

# **RISC-V**

## **Conjunto de Instruções**

GEX 612 - Organização de Computadores

Prof. Luciano L. Caimi  
[lcaimi@uffs.edu.br](mailto:lcaimi@uffs.edu.br)

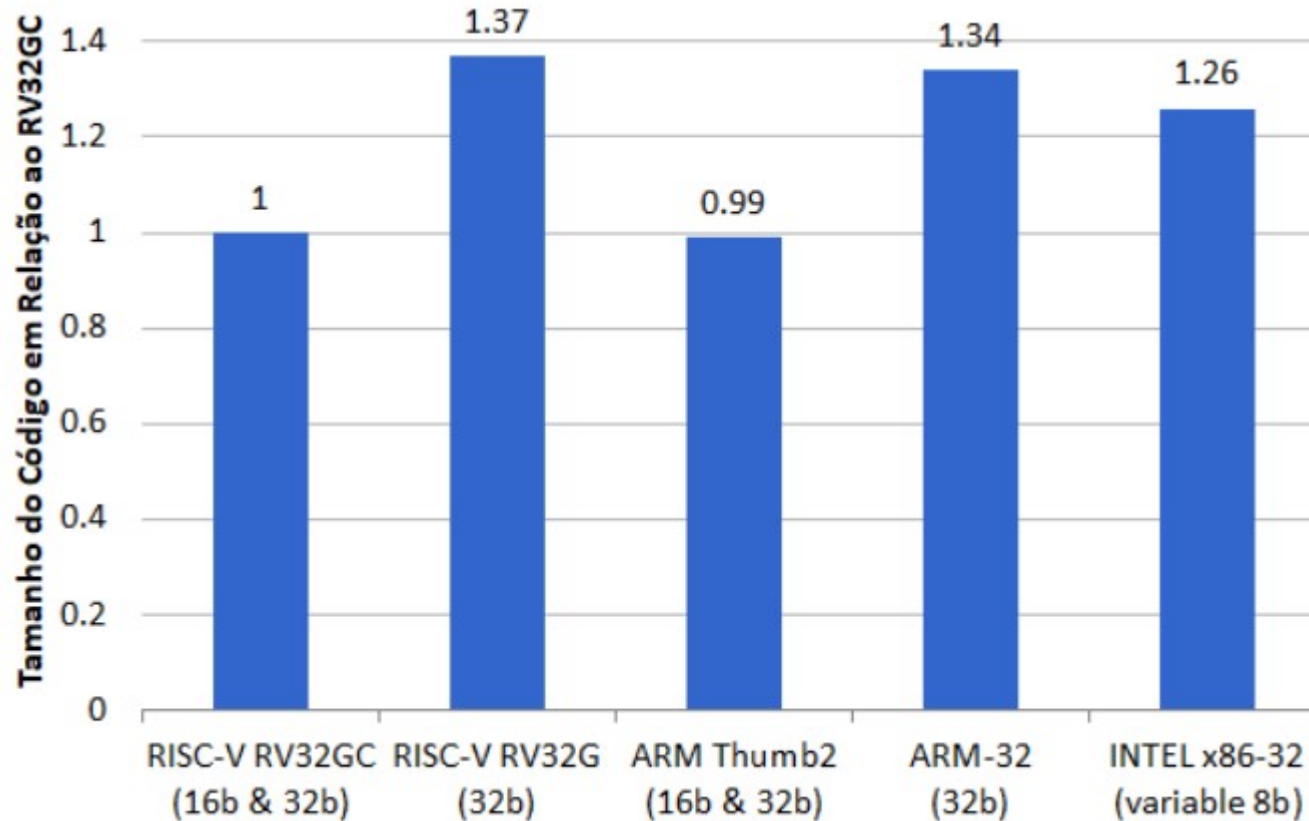
- ISA (Instruction Set Architecture) RISC moderna
  - Introduzida em 2011
- ISA aberta! (uso livre e livre de royalties)
- Funcionalidades e características desenvolvidas com base nos acertos e erros de ISAs que já estão no mercado há mais de 30 anos! (x86, ARM, MIPS)
- Veja figura 2.7 do livro “Guia Prático RISC-V: Atlas de uma arquitetura aberta”

# RISC-V: Introdução



- Mais simples do que ARM e x86

Tamanho relativo de programas do benchmark SPEC CPU2006 compilados com o GCC



Fonte: Livro Guia Prático RISC-V: Atlas de uma arquitetura aberta

# RISC-V: Introdução



- Mantida atualmente pela Fundação RISC-V

- [www.riscv.org](http://www.riscv.org)
- Fundação aberta e sem fins lucrativos
- Mais de 325 empresas parceiras!

