

MICROPROCESSADORES E MICROCONTROLADORES



REPRESENTANDO INSTRUÇÕES

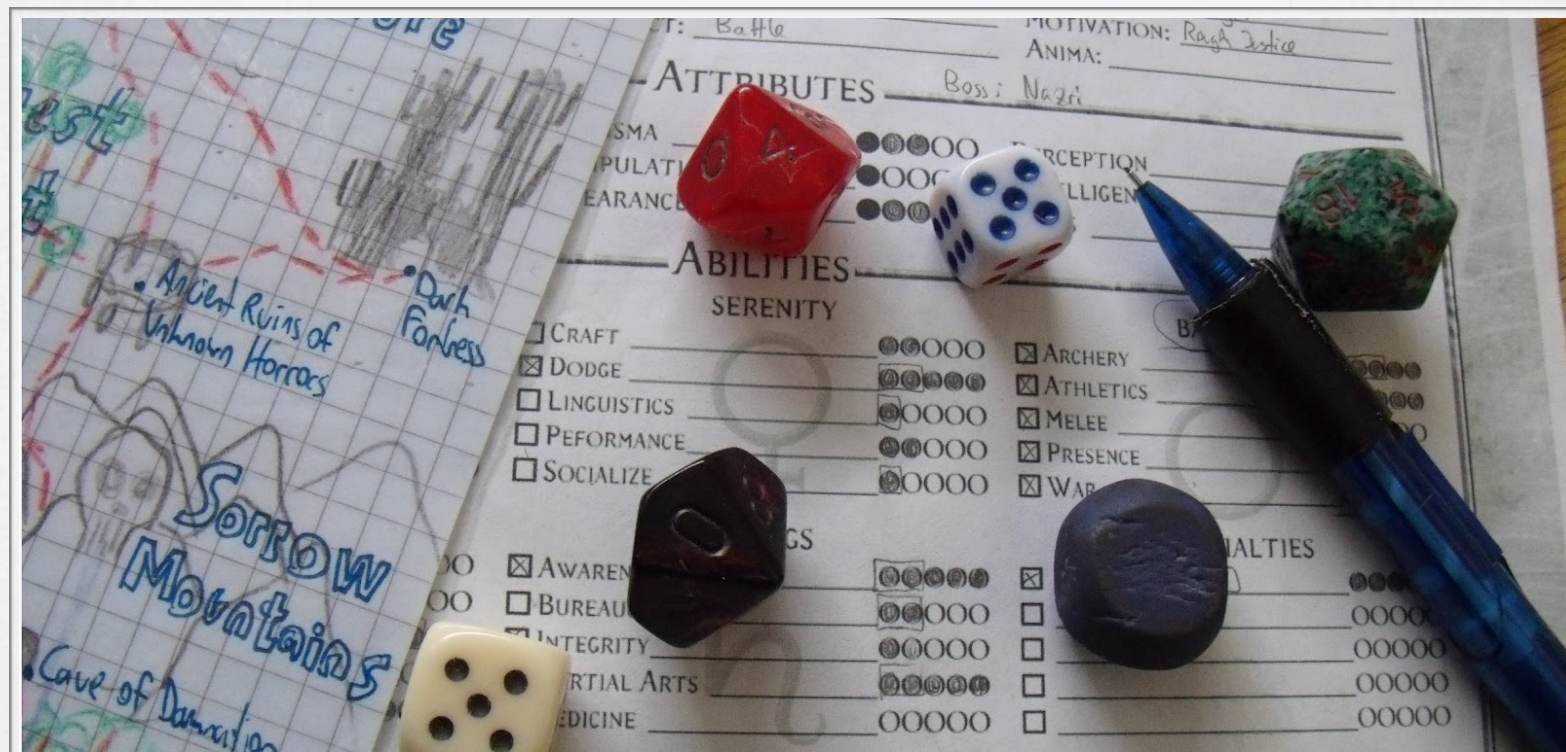
Conceito do programa armazenado

A memória do computador armazena tanto dados de diferentes tipos quanto instruções, o que possibilita o desenvolvimento para computadores.



REPRESENTANDO INSTRUÇÕES

Código de máquina: instruções são representadas por números de 16 bits.



REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits na memória, separados da seguinte maneira:

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits na memória, separados da seguinte maneira:



Símbolo para a instrução

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits na memória, separados da seguinte maneira:



Número do registrador-fonte

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits na memória, separados da seguinte maneira:



Número do registrador-destino

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

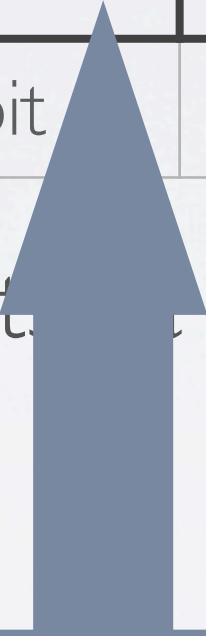
Cada instrução ocupa 16 bits na memória, separados da seguinte maneira:

**Como temos 16 registradores,
precisamos de 4 bits para cada um
desses campos**

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits de memória, separados da seguinte maneira:

