)*
  - *Παράδειγμα A XOR -1 = NOT A (-1 = 0xFFFFFFFF)*

Συμβολική γλώσσα – Λογικές εντολές

- Παραδείγματα:

		Source Registers			
	s1	0100 0110	1010 0001	1111 0001	1011 0111
	s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code		Result			
`and s3, s1, s2`	s3	0100 0110	1010 0001	0000 0000	0000 0000
`or  s4, s1, s2`	s4	1111 1111	1111 1111	1111 0001	1011 0111
`xor s5, s1, s2`	s5	1011 1001	0101 1110	1111 0001	1011 0111

Συμβολική γλώσσα – Λογικές εντολές

- Επιλεγμένες ασκήσεις:
  - *-1484 = 0xA34 σε αναπαράσταση συμπληρώματος ως προς 2 στα 12 bit*

Source Values

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100
					sign-extended	A	3	4

Assembly Code

```
andi s5, t3, -1484
ori  s6, t3, -1484
xori s7, t3, -1484
```

Result

s5	0011	1010	0111	0101	0000	1000	0010	0100
s6	1111	1111	1111	1111	1111	1111	0111	1111
s7	1100	0101	1000	1010	1111	0111	0101	1011

Προσοχή στην επέκταση προσήμου!

Συμβολική γλώσσα – Εντολές Ολίσθησης

- Η αρχιτεκτονική RISC-V ορίζει τις ακόλουθες εντολές ολίσθησης:
  - *SLL/SLLI* (***shift left logical, λογική ολίσθηση προς τα αριστερά***)
    - Οι αριστερές ολισθήσεις γεμίζουν τα λιγότερο σημαντικά κενά bit με μηδενικά
  - *SRL/SRLI* (***logical shift right, λογική ολίσθηση προς τα δεξιά***)
    - Οι δεξιές λογικές ολισθήσεις γεμίζουν τα περισσότερο σημαντικά κενά bit με μηδενικά
  - *SRA/SRAI* (***arithmetic shift right, αριθμητική ολίσθηση προς τα δεξιά***)
    - Οι δεξιές αριθμητικές ολισθήσεις γεμίζουν τα περισσότερο σημαντικά κενά bit με το bit του προσήμου
- Για την εκτέλεση της εντολής ολίσθησης με σταθερή ή μεταβλητή ποσότητα ολίσθησης χρησιμοποιούνται δύο τελεστέοι προέλευσης:
  - *ο πρώτος τελεστέος προέλευσης του οποίου το περιεχόμενο ολισθαίνει είναι πάντα αποθηκευμένος σε έναν καταχωρητή προέλευσης*
  - *ο δεύτερος τελεστέος προέλευσης ορίζει την* ***ποσότητα ολίσθησης*** *και είναι είτε άμεσος τελεστέος με τιμές από 0 μέχρι 31 (SLLI, SRLI, SRAI), είτε μεταβλητή αποθηκευμένη σε καταχωρητή προέλευσης (SLL, SRL, SRA)*
- Μετά την εκτέλεση της εντολής ολίσθησης προκύπτει ένας τελεστέος προορισμού, που αποθηκεύεται σε έναν καταχωρητή προορισμού

Συμβολική γλώσσα – Εντολές Ολίσθησης

- Παραδείγματα: Η μεταβλητή ποσότητα ολίσθησης είναι αποθηκευμένη στα 5 λιγότερα σημαντικά ψηφία του καταχωρητή t2
- SLL: shift left logical
  - *Παράδειγμα:* `SLL t0, t1, t2  # t0 = t1 << t2`
- SRL: shift right logical
  - *Παράδειγμα:* `SRL t0, t1, t2  # t0 = t1 >> t2`
- SRA: shift right arithmetic
  - *Παράδειγμα:* `SRA t0, t1, t2  # t0 = t1 >>> t2`

Συμβολική γλώσσα – Εντολές Ολίσθησης

- Παραδείγματα: Η σταθερή ποσότητα ολίσθησης μεγέθους 5 bit [0, 31] είναι αποθηκευμένη στην ίδια την εντολή
- SLLI: shift left logical immediate
  - *Παράδειγμα:* `SLL t0, t1, 23  # t0 = t1 << 23`
- SRLI: shift right logical immediate
  - *Παράδειγμα:* `SRL t0, t1, 18  # t0 = t1 >> 18`
- SRAI: shift right arithmetic immediate
  - *Παράδειγμα:* `SRA t0, t1, 5  # t0 = t1 >>> 5`

Συμβολική γλώσσα – Εντολή SLT

- Η εντολή SLT (set if less than) «θέσε εάν είναι μικρότερο από» (τύπου R) συγκρίνει το περιεχόμενο δύο τελεστέων προέλευσης και ενημερώνει έναν τελεστέο προορισμού με την τιμή:
  - *1, εάν ισχύει η συνθήκη, ή 0, εάν δεν ισχύει η συνθήκη*
- Μορφή εντολής: SLT destination reg., source reg. 1, source reg. 2
  - *Παράδειγμα:* **`SLT t0, s1, s2`** *(Εάν s1 < s2, τότε t0 = 1, αλλιώς t0 = 0)*
- Στην εντολή SLT η σύγκριση γίνεται μεταξύ προσημασμένων αριθμών
  - *το 0x80000000 είναι μικρότερο από οποιονδήποτε άλλο αριθμό επειδή είναι ο πιο αρνητικός αριθμός στο σύστημα συμπληρώματος ως προς δύο*
  - *Το 0xFFFFFFFF είναι το -1 και είναι μικρότερο από το 0*
- Για την εκτέλεση της εντολής SLT χρησιμοποιούνται δύο τελεστέοι προέλευσης που είναι αποθηκευμένοι αντίστοιχα σε δύο καταχωρητές προέλευσης και εκτελείται η πράξη της αφαίρεσης, ώστε να παραχθούν τα απαραίτητα ALU flags N,C,V
- Μετά την εκτέλεση της εντολής SLT προκύπτει το αποτέλεσμα 0 ή 1, που αποθηκεύεται σε έναν καταχωρητή προορισμού

Συμβολική γλώσσα – Εντολή SLTU

- Η εντολή SLTU (set if less than - unsigned) «θέσε εάν είναι μικρότερο από» (τύπου R) συγκρίνει το περιεχόμενο δύο τελεστέων προέλευσης και ενημερώνει έναν τελεστέο προορισμού με την τιμή:
  - *1, εάν ισχύει η συνθήκη, ή 0, εάν δεν ισχύει η συνθήκη*
- Μορφή εντολής: SLTU destination reg., source reg. 1, source reg. 2
  - *Παράδειγμα:* **`SLTU t0, s1, s2`** *(Εάν s1 < s2, τότε t0 = 1, αλλιώς t0 = 0)*
- Στην εντολή SLT η σύγκριση γίνεται μεταξύ μη προσημασμένων αριθμών
  - *το 0x80000000 είναι μεγαλύτερο από το 0x7FFFFFFF, αλλά μικρότερο από το 0x80000001, επειδή όλοι οι αριθμοί είναι θετικοί*
  - *Το 0xFFFFFFFF είναι ο μεγαλύτερος θετικός αριθμός*
- Για την εκτέλεση της εντολής SLT χρησιμοποιούνται δύο τελεστέοι προέλευσης που είναι αποθηκευμένοι αντίστοιχα σε δύο καταχωρητές προέλευσης και εκτελείται η πράξη της αφαίρεσης, ώστε να παραχθούν τα απαραίτητα ALU flags N,C,V
- Μετά την εκτέλεση της εντολής SLT προκύπτει το αποτέλεσμα 0 ή 1, που αποθηκεύεται σε έναν καταχωρητή προορισμού

Συμβολική γλώσσα – Εντολές διακλάδωσης Bxx(U)

- Η αρχιτεκτονική RISC-V υποστηρίζει εντολές διακλάδωσης υπό συνθήκη (conditional *branch instructions*) (τύπου S/B), ώστε να αγνοεί ή να επαναλαμβάνει τμήματα κώδικα
  - *Όταν ικανοποιείται η **συνθήκη της σύγκρισης**, οι εντολές διακλάδωσης αλλάζουν την ροή του προγράμματος τροποποιώντας το περιεχόμενο του program counter (PC), ώστε η επόμενη προς εκτέλεση εντολή να μην είναι η αμέσως επόμενη στη διεύθυνση PC+4*
- Μορφή εντολής: **Bxx(U) source reg. 1, source reg. 2, label**
  - Οι ετικέτες (label) υποδεικνύουν τη διεύθυνση της εντολής που θα εκτελεσθεί, όταν ικανοποιείται η συνθήκη (διεύθυνση στόχος διακλάδωσης (branch target address, BTA)
    - Οι ετικέτες (label) ακολουθούνται από άνω και κάτω τελεία (:)
  - Η διεύθυνση BTA δηλώνει απόσταση μεταξύ εντολών (όχι bytes) και μετατρέπεται σε άμεσο τελεστέο μεγέθους 12 bit (**immediate12**) κατά τη συμβολομετάφραση σε γλώσσα μηχανής
  - *Κατά την εκτέλεση της εντολής υπολογίζεται σε ξεχωριστό αθροιστή η διεύθυνση BTA ως:*

$$BTA = PC + sign\_ext(immediate12 \times 2)$$

Συμβολική γλώσσα – Εντολές διακλάδωσης Bxx(U)

- Η RISC-V έχει 4 εντολές διακλάδωσης για προσημασμένες συγκρίσεις και 2 εντολές διακλάδωσης για μη προσημασμένες συγκρίσεις (με U):
  - branch if equal (**`BEQ`**)
  - branch if not equal (**`BNE`**)
  - branch if less than (**`BLT/BLTU`**)
  - branch if greater than or equal (**`BGE/BGEU`**)
- Οι εντολές διακλάδωσης:
  - branch if greater than (**`BGT/BGTU`**)
  - branch if less than or equal (**`BLE/BLEU`**)

  ορίζονται ως ψευδοεντολές γιατί προκύπτουν με **απλή εναλλαγή των καταχωρητών προέλευσης (έστω s0 και s1)** των εντολών **`BLT/BLTU`** και **`BGE/BLEU`**, αντίστοιχα, γιατί:

  s0 < s1 => s1 > s0

  s0 ≥ s1 => s1 ≤ s0

Συμβολική γλώσσα – Εντολές διακλάδωσης Bxx

- Παράδειγμα διακλάδωσης υπό συνθήκη με την εντολή BEQ (6.12):
  - *Υιοθετούμε την απαίτηση ότι οι ετικέτες τοποθετούνται στην αρχή της γραμμής, χωρίς κενό, ενώ πριν από τις εντολές πρέπει να υπάρχει κενό*
  - *Οι ετικέτες δεν μπορεί να είναι δεσμευμένες λέξεις*

Κώδικας συμβολικής γλώσσας της RISC-V

```
 ADDI s0, zero, 4      # s0 = 0 + 4 = 4
 ADDI s1, zero, 1      # s1 = 0 + 1 = 1
 SLLI s1, s1, 2        # s1 = 1 << 2 = 4
 BEQ  s0, s1, TARGET   # s0==s1, ισχύει η συνθήκη
 ADDI s1, s1, 1        # δεν εκτελείται
 SUB  s1, s1, s0       # δεν εκτελείται
TARGET:                # ετικέτα
 ADD  s1, s1, s0       # s1 = 4 + 4 = 8
```

Όταν η εκτέλεση του προγράμματος φθάσει στην εντολή BEQ, η επόμενη εντολή που εκτελείται είναι η εντολή ADD που βρίσκεται αμέσως μετά από την ετικέτα με το όνομα TARGET

Συμβολική γλώσσα – Εντολές διακλάδωσης Bxx

- Παράδειγμα διακλάδωσης υπό συνθήκη με την εντολή BNE (6.13):
  - *Υιοθετούμε την απαίτηση ότι οι ετικέτες τοποθετούνται στην αρχή της γραμμής, χωρίς κενό, ενώ πριν από τις εντολές πρέπει να υπάρχει κενό*
  - *Οι ετικέτες δεν μπορεί να είναι δεσμευμένες λέξεις*

Κώδικας συμβολικής γλώσσας της RISC-V

```
 ADDI s0, zero, 4      # s0 = 0 + 4 = 4
 ADDI s1, zero, 1      # s1 = 0 + 1 = 1
 SLLI s1, s1, 2        # s1 = 1 << 2 = 4
 BNE  s0, s1, TARGET   # s0==s1, δεν ισχύει η συνθήκη
 ADDI s1, s1, 1        # s1 = 4 + 1 = 5
 SUB  s1, s1, s0       # s1 = 5 – 4 = 1
TARGET:                # ετικέτα
 ADD  s1, s1, s0       # s1 = 1 + 4 = 5
```

