5a
RISC-V in ukazi load/store

BRANKO ŠTER

Prevajanje ukazov

Prevajalnik programe, napisane v višjem programskem jeziku, lahko prevede v zbirni jezik (zbirnik pa nato v strojni jezik), pogosto pa kar neposredno v strojni jezik

- Primer 1 (iz jezika C v zbirni jezik):
  - Predpostavimo zaenkrat, da se vrednosti spremenljivk a, b in c že nahajajo v registrih x5, x6 in x7

```
a = b + c;              // v jeziku C
add x5, x6, x7          ; Pomen: x5 ← x6 + x7
```

- Primer 2:

```
a = b + c + d + e;      // x1: a, x2: b, x3: c, x4: d, x5: e
add x1, x2, x3          ; 3 ukazi
add x1, x1, x4          ;   v zbirnem
add x1, x1, x5          ;     jeziku
```

- Primer 3:

```
A[12] = h + A[8];          // x1: A(=naslov), x3: h

lw  x2, 32(x1)    ; x2 ← M[x1+32]      (32=8*4)
add x2, x2, x3    ; x2 ← x2 + x3
sw  x2, 48(x1)    ; M[x1+48]← x2       (48=12*4)
```

- Operand je lahko tudi konstanta
  - **takojšnji** (**immediate**) operand

```
addi  x1, x2, 5          ; x1 ← x2 + 5
```

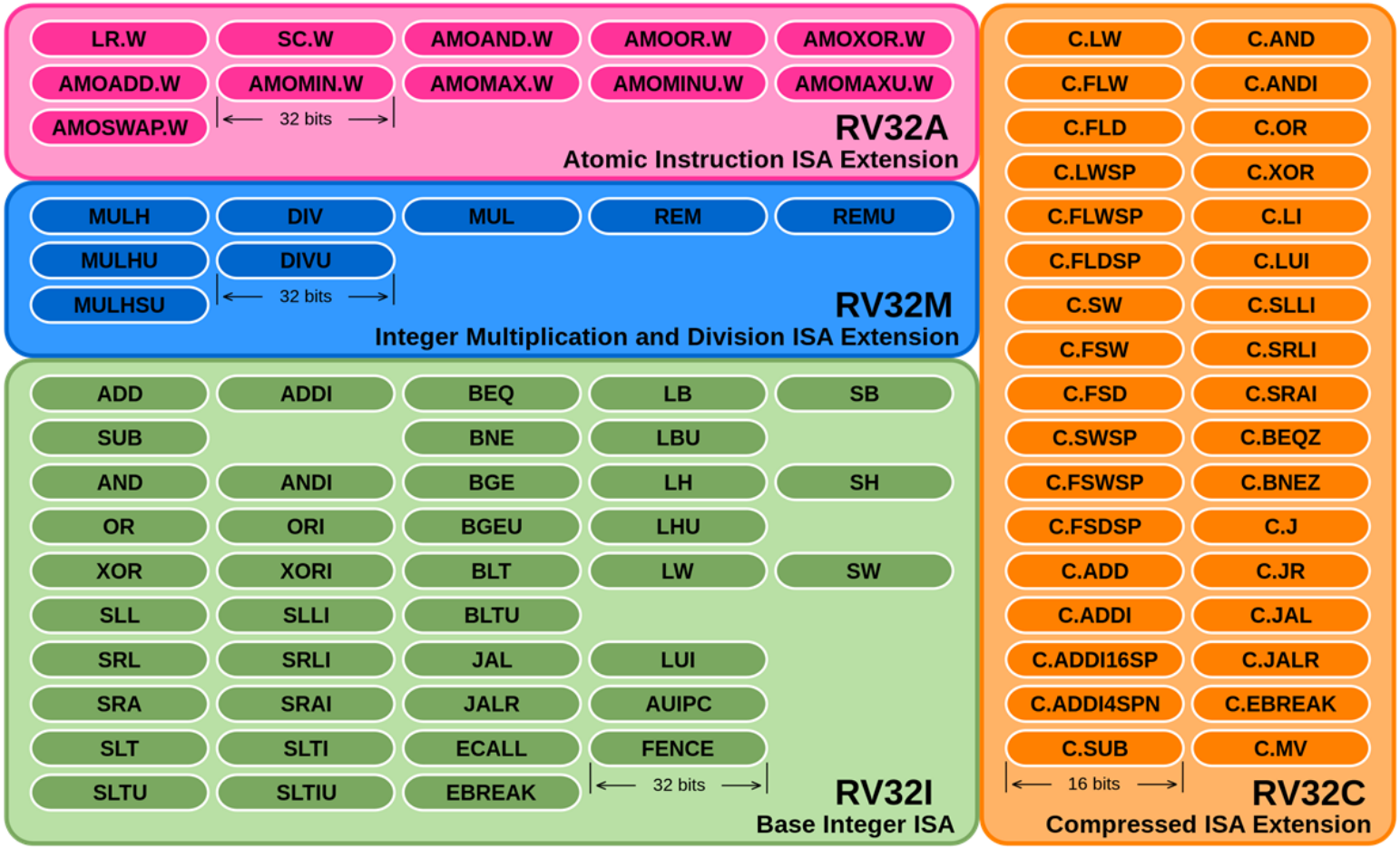
(add immediate)

