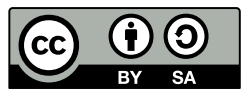
RISC architektūros

Saulius Gražulis

Vilnius, 2020

Vilniaus universitetas, Matematikos ir informatikos fakultetas
Informatikos institutas



x86

Privalumai:

- Suderinamumas per paskutinius 40 metų
- Greitas naujo ir seno kodo vykdymas
- Greičiausios sistemos

x86

Trūkumai:

- Dideli lustai (kristalai)
- Daug tranzistorių – didelė išsklaidoma galia
- Senos funkcijos palaikomos tik dėl suderinamumo
- Funkcijos dubliuojasi
- Daug funkcijų nenaudojamos daugelyje programų
- Daug specializuotų operacijų or registrų – keblumai kompiliatorių konstruktoriams

"AMD's 80x86 architect, Mike Johnson, famously quipped, "The x86 really isn't all that complex—it just doesn't make a lot of sense"

(Waterman 2016)

Argumentai RISC naudai

- “Semiconductor memories are both fast and relatively inexpensive” (Patterson et al. 2003)
- “it is difficult to have ”rational” implementations” (ibid.)
  - IBM 370: Peuto and Shustek have discovered that a sequence of load instructions is faster than a load multiple instruction for fewer than 4 registers; this covers 40% of cases in typical programs (Patterson et al. 2003; Peuto et al. 1977)
  - Patterson found that for VAX 11/780 replacing complex INDEX instruction (calculates address of an array element, checks bounds) by several simple instructions (COMPARE, JUMP LESS UNSIGNED, ADD, MUL) can calculate the same function **45%** faster (in special cases, **60%** faster!) (Patterson et al. 2003)

Argumentai RISC naudai

- “measurements of a particular IBM 360 compiler found that 10 instructions accounted for 80% of all instructions executed, 16 for 90%, 21 for 95%, and 30 for 99%”[1] (Patterson et al. 2003; Alexander et al. 1975).
- “pushing a register on the stack with **PUSHL R0** is slower than pushing it with the move instruction **MOVL R0,-(SP)** on the VAX 11/78” (Patterson et al. 2003)

[1] IBM 360 had 8 bit opcodes, so would have no more than 256 different instructions – with various operands (“IBM System/360 Principles of Operation”, p. 14)

RISC

Techniniai ypatumai:

	RISC	CISC
Instructions	Simple: Load/Store regs, operations **only** in regs	Complex tasks for multiple data types, both in regs and in memory
Instr. formats	Fixed length, two main types: load/store & `R := R op R`	Variable length, may types: load/store, `R := R op R`, `R := R op Mem`, `R := Mem op Mem`
Registers	16–32 general purpose	Specialised or 8–16 general purpose

Adapted from: (Jamil 1995)

RISC

Dizaino principai:

	RISC	CISC
Design objective	Trade off program length, minimise time to execute instruction	Minimise program length, maximise work/instruction
Implementation	Hard-wired	Microprogrammed
Caching	Essential (at least for code)	Useful ($\Rightarrow$ nowadays, essential)
Compiler design	Find best instruction ordering & register allocation	Find best/right instructions
Philosophy	Move all (complicated) functions to software	Move any useful software function into hardware

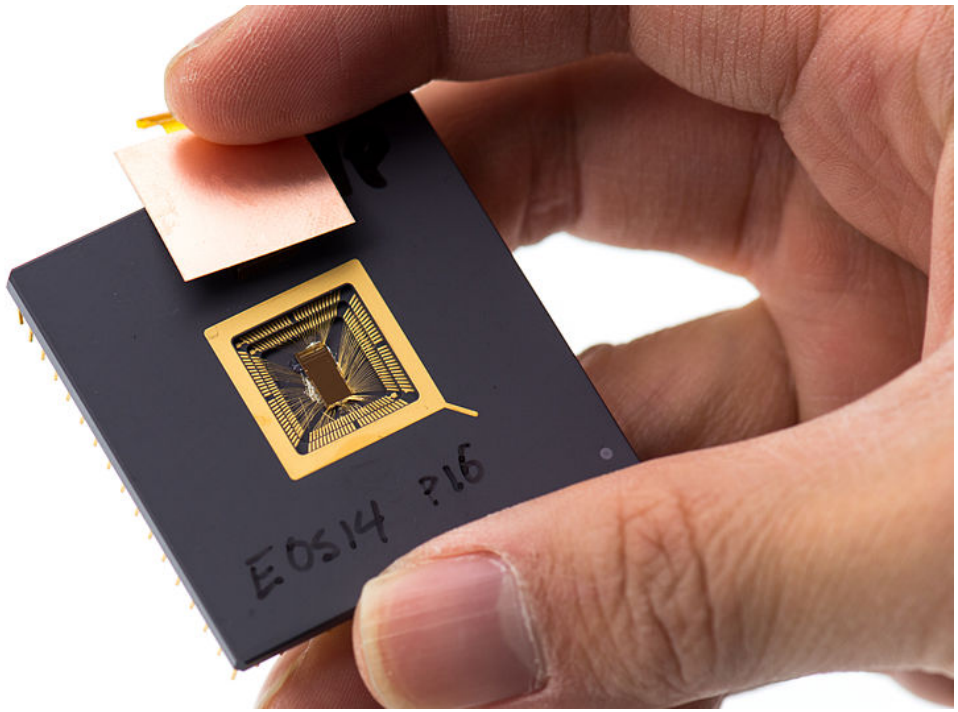
Adapted from: (Jamil 1995)