Εκτελούνται όλες οι εντολές του προγράμματος

Συμβολική γλώσσα – Υλοποίηση συγκρίσεων

- Στις εντολές SLT(U) και Bxx(U) τύπου R η σύγκριση γίνεται μεταξύ δύο τελεστέων προέλευσης που είναι αποθηκευμένοι αντίστοιχα σε καταχωρητές προέλευσης, έστω s0 και s1
  - Η σύγκριση υλοποιείται ως αφαίρεση στη μονάδα ALU (s0-s1) με κατάλληλη αξιοποίηση των παραγόμενων σημαίων (ALU flags)

ALU flags	Περιγραφή
N	Αρνητικό αποτέλεσμα (Negative)
Z	Μηδενικό αποτέλεσμα (Zero)
C	Ο αθροιστής παρήγαγε κρατούμενο εξόδου (Carry)
V	Ο αθροιστής υπερχείλισε (oVerflowed)

Εντολές SLT(U) και Bxx(U)	Συνθήκη σύγκρισης (=1)
BEQ	Z
BNE	$\bar{Z}$
SLT/BLT (s0-s1) & BGT (s1-s0)	$N \oplus V$
BGE (s0-s1) & BLE (s1-s0)	$\overline{N \oplus V}$
SLTU/BLTU (s0-s1) & BGTU (s1-s0)	$\bar{C}$
BGEU (s0-s1) & BLEU (s1-s0)	C

Συμβολική γλώσσα – Εντολές διακλάδωσης Bxx(U)

- Εάν είναι διαθέσιμες μόνο οι εντολές SLT(U) και BEQ/BNE, τότε οι εντολές BLT(U), BGE(U), BGT(U) και BLE(U) υλοποιούνται ως εξής:

Σύγκριση	Εντολές Bxx(U)	Υλοποίηση με εντολές SLT(U) και BEQ/BNE
`s0 = s1`	`BEQ s0, s1, label`	`BEQ s0, s1, label`
`s0 ≠ s1`	`BNE s0, s1, label`	`BNE s0, s1, label`
`s0 < s1`	`BLT(U) s0, s1, label`	`SLT(U) t0, s0, s1` `BNE t0, zero, label`
`s0 ≥ s1`	`BGE(U) s0, s1, label`	`SLT(U) t0, s0, s1` `BEQ t0, zero, label`
`s0 > s1`	`BGT(U) s0, s1, label`	`SLT(U) t0, s1, s0` `BNE t0, zero, label`
`s0 ≤ s1`	`BLE(U) s0, s1, label`	`SLT(U) t0, s1, s0` `BNE t0, zero, label`

Συμβολική γλώσσα – Εντολή άλματος J

- Η αρχιτεκτονική RISC-V υποστηρίζει εντολές (απλού) άλματος (jump) (τύπου U/J), χωρίς συνθήκη ώστε να συνεχίσει την ροή του προγράμματος από άλλο σημείο του κώδικα
- Μορφή εντολής: J label
  - Οι ετικέτες (label) υποδεικνύουν τη διεύθυνση της εντολής που θα συνεχιστεί η εκτέλεση του προγράμματος (διεύθυνση στόχος άλματος (jump target address, JTA)
    - Οι ετικέτες (label) ακολουθούνται από άνω και κάτω τελεία (:)
  - Η διεύθυνση JTA δηλώνει απόσταση μεταξύ εντολών (όχι bytes) και μετατρέπεται σε άμεσο τελεστέο μεγέθους 20 bit (immediate20) κατά τη συμβολομετάφραση σε γλώσσα μηχανής
  - *Κατά την εκτέλεση της εντολής υπολογίζεται σε ξεχωριστό αθροιστή η διεύθυνση JTA ως:*

$$JTA = PC + sign\_ext(immediate20 \times 2)$$

Τις εντολές JAL (jump and link), JALR (jump and link register) και JR (jump register) θα τις περιγράψουμε στην κλήση συναρτήσεων και επιστροφή από συναρτήσεις

Συμβολική γλώσσα – Εντολή άλματος J

- Παράδειγμα άλματος χωρίς συνθήκη με την εντολή J (6.14):
  - *Υιοθετούμε την απαίτηση ότι οι ετικέτες τοποθετούνται στην αρχή της γραμμής, χωρίς κενό, ενώ πριν από τις εντολές πρέπει να υπάρχει κενό*
  - *Οι ετικέτες δεν μπορεί να είναι δεσμευμένες λέξεις*

Κώδικας συμβολικής γλώσσας της RISC-V

```
 J     TARGET          # άλμα στο TARGET
 SRAI  s1, s1, 2       # δεν εκτελείται
 ADDI  s1, s1, 1       # δεν εκτελείται
 SUB   s1, s1, s0      # δεν εκτελείται
TARGET:
 ADD   s1, s1, s0      # s1 = s1 + s0
```

Μόλις εκτελεσθεί η εντολή J, η επόμενη εντολή που εκτελείται είναι η εντολή ADD που βρίσκεται αμέσως μετά από την ετικέτα με το όνομα TARGET

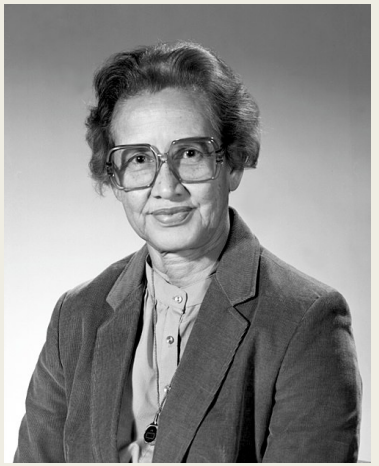
Προγραμματισμός

- Οι γλώσσες συγγραφής προγραμμάτων λογισμικού όπως η C ή η Java αποκαλούνται γλώσσες προγραμματισμού υψηλού επιπέδου επειδή είναι γραμμένες σε υψηλότερο επίπεδο «αφαίρεσης» από ό,τι η συμβολική γλώσσα
- Πολλές γλώσσες υψηλού επιπέδου χρησιμοποιούν κοινές δομές λογισμικού, όπως εντολές **if/else, βρόχους for και while, αριθμοδεικτοδότηση πινάκων και κλήσεις συναρτήσεων**
- Στη συνέχεια θα διερευνήσουμε πώς μπορούμε να «μεταφράζουμε» τις δομές υψηλού επιπέδου σε κώδικα συμβολικής γλώσσας για την αρχιτεκτονική RISC-V

Ada Lovelace, 1815–1852

- Βρετανίδα μαθηματικός που έγραψε το πρώτο δημοσιευμένο πρόγραμμα υπολογιστή.
  - *Το πρόγραμμα υπολόγιζε τους αριθμούς Bernoulli χρησιμοποιώντας την Αναλυτική Μηχανή (Analytical Engine) του Charles Babbage*
- Η Ada ήταν κόρη του ποιητή Λόρδου Βύρωνα
- Η γλώσσα υπολογιστών Ada, που δημιουργήθηκε για λογαριασμό του Υπουργείου Άμυνας των Ηνωμένων Πολιτειών, πήρε το όνομά της από τη Lovelace
  - *Το εγχειρίδιο αναφοράς για τη γλώσσα εγκρίθηκε στις 10 Δεκεμβρίου 1980 και το Στρατιωτικό Πρότυπο του Υπουργείου Άμυνας για τη γλώσσα, MIL-STD-1815, έλαβε τον αριθμό του έτους γέννησής της*
- Το 1981, ο Σύνδεσμος για τις Γυναίκες στην Πληροφορική εγκαινίασε το Βραβείο Ada Lovelace
- Από το 1998, η Βρετανική Εταιρεία Υπολογιστών (BCS) έχει απονείμει το Μετάλλιο Lovelace και το 2008 ξεκίνησε έναν ετήσιο διαγωνισμό για φοιτήτριες.

Katherine Johnson, 1918–2020



- Η Creola Katherine Johnson, η πρώτη Αφροαμερικανή που εργάστηκε στη NASA, ήταν μαθηματικός που ασχολήθηκε με την επιστήμη των υπολογιστών
- Σε ηλικία 18 ετών αποφοίτησε με άριστα από το Πανεπιστήμιο της West Virginia αποκτώντας δύο πτυχία, στα Μαθηματικά και στα Γαλλικά
- Στη NASA εργάστηκε ως «άνθρωπος-υπολογιστής»: μια ομάδα ανθρώπων, κυρίως γυναικών, που εκτελούσαν ακριβείς υπολογισμούς
- Το 1961 η Johnson υπολόγισε την τροχιά του Alan Shepard, του πρώτου Αμερικανού στο διάστημα
- Στα πρώτα βήματά της η NASA αποθάρρυνε τις γυναίκες από το να βάζουν το όνομά τους σε εκθέσεις και αναφορές, παρότι εκείνες έκαναν την περισσότερη δουλειά
- Οι συνάδελφοί της στη NASA εμπιστεύονταν τους υπολογισμούς της, δηλαδή η Johnson ουσιαστικά διευκόλυνε την υιοθέτηση των ηλεκτρονικών υπολογιστών επαληθεύοντας τους υπολογισμούς τους
- Το 2015 ο πρόεδρος Μπαράκ Ομπάμα της απένειμε το Προεδρικό Μετάλλιο της Ελευθερίας (Presidential Medal of Freedom).

Προγραμματισμός – Η εντολή if

- Παράδειγμα μεταγλώττισης της εντολής if (6.15)

Κώδικας γλώσσας υψηλού επιπέδου
`if (apples = = oranges)`
`  f = g + h;`
`apples = oranges – h;`

Κώδικας συμβολικής γλώσσας της RISC-V
`# s0 = apples, s1 = oranges`
`# s2 = f, s3 = g, s4 = h`
`  BNE s0, s1, L1 # ελέγχει αν (s0 != s1)`
`  ADD s2, s3, s4 # f = g + h αν (s0 = s1)`
`L1: SUB s0, s1, s4 # apples = oranges – h`

Παρατηρείστε ότι ελέγχεται η αντίθετη συνθήκη **apples ≠ oranges**

Προγραμματισμός – Η εντολή if/else

- Παράδειγμα μεταγλώττισης της εντολής if/else (6.16)

Κώδικας γλώσσας υψηλού επιπέδου

```
if (apples == oranges)
    f = g + h;
else
    apples = oranges - h;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = apples, s1 = oranges
# s2 = f, s3 = g, s4 = h
  BNE s0, s1, L1 # ελέγχει αν (s0 != s1)
  ADD s2, s3, s4 # f = g + h αν (s0 == s1)
  J L2 # αν (s0 = s1)
L1: SUB s0, s1, s4 # apples = oranges - h αν (s0 != s1)
L2:
```

Παρατηρείστε ότι ελέγχεται η αντίθετη συνθήκη **apples ≠ oranges**

Προγραμματισμός – Η εντολή Switch/Case

- Παράδειγμα μεταγλώττισης της εντολής Switch/Case (6.17)

Κώδικας γλώσσας υψηλού επιπέδου

```
switch (button) {
  case 1: amt = 20; break;
  case 2: amt = 50; break;
  case 3: amt = 100; break;
  default: amt = 0;
}
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = button, s1 = amt
CASE1:
  ADDI t0, zero, 1 # t0 = 1
  BNE s0, t0, CASE2 # ελέγχεται αν (button != 1)
  ADDI s1, zero, 20 # αν (button == 1), amt = 20
  J DONE # έξοδος από την περίπτωση
CASE2:
  ADDI t0, zero, 2 # t0 = 2
  BNE s0, t0, case3 # ελέγχεται αν (button != 2)
  ADDI s1, zero, 50 # αν (button == 2), amt = 50
  J DONE # έξοδος από την περίπτωση
CASE3:
  ADDI t0, zero, 3 # t0 = 3
  BNE s0, t0, default  # ελέγχεται αν (button != 3)
  ADDI s1, zero, 100 # αν (button == 3), amt = 100
  J DONE # έξοδος από την περίπτωση
DEFAULT:
  ADD s1, zero, zero # amt=0
DONE:
```