>\$50B		>\$5B, <\$50B		>\$0.5B, <\$5B	
Google	USA	BAE Systems	UK	AMD	USA
Huawei	China	MediaTek	Taiwan	Andes Technology	China
IBM	USA	Micron Tech.	USA	C-SKY Microsystems	China
Microsoft	USA	Nvidia	USA	Integrated Device Tech.	USA
Samsung	Korea	NXP Semi.	Netherlands	Mellanox Technology	Israel
		Qualcomm	USA	Microsemi Corp.	USA
		Western Digital	USA		

Alguns dos maiores membros corporativos da Fundação RISC-V  
(VI Workshop RISC-V - 2017/05)

# RISC-V: Introdução



RISC-V 基金会成员已经超过210家



可应用于所有的计算设备的开源、可拓展的指令集



# RISC-V: Introdução



- Alguns links interessantes:
  - <https://en.wikipedia.org/wiki/RISC-V>
  - <https://www.programmersought.com/article/50234370266/>
  - <https://www.zdnet.com/article/risc-v-opens-up-processor-design/>
  - <https://riscv.org/wp-content/uploads/2016/04/RISC-V-Offers-Simple-Modular-ISA.pdf>

- Arquitetura LOAD/STORE
- Memória endereçada à bytes
  - Cada palavra de memória tem 1 byte
  - Tipos de dados maiores do que 1 byte ocupam múltiplas células de memória, consecutivas

Endereço	Célula
0x40	2Bh
0x41	0Ch
0x42	37h
0x43	A3h
0x44	3Fh
0x45	75h
0x46	D7h
0x47	E4h

- Tipos básicos de dados:
  - `byte`: 1 byte
  - `unsigned byte`: 1 byte (sem sinal)
  - `halfword`: 2 bytes
  - `unsigned halfword`: 2 bytes (sem sinal)
  - `word`: 4 bytes
  - `unsigned word`: 4 bytes (sem sinal)
- ISA suporta diretamente estes tipos



- ISA Modular:
  - **RV32I**: Conjunto base de instruções para operações com números inteiros de 32 bits
  - **RV32M**: Instruções de multiplicação e divisão
  - **RV32F** e **RV32D**: Instruções de ponto-flutuante (single e double)
  - **RV32A**: Instruções atômicas
  - **RV32C**: Instruções compactas, de 16 bits
  - **RV32V**: Instruções vetoriais (SIMD)

# RISC-V: Arquitetura

- ISA Modular:

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	An extension to support vector operations
B	An extension to support operations on bit fields
T	An extension to support transactional memory
P	An extension to support packed SIMD instructions
RV128I	A base instruction set providing a 128-bit address space

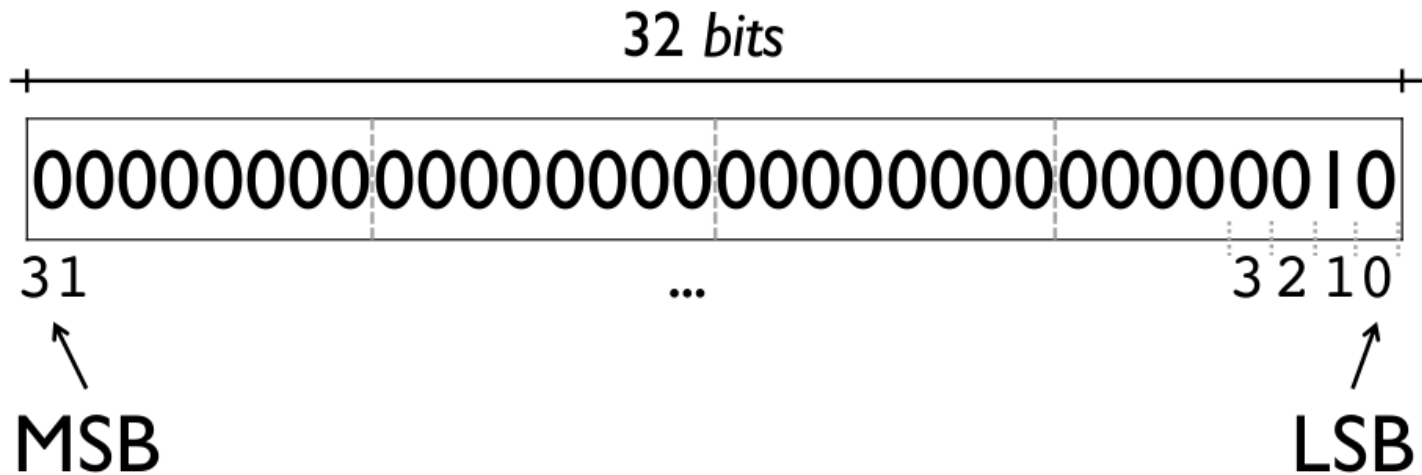


Neste curso focaremos no conjunto **RV32I**:

- Conjunto base de instruções de 32 bits
- Banco de registradores com 32 registradores de 32 bits
- Classes de instruções:
  - Movimentação de dados (load e store)
  - Operações lógicas e aritméticas
  - Comparação de valores e desvios condicionais
  - Desvios incondicionais
  - Chamadas de função

# RISC-V: Arquitetura

Registradores no RISC-V:



## Registradores no RISC-V:

- **PC (Program Counter):** Contador de Programa
- **Banco de Registradores:** 32 registradores

x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15
x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x30	x31

## Registradores no RISC-V:

- Banco de Registradores: **Mnemônicos**

zero	ra	sp	gp	tp	t0	t1	t2	s0	s1	a0	a1	a2	a3	a4	a5
x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15
x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x30	x31
a6	a7	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	t3	t4	t5	t6

# RISC-V: Arquitetura



## Registradores no RISC-V:

Mnemônico	Convenção	Salvo
pc	Contador de Programa	-
a0, a1	Argumentos/Retorno de função	origem
a2-a7	Argumentos de função	origem
s0-s11	Registrador Salvo	destino
t0-t6	Temporário	origem
zero	Contém sempre o valor 0 (zero)	-
ra	Endereço de retorno	origem
sp	Ponteiro de pilha	destino
gp	Ponteiro global	-
tp	Ponteiro de thread	-

# RISC-V: ISA (Instruction Set Architecture)



- Arquitetura **Load/Store**: Os valores têm que ser carregados nos registradores antes de realizar-se operações
- ISA Modular
  - ISA básico padronizado e pequeno
  - Múltiplas extensões padronizadas
- Projetado para Expansão e Especialização
  - Codificação das instruções variável em tamanho
  - Espaço disponível para crescimento de Opcodes/Extensões



# RISC-V: ISA (Instruction Set Architecture)



Base	Descrição	# instruções	Versão	Congelada
RV32I	Conjunto de Instruções com Inteiros - 32 bits	49	2.0	Sim
RV64I	Conjunto de Instruções com Inteiros - 64 bits	14	2.1	Sim
RV128I	Conjunto de Instruções com Inteiros - 128 bits	14	1.7	Não
Extensão	Descrição	# instruções	Versão	Congelada
M	Extensão para Multiplicação e Divisão de Inteiros	8	2.0	Sim
F	Extensão para Ponto Flutuante de Precisão Simples	25	2.2	Sim
D	Extensão para Ponto Flutuante de Precisão Dupla	25	2.2	Sim
V	Extensão para Operações Vetoriais	186	0.2	Não

Fonte: <https://en.wikipedia.org/wiki/RISC-V>

# RISC-V: ISA (Instruction Set Architecture)



## Formato das Instruções:

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1				funct3			rd				opcode								
Immediate	imm[11:0]												rs1				funct3			rd				opcode								
Upper Immediate	imm[31:12]																				rd				opcode							
Store	imm[11:5]							rs2					rs1				funct3			imm[4:0]				opcode								
Branch	[12]	imm[10:5]							rs2					rs1				funct3			imm[4:1]			[11]	opcode							
Jump	[20]	imm[10:1]										[11]	imm[19:12]							rd				opcode								

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):** Destination register specifies register which will receive result of computation

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2				rs1				funct3				rd				opcode								
Immediate	imm[11:0]												rs1				funct3				rd				opcode							
Upper Immediate	imm[31:12]																				rd				opcode							
Store	imm[11:5]							rs2				rs1				funct3				imm[4:0]				opcode								
Branch	[12]	imm[10:5]							rs2				rs1				funct3				imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd				opcode							



32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2				rs1				funct3				rd				opcode								
Immediate	imm[11:0]												rs1				funct3				rd				opcode							
Upper Immediate	imm[31:12]																				rd				opcode							
Store	imm[11:5]							rs2				rs1				funct3				imm[4:0]				opcode								
Branch	[12]	imm[10:5]						rs2				rs1				funct3				imm[4:1]				[11]	opcode							
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd				opcode							



## Formato das Instruções:

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2			rs1		funct3		rd			opcode		R-type	
imm[11:0]						rs1		funct3		rd			opcode		I-type		
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type	
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode	B-type
imm[31:12]										rd			opcode		U-type		
imm[20]		imm[10:1]				imm[11]		imm[19:12]			rd			opcode		J-type	