Kai kurie RISC kompiuteriai

Praeityje...

Kai kurie RISC kompiuteriai

Ir, tikriausiai, ateityje...



- Atvira RISC tipo architektūra :)
- Prieinama atvirų licenzijų pagrindu
- Nereikalauja mokesčių už naudojimą
- Eilė kompanijų gamina arba ruošiasi gaminti aparatūrą (“geležį”).
- Prieinami atviro kodo įrankiai (kompiliatoriai, asembleriai, OS).

RISC-V variantai

- 3 galimi adreso pločiai:
  - RV32
  - RV64
  - RV128
- Daug išplėtimo galimybių:
  - I – base integer ISA
  - M – hardware multiply and divide
  - A – atomic synchronisation support
  - F – single precision floating point
  - D – double precision floating point

RV32G = RV32IMAFD

RISC-V variantai

- 3 galimi adreso pločiai:
  - RV32
  - RV64
  - RV128
- Egzotiškesni išplėtimai:
  - S – Supervisor mode is implemented
  - Q – Quad-precision (128 bit) floating point is supported
  - C – Compressed (i.e., 16 bit) instructions are supported
  - E – Embedded microprocessors, with only 16 registers

RV32G = RV32IMAFD

RISC-V variantai

- 3 galimi adreso pločiai:
  - RV32
  - RV64
  - RV128
- Išplėtimai ateičiai:
  - L – Decimal arithmetic instructions
  - V – Vector arithmetic instructions
  - P – Packed SIMD instructions
  - B – Bit manipulation instructions
  - T – Transactional memory support

RV32G = RV32IMAFD

Atmintis

- 32, 64 arba 128 bitų virtualus adresas
- Jauniausias baitas ties jauniausiu adresu (little-endian)
- Palaikomas neišlygintas priėjimas prie atminties
- Virtuali atmintis: puslapiavimas
- Įvestis/išvestis: atvaizduota į atmintį

RISC-V registrai

- 32 bendros paskirties registrai
- jei palaikomi slankaus kablelio skaičiai, dar 32 slankaus kablelio registrai.
- “it would be possible to define a non-standard subset integer RISC-V ISA with 16 registers” (Waterman et al. 2014)
- Registrų plotis 32, 64 arba 128 bitai (lygus adreso pločiui)

RISC-V registrų rinkinys

31	0
x0/zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	
x14	
x15	

32

31	0
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x29	
x30	
x31	

32

31	0
pc	

32

RV32I Programuotojui matomi registrai (Waterman 2016)

Komandos

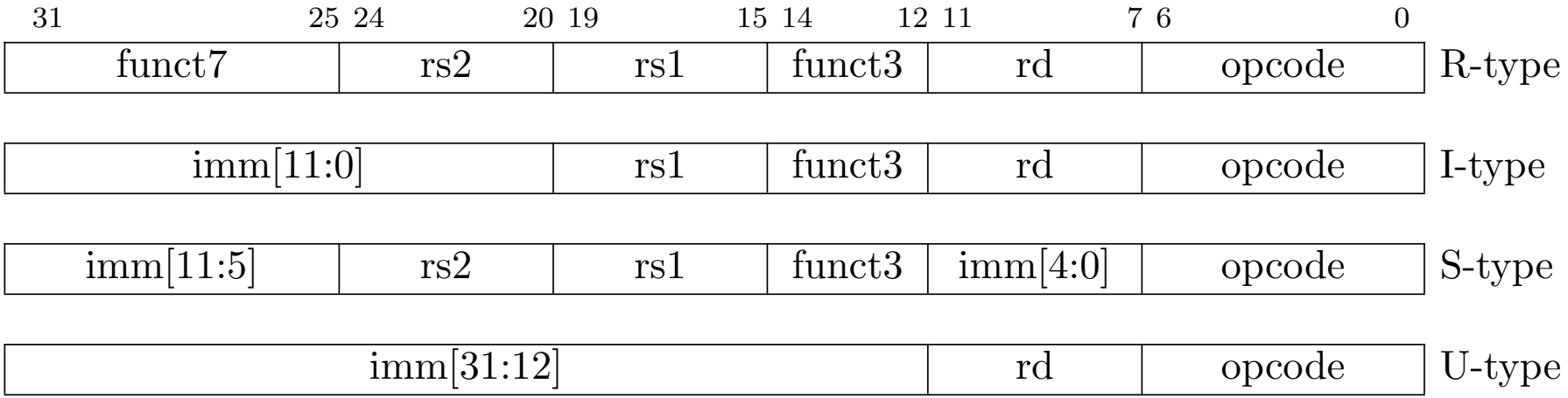
- visos pagrindinės komandos yra 32 bitų pločio
- turi būti ties žodžio riba
- ... bet 16 bitų supakavimas gali ribą susmulkinti iki 16 bitų
- RV64 ir RV128 komandos taip pat 32 bitų
- RISC-V yra trijų adresų mašina:

```
add x4,x5,x7 # x4 = x5 + x7
```