Ελέγχεται η αντίθετη συνθήκη

Ελέγχεται η αντίθετη συνθήκη

Ελέγχεται η αντίθετη συνθήκη

Προγραμματισμός – Βρόχος while

- Παράδειγμα μεταγλώττισης του βρόχου while (6.18)
  - *Ο βρόχος while εκτελεί επανειλημμένα ένα τμήμα κώδικα μέχρι να **μην** ικανοποιείται μία συνθήκη*
  - *Στο παράδειγμα βρίσκει την τιμή του x για την οποία ισχύει $2^x = 128$*
    - Εκτελείται 7 φορές

Κώδικας γλώσσας υψηλού επιπέδου

```
int pow = 1;
int x = 0;
while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = pow, s1 = x
  ADDI s0, zero, 1 # pow = 1
  ADD s1, zero, zero # x = 0
  ADDI t0, zero, 128 # t0 = 128
WHILE: BEQ s0, t0, DONE # ελέγχεται αν pow == 128
  SLLI s0, s0, 1 # pow = pow * 2
  ADDI s1, s1, 1 # x = x + 1
  J WHILE # Επανάληψη βρόχου
DONE:
```

Παρατηρείστε ότι ελέγχεται η αντίθετη συνθήκη **`pow = 128`**

Προγραμματισμός – Βρόχος for

- Παράδειγμα μεταγλώττισης του βρόχου for (6.20)
  - *Ο βρόχος for συνδυάζει τον καθορισμό της αρχικής τιμής, τον έλεγχο της συνθήκης και την τροποποίηση της μεταβλητής σε ένα σημείο*
    - Χρησιμοποιείται όταν είναι γνωστό το πλήθος των επαναλήψεων του βρόχου
  - *Στο παράδειγμα προστίθενται οι αριθμοί από το 0 έως το 9*
    - ο βρόχος επαναλαμβάνεται 10 φορές (το i αυξάνεται κατά 1 στο τέλος του βρόχου, ενώ η συνθήκη (i < 10) ελέγχεται στην αρχή του βρόχου

Κώδικας γλώσσας υψηλού επιπέδου

```
int sum = 0;
int i;
for (i = 0; i < 10; i = i + 1) {
    sum = sum + i;
}
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = i, s1 = sum
  ADDI  s1, zero, 0 # sum = 0
  ADDI  s0, zero, 0 # i = 0
  ADDI  t0, zero, 10 # t0 = 10
FOR: BGE s0,  t0, DONE # αν i >= 10 τερματισμός βρόγχου
  ADD s1,  s1, s0 # sum = sum + i
  ADDI s0, s0, 1 # i = i + 1
  J FOR # Επανάληψη βρόχου
DONE:
```

Παρατηρείστε ότι ελέγχεται η αντίθετη συνθήκη `i ≥ 10`

Προγραμματισμός – Προσπέλαση πίνακα

- Για λόγους διευκόλυνσης της αποθήκευσης και της προσπέλασης, παρόμοια ομοειδή δεδομένα (στοιχεία) μπορούν να ομαδοποιούνται σε έναν **πίνακα** (*array*)
  - *Τα περιεχόμενα του πίνακα ονομάζονται στοιχεία και αποθηκεύονται σε **συνεχόμενες λέξεις δεδομένων των 32 bit (4 bytes)** στη μνήμη*
  - *Κάθε στοιχείο του πίνακα προσδιορίζεται από έναν αριθμό που είναι γνωστός ως ο **αριθμοδείκτης** (index) του πίνακα*
  - *Το πλήθος των στοιχείων του πίνακα συνιστά το **μήκος** (length) του*
  - *Κάθε διαδοχική διεύθυνση στοιχείου αυξάνεται κατά 4*
  - *Παράδειγμα του πίνακα* **scores[200]** *με 200 στοιχεία βαθμολογίας και διεύθυνση βάσης (base address) του πρώτου στοιχείου του πίνακα την 0x14000000*

Προγραμματισμός – Προσπέλαση πίνακα με βρόχος for

- Παράδειγμα μεταγλώττισης της προσπέλασης του πίνακα με βρόχο for (6.21)
  - *Στο παράδειγμα προστίθενται 10 μονάδες σε κάθε στοιχείο του πίνακα*
    - Ο βρόχος επαναλαμβάνεται 200 φορές (το i αυξάνεται κατά 1 στο τέλος του βρόχου, ενώ η συνθήκη (i < 200) ελέγχεται στην αρχή του βρόχου
    - Η διεύθυνση του πίνακα αυξάνεται κατά 4 γιατί κάθε στοιχείο του πίνακα έχει μέγεθος 4 byte και η μνήμη είναι προσπελάσιμη κατά byte
    - Χρησιμοποιείται ο τρόπος διευθυνσιοδότησης βάσης (η σχετική απόσταση (offset) είναι 0)

Κώδικας γλώσσας υψηλού επιπέδου

```
int i;
int scores[200];
...
for (i = 0; i < 200; i = i + 1)
    scores[i] = scores[i] + 10;
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s0 = διεύθυνση βάσης, s1 = i
ADDI s1, zero, 0 # i = 0
ADDI t2, zero, 200 # t2 = 200
FOR:
BGE s1, t2, DONE # if i >= 200 τερματισμός βρόχου
SLLI t0, s1, 2 # t0 = i * 4
ADD t0, t0, s0 # διεύθυνση του scores[i]
LW t1, 0(t0) # t1 = scores[i]
ADDI t1, t1, 10 # t1 = scores[i] + 10
SW t1, 0(t0) # scores[i] = t1
ADDI s1, s1, 1 # i = i + 1
J FOR # Επανάληψη βρόγχου
DONE:
```

Παρατηρείστε ότι ελέγχεται η αντίθετη συνθήκη **i ≥ 200**

Προγραμματισμός – Κλήσεις συναρτήσεων

- Οι γλώσσες προγραμματισμού υποστηρίζουν **συναρτήσεις** (*functions*), καθώς και **διαδικασίες** ή **υπορουτίνες** (*procedures* ή *subroutines*)
  - *με τις συναρτήσεις είναι εφικτή η επαναχρησιμοποίηση κώδικα και η ενίσχυση της αναγνωσιμότητας ενός προγράμματος*
- Οι συναρτήσεις έχουν εισόδους που ονομάζονται **ορίσματα** (*arguments*), και μια έξοδο που ονομάζεται **επιστρεφόμενη τιμή** (*return value*)
  - *οι διαδικασίες έχουν περισσότερες επιστρεφόμενες τιμές*
- Συμβάσεις για την κλήση συναρτήσεων
  - *Η **καλούσα συνάρτηση (caller)***
    - Περνά τα ορίσματα στην καλούμενη συνάρτηση
    - Συνεχίζει την εκτέλεση του προγράμματος στην καλούμενη συνάρτηση με εντολή άλματος και σύνδεσης (για δυνατότητα επιστροφής)
  - *Η **καλούμενη συνάρτηση (callee)***
    - Εκτελεί τη συνάρτηση
    - Επιστρέφει το αποτέλεσμα στην καλούσα συνάρτηση
    - Επιστρέφει την εκτέλεση του προγράμματος στο σημείο κλήσης της
    - Προστατεύει καταχωρητές και μνήμη που χρησιμοποιεί η καλούσα συνάρτηση

Προγραμματισμός – Κλήσεις συναρτήσεων στη RISC-V

- Στην αρχιτεκτονική RISC-V ισχύουν οι εξής συμβάσεις:
  - *η καλούσα συνάρτηση (caller) τοποθετεί έως και **οκτώ ορίσματα** στους καταχωρητές **a0-a7** πριν την κλήση της συνάρτησης*
  - *η καλούμενη συνάρτηση (callee) τοποθετεί την επιστρεφόμενη τιμή στον καταχωρητή **a0** πριν ολοκληρωθεί η εκτέλεσή της*
    - για επιστρεφόμενη ακεραία τιμή των 64 bit χρησιμοποιούνται οι a1-a0
  - *η καλούσα συνάρτηση αποθηκεύει τη διεύθυνση επιστροφής (PC+4) στον καταχωρητή διεύθυνσης επιστροφής (return address, ra)*
    - με την κλήση της καλούμενης συνάρτησης με μία εντολή άλματος και σύνδεσης (JAL, JALR) της RISC-V
  - *η καλούμενη συνάρτηση πρέπει να προστατεύσει τους αποθηκευμένους καταχωρητές **s0-s11** και το **ra** (όταν η ίδια καλεί μία άλλη συνάρτηση), καθώς και την **στοίβα (stack)***
    - η **στοίβα** χρησιμοποιείται για προσωρινή αποθήκευση τιμών και μεταβλητών
  - *Με την ολοκλήρωση της καλούμενης συνάρτησης η εκτέλεση του προγράμματος συνεχίζεται από την καλούσα συνάρτηση αμέσως μετά την εντολή κλήσης της καλούμενης συνάρτησης*
    - με την αντιγραφή του return address στον μετρητή PC με την εντολή άλματος JR της RISC-V (PC = ra)

Συμβολική γλώσσα – Εντολή άλματος και σύνδεσης JAL

- Η αρχιτεκτονική RISC-V υποστηρίζει εντολές άλματος (jump) και σύνδεσης (link) (τύπου U/J) για την κλήση συναρτήσεων (και διαδικασιών)
- Μορφή εντολής: JAL destination_reg Rd, label
  - Για να επιτευχθεί η σύνδεση, σε έναν καταχωρητή προορισμού αποθηκεύεται η διεύθυνση της αμέσου επόμενης εντολής μετά την JAL, δηλαδή το PC+4
    - Rd = PC+4
    - ο καταχωρητής return address (ra) είναι default και δεν απαιτεί δήλωση
      `JAL label => JAL ra, label (Rd = ra)`
  - Οι ετικέτες (label) υποδεικνύουν τη διεύθυνση της πρώτης εντολής της συνάρτησης (διαδικασίας) που θα συνεχιστεί η εκτέλεση του προγράμματος (διεύθυνση στόχος άλματος (jump target address, JTA)
    - Οι ετικέτες (label) ακολουθούνται από άνω και κάτω τελεία (:)
  - Η διεύθυνση JTA δηλώνει απόσταση μεταξύ εντολών (όχι bytes) και μετατρέπεται σε άμεσο τελεστέο μεγέθους 20 bit (immediate20) κατά τη συμβολομετάφραση σε γλώσσα μηχανής
  - *Κατά την εκτέλεση της εντολής υπολογίζεται σε ξεχωριστό αθροιστή η JTA:*

    PC = JTA = PC + sign_ext(immediate20 x 2)

Η εντολή J (jump) είναι ψευδοεντολή της εντολής JAL (jump and link)
`J label => JAL zero, label`

Συμβολική γλώσσα – Εντολή άλματος και σύνδεσης JALR

- Η αρχιτεκτονική RISC-V υποστηρίζει εντολές άλματος (jump) και σύνδεσης (link) (τύπου I) για την κλήση συναρτήσεων (και διαδικασιών)
- Μορφή εντολής: **JALR destination_reg Rd, source_reg Rs1, immediate12**
  - Για να επιτευχθεί η σύνδεση, σε έναν καταχωρητή προορισμού αποθηκεύεται η διεύθυνση της αμέσου επόμενης εντολής μετά την JAL, δηλαδή το PC+4
    - Rd = PC+4
    - χρησιμοποιείται συνήθως ο καταχωρητής **return address (ra)**
  - Ο καταχωρητής προέλευσης και ο προσημασμένος άμεσος τελεστέος μεγέθους 12 bit (immediate12) υποδεικνύουν τη διεύθυνση της πρώτης εντολής της συνάρτησης (διαδικασίας) που θα συνεχιστεί η εκτέλεση του προγράμματος (διεύθυνση στόχος άλματος (jump target address, JTA)
  - *Για παράδειγμα, η εντολή* ***jalr ra, s1, 0x4C*** *εκτελεί άλμα προς τη διεύθυνση s1 + 0x4C και τοποθετεί την τιμή PC+4 στον καταχωρητή ra*
  - *Κατά την εκτέλεση της εντολής υπολογίζεται στη μονάδα ALU η JTA:*

    PC = JTA = source_reg + Sing_Ext(immediate12)

Η εντολή JR (jump register) είναι ψευδοεντολή της εντολής JALR (jump and link register)
`JR ra => JALR zero, ra, 0`

Προγραμματισμός – Κλήσεις συναρτήσεων

- Παράδειγμα (6.22) απλής κλήσης και επιστροφής από συνάρτηση χωρίς ανταλλαγή τιμών (η main καλεί τη συνάρτηση simple)
  - *χρησιμοποιούνται οι εντολές JAL (jump and link) και JR (jump register)*
  - *η JAL στη διεύθυνση PC εκτελεί δύο μεταφορές δεδομένων:*
    - ra = PC+4 = 0x00000304
    - PC = PC + sign_ext(immediate20 x 2)
      = 0x300 + 0x21C* = 0x51C
  - *Η JR επιστρέφει από τη συνάρτηση*
    - PC = ra = 0x00000304

