Mikroprocesorji

Zgradba mikroprocesorja RISC V in zbirnik

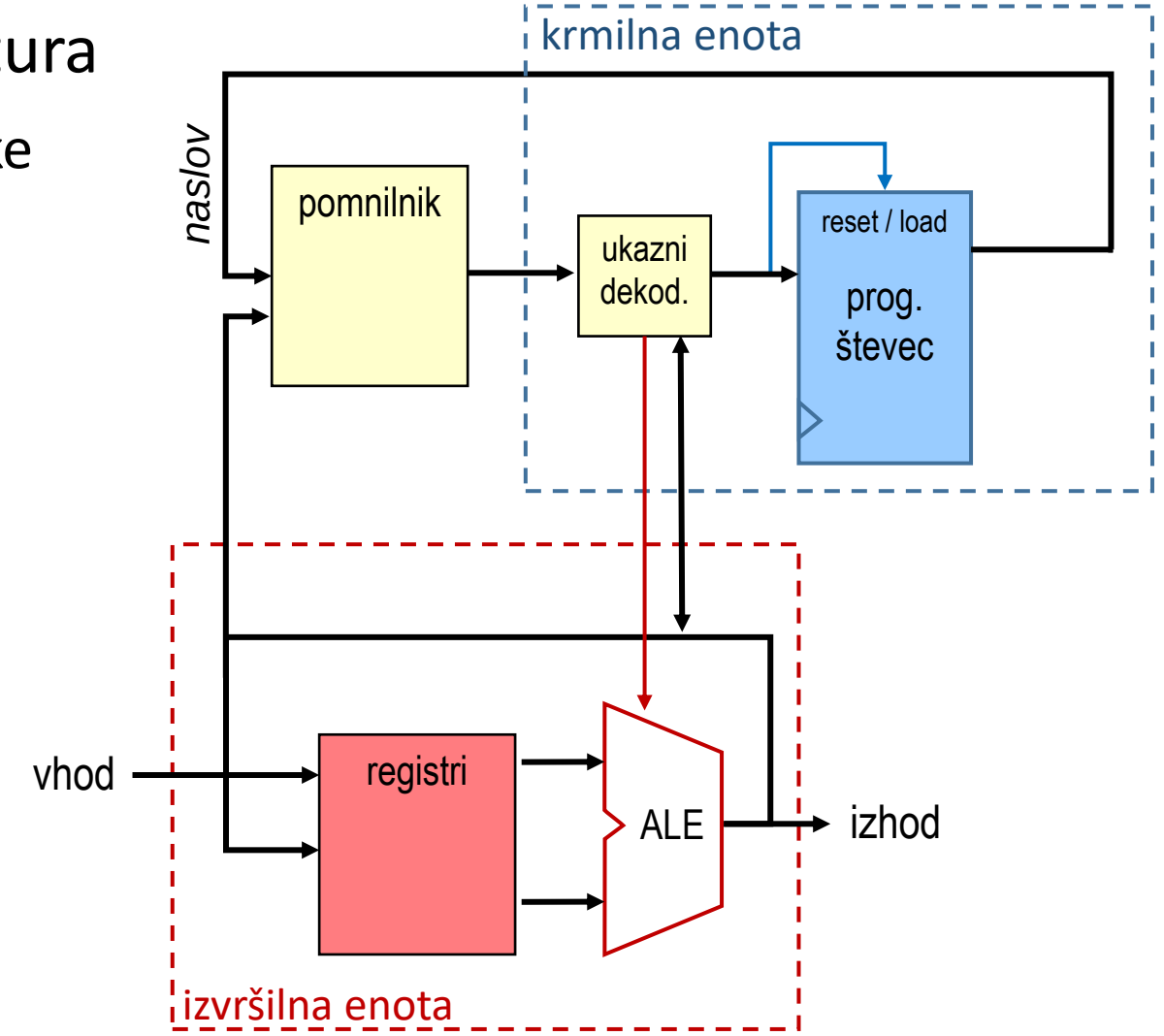
Andrej Trost

UNIVERZA V LJUBLJANI
FE
Fakulteta za elektrotehniko

Mikroprocesor

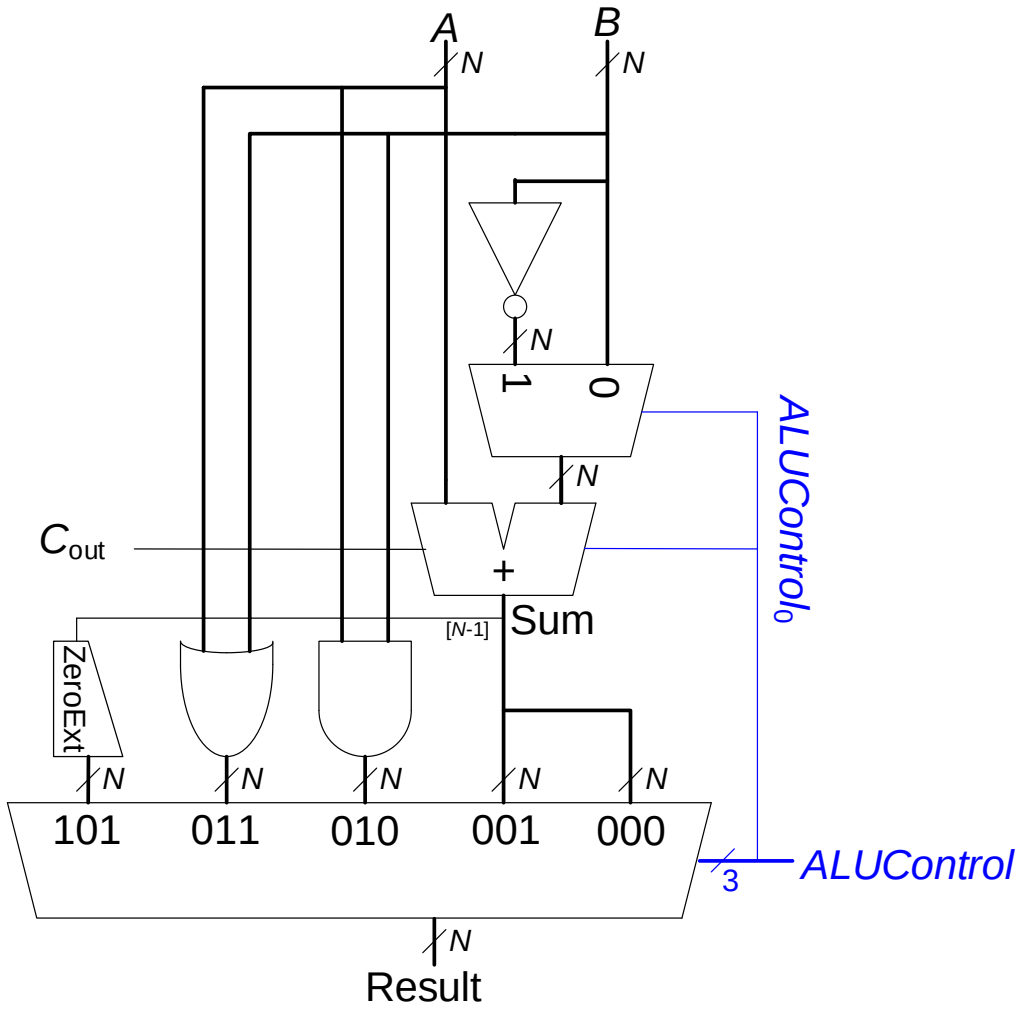
Von Neumannova arhitektura

- pomnilnik za ukaze in podatke
- centralna procesna enota (CPE)



Aritmetično logična enota

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT



Harwardska arhitektura

Delovanje CPE določa nabor ukazov

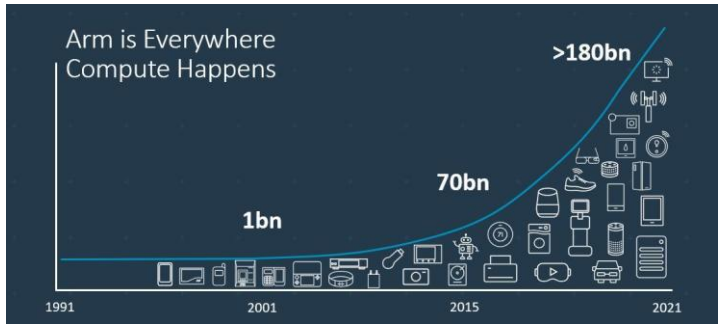
- kompleksen (CISC) ali reduciran nabor ukazov (RISC)

Pravila načrtovanja mikroprocesorja RISC

1. preprosta (regularna) zgradba
2. hitra izvedba pogostih ukazov
3. manjše vezje je hitrejše
4. dober načrt zahteva dobre kompromise

Aktualne arhitekture procesorjev RISC

- MIPS (1980)
  - Silicon Graphics, Nintendo, Cisco, >300 miljonov čipov
- ARM (1985)
  - Advanced RISC Machine



- RISC-V (2010)
  - odprta arhitektura ukazov (32, 64 ali 128 bitni)
  - osnovna: celoštevilske operacije RV32I oz. RV32E (le 16 registrov)
  - M – množenje in deljenje, A – atomični ukazi,
  - F – op. s plavajočo vejico, D – plavajoča vejica dvojne natančnosti
  - C – zgoščeno kodiranje ukazov (16 bitov)

Mikroprocesor RISC-V

- odprta arhitekturo ukazov, ne potrebujemo licence za izdelavo CPE
- začetek na UC Berkeley, standard določa neprofitna fundacija RISC-V