Ukazna arhitektura (ISA)

- **Ukazna arhitektura** (**Instruction Set Architecture**, **ISA**)
  - natančno definira vse ukaze (nabor ukazov) nekega procesorja
  - ne govori pa o implementaciji
- RISC-V se vedno bolj uveljavlja kot odprta RISC arhitektura
  - Druge RISC arhitekture: ARM, MIPS, ...

- RISC-V je ukazna arhitektura (instruction-set architecture, ISA), ki je bila prvotno razvita za raziskave in poučevanje računalniških arhitektur
  - vse bolj pa postaja tudi standard na področju odprtih računalniških arhitektur za industrijske implementacije
  - RISC-V ISA je definirana brez detajlov implementacije
  - RISC-V je dejansko družina
    - **RV32I - celoštevilska 32-bitna** (XLEN=32)
    - RV64I - celoštevilska 64-bitna (XLEN=64)
    - RV128I - celoštevilska 128-bitna (XLEN=128)
    - RV32E - za majhne mikrokrmilnike (Embedded)
    - Razširitve (extensions):
      - M (množenje/deljenje),
      - F (plavajoča vejica, enojna natančnost),
      - D (plavajoča vejica, dvojna natančnost),
      - A (atomski ukazi),
      - ...

RV32IMAC



By Eduardo Corpeño - Own work using: Inkscape. This is the first illustration of a series on the RISC-V ISAs, which is available on risc-v-diagrams on GitHub, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=118039169

Lastnosti RISC-V

- 8-bitna pomnilniška beseda
- 32-bitni pomnilniški naslov
- Način shranjevanja operandov v CPE
  - 32 32-bitnih splošnonamenskih registrov x0, x1, …, x31
    - Vsebina x0 je vedno 0 (pri pisanju vanj se ne zgodi nič)
- Število eksplicitnih operandov v ukazu
  - vsi ALE ukazi imajo 3 eksplicitne operande
    - Tip R ima dva izvorna (source) registra *rs1* in *rs2* ter en ciljni destination register *rd*
    - Tip I ima en izvorni (source) register *rs1*, 12-bitni takojšnji (immediate) operand *imm* ter en ciljni destination register *rd*