Κώδικας γλώσσας υψηλού επιπέδου

```
int main() {
    ...
    simple();
    ...
}
void simple() {
    return;
}
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
0x00000300 MAIN: JAL SIMPLE # κλήση συνάρτησης simple
0x00000304    ...
...
0x0000051C SIMPLE: JR ra    # επιστροφή
```

*Immediate20 = 0x10E

void σημαίνει ότι δεν υπάρχει επιστροφή τιμής

Προγραμματισμός – Κλήσεις συναρτήσεων

- Παράδειγμα (6.22) απλής κλήσης και επιστροφής από συνάρτηση χωρίς ανταλλαγή τιμών (η main καλεί τη συνάρτηση simple)
  - *χρησιμοποιούνται οι εντολές JALR (jump and link register) και JR (jump register)*
  - *η JALR στη διεύθυνση PC εκτελεί δύο μεταφορές δεδομένων:*
    - ra = PC+4 = 0x00000304
    - PC = 0x00000000 + 0x0000051C
      = 0x0000051C
  - *Η JR επιστρέφει από τη συνάρτηση*
    - PC = ra = 0x00000304

Κώδικας γλώσσας υψηλού επιπέδου

```
int main() {
    ...
    simple();
    ...
}
void simple() {
    return;
}
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
0x00000300 MAIN: JALR ra, zero, 0x51C # κλήση συνάρτησης simple
0x00000304    ...
...
0x0000051C SIMPLE: JR ra              # επιστροφή
```

void σημαίνει ότι δεν υπάρχει επιστροφή τιμής

Προγραμματισμός – Κλήσεις συναρτήσεων

- Παράδειγμα (6.23) κλήσης και επιστροφής από συνάρτηση με ορίσματα και επιστρεφόμενη τιμή (χωρίς χρήση στοίβας) (η main καλεί τη συνάρτηση dif)
  - *χρησιμοποιούνται οι εντολές JAL και JR*

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s7 = y μεταβλητή της MAIN
MAIN:
   ...
   ADDI  a0, zero, 2 # όρισμα 0 = 2
   ADDI  a1, zero, 3 # όρισμα 1 = 3
   ADDI  a2, zero, 4 # όρισμα 2 = 4
   ADDI  a3, zero, 5 # όρισμα 3 = 5
   JAL DIF           # κλήση dif
   ADD s7, a0, zero  # y = επιστρ/μενη τιμή
   ...
# s3 = result μεταβλητή της DIF
DIF:
   ADD s0, a0, a1    # t0 = f+g
   ADD s1, a2, a3    # t1 = h+i
   SUB s2, t0, t1    # result = s0 - s1
   ADD a0, s2, zero  # a0 = επιστρ/μενη τιμή
   JR ra             # επιστροφή main
```

Κώδικας γλώσσας υψηλού επιπέδου

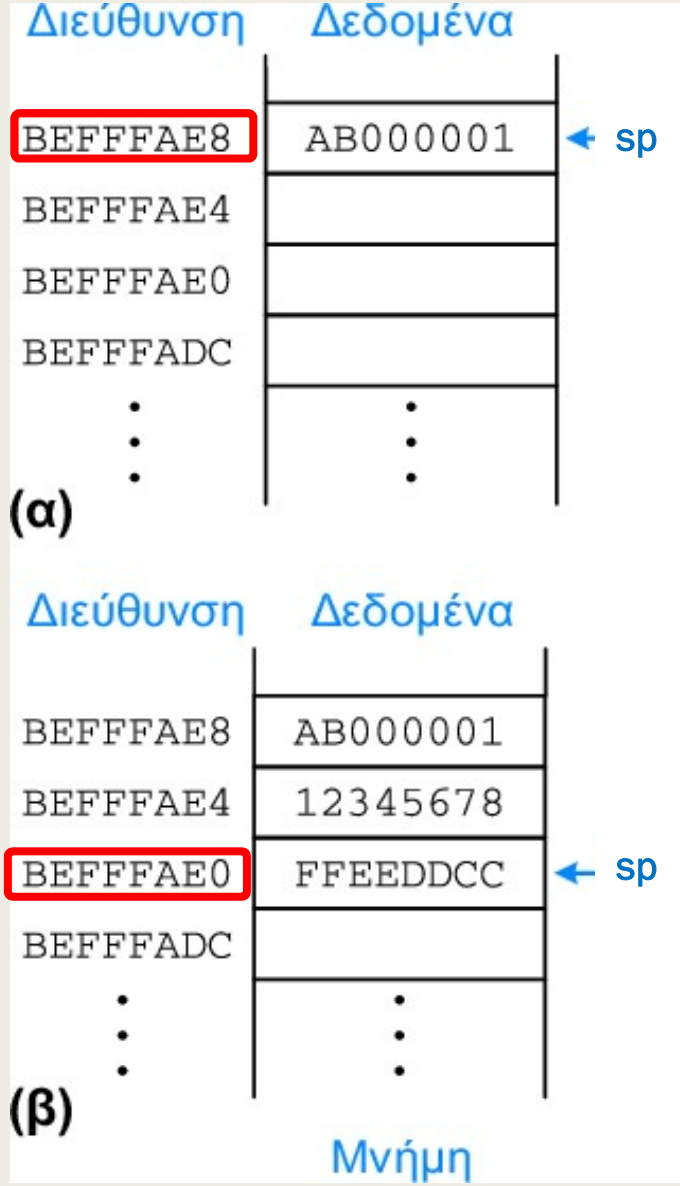
```
int main() {
    int y;
    ...
    y = dif(2, 3, 4, 5);
    ...
}

int dif(int f, int g, int h, int i) {
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

Ο καταχωρητές s0, s1, s2 δεν προστατεύονται!

Η στοίβα (stack)

- Η στοίβα είναι ένα τμήμα μνήμης που χρησιμοποιείται για την προσωρινή αποθήκευση καταχωρητών που πρέπει να προστατευθούν κατά την κλήση και εκτέλεση μιας συνάρτησης
- Η στοίβα είναι μια ουρά LIFO (last-in-first-out):
  - *Το τελευταίο στοιχείο που τοποθετείται (push) πάνω στη στοίβα είναι και το πρώτο που μπορεί να ανακτηθεί (pop).*
- Η *κορυφή της στοίβας* αντιστοιχεί στην πιο πρόσφατα αποθηκευμένη λέξη δεδομένων
- Η αρχιτεκτονική RISC-V χρησιμοποιεί τον sp (stack pointer) του αρχείου καταχωρητών ως δείκτη στοίβας που δείχνει στην κορυφή της στοίβας
  - *Ο sp ξεκινάει σε μια υψηλή διεύθυνση μνήμης και μειώνεται κατά 4 κάθε φορά που νέα δεδομένα μπαίνουν στη στοίβα*
    - Η στοίβα (α) πριν από την επέκταση και (β) μετά από επέκταση δύο λέξεων



Προγραμματισμός – Κλήσεις συναρτήσεων

- Η στοίβα είναι ένα τμήμα μνήμης που χρησιμοποιείται για την προσωρινή αποθήκευση καταχωρητών που πρέπει να προστατευθούν κατά την κλήση και εκτέλεση μιας συνάρτησης
  - *Οι συναρτήσεις αποθηκεύουν τα περιεχόμενα των καταχωρητών στη στοίβα προτού τα τροποποιήσουν, και μετά τα ανακτούν από τη στοίβα πριν την επιστροφή*
  - *Οι συναρτήσεις εκτελούν τα ακόλουθα βήματα:*
    1. Δέσμευση κατάλληλου χώρου στη στοίβα που ονομάζεται πλαίσιο στοίβας μειώνοντας κατάλληλα τον **sp**
    2. Αποθήκευση των τιμών των καταχωρητών στη στοίβα
    3. Χρήση των καταχωρητών εντός της συνάρτησης
    4. Επαναφορά των αρχικών τιμών των καταχωρητών από τη στοίβα
    5. Αποδέσμευση χώρου στη στοίβα αυξάνοντας τον **sp**
  - *Η αρχή της τμηματικότητας (modularity) ορίζει ότι κάθε συνάρτηση πρέπει να προσπελάζει μόνο το δικό της πλαίσιο στοίβας, και όχι τα πλαίσια που ανήκουν σε άλλες συναρτήσεις*

Προγραμματισμός – Κλήσεις συναρτήσεων

- Παράδειγμα (6.24) κλήσης και επιστροφής από συνάρτηση με ορίσματα και επιστρεφόμενη τιμή (με χρήση στοίβας) (η main καλεί τη συνάρτηση dif)

Κώδικας συμβολικής γλώσσας της RISC-V

```
# s3 = result μεταβλητή της DIF
DIF:
#  Δέσμευση πλαισίου στοίβας για 3 καταχωρητές
   ADDI sp, sp, -12
#  Αποθήκευση των καταχωρητών s0, s1, s2
#  στο πλαίσιο στοίβας
   SW s0, 8(sp)
   SW s1, 4(sp)
   SW s2, 0(sp)
   ADD s0, a0, a1    # t0 = f+g
   ADD s1, a2, a3    # t1 = h+i
   SUB s2, t0, t1    # result = s0 - s1
   ADD a0, s2, zero  # a0 = επιστρ/μενη τιμή
#  Ανάκτηση των καταχωρητών s0, s1, s2
#  από το πλαίσιο στοίβας
   LW s0, 8(sp)
   LW s1, 4(sp)
   LW s2, 0(sp)
#  Αποδέσμευση πλαισίου στοίβας για 3 καταχωρητές
   ADDI sp, sp, 12
   JR ra             # επιστροφή main
```

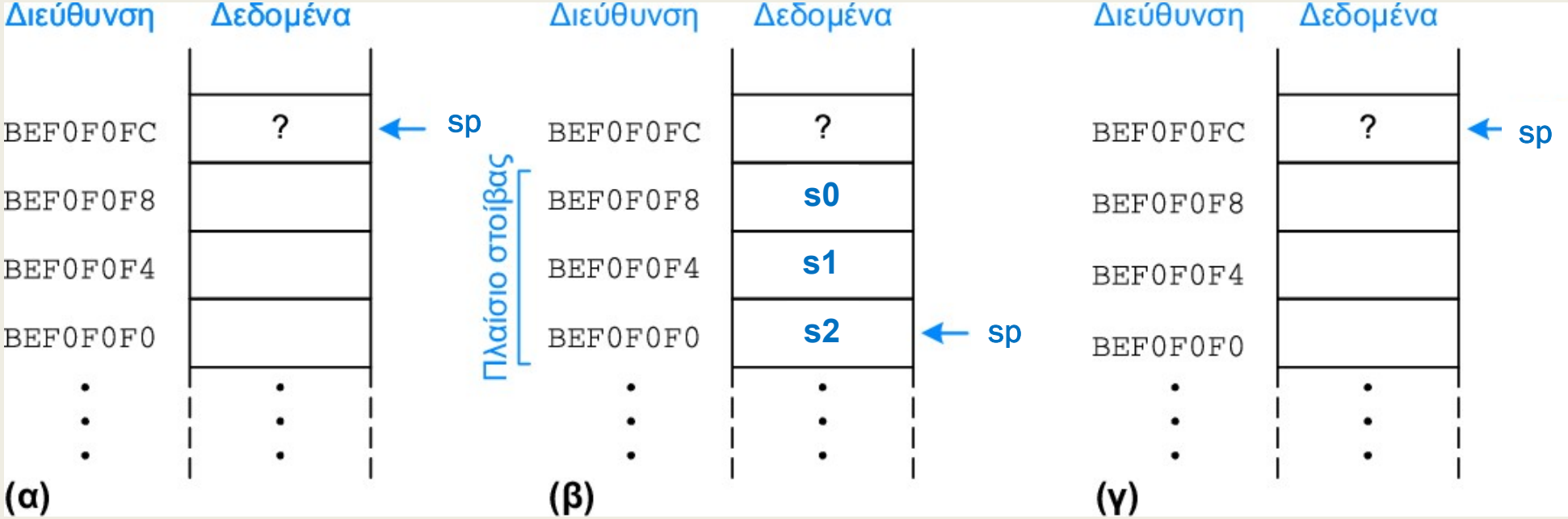
Κώδικας γλώσσας υψηλού επιπέδου

```
int dif(int f, int g, int h, int i) {
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

Ο καταχωρητές s0, s1, s2 τώρα προστατεύονται!

Προγραμματισμός – Κλήσεις συναρτήσεων

- Η στοίβα: (α) πριν από, (β) κατά τη διάρκεια και (γ) μετά από την κλήση της συνάρτησης dif



Εάν μία καλούμενη συνάρτηση καλεί μία άλλη συνάρτηση, τότε και ο καταχωρητής επιστροφής διεύθυνσης ra (return address) πρέπει να αποθηκευτεί στη στοίβα

Προγραμματισμός – Κλήσεις συναρτήσεων

- Η αρχιτεκτονική RISC-V χωρίζει τους καταχωρητές του αρχείου καταχωρητών σε δύο κατηγορίες, ώστε να αποφεύγονται άσκοπες αποθηκεύσεις στη στοίβα
  - *οι **διατηρούμενοι καταχωρητές** αποθηκεύονται από την **καλούμενη συνάρτηση**, όταν τους χρησιμοποιεί*
  - *οι **μη διατηρούμενοι καταχωρητές** αποθηκεύονται από την **καλούσα συνάρτηση**, όταν τους χρησιμοποιεί*

Διατηρούμενοι	Μη Διατηρούμενοι
Αποθηκευόμενοι καταχωρητές: s0–s11	Προσωρινοί καταχωρητής: t0–t6
Δείκτης στοίβας: sp	Καταχωρητές ορισμάτων: a0–a7
Διεύθυνση επιστροφής: ra	
Στοίβα πάνω από τον δείκτη στοίβας	Στοίβα κάτω από τον δείκτη στοίβας

Προγραμματισμός – Κλήσεις συναρτήσεων

- Παράδειγμα (6.26) κλήσης και επιστροφής από συνάρτηση με ορίσματα και επιστρεφόμενη τιμή (με χρήση στοίβας) (η main καλεί τη συνάρτηση dif)
  - *Μείωση της χρήσης διατηρούμενων καταχωρητών από την dif*
    - Χρησιμοποιούνται αποκλειστικά μη διατηρούμενοι καταχωρητές
  - *Δεν απαιτείται πλέον η χρήση της στοίβας*

Κώδικας γλώσσας υψηλού επιπέδου