Zbirniški ukazi RISC-V

- obravnavali bomo nekaj ukazov iz osnovnega nabora RV32I za delo z 32-bitnimi celoštevilskimi podatki
- preprosta zgradba
  - večina ukazov ima 3 argumente: rd = register za rezultat, rs1,rs2 = register
- Primer ukazov za vsoto ali razliko podatkov v registrih in prištevanje konstante (za znakom # je komentar, ki razlaga ukaz)

```
add  s2, s1, s0    # s2=s1+s0
sub  s2, s1, s0    # s2=s1-s0
addi s3, s2, 1     # s3=s2+1
```

  - oznaka i pomeni, da je en izmed argumentov konstanta (immediate) in ne vsebina registra

Strojna koda

- prevajalnik pretvori zbirniške ukaze v strojno kodo
  - ukaz se pretvori v 32-bitno besedo
  - posamezni biti določajo kodo in funkcijo ukaza, naslove registrov (5 bitov, ker imamo 32 registrov) in/ali takojšnjih konstant (immediate)

- primer strojne kode, ki se shrani v pomnilnik (šestnajstiški zapis)

ram

0	00848933
4	40848933
8	---

```
add s2,s1,s0
sub s2,s1,s0
```

https://www.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/

Psevdoukazi zbirnika

- zbirnik pozna poleg ukazov, ki se prevedejo v strojno kodo tudi psevdoukaze zaradi lažjega pisanja programov
- nalaganje konstante (li = Load Immediate)
  - RISC-V nima strojnega ukaza za nalaganje konstante v register (LOAD)
  - za opravilo uporabimo ukaz za prištevanje konstante, pri katerem seštejemo x0 + konst (x0 ima vedno vrednost 0)

```
li x2, 5   # x2=5
```

naredi isto kot strojni ukaz

```
addi x2, x0, 5
```

- premik podatka med registri (mv = move)
  - namesto strojnega ukaza za premikanje bo RISC-5 izvedel seštevanje

```
mv x3, x2 # x3=x2
```

naredi isto kot strojni ukaz

```
add  x3, x2, x0
```

Registri

- RISC-V ima 32 registrov x0 - x31 (izvedba RV32E ima le prvih 16)
  - x0 je poseben: predstavlja konstanto 0 in ga lahko le beremo 32-bitnih vrednosti

ime	oznaka	uporaba (Application Binary Interface)
zero	x0	konstanta 0, tega registra ne moremo zapisovati
ra	x1	povratni naslov (return)
sp	x2	kazalec na sklad
gp	x3	globalni kazalec
tp	x4	kazalec niti
t0-2	x5-7	začasne spremenljivke
s0/fp	x8	spremenljivke / kazalec okvirja
s1	x9	spremenljivke
a0-1	x10-11	argumenti in izhodi funkcij
a2-7	x12-17	argumenti funkcij
s2-11	x18-27	spremenljivke
t3-6	x28-31	začasne spremenljivke

v zbirniškem programu zapišemo ime registra (npr. zero) ali pa oznako (x0)

```
addi s1, zero, 1
```

ali

```
addi x9, x0, 1
```

Spremenljivke in pomnilnik

- spremenljivke shranjujejo podatke, ki jih potrebuje program
- če je dovolj prostora v registrih, dodeli prevajalnik posamezen register
  - s podatki v registrij lahko CPE neposredno izvaja računske operacije
- sicer pa so podatki shranjeni v pomnilniku
  - za izvedbo operacij, se mora naprej podatek naložiti (lw=load word) v register, po operaciji pa shraniti nazaj v pomnilnik (sw=store word)
- Primer: izračun a = b+c,

  a=RAM[100], b=RAM[104], c=RAM[108]

```
lw   x5, 0x104(x0)      # naloži b
lw   x6, 0x108(x0)      # naloži c
add  x4, x5, x6
sw   x4, 0x100(x0)      # shrani a
```

Nalaganje podatkov iz pomnilnika

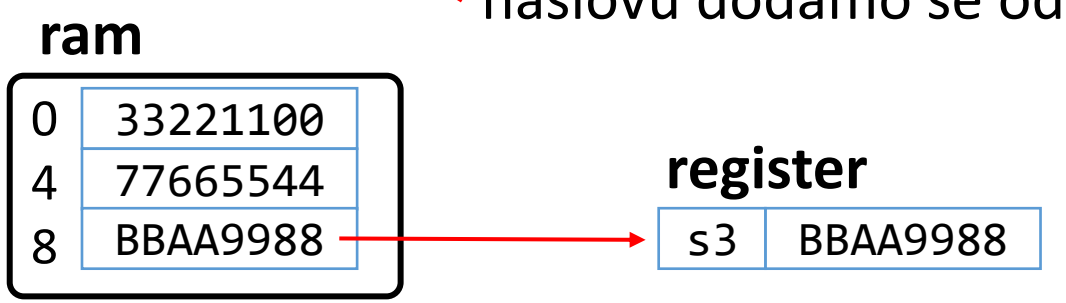
- v pomnilniku so 32 bitne besede na naslovih, ki se povečujejo za 4
  - pomnilniške naslove štejemo po bajtih (8-bit), vsaka beseda zavzame 4 bajte
- nalaganje besede (lw = Load Word)