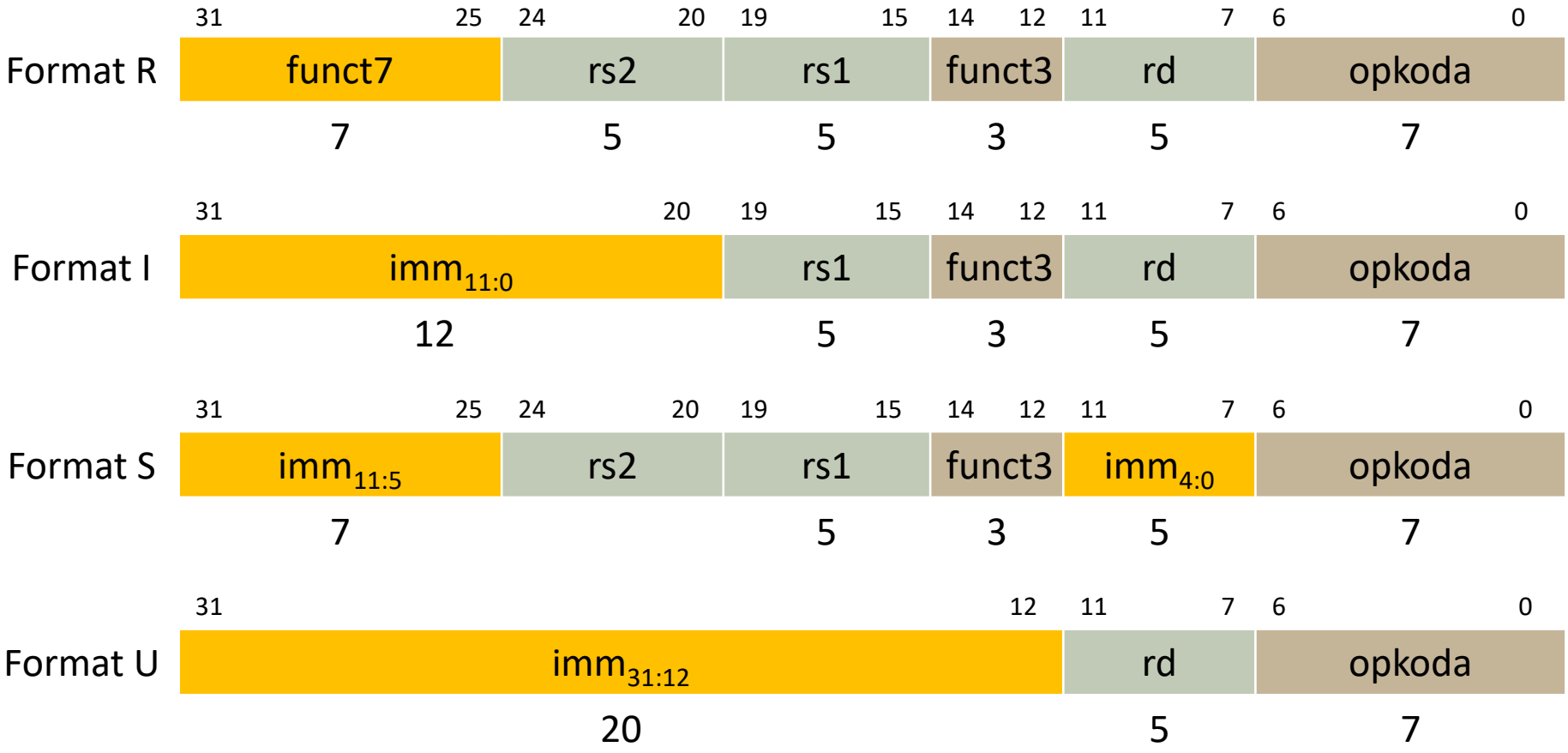
Lokacija operandov in načini naslavljanja

- Lokacija operandov
  - registrsko-registrski (load/store) računalnik
    - pomnilniški operandi nastopajo samo v ukazih load in store
  - pri ALE ukazih 2 operanda v registrih
    - tretji v registru ali takojšnji
  - dostop do operandov v pomnilniku le z load in store