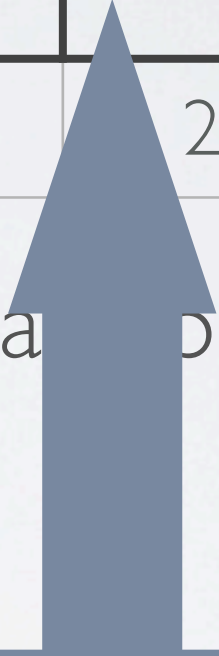


**Indicador do modo de
endereçamento do
registrador-destino
(2 possíveis)**

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits na memória, divididos da seguinte maneira:




**Indicador do modo de
endereçamento do
registrador-fonte
(4 possíveis)**

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Cada instrução ocupa 16 bits na memória, separados da seguinte maneira:



**Indicador do tamanho de operação da instrução:
word (0) ou byte (1)**

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*

Op-code	S-reg	Ad	B/W	As	D-reg

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*



Op-code	Rs	Ad	B/W	As	D-reg
4					

4 é o opcode para "mov"

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*



Op-code	S-reg	D-reg	B/W	As	D-reg
4	5				

**5 é o número do
registrador R5**

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*



Op-code	S-reg	Ad	B/W	As	D-reg
4	5				6

**6 é o número do
registrador R6**

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*




Op-code	S-reg	Ad	B/W	As	D-reg
4	5			0	6

Este bit 0 indica que a CPU deve ler o próprio conteúdo de R5 como fonte

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*


Op-code	S-reg	Ad	B/W	As	D-reg
4	5	0		0	6



Este bit 0 indica que a CPU deve escrever o resultado no próprio registrador R6

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*



Op-code	S-reg	Ad	B/W	As	D-reg
4	5	0	0	0	6

**Este bit 0 indica que a
operação deve ser feita
na palavra completa
(2 bytes)**

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*

Op-code	S-reg	Ad	B/W	As	D-reg
4	5	0	0	0	6
0100	0101	0	0	00	0110

**Conversão dos campos para
valores binários**

REPRESENTANDO INSTRUÇÕES

Exemplo completo: instrução *mov.w R5, R6*

Op-code	S-reg	Ad	B/W	As	D-reg
4	5	0	0	0	6
0100	0101	0	0	00	0110

A instrução *mov.w R5, R6* é armazenada como:

0100010100000110 ou 0x4506

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Algumas instruções ficariam muito limitadas se esse fosse o único formato possível. Por exemplo, a instrução

mov.w #5, R6

não precisa de 2 registradores, e precisa de um campo de 16 bits para representar o valor constante inserido em R6.

Outras instruções seguem lógicas bem diferentes.

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

A instrução *mov.w #5, R6* é representada como:

Op-code = 4

S-reg = 0

Ad = 0

B/W = 0

As = 3

D-reg = 6

2 bytes extras = 5

resultando em 0x4036 0x0005.

REPRESENTANDO INSTRUÇÕES

Op-code	S-reg	Ad	B/W	As	D-reg
4 bits	4 bits	1 bit	1 bit	2 bits	4 bits

Formato I (dois operandos)

Op-code	B/W	Ad	D/S-Reg
9 bits	1 bit	2 bits	4 bits

Formato II (um operando)