Primer:

- prenos podaka iz pomnilnika na naslovu 8 v register s3

```
lw s3, 8(zero)
```

naslovu dodamo še odmik, ki je v našem primeru 0

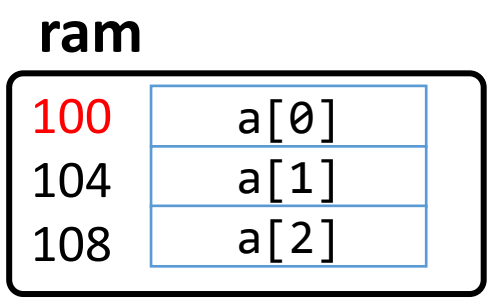


Branje elementov zbirke

- lw in sw imata 3 argumente: **`register, odmik(naslovni_reg)`**
  - naslov pomnilnika je vsota naslovnega registra in odmika
- uporabno za branje podatka iz zbirke 32-bitnih vrednosti

t0 = a[0] in t1 = a[2]

- bazni naslov zbirke (naslov a[0])
- npr. zbirka a je na naslovu 0x100




```
li s0,0x100       # s0=bazni naslov a[]
lw t0,0(s0)       # branje a[0]
lw t1,8(s0)       # branje a[2]
```

Ukazi s konstanto

- izsek kode v jezku C se prevede v:

```
int a = -372;
int b = a + 6;
```

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

- omejitev RISC-V: konstanta je 12-bitna predznačena vrednost
  - večina konstant predstavlja majhne vrednosti
- RISC-V ne pozna odštevanja s konstanto. Zakaj?

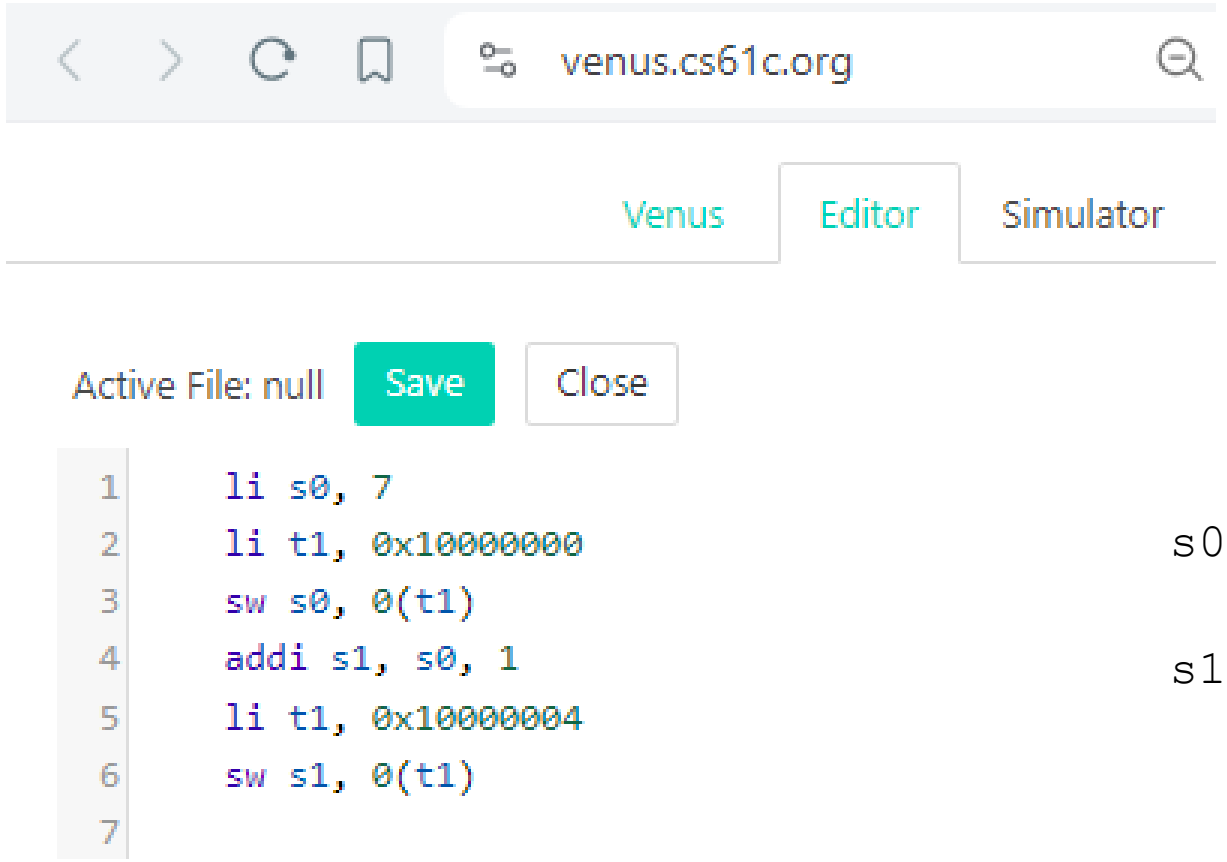
32-bitne konstante naložimo v register z dvema ukazoma:
lui (load upper immediate, 20 bitov) in **addi**

```
int a = 0xFEDC8765;
```

```
# naloži zgornjih 20 bitov
lui  s0, 0xFEDC8
# in prištej spodnjih 12
addi s0, s0, 0x765
```

