Chapter 2: Instructions: Language of the Computer

2.6. Logical Operations

Arquitetura e Organização de Computadores

Introdução: 3 classes de instruções

- Aritméticas e lógicas:
  - **`add`**, **`sub`**, **`addi`**
  - **`and`**, **`or`**, **`xor`**, **`not`**, **`shift left`**, **`shift right`**
- Carregar da memória e armazenar na memória:
  - **`lw`**, **`sw`**
  - **`ld`**, **`sd`**
- Transferência de controle (alteram a seqüência de execução das instruções):
  - Ramificação condicional: **`bne`**, **`beq`**
  - "Pulo" não condicional: **`j`**
  - Chamada e retorno de procedimentos: **`jal`**, **`jar`**

Operações Lógicas

- Operam em campos de bits dentro de uma word ou, até mesmo, em bits individuais. São chamadas de **bitwise**.
- São úteis para extrair, inserir ou modificar grupos de bits em uma word.
- As principais são:

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

Operações de deslocamento: shifts

- Movem todos os bits em uma word para a esquerda ou para a direita, e completam os bits vazios com 0 ou 1 (depende da instrução). Para que isso? Descubra com os códigos a seguir!

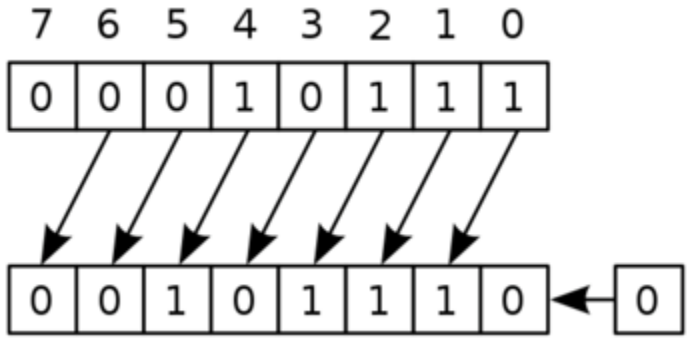
```c
#include <stdio.h>

int main(void) {
    int i = 23;
    int j = i << 1;
    int k = i << 2;
    int m = i << 4;
    printf("i = %d\n", i);
    printf("j = %d\n", j);
    printf("k = %d\n", k);
    printf("m = %d\n", m);
    return 0;
}
```

```c
#include <stdio.h>

int main(void) {
    int i = 368;
    int j = i >> 1;
    int k = i >> 2;
    int m = i >> 4;
    printf("i = %d\n", i);
    printf("j = %d\n", j);
    printf("k = %d\n", k);
    printf("m = %d\n", m);
    return 0;
}
```

Shift Left Logical

```c
#include <stdio.h>

int main(void) {
    int i = 23;
    int j = i << 1;
    int k = i << 2;
    int m = i << 4;
    printf("i = %d\n", i);
    printf("j = %d\n", j);
    printf("k = %d\n", k);
    printf("m = %d\n", m);
    return 0;
}
```

```
> ./main
i = 23
j = 46
k = 92
m = 368
```



- Deslocar à esquerda por **n** bits, é o mesmo que multiplicar o número original por 2^n
- Duas instruções: **shift left logical** e **shift left logical immediate**
  - `sll`
  - `slli`

Shift Left Logical Immediate

- A instrução **`slli`** usa uma variação do Formato-I de instruções:

Formato I

Formato I - variação

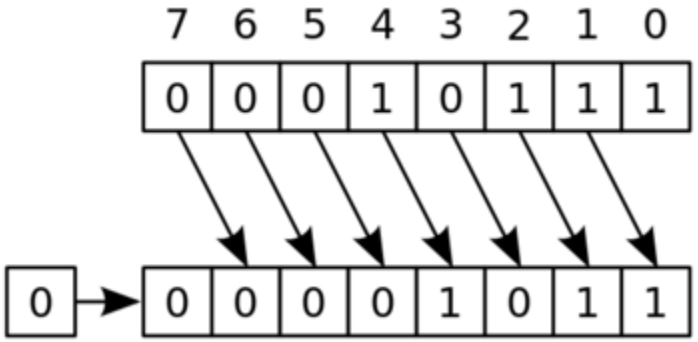
- Exemplo:

```
slli x11, x19, 4
```

Shift Right Logical

```c
#include <stdio.h>

int main(void) {
    int i = 368;
    int j = i >> 1;
    int k = i >> 2;
    int m = i >> 4;
    printf("i = %d\n", i);
    printf("j = %d\n", j);
    printf("k = %d\n", k);
    printf("m = %d\n", m);
    return 0;
}
```

```
./main
i = 368
j = 184
k = 92
m = 23
```



- Deslocar à direita por **n** bits, é o mesmo que dividor o número original por $\mathbf{2^n}$
- Duas instruções : **shift right logical** e **shift right logical immediate**
  - **`srl`**
  - **`srli`**

