

cin.ufpe.br



UNIVERSIDADE FEDERAL DE PERNAMBUCO

Infra-estrutura Hardware

Infra-estrutura de Hardware

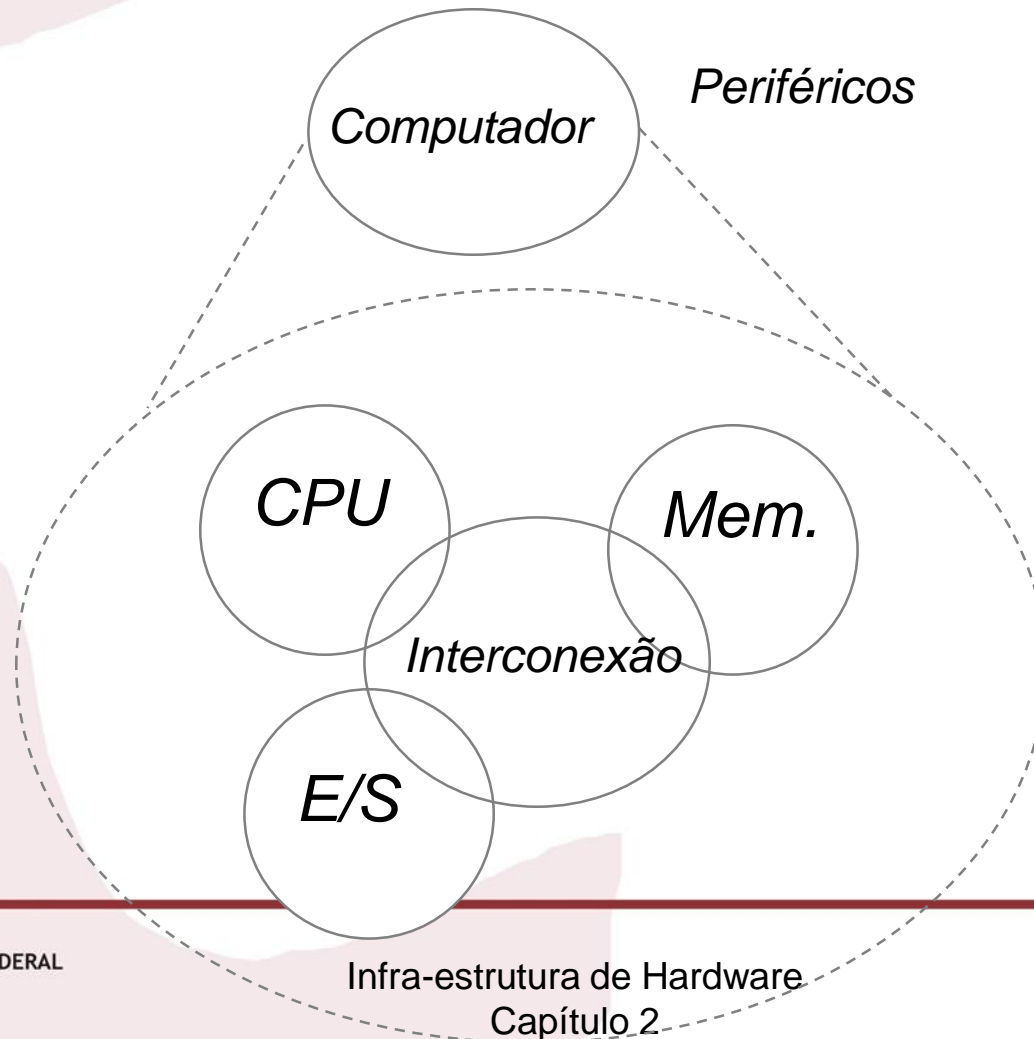
ARQUITETURA DO PROCESSADOR MIPS/ RISC V

Roteiro da Aula

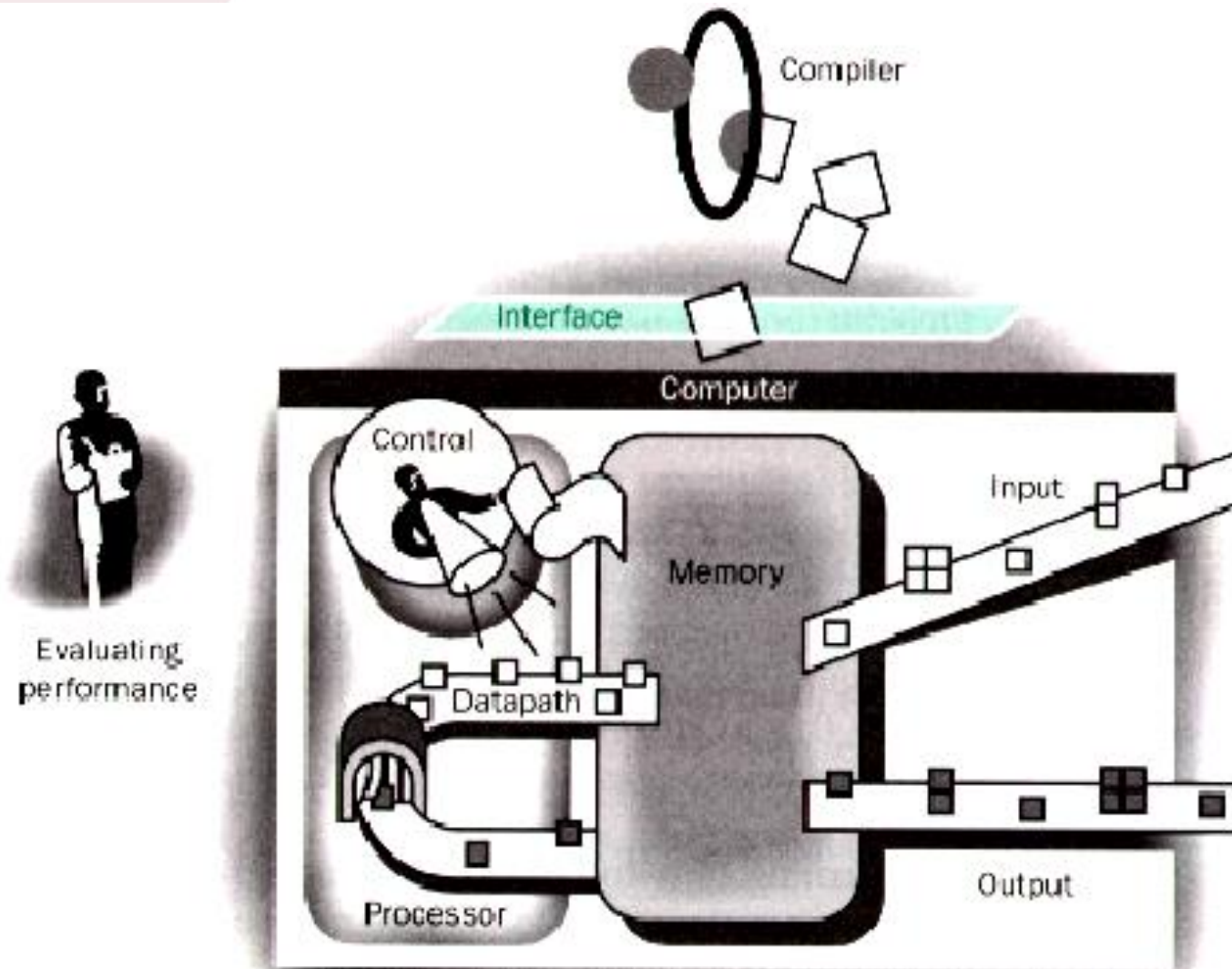


- Introdução
- Operações Aritméticas
 - Representação dos operandos
 - Uso de registradores
- Representando as instruções
- Mais operações sobre dados
- Desvios condicionais
- Funções
 - MIPS
 - Outros processadores
- Modos de Endereçamento
 - MIPS
 - Outros processadores

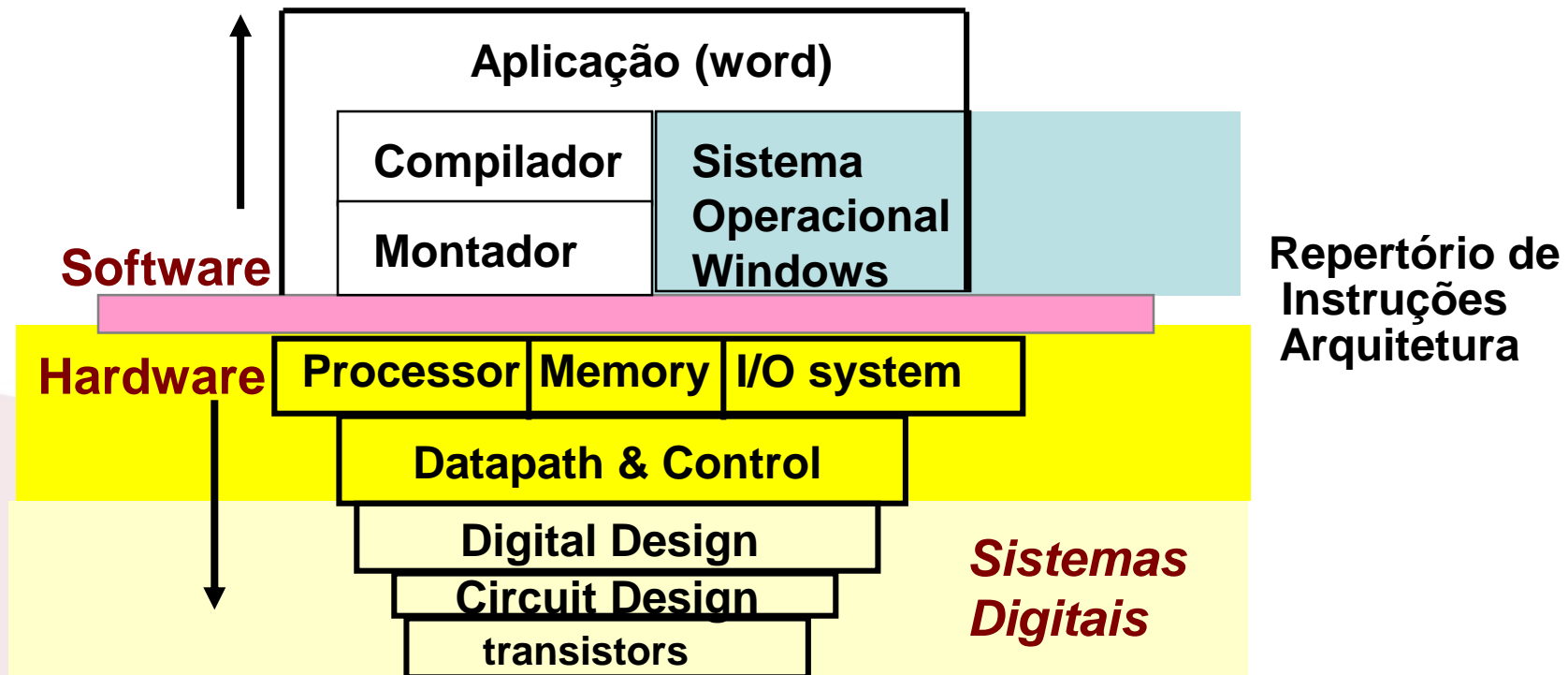
Componentes de um Computador: Hardware



Computador: Hardware + Software



Computador: Hardware + Software



Conceitos Básicos de Arquitetura de Computadores

Capítulo 2

Representação da Informação



Programa em
Linguagem de alto
nível (e.g., C)

Compilador

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Programa em linguagem
assembly (e.g., MIPS)

Montador

```
lw $to,    0($2)  
lw $t1,    4($2)  
sw $t1,    0($2)  
sw $t0,    4($2)
```

Programa em
linguagem de
Máquina (MIPS)

Hardware

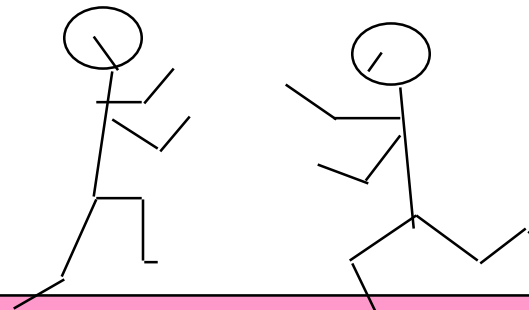
```
1000 1100 0100 1000 0000 0000 0000 0000  
1000 1100 0100 1001 0000 0000 0000 0100  
1010 1100 0100 1001 0000 0000 0000 0000  
1010 1100 0100 1000 0000 0000 0000 0100
```



Interface entre hw e sw: Repertório de Instruções:

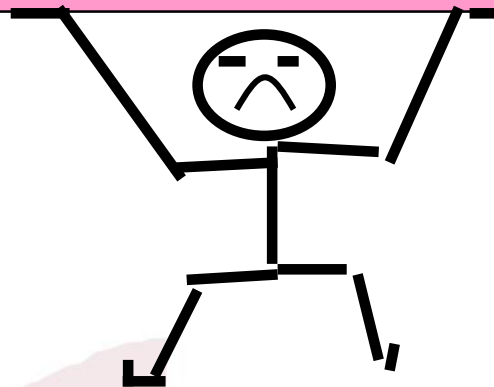


software



Repertório de Instruções

hardware

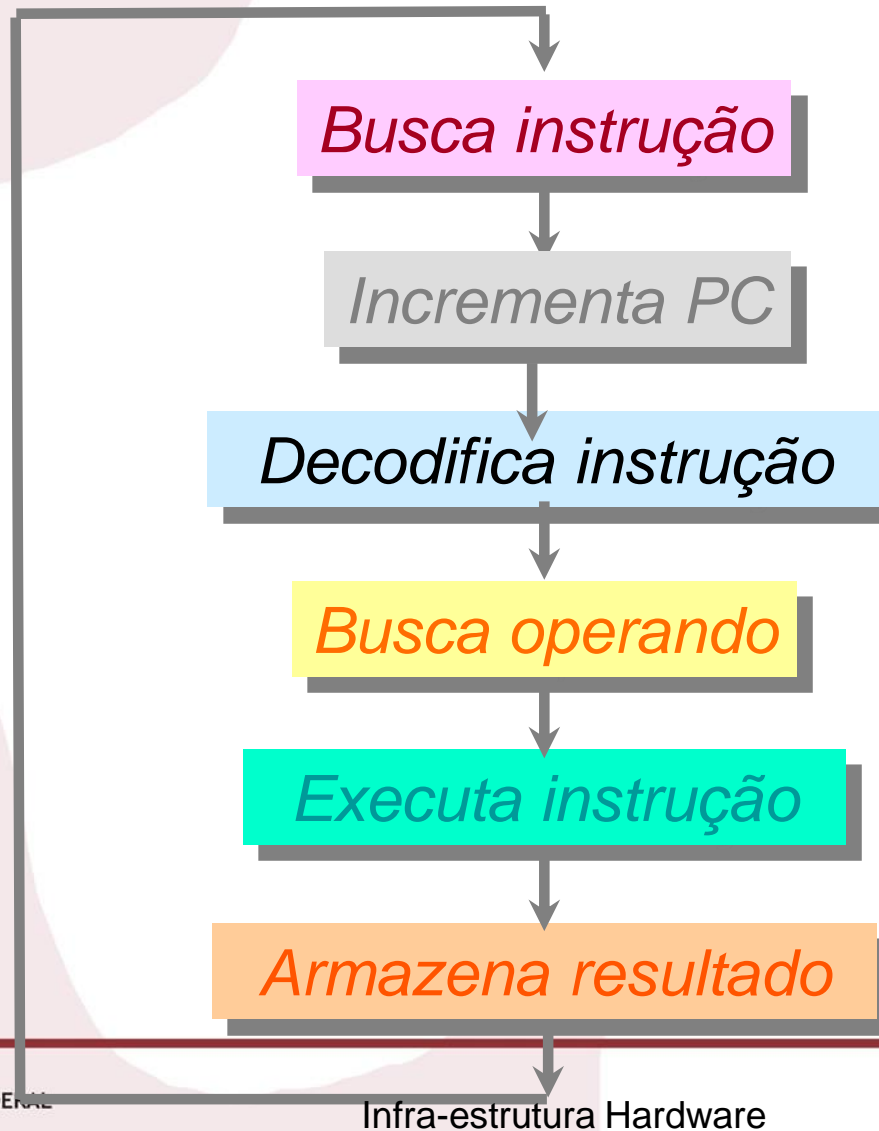


UNIVERSIDADE FEDERAL
DE PERNAMBUCO

Infra-estrutura Hardware

cin.ufpe.br

Executando um programa



QUAIS INSTRUÇÕES QUE UM PROCESSADOR EXECUTA?

Conjunto de Instruções (ISA)



- O repertório de instruções de um computador
- Diferentes computadores têm diferentes conjuntos de instruções
 - Mas com muitos aspectos em comum
- Os primeiros computadores tinham conjuntos de instruções muito simples
 - Implementação simplificada
- Muitos computadores modernos também possuem conjuntos de instruções simples



O ISA do RISC-V



- Usado como o exemplo ao longo do livro...
- Desenvolvido na UC Berkeley como ISA aberto
- Agora gerenciado pela Fundação RISC-V (riscv.org)
- Típico de muitos ISAs modernos
- ISAs similares têm uma grande parcela do mercado principal incorporado
 - Aplicações em eletrônicos de consumo, equipamentos de rede / armazenamento, câmeras, impressoras, ...



Operações aritméticas



- Aritméticas

- add a,b,c

$$a = b + c$$

- $a = b + c + d + e$?

- sub a,b,c

$$a = b - c$$

*Todas as instruções aritméticas possuem 3 endereços:
destino, fonte 1, fonte 2*

A simplicidade é favorecida pela regularidade



Expressões Aritméticas



- $f = (g + h) - (i + j)$
 - Variáveis auxiliares: $t0$ e $t1$
 - add $t0, g, h$
 - add $t1, i, j$
 - sub $f, t0, t1$



Operandos no Hardware



- Para se garantir o desempenho....
- Operandos em registradores
- Vantagens:
 - leitura e escrita em registradores são muito mais rápidas que em memória
- Princípio de Projeto 2: Menor é mais rápido
 - memória principal: milhões de locais ...