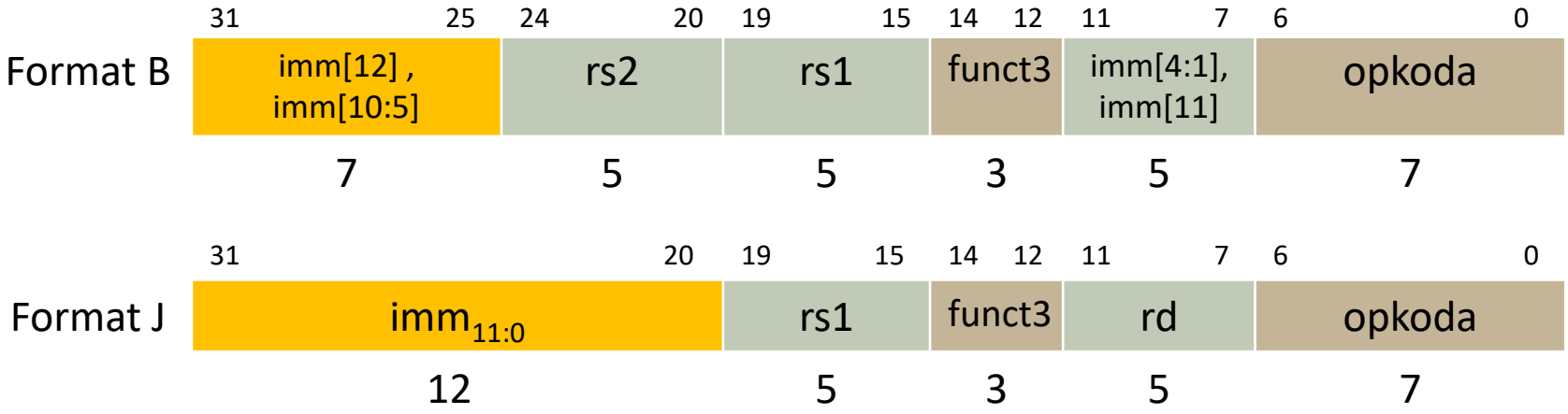
- Operacije in operandi
  - pomnilniški operandi so lahko 8-, 16- ali 32-bitni (bajt, polbeseda, beseda)
  - 16- in 32-bitni operandi so v pomnilniku shranjeni po **pravilu tankega konca** (**little endian rule**)
    - dobro je, da so **poravnani** (**aligned**), kar pomeni, da je operand, ki je sestavljen iz več bajtov, na naslovu, ki je deljiv s številom bajtov
      - 16-biten operand je na naslovu, deljivem z 2
      - 32-biten operand na naslovu, deljivem s 4
    - sicer je potreben prenos operanda v dveh kosih
  - vse ALE operacije so 32-bitne
    - 8- in 16-bitni operandi se pri load pretvorijo v 32-bitne
      - Razširitev ničle pri nepredznačenih (LBU, LHU)
      - Razširitev predznaka pri predznačenih (LB, LH)

Format ukazov pri RV32I:

- vsi ukazi so 32-bitni in poravnani
- 4 osnovni formati (R, I, S, U)



- 2 dodatna formata (B, J):



Format ukaza nam pove, kaj pomenijo posamezni biti v strojni kodi ukaza

- rs (source register): iz njega se bere,
- rd (destination register): vanj se piše

Število ukazov nabora RV32I:

- 40
- ni ukazov za množenje, deljenje
- ni ukazov v plavajoči vejici

Vrste ukazov

- Ukaze RISC-V delimo v več skupin:
  1. ukazi za prenos podatkov (load, store)
     - gre za prenos *operandov* med registri in pomnilnikom
     - nalaganje (iz pomnilnika) in shranjevanje (v pomnilnik)
  2. ALE ukazi
     - aritmetične in logične operacije
  3. kontrolni ukazi
     - skoki
  4. sistemski ukazi

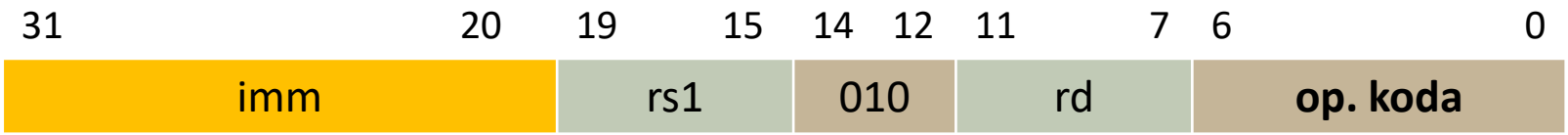
Ukazi za prenos podatkov (load/store)