Simulator Venus: https://venus.cs61c.org/

- v urejevalnik napišemo zbirniški program

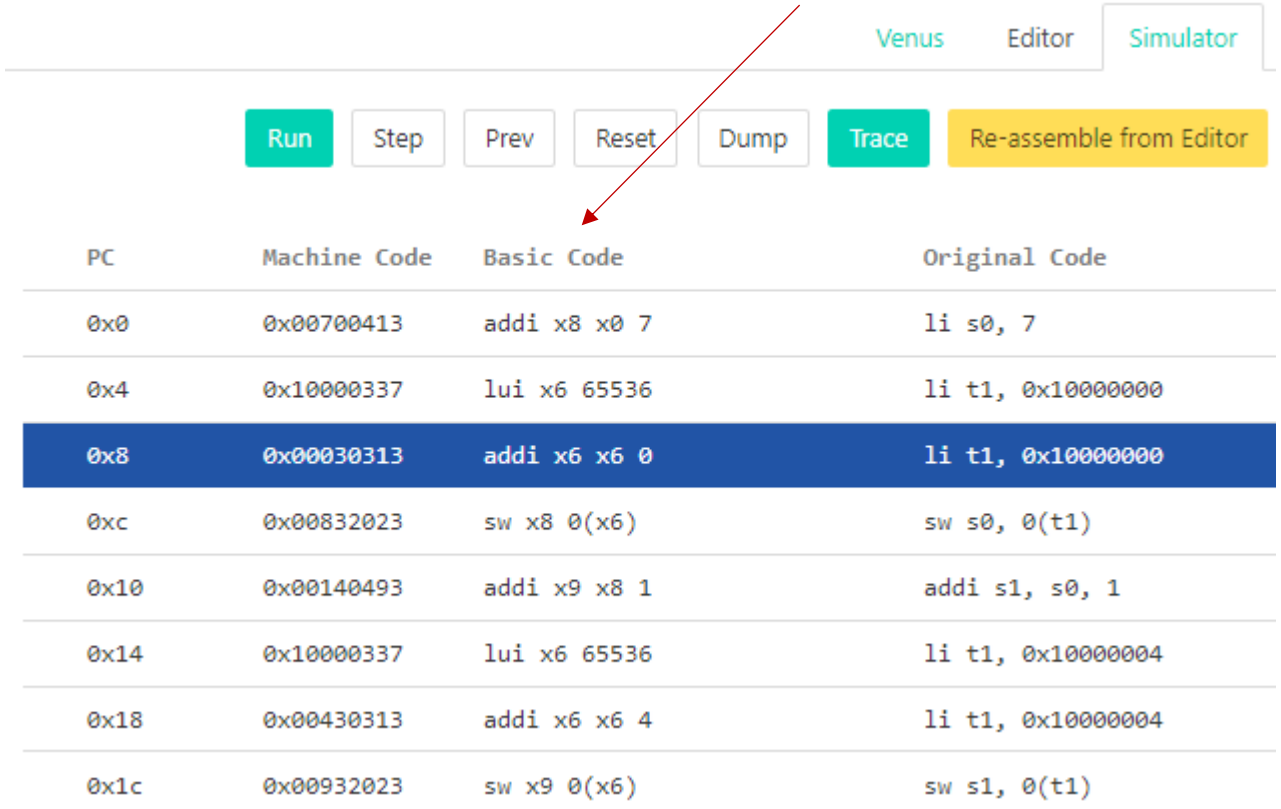


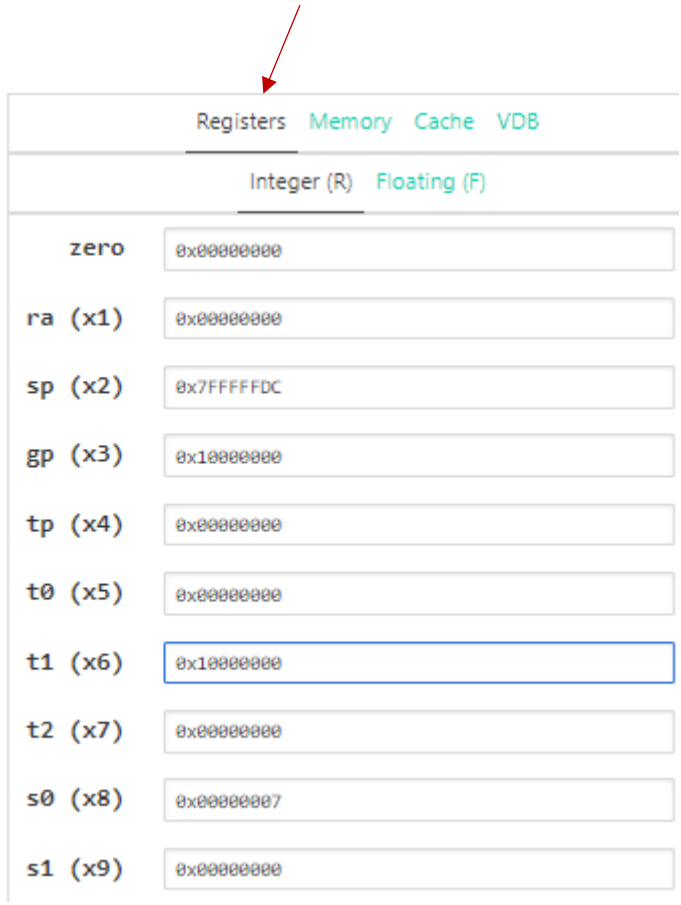

```
s0 = 7;       //shrani v RAM 0x10000000