Operandos em Registradores



- Instruções aritméticas usam operandos de registro
- MIPS possui 32 registradores de 32 bits cada
 - Os dados de 32 bits são chamados de "palavra"
- RISC-V possui 32 registradores de 64 bits cada
 - Use para dados com acesso frequente
 - Os dados de 64 bits são chamados de "palavra dupla"
 - 32 registradores de uso geral de 64 bits: x0 a x30



RISC-V Registradores



x0: o valor constante 0

x1: endereço de retorno

x2: apontador de pilha

x3: ponteiro global

x4: ponteiro de linha

x5 - x7, x28 - x31: temporários

x8: ponteiro do quadro

x9, x18 - x27: registros salvos

x10 - x11: argumentos de função / resultados

x12 - x17: argumentos de função



Operandos no Hardware



- Registradores RISC V
 - x5 - x7, x28 - x31: armazenam variáveis temporárias
 - x9, x18 - x27: armazenam variáveis do programa
- Registradores MIPS
 - \$s0, \$s1, : armazenam variáveis do programa
 - \$t0, \$t1, : armazenam variáveis temporárias
- C code:
 - $f = (g + h) - (i + j)$; f, \dots, j in x19, x20, ..., x23
 - Código RISC-V:
 - add x5, x20, x21
 - add x6, x22, x23
 - sub x19, x5, x6

Operandos na Memória



- Manipulando arrays:
 - Armazenados na memória
- Instruções que permitam transferência de informação entre memória e registrador

Instruções de Transferência de Dados
load double word - ld

Array: endereço inicial de memória
elemento a ser transferido



Operandos na Memória



- Arrays no MIPS/ RISC V:
 - endereço inicial:
 - registrador
 - deslocamento:
 - valor na instrução
- Instrução **L**oad **D**ouble Word:
 - Copia conteúdo de palavra dupla (64bits) de memória para registrador.
 - `ld reg_dst, desl(reg_end_inicial)`



Operandos na Memória



- Memória armazena estruturas de dados
 - Arrays, structures, dynamic data
- Memória armazena variáveis
 - Carrega variáveis para registradores
 - Armazena registradores na memória
- Vários tipos de dados:
 - Inteiros 16, 32 ou 64 bits
 - Caracteres 1 byte
 - Dados lógicos 1 byte

Operandos na Memória



- Melhor uso efetivo:
 - Memória endereçada por byte
- Cada endereço identifica 1 byte
- Como armazenar tipo de dado com vários bytes?
 - RISC-V é Little Endian
 - Byte menos significativo no menor endereço
 - Big Endian: byte mais significativo no maior endereço
 - RISC-V não requer palavras alinhadas
 - Difere de outros ISAs

Operandos na Memória - Endianess

- MIPS/ RISC V
 - Inteiros com 32 bits
 - Memória endereçada por byte
 - Big Endian (MIPS) Little Endian (RISC V – x86)

b_3	b_2	b_1	b_0
-------	-------	-------	-------

b_3	b_2	b_1	b_0
-------	-------	-------	-------

10	b_3
11	b_2
12	b_1
13	b_0
14	
15	
16	
17	

10	b_0
11	b_1
12	b_2
13	b_3
14	
15	
16	
17	

Infra-estrutura Hardware



UNIVERSIDADE FEDERAL
DE PERNAMBUCO

ufpe.br

Operandos na Memória



- MIPS/ RISC V
 - Inteiros com 32 bits
 - Memória endereçada por byte



$$\text{End}(A[0]) = 10$$

$$\text{End}(A[1]) = 14$$

$$\text{End}(A[i]) = \text{End-inicial} + i \times 4$$



Operandos na Memória



- RISC V
 - Inteiros com 64 bits
 - Memória endereçada por byte



$$\text{End}(A[0]) = 10$$

$$\text{End}(A[1]) = 18$$

$$\text{End}(A[i]) = \text{End-inicial} + i \times 8$$



Operandos na Memória



- Escrita em Memória
- Instrução **S** Store **D** Double Word:
 - Cópia conteúdo de palavra (64bits) de registrador para memória.
 - `sd reg_alvo, desl(reg_end_inicial)`
- $A[2] = h + A[8]$
 - endereço inicial de A em x22 e h em x9
 - `ld x5, 64(x22)`
 - `add x5, x9, x5`
 - `sd x5, 16(x22)`



Operandos na Memória



- Array com variável de indexação
- $A[i] = h + A[i]$
 - endereço inicial de A em x5 e h e i estão em x10 e x11, respectivamente

add x21, x11, x11

add x21, x21, x21

add x21, x21, x21

add x21, x21, x5

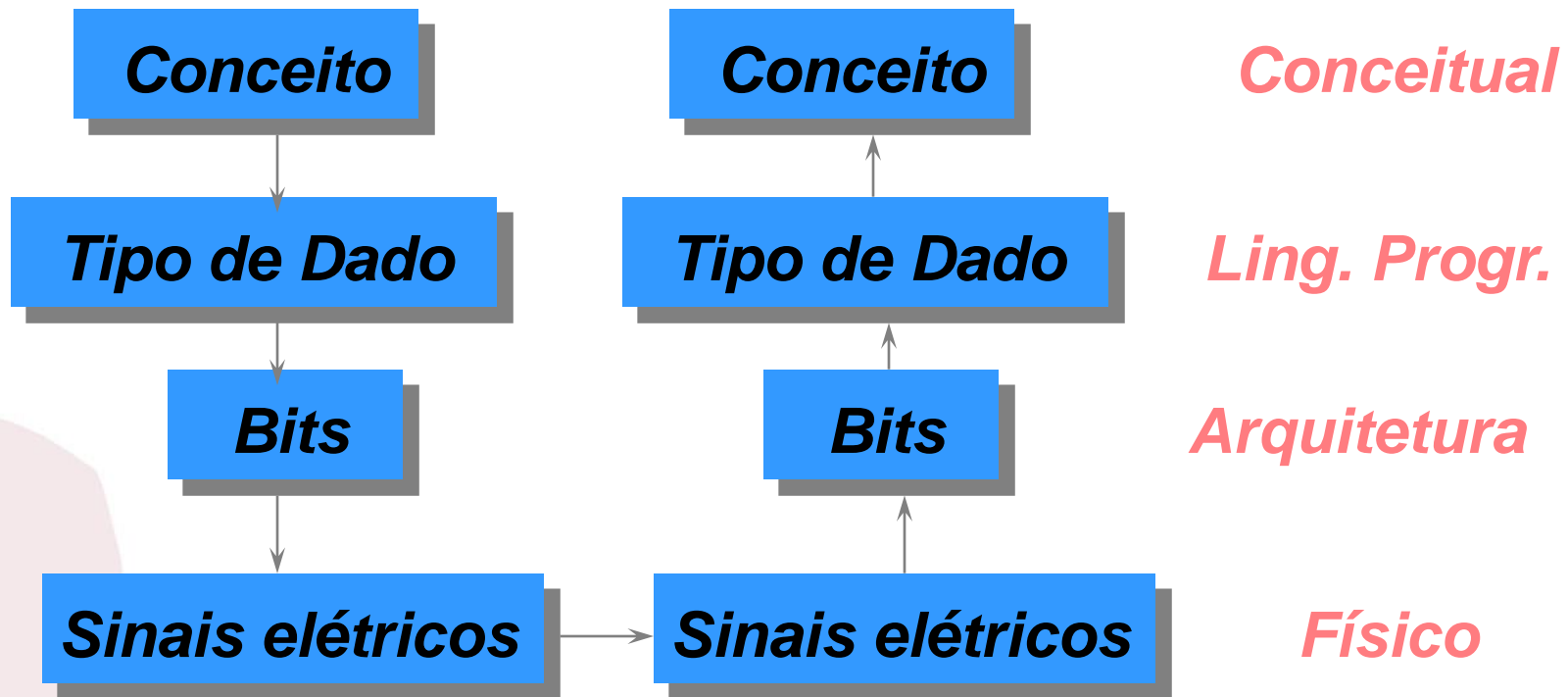
ld x22, 0(x21)

add x22, x10, x22

sd x22, 0(x21)



Representando Números



Inteiros

- Representação binária

- sinal-magnitude

00000...000001010

+ 10

10000...000001010

- 10

- complemento a 1

00000...000001010

+ 10

11111...111110101

- 10

- complemento a 2

00000...000001010

+ 10

11111...11110110

- 10

Inteiros sem sinal (MIPS)

- Dado um número de n-bits

$$x = x_{n-1} 2^{n-1} + x_{n-2} 2^{n-2} + \dots + x_1 2^1 + x_0 2^0$$

- Range: 0 até $+2^n - 1$

- Exemplo

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Usando 32 bits

- 0 até +4,294,967,295

Inteiros sem sinal (RISC V)



- Dado um número de n-bits

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 até $+2^n - 1$

- Exemplo

- $0000\ 0000\ \dots\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Usando 64 bits:

- 0 até $+18,446,774,073,709,551,615$



Inteiros com sinal 2s-Complement - MIPS



- Dado um número de n-bits

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} até $+2^{n-1} - 1$

- Exemplo

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Usando 32 bits

- $-2,147,483,648$ até $+2,147,483,647$



Inteiros com sinal 2s-Complement – RISC V



- Dado um número de n-bits

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} até $+2^{n-1} - 1$

- Exemplo

$$\begin{aligned} & - 1111 \ 1111 \ \dots \ 1111 \ 1100_2 \\ & = -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ & = -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

- Usando 64 bits: $-9,223,372,036,854,775,808$
até $9,223,372,036,854,775,807$



Inteiros com sinal 2s-Complement



- MIPS - Bit 31 – bit de sinal
- RISC V – Bit 63 – bit de sinal
 - 1 para números negativos
 - 0 para números não negativos
- $-(-2^n - 1)$ não pode ser representado
- Números positivos com única representação
- Algumas representações
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Maior negativo: 1000 0000 ... 0000
 - Maior positivo: 0111 1111 ... 1111



Gerando número negativo



- Complementar e adicionar 1
 - Complementar significa $1 \rightarrow 0, 0 \rightarrow 1$

$$\begin{array}{l} \text{---} \\ x + \bar{x} = 1111\dots111_2 = -1 \\ \text{---} \\ x + 1 = -x \end{array}$$

- Exemplo: negar +2
 - $+2 = 0000\ 0000 \dots 0010_2$
 - $-2 = 1111\ 1111 \dots 1101_2 + 1$
 $= 1111\ 1111 \dots 1110_2$



Extensão de Sinal



- Representando um número usando mais bits
 - Preservar o valor numérico
- Replicar o bit de sinal para a esquerda
 - c.f. valores sem sinal: estender com 0s
- Exemplos: 8 bits a 16 bits
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- No conjunto de instruções RISC-V
 - lb: byte carregado por extensão de sinal
 - lbu: byte carregado com extensão de zeros



Operandos Constantes Imediatos

- Valores constantes especificados na instrução
`addi x22, x22, 4`
- Implemente o comum de forma eficiente
 - Constantes “pequenas” são comuns
 - Operandos imediatos evitam instrução de load



Character Data

- Caracteres Byte-encoded
 - ASCII: 128 caracteres
 - 95 gráficos, 33 controle

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Character Data



- Unicode: caracteres 32-bit

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

Dados lógicos

- Representação
 - Um byte

00000000

Verdadeiro

00000001

Falso

- Um bit em uma palavra

0000....0000100000

Falso

Verdadeiro

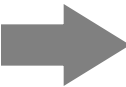
Ponto Flutuante



- Representação

$$3,14 = 0,314 \times 10^1 = 3,14 \times 10^0$$

$$0,000001 = 0,10 \times 10^{-5} = 1,00 \times 10^{-6}$$

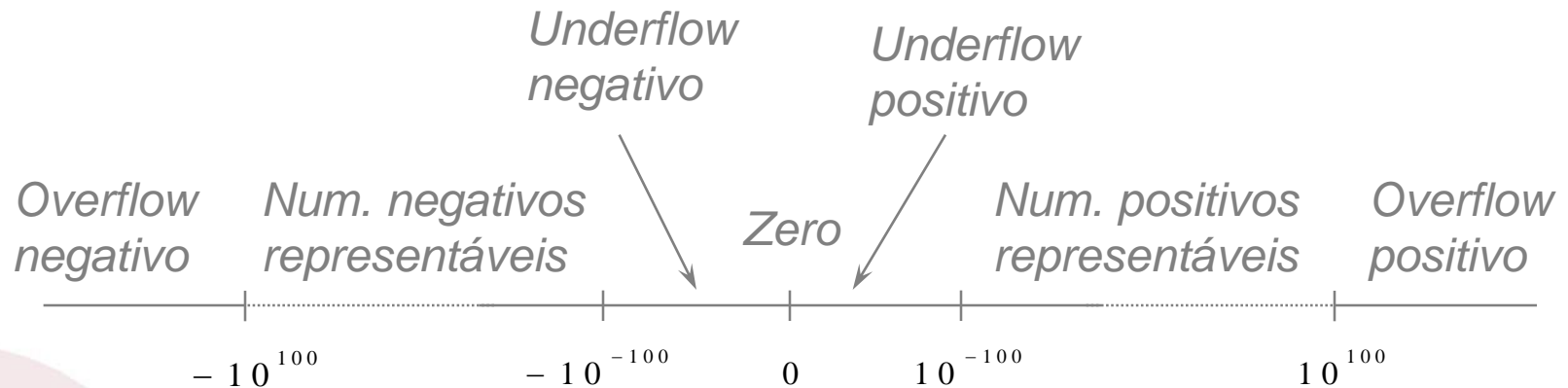

$$n = f \times 10^e$$

<i>sinal</i>	<i>expoente</i>	<i>mantissa (fração)</i>
--------------	-----------------	--------------------------



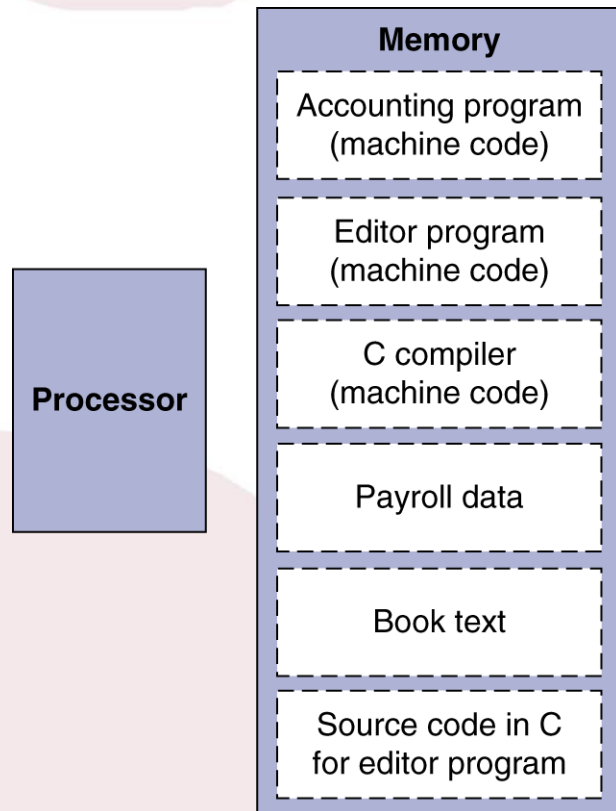
Ponto Flutuante

- Padrão IEEE



Item	Precisão simples	Precisão dupla
Sinal	1	1
Expoente	8	11
Mantissa	23	52
Total	32	64

Computador de Programa Armazenado



- Instruções representadas em binário, assim como dados
 - Instruções e dados armazenados na memória
- Programas podem operar em programas
 - p. ex., compiladores, linkers, ...
- A compatibilidade binária permite que os programas compilados funcionem em diferentes computadores
 - ISAs padronizados

Representando Instruções



- As instruções são codificadas em binário
 - Código de máquina
- Instruções RISC-V
 - Codificadas como instrução de 32 bits
 - Pequeno número de formatos e de códigos de operação (opcode),
 - números de registradores,...
- Regularidade!



Hexadecimal

- Base 16
 - Representação compacta de sequencia de bits
 - 4 bits por dígito hexadecimal

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Exemplo: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V: Formato de Instrução Tipo R



- Campos da instrução:
 - opcode: Código da instrução
 - rd: número do reg. destino
 - funct3: 3-bit Código da função (opcode adicional)
 - rs1: nr. do registrador fonte (1. operando)
 - rs2: nr. do registrador fonte (2. operando)
 - funct7: 7-bit Código da função (opcode adicional)

Exemplo : Formato de Instrução Tipo R

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

add x9, x20, x21

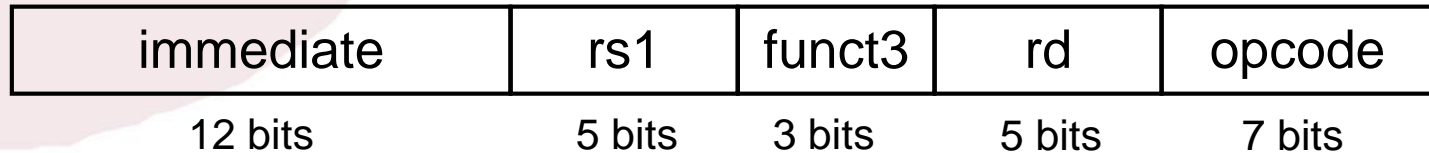
0	21	20	0	9	51
---	----	----	---	---	----

0000000	10101	10100	000	01001	0110011
---------	-------	-------	-----	-------	---------

0000 0001 0101 1010 0000 0100 1011 0011_{two} =
015A04B3₁₆



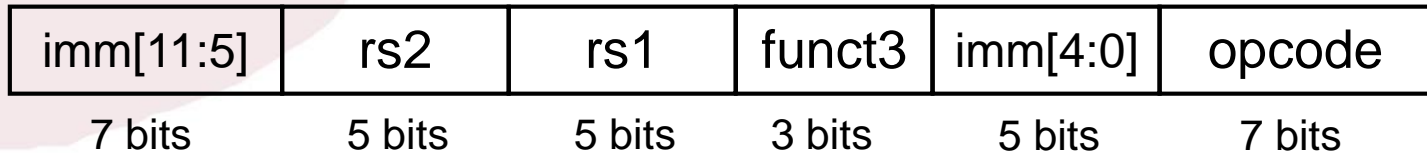
RISC-V: Formato de Instrução Tipo I



- Instruções com constantes (Immediate) e instruções load
 - rs1: nr. registrador fonte ou registrador base
 - immediate: operando constante ou offset adicionado ao end. base
 - 2s-complement, extensão sinal
- *Princípio de Projeto 3*: Bons projetos demandam bons compromissos
 - Formatos diferentes complicam a decodificação, mas permitem instruções de 32 bits uniformes
 - Mantenha os formatos o mais semelhantes possível



RISC-V : Formato de Instrução Tipo S



- Formato imediato para instruções store
 - rs1: nr. registrador base
 - rs2: nr. registrador fonte
 - immediate: offset adicionado ao end. base
 - Dividir para que os campos rs1 e rs2 sempre estejam no mesmo lugar



Formatos RISC V

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	Format	immediate		rs1	funct3	rd	opcode
addi (add immediate)	I	constant		reg	000	reg	0010011
ld (load doubleword)	I	address		reg	011	reg	0000011
Instruction	Format	immed- iate	rs2	rs1	funct3	immed- iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

OUTRAS INSTRUÇÕES DE PROCESSAMENTO DO DADO

Operações Lógicas

- Instruções para manipulação bit a bit

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srlr
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

- Útil para extrair e inserir grupos de bits em uma palavra

Operações de Deslocamento

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: quantas posições deslocar
- Shift para esquerda lógico
 - Deslocar para a esquerda e preenche com 0's
 - $sll\ i$ por i bits multiplica por 2^i
- Shift para a direita lógico
 - Deslocar para a direita e preenche com 0's
 - $srl\ i$ por i bits divide por 2^i (números sem sinal apenas)

Operações de Deslocamento

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: quantas posições deslocar
- Shift para a direita aritmético
 - Deslocar para a direita e preencher com bit mais significativo
 - $sra\ i$ por i bits divide por 2^i (números com sinal apenas)

Instrução AND



- Útil para mascarar bits em uma palavra
 - Preserva alguns bits, e muda outros para 0
- and x9, x10, x11

x10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x11 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

x9 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000



Instrução OR



- Útil para incluir bits em uma palavra
 - Muda alguns bits em 1, deixa os outros inalterados
- or x9, x10, x11

x10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x11 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

x9 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

Instrução XOR



- Operação diferencial
 - Complementa os valores dos bits
- xor x9, x10, x12 // operação NOT

x10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x12 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

x9 11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Controle de Fluxo



- Alterar a sequência de execução das instruções:

Ling. alto nível

- *If ...then ...else*
- *case*
- *while*
- *for*

Linguagem máquina

- *Desvio incondicional*
- *Desvio condicional a comparações entre variáveis e/ou valores*



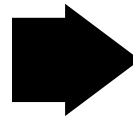
Desvios no RISC V



- Realizado pela instrução
 - branch if equal
 - beq reg1, reg2, label

Ling. alto nível

```
If a=b then  
    a:= a+c  
else  
    a:= a-c  
end if;
```



Linguagem máquina

```
ld x9,a  
ld x10,b  
ld x11,c  
beq x9, x10, end1  
sub x21, x9, x11  
beq x0,x0, end2  
end1: add x21, x9, x11  
end2: sd x21, a
```



Desvios condicionais no RISC V



- Instruções de comparação e desvio:
 - beq regd, regs, deslocamento
 $PC = PC + (\text{deslocamento} * 2)$ se $\text{regd} = \text{regs}$,
 - bne regd, regs, deslocamento
 $PC = PC + (\text{deslocamento} * 2)$ se $\text{regd} \neq \text{regs}$,



Compilando If Statements

- C code:

```
if (i==j) f = g+h;  
else f = g-h;
```

– f, g, ... in x19, x20, ...