- Uporabljajo format I z baznim naslavljanjem (bazni register je rs1)

Load word: `lw rd, imm(rs1)` ; rd $\leftarrow_{32}$ M[rs1 + se(imm)]

Npr.: `lw x5, 30(x6)` ; x5 $\leftarrow_{32}$ M[30 + x6]

Format I:



12 bitov	5 bitov	3 biti	5 bitov	7 bitov
	x6		x5	lw
000000011110	00110	010	00101	0000011
30	6	2	5	3

Celoten ukaz v strojni kodi: 0x01E32283

Torej: ukaz v zbirnem jeziku lw x5,30(x6) zbirnik 'prevede' v strojni ukaz 00000001111000110010001010000011

- M[x] je vsebina pomnilniške besede na naslovu x
- Znak $\leftarrow_{32}$ pomeni 32-bitni prenos iz (ali v) naslovov x, x+1, x+2, x+3 po pravilu debelega konca
- Znak $\leftarrow_{16}$ pomeni 16-bitni prenos iz (ali v) naslovov x, x+1
- Znak $\leftarrow_{8}$ pomeni 8-bitni prenos iz (ali v) naslov x
- Znak $\leftarrow_{raz}$ pomeni razširitev bita

Pri ukazih store je *Rd* izvor

Razširitev operanda

Kaj, če je vrednost, ki jo želimo naložiti v (32-bitni) register, krajša od njegove dolžine?

- Na katero vrednost postavimo preostale bite?

2 možnosti:

- razširitev predznaka
  - lb (load byte (signed))
    - Npr.: 0x6F (01101111) se razširi v 0x0000006F (00000000 00000000 00000000 01101111),

      0x94 (10010100) se razširi v 0xFFFFFF94 (11111111 11111111 11111111 10010100)
  - lh (load halfword (signed))
    - Npr.: 0x73A1 (01110011 10100001) se razširi v 0x000073A1 (00000000 00000000 01110011 10100001),

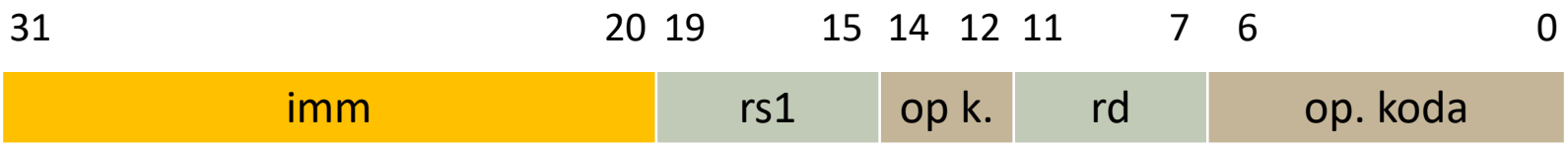
      0xC40A (11000100 00001010) se razširi v 0xFFFFC40A (11111111 11111111 11000100 00001010)
- razširitev ničle
  - lbu (load byte unsigned)
    - Npr., 0x94 (10010100) se razširi v 0x00000094 (00000000 00000000 00000000 10010100)
  - lhu (load halfword unsigned)
    - Npr., 0xC40A (11000100 00001010) se razširi v 0x0000C40A (00000000 00000000 11000100 00001010)

Load byte:

`lb x5,80(x6)` ; $x5_{31..8} \leftarrow_{raz} M[80 + x6]_7$, $x5_{7..0} \leftarrow_8 M[80 + x6]$

$x5_{31..8}$ so biti od 31 do 8 (najvišjih 24 bitov),

$\leftarrow_{raz}$ pomeni razširitev predznaka



12 bitov	5 bitov	3biti	5 bitov	7 bitov
	x6	lb*	x5	lb*
000001010000	00110	000	00101	0000011
80	6	0	5	3