s1 = s0 + 1; //shrani v RAM 0x10000004
```

Simulator Venus: https://venus.cs61c.org/

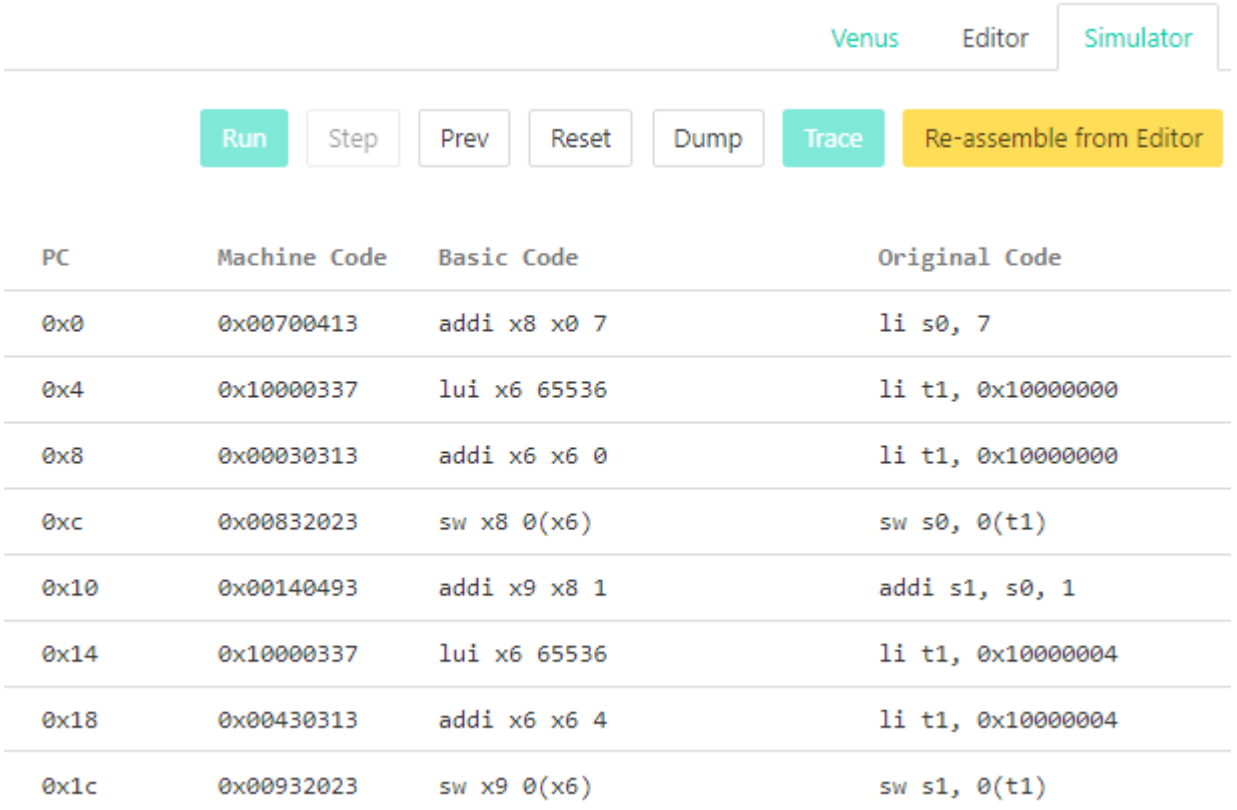
- prevedemo in izvedemo korake simulacije, opazujemo registre
- psevdoukaze prevede v osnovne ukaze