- RISC-V code:

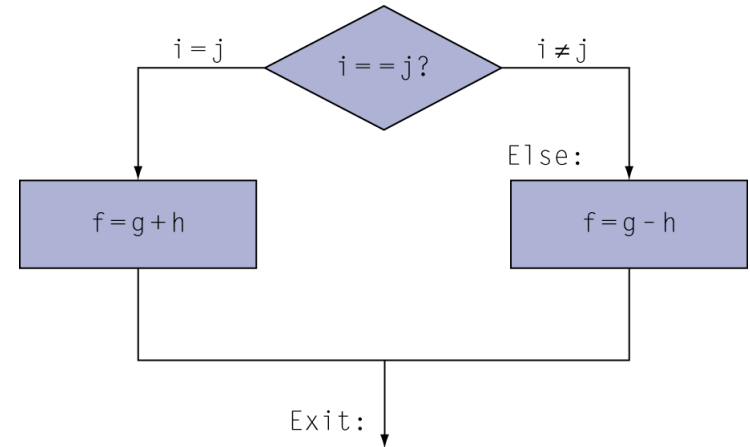
```
bne x22, x23, Else
```

```
add x19, x20, x21
```

```
beq x0,x0,Exit // unconditional
```

```
Else: sub x19, x20, x21
```

```
Exit: ...
```



Assembler calculates addresses

Compilando Loop Statements



- C code:

```
while (save[i] == k) i += 1;
```

– i em x22, k em x24, endereço de save em x25

- Código RISC-V:

```
Loop: slli x10, x22, 3
      add x10, x10, x25
      ld  x9, 0(x10)
      bne x9, x24, Exit
      addi x22, x22, 1
      beq x0, x0, Loop
```

Exit: ...



Mais condicionais



- `blt rs1, rs2, L1`
 - if ($rs1 < rs2$) branch to instruction labeled L1
- `bge rs1, rs2, L1`
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- Exemplo
 - if ($a > b$) $a += 1$;
 - a em x22, b em x23
 - `bge x23, x22, Exit` // branch if $b \geq a$
 - `addi x22, x22, 1`

Exit:



Signed vs. Unsigned



- Comparação com sinal: blt, bge
- Comparação sem sinal: bltu, bgeu
- Exemplo
 - $x_{22} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x_{23} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x_{22} < x_{23} \quad // \quad \text{signed}$
 - $-1 < +1$
 - $x_{22} > x_{23} \quad // \quad \text{unsigned}$
 - $+4,294,967,295 > +1$



RISC V



Instrução	Descrição
nop	No operation
ld reg, desl(reg_base)	reg. = mem (reg_base+desl)
sd reg, desl(reg_base)	Mem(reg_base+desl) = reg
add regi, regj, regk	Regi. <- Regj. + Regk
addi regi, regj, cte	Regi <- Regj + cte
sub regi, regj, regk	Regi. <- Regj. - Regk
and regi, regj, regk	Regi. <- Regj. and Regk
srli regd, regs, n	Desloca regs para direita logico n vezes e armazena em regd
srai regd, regs, n	Desloca regs para dir. aritm. N vezes e armazena em regd
slli regd, regs, n	Desloca regs para esquerda n vezes
beq regi, regj , desl	PC=PC+desl*2 se regi = regj
bne regi, regj, desl	PC=PC+desl*2 se regi <> regj

Procedimentos e Funções



- Implementação de procedimentos e funções

Ling. alto nível

Programa principal

Var i,j,k: integer

Procedure A (var x: integer);

...

Begin

...(corpo do procedimento)

End;

Begin

...

A(k); (chamada do procedimento)

....

End;

*Retorno após
a chamada*



*O endereço de
retorno deve ser salvo...*

mas onde?



Onde salvar o endereço de retorno?



- registradores
 - só permite chamadas seriais
- pilha
 - permite aninhamento e recursividade

```
...  
Procedure A  
begin  
  Procedure B  
  begin  
    Procedure C  
    begin  
      ...  
      C  
      ...  
    end  
  end  
end  
...
```

topo da
pilha



end. ret. de C
end. ret. de C
end. ret. de B
end. ret. de A

Chamada de função – RISC V



- Instruções:
 - Jal: Jump and Link
 - jal x1, ProcedureLabel
 - guarda endereço de retorno em \$ra (reg. 1)
 - muda fluxo de controle
 - jalr - jump and link to register
 - jalr x0, 0(x1)
 - recupera endereço de retorno de \$ra



Suporte Função – RISC V



100 jal x1,A
104 sub....

(300) A: add \$t0, \$t0, \$s2
.....

jalr x0, 0(x1)

PC

x1



Non-Leaf Funções



- Procedimentos que chamam outros procedimentos
- Para chamadas aninhadas, o chamador precisa salvar na pilha:
 - Seu endereço de retorno
 - Quaisquer argumentos e temporários necessários após a chamada
- Restaurar da pilha após a chamada



Suporte Funções Aninhadas – RISC V



100 jal x1, A

104 sub....

(500) B: sub \$t0, \$t0, \$s2

(300) A: add \$t0, \$t0, \$s2

.....

.....

PC

(340) jal x1, B

jalr x0,0(x1)

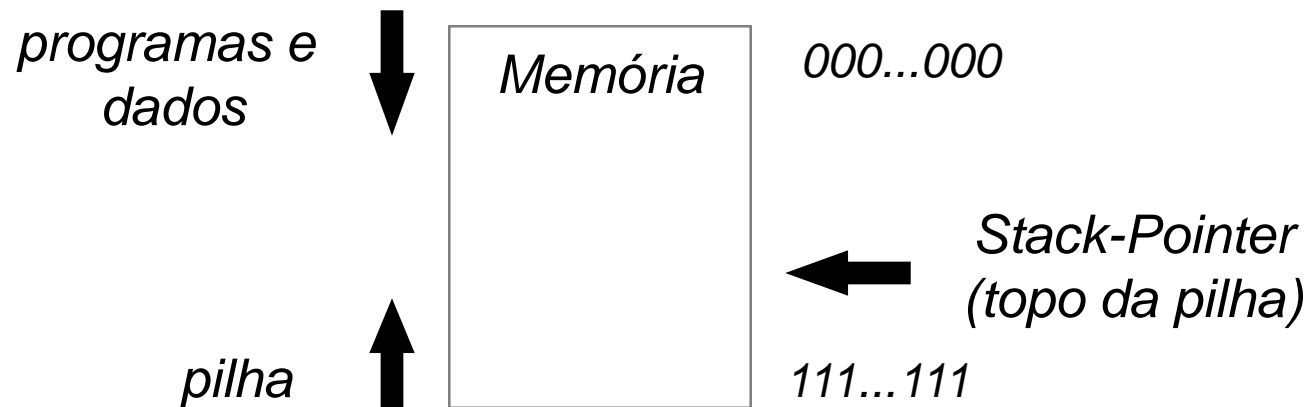
x1

jalr x0,0(x1)



Implementando a pilha

- Utiliza parte da memória como pilha



– SP: stack-pointer

Registradores RISC V

Register	ABI Name	Description	Saver
x0	zero	hardwired zero	-
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5-7	t0-2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-11	a0-1	function arguments / return values	Caller
x12-17	a2-7	function arguments	Caller
x18-27	s2-11	saved registers	Callee
x28-31	t3-6	temporary registers	Caller

Chamade de Procedimento



- Passos
 1. Armazena parametros em registradores
 2. Transfere controle para o procedimento
 3. Adquire armazenamento para procedimento (registradores)
 4. Realiza atribuições do procedimento
 5. Armazena resultados nos registradores
 6. Retorna



Chamada Funções no RISC V



- Passos
 1. Coloque os parâmetros nos registradores de x10 a x17
 2. Transferir o controle para o procedimento
 3. Adquirir armazenamento para procedimento
 4. Realize as operações do procedimento
 5. Coloque o resultado no registrador para o chamador
 6. Retornar ao local da chamada (endereço em x1)

Layout da Memória



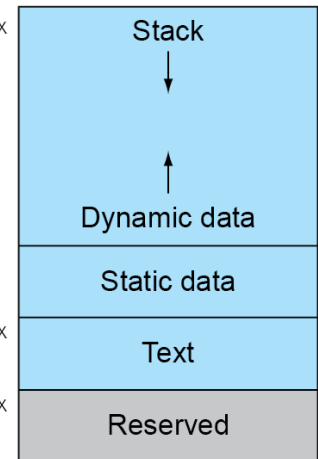
- Texto: código do programa
- Dados estáticos: var. globais
 - e.g., C: static variable, constant arrays e strings
 - (x3) \$gp inicializado para endereçar a partir de \pm offsets
- Dados dinâmicos: : heap
 - E.g., malloc em C, new em Java
- Stack: armazenamento automático

SP → 0000 003f ffff fff0_{hex}

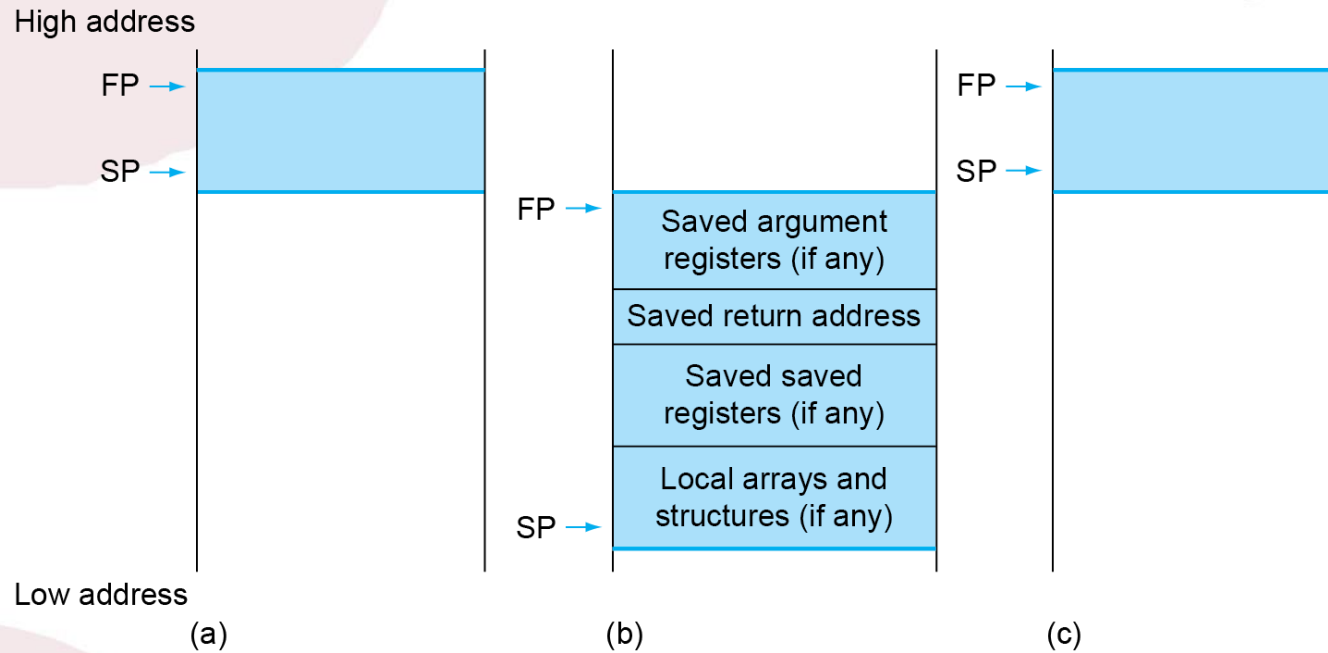
0000 0000 1000 0000_{hex}

PC → 0000 0000 0040 0000_{hex}

0



Dado Local na Pilha



- Dados locais alocado por quem chama
 - e.g., C variáveis automaticas
- Procedure frame (activation record)
 - Usado por alguns compiladores para gerenciar o armazenamento de pilha

Exemplo Leaf Procedure

- C code:

```
long long int leaf_example (  
    long long int g, long long int h,  
    long long int i, long long int j) {  
    long long int f;  
    f = (g + h) - (i + j);  
    return f;  
}
```

- Argumentos g, ..., j em x10, ..., x13
- f em x20
- temporários x5, x6
- Tem que salvar x5, x6, x20 na pilha

Dados locais na Pilha

High address

SP →

SP →

SP →

Contents of register x5

Contents of register x6

Contents of register x20

Low address

(a)

(b)

(c)

Exemplo Leaf Procedure

- RISC-V code:

leaf_example:

addi sp,sp,-24

Save x5, x6, x20 on stack

sd x5,16(sp)

sd x6,8(sp)

sd x20,0(sp)

add x5,x10,x11

$x5 = g + h$

add x6,x12,x1

$x6 = i + j$

sub x20,x5,x6

$f = x5 - x6$

addi x10,x20,0

copy f to return register

ld x20,0(sp)

Resore x5, x6, x20 from stack

ld x6,8(sp)

ld x5,16(sp)

addi sp,sp,24

jalr x0,0(x1)

Return to caller



Exemplo Non-Leaf Procedure



- C code:

```
long long int fact (long long int  
n)  
{  
    if (n < 1) return f;  
    else return n * fact(n - 1);  
}
```

- Argumento n em x10
- Resultado em x10

Exemplo Non-Leaf Procedure



- RISC-V code:

fact:

```
    addi sp,sp,-16
    sd    x1,8(sp)
    sd    x10,0(sp)
    addi  x5,x10,-1
    bge   x5,x0,L1
    addi  x10,x0,1
    addi  sp,sp,16
    jalr  x0,0(x1)
L1: addi  x10,x10,-1
    jal   x1,fact
    addi  x6,x10,0
    ld    x10,0(sp)
    ld    x1,8(sp)
    addi  sp,sp,16
    mul   x10,x10,x6
    jalr  x0,0(x1)
```

Save return address and n on stack

$x5 = n - 1$

if $n \geq 1$, go to L1

Else, set return value to 1

Pop stack, don't bother restoring values

Return

$n = n - 1$

call fact(n-1)

move result of fact(n - 1) to x6

Restore caller's n

Restore caller's return address

Pop stack

return $n * \text{fact}(n-1)$

return

Chamada de função em outros processadores

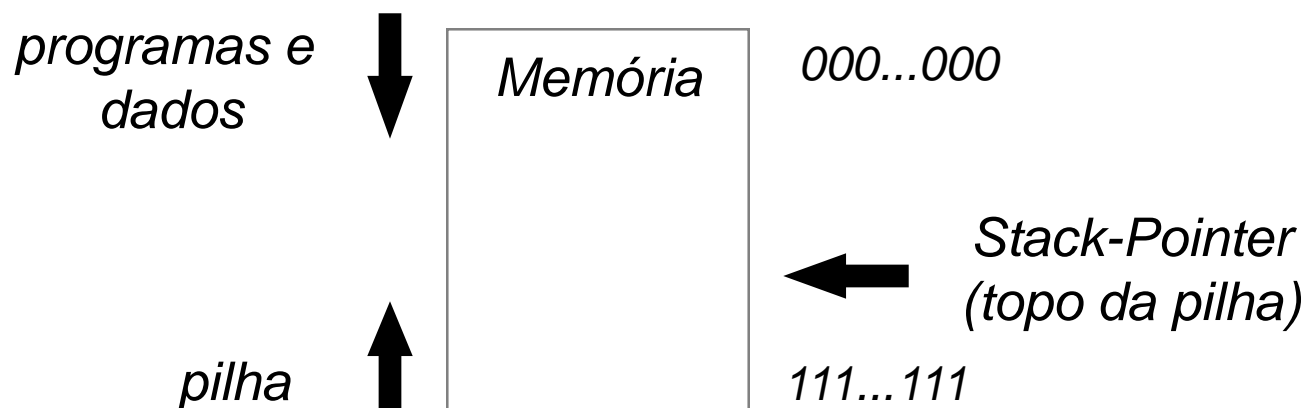
- Instruções:
 - call
 - empilha endereço de retorno
 - muda fluxo de controle
 - ret
 - recupera endereço de retorno
- Outras instruções de suporte...
 - Salvar todos registradores na pilha
 - alocar parte da pilha para armazenar variáveis locais e parâmetros



Usando a pilha em outros processadores



- Utiliza parte da memória como pilha



- SP: Registrador adicional
- Instruções adicionais:
 - push reg: $\text{mem}(\text{SP}) \leftarrow \text{reg}$; decrementa SP
 - pop reg: incrementa SP, $\text{reg} \leftarrow \text{mem}(\text{SP})$;

RISC V vs. Outros processadores



- Endereço de retorno:
 - RISC V: registrador
 - Outras: Memória
 - Melhor desempenho
- Acesso à Pilha:
 - RISC V: instruções ld e sd
 - Outras: instruções adicionais
 - Menor complexidade na implementação
 - Compilador mais complexo



RISC V vs. Outros processadores



- Chamadas aninhadas ou recursivas
 - RISC V: implementada pelo compilador
 - Outras: suporte direto da máquina
 - Compilador mais complexo



RISC V

Instrução	Descrição
nop	No operation
ld reg, desl(reg_base)	reg. = mem (reg_base+desl)
sd reg, desl(reg_base)	Mem(reg_base+desl) = reg
add regi, regj, regk	Regi. <- Regj. + Regk
sub regi, regj, regk	Regi. <- Regj. - Regk
and regi, regj, regk	Regi. <- Regj. and Regk
addi regi, regj, cte	Regi = Regj + cte
srli regd, regs, n	Desloca regs para direita n vezes sem preservar sinal, armazena valor deslocado em regd
srai regd, regs, n	Desloca regs para dir. n vezes preservando o sinal, armazena valor deslocado em regd.
slli regd, regs, n	Desloca regs para esquerda n vezes, armazena valor deslocado em regd.
beq regi, regj, desl	PC = PC + desl*2 se regi = regj
bne regi, regj, desl	PC = PC + desl *2 se regi <> regj
jal ra,end	ra = pc, pc = end
jalr ra, desl(reg-dst)	Ra=Pc Pc= reg-dst+desl
break	Para a execução do programa

Dado Caractere



- Conjuntos de caracteres codificados por byte
 - ASCII: 128 caracteres
 - 95 gráfico, 33 controle
 - Latim-1: 256 caracteres
 - ASCII, mais 96 caracteres gráficos
 - Unicode: conjunto de caracteres de 32 bits
 - Usado em Java, caracteres largos C ++,...
 - A maioria dos alfabetos, além de símbolos
 - UTF-8, UTF-16: codificações de comprimento variável

Operações Byte/Halfword/Word



- RISC-V byte/halfword/word load/store
 - Load byte/halfword/word: Sign extend to 64 bits in rd
 - `lb rd, offset(rs1)`
 - `lh rd, offset(rs1)`
 - `lw rd, offset(rs1)`
 - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
 - `lbu rd, offset(rs1)`
 - `lhu rd, offset(rs1)`
 - `lwu rd, offset(rs1)`
 - Store byte/halfword/word: Store rightmost 8/16/32 bits
 - `sb rs2, offset(rs1)`
 - `sh rs2, offset(rs1)`
 - `sw rs2, offset(rs1)`



Instrução	Descrição
nop	No operation
ld reg, desl(reg_base)	reg. = mem (reg_base+desl)
sd reg, desl(reg_base)	Mem(reg_base+desl) = reg
lw reg, desl(reg_base)	reg. (31:0)= mem [reg_base+desl](31:0) , extensão do sinal
sw reg, desl(reg_base)	Mem[reg_base+desl](31:0) = reg(31:0)
lh reg, desl(reg_base)	reg. (15:0)= mem [reg_base+desl](15:0) , extensão do sinal
sh reg, desl(reg_base)	Mem[reg_base+desl](15:0) = reg(15:0)
lbu reg, desl(reg_base)	reg. (7:0)= mem [reg_base+desl](7:0) , completa com zeros
sb (reg, desl(reg_base))	Mem[reg_base+desl](7:0) = reg(7:0)
add regi, regj, regk	Regi. <- Regj. + Regk
sub regi, regj, regk	Regi. <- Regj. - Regk
and regi, regj, regk	Regi. <- Regj. and Regk
addi regi, regj, cte	Regi = Regj + cte
srli regd, regs, n	Desloca regs para direita n vezes sem preservar sinal, armazena valor deslocado em regd
srai regd, regs, n	Desloca regs para dir. n vezes preservando o sinal, armazena valor deslocado em regd.
slli regd, regs, n	Desloca regs para esquerda n vezes, armazena valor deslocado em regd.
slt regi, regj, regk	Regi = 1 se (regj) < (regk) caso contrário regi= 0
slti regi, regj, immed	Regi = 1 se (regj) < extensão sinal de immed, caso contrário regi= 0
beq regi, regj, desl	PC = PC + desl*1 se regi = regj
bne regi, regj, desl	PC = PC + desl *1 se regi <> regj
bge regi, regj, desl	PC = PC + desl*1 se regi >=regj
blt regi, regj, desl	PC = PC + desl*1 se regi < regj
jal ra, end	ra = pc, pc = end, se Ra=0 não guarda PC
jalr ra, desl(reg-dst)	Ra=Pc Pc= reg-dst+desl, se Ra=0 não guarda PC
break	Para a execução do programa

Exemplo String Copy

- C code:

- Null-terminated string

```
void strcpy (char x[], char y[])  
{ size_t i;  
  i = 0;  
  while ((x[i]=y[i])!='\0')  
    i += 1;  
}
```

Exemplo String Copy

- RISC-V code:

strcpy:

```
    addi sp,sp,-8           // adjust stack for 1
doubleword
    sd    x19,0(sp)         // push x19
    add   x19,x0,x0         // i=0
L1:  add   x5,x19,x10        // x5 = addr of y[i]
     lbu   x6,0(x5)         // x6 = y[i]
     add   x7,x19,x10        // x7 = addr of x[i]
     sb    x6,0(x7)         // x[i] = y[i]
     beq   x6,x0,L2         // if y[i] == 0 then exit
     addi  x19,x19,1         // i = i + 1
     jal   x0,L1            // next iteration of loop
L2:  ld    x19,0(sp)         // restore saved x19
     addi  sp,sp,8          // pop 1 doubleword from stack
     jalr  x0,0(x1)         // and return
```

Constante 32-bits



- A maioria das constantes é pequena
 - 12 bits imediatos é suficiente
- Para constantes de 32 bits
- `lui rd, constant`
 - Copia a constante de 20 bits em bits [31:12] de rd
 - Estende o bit 31 para bits [63:32]
 - Limpa os bits [11: 0] de rd para 0

```
lui x19, 976 // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000
---------------------	---------------------	--------------------------	----------------

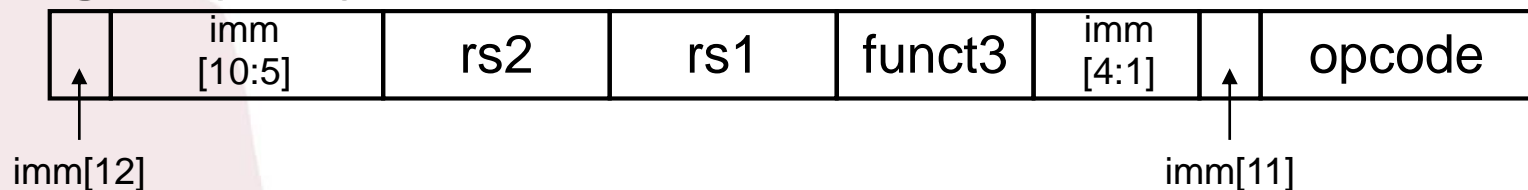
```
addi x19,x19,128 // 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000
---------------------	---------------------	--------------------------	----------------



Desvios Condicionais

- As instruções de filial especificam
 - Opcode, dois registradores, endereço de destino
- A maioria dos alvos do desvio estão próximos
 - Para frente ou para trás
 - SB format:

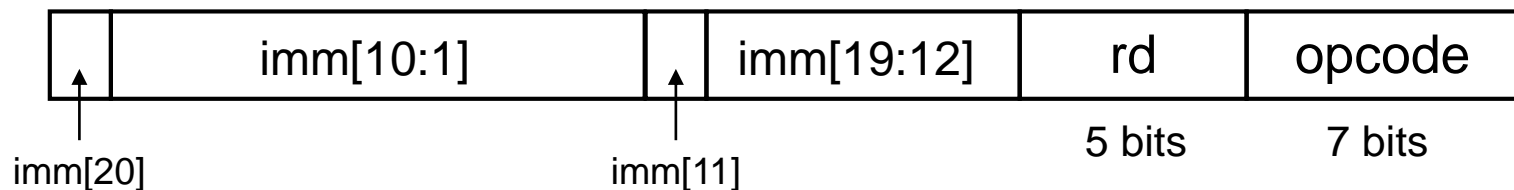


■ PC-relative addressing

- Target address = PC + immediate × 2

Desvios Incondicionais

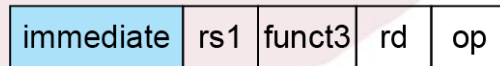
- Endereço de desvio e link (jal) usa imediato de 20 bits para maior alcance
- Formato UJ:



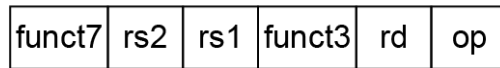
- Para saltos longos, por exemplo, para endereço absoluto de 32 bits
 - lui: carrega end. [31:12] para registrador
 - jalr: adiciona end. [11:0] ao reg. e desvia

RISC-V Endereçamentos

1. Immediate addressing



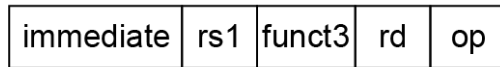
2. Register addressing



Registers

Register

3. Base addressing



Memory

Register

+

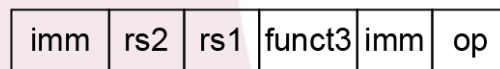
Byte

Halfword

Word

Doubleword

4. PC-relative addressing



Memory

PC

+

Word

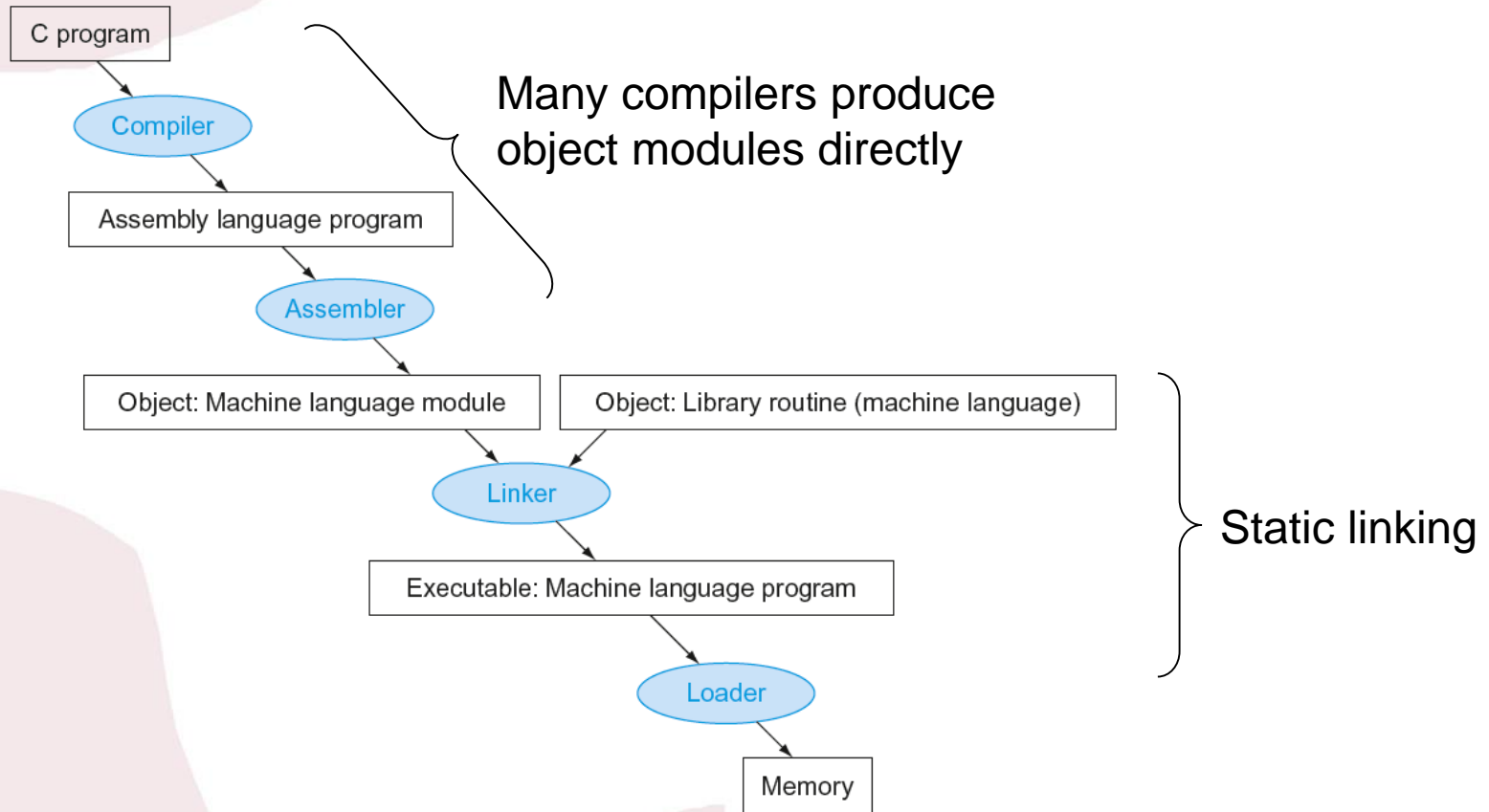
RISC-V Formatos de Instrução



Name (Field Size)	Field					Comments	
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format



Tradução e Startup



Simulador Venus



- Código fonte: <https://github.com/kvakil/venus>



Simulador Venus



- Código fonte: <https://github.com/kvakil/venus>
- Disponível para uso online: <http://www.kvakil.me/venus/>



Simulador Venus



- Suporta o padrão RV32IM.
 - <https://riscv.org/specifications/>
 - <https://rv8.io/isa.html>



Simulador Venus



- Suporta o padrão RV32IM.
 - https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISC_VGreenCardv8-20151013.pdf

Loads	Load Byte	I	LB	rd,rs1,imm		
	Load Halfword	I	LH	rd,rs1,imm		
	Load Word	I	LW	rd,rs1,imm	L{D Q}	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm		
	Load Half Unsigned	I	LHU	rd,rs1,imm	L{W D}U	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	S{D Q}	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2	SLL{W D}	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt	SLLI{W D}	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2	SRL{W D}	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt	SRLI{W D}	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRA{W D}	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt	SRAI{W D}	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADD{W D}	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDI{W D}	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUB{W D}	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		

Optional Compress		
Category	Name	Fmt
Loads	Load Word	CL
	Load Word SP	CI
	Load Double	CL
	Load Double SP	CI
	Load Quad	CL
	Load Quad SP	CI



Exemplo de Programa em Linguagem de Montagem



- Escreva o seguinte código na aba **Editor**

www.kvakil.me/venus/

Editor

Simulator

```
1 addi a0, zero, 5
2 addi a1, zero, 10
3 addi a2, zero, 5
4 add a3,a0,a1
5 sub a3,a3,a2
6
```

```
addi a0, zero, 5
addi a1, zero, 10
addi a2, zero, 5
add a3,a0,a1
sub a3,a3,a2
```



Exemplo de Programa em Linguagem de Montagem



- Vá para aba **Simulator** para executar o código

Editor

Simulator

Run

Step

Prev

Reset

Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

console output

Infra-estrutura Hardware

Registers

Memory

zero

0

ra (x1)

0

sp (x2)

2147483632

gp (x3)

268435456

tp (x4)

0

t0 (x5)

0

t1 (x6)

0

t2 (x7)

0

s0 (x8)

0

s1 (x9)

0

a0 (x10)

0

Display Settings

Decimal

Exemplo de Programa em Linguagem de Montagem



- **Original code** seria o código em uma escrita com diretivas de alto nível. (ex: zero)
- **Basic code** seria o código com as diretivas traduzidas para assembly.

Editor

Simulator

Run

Step

Prev

Reset

Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

console output

RegistersMemory

zero	0
ra (x1)	0
sp (x2)	2147483632
gp (x3)	268435456
tp (x4)	0
t0 (x5)	0
t1 (x6)	0
t2 (x7)	0
s0 (x8)	0
s1 (x9)	0
a0 (x10)	0

Display Settings

Decimal

Infra-estrutura Hardware

Exemplo de Programa em Linguagem de Montagem



- Diretivas suportadas
 - <https://github.com/kvakil/venus/wiki/Assembler-Directives>

Directive	Effect
<code>.data</code>	Store subsequent items in the [[static segment
<code>.text</code>	Store subsequent instructions in the [[text segment
<code>.byte</code>	Store listed values as 8-bit bytes.
<code>.ascii</code>	Store subsequent string in the data segment and add null-terminator.
<code>.word</code>	Store listed values as unaligned 32-bit words.
<code>.globl</code>	Makes the given label global.
<code>.float</code>	Reserved.
<code>.double</code>	Reserved.
<code>.align</code>	Reserved.



Exemplo de Programa em Linguagem de Montagem



- Visualize as informações armazenadas nos **registradores** do RISC V

Editor

Simulator

Run

Step

Prev

Reset

Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

console output

Registers

Memory

zero

0

ra (x1)

0

sp (x2)

2147483632

gp (x3)

268435456

tp (x4)

0

t0 (x5)

0

t1 (x6)

0

t2 (x7)

0

s0 (x8)

0

s1 (x9)

0

a0 (x10)

0

Display Settings



Decimal

Infra-estrutura Hardware

Exemplo de Programa em Linguagem de Montagem



- Visualize as informações armazenadas nos **registradores** do RISC V



	Registers	Memory
zero	0	
ra (x1)	0	
sp (x2)	2147483632	
gp (x3)	268435456	
tp (x4)	0	
t0 (x5)	0	
t1 (x6)	0	
t2 (x7)	0	
s0 (x8)	0	



Exemplo de Programa em Linguagem de Montagem



- Defina a base numérica em que valores serão visualizados

<code>s0 (x8)</code>	<input type="text" value="0"/>
<code>s1 (x9)</code>	<input type="text" value="0"/>
<code>a0 (x10)</code>	<input type="text" value="0"/>

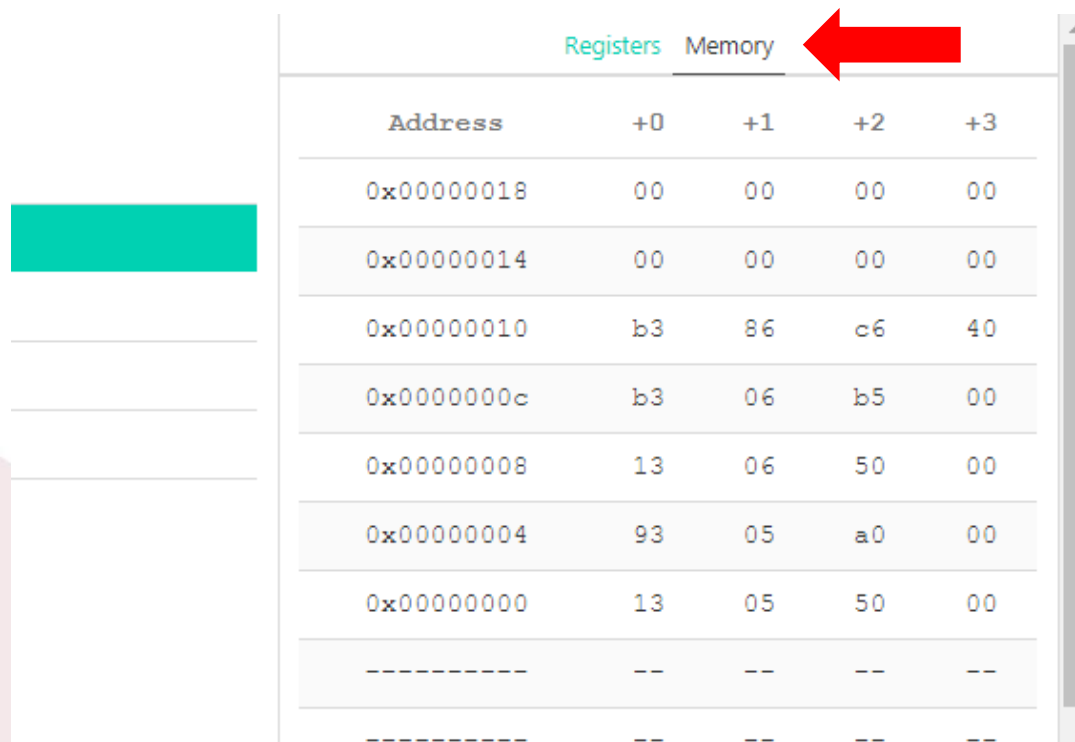
Display Settings



Exemplo de Programa em Linguagem de Montagem



- Visualize as informações armazenadas na **memória** do RISC V

A screenshot of a memory viewer interface. On the left, there is a vertical teal bar and several horizontal lines. The main area is a table with columns for "Address" and four offset columns labeled "+0", "+1", "+2", and "+3". A red arrow points to the "Memory" tab at the top of the table. The table contains several rows of hexadecimal addresses and their corresponding values in the four offset columns.


	Registers	Memory			
		+0	+1	+2	+3
	Address				
	0x00000018	00	00	00	00
	0x00000014	00	00	00	00
	0x00000010	b3	86	c6	40
	0x0000000c	b3	06	b5	00
	0x00000008	13	06	50	00
	0x00000004	93	05	a0	00
	0x00000000	13	05	50	00
	-----	--	--	--	--
	-----	--	--	--	--




Exemplo de Programa em Linguagem de Montagem



- Visualize as informações armazenadas na **memória** do RISC V



	Registers	Memory			
Address	+0	+1	+2	+3	
0x00000018	00	00	00	00	 Deslocamento de Bytes
0x00000014	00	00	00	00	
0x00000010	b3	86	c6	40	
0x0000000c	b3	06	b5	00	
0x00000008	13	06	50	00	
0x00000004	93	05	a0	00	
0x00000000	13	05	50	00	
-----	--	--	--	--	
-----	--	--	--	--	



Exemplo de Programa em Linguagem de Montagem



- Vá direto para a região de memória específica

	0x0ffffec	0	0	0
	0x0ffffe8	0	0	0

Jump to	-- choose -- ▾	Up	Down
	Text Data Heap Stack		

Display Settings



Exemplo de Programa em Linguagem de Montagem



- O comando **run** executará todas as instruções sem parar



Run

Step

Prev

Reset

Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

console output

Infra-estrutura Hardware

Registers	
zero	0
ra (x1)	0
sp (x2)	2147483632
gp (x3)	268435456
tp (x4)	0
t0 (x5)	0
t1 (x6)	0
t2 (x7)	0
s0 (x8)	0
s1 (x9)	0
a0 (x10)	0
Display Settings	
Decimal	

Debugging



- Se alguma instrução tiver **breakpoint** o comando run parará nessa instrução
- Ao clicar na instrução você adicionará o breakpoint.

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2



Debugging



- O comando **step** executa instrução por instrução

Run Step Prev Reset Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

console output

Registers Memory

zero	0x00000000
ra (x1)	0x00000000
sp (x2)	0x7fffffff0
gp (x3)	0x10000000
tp (x4)	0x00000000
t0 (x5)	0x00000000
t1 (x6)	0x00000000
t2 (x7)	0x00000000
s0 (x8)	0x00000000
s1 (x9)	0x00000000
a0 (x10)	0x00000005



Debugging



- Valores dos registradores são atualizados de acordo com a instrução

Run Step Prev Reset Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

a0
cor (x10) 0x00000005

Registers Memory

Register	Value
zero	0x00000000
ra (x1)	0x00000000
sp (x2)	0x7fffffff0
gp (x3)	0x10000000
tp (x4)	0x00000000
t0 (x5)	0x00000000
t1 (x6)	0x00000000
t2 (x7)	0x00000000
s0 (x8)	0x00000000
s1 (x9)	0x00000000
a0 (x10)	0x00000005



Debugging



- É possível voltar cada passo

Editor Simulator

Run Step Prev Reset Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi a0, zero, 5
0x00a00593	addi x11 x0 10	addi a1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

Registers Memory

zero	0
ra (x1)	0
sp (x2)	2147483632
gp (x3)	268435456
tp (x4)	0
t0 (x5)	0
t1 (x6)	0



Debugging



- Ou voltar para o começo

Editor

Simulator

Run

Step

Prev

Reset

Dump

Machine Code	Basic Code	Original Code
0x00500513	addi x10 x0 5	addi x0, zero, 5
0x00a00593	addi x11 x0 10	addi x1, zero, 10
0x00500613	addi x12 x0 5	addi a2, zero, 5
0x00b506b3	add x13 x10 x11	add a3,a0,a1
0x40c686b3	sub x13 x13 x12	sub a3,a3,a2

Registers

Memory

zero	0
ra (x1)	0
sp (x2)	2147483632
gp (x3)	268435456
tp (x4)	0
t0 (x5)	0
t1 (x6)	0



Console output



- Visualize os dados no console

[Run](#) [Step](#) [Prev](#) [Reset](#) [Dump](#)

Machine Code	Basic Code	Original Code
0x00100513	addi x10 x0 1	addi a0 x0 1 # print_int ecall
0x02a00593	addi x11 x0 42	addi a1 x0 42 # integer 42
0x00000073	ecall	ecall


42



Console output



- Para visualizar:
 - Carregue a ID (1 para inteiro e 4 para string) no registrador a0 e carregue o argumento em a1 – a7.



ID (a0)	Name	Description
1	print_int	prints integer in a1
4	print_string	prints the null-terminated string whose address is in a1
9	sbrk	allocates a1 bytes on the heap, returns pointer to start in a0
10	exit	ends the program
11	print_character	prints ASCII character in a1
17	exit2	ends the program with return code in a1



Console output



- Para visualizar:
 - Em seguida, chame a instrução `ecall` (environment call)



Console output



- Exemplo para testar:
 - Escreva este código e em seguida clique em **run**

```
addi a0 x0 1      # print_int ecall
addi a1 x0 42     # integer 42
ecall
```



Console output



- Visualize os dados no console
 - Deverá aparecer o valor 42 no console

<div>Run Step Prev Reset Dump</div>		
Machine Code	Basic Code	Original Code
0x00100513	addi x10 x0 1	addi a0 x0 1 # print_int ecall
0x02a00593	addi x11 x0 42	addi a1 x0 42 # integer 42
0x00000073	ecall	ecall

42



Console output



- Para visualizar:
 - Mas detalhes sobre environment call:
 - <https://github.com/kvakil/venus/wiki/Environmental-Calls>

ID (a0)	Name	Description
1	print_int	prints integer in a1
4	print_string	prints the null-terminated string whose address is in a1
9	sbrk	allocates a1 bytes on the heap, returns pointer to start in a0
10	exit	ends the program
11	print_character	prints ASCII character in a1
17	exit2	ends the program with return code in a1



Outro exemplo...carregando dado da memória e loop

Escreva o seguinte código no espaço Editor

.data

a: .word 4

.text

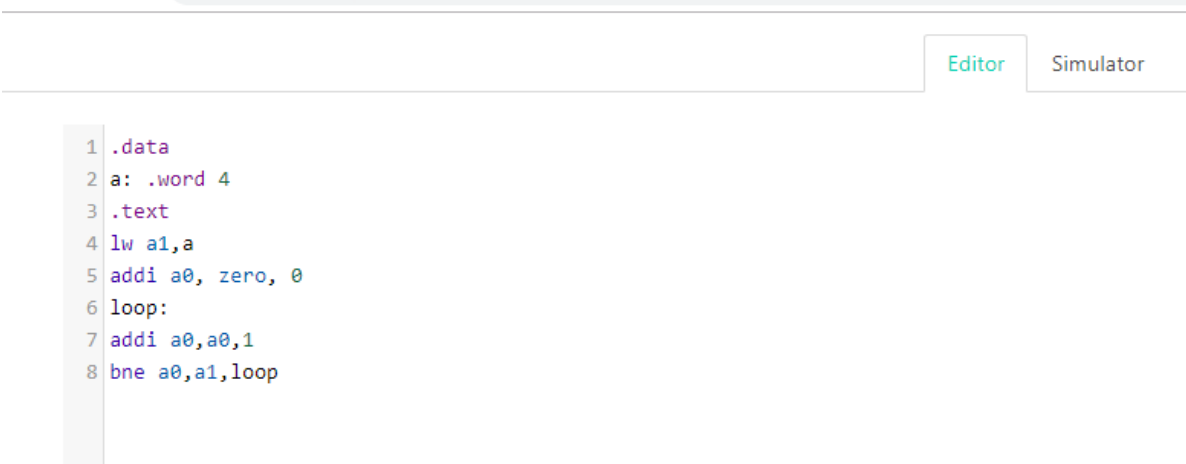
lw a1,a

addi a0, zero, 0

loop:

addi a0,a0,1

bne a0,a1,loop



```
1 .data
2 a: .word 4
3 .text
4 lw a1,a
5 addi a0, zero, 0
6 loop:
7 addi a0,a0,1
8 bne a0,a1,loop
```



Outro exemplo...carregando dado da memória e loop



A memória de dados começa no endereço 0x10000000
Que é o mesmo apontado pelo registrador gp

gp

0x10000018	0	0	0	0
0x10000014	0	0	0	0
0x10000010	0	0	0	0
0x1000000c	0	0	0	0
0x10000008	0	0	0	0
0x10000004	0	0	0	0
0x10000000	4	0	0	0

Jump to -- choose -- Up Down



Outro exemplo...carregando dado da memória e loop



Instrução lw a1,a terminou
registrador x11 (ou a1) agora está com o valor 4

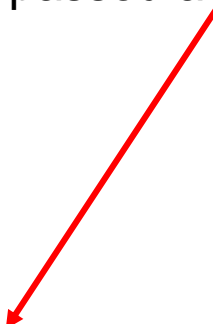
	0x00000000
a0 (x10)	0x00000000
a1 (x11)	0x00000004
a2 (x12)	0x00000000



Outro exemplo...carregando dado da memória e loop



Após executar o run o registrador x10 (ou a0) passou a ter o valor 4



a0 (x10)	0x00000004
a1 (x11)	0x00000004



Outros tipos de dados



- Para declarar uma string:
 - Nome: .asciiz *"meu nome é.."*
 - Exemplo:
 - msg:.asciiz"1-2-3-4-5"
 - lb x5, 0(gp) #colocando o valor que esta na memoria



Outros tipos de dados



- Escrevam o seguinte código:
- .data
- msg:.asciiz "1-2-3-4-5"
- .text
- lb a2,(0)gp
- lb a3,(1)gp
- lb a4,(2)gp
- lb a5,(3)gp
- lb a6,(4)gp
- lb a7,(5)gp



Outros tipos de dados

- Observe que o registrador gp aponta para o início da string na memória

The screenshot shows a debugger interface. On the left, the 'gp (x3)' register is highlighted with the value '0x10000000'. A red arrow points from this value to the first row of a memory dump table. The table lists memory addresses and their corresponding ASCII values. The first row, at address 0x10000000, contains the characters '1', '-', '2', and '-', forming the string '1-2-'. Below the table are controls for 'Jump to' and 'Display Settings' (set to ASCII). The system tray at the bottom shows the time as 09:54 on 12/03/2019.

0x10000010	?	?	?	?
0x1000000c	?	?	?	?
0x10000008	'5'	?	?	?
0x10000004	'3'	'-'	'4'	'-'
0x10000000	'1'	'-'	'2'	'-'

Jump to: -- choose -- Up Down

Display Settings: ASCII

PT 09:54 12/03/2019

Outros tipos de dados

- Execute run e verifique os valores da string nos registradores

a2 (x12)	'1'
a3 (x13)	'-'
a4 (x14)	'2'
a5 (x15)	'-'
a6 (x16)	'3'
a7 (x17)	'-'

Outros tipos de dados



- Outra maneira similar

- .data
- msg:.asciiz "1-2-3-4-5"
- .text
- la a1, msg
- lb a2,(0)a1
- lb a3,(1)a1
- lb a4,(2)a1
- lb a5,(3)a1
- lb a6,(4)a1

A instrução la
Carrega o endereço base de msg



Simulador Ripes



- Local: <https://github.com/mortbopet/Ripes>
- Funciona no ubuntu 16 e no windows 10 64 bits
- Para windows 10:
 - Acesse <https://github.com/mortbopet/Ripes/releases>
 - Baixar o arquivo Ripes_1.0_win.zip
 - Descomprima e Execute Ripes.exe
 - Talvez seja necessário instalar o msvcp140.dll:
 - <https://www.microsoft.com/en-us/download/details.aspx?id=48145>



Exemplo de Programa em Linguagem de Montagem

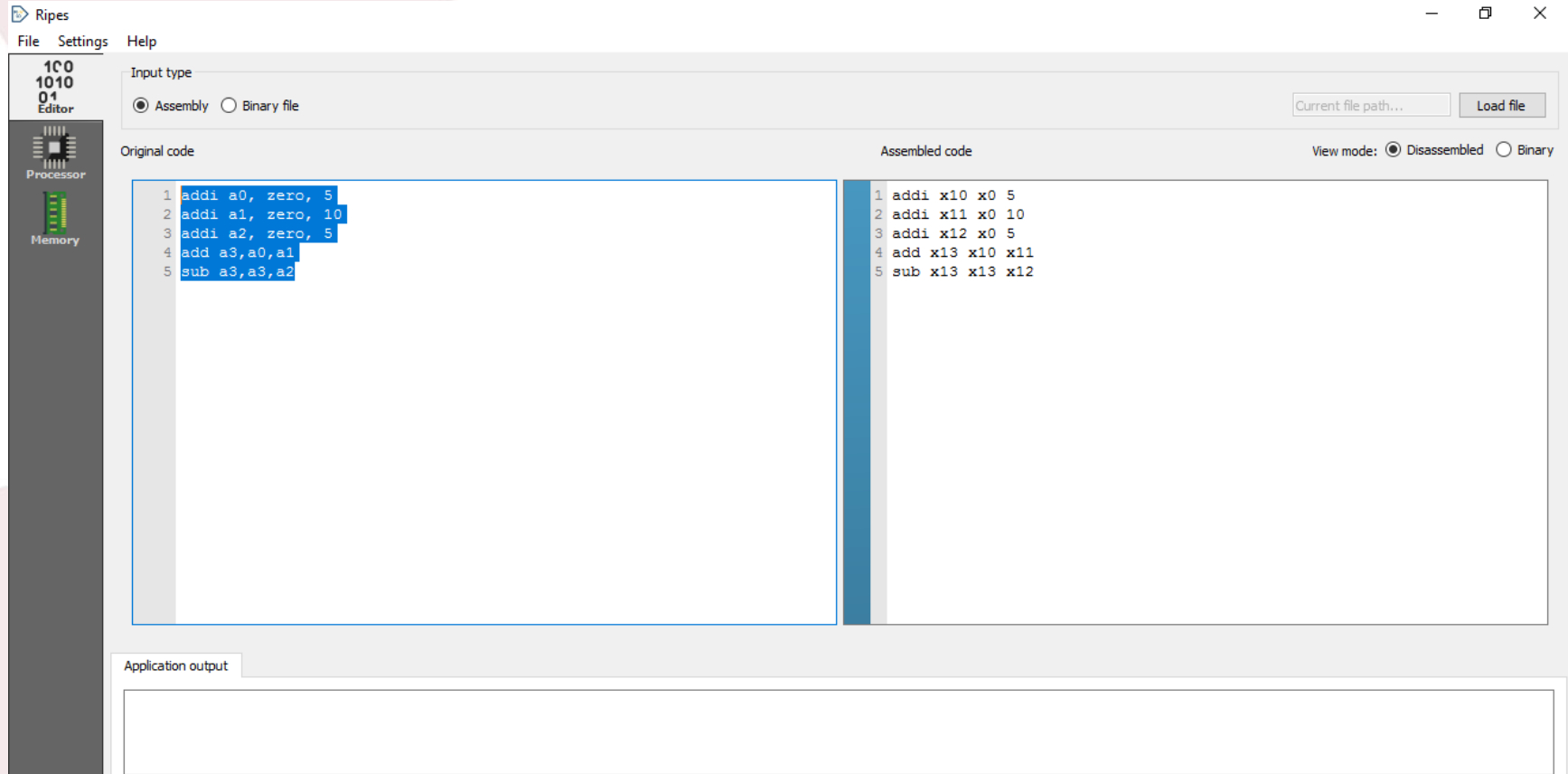


Escreva o seguinte código no espaço “original code”

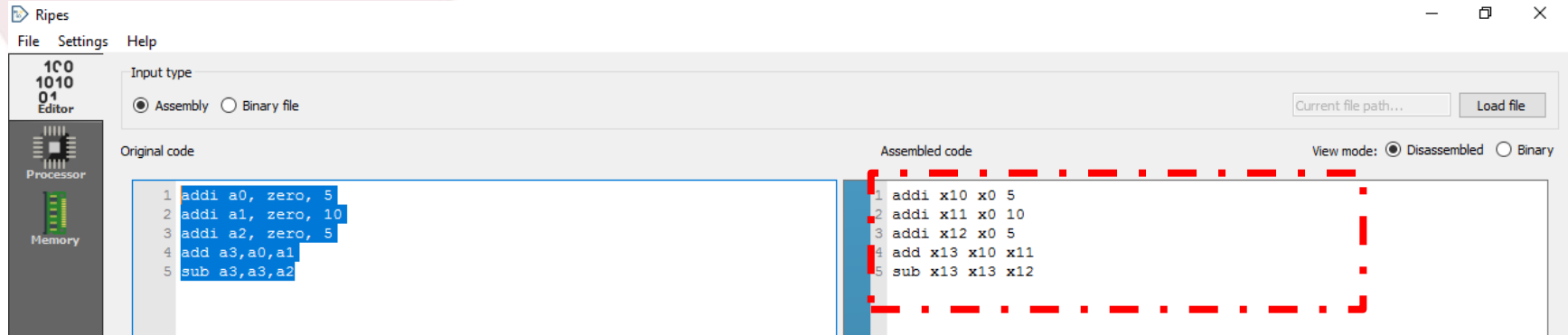
```
addi a0, zero, 5  
addi a1, zero, 10  
addi a2, zero, 5  
add a3,a0,a1  
sub a3,a3,a2
```



Exemplo de Programa em Linguagem de Montagem



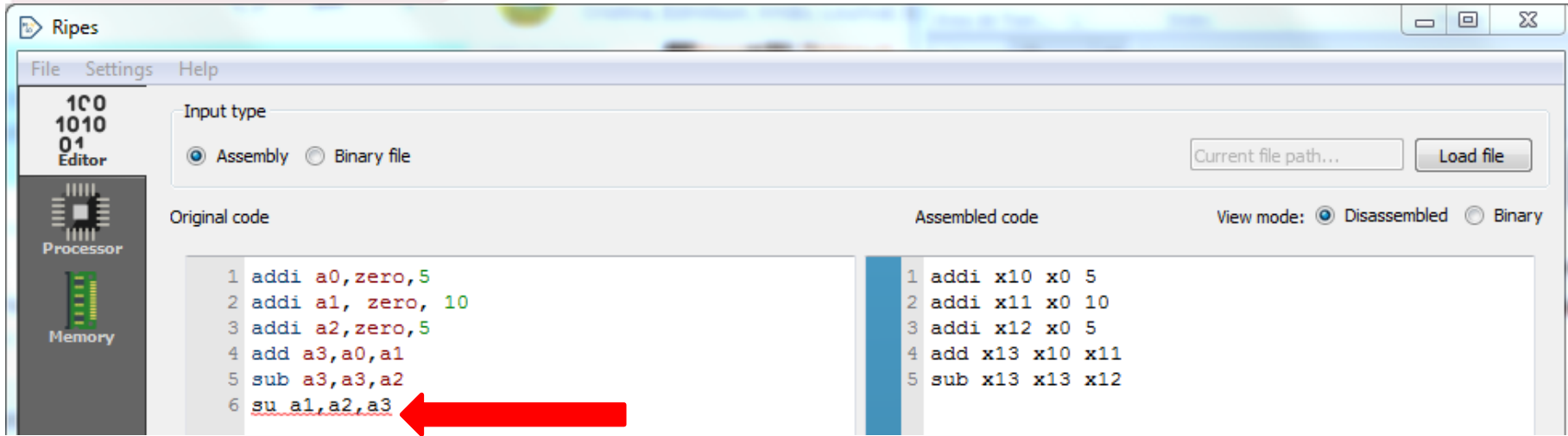
Exemplo de Programa em Linguagem de Montagem



- O código “alto nível” em **Original code** é convertido para um código mais baixo nível em **Assembled code**
 - Converte algumas diretivas para código assembly



Exemplo de Programa em Linguagem de Montagem



- O simulador **não** mostra o que está errado, apenas coloca um tracejado vermelho abaixo da linha que está com algum erro
- Se o código estiver correto ele gera automaticamente a simulação



Exemplo de Programa em Linguagem de Montagem



Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autostepping (F5) Step (F6) Reset (F7) Execution speed: [Slider] ☒ Display all signal values

Registers

a2 x(12) 0

a3 x(13) 0

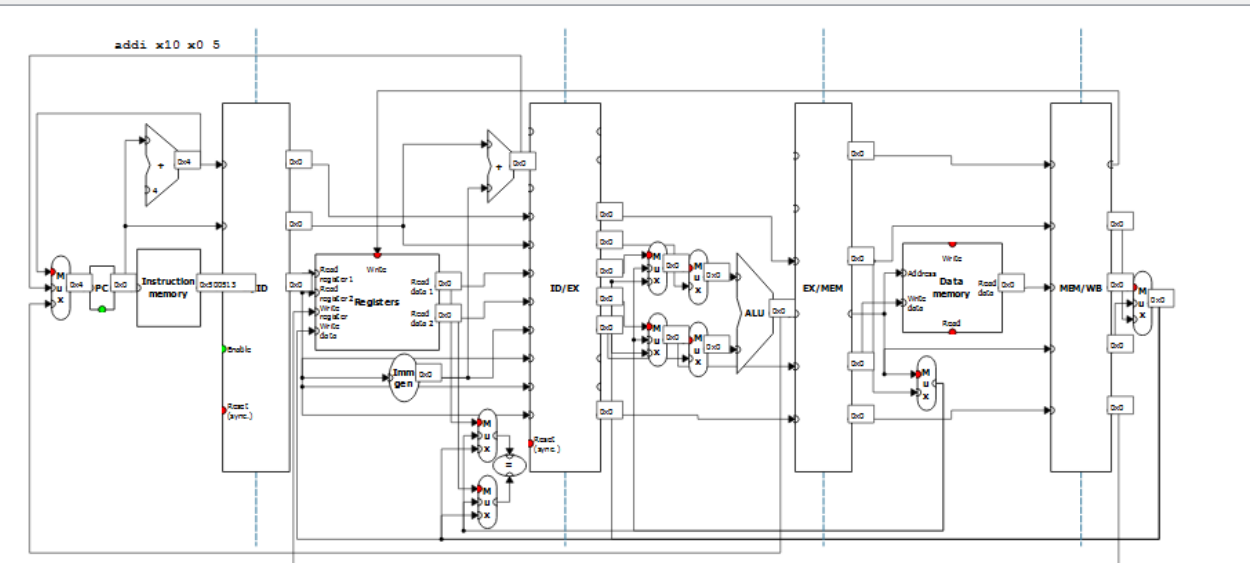
a4 x(14) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	IF	addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Ativa a visualização dos dados na saída de cada registrador

Ripes

File Settings Help

100 1010 01 Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [Slider]

☒ Display all signal values

Registers

a2 x(12) 0

a3 x(13) 0

a4 x(14) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	IF	addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Visualização do conteúdo dos registradores

Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [slider] ☒ Display all signal values

Registers

a2 x(12) 0

a3 x(13) 0

a4 x(14) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	IF	addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

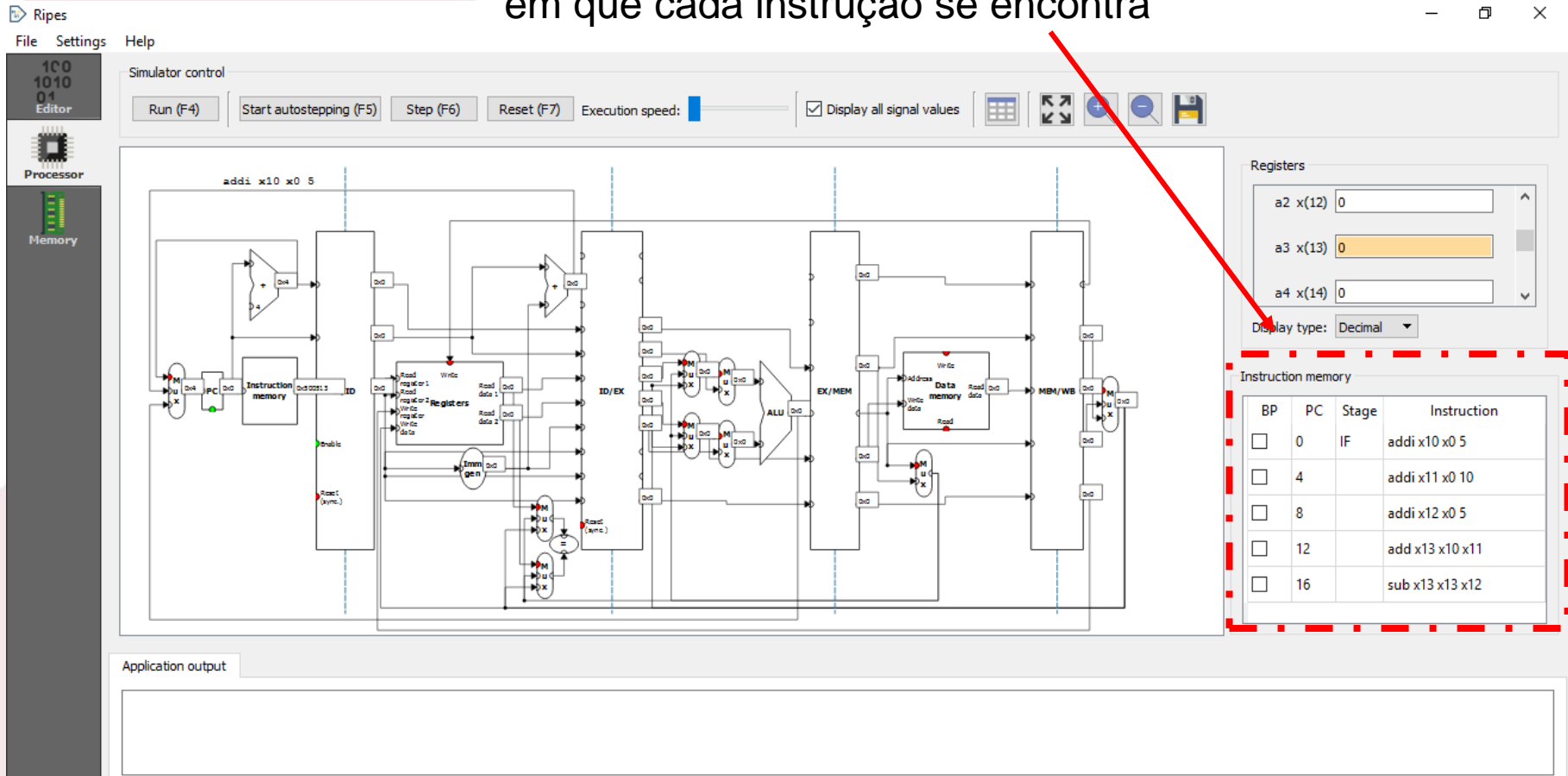
Application output

The screenshot shows the Ripes simulator interface. On the left is a sidebar with icons for the Editor, Processor, and Memory. The main window displays a detailed block diagram of a processor with stages: Instruction memory, ID, EX/MEM, and MB/WB. A red dashed box highlights the 'Registers' panel on the right, which shows registers a2, a3, and a4 all containing the value 0. Below the registers is the 'Instruction memory' panel, which contains a table of instructions. A red arrow points from the text 'Visualização do conteúdo dos registradores' to the registers panel. The 'Simulator control' bar at the top includes buttons for Run, Step, and Reset, along with an execution speed slider and a checkbox for displaying signal values. The 'Application output' area at the bottom is currently empty.

Exemplo de Programa em Linguagem de Montagem



Visualiza a posição na memória e o estágio em que cada instrução se encontra



Exemplo de Programa em Linguagem de Montagem



Execução passo a passo

Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) **Step (F6)** Reset (F7) Execution speed: [Slider] [X] Display all signal values [Icons]

addi x10 x0 5

Registers

a2 x(12) 0

a3 x(13) 0

a4 x(14) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	IF	addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Execução passo a passo

Ripes
File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) **Step (F6)** Reset (F7) Execution speed: [Slider] ☐ Display all signal values

addi x11 x0 10 addi x10 x0 5

Registers

t1 x(6) 0
t2 x(7) 0
s0 x(8) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	ID	addi x10 x0 5
<input type="checkbox"/>	4	IF	addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Execução passo a passo

Ripes
File Settings Help

100 1010 01 Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [Slider] ☐ Display all signal values

addi x12 x0 5 addi x11 x0 10 addi x10 x0 5

Registers

s0 x(8) 0
s1 x(9) 0
a0 x(10) 0
a1 x(11) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	EX	addi x10 x0 5
<input type="checkbox"/>	4	ID	addi x11 x0 10
<input type="checkbox"/>	8	IF	addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Execução passo a passo

Ripes
File Settings Help

100 1010 01 Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [Slider] ☐ Display all signal values

add x13 x10 x11 addi x12 x0 5 addi x11 x0 10 addi x10 x0 5

Registers

s0 x(8) 0
s1 x(9) 0
a0 x(10) 0
a1 x(11) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	MEM	addi x10 x0 5
<input type="checkbox"/>	4	EX	addi x11 x0 10
<input type="checkbox"/>	8	ID	addi x12 x0 5
<input type="checkbox"/>	12	IF	add x13 x10 x11
<input type="checkbox"/>	16		sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Execução passo a passo

Ripes
File Settings Help

1C0
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) **Step (F6)** Reset (F7) Execution speed: [Slider] ☐ Display all signal values [Icons]

sub x13 x13 x12 add x13 x10 x11 addi x12 x0 5 addi x11 x0 10 addi x10 x0 5

Registers

s1 x(9) 0
a0 x(10) 0
a1 x(11) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0	WB	addi x10 x0 5
<input type="checkbox"/>	4	MEM	addi x11 x0 10
<input type="checkbox"/>	8	EX	addi x12 x0 5
<input type="checkbox"/>	12	ID	add x13 x10 x11
<input type="checkbox"/>	16	IF	sub x13 x13 x12

Application output



UNIVERSIDADE FEDERAL
DE PERNAMBUCO

Infra-estrutura Hardware

cin.ufpe.br

Exemplo de Programa em Linguagem de Montagem



Instrução `addi x10 x0 5` terminou
registrador `x10` agora está com o valor 5

Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autostepping (F5) Step (F6) Reset (F7) Execution speed: [slider] ☐ Display all signal values [calculator icon] [full screen icon] [zoom in icon] [zoom out icon] [save icon]

sub x13 x13 x12 add x13 x10 x11 addi x12 x0 5 addi x11 x0 10

Registers

s1 x(9) 0

a0 x(10) 5

a1 x(11) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0		addi x10 x0 5
<input type="checkbox"/>	4	WB	addi x11 x0 10
<input type="checkbox"/>	8	MEM	addi x12 x0 5
<input type="checkbox"/>	12	EX	add x13 x10 x11
<input type="checkbox"/>	16	ID	sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Instrução `addi x11 x0 10` terminou
registrador `x11` agora está com o valor 10 -

Ripes
File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autostepping (F5) Step (F6) Reset (F7) Execution speed: [slider] ☐ Display all signal values

sub x13 x13 x12 sub x13 x13 x12 add x13 x10 x11 addi x12 x0 5

Registers

a0 x(10) 5
a1 x(11) 10
a2 x(12) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0		addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8	WB	addi x12 x0 5
<input type="checkbox"/>	12	MEM	add x13 x10 x11
<input type="checkbox"/>	16	EX/ID	sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Instrução `addi x12 x0 5` terminou
registrador `x12` agora está com o valor 5

Ripes
File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [slider] ☐ Display all signal values [calculator icon] [full screen icon] [zoom in icon] [zoom out icon] [save icon]

sub x13 x13 x12

sub x13 x13 x12

add x13 x10 x11

Registers

a0 x(0) 5
a1 x(11) 10
a2 x(12) 5
a3 x(13) 0

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0		<code>addi x10 x0 5</code>
<input type="checkbox"/>	4		<code>addi x11 x0 10</code>
<input type="checkbox"/>	8		<code>addi x12 x0 5</code>
<input type="checkbox"/>	12	WB	<code>add x13 x10 x11</code>
<input type="checkbox"/>	16	ID/MEM	<code>sub x13 x13 x12</code>

Application output



Exemplo de Programa em Linguagem de Montagem



Instrução add x13 x0 x11 terminou
registrador x13 agora está com o valor 15

Ripes
File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [slider] ☐ Display all signal values

sub x13 x13 x12

sub x13 x13 x12

Registers

a0 x(10) 5
a1 x(11) 10
a2 x(12) 5
a3 x(13) 15

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0		addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16	ID/MEM/WB	sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Instrução sub x13 x13 x12 terminou

registrador x13 agora está com o valor 10

Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Simulator control

Run (F4) Start autosteping (F5) Step (F6) Reset (F7) Execution speed: [slider] ☐ Display all signal values [icons]

sub x13 x13 x12

sub x13 x13 x12

Registers

a0 x(10) 5

a1 x(11) 10

a2 x(12) 5

a3 x(13) 10

Display type: Decimal

Instruction memory

BP	PC	Stage	Instruction
<input type="checkbox"/>	0		addi x10 x0 5
<input type="checkbox"/>	4		addi x11 x0 10
<input type="checkbox"/>	8		addi x12 x0 5
<input type="checkbox"/>	12		add x13 x10 x11
<input type="checkbox"/>	16	ID/MEM	sub x13 x13 x12

Application output



Exemplo de Programa em Linguagem de Montagem



Também mostra os valores dos registradores

Também mostra os valores da memória

Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Registers

sp	x(2)	7ffffff0
gp	x(3)	10000000
tp	x(4)	00000000
t0	x(5)	00000000
t1	x(6)	00000000
t2	x(7)	00000000
s0	x(8)	00000000
s1	x(9)	00000000
a0	x(10)	00000000
a1	x(11)	00000000
a2	x(12)	00000000
a3	x(13)	00000000

Display type: Hex

Memory accesses

Go to selected access address

Cycle	PC	Instruction	Access address
-------	----	-------------	----------------

Memory

Address	+0	+1	+2	+3
0x8000000c	0	0	0	0
0x80000008	0	0	0	0
0x80000004	0	0	0	0
0x80000000	0	0	0	0
0x7ffffffc	0	0	0	0
0x7ffffff8	0	0	0	0
0x7ffffff4	0	0	0	0
0x7ffffff0	0	0	0	0
0x7fffffec	0	0	0	0
0x7ffffe8	0	0	0	0
0x7ffffe4	0	0	0	0
0x7ffffe0	0	0	0	0
0x7ffffdc	0	0	0	0
0x7ffffd8	0	0	0	0
0x7ffffd4	0	0	0	0

Display type: Unsigned Go to: Select Up Down Save Address



Exemplo de Programa em Linguagem de Montagem

Muda para visualizar a memória de dados, pilha ou de instruções

Display type: Decimal ▼

Go to: Select ▼

- Select
- Address...
- Text
- Data
- Stack

Up Down

Muda o tipo de dado a ser visualizado

Outro exemplo...carregando dado da memória e loop

Para carregar dados na memória precisamos utilizar a diretiva `.data`

```
.data
```

```
a: .word 4
```

```
.text
```

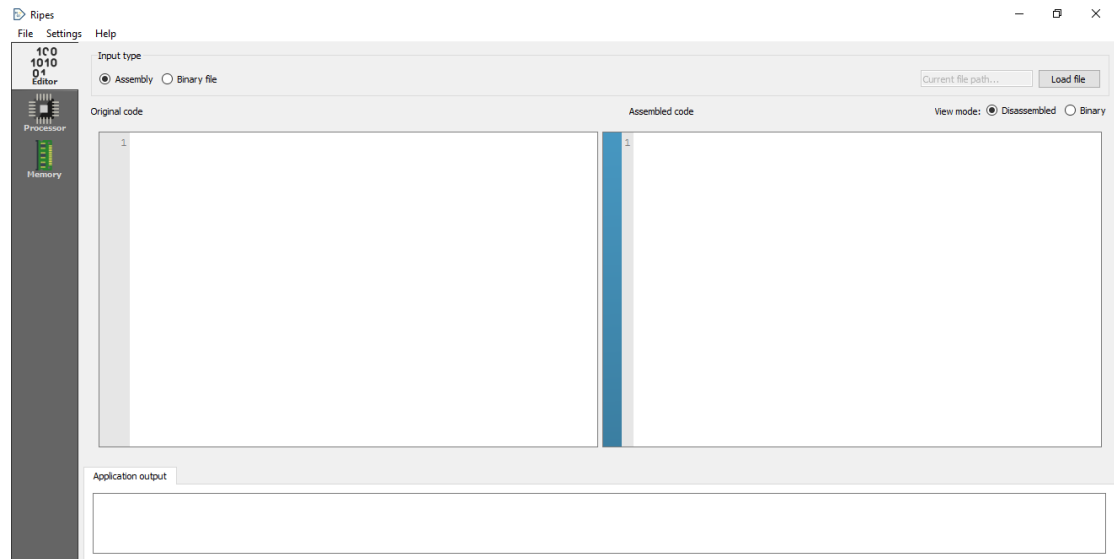
```
lw a1,a
```

```
addi a0, zero, 0
```

```
loop:
```

```
addi a0,a0,1
```

```
bne a0,a1,loop
```



Outro exemplo...carregando dado da memória e loop

Escreva o seguinte código no espaço “original code”

```
.data
a: .word 4
.text
lw a1,a
addi a0, zero, 0
loop:
addi a0,a0,1
bne a0,a1,loop
```



Outro exemplo...carregando dado da memória e loop



Ripes

File Settings Help

100
1010
01
Editor

Processor

Memory

Registers

zero x(0) 0

ra x(1) 0

sp x(2) 2147483632

gp x(3) 268435456

tp x(4) 0

t0 x(5) 0

t1 x(6) 0

t2 x(7) 0

s0 x(8) 0

s1 x(9) 0

a0 x(10) 0

a1 x(11) 0

Display type: Decimal

Memory accesses

Go to selected access address

Cycle PC Instruction Access address

A memória de instrução começa no endereço 0x10000000
Que é o mesmo apontado pelo registrador gp

Memory

Address	+0	+1	+2	+3
0x1000001c	0	0	0	0
0x10000018	0	0	0	0
0x10000014	0	0	0	0
0x10000010	0	0	0	0
0x1000000c	0	0	0	0
0x10000008	0	0	0	0
0x10000004	0	0	0	0
0x10000000	4	0	0	0
0x0ffffffc	0	0	0	0
0x0ffffff8	0	0	0	0
0x0ffffff4	0	0	0	0
0x0ffffff0	0	0	0	0
0x0fffffec	0	0	0	0
0x0fffffe8	0	0	0	0
0x0fffffe4	0	0	0	0

Display type: Unsigned Go to: Select Up Down Save Address

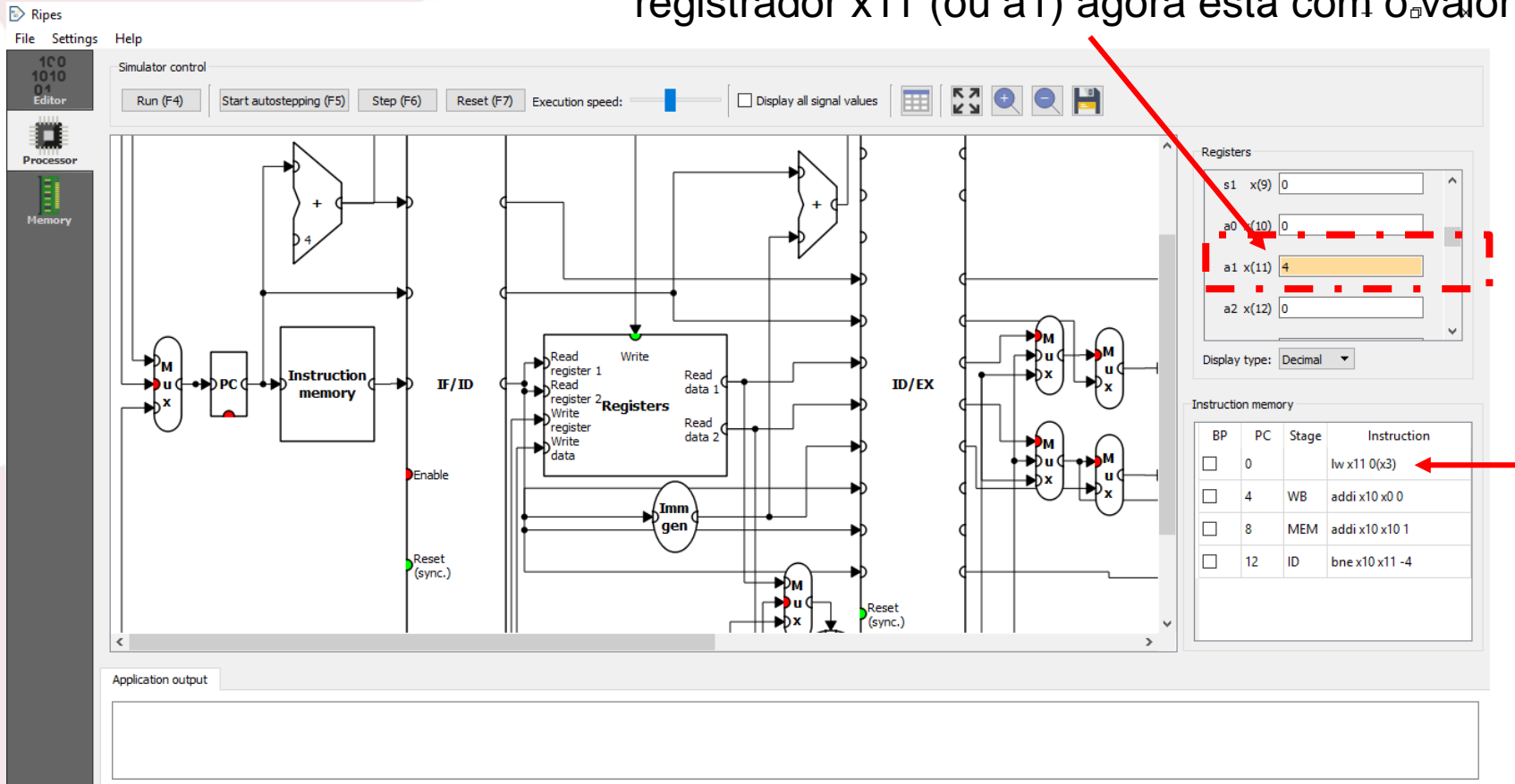
Application output



Outro exemplo...carregando dado da memória e loop



Instrução lw a1,0(gp) terminou
registrador x11 (ou a1) agora está com o valor 4



Outros tipos de dados



- É possível inicializar dados na memória usando diretivas como as presentes aqui
- Por exemplo, para declarar uma variável inteira a:
 - a: .word *valor*
 - String nome:
 - Nome: .string “*meu nome é..*”
 - Nome: .asciz “*meu nome é..*”



Outros tipos de dados



- Por exemplo, para declarar uma variável inteira a:
 - a: .word *valor*
 - Exemplo: a: .word 10
 b: .word 20



Outros tipos de dados



- Para declarar uma string:
 - Nome: `.string` “*meu nome é..*”
 - Nome: `.asciz` “*meu nome é..*”
 - Exemplo:
 - `msg:.string "1-2-3-4-5"`
 - `lb x5, 0(gp) #colocando o valor que esta na memoria`



Outros tipos de dados

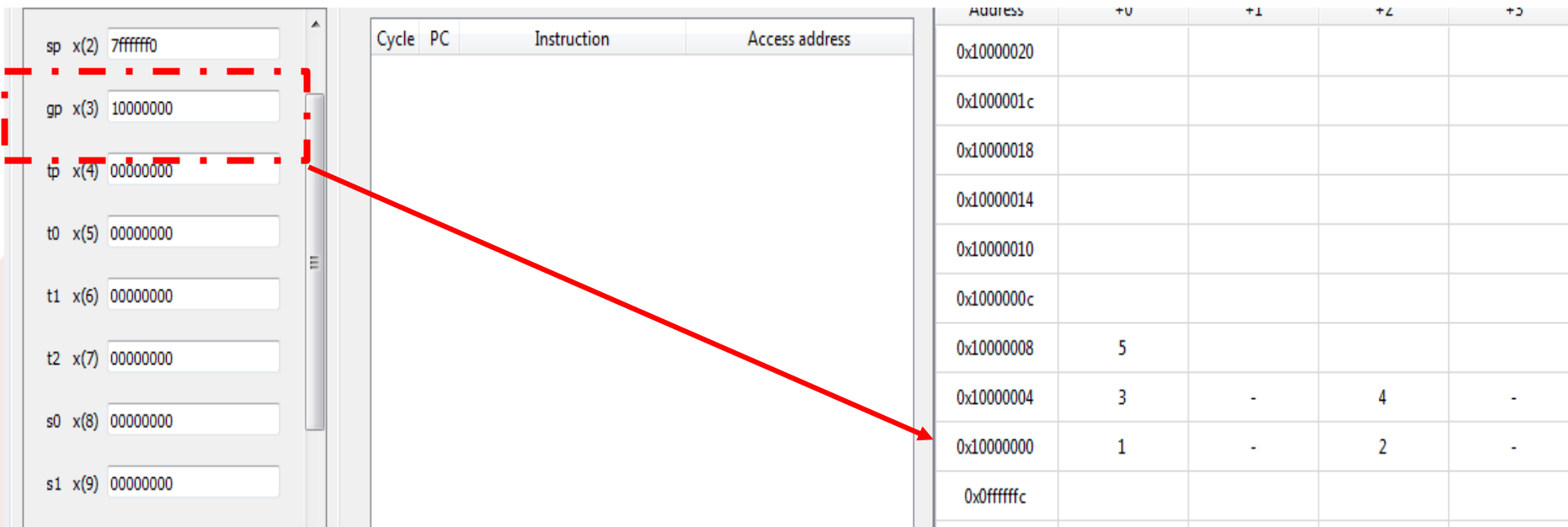


- Escrevam o seguinte código:
- .data
- msg:.string "1-2-3-4-5"
- .text
- lb a2,(0)gp
- lb a3,(1)gp
- lb a4,(2)gp
- lb a5,(3)gp
- lb a6,(4)gp
- lb a7,(5)gp



Outros tipos de dados

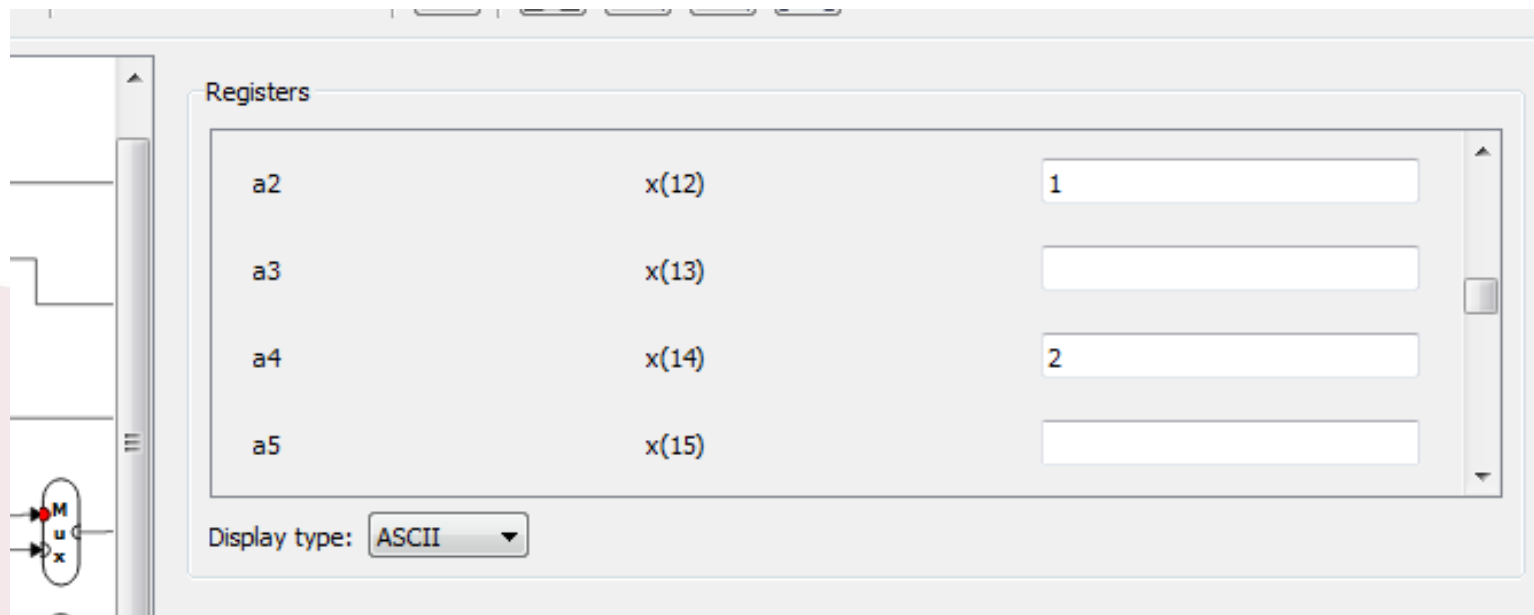
- Observe que o registrador gp aponta para o início da string



	Address	+0	+1	+2	+3
sp x(2)	7fffffff0				
gp x(3)	10000000				
tp x(4)	00000000				
t0 x(5)	00000000				
t1 x(6)	00000000				
t2 x(7)	00000000				
s0 x(8)	00000000				
s1 x(9)	00000000				
	0x10000020				
	0x1000001c				
	0x10000018				
	0x10000014				
	0x10000010				
	0x1000000c				
	0x10000008	5			
	0x10000004	3	-	4	-
	0x10000000	1	-	2	-
	0x0ffffffc				

Outros tipos de dados

- Execute run e verifique os valores da string nos registradores



Obs: caracteres não numéricos não são mostrados corretamente, mas o valor correto está no registrador

Outros tipos de dados

- Outra maneira similar

- .data
- msg:.string "1-2-3-4-5"
- .text
- la a1, msg
- lb a2,(0)a1
- lb a3,(1)a1
- lb a4,(2)a1
- lb a5,(3)a1
- lb a6,(4)a1

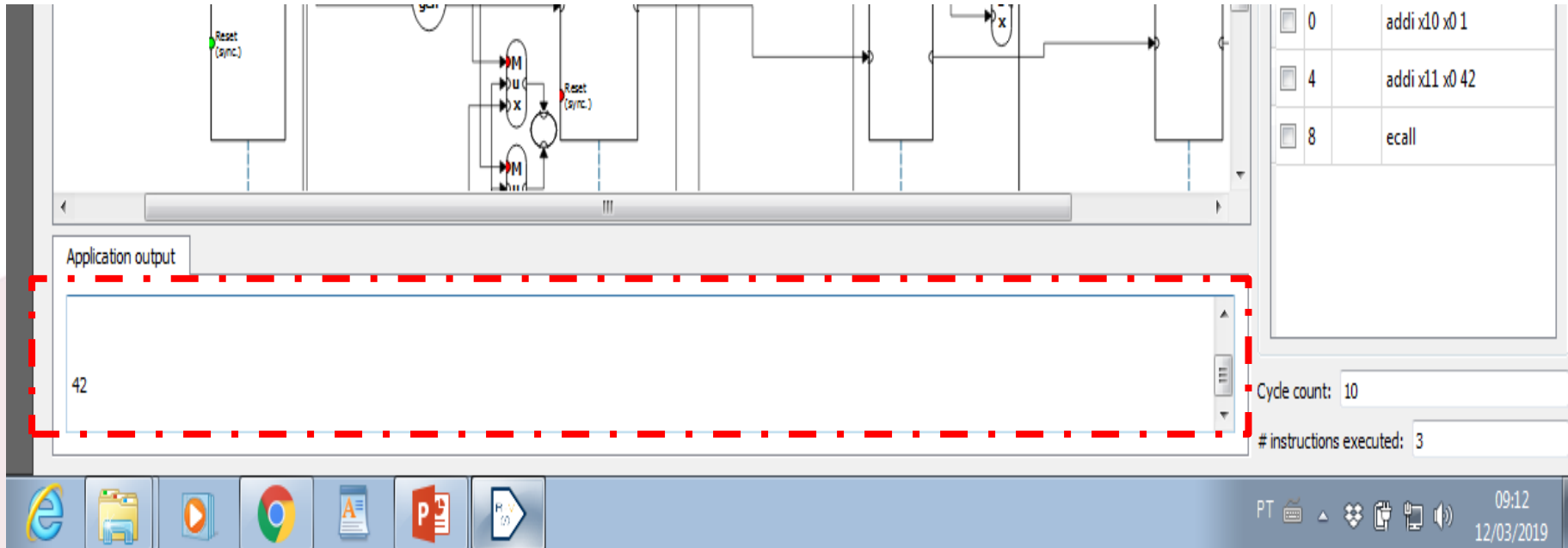
A instrução la

Carrega o endereço base de msg



Console output

- Visualize os dados no console



The screenshot displays a hardware simulation environment. At the top, a logic diagram shows a processor core with various components like registers, ALU, and control logic. Below the diagram is a console window titled "Application output" which contains the number "42". To the right of the console, a table lists instructions and their addresses:


0	addi x10 x0 1
4	addi x11 x0 42
8	ecall

Below the table, the "Cycle count" is 10 and the "# instructions executed" is 3. The bottom of the image shows a Windows taskbar with icons for Edge, File Explorer, VLC, Chrome, Word, PowerPoint, and OneDrive. The system clock in the bottom right corner shows 09:12 on 12/03/2019.

Console output



- Para visualizar:
 - Carregue a ID (1 para inteiro e 4 para string) no registrador a0 e carregue o argumento em a1 – a7.



ID (a0)	Name	Description
1	print_int	prints integer in a1
4	print_string	prints the null-terminated string whose address is in a1
9	sbrk	allocates a1 bytes on the heap, returns pointer to start in a0
10	exit	ends the program
11	print_character	prints ASCII character in a1
17	exit2	ends the program with return code in a1



Console output



- Para visualizar:
 - Em seguida, chame a instrução `ecall` (environment call)



Console output



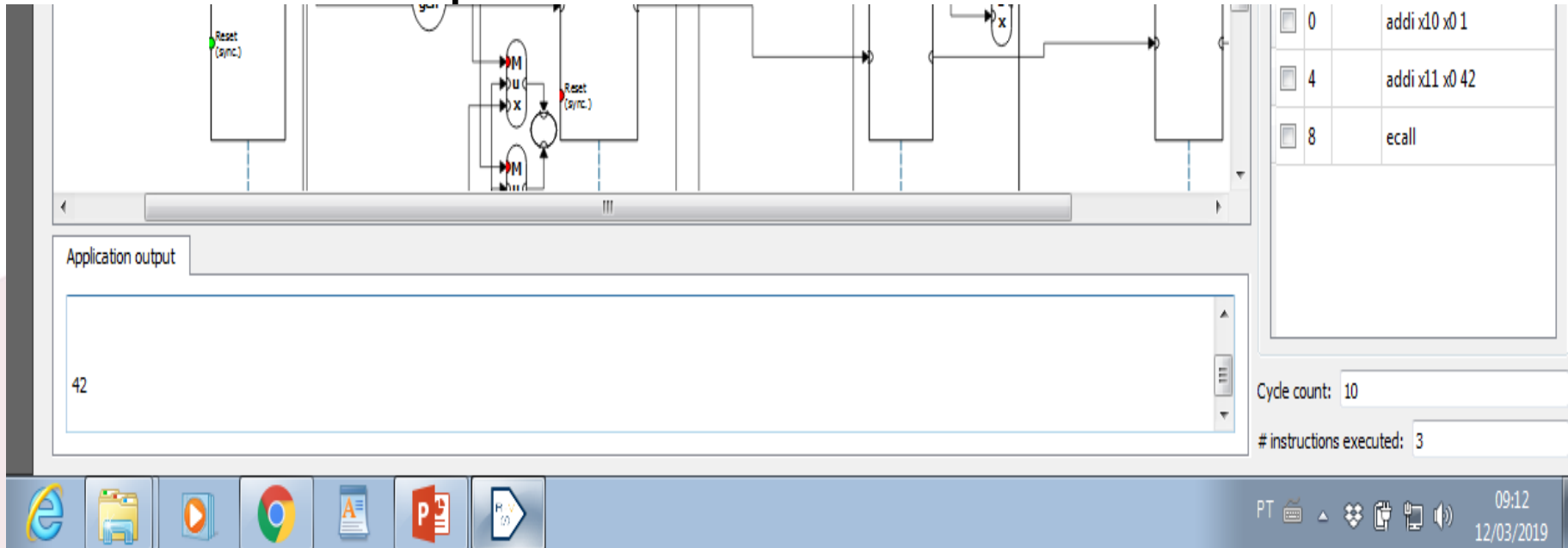
- Exemplo para testar:
 - Escreva este código e em seguida clique em **run**

```
addi a0 x0 1      # print_int ecall
addi a1 x0 42     # integer 42
ecall
```



Console output

- Visualize os dados no console
 - Deverá aparecer o valor 42 no console



The screenshot displays a hardware simulation environment. At the top, a logic diagram shows a processor with two 'Reset (sync.)' buttons. Below the diagram is a large 'Application output' window containing the number '42'. To the right, a table lists instructions: address 0 has 'addi x10 x0 1', address 4 has 'addi x11 x0 42', and address 8 has 'ecall'. Below the table, 'Cycle count' is 10 and '# instructions executed' is 3. The Windows taskbar at the bottom shows icons for Edge, File Explorer, VLC, Chrome, Word, PowerPoint, and OneDrive. The system tray on the right shows the date and time as 09:12 on 12/03/2019.

0	addi x10 x0 1
4	addi x11 x0 42
8	ecall

Application output

42

Cycle count: 10

instructions executed: 3

Console output



- Para visualizar:
 - Mas detalhes sobre environment call:
 - <https://github.com/kvakil/venus/wiki/Environmental-Calls>

ID (a0)	Name	Description
1	print_int	prints integer in a1
4	print_string	prints the null-terminated string whose address is in a1
9	sbrk	allocates a1 bytes on the heap, returns pointer to start in a0
10	exit	ends the program
11	print_character	prints ASCII character in a1
17	exit2	ends the program with return code in a1



Outros tipos de dados

Directive	Arguments	Description
<code>.2byte</code>		16-bit comma separated words (unaligned)
<code>.4byte</code>		32-bit comma separated words (unaligned)
<code>.8byte</code>		64-bit comma separated words (unaligned)
<code>.half</code>		16-bit comma separated words (naturally aligned)
<code>.word</code>		32-bit comma separated words (naturally aligned)
<code>.dword</code>		64-bit comma separated words (naturally aligned)
<code>.byte</code>		8-bit comma separated words

<code>.dtpreldword</code>		64-bit thread local word
<code>.dtprelword</code>		32-bit thread local word
<code>.sleb128</code>	expression	signed little endian base 128, DWARF
<code>.uleb128</code>	expression	unsigned little endian base 128, DWARF
<code>.asciz</code>	"string"	emit string (alias for <code>.string</code>)
<code>.string</code>	"string"	emit string
<code>.incbin</code>	"filename"	emit the included file as a binary sequence of octets
<code>.zero</code>	integer	zero bytes

Outros Modos de Endereçamento

Operandos no RISC-V

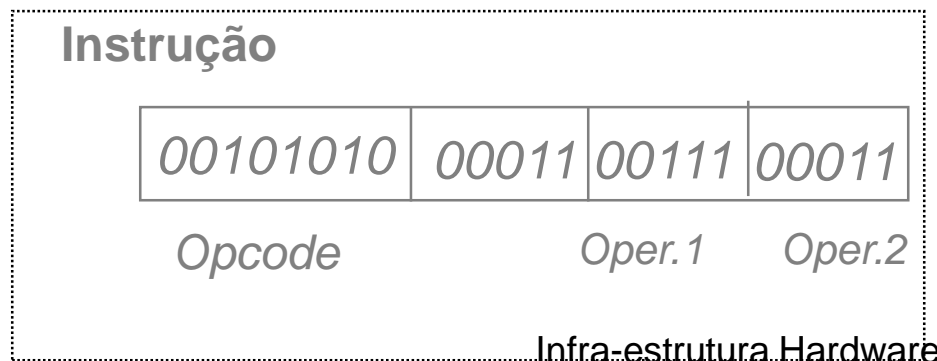


- Aritméticas
 - Registradores
- Load, store
 - Memória



Endereçamento de registrador

- Operações aritméticas:
 - O operando está em um registrador e a instrução contem o número do registrador
 - ADD R3, R3, R7
 - $R3 \leftarrow R3 + R7$

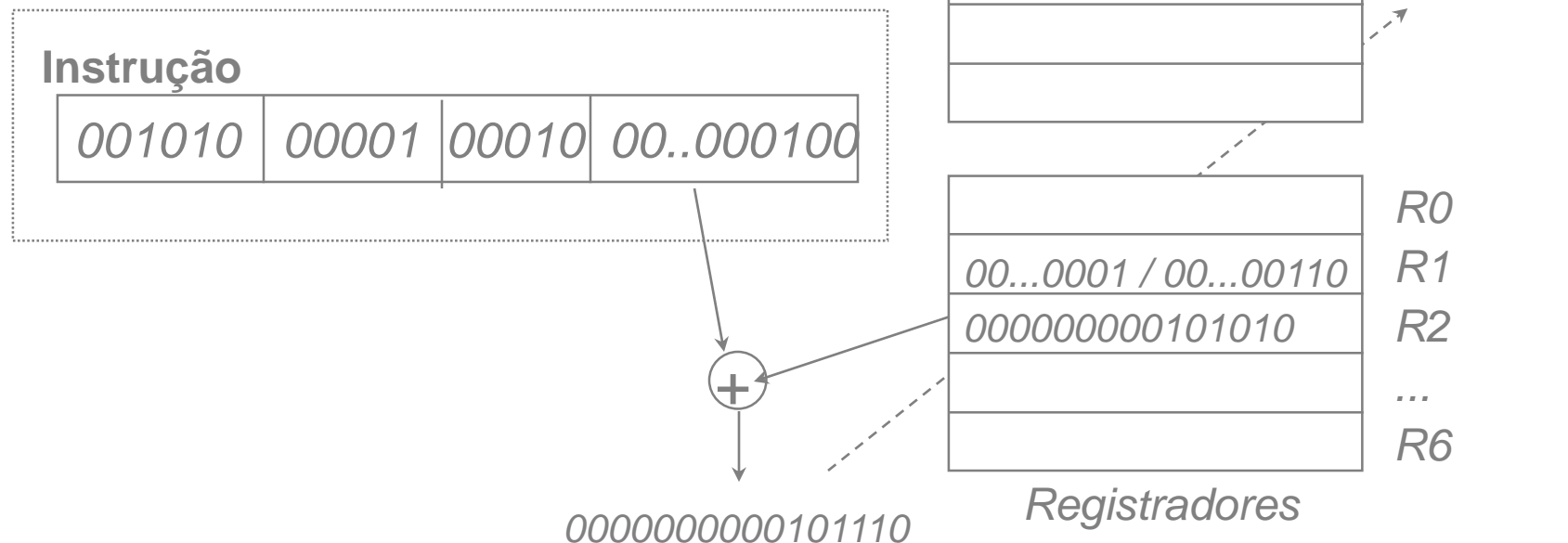


	R0
	R1
	R2
00...0001 / 00...00110	R3
	R4
	R5
	R6
00...000101	R7

Registradores

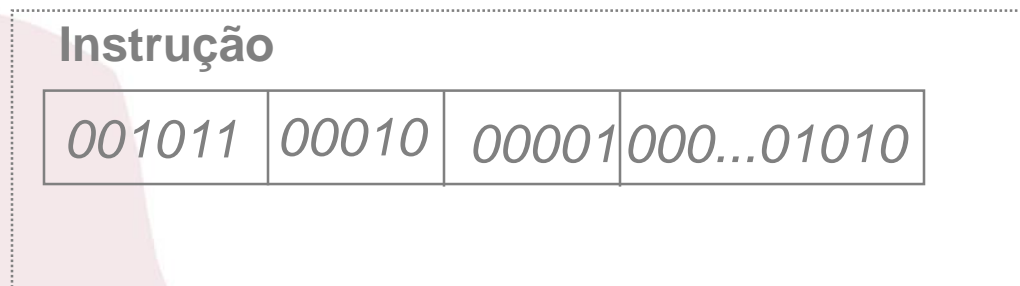
Endereçamento base

- Instruções de acesso à memória:
 - Instrução: deslocamento
 - Registrador de base: end- inicial
 - Lw R1, desl(R2)



Endereçamento imediato

- Operações aritméticas e de comparação:
 - O operando é especificado na instrução
 - ADDI R1, R2, #A



	R0
00...0000 / 00...01010	R1
	R2
	R3
	R4
	R5
	R6
	R7

- Modo bastante frequente
- Operando constante

Endereços no RISC V



- Endereço na Instrução
 - Jal ra, endereço
 - Ra= PC, PC = endereço
- Endereço em Registrador
 - Jalr ra, desl(reg)
 - Ra=PC, PC = desl+(reg)
- Endereço relativo a registrador
 - Beq deslocamento
 - PC = PC + deslocamento*2



Endereçamento (Pseudo)Direto



- Instrução de Desvio:
 - o endereço da próxima instrução é especificado na instrução
 - $Jal\ ra, end1$
 - $Ra = PC\ PC \leftarrow end1$

Instrução

001010	000000000000000000000000101010
--------	--------------------------------

Memória

Add R1, R1, R3
J 000000.....101010

PC=00..101010

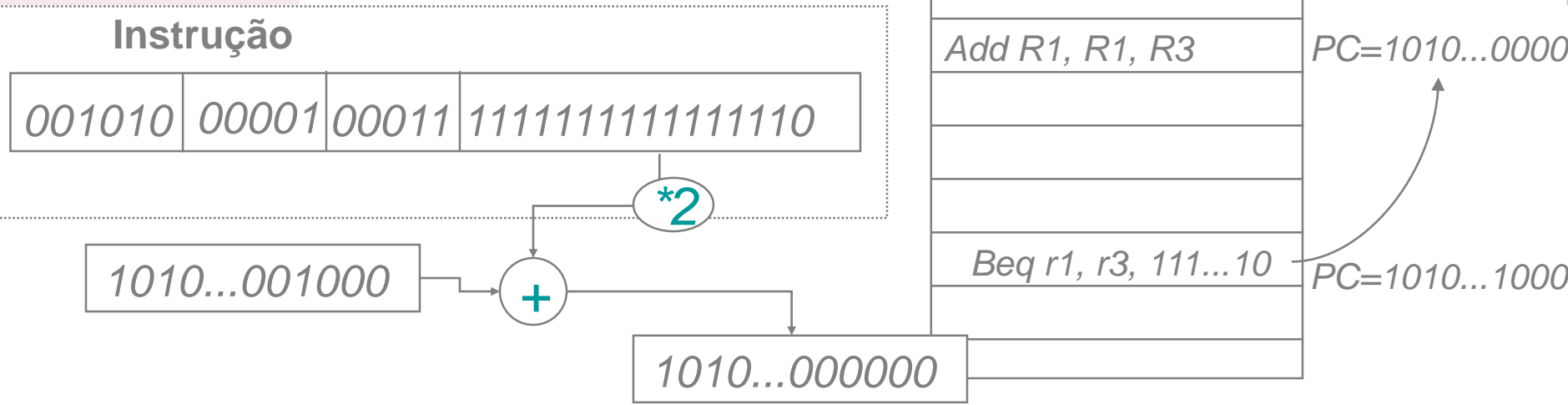
PC=0..101110

Infra-estrutura Hardware

Endereçamento Relativo a PC



- Instrução de Branch:
 - o número de instruções a serem puladas a partir da instrução é especificado na instrução
 - Beq R1, R3, desl1
 - $PC \leftarrow PC + desl1 * 2$



Endereçamento de Registrador

- Instrução de Desvio:
 - o endereço do operando na memória é especificado em um registrador
 - Jalr ra,desl(R1)
 - Ra=PC
 - PC \leftarrow desl+(R1)

Instrução

001010	00001
--------	-------

R1

00000..000000101010

Memória

Add R2, R2, R3
Jr R1

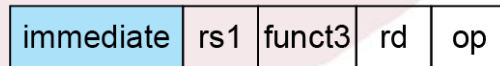
PC=00..101010

Jr R1

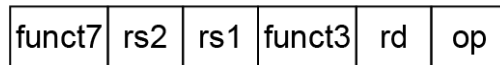
PC=00..101110

RISC-V Endereçamentos

1. Immediate addressing



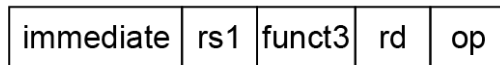
2. Register addressing



Registers

Register

3. Base addressing



Memory

Register

+

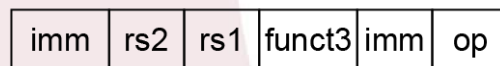
Byte

Halfword

Word

Doubleword

4. PC-relative addressing



Memory

PC

+

Word

Em geral...

- Aritméticas:
 - Operandos em registradores
 - Operandos em memórias
 - ...
- Vários modos de endereçamento

Como especificar na instrução onde está o operando e como este pode ser acessado ?



Campo na Instrução

Endereçamento em outras arquiteturas



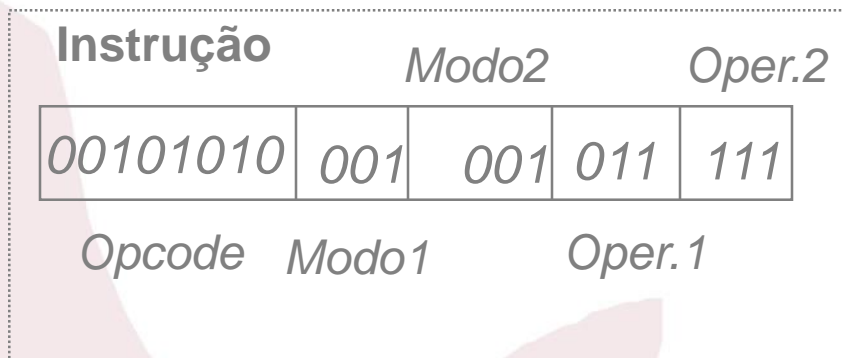
- Aritméticas
 - Registrador-Registrador
 - Registrador - Memória
 - Memória - registrador
- Exemplo: Adição



Endereçamento de registrador



- O operando está em um registrador e a instrução contém o número do registrador
 - ADD R3, R3
 - $R3 \leftarrow R3 + R3$



	R0
	R1
	R2
00...0001 / 00...00110	R3
	R4
	R5
	R6
00...000101	R7

Registradores
cin.ufpe.br



Endereçamento Direto



- O endereço do operando na memória é especificado na instrução
 - ADD R1, end2
 - $R1 \leftarrow R1 + [\text{end2}]$

Registradores

	R0
00...0001 / 00...00010	R1
	R2
	R3
	R4
	R5
	R6
	R7

Memória

00000...0000001	00..101010

Instrução

	Modo1	Modo2	
00101010	001	010	001
0000000000000101010			

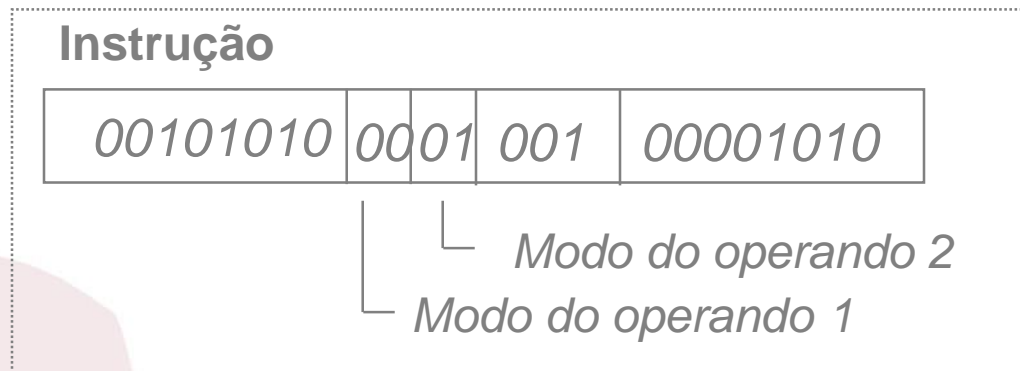
– instruções maiores



Endereçamento imediato



- O operando é especificado na instrução
 - ADD R1, #A



	R0
00...0000 / 00...01010	R1
	R2
	R3
	R4
	R5
	R6
	R7

Registradores

- Modo bastante frequente
- Operando constante



Endereçamento indireto

- O endereço (reg. ou memória) contem o endereço do operando

- ADD R1,(R2)

- $R1 \leftarrow R1 + \text{mem}(R2)$

Instrução

00101010	00	11	001	010
----------	----	----	-----	-----

└─ Modo do operando 2

└─ Modo do operando 1

- Endereçamento variável

- ponteiros

Memória

00000000000000101

00..101010

00...0001 / 00...00110
00000000000101010

R0

R1

R2

R3

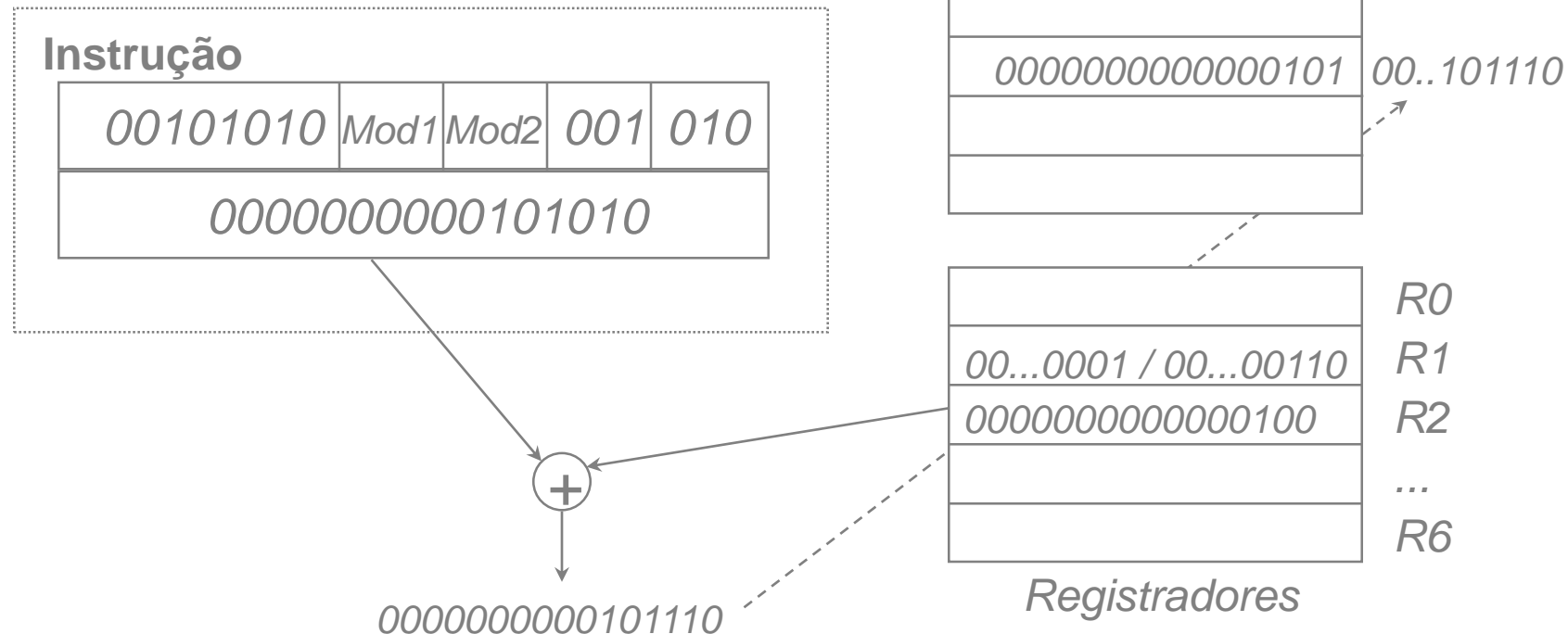
...

Registradores

Endereçamento indexado



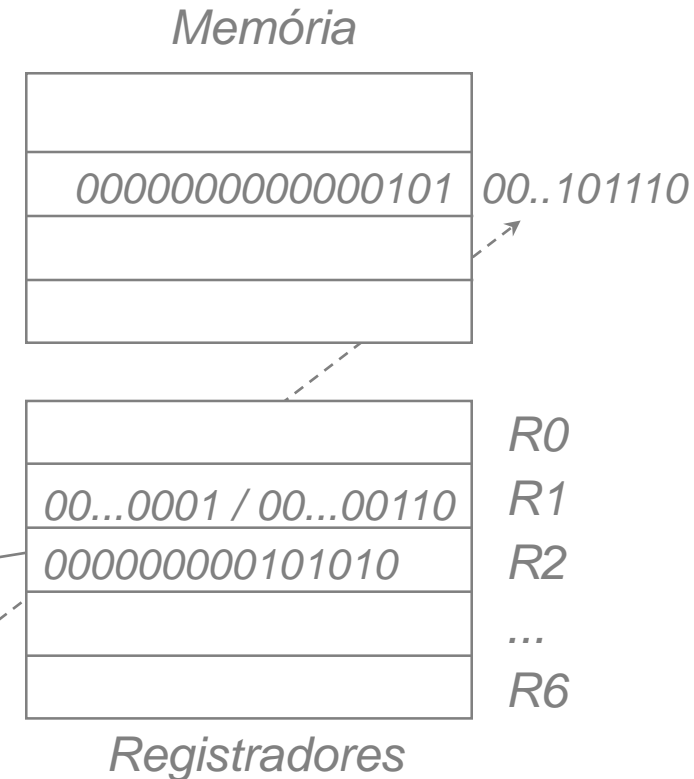
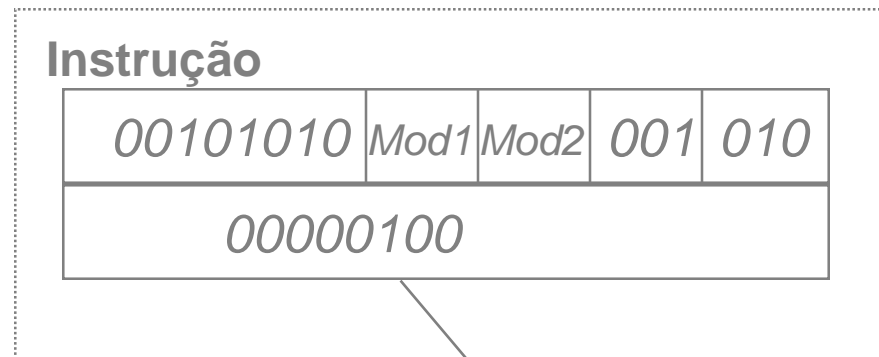
- Instrução: endereço inicial do array
- Registrador de índice: deslocamento
 - ADD R1, [R2]end



Endereçamento base



- Instrução: deslocamento
- Registrador de base: end- inicial
 - ADD R1, desl(R2)



+



0000000000101110

Modos de Endereçamento

Addressing mode	Example	Meaning
Register	Add R4,R3	$R4 \leftarrow R4 + R3$
Immediate	Add R4,#3	$R4 \leftarrow R4 + 3$
Displacement	Add R4,100(R1)	$R4 \leftarrow R4 + \text{Mem}[100 + R1]$
Register indirect	Add R4,(R1)	$R4 \leftarrow R4 + \text{Mem}[R1]$
Indexed / Base	Add R3,(R1+R2)	$R3 \leftarrow R3 + \text{Mem}[R1 + R2]$
Direct or absolute	Add R1,(1001)	$R1 \leftarrow R1 + \text{Mem}[1001]$
Memory indirect	Add R1,@(R3)	$R1 \leftarrow R1 + \text{Mem}[\text{Mem}[R3]]$
Auto-increment	Add R1,(R2)+	$R1 \leftarrow R1 + \text{Mem}[R2]; R2 \leftarrow R2 + d$
Auto-decrement	Add R1,-(R2)	$R2 \leftarrow R2 - d; R1 \leftarrow R1 + \text{Mem}[R2]$
Scaled	Add R1,100(R2)[R3]	$R1 \leftarrow R1 + \text{Mem}[100 + R2 + R3 * d]$

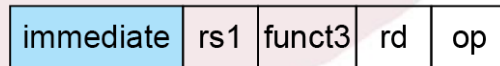
Why Auto-increment/decrement? Scaled?

Endereçamento de desvio

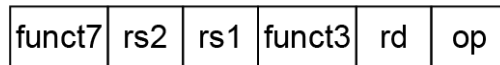
- Especificação do endereço de desvio:
 - Absoluto:
 - ⌘ **PC**  **Endereço de Desvio**
 - restrito a algumas funções do S.O.
 - implícito: vetor de interrupções
 - Relativo (PC = endereço base):
 - ⌘ **PC**  **PC + Deslocamento (instrução)**
 - permite relocação
 - codificação econômica (poucos bits para o deslocamento)

RISC-V Endereçamentos

1. Immediate addressing



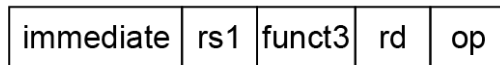
2. Register addressing



Registers

Register

3. Base addressing



Memory

Register

+

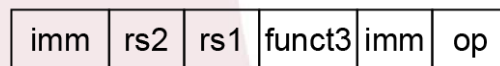
Byte

Halfword

Word

Doubleword

4. PC-relative addressing



Memory

PC

+

Word

Resumindo



- Organização de um computador
- Definição de arquitetura
 - Tipos de Dados
 - Inteiros
 - Booleanos
 - Ponto-Flutuante
 - Formato das instruções
 - Conjunto de registradores



Resumindo



- ...Definição de Arquitetura
 - Repertório de instruções
 - sobre o dado
 - movimentação
 - transformação
 - codificação
 - aritméticas
 - lógicas
 - alteração do fluxo de execução
 - desvios condicionais
 - desvios incondicionais
 - Subrotinas



Resumindo



- ...Definição de Arquitetura
 - Modos de Endereçamento: Dados
 - Operações Aritméticas e de Comparação:
 - Registrador
 - Imediato
 - Load/Store:
 - Base
 - Modos de Endereçamento: Instruções
 - Desvio Incondicional
 - (pseudo) direto
 - Indireto de registrador
 - Desvio Condicional
 - Relativo ao PC
 - Subrotina
 - (pseudo) direto