# RISC-V: ISA

## Instruções RV32I:

Instruction type/opcode	Instruction meaning
<i>Data transfers</i>	<i>Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR</i>
lb, lbu, sb	Load byte, load byte unsigned, store byte (to/from integer registers)
lh, lhu, sh	Load half word, load half word unsigned, store half word (to/from integer registers)
lw, lwu, sw	Load word, store word (to/from integer registers)
ld, sd	Load doubleword, store doubleword
<i>Arithmetic/logical</i>	<i>Operations on data in GPRs.</i>
add, addi, addw, addiw, sub, subi, subw, subiw	Add and subtract, with both word and immediate versions
slt, sltu, slti, sltiu	set-less-than with signed and unsigned, and immediate
and, or, xor, andi, ori, xori	and, or, xor, both register-register and register-immediate
lui	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
auipc	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
sll, srl, sra, slli, srli, srai, sllw, slliw, srli, srlw, srai, sraiw	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
mul, mulw, mulh, mulhsu, mulhu, div, divw, divu, rem, remu, remw, remuw	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
<i>Control</i>	<i>Conditional branches and jumps; PC-relative or through register</i>
beq, bne, blt, bge, bltu, bgeu	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
jal, jalr	Jump and link address relative to a register or the PC

# RISC-V: ISA



# RISC-V: ISA



## Instruções RV32I

imm[31:12]					rd	0110111
imm[31:12]					rd	0010111
imm[20 10:1 11 19:12]					rd	1101111
imm[11:0]			rs1	000	rd	1100111
imm[12 10:5]		rs2	rs1	000	imm[4:1 11]	1100011
imm[12 10:5]		rs2	rs1	001	imm[4:1 11]	1100011
imm[12 10:5]		rs2	rs1	100	imm[4:1 11]	1100011
imm[12 10:5]		rs2	rs1	101	imm[4:1 11]	1100011
imm[12 10:5]		rs2	rs1	110	imm[4:1 11]	1100011
imm[12 10:5]		rs2	rs1	111	imm[4:1 11]	1100011
imm[11:0]			rs1	000	rd	0000011
imm[11:0]			rs1	001	rd	0000011
imm[11:0]			rs1	010	rd	0000011
imm[11:0]			rs1	100	rd	0000011
imm[11:0]			rs1	101	rd	0000011
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011

imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRWR
csr			rs1	010	rd	1110011	CSRWS
csr			rs1	011	rd	1110011	CSRWC
csr			zimm	101	rd	1110011	CSRWI
csr			zimm	110	rd	1110011	CSRWSI
csr			zimm	111	rd	1110011	CSRWC



Example	Instruction	Meaning
add x1,x2,x3	Add	$\text{Regs}[x1] \leftarrow \text{Regs}[x2] + \text{Regs}[x3]$
addi x1,x2,3	Add immediate unsigned	$\text{Regs}[x1] \leftarrow \text{Regs}[x2] + 3$
lui x1,42	Load upper immediate	$\text{Regs}[x1] \leftarrow 0^{32} \# \# 42 \# \# 0^{12}$
sll x1,x2,5	Shift left logical	$\text{Regs}[x1] \leftarrow \text{Regs}[x2] \ll 5$
slt x1,x2,x3	Set less than	$\text{if } (\text{Regs}[x2] < \text{Regs}[x3])$ $\text{Regs}[x1] \leftarrow 1 \text{ else } \text{Regs}[x1] \leftarrow 0$

# RISC-V: ISA



Example	Instruction	Meaning
lw x1,60(x2)	Load word	$\text{Regs}[x1] \leftarrow_{32} \text{Mem}[60 + \text{Regs}[x2]]$
lwu x1,60(x2)	Load word unsigned	$\text{Regs}[x1] \leftarrow_{32} 0^{32} \text{Mem}[60 + \text{Regs}[x2]]$
lh x1,40(x3)	Load half word	$\text{Regs}[x1] \leftarrow_{16} (\text{Mem}[40 + \text{Regs}[x3]])$
lhu x1,40(x3)	Load half word unsigned	$\text{Regs}[x1] \leftarrow_{16} 0^{32} \text{Mem}[40 + \text{Regs}[x3]]$
lb x1,40(x3)	Load byte	$\text{Regs}[x1] \leftarrow_8 (\text{Mem}[40 + \text{Regs}[x3]])$
lbu x1,40(x3)	Load byte unsigned	$\text{Regs}[x1] \leftarrow_8 0^{32} \text{Mem}[40 + \text{Regs}[x3]]$
sw x3,500(x4)	Store word	$\text{Mem}[500 + \text{Regs}[x4]] \leftarrow_{32} \text{Regs}[x3]$
sh x3,502(x2)	Store half	$\text{Mem}[502 + \text{Regs}[x2]] \leftarrow_{16} \text{Regs}[x3]$
sb x2,41(x3)	Store byte	$\text{Mem}[41 + \text{Regs}[x3]] \leftarrow_8 \text{Regs}[x2]$