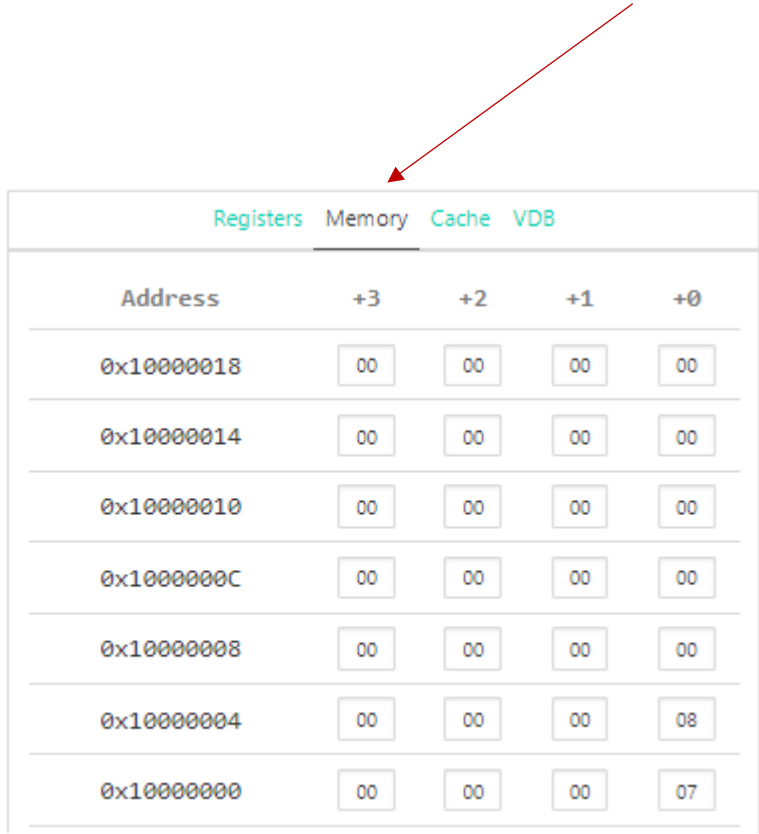


Simulator Venus: https://venus.cs61c.org/

- prevedemo in izvedemo korake simulacije, opazujemo registre in RAM



PC	Machine Code	Basic Code	Original Code
0x0	0x00700413	addi x8 x0 7	li s0, 7
0x4	0x10000337	lui x6 65536	li t1, 0x10000000
0x8	0x00030313	addi x6 x6 0	li t1, 0x10000000
0xc	0x00832023	sw x8 0(x6)	sw s0, 0(t1)
0x10	0x00140493	addi x9 x8 1	addi s1, s0, 1
0x14	0x10000337	lui x6 65536	li t1, 0x10000004
0x18	0x00430313	addi x6 x6 4	li t1, 0x10000004
0x1c	0x00932023	sw x9 0(x6)	sw s1, 0(t1)



Address	+3	+2	+1	+0
0x10000018	00	00	00	00
0x10000014	00	00	00	00
0x10000010	00	00	00	00
0x1000000C	00	00	00	00
0x10000008	00	00	00	00
0x10000004	00	00	00	08
0x10000000	00	00	00	07

Logične operacije

```
add, sub
addi, lui
and,or,xor
andi,ori..
```

- Logične operacije so podobne oblike kot seštevanje

	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

https://www.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/

Pomikanje

```
add, sub
addi, lui
and,or,xor
andi,ori...
sll,srl,sra
slli,srai..
```

- število bitov za pomik je v spodnjih 5 bitih registra

logični pomik v levo: **`sll`** (shift left logical)

```
sll t0, t1, t2    # t0 = t1 << t2
```

logični pomik v desno: **`srl`** (shift right logical)

```
srl t0, t1, t2    # t0 = t1 >> t2
```

aritmetični pomik v desno z upoštevanjem predznaka: **`sra`**

```
sra t0, t1, t2    # t0 = t1 >>> t2
```

ali pa podano kot konstanta med 0 in 31

```
# pomik v desno za 5 mest t1 >> 5
srai t0, t1, 5
```

Vejitve programa

- skok na drugo mesto v program, parameter je odmik naslova
  - v zbirniku uporabimo oznake, odmik pa izračuna prevajalnik
- vrste vejitev
  - **Pogojna: primerja dva registra in skoči, če je pogoj izpolnjen**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch if less than (`blt`)
    - branch if greater than or equal (`bge`)
  - **Brezpogojna**
    - jump (`j`)

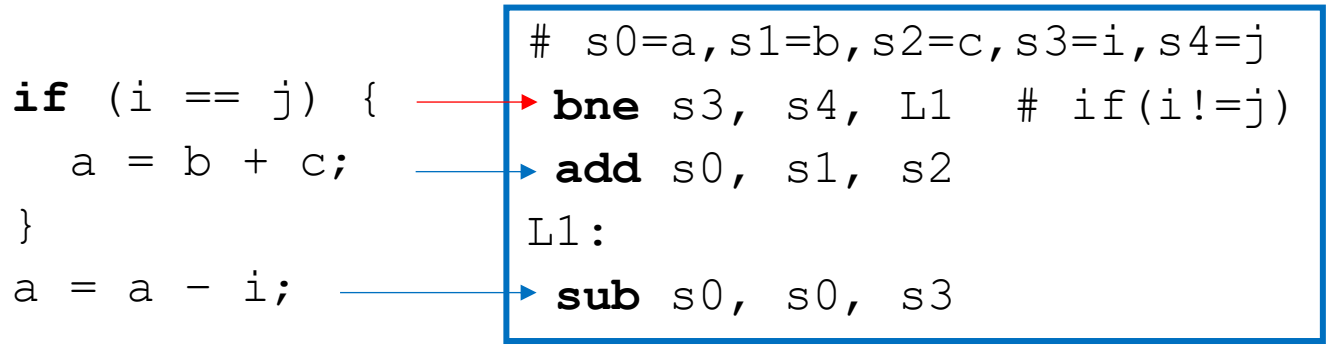
Pogojna vejitev

```
 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, naprej   # skok se izvede
 addi s1, s1, 1        # se ne izvede
 sub  s1, s1, s0       # se ne izvede

naprej:                # oznaka
 add  s1, s1, s0       # s1 = 4 + 4 = 8
```

- oznake lokacije ukaza imajo na koncu dvopičje (:)
- če bi namesto **beq** uporabili **bne**, se skok ne bi izvedel

Pogojni stavek (if)



- zbirnik preverja nasproten pogoj (i != j)

Pogojni stavek (if/else)

- Uporabimo še brezpogojni skok **`j`** (jump)
  - **`j`** je psevdoukaz, strojni ukaz je **`jal x0, odmik`**

```
if (i == j) {
  a = b + c;
} else {
  a = a - i;
}
```

```
# s0=a,s1=b,s2=c,s3=i,s4=j
 bne s3, s4, L1   # if(i!=j)
 add s0, s1, s2
 j done
L1:               # else
 sub s0, s0, s3
done:
```

- visokonivojski in strukturian jezik C je bolj prijazen za programerja!

Zanka while

- zanka se pretvori v zaporedje stavkov

start:
  test pogoja
  če ni izpolnjen, skoči na izhod
  telo zanke
  skok na start
izhod:

```
i = 0;
while (i < 10) {
   i = i + 1;
}
```

```
 addi s3,zero,0   # i = 0
 addi s4,zero,10 #konst 10
start:
 bge  s3,s4,izhod #if(i>=10)
 addi s3,s3,1
 j start
izhod:
```

Zanka for

for (inicializacija; pogoj; operacija) stavek;

```
// vsota števil 0 do 9

int sum = 0;
int i;

for (i=0; i!=10; i++) {
   sum = sum + i;
}
```

```
# s0=i, s1=sum
 addi s1,zero,0      #sum=0
 add  s0,zero,zero   #i=0
 addi t0,zero,10     #konst 10
for:
 beq  s0,t0,done   # if(i==10)
 add  s1,s1,s0
 addi s0,s0,1
 j    for
done:
```

Zanka in zbirka podatkov

Povečaj vse elemente zbirke za 10, zbirka je na naslovu 0x200

```
int array[100];
int i;

for (i=0; i<100; i++) {
 array[i] = array[i]+10;
}
```

```
# s0=array, s1=i
li s0,0x200 # s0=bazni naslov
li s1,0     # i = 0
li t2,100   # t2 = 100
```

naslov array[i] je bazni naslov + i*4

```
loop:
 bge  s1,t2,done   # if(s1>=100)
 slli t0,s1,2      # t0=i*4 (ofset)
 add  t0,t0,s0     # naslov array[i]
 lw   t1,0(t0)     # t1=array[i]
 addi t1,t1,10     # t1=t1+10
 sw   t1,0(t0)     # array[i]=t1
 addi s1,s1,1      # i++
 j    loop
done:
```

Primer: največji skupni delitelj

- funkcija izračuna največji skupni delitelj dveh števil

```
// x >= 0 && y > 0
int gcd(int a, int b) {
 int t;
 while(a != 0) {
  if(a >= b) a = a - b;
  else {
     t = a; a = b; b = t;
  }
 }
 return b;
}
```

```
# a0=a, a1=b, t0=t
gcd:
 beq a0, zero, done    # if(a == 0)
 blt a0, a1, b_bigger # if(a < b)
 sub a0, a0, a1
 j begin               # while
b_bigger:
 mv t0, a0             # t = a
 mv a0, a1             # a = b
 mv a1, t0             # b = t
 j gcd
done:
 add a0, a1, zero      # vrni b
 jr ra
```