** lb sta oba dela skupaj (3+7 bitov)*

Celoten ukaz v strojni kodi: 0x05010083

Load byte unsigned

```
lbu  x5,80(x6)
```

pomen: $x5_{31..8} \leftarrow_{raz} 0$, $x5_{7..0} \leftarrow_{8} M[80 + x6]$

tu se 0x6F razširi enako kot 0x94

Podobno za 16-bitne besede:

- **Load halfword**
  - halfword (polbeseda) je 16 bitov: 2B
- **Load halfword unsigned**

Ukazi za prenos podatkov (load/store):

Format	Op. koda	Ukaz	Opis
I	000 0000011	LB	Load byte
I	001 0000011	LH	Load halfword
I	010 0000011	LW	Load word
I	100 0000011	LBU	Load byte unsigned
I	101 0000011	LHU	Load halfword unsigned
S	000 0100011	SB	Store byte
S	001 0100011	SH	Store halfword
S	010 0100011	SW	Store word

- Odmik je 12-biten z razširitvijo predznaka (torej je lahko tudi negativen)
- Pri ukazih load za 8- in 16-bitne operande sta 2 varianti:
  - običajna (signed): razširitev predznaka (do 32 bitov)
  - unsigned: razširitev ničle (do 32 bitov)

Osnovne direktive zbirnika

`.data`	– začetek podatkovnega segmenta
`.text`	– začetek ukaznega segmenta
`.byte <n1>(,<n2>..)`	– določi zaporedna 8-bitna števila
`.half <n1>(,<n2>..)`	– določi zaporedna 16-bitna števila
`.word <n1>(,<n2>..)`	– določi zaporedna 32-bitna števila
`.dword <n1>(,<n2>.. )`	– določi zaporedna 32-bitna števila
`.align <n>`	– poravnava

Direktive so namenjene zbirniku (programu), ne procesorju!

Primer programa v zbirnem jeziku za RISC-V

```
        .data   (podatkovni segment; vzemimo, da se začne na naslovu 0x400 = 1024_10)
A:      .byte 5         (bajt z vrednostjo 5 na naslovu A = 1024)
B:      .byte 6         (bajt z vrednostjo 6 na naslovu B = 1025)
C:      .byte 0         (bajt z vrednostjo 0 na naslovu C = 1026)

        .text                   (kodni segment; običajno bo kar na naslovu 0)
        lb x2, A(x0)            (x2 ← M[1024 + 0])
        lb x3, B(x0)            (x3 ← M[1025 + 0])
        add x4, x2, x3          (x4 ← x2 + x3)
        addi x5, x0, C          (x5 ← 1026)
        sb x4, 0(x5)            (x4 → M[1026+0])
```

Uporaba oznak

Namesto oznake (labele) A lahko pišemo tudi kar neposredno naslov 0x400 (vsaka labela vsebuje pomnilniški naslov – bodisi ukaza, bodisi operanda)

```
lb x5, 0x400(x0)
```

(Če je bazni register različen od 0, ga je prej seveda treba naložiti)

Toda, pozor pri operacijah store!

- Npr. **`sb x4,C(x0)`** ne deluje, kot bi pričakovali
- Tak ukaz Ripes tolmači kot psevdoukaz in pred njim doda ukaz auipc (to bomo obravnavali kasneje), ki baznemu registru da vrednost PC.
- Ker pa se x0 ne more spremeniti, ne dobi te vrednosti
- Rešitev je, da namesto x0 uporabimo katerikoli register drug register, za katerega nam ni pomembno, ali se njegova vrednost spremeni (služi le kot začasni register), npr. **`sb x4,C(x8)`**
- Deluje pa tudi **`sb x4,0x402(x0)`**, vendar pri 'ročnem' pisanju v zbirnem jeziku to ni prikladno, ker programmer običajno ne pozna naslova (razen v takem preprostem primeru, kot je naš)