```
int dif(int f, int g, int h, int i) {
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

Κώδικας συμβολικής γλώσσας της RISC-V

```
# a0 = result μεταβλητή της DIF
DIF:
  ADD t0, a0, a1    # t0 = f+g
  ADD t1, a2, a3    # t1 = h+i
  SUB a0, t0, t1    # result = t0 - t1
  JR ra             # επιστροφή main
```

Προγραμματισμός – Ψευδοεντολές στη RISC-V

Pseudoinstruction	RISC-V Instructions	Description	Operation
`j label`	`jal zero, label`	jump	`PC = label`
`jr ra`	`jalr zero, ra, 0`	jump register	`PC = ra`
`mv t5, s3`	`addi t5, s3, 0`	move	`t5 = t3`
`not s7, t2`	`xori s7, t2, -1`	one's complement	`s7 = ~t2`
`nop`	`addi zero, zero, 0`	no operation	
`li s8, 0x7EF`	`addi s8, zero, 0x7EF`	load 12-bit immediate	`s8 = 0x7EF`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`	load 32-bit immediate	`s8 = 0x56789DEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`	branch if >	`if (s1 > t3), PC = L3`
`bgez t2, L7`	`bge t2, zero, L7`	branch if ≥ 0	`if (t2 ≥ 0), PC = L7`
`call L1`	`jal L1`	call nearby function	`PC = L1, ra = PC + 4`
`call L5`	`auipc ra, imm31:12` `jalr ra, ra, imm11:0`	call far away function	`PC = L5, ra = PC + 4`
`ret`	`jalr zero, ra, 0`	return from function	`PC = ra`

Η εντολή **AUIPC** (add upper immediate to PC) προσθέτει έναν αριστερό άμεσο τελεστέο μεγέθους 20 bit με 0 στα υπόλοιπα δεξιά bit στον μετρητή PC και και τοποθετεί το αποτέλεσμα στον καταχωρητή προορισμού.

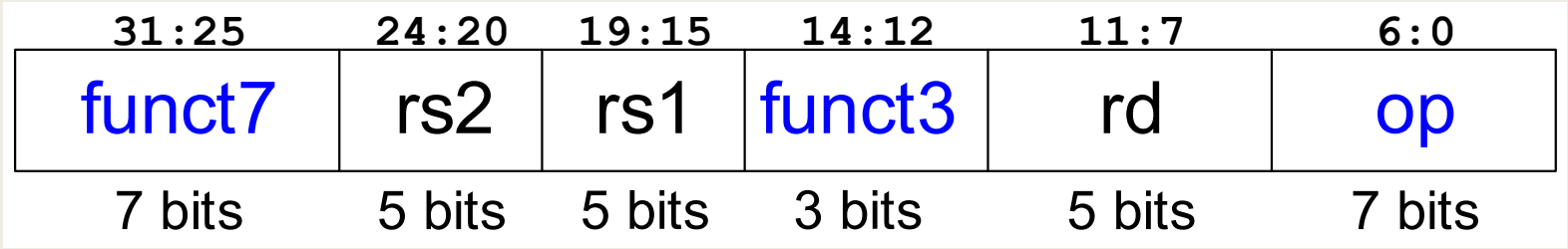
Η ψευδοεντολή **NOP** (no operation, μη πράξη), χρησιμοποιείται για την απαραίτητη καθυστέρηση, όταν υποστηρίζεται διοχέτευση. Ο μετρητής PC αυξάνεται κατά 4 αλλά δεν τροποποιούνται άλλοι καταχωρητές ή τιμές που είναι αποθηκευμένες στη μνήμη.

Γλώσσα μηχανής

- Η συμβολική γλώσσα είναι πιο ευανάγνωστη για τους ανθρώπους
  Όμως τα ψηφιακά κυκλώματα κατανοούν μόνο τις τιμές 0 και 1
- Ο συμβολομεταφραστής (assembler) μεταφράζει τις εντολές στη συμβολική γλώσσα στη δυαδική έκδοσή τους (σε γλώσσα μηχανής)
- Η αρχιτεκτονική RISC-V ορίζει εντολές σε γλώσσα μηχανής σταθερής κωδικοποίησης των 32 bit με τέσσερεις κύριες μορφές εντολών:
  - *Τύπου R (register)*,
  - *Τύπου I (immediate)*
  - *Τύπου S/B (store/branch)*
  - *Τύπου U/J (upper immediate/jump)*

Γλώσσα μηχανής – Εντολές τύπου R

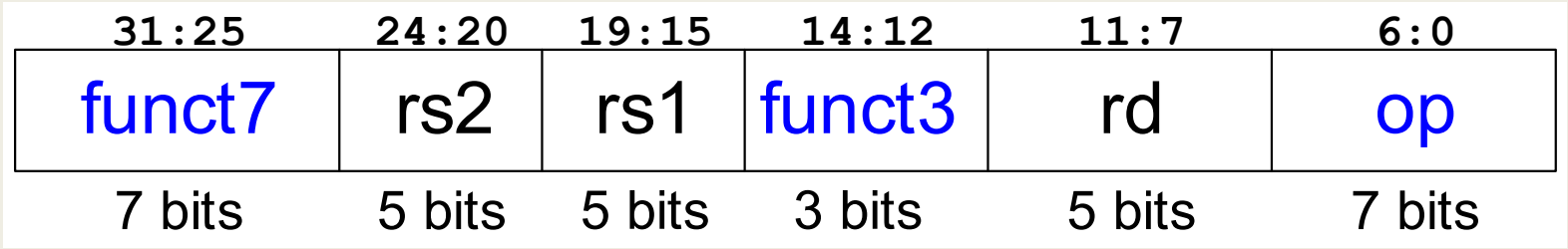
- Γενική μορφή εντολής:



- Πεδία εντολής:
  - ***Rs1*** *(source register 1): καταχωρητής προέλευσης 1 (19:15)*
  - ***Rs2*** *(source register 2): καταχωρητής προέλευσης 2 (24:20)*
  - ***Rd*** *(destination register): καταχωρητής προορισμού (11:7)*
  - ***op*** *(operation code, opcode): τύπος εντολής + είδος πράξης (6:0)*
  - ***funct3*** *(function 3 bit): κωδικός λειτουργίας (14:12)*
  - ***funct7*** *(function 7 bit): κωδικός λειτουργίας (31:25)*

Γλώσσα μηχανής – Εντολές τύπου R

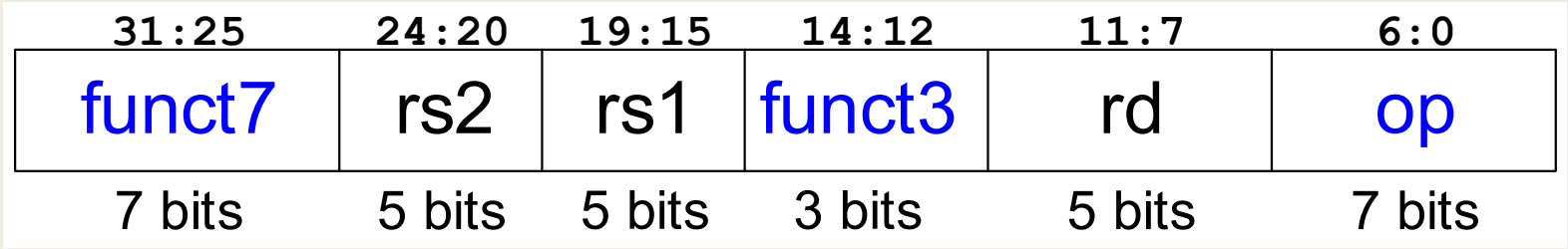
- Γενική μορφή εντολής:



- Κώδικας συμβολικής γλώσσας: πράξη:
  - `ADD  Rd, Rs1, Rs2` `Rd = Rs1 + Rs2`
  - `SUB  Rd, Rs1, Rs2` `Rd = Rs1 – Rs2`
  - `AND  Rd, Rs1, Rs2` `Rd = Rs1 and Rs2`
  - `OR  Rd, Rs1, Rs2` `Rd = Rs1 or  Rs2`
  - `XOR  Rd, Rs1, Rs2` `Rd = Rs1 xor Rs2`
  - `SLL  Rd, Rs1, Rs2` `Rd = Rs1 <<  Rs2(4:0)`
  - `SRL  Rd, Rs1, Rs2` `Rd = Rs1 >>  Rs2(4:0)`
  - `SRA  Rd, Rs1, Rs2` `Rd = Rs1 >>> Rs2(4:0)`
  - `SLT  Rd, Rs1, Rs2` `Rd = 1 if rs1<rs2, else 0`
  - `SLTU Rd, Rs1, Rs2` `Rd = 1 if rs1<rs2, else 0`

Γλώσσα μηχανής – Εντολές τύπου R

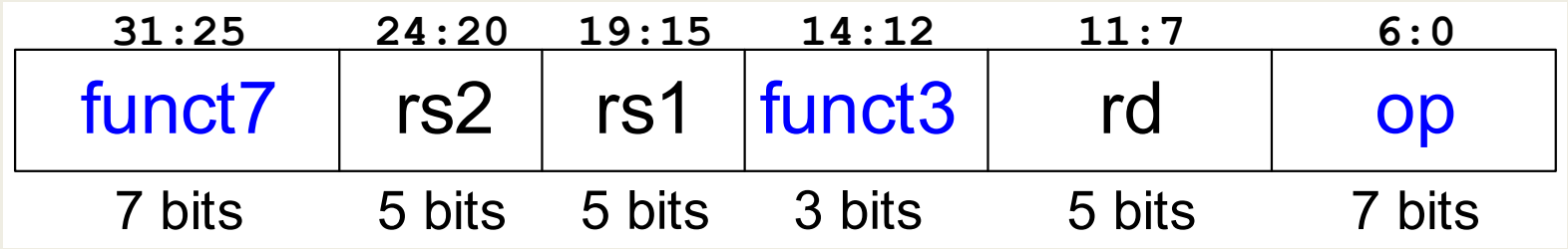
- Γενική μορφή εντολής:



- Πεδία ελέγχου (op, funct3, funct7):
  - **ADD** 0110011 000 0000000
  - **SUB** 0110011 000 0100000
  - **AND** 0110011 111 0000000
  - **OR** 0110011 110 0000000
  - **XOR** 0110011 100 0000000
  - **SLL** 0110011 001 0000000
  - **SRL** 0110011 101 0000000
  - **SRA** 0110011 101 0100000
  - **SLT** 0110011 010 0000000
  - **SLTU** 0110011 011 0000000

Γλώσσα μηχανής – Εντολές τύπου R

- Γενική μορφή εντολής:



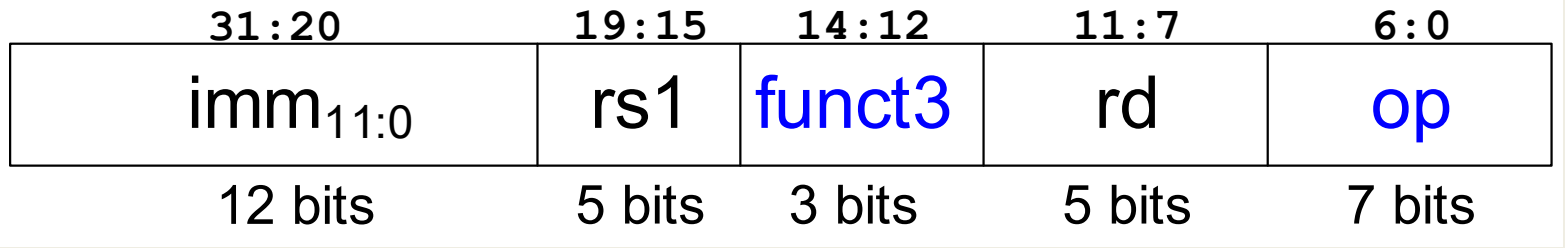
- Παραδείγματα:

Assembly **Machine Code**

	funct7	rs2	rs1	funct3	rd	op	
`add s2, s3, s4` `add x18,x19,x20`	0000 000	10100	10011	000	10010	011 0011	(0x01498933)
`sub t0, t1, t2` `sub x5, x6, x7`	0100 000	00111	00110	000	00101	011 0011	(0x407302B3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Γλώσσα μηχανής – Εντολές τύπου I

- Γενική μορφή εντολής:

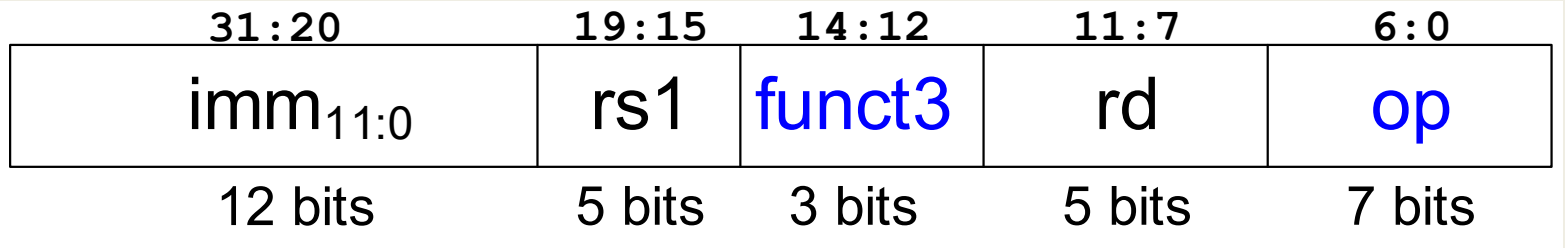


- Πεδία εντολής:
  - ***Rs1*** *(source register 1): καταχωρητής προέλευσης 1 (19:15)*
  - ***Rd*** *(destination register): καταχωρητής προορισμού (11:7)*
  - ***imm*** *(immediate): προσημασμένος άμεσος τελεστέος των 12 bit σε απεικόνιση συμπληρώματος ως προς 2 (11:0)*
    - imm(11:5) αντί funct7 (31:25) και imm(4:0) αντί Rs2 (24:20)
    - Στις ολισθήσεις η ποσότητα ολίσθησης καθορίζεται ως μη προσημασμένος άμεσος τελεστέος των 5 bit (uimm) αντί του Rs2 (24:20).
      Τα υπόλοιπα ψηφία του πεδίου imm χρησιμοποιούνται ως **funct7!**
  - ***op*** *(operation code, opcode): τύπος εντολής + είδος πράξης (6:0)*
  - ***funct3*** *(function 3 bit): κωδικός λειτουργίας (14:12)*

Στο πεδίο imm εφαρμόζεται επέκταση προσήμου από τα 12 bit στα 32 bit σε όλες τις εντολές εκτός από τις εντολές ολίσθησης

Γλώσσα μηχανής – Εντολές τύπου I

- Γενική μορφή εντολής:

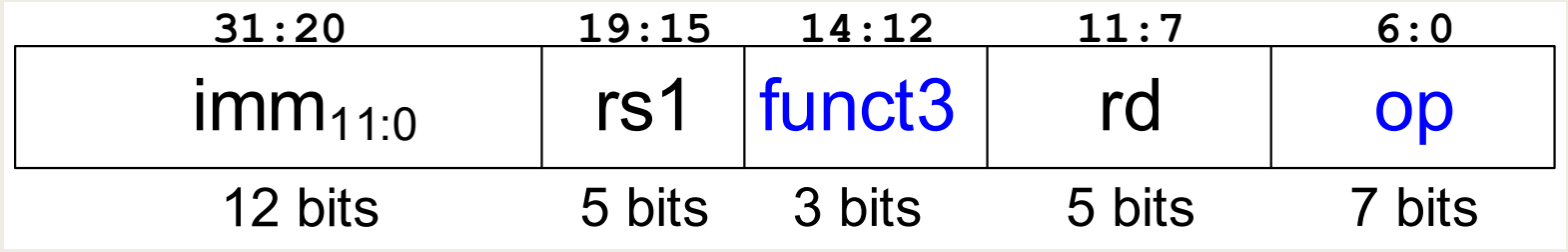


- Κώδικας συμβολικής γλώσσας: πράξη:
  - **ADDI Rd, Rs1, imm** **Rd = Rs1 + S_Ext(imm)**
  - **JALR Rd, Rs1, imm** **PC = Rs1 + S_Ext(imm)**
  - **Rd = PC + 4**
  - **ANDI Rd, Rs1, imm** **Rd = Rs1 and S_Ext(imm)**
  - **ORI Rd, Rs1, imm** **Rd = Rs1 or S_Ext(imm)**
  - **XORI Rd, Rs1, imm** **Rd = Rs1 xor S_Ext(imm)**
  - **SLLI Rd, Rs1, uimm** **Rd = Rs1 << imm(4:0)**
  - **SRLI Rd, Rs1, uimm** **Rd = Rs1 >> imm(4:0)**
  - **SRAI Rd, Rs1, uimm** **Rd = Rs1 >>> imm(4:0)**
  - **LW Rd, imm(Rs1)** **Rd = mem[Rs1+S_Ext(imm)]**

Η αρχιτεκτονική RISC-V υποστηρίζει και άλλες εντολλές φόρτωσης όπως LB, LBU (byte), LH, LHU (half word), καθώς και εντολές SLTI και SLTIU

Γλώσσα μηχανής – Εντολές τύπου I

- Γενική μορφή εντολής:



- Πεδία ελέγχου (op, funct3, funct7 (μόνο στις ολισθήσεις):
  - **ADDI** 0010011 000
  - **JALR** 1100111 000
  - **ANDI** 0010011 111
  - **ORI** 0010011 110
  - **XORI** 0010011 100
  - **SLLI** 0010011 001 0000000
  - **SRLI** 0010011 101 0000000
  - **SRAI** 0010011 101 0100000
  - **LW** 0000011 010
  - **LH** 0000011 001
  - **LB** 0000011 000

Γλώσσα μηχανής – Εντολές τύπου I

- Γενική μορφή εντολής:

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

- Παραδείγματα:

Assembly	$imm_{11:0}$	rs1	funct3	rd	op	Machine Code
addi s0, s1, 12 addi x8, x9, 12	0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)
addi s2, t1, -14 addi x18,x6, -14	1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)
lw t2, -6(s3) lw x7, -6(x19)	1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)
lb s4, 0x1F(s4) lb x20,0x1F(x20)	0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)
slli s2, s7, 5 slli x18, x23, 5	0000 0000 0101	10111	001	10010	001 0011	(0x005B9913)
srai t1, t2, 29 srai x6, x7, 29	0100 0001 1101	00111	101	00110	001 0011	(0x41D3D313)
	12 bits	5 bits	3 bits	5 bits	7 bits	

Γλώσσα μηχανής – Εντολές τύπου S/B

- Γενική μορφή εντολής:

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

- Πεδία εντολής για εντολές τύπου S/B:
  - ***Rs1*** *(source register 1): καταχωρητής προέλευσης 1 (19:15)*
  - ***Rs2*** *(source register 2): καταχωρητής προέλευσης 2 (24:20)*
  - ***imm*** *(immediate) τύπου S (store): προσημασμένος άμεσος τελεστέος των 12 bit σε απεικόνιση συμπληρώματος ως προς 2 (11:0)*
    - imm(11:5) αντί funct7(31:25) και imm(4:0) αντί Rd (11:7)
  - ***imm*** *(immediate) τύπου B (store): προσημασμένος άμεσος τελεστέος των 13 bit σε απεικόνιση συμπληρώματος ως προς 2 (12:0)*
    - imm(12,10:5) αντί funct7(31:25) και imm(4:1,11) αντί Rd(11:7), **imm(0)=0**
  - ***op*** *(operation code, opcode): τύπος εντολής + είδος πράξης (6:0)*
  - ***funct3*** *(function 3 bit): κωδικός λειτουργίας (14:12)*

Στο πεδίο imm εφαρμόζεται επέκταση προσήμου από τα 12/13 bit στα 32 bit

Γλώσσα μηχανής – Εντολές τύπου S/B

- Γενική μορφή εντολής:

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

- Κώδικας συμβολικής γλώσσας: πράξη:
  - **SW Rs2, imm(Rs1)** **mem[Rs1+S_Ext(imm)] = Rs2**
  - **BEQ Rs1, Rs2, label branch BTA if Rs1 = Rs2**
  - **BNE Rs1, Rs2, label branch BTA if Rs1 ≠ Rs2**
  - **BLT Rs1, Rs2, label branch BTA if Rs1 < Rs2**
  - **BGE Rs1, Rs2, label branch BTA if Rs1 ≥ Rs2**
  - **BLTU Rs1, Rs2, label branch BTA if Rs1 < Rs2**
  - **BGEU Rs1, Rs2, label branch BTA if Rs1 ≥ Rs2**

BTA = PC + S_Ext (imm(12:1)x2)

Η αρχιτεκτονική RISC-V υποστηρίζει και άλλες εντολές φόρτωσης όπως SB (byte), SH (half word). Για την εντολή Branch ορίζονται και ψευδοεντολές

Γλώσσα μηχανής – Εντολές τύπου S/B

- Γενική μορφή εντολής:

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

- Πεδία ελέγχου (op, funct3):
  - **SW** **0100011** **010**
  - **SH** **0100011** **001**
  - **SB** **0100011** **000**
  - **BEQ** **1100011** **000**
  - **BNE** **1100011** **001**
  - **BLT** **1100011** **100**
  - **BGE** **1100011** **101**
  - **BLTU** **1100011** **110**
  - **BGEU** **1100011** **111**

Γλώσσα μηχανής – Εντολές τύπου S/B

- Γενική μορφή εντολής:

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

- Παραδείγματα εντολών τύπου S (store):

Assembly	Machine Code						
	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
`sw t2, -6(s3)` `sw x7, -6(x19)`	1111 111	00111	10011	010	11010	010 0011	(0xFE79AD23)
`sh s4, 23(t0)` `sh x20,23(x5)`	0000 000	10100	00101	001	10111	010 0011	(0x01429BA3)
`sb t5, 0x2D(zero)` `sb x30,0x2D(x0)`	0000 001	11110	00000	000	01101	010 0011	(0x03E006A3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Γλώσσα μηχανής – Εντολές τύπου S/B

- Γενική μορφή εντολής:

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

- Παραδείγματα εντολών τύπου B (branch):

```
# RISC-V Assembly
0x70      beq  s0, t5, L1     1
0x74      add  s1, s2, s3     2
0x78      sub  s5, s6, s7     3
0x7C      lw   t0, 0(s1)      4
0x80 L1:  addi s1, s1, -15
```