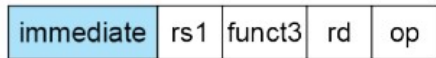
Example	Instruction	Meaning
<code>jal x1,offset</code>	Jump and link	$\text{Regs}[\text{x1}] \leftarrow \text{PC} + 4; \text{PC} \leftarrow \text{PC} + (\text{offset} \ll 1)$
<code>jalr x1,x2,offset</code>	Jump and link register	$\text{Regs}[\text{x1}] \leftarrow \text{PC} + 4; \text{PC} \leftarrow \text{Regs}[\text{x2}] + \text{offset}$
<code>beq x3,x4,offset</code>	Branch equal zero	$\text{if } (\text{Regs}[\text{x3}] == \text{Regs}[\text{x4}]) \text{PC} \leftarrow \text{PC} + (\text{offset} \ll 1)$
<code>bgt x3,x4,name</code>	Branch not equal zero	$\text{if } (\text{Regs}[\text{x3}] > \text{Regs}[\text{x4}]) \text{PC} \leftarrow \text{PC} + (\text{offset} \ll 1)$



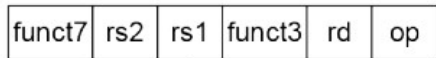
# RISC-V: modos de endereçamento



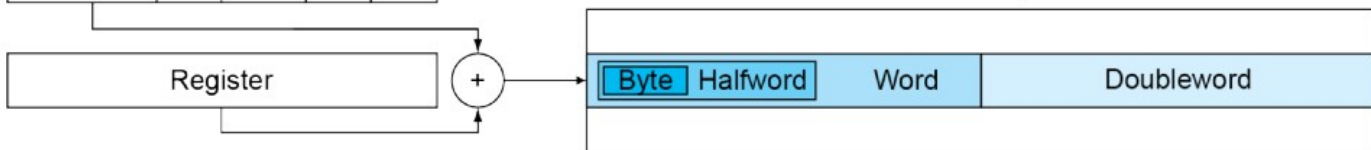
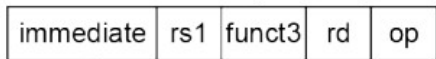
## 1. Immediate addressing



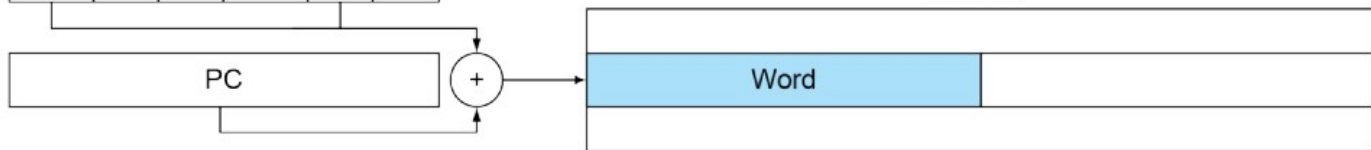
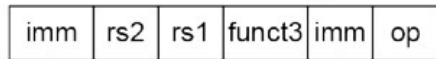
## 2. Register addressing



## 3. Base addressing, i.e., displacement addressing



## 4. PC-relative addressing





# Exemplo 1



pseudo-código:

```
if (x < 10)
    a = a + 1;
else
    a = -1;
```

Mapeamento de registradores:

```
s0 ← x
s1 ← a
```

```
                .data

                .text
main:
    addi t0, zero, 10
    addi t1, zero, 1
if:
    blt s0, t0, true
    addi s1, zero, -1
    j     fim
true:
    add s1, s1, t1

fim:
    nop
```



## Exemplo 2

pseudo-código:

```
a = 100;  
for(x = 0; x < 10; x++)  
    a = a + x    ;
```

Mapeamento de registradores:

```
s0 ← x  
s1 ← a
```

laço for:

- 1) inicializar variáveis
- 2) testar condição de parada
- 3) corpo do laço
- 4) atualizar variável de controle
- 5) voltar para o teste de condição de parada



## Exemplo 2

pseudo-código:

```
a = 100;
for(x = 0; x < 10; x++)
    a = a + x    ;
```

Mapeamento de registradores:

```
s0 ← x
s1 ← a
```

```
.data

.text
main:
    addi s1, zero, 100
inicializa:
    addi t0, zero, 10
    add  s0, zero, zero
condicao:
    beq s0, t0, fim
corpo:
    add s1, s1, s0
atualiza:
    addi s0, s0, 1
    j  condicao
fim:
    nop
```



## Exemplo 3



Descrição: comparar os valores presentes em cada elemento dos vetores A e B e colocar, respectivamente, 1, 0 ou -1 se valor de A for maior, igual ou menor que o valor de B, no mesmo elemento do vetor C (vetores possuem 5 elementos cada).