Shift Right Logical Immediate

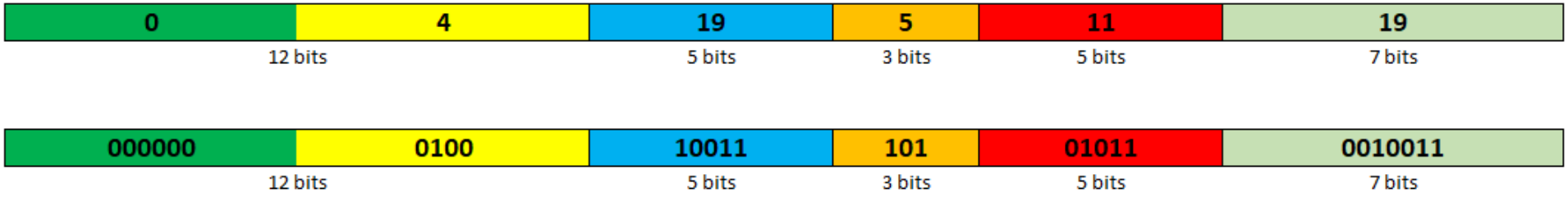
- A instrução **`srli`** usa uma variação do Formato-I de instruções:

Formato I

Formato I - variação

- Exemplo:

```
srli x11, x19, 4
```



Deslocamentos até agora

- Deslocamentos à esquerda:
  - **`sll`**
  - **`slli`**

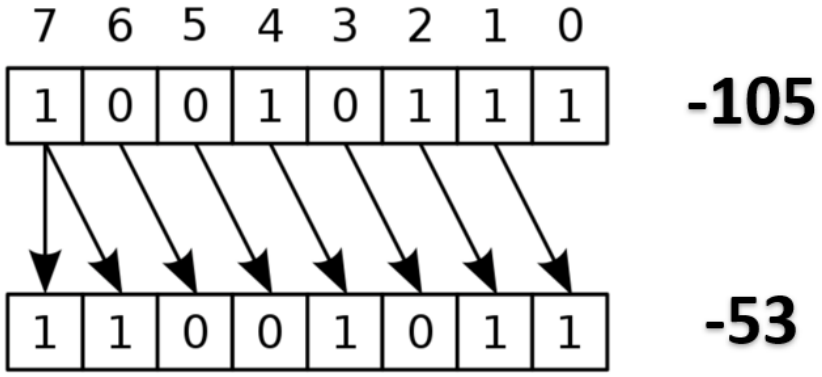
- Descolamentos à direita:
  - **`srl`**
  - **`srli`**

- Qual a diferença?
  - **`slli`** e **`srli`** utilizam uma constante
  - **`sll`** e **`srl`** utilizam outro registrador para fazer o deslocamento (usam o formato R de instruções)

Mais um descolamento: `srai/sra`

- RISC-V ainda tem mais dois deslocamentos à direita, utilizados em situações especiais: **shift right arithmetic** e **shift right arithmetic immediate**
  - sra
  - srai

- A diferença do deslocamento **lógico** à direita (shift right logic e shift right logic immediate) é que no deslocamento aritmético os bits vazios não são preenchidos com 0, são preenchidos com o bit do sinal (que pode ser 0 ou 1)



“Nunca vou usar isso na minha vida”

- Vamos aprender um pouco de Lisp?

```
1  (defparameter *menor* 1)
2
3  (defparameter *maior* 100)
4
5  (defun adivinhe-o-numero ()
6    (ash (+ *menor* *maior*) -1))
7
8  (defun menor ()
9    (setf *maior* (- (adivinhe-o-numero) 1))
10   (adivinhe-o-numero))
11
12 (defun maior ()
13   (setf *menor* (+ (adivinhe-o-numero) 1))
14   (adivinhe-o-numero))
15
16 (defun comecar ()
17   (defparameter *menor* 1)
18   (defparameter *maior* 100)
19   (adivinhe-o-numero))
20
21 (comecar)
```

E as outras operações lógicas?

- Operadores: lógicos e bitwise
  - Lógicos:
    - AND (&&)
    - OR (||)
    - NOT (!) ← NÃO TEM DIRETAMENTE EM RISC-V!
  - Bitwise:
    - AND (&)
    - OR (|)
    - XOR (^)
    - COMPLEMENT (~) ← NÃO TEM DIRETAMENTE EM RISC-V

- Observação:
  - Unários: NOT, COMPLEMENT
  - Binários: AND, OR, XOR

AND lógico

- AND lógico retorna verdadeiro se os operandos são diferentes de zero (se são “true”)

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

```c
#include <stdio.h>
int main()
{
    int a = 5, b = 3, c = 0, d = 0, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = (a && b);
    printf("(a && b) is %d \n", resultado);

    resultado = (a && c);
    printf("(a && c) is %d \n", resultado);

    resultado = (c && d);
    printf("(c && d) is %d \n", resultado);

    return 0;
}
```

```
> ./main
a = 5, b = 3, c = 0, d = 0
(a && b) is 1
(a && c) is 0
(c && d) is 0
```

AND bitwise

- AND bitwise funciona como uma “máscara” bit a bit, sendo útil para selecionar bits específicos de uma word

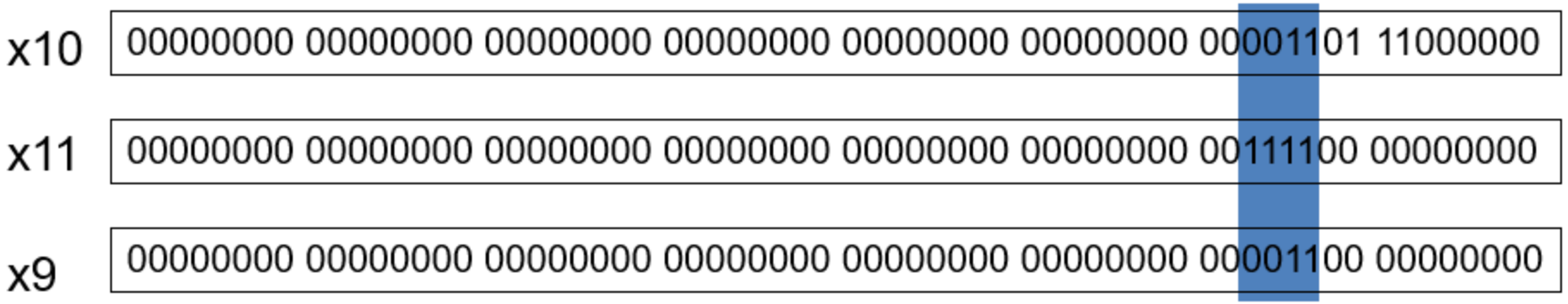
```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25, c = 4, d = 2, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = (a & b);
    printf("(a & b) is %d \n", resultado);

    resultado = (a & c);
    printf("(a & c) is %d \n", resultado);

    resultado = (c & d);
    printf("(c & d) is %d \n", resultado);

    return 0;
}
```

```
./main
a = 12, b = 25, c = 4, d = 2
(a & b) is 8
(a & c) is 4
(c & d) is 0
```

AND bitwise

- AND bitwise funciona como uma “máscara” bit a bit, sendo útil para selecionar bits específicos de uma word

```
and x9,x10,x11
```



OR lógico

- OR lógico retorna verdadeiro se pelo menos um dos operandos for diferente de zero (“true”)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25, c = 0, d = 0, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = (a || b);
    printf("(a || b) is %d \n", resultado);

    resultado = (a || c);
    printf("(a || c) is %d \n", resultado);

    resultado = (c || d);
    printf("(c || d) is %d \n", resultado);

    return 0;
}
```

```
./main
a = 12, b = 25, c = 0, d = 0
(a || b) is 1
(a || c) is 1
(c || d) is 0
```

OR bitwise

- OR bitwise funciona como uma “máscara” bit a bit, sendo útil para inserir bits específicos dentro uma word

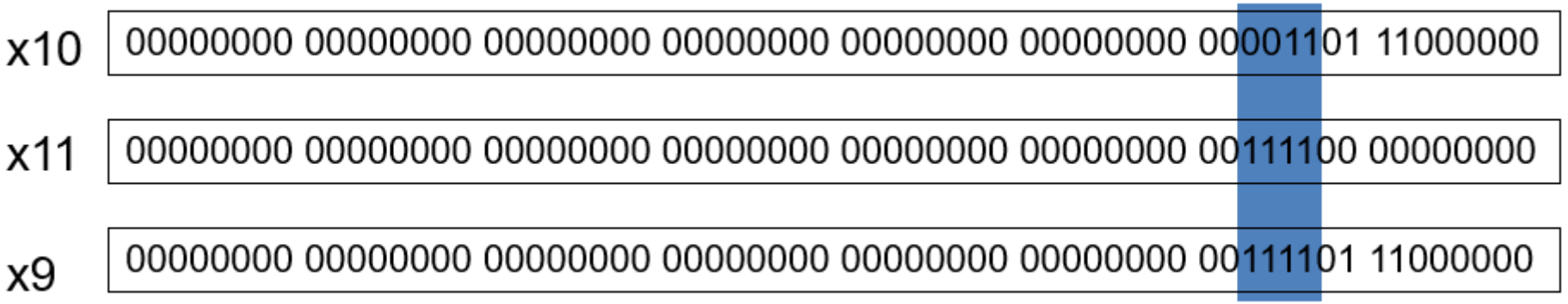
```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25, c = 5, d = 9, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = (a | b);
    printf("(a | b) is %d \n", resultado);

    resultado = (a | c);
    printf("(a | c) is %d \n", resultado);

    resultado = (c | d);
    printf("(c | d) is %d \n", resultado);

    return 0;
}
```

```
> ./main
a = 12, b = 25, c = 5, d = 9
(a | b) is 29
(a | c) is 13
(c | d) is 13
```

OR bitwise

- OR bitwise funciona como uma “máscara” bit a bit, sendo útil para inserir bits específicos dentro uma word

```
or x9,x10,x11
```



NOT lógico

- Inverte o resultado lógico: o que é true vira false, e o que é false vira true

```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25, c = 0, d = 0, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = !a;
    printf("!a is %d \n", resultado);

    resultado = !c;
    printf("!c is %d \n", resultado);

    resultado = !(a && c);
    printf("!(a && c) is %d \n", resultado);

    return 0;
}
```

```
./main
a = 12, b = 25, c = 0, d = 0
!a is 0
!c is 1
!(a && c) is 1
```

NOT bitwise

- Não existe diretamente em RISC-V, para manter o formato da instrução com 3 operandos!

```
or x9,x10,x11

not x9, x10, ?
```

- Pode ser feito INDIRETAMENTE em RISC-V ao fazer um XORI onde o segundo operando é um binário com todos os bits iguais a 1.

XOR bitwise

- É o OU EXCLUSIVO: resulta em “true” se os operandos foram diferentes, e em “false” se forem iguais.
- Inverte todos os bits, bit a bit!

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

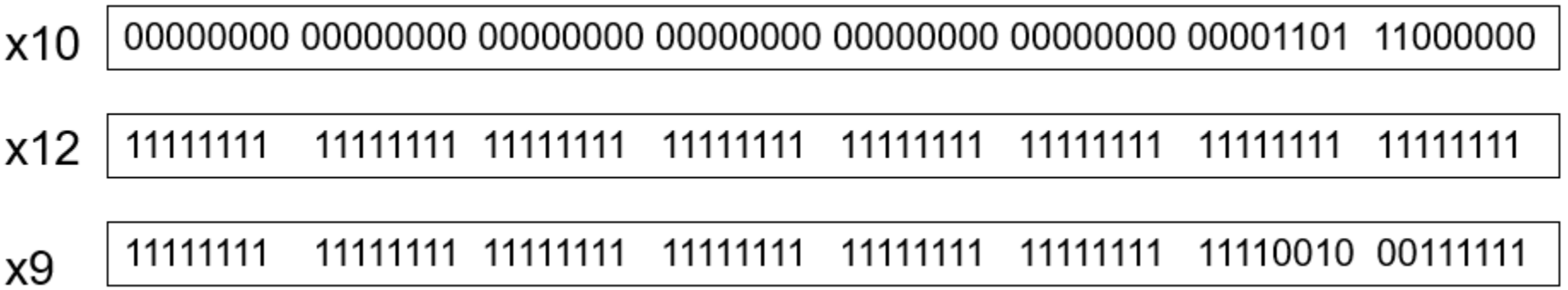
```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25, c = 19, d = 0, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = a ^ b;
    printf("a ^ b é %d \n", resultado);

    resultado = a ^ c;
    printf("a ^ c é %d \n", resultado);

    resultado = b ^ c;
    printf("b ^ c é %d \n", resultado);

    return 0;
}
```

```
./main
a = 12, b = 25, c = 19, d = 0
a ^ b é 21
a ^ c é 31
b ^ c é 10
```

XOR bitwise

- É o OU EXCLUSIVO: resulta em “true” se os operandos foram diferentes, e em “false” se forem iguais.
- Inverte todos os bits, bit a bit!

```
xor x9,x10,x12
```



COMPLEMENT bitwise

- É como se fosse um XOR, mas é unário, ou seja, só atua sobre um único operando, bit a bit.
- Inverte todos os bits, bit a bit!

```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25, c = 19, d = 0, resultado;

    printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    resultado = ~a;
    printf("~a é %d \n", resultado);

    resultado = ~b;
    printf("~b é %d \n", resultado);

    resultado = ~d;
    printf("~d é %d \n", resultado);

    return 0;
}
```

```
> ./main
a = 12, b = 25, c = 19, d = 0
~a é -13
~b é -26
~d é -1
```

COMPLEMENT bitwise

- Por que não tem diretamente em RISC-V?
- Pode ser feita com um xori

Demais instruções lógicas

- As instruções lógicas do tipo bitwise, em RISC-V, são:
  - and
  - andi
  - or
  - ori
  - xor
  - xori

- A diferença entre as instruções e as instruções immediate é o uso ou não se um segundo registrador ou constante

Hora de Esfriar a Cabeça!