L1 is 4 instructions (i.e., **16 bytes**) past `beq`

$imm_{12:0}$ = 16	0		0	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12		11	10	9	8	7	6	5		4	3	2	1	0

Assembly

```
beq s0, t5, L1
beq x8, x30, 16
```

Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
0000 000	11110	01000	000	1000 0	110 0011	(0x01E40863)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Γλώσσα μηχανής – Εντολές τύπου U/J

- Γενική μορφή εντολής:

- Πεδία εντολής για εντολές τύπου U/J:
  - ***Rd*** *(destination register): καταχωρητής προορισμού (11:7)*
  - ***imm*** *(immediate) τύπου U (upper immediate): προσημασμένος άμεσος τελεστέος των 20 bit σε απεικόνιση συμπληρώματος ως προς 2 (31:12)*
    - imm(31:25) αντί funct7(31:25), imm(24:20) αντί Rs2 (24:20), imm(19:15) αντί Rs1(19:15) και imm(14:12) αντί funct3 (14:12),
  - ***imm*** *(immediate) τύπου J (jump): προσημασμένος άμεσος τελεστέος των 21 bit σε απεικόνιση συμπληρώματος ως προς 2 (20:0)*
    - imm(20,10:5) αντί funct7(31:25), imm(4:1,11) αντί Rs2(24:20), imm(19:15) αντί Rs1(19:15) και imm(14:12) αντί funct3 (14:12), imm(0)=0
  - ***op*** *(operation code, opcode): τύπος εντολής + είδος πράξης (6:0)*
  - ***funct3*** *(function 3 bit): κωδικός λειτουργίας (14:12)*

Στο πεδίο imm εφαρμόζεται επέκταση προσήμου από τα 20/21 bit στα 32 bit

Γλώσσα μηχανής – Εντολές τύπου U/J

- Γενική μορφή εντολής:

- Κώδικας συμβολικής γλώσσας: πράξη:
  - **LUI Rd, upimm** **Rd = (upimm,0x000)**
  - **AUIPC Rd, upimm** **Rd = PC + (upimm,0x000)**
  - **JAL Rd, label** **jump JTA**
  - **Rd = PC + 4**

```
JTA = PC + S_Ext (imm(20:1)x2)
```

Γλώσσα μηχανής – Εντολές τύπου U/J

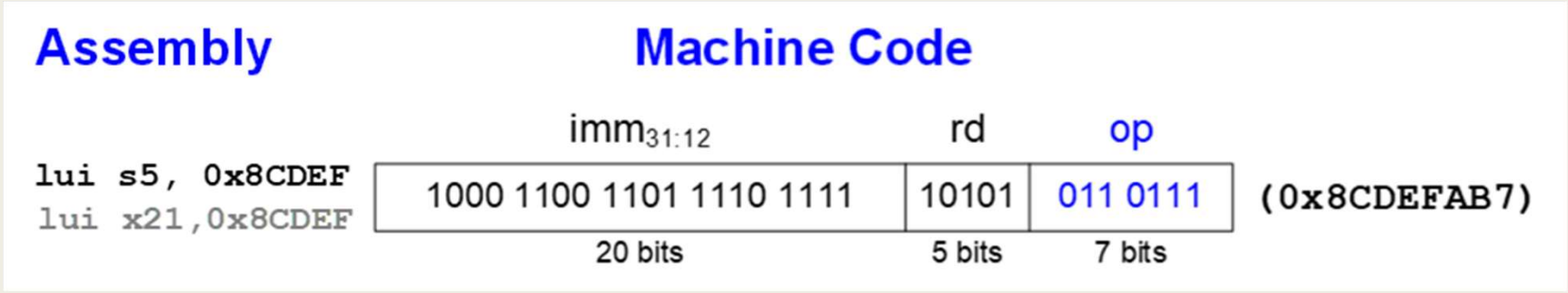
- Γενική μορφή εντολής:

- Πεδία ελέγχου (op):
  - **LUI 0110111**
  - **AUIPC 0010111**
  - **JAL 1101111**

Γλώσσα μηχανής – Εντολές τύπου U/J

- Γενική μορφή εντολής:

- Παραδείγματα εντολών τύπου U (upper immediate):



Γλώσσα μηχανής – Εντολές τύπου U/J

- Γενική μορφή εντολής:

31:12	11:7	6:0	
$imm_{31:12}$	rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$	rd	op	**J-Type**
20 bits	5 bits	7 bits	

- Παραδείγματα εντολών τύπου J (jump):

```
# Address          RISC-V Assembly
0x0000540C         jal ra, func1
0x00005410         add s1, s2, s3
...                ...

0x000ABC04  func1: add s4, s5, s8
...                ...
```

func1 is 0xA67F8 bytes past **jal**

0xABC04 – 0x540C = 0XA67F8

imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0
bit number	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly

```
jal ra, func1
jal x1, 0xA67F8
```

Machine Code

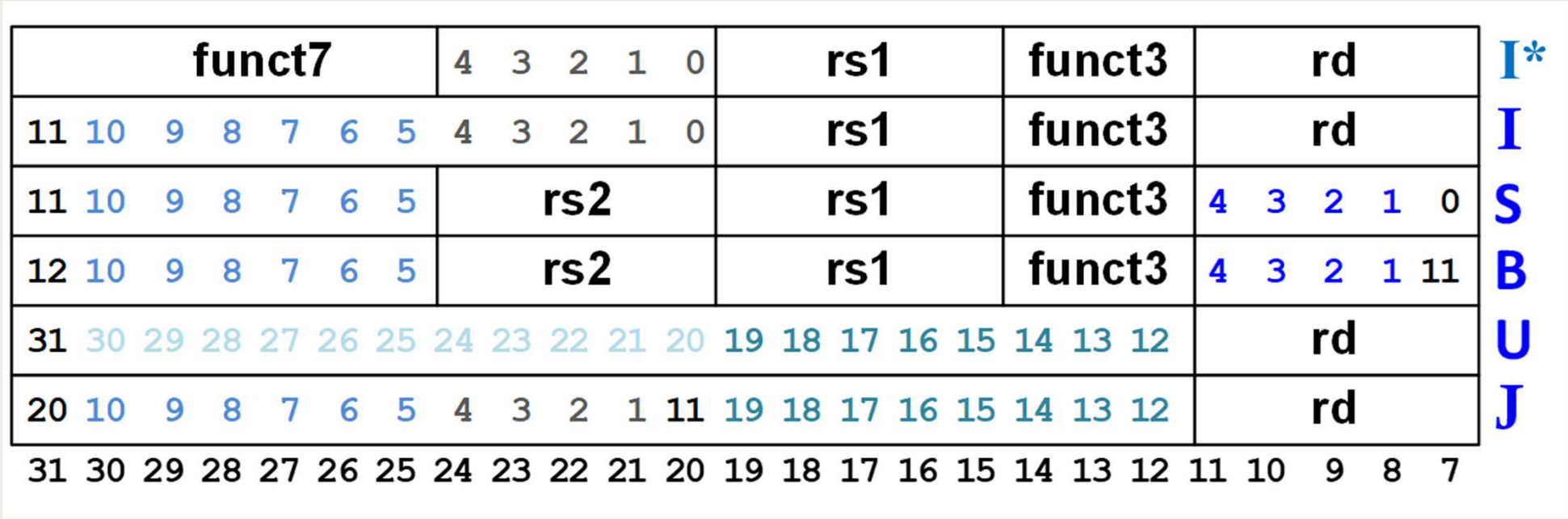
$imm_{20,10:1,11,19:12}$	rd	op
0111 1111 1000 1010 0110	00001	110 1111
20 bits	5 bits	7 bits

(0x7F8A60EF)

Γλώσσα μηχανής – Μορφή εντολών RISC-V

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

Γλώσσα μηχανής – Κωδικοποίηση άμεσου τελεστέου



- Τα bit του άμεσου τελεστέου αποθηκεύονται στις περισσότερες των περιπτώσεων στη σειρά σε κατάλληλα πεδία της εντολής
  - *Απλοποιεί το υλικό για την κατασκευή του μικροεπεξεργαστή*
- Οι άμεσοι τελεστέοι υπόκεινται σε **επέκταση προσήμου στα 32 bit** πριν τη χρήση τους
  - *το bit προσήμου του άμεσου τελεστέου βρίσκεται στο MSB της εντολής*
- * Υπενθυμίζεται ότι στις ολισθήσεις τύπου I η σταθερή ποσότητα ολίσθησης καθορίζεται ως μη προσημασμένος άμεσος τελεστέος των 5 bit (uimm) στη θέση του Rs2 (24:20).
  - *Ειδική περίπτωση: Δεν απαιτεί επέκταση προσήμου*

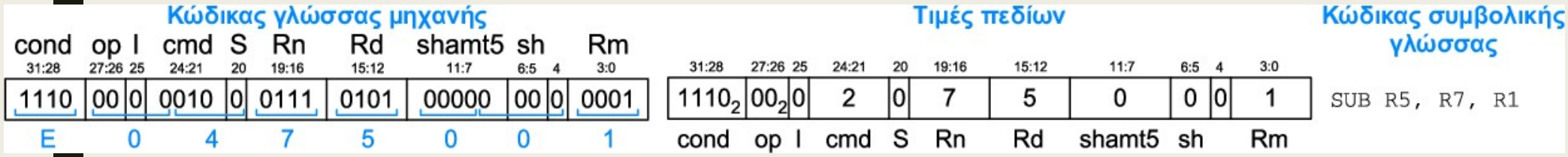
Τρόποι Διευθυνσιοδότησης

- Η αρχιτεκτονική RISC-V χρησιμοποιεί τέσσερεις κύριους τρόπους διευθυνσιοδότησης:
  1. Διευθυνσιοδότηση καταχωρητή (*register addressing*)
  2. Άμεση διευθυνσιοδότηση (*immediate addressing*)
  3. Διευθυνσιοδότηση βάσης (*base addressing*)
  4. PC–σχετική διευθυνσιοδότηση (*PC – relative addressing*)

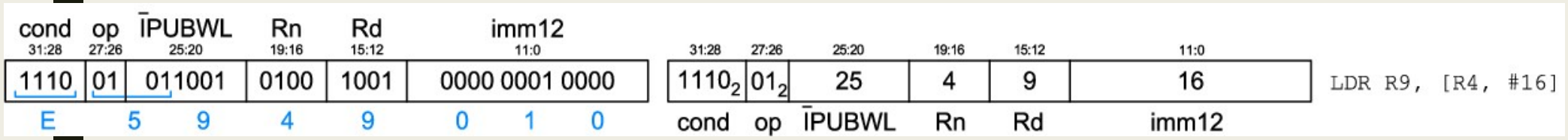
Τρόπος διευθυνσιοδότησης	Παράδειγμα	Περιγραφή
Διευθυνσιοδότηση καταχωρητή	ADD to, s0, s1	t0 ← s0 + s1
Άμεση διευθυνσιοδότηση	SUB t3, s2, 25	t3 ← s2 – 25
Διευθυνσιοδότηση βάσης με σχετική απόσταση άμεσου τελεστέου	SW s0, 77(s1)	mem[s1+77] ← s0
PC–σχετική διευθυνσιοδότηση	J LABEL1	Άλμα στο LABEL1

Γλώσσα μηχανής – Ερμηνεία του κώδικα

- Για να ερμηνεύσει κανείς τη γλώσσα μηχανής, πρέπει πρώτα να αποκωδικοποιήσει τα πεδία κάθε λέξης εντολής των 32 bit
- Όλες οι μορφές εντολών ξεκινούν με ένα πεδίο συνθήκης ***cond*** των 4 bit και ένα πεδίο ***op*** των 2 bit.
  - *Πρώτα εξετάζουμε την τιμή του πεδίου op, ώστε να προσδιορισθούν και τα υπόλοιπα πεδία*
- *Παραδείγματα μετάφρασης γλώσσας μηχανής σε συμβολική γλώσσα:*
  - *0xE0475001 → 1110 0000 0100 0111 0101 0000 0000 0001*
    - Εντολή επεξεργασίας δεδομένων