- 16-bit instructions are optionally supported to compress code (the “C” extension), but they are *synonyms* of the standard 32-bit instructions

(Porter 2018)

Komandų formatai



(Waterman et al. 2014)

Komandų tipai

- R-tipo instrukcijos:

```
add x3,x5,x6 # x3 = x5 + x6
```

Komandų tipai

- R-tipo instrukcijos:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-tipo instrukcijos:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

Komandų tipai

- R-tipo instrukcijos:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-tipo instrukcijos:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-tipo instrukcijos:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

Komandų tipai

- R-tipo instrukcijos:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-tipo instrukcijos:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-tipo instrukcijos:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

- B-tipo instrukcijos (S tipo variantas):

```
blt x4,x6,loop # if x4<x6, goto offset + pc
```

Komandų tipai

- R-tipo instrukcijos:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-tipo instrukcijos:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-tipo instrukcijos:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

- B-tipo instrukcijos (S tipo variantas):

```
blt x4,x6,loop # if x4<x6, goto offset + pc
```

- U-tipo instrukcijos:

```
lui x4,0x12AB7 # x4 = value<<12
auipc x4,0x12AB7 # x4 = (value<<12) + pc
```

Komandų tipai

- R-tipo instrukcijos:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-tipo instrukcijos:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-tipo instrukcijos:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

- B-tipo instrukcijos (S tipo variantas):

```
blt x4,x6,loop # if x4<x6, goto offset + pc
```

- U-tipo instrukcijos:

```
lui x4,0x12AB7 # x4 = value<<12
auipc x4,0x12AB7 # x4 = (value<<12) + pc
```

- J-tipo instrukcijos (U tipo variantas):

```
jal # call: pc = offset + pc; x4 = ret add
```

Adresavimo režimai

Tik 2 adresavimo režimai!

- betarpiškas (operandas komandoje)
- netiesioginis, naudojant registrą ir postūmį

Betarpiškos reikšmės:

```
addi x5,x0,21

lui x6,x0,0x1234
addi x6,x0,0x5678 # x6 = 0x12345678
```

Atminties komandos

Instruction	Format	Meaning
`lb rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw rd, imm[11:0](rs1)`	I	Load word
`sb rs2, imm[11:0](rs1)`	S	Store byte
`sh rs2, imm[11:0](rs1)`	S	Store half-word
`sw rs2, imm[11:0](rs1)`	S	Store word
`fence pred, succ`	I	Memory ordering fence
`fence.i`	I	Instruction memory ordering fence

(Waterman 2016)

x0

Registre `x0` visada yra `0`, o rašymas į šį registrą ignoruojamas. Tarpregistrinė `MOV` komanda nebereikalinga!

```
add x6,x7,x0
```

naudojama vietoje

```
mov x6,x7
```

x1

Registras `x1` **pagal susitarimą** naudojamas grįžimo adresui, kviečiant paprogrames

Grįžimo komanda tampa tiesiog `jr x1` (Jump using Register)

Rekursinėms paprogramėms `x1` reikia išsaugoti steke!

