Arquitectura de Computadores I

Introdução à arquitectura RISC-V

Miguel Barão

ESCOLA DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Resumo

Arquitectura de Conjunto de Instruções

Instruções Aritméticas

Instruções Lógicas

Código Máquina

Instruções de controlo de fluxo

Instruções de movimento de dados

Carregar números grandes em registos

Arquitectura de Conjunto de Instruções

Arquitectura e implementação

Definição (Arquitectura de conjunto de instruções)

A ISA (*Instruction Set Architecture*) é a descrição funcional de um computador. Essencialmente, é como o software vê o hardware. Consiste na especificação dos registos disponíveis para serem usados num programa, instruções suportadas e respectiva função, etc. Não especifica detalhes de implementação como por exemplo o tamanho da memória cache.

Arquitectura e implementação

Definição (Arquitectura de conjunto de instruções)

A ISA (*Instruction Set Architecture*) é a descrição funcional de um computador. Essencialmente, é como o software vê o hardware. Consiste na especificação dos registos disponíveis para serem usados num programa, instruções suportadas e respectiva função, etc. Não especifica detalhes de implementação como por exemplo o tamanho da memória cache.

Definição (Implementação da arquitectura)

Diz respeito à implementação em hardware da especificação definida na arquitectura (ISA). Podem existir várias implementações diferentes da mesma arquitectura (vários fabricantes produzem CPUs diferentes mas compatíveis entre si)

Exemplos

Arquitectura	Implementação
x86-64 (ou x64)	Intel Core i5-5200U AMD Ryzen Threadripper PRO 3990X Intel Xeon W-2125
AArch64	Apple A7 (usado no iphone 5S) ARM-Cortex A72 (usado no raspberry pi 4)

Exemplos

Arquitectura	Implementação
x86-64 (ou x64)	Intel Core i5-5200U AMD Ryzen Threadripper PRO 3990X Intel Xeon W-2125
AArch64	Apple A7 (usado no iphone 5S) ARM-Cortex A72 (usado no raspberry pi 4)

A mesma arquitectura pode ter diversas implementações com objectivos diferentes:

- Baixo consumo, para dispositivos móveis.
- Grande desempenho, para jogos e computação científica.
- Elevada fiabilidade, para servidores e workstations.

Registos

Um processador inclui:

Registos genéricos Permitem armazenar temporariamente números inteiros e endereços para serem usados nas operações das instruções seguintes. Existe um pequeno conjunto destes (tipicamente até 32).

Registos especializados Têm como função armazenar informação com significados específicos (*e.g.*, *program counter*, *status register*).

Registos internos Usados para as operações do processador, mas não acessíveis pelas instruções (*e.g. instruction register*).

Registos

Um processador inclui:

Registos genéricos Permitem armazenar temporariamente números inteiros e endereços para serem usados nas operações das instruções seguintes. Existe um pequeno conjunto destes (tipicamente até 32).

Registos especializados Têm como função armazenar informação com significados específicos (*e.g.*, *program counter*, *status register*).

Registos internos Usados para as operações do processador, mas não acessíveis pelas instruções (*e.g. instruction register*).

Os registos podem ser acessíveis apenas para leitura, para escrita ou para leitura e escrita.

Processador: registos RISC-V

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

pc

Processador RISC-V

- 32 registos genéricos de 32 bits (`x0..x31`).
- 1 registo especial: program counter (`pc`), contém endereço correspondente à instrução que está em execução.

Exemplo: alguns dos registos RISC-V

- Os registos genéricos têm todos o mesmo comportamento.
- Todos são de leitura e escrita, excepto o `zero`.
- O `zero` tem valor `0x00000000` e não pode ser modificado.

Exemplo: alguns dos registos RISC-V

- Os registos genéricos têm todos o mesmo comportamento.
- Todos são de leitura e escrita, excepto o `zero`.
- O `zero` tem valor `0x00000000` e não pode ser modificado.

A utilização dos registos é definida por convenção (ABI):

`t0 -- t6` valores temporários, podem ser usados livremente.

Exemplo: alguns dos registos RISC-V

- Os registos genéricos têm todos o mesmo comportamento.
- Todos são de leitura e escrita, excepto o `zero`.
- O `zero` tem valor `0x00000000` e não pode ser modificado.

A utilização dos registos é definida por convenção (ABI):

`t0 -- t6` valores temporários, podem ser usados livremente.

`a0 -- a7` usados para passar argumentos para funções.
(iremos estudar mais adiante)

Exemplo: alguns dos registos RISC-V

- Os registos genéricos têm todos o mesmo comportamento.
- Todos são de leitura e escrita, excepto o `zero`.
- O `zero` tem valor `0x00000000` e não pode ser modificado.

A utilização dos registos é definida por convenção (ABI):

`t0 -- t6` valores temporários, podem ser usados livremente.

`a0 -- a7` usados para passar argumentos para funções.
(iremos estudar mais adiante)

`a0 -- a1` usados para retornar o resultado de uma função.
(iremos estudar mais adiante)

Exemplo: alguns dos registos RISC-V

- Os registos genéricos têm todos o mesmo comportamento.
- Todos são de leitura e escrita, excepto o `zero`.
- O `zero` tem valor `0x00000000` e não pode ser modificado.

A utilização dos registos é definida por convenção (ABI):

`t0 -- t6`	valores temporários, podem ser usados livremente.
`a0 -- a7`	usados para passar argumentos para funções. (iremos estudar mais adiante)
`a0 -- a1`	usados para retornar o resultado de uma função. (iremos estudar mais adiante)
`s0 -- s11`	valores temporários, repor após utilização. (iremos estudar mais adiante)

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0xffe28293

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0xffe28293

Pseudocódigo:

Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:

```
z = x + y;
```

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0xffe28293

Pseudocódigo:

Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:

```
z = x + y;
```

Código máquina:

00000000011000101000001110110011

(em hexadecimal: 0x026283b3)

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0xffe28293

Pseudocódigo:
Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
`00000000011000101000001110110011`
(em hexadecimal: `0x026283b3`)

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0x026283b3

Pseudocódigo:
Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
`00000000011000101000001110110011`
(em hexadecimal: `0x026283b3`)

Vamos supor que a instrução está em memória no endereço `0x00400314`.

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0x026283b3

Pseudocódigo:
Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
`00000000011000101000001110110011`
(em hexadecimal: `0x026283b3`)

Ciclo de execução: Fetch → Decode → Execute.

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0x026283b3

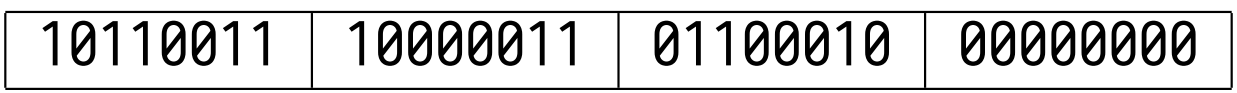
Pseudocódigo:
Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
`00000000011000101000001110110011`
(em hexadecimal: `0x026283b3`)

Fetch (little endian): A instrução fica no *instruction register* (ir)



Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0x026283b3

Pseudocódigo:
Somar t0 e t1 e guardar em t2.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
00000000011000101000001110110011
(em hexadecimal: 0x026283b3)

Decode: A instrução é descodificada

0000000	00110	00101	000	00111	0110011
add	t1	t0	add	t2	add

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0xf2a4c055
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0x026283b3

Pseudocódigo:
Somar t0 e t1 e guardar em t2.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
00000000011000101000001110110011
(em hexadecimal: 0x026283b3)

Execute: A instrução é executada. O CPU soma os valores dos registos t0 e t1 e guarda o resultado no registo t2.

Exemplo: instrução soma dois registos

Reg	Nome	Valor
x0	(zero)	0x00000000
x1	(ra)	0x00400100
⋮		⋮
x5	(t0)	0x00000003
x6	(t1)	0xffffffff
x7	(t2)	0x00000002
x28	(t3)	0x12348765
⋮		⋮
pc		0x00400314
ir		0x026283b3

Pseudocódigo:
Somar `t0` e `t1` e guardar em `t2`.

Linguagem C:
```
z = x + y;
```

Assembly RISC-V:
```
add t2, t0, t1
```

Código máquina:
`00000000011000101000001110110011`
(em hexadecimal: `0x026283b3`)

O que são programas?

Como se constroi um programa?

- Escreve-se um ficheiro numa linguagem de alto nível.
- Converte-se em código máquina com um compilador.
- O ficheiro “executável” contém código máquina e dados.

Como se executa um programa?

- O sistema operativo carrega o programa executável para memória.
- Coloca o endereço da primeira instrução a executar no registo *program counter* (`pc`).

Linguagens de programação de alto e baixo nível

"Alto" e "baixo" nível referem-se ao nível de abstracção do hardware.

Linguagens de programação de alto e baixo nível

"Alto" e "baixo" nível referem-se ao nível de abstracção do hardware.

Linguagens de alto nível:

- Independentes do computador.
- Podem ser compiladas para diferentes arquitecturas.
- Abstracções: arrays, strings, funções, objectos, etc.
- C, C++, Rust, Scheme, Haskell, etc.

Linguagens de programação de alto e baixo nível

"Alto" e "baixo" nível referem-se ao nível de abstracção do hardware.

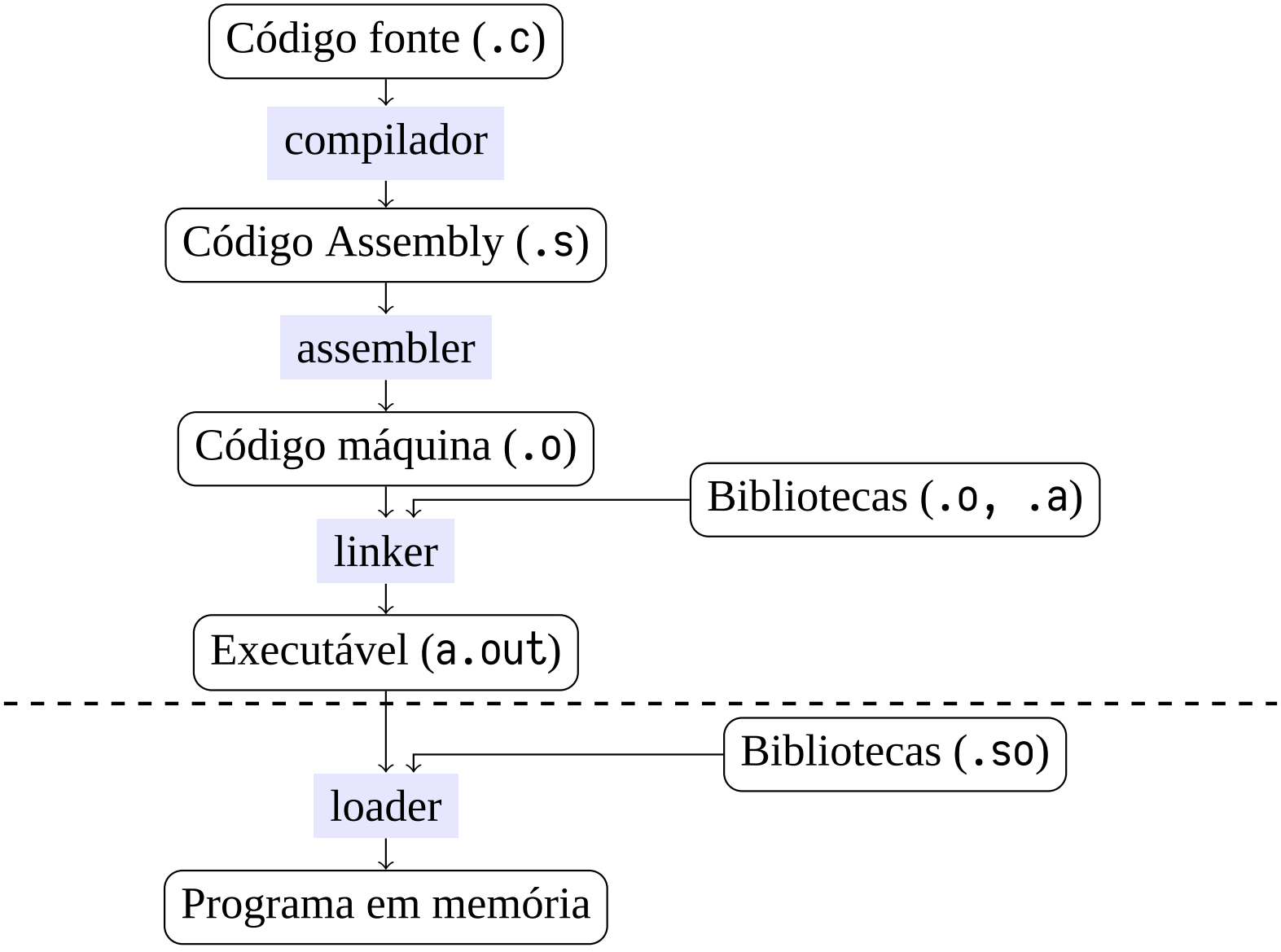
Linguagens de alto nível:

- Independentes do computador.
- Podem ser compiladas para diferentes arquitecturas.
- Abstracções: arrays, strings, funções, objectos, etc.
- C, C++, Rust, Scheme, Haskell, etc.

Linguagens de baixo nível:

- Específicas para cada arquitectura.
- Programas não correm em arquitecturas diferentes.
- Obrigam a lidar com: instruções, registos, endereços, gestão de memória, etc.
- Exemplos: Assembly e Código Máquina para RISC-V, x86-64, AArch, MIPS, SPARC, POWER, etc.

Processo de compilação e execução de um programa



Exemplo de código assembly (1)

Pretende-se multiplicar $y = 3x$.

Exemplo de código assembly (1)

Pretende-se multiplicar $y = 3x$.

Em linguagem C seria

```
y = 3 * x;
```

Exemplo de código assembly (1)

Pretende-se multiplicar $y = 3x$.

Em linguagem C seria

```
y = 3 * x;
```

Em RISC-V, vamos assumir que

- valor de x está inicialmente no registo `t0`,
- valor de y vai ser colocado em `t1`.

Exemplo de código assembly (1)

Pretende-se multiplicar $y = 3x$.

Em linguagem C seria

```
y = 3 * x;
```

Em RISC-V, vamos assumir que

- valor de x está inicialmente no registo `t0`,
- valor de y vai ser colocado em `t1`.

Solução usando a instrução add:

```
1  add t1, t0, t0      # t1 = t0 + t0 = 2t0
2  add t1, t1, t0      # t1 = t1 + t0 = 2t0 + t0 = 3t0
```

Exemplo de código assembly (2)

Pretende-se multiplicar $y = 5x$.

Exemplo de código assembly (2)

Pretende-se multiplicar $y = 5x$.

Solução 1: usando apenas somas:

```
1  add  t1, t0, t0    # t1 = 2t0
2  add  t1, t1, t1    # t1 = 2t0 + 2t0 = 4t0
3  add  t1, t1, t0    # t1 = 4t0 + t0 = 5t0
```

Exemplo de código assembly (2)

Pretende-se multiplicar $y = 5x$.

Solução 1: usando apenas somas:

```
1  add  t1, t0, t0    # t1 = 2t0
2  add  t1, t1, t1    # t1 = 2t0 + 2t0 = 4t0
3  add  t1, t1, t0    # t1 = 4t0 + t0 = 5t0
```

Solução 2: usando somas e deslocamentos

```
1  slli t1, t0, 2     # t1 = 4t0
2  add  t1, t1, t0    # t1 = 4t0 + t0 = 5t0
```

- A instrução `slli` (*shift left logical immediate*) desloca os bits de `t0` duas posições para a esquerda e guarda o resultado em `t1`.
- Cada deslocamento de 1 bit corresponde a multiplicar por 2.

Instruções Aritméticas

Instruções aritméticas *register-register*

Realizam uma operação entre dois registos:

```
1  add   t2, t0, t1      # t2 = t0 + t1
2  sub   t2, t0, t1      # t2 = t0 - t1
3  mul   t2, t0, t1      # t2 = t0 * t1
4  div   t2, t0, t1      # t2 = t0 / t1          (round to 0)
5  rem   t2, t0, t1      # t2 = t0 % t1
6  slt   t2, t0, t1      # t2 = t0 < t1 ? 1 : 0
```

Instruções aritméticas *register-register*

Realizam uma operação entre dois registos:

```
1  add   t2, t0, t1     # t2 = t0 + t1
2  sub   t2, t0, t1     # t2 = t0 - t1
3  mul   t2, t0, t1     # t2 = t0 * t1
4  div   t2, t0, t1     # t2 = t0 / t1          (round to 0)
5  rem   t2, t0, t1     # t2 = t0 % t1
6  slt   t2, t0, t1     # t2 = t0 < t1 ? 1 : 0
```

Nota:
As instruções de multiplicar e dividir `mul, div, rem`, são opcionais.
Fazem parte da extensão RV32M da arquitectura.

Exemplo

Calcular em aritmética inteira

$$z = \frac{x - 2y}{8w}.$$

Em linguagem C:

```
z = (x - 2 * y) / (8 * w);
```

Exemplo

Calcular em aritmética inteira

$$z = \frac{x - 2y}{8w}.$$

Em linguagem C:

```c
z = (x - 2 * y) / (8 * w);
```

Em Assembly, assumindo que os valores x, y, w, z são guardados nos registos `t0,t1,t2,t3`:

```
1  sub  t4, t0, t1    # t4 = x - y
2  sub  t4, t4, t1    # t4 = x - 2y
3  slli t5, t2, 3     # t5 = 8w
4  div  t3, t4, t5    # z = (x-2y) / (8w)
```

mas há várias maneiras diferentes de fazer a mesma coisa!

Instruções aritméticas *register-immediate*

Realizam uma operação entre um registo e um número (*immediate*) codificado no código máquina da instrução:

```
1  addi  t2, t0, -123   # t2 = t0 + (-123)
2  slti  t2, t0, -123   # t2 = t0 < (-123) ? 1 : 0
```

- Código máquina: 32 bits.
- Tamanho dos registos: 32 bits.
- *Immediate*: 12 bits, complemento para 2.

Para somar é necessário fazer extensão de sinal do *immediate*:

- 111111000101 (12 bits)
- 11111111111111111111111111000101 (32 bits)

Exemplo

Calcular em aritmética inteira

$$y = (x + 1)(x - 1).$$

Exemplo

Calcular em aritmética inteira

$$y = (x + 1)(x - 1).$$

Em linguagem C:

```
y = (x + 1) * (x - 1);
```

Exemplo

Calcular em aritmética inteira

$$y = (x + 1)(x - 1).$$

Em linguagem C:

```c
y = (x + 1) * (x - 1);
```

Em Assembly, supondo que x, y são guardados nos registos `t0,t1`:

```
1  addi t2, t0, 1        # t2 = x + 1
2  addi t3, t0, -1       # t3 = x - 1
3  mul  t1, t2, t3       # y = (x + 1)(x - 1)
```

Inteiros *signed* e *unsigned*

Em linguagem C:

```
unsigned int x = 4294967295;  /*     0, . . . , 2^32 − 1   */
signed int y = -1;            /* −2^31, . . . , 2^31 − 1   */
```

em binário são ambos `0b11111111111111111111111111111111`.

Inteiros *signed* e *unsigned*

Em linguagem C:

```
unsigned int x = 4294967295;  /*     0, ..., 2^32 - 1   */
signed int y = -1;            /* -2^31, ..., 2^31 - 1   */
```

em binário são ambos `0b11111111111111111111111111111111`.

Em Assembly:

- não há distinção nos números (são simplesmente 32 bits).
- instruções diferentes para operações *signed* e *unsigned*.

Inteiros *signed* e *unsigned*

Em linguagem C:

```
unsigned int x = 4294967295;  /*     0, ..., 2^32 - 1   */
signed int y = -1;            /* -2^31, ..., 2^31 - 1   */
```

em binário são ambos `0b11111111111111111111111111111111`.

Em Assembly:

- não há distinção nos números (são simplesmente 32 bits).
- instruções diferentes para operações *signed* e *unsigned*.

Signed	*Unsigned*
`div t2, t0, t1`	`divu t2, t0, t1`
`rem t2, t0, t1`	`remu t2, t0, t1`
`slt t2, t0, t1`	`sltu t2, t0, t1`
`slti t2, t0, 123`	`sltiu t2, t0, 123`

Exemplos: *signed vs unsigned*

```
addi  t0, zero, 8    # t0 = 8  = 0x00000008
addi  t1, zero, -2   # t1 = -2 = 0xfffffffe
```

Exemplos: *signed vs unsigned*

```
addi  t0, zero, 8     # t0 = 8  = 0x00000008
addi  t1, zero, -2    # t1 = -2 = 0xfffffffe

div   t2, t0, t1      # t2 = 8 / (-2) = -4
divu  t2, t0, t1      # t2 = 8 / 4294967294 = 0
```

Exemplos: *signed vs unsigned*

```
addi  t0, zero, 8     # t0 = 8  = 0x00000008
addi  t1, zero, -2    # t1 = -2 = 0xfffffffe

div   t2, t0, t1      # t2 = 8 / (-2) = -4
divu  t2, t0, t1      # t2 = 8 / 4294967294 = 0

slti  t3, t1, 4       #          -2 < 4? sim ⇒ t3 = 1
sltiu t3, t1, 4       # 4294967294 < 4? não ⇒ t3 = 0
```

Porque é que não existem versões *unsigned* das instruções add e sub?

Exemplos: *signed vs unsigned*

```
addi  t0, zero, 8     # t0 = 8  = 0x00000008
addi  t1, zero, -2    # t1 = -2 = 0xfffffffe

div   t2, t0, t1      # t2 = 8 / (-2) = -4
divu  t2, t0, t1      # t2 = 8 / 4294967294 = 0

slti  t3, t1, 4       #          -2 < 4? sim ⇒ t3 = 1
sltiu t3, t1, 4       # 4294967294 < 4? não ⇒ t3 = 0
```

Porque é que não existem versões *unsigned* das instruções add e sub?
Porque o resultado seria o mesmo:

```
    00000000000000000000000000001000   (8)
+   11111111111111111111111111111110   (-2)
-------------------------------------
1   00000000000000000000000000000110   (6)
```

Instruções Lógicas

Instruções lógicas (bit-a-bit)

	0	0	1	1
and	0	1	0	1
	0	0	0	1

	0	0	1	1
or	0	1	0	1
	0	1	1	1

	0	0	1	1
xor	0	1	0	1
	0	1	1	0

Instruções lógicas (bit-a-bit)

	0	0	1	1
and	0	1	0	1
	0	0	0	1

	0	0	1	1
or	0	1	0	1
	0	1	1	1

	0	0	1	1
xor	0	1	0	1
	0	1	1	0

Register-Register:

```
1  and t2, t0, t1      # t2 = t0 & t1
2  or  t2, t0, t1      # t2 = t0 | t1
3  xor t2, t0, t1      # t2 = t0 ^ t1
```

Instruções lógicas (bit-a-bit)

	0	0	1	1
and	0	1	0	1
	0	0	0	1

	0	0	1	1
or	0	1	0	1
	0	1	1	1

	0	0	1	1
xor	0	1	0	1
	0	1	1	0

Register-Register:

```
1  and t2, t0, t1      # t2 = t0 & t1
2  or  t2, t0, t1      # t2 = t0 | t1
3  xor t2, t0, t1      # t2 = t0 ^ t1
```

Register-Immediate:

```
1  andi t2, t0, 0x7f0  # t2 = t0 & 0x000007f0
2  ori  t2, t0, 0xff   # t2 = t0 | 0x000000ff
3  xori t2, t0, -1     # t2 = t0 ^ 0xffffffff
```

As instruções lógicas também realizam extensão do sinal.

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1 and t2, t0, t1
```

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1 and t2, t0, t1      # 0b00000000000110100000001000111000
2 or  t3, t0, t1
```

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1  and t2, t0, t1      # 0b00000000000110100000001000111000
2  or  t3, t0, t1      # 0b00010010111111111111011101111100
3  xor t4, t0, t1
```

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1  and t2, t0, t1      # 0b00000000000110100000001000111000
2  or  t3, t0, t1      # 0b00010010111111111111011101111100
3  xor t4, t0, t1      # 0b00010010110010111111001101000100
4
5  andi t2, t2, 0x7f0
```

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1  and t2, t0, t1      # 0b00000000000110100000001000111000
2  or  t3, t0, t1      # 0b00010010111111111111011101111100
3  xor t4, t0, t1      # 0b00010010110010111111001101000100
4
5  andi t2, t2, 0x7f0 # 0b00000000000000000000001000110000
6  ori  t3, t3, 0xff
```

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1  and t2, t0, t1      # 0b00000000000110100000001000111000
2  or  t3, t0, t1      # 0b00010010111111111111011101111100
3  xor t4, t0, t1      # 0b00010010110010111111001101000100
4
5  andi t2, t2, 0x7f0  # 0b00000000000000000000001000110000
6  ori  t3, t3, 0xff   # 0b00010010111111111111011111111111
7  xori t4, t4, -1
```

Exemplo: operações lógicas bit-a-bit

Supondo que

- `t0 = 0x00ffa53c = 0b00000000111111111010010100111100`
- `t1 = 0x12345678 = 0b00010010001101000101011001111000`

qual o resultado de

```
1  and t2, t0, t1      # 0b00000000000110100000001000111000
2  or  t3, t0, t1      # 0b00010010111111111111011101111100
3  xor t4, t0, t1      # 0b00010010110010111111001101000100
4
5  andi t2, t2, 0x7f0  # 0b00000000000000000000001000110000
6  ori  t3, t3, 0xff   # 0b00010010111111111111011111111111
7  xori t4, t4, -1     # 0b11101101001101000000110010111011
```

`t2 = 0x00000430,  t3 = 0x12fff7ff,  t4 = 0xed340cbb`

Set, Reset e *Toggle*

A	B	A & B
0	0	0
0	1	0
1	0	B
1	1	B

A	B	A | B
0	0	B
0	1	B
1	0	1
1	1	1

A	B	A ^ B
0	0	B
0	1	B
1	0	~ B
1	1	~ B

Set, Reset e *Toggle*

A	B	A & B
0	0	0
0	1	0
1	0	B
1	1	B

A	B	A | B
0	0	B
0	1	B
1	0	1
1	1	1

A	B	A ^ B
0	0	B
0	1	B
1	0	~ B
1	1	~ B

Dependendo do bit A,

- AND permite colocar bits a 0 (*Reset*) ou copiar B;
- OR permite colocar bits a 1 (*Set*) ou copiar B;
- XOR permite comutar entre 0 e 1 (*Toggle*) ou copiar B.

Set, Reset e *Toggle*

A	B	A & B
0	0	0
0	1	0
1	0	B
1	1	B

A	B	A | B
0	0	B
0	1	B
1	0	1
1	1	1

A	B	A ^ B
0	0	B
0	1	B
1	0	~ B
1	1	~ B

Dependendo do bit A,

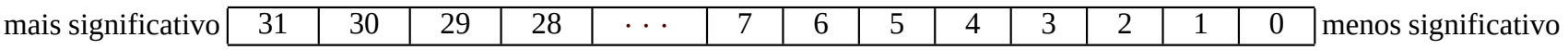
- AND permite colocar bits a 0 (*Reset*) ou copiar B;
- OR permite colocar bits a 1 (*Set*) ou copiar B;
- XOR permite comutar entre 0 e 1 (*Toggle*) ou copiar B.

Conclusão

As operações `and` / `or` / `xor` correspondem a *reset* / *set* / *toggle*.
O bit A controla se a operação é ou não aplicada.

Exemplo: Set / Reset / Toggle

Os 32 bits de um registo estão numerados de 0 a 31:

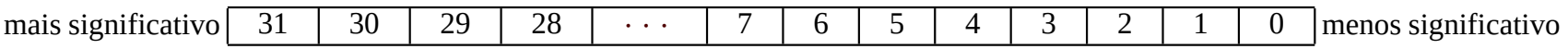


O que fazem as seguintes instruções?

```
1  andi t0, t0, 0xff
```

Exemplo: Set / Reset / Toggle

Os 32 bits de um registo estão numerados de 0 a 31:

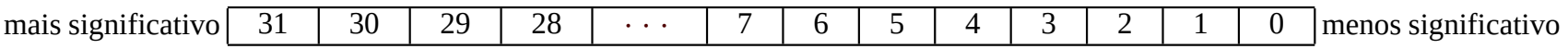


O que fazem as seguintes instruções?

```
1  andi t0, t0, 0xff    # reset de todos os bits excepto
2                       #      os 8 menos significativos.
3
4  ori  t1, t1, 0x8
```

Exemplo: Set / Reset / Toggle

Os 32 bits de um registo estão numerados de 0 a 31:

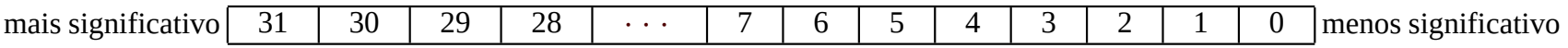


O que fazem as seguintes instruções?

```
1  andi t0, t0, 0xff     # reset de todos os bits excepto
2                        #      os 8 menos significativos.
3
4  ori  t1, t1, 0x8      # set do bit 3, os outros ficam
5                        #                    inalterados.
6
7  xori t2, t2, 1
```

Exemplo: Set / Reset / Toggle

Os 32 bits de um registo estão numerados de 0 a 31:



O que fazem as seguintes instruções?

```
1  andi t0, t0, 0xff     # reset de todos os bits excepto
2                        #      os 8 menos significativos.
3
4  ori  t1, t1, 0x8      # set do bit 3, os outros ficam
5                        #                    inalterados.
6
7  xori t2, t2, 1        # toggle do bit 0, os outros
8                        #                ficam inalterados.
```

Instruções de deslocamento de bits (*shift*)

Deslocam os bits de um registo para a esquerda ou direita.

Instruções *Register-Register*:

```
1  sll  t2, t0, t1     # t2 = t0 << t1
2  srl  t2, t0, t1     # t2 = t0 >>> t1       (zero-extend)
3  sra  t2, t0, t1     # t2 = t0 >> t1        (sign-extend)
```

Instruções de deslocamento de bits (*shift*)

Deslocam os bits de um registo para a esquerda ou direita.

Instruções *Register-Register*:

```
1  sll  t2, t0, t1     # t2 = t0 << t1
2  srl  t2, t0, t1     # t2 = t0 >>> t1       (zero-extend)
3  sra  t2, t0, t1     # t2 = t0 >> t1        (sign-extend)
```

Instruções *Register-Immediate*:

```
1  slli t2, t0, 31     # t2 = t0 << 31
2  srli t2, t0, 31     # t2 = t0 >>> 31       (zero-extend)
3  srai t2, t0, 31     # t2 = t0 >> 31        (sign-extend)
```

- *shift right logical* faz extensão com zeros.
- *shift right arithmetic* faz extensão com o bit de sinal.

Exemplo

Suponha que `t0 = 0x12345678`.

Pretende-se trocar a ordem dos 16 bits da esquerda com os 16 da direita para ficar `0x56781234` no registo `t0`.

Exemplo

Suponha que `t0 = 0x12345678`.

Pretende-se trocar a ordem dos 16 bits da esquerda com os 16 da direita para ficar `0x56781234` no registo `t0`.

Em linguagem C seria:

```
y = (x >> 16) | (x << 16);
```

Exemplo

Suponha que `t0 = 0x12345678`.

Pretende-se trocar a ordem dos 16 bits da esquerda com os 16 da direita para ficar `0x56781234` no registo `t0`.

Em linguagem C seria:

```c
y = (x >> 16) | (x << 16);
```

Em Assembly:

```
1  srli t1, t0, 16      # t1 = 0x00001234
2  slli t2, t0, 16      # t2 = 0x56780000
3  or   t0, t1, t2      # t0 = 0x56781234
```

Resumo das instruções

	Register-Register	*Register-Immediate*
Aritméticas	`add t2, t0, t1`	`addi t2, t0, 1234`
	`sub t2, t0, t1`	
	`mul t2, t0, t1`	
	`divu t2, t0, t1`	
	`remu t2, t0, t1`	
	`sltu t2, t0, t1`	`sltiu t2, t0, 1234`
Lógicas	`and t2, t0, t1`	`andi t2, t0, 0xff`
	`or  t2, t0, t1`	`ori  t2, t0, 0xff`
	`xor t2, t0, t1`	`xori t2, t0, 0xff`
	`sll t2, t0, t1`	`slli t2, t0, 31`
	`srl t2, t0, t1`	`srli t2, t0, 31`
	`sra t2, t0, t1`	`srai t2, t0, 31`

Nenhuma instrução detecta *overflow*!

Código Máquina

Código máquina

Formatos binários das instruções:

Tipo R instruções *register-register*

Tipo I instruções *register-immediate*

Código máquina

Formatos binários das instruções:

Tipo R instruções *register-register*

Tipo I instruções *register-immediate*

- Assembly: `add t2, t0, t1`

funct 7 bits	rs2 5 bits	rs1 5 bits	funct 3 bits	rd 5 bits	opcode 7 bits
0000000	00110	00101	000	00111	0110011

Código máquina: `0x016283b3`

Código máquina

Formatos binários das instruções:

Tipo R instruções *register-register*

Tipo I instruções *register-immediate*

- Assembly: `add t2, t0, t1`

funct 7 bits	rs2 5 bits	rs1 5 bits	funct 3 bits	rd 5 bits	opcode 7 bits
0000000	00110	00101	000	00111	0110011

Código máquina: `0x016283b3`

- Assembly: `addi t2, t0, 1`

imm 12 bits	rs1 5 bits	funct 3 bits	rd 5 bits	opcode 7 bits
000000000001	00101	000	00111	0010011

Código máquina: `0x00128393`

Exercício

0000000	rs2	rs1	000	rd	0110011	add rd, rs1, rs2
0100000	rs2	rs1	000	rd	0110011	sub rd, rs1, rs2
imm		rs1	000	rd	0010011	addi rd, rs1, imm
0000000	shamt	rs1	001	rd	0010011	slli rd, rs1, shamt

Endereço	Byte
0x00400117	0x00
0x00400116	0xc2
0x00400115	0x92
0x00400114	0x93
0x00400103	0xff
0x00400102	0xe2
0x00400101	0x82
0x00400100	0x93

O que faz este programa?

Exercício

0000000	rs2	rs1	000	rd	0110011	add rd, rs1, rs2
0100000	rs2	rs1	000	rd	0110011	sub rd, rs1, rs2
imm		rs1	000	rd	0010011	addi rd, rs1, imm
0000000	shamt	rs1	001	rd	0010011	slli rd, rs1, shamt

Endereço	Byte
0x00400117	0x00
0x00400116	0xc2
0x00400115	0x92
0x00400114	0x93
0x00400103	0xff
0x00400102	0xe2
0x00400101	0x82
0x00400100	0x93

O que faz este programa?

1. Código máquina: `0xffe28293 0x00c29293`
2. Em binário:
   `111111111110 00101 000 00101 0010011`
   `0000000 01100 00101 001 00101 0010011`
3. Em Assembly:
   ```
   addi x5, x5, -2
   slli x5, x5, 12
   ```
   onde o registo `x5` tem nome `t0`.
4. Em linguagem C:
   ```
   n = (n - 2) << 12;
   ```
5. Calcula $4096(n-2)$.

Instruções de controlo de fluxo

Instruções de salto condicional (*branches*)

As instruções de *branch* permitem mudar o rumo de execução de um programa se uma dada condição for satisfeita.

```
beq  t0, t1, label   # branch if t0 == t1
bne  t0, t1, label   # branch if t0 ≠ t1
blt  t0, t1, label   # branch if t0 < t1
bltu t0, t1, label   # branch if t0 < t1 (unsigned)
bge  t0, t1, label   # branch if t0 ≥ t1
bgeu t0, t1, label   # branch if g0 ≥ t1 (unsigned)
```

- A `label` marca uma determinada posição no código (corresponde a um endereço).
- Se a condição é verdadeira, o *branch* salta para essa posição.
- Se a condição é falsa, o *branch* não tem efeito.

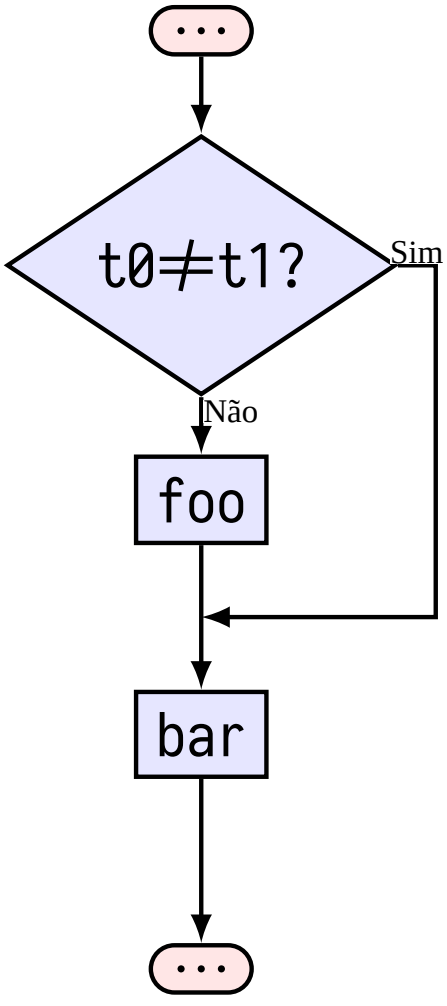
Exemplos de saltos condicionais (*if*)

```
1  if (x == y) {
2      /* foo */
3  }
4  /* bar */
```

Exemplos de saltos condicionais (*if*)

```
1  if (x == y) {
2      /* foo */
3  }
4  /* bar */
```

```
1          bne t0, t1, SALTA
2
3          #
4          # foo
5          #
6
7  SALTA:  #
8          # bar
9          #
```



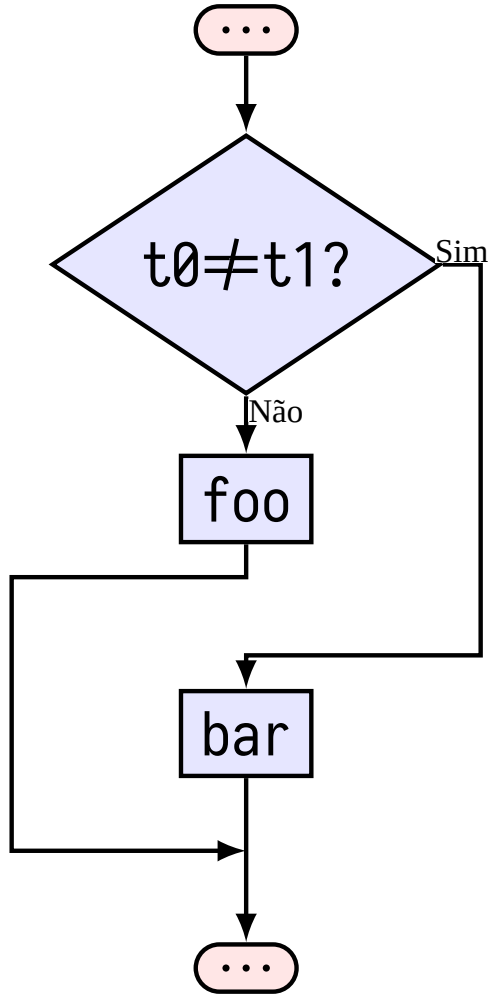
Exemplos de saltos condicionais (*if/else*)

```
1  if (x == y) {
2      /* foo */
3  } else {
4      /* bar */
5  }
```

Exemplos de saltos condicionais (*if/else*)

```c
if (x == y) {
    /* foo */
} else {
    /* bar */
}
```

```
        bne t0, t1, ELSE
        # foo
        beq zero, zero, END
ELSE:   # bar
END:
```



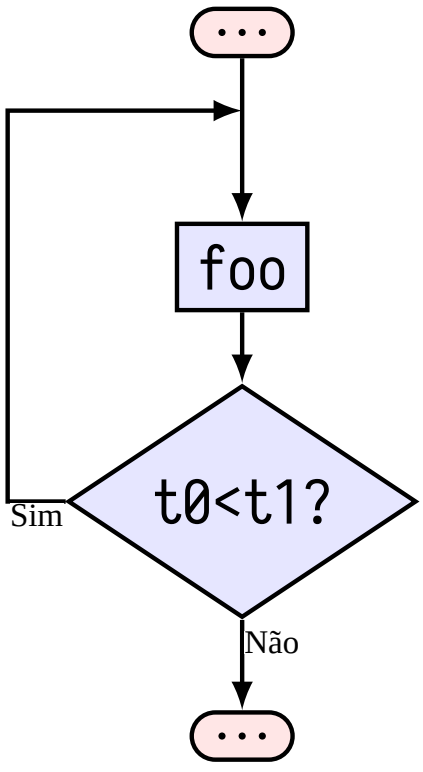
Exemplos de ciclos (*do/while*)

```
1  do {
2      /*
3          foo
4      */
5  } while (x < y);
```

Exemplos de ciclos (*do/while*)

```
1  do {
2      /*
3          foo
4      */
5  } while (x < y);
```

```
1  DO:     #
2          # foo
3          #
4          blt t0, t1, DO
```



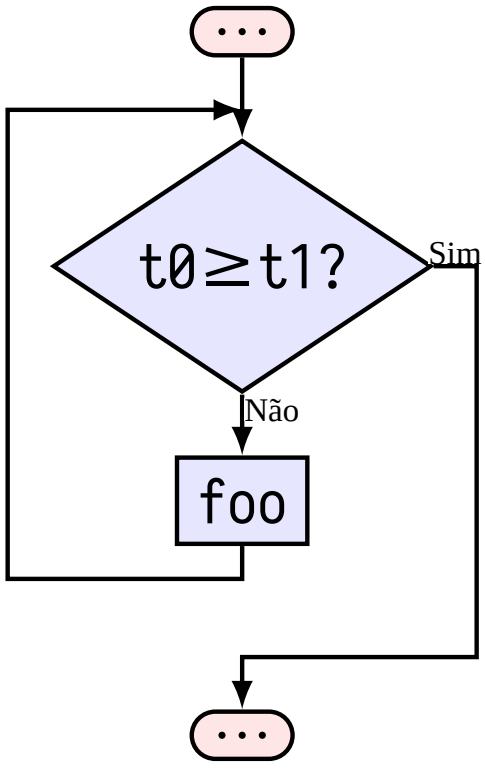
Exemplos de ciclos (*while*)

```
1  while (x < y) {
2      /*
3          foo
4      */
5  }
```

Exemplos de ciclos (*while*)

```
1  while (x < y) {
2      /*
3          foo
4      */
5  }
```

```
1  WHILE:  bge t0, t1, END
2          #
3          # foo
4          #
5          beq zero, zero, WHILE
6  END:
```



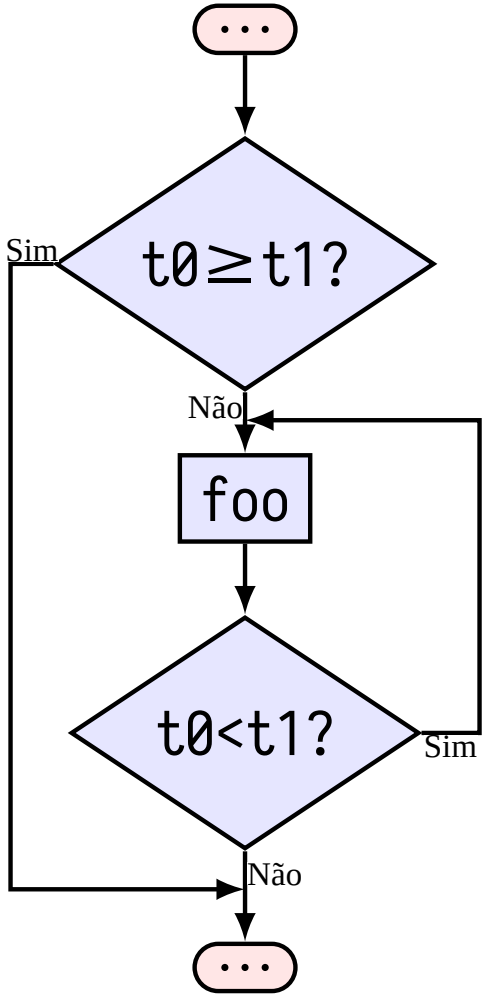
Exemplos de ciclos (*while*) - versão alternativa

```
1  if (x < y) {
2      do {
3        /*
4            foo
5        */
6      } while (x < y);
7  }
```

Exemplos de ciclos (*while*) - versão alternativa

```c
if (x < y) {
    do {
      /*
          foo
      */
    } while (x < y);
}
```

```
        bge t0, t1, END
WHILE:  #
        # foo
        #
        blt t0, t1, WHILE
END:
```



Instruções de salto incondicional (*jumps*)

Existem duas instruções de salto incondicional:

```
1  jal  t1, label     # t1 = PC + 4; PC += imm
2  jalr t1, imm(t0)   # t1 = PC + 4; PC = (t0 + imm) & (-2)
```

- A instrução `jal` (*jump and link*) guarda o endereço da instrução seguinte num registo e salta para posição indicada na `label`.

Instruções de salto incondicional (*jumps*)

Existem duas instruções de salto incondicional:

```
1  jal  t1, label     # t1 = PC + 4; PC += imm
2  jalr t1, imm(t0)   # t1 = PC + 4; PC = (t0 + imm) & (-2)
```

- A instrução `jal` (*jump and link*) guarda o endereço da instrução seguinte num registo e salta para posição indicada na `label`.
- A instrução `jalr` (*jump and link register*) guarda o endereço da instrução seguinte num registo e salta para o endereço de memória calculado por `t0` mais o deslocamento `imm`.
  O endereço de destino tem de ser múltiplo de 4.

Instruções de salto incondicional (*jumps*)

Existem duas instruções de salto incondicional:

```
1  jal  t1, label     # t1 = PC + 4; PC += imm
2  jalr t1, imm(t0)   # t1 = PC + 4; PC = (t0 + imm) & (-2)
```

- A instrução `jal` (*jump and link*) guarda o endereço da instrução seguinte num registo e salta para posição indicada na `label`.
- A instrução `jalr` (*jump and link register*) guarda o endereço da instrução seguinte num registo e salta para o endereço de memória calculado por `t0` mais o deslocamento `imm`.
  O endereço de destino tem de ser múltiplo de 4.
- O endereço guardado no registo (*e.g.* `t1`) permite saber a origem do salto, para mais tarde retomar a execução nesse ponto.
- Se não quisermos guardar a origem do salto, usamos o `zero`.

Exemplos de saltos incondicionais

As instruções `jal` e `jalr` são normalmente usadas para "ida-e-volta".

O que faz o seguinte código?

```
1          # ...
2
3          addi a0, zero, 3
4          jal  ra, xpto      # endereço instr (5) -> ra
5          addi s0, a0, -1
6
7          # ...
8
9  xpto:   slli a0, a0, 1
10         addi a0, a0, 1
11         jalr zero, 0(ra)   # salta para linha (5)
```

Exemplos de saltos incondicionais

O que faz o seguinte código?

```
1          addi a0, zero, 1
2          jal ra, xpto
3          add s0, a0, zero
4
5          addi a0, zero, 2
6          jal ra, xpto
7          sub s0, s0, a0
8
9          # ...
10
11 xpto:   slli a0, a0, 1
12         addi a0, a0, 1
13         jalr zero, 0(ra)
```

Exemplos de saltos incondicionais

O que faz o seguinte código?

```
1           addi a0, zero, 1    # a0 = 1
2           jal ra, xpto        # jump xpto. save pc+4 -> ra
3           add s0, a0, zero    # s0 = a0 = 3
4
5           addi a0, zero, 2    # a0 = 2
6           jal ra, xpto        # jump xpto. save pc+4 -> ra
7           sub s0, s0, a0      # s0 = 3 - a0 = 3 - 5 = -2
8
9           # qual o valor em s0 neste ponto?
10
11  xpto:   slli a0, a0, 1      # 2*a0
12          addi a0, a0, 1      # 2*a0 + 1
13          jalr zero, 0(ra)    # return 2*a0 + 1
```

Instruções de movimento de dados

Instruções de movimento de dados (*load/store*)

- O processador necessita de aceder aos dados do programa.
- Em RISC-V os acessos são feitos por instruções de *load* e *store*.

```
lw t1, 0(t0)        # load word       (32bit)
lh t1, 0(t0)        # load half-word  (16bit)
lb t1, 0(t0)        # load byte       (8bit)

sw t1, 0(t0)        # store word      (32bit)
sh t1, 0(t0)        # store half-word (16bit)
sb t1, 0(t0)        # store byte      (8bit)
```

O endereço é calculado somando o deslocamento `0` ao valor de `t0`.
A ordenação de bytes é *little endian*.

Exemplos

```
lw t1, 4(t0)        # t1 = mem[t0 + 4]
                    # acede à posição t0+4 e
                    # le uma word (4 bytes) para t1

sw t1, -8(t0)       # mem[t0 - 8] = t1
                    # acede à posição t0 - 8 e
                    # guarda a word (4 bytes) do t1

lw t1, 3(zero)      # erro! endereço não é múltiplo de 4
lh t1, 1(zero)      # erro! endereço não é múltiplo de 2
lb t1, 1(zero)      # ok! load byte não tem restrições
```

O deslocamento (*offset*) é um número de 12 bits com sinal.
O endereço final tem de estar "alinhado" (*words*, *half-words*).

Exemplo: reordenação de bytes (versão ineficiente)

Suponha que em memória está um número 32 bits *big endian*.
Para ser válido em RISC-V, tem de se reordenar para *little endian*.

```
1   lw t0, 0(t3)       # le número big endian -> t0
2
3   sb t0, 3(t3)       # guarda o LSB de t0 como MSB
4
5   srli t0, t0, 8     # LSB passa a ser o byte 1
6   sb t0, 2(t3)       # guarda byte 1
7
8   srli t0, t0, 8     # LSB passa a ser o byte 2
9   sb t0, 1(t3)       # guarda o byte 2
10
11  srli t0, t0, 8     # LSB passa a ser o byte 3
12  sb t0, 0(t3)       # guarda o byte 3
```

Extensão de sinal/zeros

As instruções `lh` e `lb` lêem menos de 32 bits $\Rightarrow$ necessário estender.

Extensão com bit de sinal:

```
lh t1, 0(t0)        # load half-word
lb t1, 0(t0)        # load byte
```

Extensão com zeros:

```
lhu t1, 0(t0)       # load half-word unsigned
lbu t1, 0(t0)       # load byte unsigned
```

As instruções `sh` e `sb` não têm este problema.
Guardam em memória a parte menos significativa dos 32 bits e ignoram o resto do registo.

Exemplos (com rasteiras...)

Na posição de memória apontada por `t0` está o número `0xff12aa99`.

Que valores vão ficar nos registos e em memória?

```
lb  t1, 0(t0)
```

Exemplos (com rasteiras...)

Na posição de memória apontada por `t0` está o número `0xff12aa99`.

Que valores vão ficar nos registos e em memória?

```
lb  t1, 0(t0) # t1 = 0xffffff99

lh  t2, 3(t0)
```

Exemplos (com rasteiras...)

Na posição de memória apontada por `t0` está o número `0xff12aa99`.

Que valores vão ficar nos registos e em memória?

```
lb  t1, 0(t0) # t1 = 0xffffff99

lh  t2, 3(t0) # runtime error: endereço não alinhado!

lbu t3, 3(t0)
```

Exemplos (com rasteiras...)

Na posição de memória apontada por `t0` está o número `0xff12aa99`.

Que valores vão ficar nos registos e em memória?

```
lb  t1, 0(t0) # t1 = 0xffffff99

lh  t2, 3(t0) # runtime error: endereço não alinhado!

lbu t3, 3(t0) # t3 = 0x000000ff

sb  t3, 1(t0)
```

Exemplos (com rasteiras...)

Na posição de memória apontada por `t0` está o número `0xff12aa99`.

Que valores vão ficar nos registos e em memória?

```
lb  t1, 0(t0) # t1 = 0xffffff99

lh  t2, 3(t0) # runtime error: endereço não alinhado!

lbu t3, 3(t0) # t3 = 0x000000ff

sb  t3, 1(t0) # mem_word[t0] = 0xff12ff99

lwu t4, 4(t0)
```

Exemplos (com rasteiras...)

Na posição de memória apontada por `t0` está o número `0xff12aa99`.

Que valores vão ficar nos registos e em memória?

```
lb  t1, 0(t0) # t1 = 0xffffff99

lh  t2, 3(t0) # runtime error: endereço não alinhado!

lbu t3, 3(t0) # t3 = 0x000000ff

sb  t3, 1(t0) # mem_word[t0] = 0xff12ff99

lwu t4, 4(t0) # syntax error: a instrução não existe!!!
```

Carregar números grandes em registos

Carregar números grandes em registos

Números até 12 bits podem ser carregados num registo com uma única instrução:

```
addi t0, zero, 2047
```

Como se carregam números maiores?

Carregar números grandes em registos

Números até 12 bits podem ser carregados num registo com uma única instrução:

```
addi t0, zero, 2047
```

Como se carregam números maiores? Usam-se as instruções `lui` (*load upper immediate*) e `auipc` (*add upper immediate to pc*)

```
lui   t0, 0x12345   # t0 = 0x12345000
auipc t0, 0x12345   # t0 = pc + 0x12345000
```

- `lui` é usada para números arbitrários.
- `auipc` é usada para posições no código (endereços).

Exemplos

Carregar `0x12345678` em `t0`:

```
lui   t0, 0x12345    # t0 = 0x12345000
addi  t0, t0, 0x678  # t0 = 0x12345000 + 0x00000678
```

Exemplos

Carregar `0x12345678` em `t0`:

```
lui   t0, 0x12345    # t0 = 0x12345000
addi  t0, t0, 0x678  # t0 = 0x12345000 + 0x00000678
```

Carregar `0xdeadbeef` em `t0`:

Exemplos

Carregar `0x12345678` em `t0`:

```
lui   t0, 0x12345    # t0 = 0x12345000
addi  t0, t0, 0x678  # t0 = 0x12345000 + 0x00000678
```

Carregar `0xdeadbeef` em `t0`:

```
lui   t0, 0xdeadb
addi  t0, t0, 0xeef  # assembler error: out-of-range
```

O número de 12 bits na instrução `addi` é interpretado pelo assembler como um número com sinal.

Mas o maior número de 12 bits com sinal é `0x7ff`...

Exemplo: 0xdeadbeef

Como 0xeef tem o bit de sinal a 1, escreve-se como sendo negativo. O complemento para 2 de 0xeef, é 0x111, portanto escrevemos

```
lui   t0, 0xdeadb
addi  t0, t0, -0x111  # tem os mesmos 12 bits que 0xeef
```

Exemplo: 0xdeadbeef

Como 0xeef tem o bit de sinal a 1, escreve-se como sendo negativo. O complemento para 2 de 0xeef, é 0x111, portanto escrevemos

```
lui   t0, 0xdeadb
addi  t0, t0, -0x111  # tem os mesmos 12 bits que 0xeef
```

Agora o valor em t0 vai ser

```
     1101 1110 1010 1101 1011 0000 0000 0000   (0xdeadb000)
+    1111 1111 1111 1111 1111 1110 1110 1111   (-0x111)
-------------------------------------------------------------
     1101 1110 1010 1101 1010 1110 1110 1111   (0xbeadaeef)
```

Exemplo: 0xdeadbeef

Como `0xeef` tem o bit de sinal a 1, escreve-se como sendo negativo. O complemento para 2 de `0xeef`, é `0x111`, portanto escrevemos

```
lui   t0, 0xdeadb
addi  t0, t0, -0x111  # tem os mesmos 12 bits que 0xeef
```

Agora o valor em `t0` vai ser

```
     1101 1110 1010 1101 1011 0000 0000 0000   (0xdeadb000)
+    1111 1111 1111 1111 1111 1110 1110 1111   (-0x111)
-------------------------------------------------------------
     1101 1110 1010 1101 1010 1110 1110 1111   (0xbeadaeef)
```

Mas ainda está errado! `0xdeadaeef` $\neq$ `0xdeadbeef`

Exemplo: 0xdeadbeef

O problema agora é na extensão de sinal do addi:

- Se o número for positivo, é estendido com zeros e não há problema.
- Se for negativo, é necessário corrigir somando +1 no lui.

```
lui   t0, 0xdeadc      # somamos +1
addi  t0, t0, -0x111   # tem os mesmos 12 bits que 0xeef
```

Exemplo: 0xdeadbeef

O problema agora é na extensão de sinal do `addi`:

- Se o número for positivo, é estendido com zeros e não há problema.
- Se for negativo, é necessário corrigir somando `+1` no `lui`.

```
lui   t0, 0xdeadc      # somamos +1
addi  t0, t0, -0x111   # tem os mesmos 12 bits que 0xeef
```

Agora o valor em `t0` vai ser

```
     1101 1110 1010 1101 1100 0000 0000 0000   (0xdeadc000)
+    1111 1111 1111 1111 1111 1110 1110 1111   (-0x111)
-------------------------------------------------------------
     1101 1110 1010 1101 1011 1110 1110 1111   (0xbeadbeef)
```

Agora sim, a solução está correcta!