  - *0xE5949010 → 1110 0101 1001 0100 1001 0000 0001 0000*
    - Εντολή μνήμης



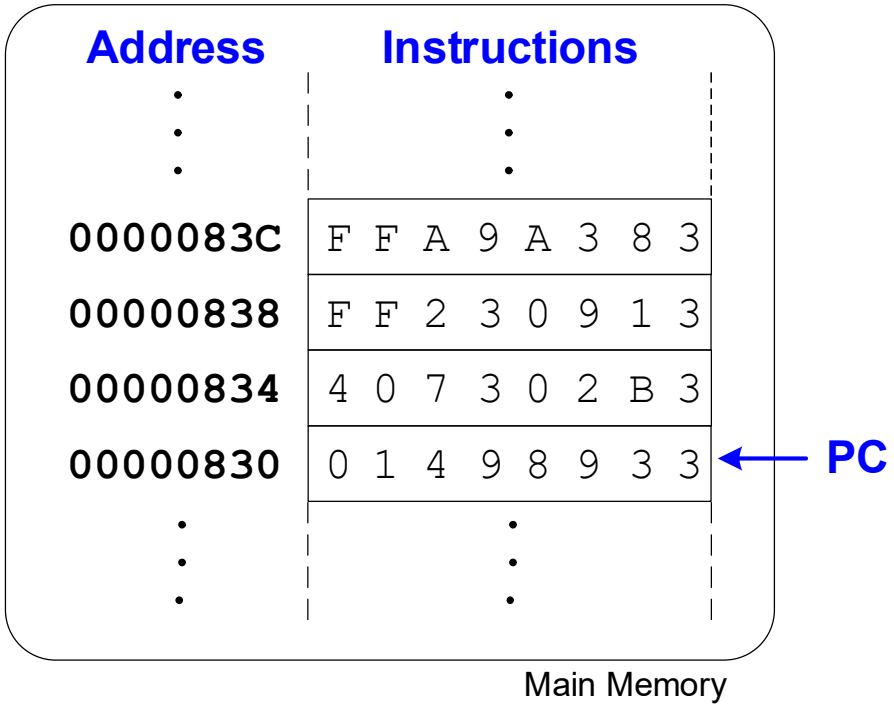
Γλώσσα μηχανής – Αποθηκευμένο πρόγραμμα

- Ένα πρόγραμμα γραμμένο σε γλώσσα μηχανής είναι μια σειρά από αριθμούς μεγέθους 32 bit που αναπαριστούν τις εντολές που μπορούν να αποθηκεύονται στη μνήμη.
  - *η έννοια του **αποθηκευμένου προγράμματος** (stored program)*
- Η εκτέλεση ενός διαφορετικού προγράμματος απαιτεί μονάχα την εγγραφή του νέου προγράμματος στη μνήμη
- Οι εντολές σε ένα αποθηκευμένο πρόγραμμα προσκομίζονται από τη μνήμη και εκτελούνται από τον επεξεργαστή
- Η διεύθυνση της τρέχουσας εντολής διατηρείται σε έναν καταχωρητή των 32 bit ο οποίος ονομάζεται μετρητής προγράμματος (program counter, PC)

Γλώσσα μηχανής – Αποθηκευμένο πρόγραμμα

- Για να εκτελεστεί ο κώδικας της Εικόνας, ο μετρητής PC παίρνει αρχικά ως τιμή τη διεύθυνση 0x00000830 και αυξάνει το περιεχόμενό του κατά 4 για να εκτελέσει την επόμενη εντολή

Assembly Code	Machine Code
`add  s2, s3, s4`	`0x01498933`
`sub  t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw   t2, -6(s3)`	`0xFFA9A383`

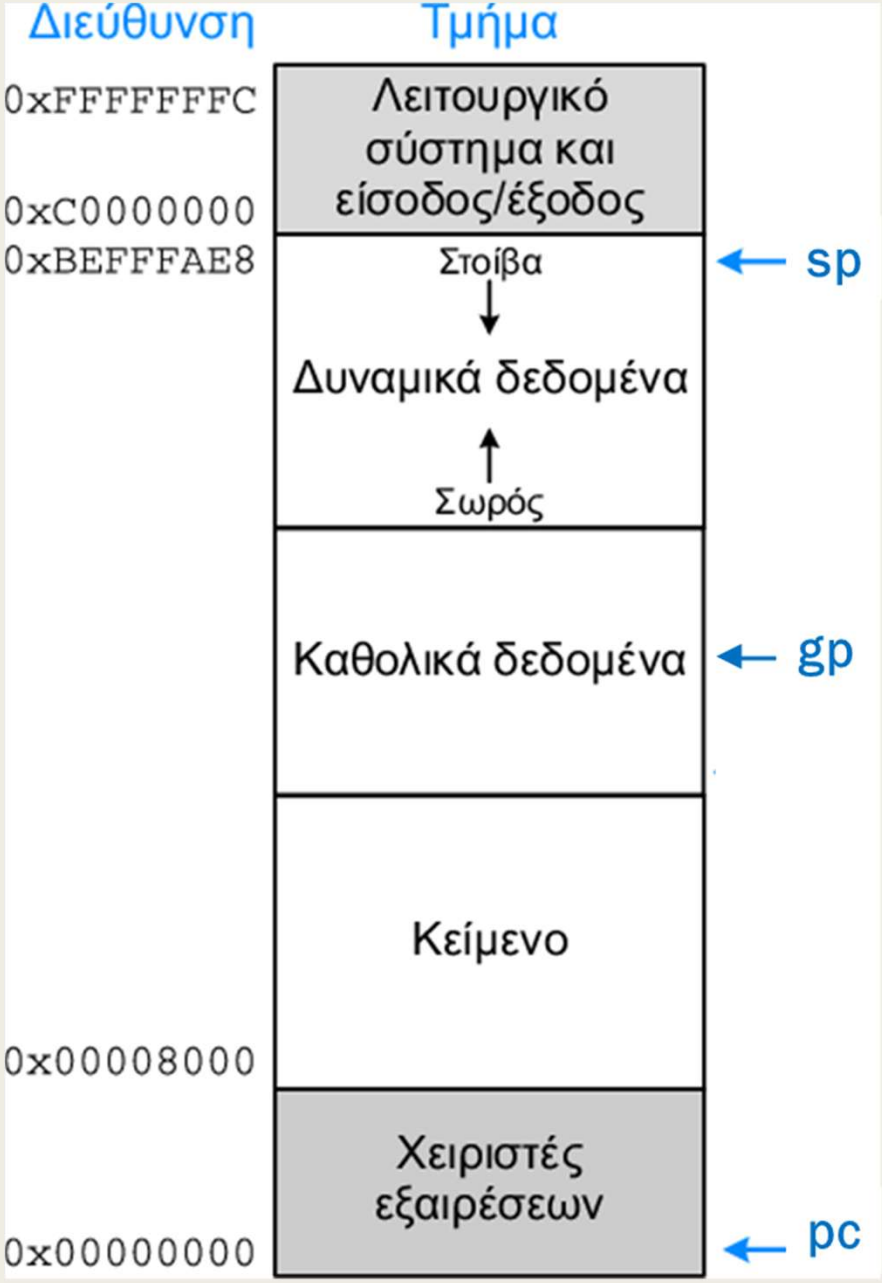


Γλώσσα μηχανής – Αποθηκευμένο πρόγραμμα

- Στην αρχιτεκτονική κατάσταση (*architectural state*) ενός επεξεργαστή είναι αποθηκευμένη η κατάσταση ενός προγράμματος
- Στην RISC-V, η αρχιτεκτονική κατάσταση περιλαμβάνει το αρχείο καταχωρητών και τον μετρητη PC
  - *Αποθηκεύοντας και επαναφέροντας αργότερα την αρχιτεκτονική κατάσταση ενός προγράμματος, το πρόγραμμα δεν αντιλαμβάνεται ότι διακόπτεται η εκτέλεσή του*

Ο χάρτης μνήμης

- Ο χώρος διευθύνσεων της αρχιτεκτονικής RISC-V καταλαμβάνει 2^{32} byte (4 GB) από 0 έως 0xFFFFFFFC
- Χωρίζεται σε 5 τμήματα:
  - *το τμήμα κειμένου, όπου* αποθηκεύεται το πρόγραμμα σε γλώσσα μηχανής
  - *το τμήμα καθολικών δεδομένων για στατικά δεδομένα, που δεσμεύεται στη μνήμη προτού ξεκινήσει η εκτέλεση του προγράμματος και προσπελαύνεται από τον καθολικό δείκτη (global pointer)*
  - *το τμήμα δυναμικών δεδομένων, που δεσμεύονται και αποδεσμεύονται δυναμικά καθ' όλη τη διάρκεια της εκτέλεσης του προγράμματος και απαρτίζεται από τη στοίβα και τον σωρό*
  - *το τμήμα για χειριστές εξαιρέσεων που ξεκινάει από τη διεύθυνση 0x0*
  - *το τμήμα για το λειτουργικό σύστημα και την είσοδο/έξοδο* με απεικόνιση στη μνήμη



Σύγκριση των αρχιτεκτονικών RISC-V και MIPS

- Η αρχιτεκτονική RISC-V εμφανίζει πολλές ομοιότητες με την αρχιτεκτονική MIPS που αναπτύχθηκε από τον John Hennessy στη δεκαετία του 1980, αλλά εξαλείφει ένα μέρος άσκοπης πολυπλοκότητας –εισάγοντας ταυτόχρονα νέα πολυπλοκότητα όπως στην περίπτωση των άμεσων τελεστέων!
- Στις ομοιότητες περιλαμβάνονται οι μορφές του κώδικα μηχανής και του κώδικα συμβολικής γλώσσας, τα μνημονικά εντολών, η ονοματοδοσία καταχωρητών, και οι συμβάσεις που χρησιμοποιούνται για τη στοίβα και τις κλήσεις.
- Στις διαφορές περιλαμβάνονται:
  - *το μέγεθος των άμεσων τελεστέων και οι κωδικοποιήσεις,*
  - *η διακλάδωση σε σχέση με τον μετρητή PC (αντί του PC + 4)*
  - *ο σταθερός ορισμός για όλες τις εντολές των πεδίων των εντολών που αφορούν τους καταχωρητές προέλευσης και προορισμού*
  - *το διαφορετικό πλήθος προσωρινών και αποθηκευμένων καταχωρητών καθώς και καταχωρητών ορισμάτων*
  - *πρόσθετη επεκτασιμότητα μέσω της συμπερίληψης περισσότερων bit ελέγχου στην εντολή*
- Διατηρώντας τους καταχωρητές rs1, rs2 και rd στα ίδια πεδία bit κάθε τύπου εντολής που τους χρησιμοποιεί, η αρχιτεκτονική RISC-V απλουστεύει το υλικό του αποκωδικοποιητή συγκριτικά με την αρχιτεκτονική MIPS

Σύγκριση των αρχιτεκτονικών RISC-V και ARM

- Η ARM είναι μια αρχιτεκτονική RISC που αναπτύχθηκε την ίδια περίπου εποχή όπως η αρχιτεκτονική MIPS, στη δεκαετία του 1980.
- Οι ομοιότητες της ARM με την RISC-V είναι, μεταξύ άλλων, οι λίγες μορφές κώδικα μηχανής και εντολών συμβολικής γλώσσας, καθώς και οι παρόμοιες συμβάσεις της στοίβας και των κλήσεων.
- Οι διαφορές της ARM από την RISC-V είναι, μεταξύ άλλων, η εκτέλεση υπό συνθήκη, οι σύνθετοι τρόποι διευθυνσιοδότησης για την προσπέλαση της μνήμης, η ικανότητά της να τοποθετεί και να ανακτά πολλούς καταχωρητές στην/από τη στοίβα με χρήση μίας εντολής, οι προαιρετικοί ολισθαίνοντες καταχωρητές προέλευσης, και οι μη συμβατικές κωδικοποιήσεις άμεσων τελεστέων.
  - *Οι άμεσοι τελεστέοι κωδικοποιούνται ως μια τιμή μεγέθους 8 bit και μια περιστροφή (rotation) μεγέθους 4 bit, και κωδικοποιούν μόνο θετικούς άμεσους τελεστέους (η αφαίρεση καθορίζεται από τα bit ελέγχου).*
- Αυτές οι δυνατότητες της ARM –ειδικά η εκτέλεση υπό συνθήκη, οι ολισθαίνοντες καταχωρητές και οι τρόποι διευθυνσιοδότησης– συνήθως υπάρχουν μόνο στις αρχιτεκτονικές CISC, αλλά η ARM τις συμπεριλαμβάνει ώστε να μειώσει το μέγεθος του προγράμματος και, άρα, της μνήμης, κάτι που είναι πολύ σημαντικό στις ενσωματωμένες συσκευές και τις συσκευές χειρός.
  - *Όμως αυτές οι σχεδιαστικές αποφάσεις έχουν ως αποτέλεσμα πιο περίπλοκο υλικό σε σχέση με την RISC-V.*