MUL

Jaunesnieji sandaugos bitai visada tie patys tiek dauginamiesiems su ženklu, tiek dauginamiesiems be ženklo $\Rightarrow$ tereikalinga viena komanda `MUL` jaunesniajai daliai:

```
mul x4,x9,x13 # x4 = x9*x13
```

MUL

Jaunesnieji sandaugos bitai visada tie patys tiek dauginamiesiems su ženklu, tiek dauginamiesiems be ženklo $\Rightarrow$ tereikalinga viena komanda `MUL` jaunesniajai daliai:

```
mul x4,x9,x13 # x4 = x9*x13
```

Vyresnioji dalis skiriasi operandams su ženklu ir be ženklo, todėl jai reikia 3 komandų:

```
MULH - operandai su ženklu
MULHU - operandai be ženklo
MULHSU - vienas operandas su ženklu, kitas - be ženklo
```

MUL

Jaunesnieji sandaugos bitai visada tie patys tiek dauginamiesiems su ženklu, tiek dauginamiesiems be ženklo $\Rightarrow$ tereikalinga viena komanda `MUL` jaunesniajai daliai:

```
mul x4,x9,x13 # x4 = x9*x13
```

Vyresnioji dalis skiriasi operandams su ženklu ir be ženklo, todėl jai reikia 3 komandų:

```
MULH - operandai su ženklu
MULHU - operandai be ženklo
MULHSU - vienas operandas su ženklu, kitas - be ženklo
```

Rekomenduojama operacijas vykdyti šia tvarka, tada aparatūra galės jas apjungti:

```
mulh x4,x9,x13 # compute upper half
mul  x5,x9,x13 # compute lower half
# The product is now in the register pair x4:x5
```

JMP

Komanda `JMP` *visada* išaugo grįžimo adresą registre (t.y. vienintelė komanda yra `jal`).

Bet, jei šis adresas nereikalingas, kaip registrą galima nurodyti `x0` :)

```
jal x0, loop
```

Stekas. x2

- Steko registras **pagal susitarimą** yra `x2`; stekas auga žemyn 16 baitų porcijomis.
- Kintamieji steke gali būti pasiekiami, naudojant nedidelius teigiamus poslinkius nuo `x2`

GP: x3

- Kvietimo susitarimas: globalūs kintamieji patalpinami viename atminties bloke, ir į šio bloko pradžią rodo `x3`
- `x3` naudojamas pagal susitarimą
- Atskiri kintamieji pasiekiame su nedideliais poslinkiais `x3` atžvilgiu
- Globalių kintamųjų registras inicializuojamas programos pradžioje ir paprastai nekeičiamas

Kiti registrai

- Registras `x4` – gijos vietiniai kintamieji
- Registrus x8,x9,x18-x27 turi išsaugoti *iškviestoji paprogramė* (“s0-s11”)
- Registrai x5-x7,x28-x31 laikomi laikinais ir paprogramės jų neišsaugo (“t0-t6”)
- Registrai x10-x17 naudojami argumentams perduoti (“a0-a7”) [2]

[2] Slankaus kablelio parametrai perduodami slankaus kablelio registruose `fn`, jei šie yra.

Pavyzdys: daugelio žodžių sudėtis

```
li t0,0x12345678
li t1,0xFFEDCBA9
li t2,0x1F
li t3,0x44
add t5,t1,t0
sltu t4,t5,t0 # Determine the carry bit.
add t6,t2,t3
add t6,t6,t4 # Add the carry bit
# t6 now contains the double-machine-word (64 bit) sum.
# Exit to the caller:
li a0,0
li a7,93
ecall
ebreak
```

Pavyzdys: daugelio žodžių sudėtis

```
li t0,0x12345678
li t1,0xFFEDCBA9
li t2,0x1F
li t3,0x44
add t5,t1,t0
add t6,t2,t3
bgeu t5,t0,nocarry # Determine the carry bit.
addi t6,t6,1 # Add the carry bit
nocarry:
# t6 now contains the double-machine-word (64 bit) sum.
# Exit to the caller:
li a0,0
li a7,93
ecall
ebreak
```

Šaltiniai

Alexander, W. C. et al. (1975). “Static and Dynamic characteristics of XPL programs”. In: *Computer* 8.11, pp. 41–46. DOI: 10.1109/c-m.1975.218804.

Jamil, T. (1995). “RISC versus CISC”. In: *IEEE Potentials* 14.3, pp. 13–16. DOI: 10.1109/45.464688.

Patterson, David A. et al. (2003). *The case for the reduced instruction set computer*. Tech. rep. Computer Science Division, University of California; Bell Laboratories, Science Research Center, pp. 25–33. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.9623.

Peuto, B. L. et al. (1977). “An instruction timing model of CPU performance”. In: *Fourth Annual Symposium on Computer Architecture*, pp. 165–178. DOI: 10.1145/800255.810667.

Porter III, Harry H. (2018). *RISC-V: an overview of the instruction set architecture*. URL: https://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf.

Waterman, Andrew (Jan. 2016). “Design of the RISC-V instruction set architecture”. PhD thesis. Electrical Engineering and Computer Sciences, University of California at Berkeley, pp. 1–117. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

Waterman, Andrew et al. (2014). *The RISC-V instruction set manual, volume I: user-level ISA, version 2.0*. Tech. rep. UCB/EECS-2014-54. Electrical Engineering and Computer Sciences, University of California at Berkeley, pp. 1–102. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf.