“Você poderia me dizer que caminho devo seguir?”
“Isso depende muito de onde você quer chegar” – disse o Gato.
“Eu não me importo para onde” – disse Alice.
“Então qualquer caminho serve” (Alice no país das Maravilhas).

Revisão
PC, Branches e Jumps

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Contador de Programa

Considere o seguinte programa em assembly do RISC-V.

Endereço	Instrução
0x0	addi x5, x0, 11
0x4	addi x6, x0, 19
0x8	add x6, x6, x5
0xC	srli x6, x6, 1
0x10	addi a0 x0 17
0x14	addi a1 x0 0
0x18	ecall

Contador de Programa

O endereço da próxima instrução a ser executada é armazenado no registrador PC (Program Counter).

Não é diretamente visível/acessível ao programador

Obs.: No x86-64 o PC é chamado de IP (Instruction Pointer).

Endereço	Instrução
0x0	addi x5, x0, 11
0x4	addi x6, x0, 19
0x8	add x6, x6, x5
0xC	srli x6, x6, 1
0x10	addi a0 x0 17
0x14	addi a1 x0 0
0x18	ecall

Durante a execução

O processador carrega a instrução no endereço apontado pelo registrador PC.

O processador acrescenta +4 no PC para apontar para próxima instrução.

Por que +4?

O Processador executa a instrução carregada.

O processo se repete.

Durante a execução

O processador carrega a instrução no endereço apontado pelo registrador PC.

O processador acrescenta +4 no PC para apontar para próxima instrução.

Memória endereçada a byte, e cada instrução ocupa uma palavra de 4 bytes.

O Processador executa a instrução carregada.

O processo se repete.

Exemplo

PC | 0x00000000

Endereço	Instrução
0x0	addi x5, x0, 11
0x4	addi x6, x0, 19
0x8	add x6, x6, x5
0xC	srli x6, x6, 1
0x10	addi a0 x0 17
0x14	addi a1 x0 0
0x18	ecall

Exemplo

PC | 0x00000004

Endereço	Instrução
0x0	addi x5, x0, 11
0x4	addi x6, x0, 19
0x8	add x6, x6, x5
0xC	srli x6, x6, 1
0x10	addi a0 x0 17
0x14	addi a1 x0 0
0x18	ecall

Exemplo

PC | 0x00000008

Endereço	Instrução
0x0	addi x5, x0, 11
0x4	addi x6, x0, 19
0x8	add x6, x6, x5
0xC	srli x6, x6, 1
0x10	addi a0 x0 17
0x14	addi a1 x0 0
0x18	ecall

Contador de Programa

Assumindo que o contador de programa possui 32 bits como os demais registradores.

Quantas instruções, no máximo, nossos programas podem ter?

Quantos bytes, no máximo, nossos programas podem ocupar?

Contador de Programa

Assumindo que o contador de programa possui 32 bits como os demais registradores.

Quantas instruções, no máximo, nossos programas podem ter?

2^{30} = 1GB.

Quantos bytes, no máximo, nossos programas podem ocupar?

2^{32} = 4GB.

Contador de Programa

Assumindo que o contador de programa possui 32 bits como os demais registradores.

Quantas instruções, no máximo, nossos programas podem ter?

2^{30} = 1GB.

Quantos bytes, no máximo, nossos programas podem ocupar?

2^{32} = 4GB.

Assumindo que a primeira instrução está no endereço 0x00000000, existe a possibilidade do PC apontar para um endereço como 0x0000005 em algum momento no RISC-V? Por quê?

Restrição de alinhamento

Toda instrução ocupa 32 bits (4 bytes). Como a memória é endereçada a byte, os saltos são de 4 em 4.

As instruções sempre começam em um endereço múltiplo do tamanho da palavra (nesse caso, 4).

Restrição de alinhamento.

Comum em muitas arquiteturas.

Não existe essa restrição em x86-64 (na verdade existe para instruções SIMD. É uma confusão!).

Branches

Instruções de **branch** são utilizadas para criar **desvios condicionais**.

Construir os ifs e loops.

`beq` <- *branch if equal* (desvie se igual).

```
beq x10, x11, ENDEREÇO #salte ao ENDEREÇO se reg1 == reg2
```

Tipo-B

Branches são instruções do Tipo-B.

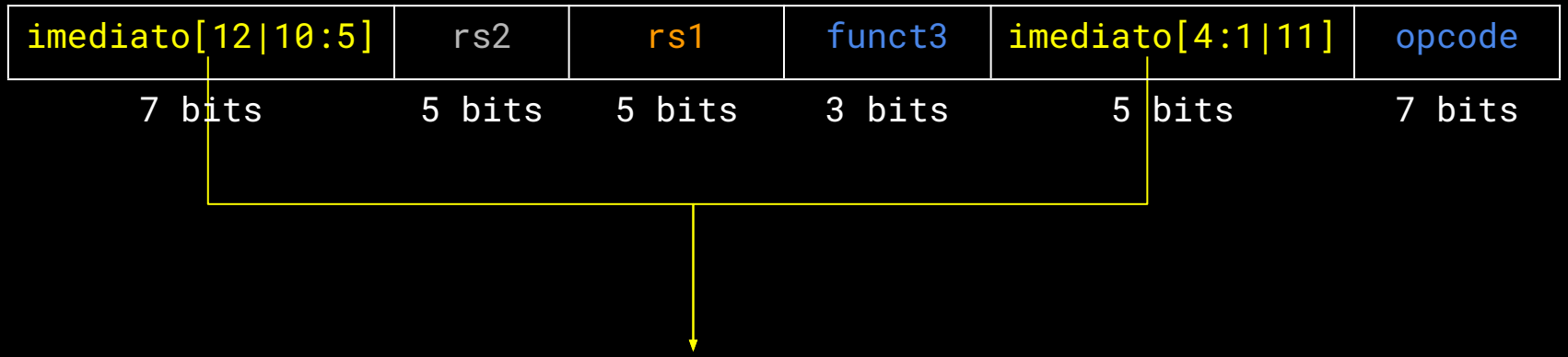
Tipo-B

Branches são instruções do Tipo-B.

Falta o bit de índice 0 no imediato!
Parece estranho (realmente é), mas
veremos adiante que não precisamos dele!

Tipo-B

Branches são instruções do Tipo-B.



`beq x10, x11, ENDEREÇO`

Armazena o ENDEREÇO. Qual o problema?

Endereçamento imediato

Se armazenarmos o endereço final de destino (endereçamento imediato), ficamos limitados a endereços de 12 bits.

Nossos programas não poderiam ter mais de 1024 instruções.

Endereçamento Relativo

O RISC-V utiliza **Endereçamento Relativo ao PC** para seus branches.

Alguns detalhes:

O valor armazenado no campo imediato **possui sinal**.

Podemos saltar para frente ou para trás, relativo ao PC. Complemento de dois.

Os projetistas escolheram possibilitar saltos de **meia palavra**.

O imediato armazena o número de meias palavras (2 bytes) que serão saltadas.

Com isso, podemos saltar 2^{12} = 4096 bytes (2^{11} = 1024 palavras) para frente ou para trás, relativo ao PC.

Endereçamento Relativo

O RISC-V utiliza **Endereçamento Relativo ao PC** para seus branches.

Alguns detalhes:

O valor armazenado no campo imediato **possui sinal**.

Podemos saltar para frente ou para trás, relativo ao PC. Complemento de dois.

Os projetistas escolheram possibilitar saltos de **meia palavra**.

O imediato armazena o número de meias palavras (2 bytes) que serão saltadas.

Com isso, podemos saltar 2^{12} = 4096 bytes (2^{11} = 1024 palavras) para frente ou para trás, relativo ao PC.

Obs.: no MIPS saltos são em palavras inteiras. Aumenta o alcance e simplifica o projeto, mas reduz a flexibilidade.

Branch

```
beq reg1, reg2, ENDEREÇO
```

Caso a condição do branch se satisfaça:

```
pc = pc + ENDERECO*2
```

Caso Contrário:

```
pc = pc + 4
```

Variantes

Demais instruções de comparação, que seguem o mesmo raciocínio do `beq`.

`bne` : branch if not equal (se diferente).

`blt` : branch if less than (se menor).

`bge`: branch if greater or equal (se maior ou igual).

Faça você mesmo

Qual o valor devemos colocar em **ENDEREÇO**, considerando que caso x18 seja igual a x19, devemos saltar para a instrução 0x00400014?

```
Endereço        Instrução
0x00400000 lw x18, 0(x5)
0x00400004 lw x19, 4(x5)
0x00400008 lw x20, 8(x5)
0x0040000C beq x18, x19, ENDEREÇO
0x00400010 addi x20, x20, 5
0x00400014 addi x20, x20, 10
```

Faça você mesmo

Qual o valor devemos colocar em **ENDEREÇO**, considerando que caso x18 seja igual a x19, devemos saltar para a instrução 0x00400014?

```
Endereço        Instrução
0x00400000 lw x18, 0(x5)
0x00400004 lw x19, 4(x5)
0x00400008 lw x20, 8(x5)
0x0040000C beq x18, x19, 4
0x00400010 addi x20, x20, 5
0x00400014 addi x20, x20, 10
```

Assembler ao resgate

Lidar com os endereços dos branches não é tarefa simples.

- Calcular o endereço pode ser confuso.
- Ao inserir uma instrução entre o branch e o seu endereço final, temos que atualizar o imediato do branch.

O montador nos poupa desse problema.

- Podemos utilizar ***labels*** (rótulos) no programa, e pedir por um desvio para o label.
- O montador se encarrega de substituir o rótulo pelo endereço correto quando o programa é montado.
- Rótulos são definidos com um nome único, seguido de dois pontos.

Exemplo

Considerando o programa do exemplo anterior:

o montador vai calcular automaticamente a distância da instrução até o label, e substituir pelo tamanho correto.

Programa Assembly

```
        lw x18, 0(x5)
        lw x19, 4(x5)
        lw x20, 8(x5)
        beq x18, x19, saida
        addi x20, x20, 5
saida:
        addi x20, x20, 10
```

Saltos incondicionais e Loops

Instruções de branch são utilizadas para criar loops.

Em grande parte das arquiteturas, são combinados branchs com instruções de salto (jumps).

No RISC-V, é comum utilizarmos branchs diretamente para realizar o **salto incondicional.**

Jumps (veremos adiante) são utilizados para saltos longos, ou para chamadas de funções.

Exemplo de Loop

```
loop:
        slli x10, x22, 2
        add x10, x10, x25
        lw x9, 0(x10)
        bne x9, x24, saida
        addi x22, x22, 1
        beq x0, x0, loop
saida:
        ...
```

Faça você mesmo

Quais os valores serão utilizados pelo montador quando os rótulos do bne e beq forem substituídos por imediatos?

```
loop:
        slli x10, x22, 2
        add x10, x10, x25
        lw x9, 0(x10)
        bne x9, x24, saida
        addi x22, x22, 1
        beq x0, x0, loop
saida:
        ...
```

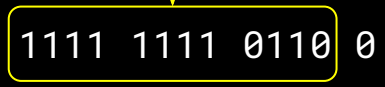
Faça você mesmo

Quais os valores serão utilizados pelo montador quando os rótulos do bne e beq forem substituídos por imediatos?

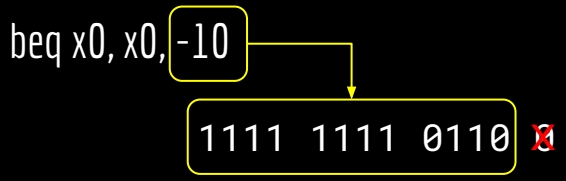
```
loop:
        slli x10, x22, 2
        add x10, x10, x25
        lw x9, 0(x10)
        bne x9, x24, 6
        addi x22, x22, 1
        beq x0, x0, -10
saida:
        ...
```

Sim, é confuso!

beq x0, x0, -10



Sim, é confuso!



O salto, em bytes, na verdade é -20. Mas o bit de índice 0 não é salvo, já que o valor é automaticamente multiplicado por 2 pelo hardware (*salto em halfwords*).

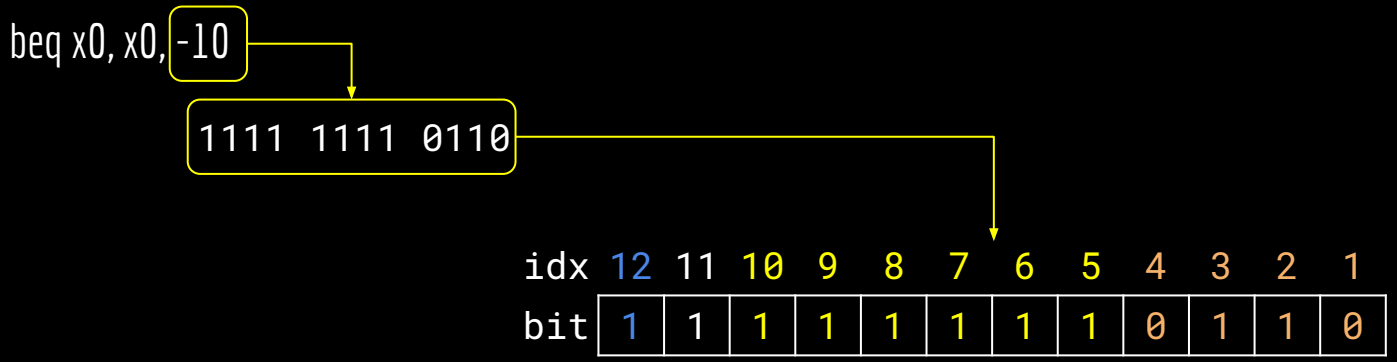
Sim, é confuso!

beq x0, x0, -10

1111 1111 0110

idx	12	11	10	9	8	7	6	5	4	3	2	1
bit	1	1	1	1	1	1	1	1	0	1	1	0

Sim, é confuso!



1111111	00000	00000	000	01101	1100011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Faça você mesmo

Considere o seguinte trecho em C:

```
if(a > b){
    a += 30;
}
b += 10;
```

Assumindo que a variável `a` está no registrador `x18`, e `b` no registrador `x19`, como fica esse trecho em Assembly do RISC-V? Utilize rótulos para resolver saltos.

Dica: a instrução `bge` faz o branch se `reg1≥reg2`.

Faça você mesmo

```
if(a > b){
    a += 30;
}
b += 10;
```

```
    bge x19, x18, saida
    addi x18, x18, 30
saida:
    addi x19, x19, 10
```

Faça você mesmo

Considere o seguinte trecho em C:

```
if(a == b){
    a += 30;
}else{
    b += 10;
}
```

Assumindo que a variável `a` está no registrador `x18`, e `b` no registrador `x19`, como fica esse trecho em Assembly do RISC-V? Utilize rótulos para resolver saltos.

Faça você mesmo

```
if(a == b){
    a += 30;
}else{
    b += 10;
}
```

```
    bne x19, x18 else
    addi x18, x18, 30
    beq x0, x0, saida
else:
    addi x19, x19, 10
saida:
```

Faça você mesmo

Considere o seguinte trecho em C:

```
while(vet[i] == k){
    i += 1;
}
vet[i] = i+10;
```

Assumindo que as variáveis `i` e `k` se encontram nos registradores `x18` e `x19`, e que a base do vetor vet está em `x20`, como fica o trecho em assembly do RISC-V? Considere ainda que o vetor é de inteiros, e que cada inteiro ocupa uma palavra.

Faça você mesmo

i e k se encontram nos registradores x18 e x19, e que a base do vetor vet está em x20, como fica o trecho em assembly do RISC-V?

```
while(vet[i] == k){
    i += 1;
}
vet[i] = i+10;
```

```
loop:
    #multiplicando i por 4 para ajustar as palavras
    slli x21, x18, 2
    add x21, x21, x20 #adicionando o deslocamento à base do vetor
    lw x22, 0(x21) #x21 = vet[i]
    bne x22, x19, saida #saia se vet[i] != k
    addi x18, x18, 1 #adicionando 1 em i
    beq x0, x0, loop #retorna para o início do loop
saida:
    addi x22, x18, 10 #x22 = i+10
    sw x22,0(x21) #vet[i] = i+10
```

Jumps

`jal`: jump and link

```
jal reg, ENDERECO
```

Salte para o endereço, e armazene o endereço da próxima instrução em reg.

Jumps

`jal`: jump and link

```
jal reg, ENDERECO
```

Salte para o endereço, e armazene o endereço da próxima instrução em reg.

PC	0x00000004
x21	?

Endereço	Instrução
0x00400000	lw x18, 0(x5)
0x00400004	lw x19, 4(x5)
0x00400008	lw x20, 8(x5)
0x0040000C	jal x21, meu_label:
0x00400010	addi x20, x20, 5
0x00400014	addi x20, x20, 10
0x00400018	meu_label:

Jumps

`jal`: jump and link

```
jal reg, ENDERECO
```

Salte para o endereço, e armazene o endereço da próxima instrução em reg.

PC	0x00000008
x21	?

Endereço	Instrução
0x00400000	lw x18, 0(x5)
0x00400004	lw x19, 4(x5)
0x00400008	lw x20, 8(x5)
0x0040000C	jal x21, meu_label:
0x00400010	addi x20, x20, 5
0x00400014	addi x20, x20, 10
0x00400018	meu_label:

Jumps

`jal`: jump and link

```
jal reg, ENDERECO
```

Salte para o endereço, e armazene o endereço da próxima instrução em reg.

PC	0x0000000C
x21	?

Endereço	Instrução
0x00400000	lw x18, 0(x5)
0x00400004	lw x19, 4(x5)
0x00400008	lw x20, 8(x5)
0x0040000C	jal x21, meu_label:
0x00400010	addi x20, x20, 5
0x00400014	addi x20, x20, 10
0x00400018	meu_label:

Jumps

`jal`: jump and link

`jal reg, ENDERECO`

Usado para retornar de uma função. Veremos na próxima aula.

Salte para o endereço, e armazene o endereço da próxima instrução em reg.

PC	0x00000018
x21	0x00000010

Endereço	Instrução
0x00400000	lw x18, 0(x5)
0x00400004	lw x19, 4(x5)
0x00400008	lw x20, 8(x5)
0x0040000C	jal x21, meu_label:
0x00400010	addi x20, x20, 5
0x00400014	addi x20, x20, 10
0x00400018	meu_label:

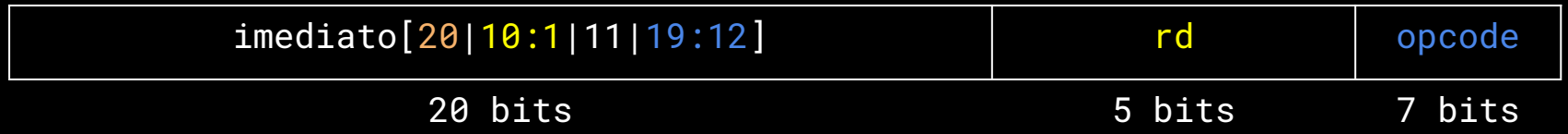
Jumps

`jal reg, ENDERECO`

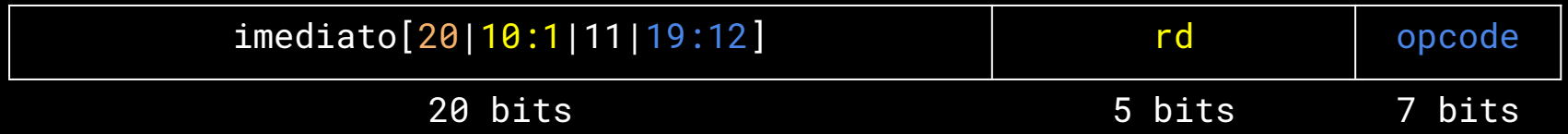
ENDERECO em *halfwords*, e somado ao PC como para os branchs.

Instrução do **Tipo-J**. Imediato de 20 bits.

Imediato em complemento de 2.



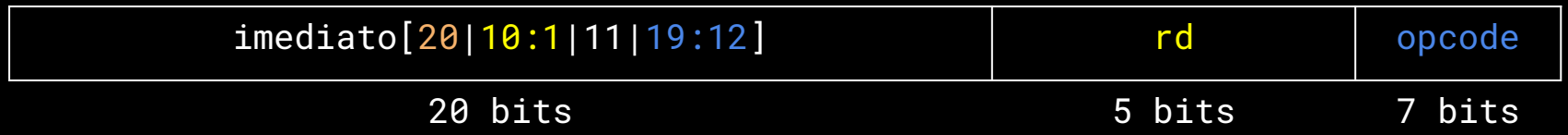
```
Endereço        Instrução
0x00400000      lw x18, 0(x5)
0x00400004      lw x19, 4(x5)
0x00400008      lw x20, 8(x5)
0x0040000C      jal x21, 6
0x00400010      addi x20, x20, 5
0x00400014      addi x20, x20, 10
0x00400018  meu_label:
```



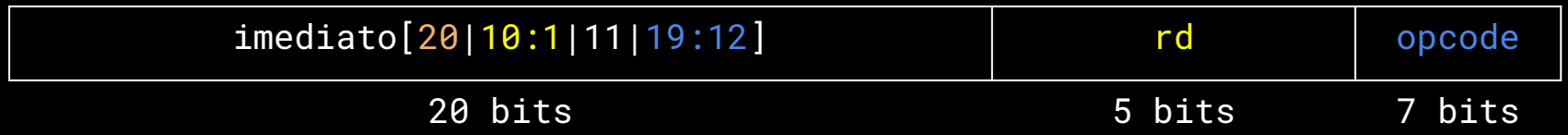
0x0040000C

jal x21, 6

0000 0000 0000 0000 0110 0

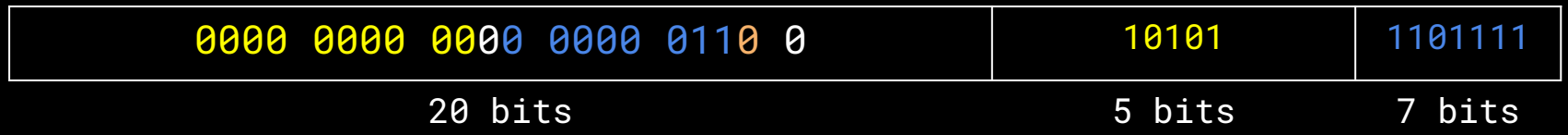


0x0040000C jal x21, 6



0x0040000C jal x21, 6

0000 0000 0000 0000 0110 0



Faça você mesmo

```
    beq x18, x19, L1
    #conjunto de instruções 1
L1:
    #conjunto de instruções 2
```

Considere que `conjunto de instruções 1` é muito grande, ultrapassando o alcance de 1024 instruções campo imediato do beq. Como resolver?

Solução

```
    bne x18, x19, L2
    jal reg_qualquer, L1
L2:
    #conjunto de instruções 1
L1:
    #conjunto de instruções 2
```

Exercícios

1. Quantas palavras podemos saltar, no máximo, utilizando uma instrução `jal`?
2. Pesquise sobre pseudo instruções do RISC-V. O que são? Para que servem? Dica: evite utilizar pseudo instruções por enquanto. Você não poderá utilizar pseudo instruções na prova!
3. Considere o programa em C a seguir:

```
if((a<b && b < 50) || a == -10){
    vet[b] = vet[b] + vet[b-20];
}else{
    a = 50;
}
b++;
```

   Assumindo que as variáveis `a` e `b` estão nos registradores `x18` e `x19`, respectivamente, e que o endereço base de vet está em `x20`. Considerando também que o vetor é de inteiros, e que cada inteiro ocupa uma palavra, escreva o programa equivalente em Assembly do RISC-V.

Exercícios

Para os próximos exercícios, utilize o Simulador Venus.

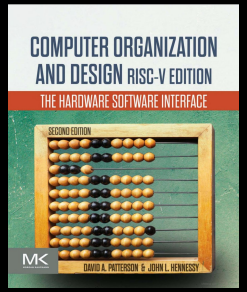
4. Faça um programa que calcula 9! e armazena o resultado em x18. Exiba o resultado na tela.
5. Faça um programa que calcula o enésimo número da sequência de Fibonacci e exibe o resultado na tela. O índice do número de Fibonacci deve estar armazenado em x18.
6. Crie um programa para um caixa eletrônico que calcula o menor número possível de cédulas que deve ser entregue a um usuário quando ele fizer um saque. Considere que a entrada do programa é o valor do saque, e a saída são as notas que o usuário receberá. Exiba as quantidades de notas como inteiros simples na tela, na seguinte ordem: notas de 50, 20, 10 e 5 reais. Se o usuário pedir um valor que não possa ser expresso com essas notas, exiba -1 na tela. Exemplo se o usuário solicitar um saque de 185 reais:

   3 1 1 1

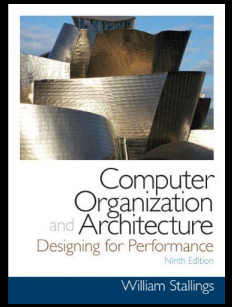
   O valor a ser sacado deve estar no registrador x18.

Referências

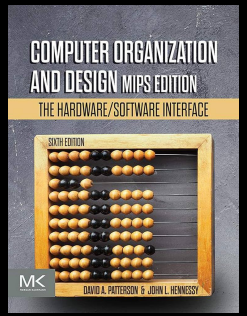
Patterson, Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. 2020.



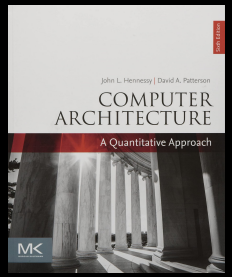
Stallings, W. Organização de Arquitetura de Computadores. 10a Ed. 2016.



Patterson, Hennessy. Computer Organization and Design MIPS Edition: The Hardware/Software Interface. 2020.



Hennessy, Patterson. Arquitetura de Computadores: uma abordagem quantitativa. 2019.



Licença