Pseudo-código:

```
for(x = 0; x < 5; x++) {  
    if(vector_A[x] == vector_B[x])  
        vector_C[x] = 0;  
    else if(vector_A[x] > vector_B[x])  
        vector_C[x] = 1;  
    else  
        vector_C[x] = -1;  
}
```

# Exemplo 3



```
.data
vector_A:    .word    -5, 4, 2, -11, 9
vector_B:    .word    9, 4, -2, 3, 6
vector_C:    .space   20
vector_len:  .word    5
```

```
.text
main:
    la    a0, vector_A
    la    a1, vector_B
    la    a2, vector_C
    lw    t0, vector_len

init:
    addi   t1, zero, 0

cond:    bge t1, t0, end_loop
    lw    s0, 0(a0)
    lw    s1, (a1)

if_equal:
    beq s0, s1, equal
```

```
if_A_greater:
    bgt s0, s1, A_greater

A_minor:
    addi   t2, zero, -1
    sw    t2, (a2)

update_control:
    addi a0, a0, 4
    addi a1, a1, 4
    addi a2, a2, 4
    addi t1, t1, 1
    j      cond

equal:
    sw    zero, (a2)
    j      update_control

A_greater:
    addi   t2, zero, 1
    sw    t2, (a2)
    j      update_control

end_loop:
    nop
```



## Exemplo 4



Descrição: Fazer uma **função** que recebe o endereço inicial de um vetor e dois índices do mesmo e realiza a troca dos valores presentes nos índices.

Pseudo-código:

```
void swap_vector(int &vector[0], int indice_1, int indice_2){  
    int aux_1, aux_2;  
  
    aux_1 = vector[indice_1];  
    aux_2 = vector[indice_2];  
  
    vector[indice_1] = aux_2;  
    vector[indice_2] = aux_1;  
}
```



## Exemplo 4



Descrição: Fazer um programa com um vetor e dois índices do mesmo e realiza a troca dos valores presentes nos índices.

Pseudo-código:

Main

```
int vector[10], int indice_1, int indice;  
int aux_1, aux_2;
```

```
aux_1 = vector[indice_1];
```

```
aux_2 = vector[indice_2];
```

```
vector[indice_1] = aux_2;
```

```
vector[indice_2] = aux_1;
```

```
}
```

# Exemplo 4



```
vector:      .data
            .word    -5, 4, 2, -11, 9
```

```
main:      .text
            la    a0, vector
            li    a1, 0
            li    a2, 1
            jal   swap_vector

            la    a0, vector
            li    a1, 3
            li    a2, 0
            jal   swap_vector

end:
            nop
            li    a7, 93
            ecall
```

```
#####
# função: swap_vector
# entrada: a0 = posicao inicial do vetor
#          a1 - indice 1
#          a2 - indice 2
# saída: não retorna nenhum valor
#####
swap_vector:
            slli  a1, a1, 2
            slli  a2, a2, 2
            add   t0, a0, a1
            lw    a3, 0(t0)
            add   t1, a0, a2
            lw    a4, 0(t1)
            sw    a3, 0(t1)
            sw    a4, 0(t0)
            ret
```

Implementar um programa usando o assembly do RISC-V que contém pelo menos 3 funções:

- void preenche\_vetor(int &vetor[0], int num\_elementos);

Esta função recebe o endereço inicial de um vetor e número de elementos do mesmo e preenche o mesmo com valores lidos do teclado;

- void imprime\_vetor(int &vetor[0], int num\_elementos);

Esta função recebe o endereço inicial de um vetor e número de elementos do mesmo e imprime o conteúdo do vetor no console;

- void ordena\_vetor(int &vetor[0], int num\_elementos);

Esta função recebe o endereço inicial de um vetor e número de elementos do mesmo e ordena o elementos do vetor em ordem decrescente (do maior para o menor);

O programa deve:

- a) preencher o vetor com 10 valores;
- b) imprimir o vetor preenchido;
- c) ordenar o vetor;
- d) imprimir o vetor ordenado.

# Instruções de formato R



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2				rs1				funct3			rd				opcode									
0000000	rs2							rs1				000				rd			0110011				ADD									
0100000	rs2							rs1				000				rd			0110011				SUB									
0000000	rs2							rs1				001				rd			0110011				SLL									
0000000	rs2							rs1				010				rd			0110011				SLT									
0000000	rs2							rs1				011				rd			0110011				SLTU									
0000000	rs2							rs1				100				rd			0110011				XOR									
0000000	rs2							rs1				101				rd			0110011				SRL									
0100000	rs2							rs1				101				rd			0110011				SRA									
0000000	rs2							rs1				110				rd			0110011				OR									
0000000	rs2							rs1				111				rd			0110011				AND									

**<MNE> rd, rs1, rs2 # reg[rd] ← reg[rs1] MNE reg[rs2]**



# Instruções de formato I



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1				funct3			rd				opcode								

## Aritméticas:

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI

**<MNEi> rd, rs1, imm # reg[rd] ← reg[rs1] MNEi imm**

# Exemplo



$$S = (A - B) + (C - 5)$$

```
.text
```

```
main:
```

```
    sub t0, s0, s1  
    addi t1, s2, -5  
    add s3, t0, t1
```

## Mapeamento de registradores:

$s0 \leftarrow A$

$s1 \leftarrow B$

$s2 \leftarrow C$

$s3 \leftarrow S$





# Instruções de formato I



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1			funct3			rd			opcode										

## Load:

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

**<MNE> rd, imm (rs1) # reg[rd] ← (reg[rs1] + imm)**

## Exemplo

$$S = (A - B) + (C - 5)$$

```
.data
A: .space 4
B: .space 4
C: .space 4
S: .space 4
```

```
.text
main:
```

```
la    t0, A
la    t1, B
la    t2, C
la    t3, S
lw    s0, 0 (t0)
lw    s1, 0 (t1)
lw    s2, 0 (t2)
sub    a0, s0, s1
addi   a1, s2, -5
add    s3, a0, a1
sw     s3, 0 (t3)
```

### Mapeamento de registradores:

```
s0 ← A
s1 ← B
s2 ← C
s3 ← S
```

# Exemplo



$$S = A[3] + B[7]$$

```
.data
vetorA: .space 40
vetorB: .space 40
```

```
.text
main:
    la s0, vetorA
    la s1, vetorB
    lw t0, 12 (s0)
    lw t1, 28 (s1)
    add t2, t0, t1
```

**Mapeamento de registradores:**

$t2 \leftarrow S$



# Instruções de formato S



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store	imm[11:5]							rs2				rs1				funct3			imm[4:0]					opcode								

## Store:

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

**<MNE> rs2, imm (rs1) # (reg[rs1] + imm) ← reg[rs2]**

# Exemplo



$$C[5] = A[3] + B[7]$$

```
.data
vetorA:.space 40
vetorB:.space 40
vetorC:.space 40

.text
main:
    la s0, vetorA
    la s1, vetorB
    la s2, vetorC
    lw t0, 12 (s0)
    lw t1, 28 (s1)
    add t2, t0, t1
    sw t2, 20 (s2)
```



# Instruções de formato B



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Branch	[12]	imm[10:5]							rs2				rs1				funct3				imm[4:1]				[11]	opcode						

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

**<MNE> rs1, rs2, imm # if (reg[rs1] <MNE> reg[rs2])**

**PC ← PC + imm**

**else**

**PC ← PC + 4**



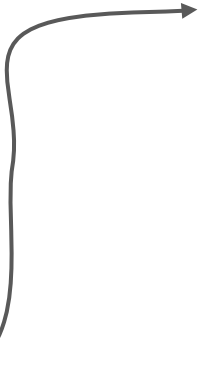
# Exemplo

```
.data
vetorA:.space 40
vetorB:.space 40
vetorC:.space 40

.text
main:
    la s0, vetorA
    la s1, vetorB
    la s2, vetorC
    lw t0, 12(s0)
    lw t1, 28(s1)
    blt t0, t1, if
else:
    sw t1, 20(s2)
    j fim
```

```
if(A[3] < B[7])
    C[5] = A[3]
else
    C[5] = B[7]
```

```
if:
    sw t0, 20(s2)
fim: nop
```

A curved arrow originates from the 'j fim' instruction in the assembly code and points to the 'if:' label in the assembly code, indicating a jump to the end of the block.

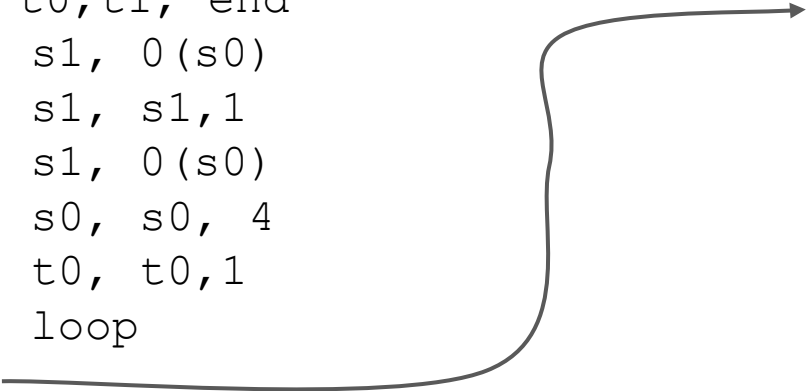


# Exemplo

```
.data
vetorA:    .space 40
           .text
main:
    la  s0, vetorA
    li  t0, 0  # add t0, zero, zero
    li  t1, 10 # addi t1, zero, 10
loop:
    beq t0, t1, end
    lw  s1, 0(s0)
    addi s1, s1, 1
    sw  s1, 0(s0)
    addi s0, s0, 4
    addi t0, t0, 1
    j   loop
```

```
int vetorA[10];
for(x=0; x<10; x++){
    vetorA[x] += 1;
}
```

end:  
nop

A curved arrow originates from the 'j loop' instruction in the assembly code and points to the 'end:' label in the assembly code, indicating a jump to the end of the loop.

# Instruções de formato J



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd				opcode							

**JAL rd, imm # reg[rd]  $\leftarrow$  PC + 4**  
**PC  $\leftarrow$  PC[31-20] : imm[1-20]**

## Jump and Link (JAL)

instrução salva o retorno no registrador de destino (normalmente ra);

Desvia em relação ao valor atual de PC

O valor imm (constante) é calculado a partir da 'distância' entre a instrução atual e o rótulo do programa



# APENDICE 01:

## Características ISA RISC-V

Fonte: Livro Guia Prático RISC-V: Atlas de uma arquitetura aberta

**UFFS - Universidade Federal da Fronteira Sul - Organização de Computadores**

# APENDICE: características ISA RISC-V



	Erros do passado			Lições aprendidas RV32I (2011)
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	
Custo	Multiplicação de inteiros mandatória	Multiplicação e divisão de inteiros mandatória	Operações de 8-bit e 16-bit. Multiplicação e divisão de inteiros mandatóriaInteger	Sem operações de 8-bit e 16-bit operations. Multiplicação e divisão de inteiros opcional (RV32M)
Simplicidade	Sem registrador zero. Execução de instrução condicional. Modos de endereço de dados complexos. Instruções de pilha (push/pop). Opção de deslocamento para instruções aritméticas e lógicas	Imediatos zero e de sinal estendido. Algumas instruções aritméticas podem causar armadilhas de overflow	Sem registrador zero. Instruções de chamada/retorno de procedimento complexo (Enter/Leave). Instruções de pilha (push/pop). Modos de endereço de dados complexos. Instruções de laço	Registrador x0 dedicado ao valor 0. Imediatos somente com sinal estendido. Modo de endereçamento de dados. Nenhuma execução condicional. Nenhuma instrução complexa de desvio/retorno. Nenhuma trap para overflow aritmético. Instruções de deslocamento separadas

Fonte: Livro Guia Prático RISC-V: Atlas de uma arquitetura aberta

**UFFS - Universidade Federal da Fronteira Sul - Organização de Computadores**

	ARM-32 (1986)	Erros do passado MIPS-32 (1986)	x86-32 (1978)	Lições aprendidas RV32I (2011)
Desempenho	Códigos de condição para desvios. Os reg. de origem e destino variam em formato de instrução. Load múltiplo. Imediatos computados. PC de propósito geral	Os registradores de origem e destino variam em formato de instrução.	Códigos de condição para desvios. No máximo 2 registradores por instrução	Instruções de compara e desvia. 3 registradores por instrução. Sem load múltiplo. Registradores de origem e destino fixados em formato de instrução. Imediatos constantes. PC não é um reg. de propósito geral
Isola a arquitetura da implementação	Expõe o comprimento do pipeline ao gravar o PC como um registrador de propósito geral	Desvio atrasado. Load atrasado. Registradores HI e LO utilizados apenas para multiplicação e divis	Registradores sem finalidade geral (AX, CX, DX, DI, SI tem usos exclusivos)	Sem desvio atrasado. Sem load atrasado. Registradores de propósito geral
Espaço para crescimento	Espaço disponível do opcode limitado	Espaço disponível do opcode limitado		Espaço de opcode disponível generoso

Fonte: Livro Guia Prático RISC-V: Atlas de uma arquitetura aberta



	Erros do passado			Lições aprendidas
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
Tamanho do programa	Apenas instruções de 32 bits (+Thumb-2 como ISA separada)	Apenas instruções de 32 bits (+microMIPS como ISA separada)	Instruções variáveis de byte, mas com más escolhas	Instruções de 32 bits + extensão RV32C de 16 bits
Facilidade de programação / compilação / linkagem	Apenas 15 registradores. Dados alinhados na memória. Modos de endereço de dados irregulares. Contadores de desempenho inconsistentes	Dados alinhados na memória. Contadores de desempenho inconsistentes	Apenas 8 registradores. Nenhum endereçamento de dados relativos ao PC. Contadores de desempenho inconsistentes	31 registradores. Os dados podem ficar desalinhados. Endereçamento de dados relativos ao PC. Modo de endereço de dados simétrico. PC contadores de desempenho definidos na arquitetura

Fonte: Livro Guia Prático RISC-V: Atlas de uma arquitetura aberta

## Verifica o maior valor entre 3 valores



```
.data

.text
main:
    addi s0, zero, 3
    addi s1, zero, 7
    addi s2, zero, 4

    teste_01:
        bgt s0, s1, s0_maior_s1
    teste_02:
        bgt s1, s2, s1_maior_de_todos

s2_maior_de_todos:
    add a0, zero, s2
    j    fim

s0_maior_s1:
    bgt s0, s2, s0_maior_de_todos
    j s2_maior_de_todos

s0_maior_de_todos:
    add a0, zero, s0
    j    fim
```

s1\_maior\_de\_todos:  
add a0, zero, s1

fim:  
nop