Op-code	C	10-bit PC offset
3 bits	3 bits	10 bits

Jumps

REPRESENTANDO INSTRUÇÕES

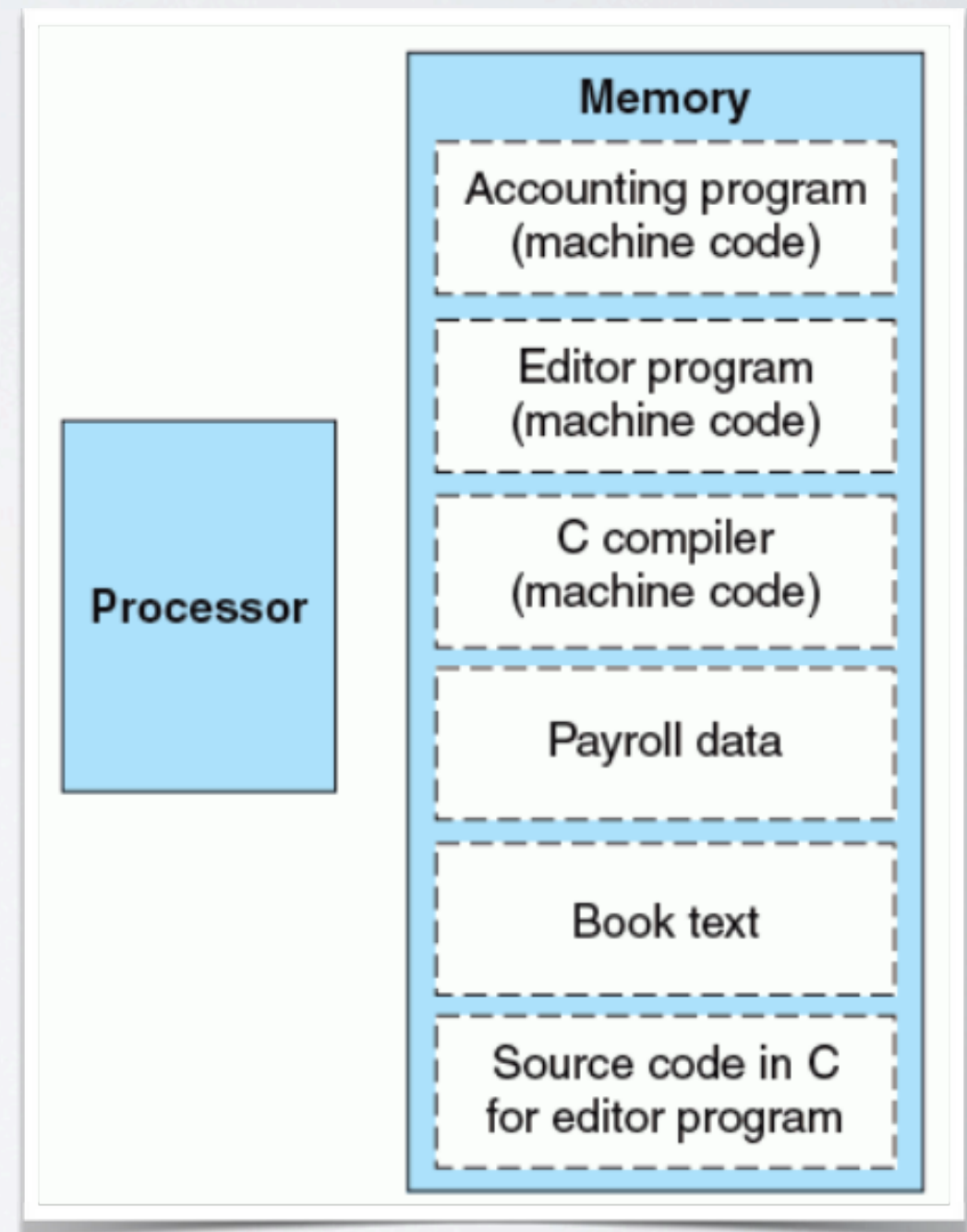
Op-code	S-reg	Ad	B/W	As	D-reg
4 bits					4 bits
Op-					Reg
9 b					bits
Op-					offset

O processador diferencia o tipo de instrução através do opcode.

REPRESENTANDO INSTRUÇÕES

Conceito do programa armazenado

A memória do computador armazena tanto dados de diferentes tipos quanto instruções, o que possibilita o desenvolvimento para computadores.



OPERAÇÕES LÓGICAS

Além da soma, da subtração e da leitura e escrita em memória, é importante executar operações lógicas:

Deslocamento de bits para a esquerda: <<

Deslocamento de bits para a direita: >>

AND bit-a-bit: &

XOR bit-a-bit: ^

Complemento bit-a-bit: ~

OPERAÇÕES LÓGICAS

Considere que R4 contém o valor 9:

0000 0000 0000 1001

A instrução

rla.w R4

pega o valor em R4, desloca-o de 1 bit para a esquerda, e guarda o resultado no próprio R4:

0000 0000 0001 0010
(18 em binário)

OPERAÇÕES LÓGICAS

Considere que R4 contém o valor 9:

0000 0000 0000 1001

Deslocar um valor binário de n bits para a esquerda é o mesmo que multiplicá-lo por 2^n .

Por exemplo, $9 \times (2^4) = 9 \times 16 = 144$

0000 0000 0001 0010
(18 em binário)

OPERAÇÕES LÓGICAS

Como o deslocamento de 1 bit para esquerda equivale a uma multiplicação por 2, a instrução *rla.w R4* é feita com uma soma: *add.w R4, R4*

rla.w é uma instrução **emulada**.

OPERAÇÕES LÓGICAS

Considere que R8 contém o valor 9:

0000 0000 0000 1001

A instrução

rra.w R8

pega o valor em R8, desloca-o de 1 bit para a direita (mantendo o bit mais significativo), e guarda o resultado no próprio R8:

0000 0000 0000 0100

(4 em binário)

OPERAÇÕES LÓGICAS

Considere que R8 contém o valor 9:

0000 0000 0000 1001

Deslocar um valor binário de 1 bit para a direita é o mesmo que dividi-lo por 2.

Tome cuidado com o bit mais significativo, que é mantido na instrução *rra.w*, pois ele representa o sinal da variável.

0000 0000 0000 0100
(4 em binário)

pega o valor
o bit mais

mantendo
o bit R8:

OPERAÇÕES LÓGICAS

Considere que R9 contém

0000 1101 0000 0000

e R8 contém

0011 1100 0000 0000

A instrução *bis R9, R8* realiza a operação OR bit-a-bit de R9 com R8 e guarda o resultado em R8:

0011 1101 0000 0000

OPERAÇÕES LÓGICAS

Considere que R9 contém

0000 1101 0000 0000

e R8 contém

0011 1100 0000 0000

A instrução *bis R9, R8* realiza a operação OR bit-a-bit de R9 com R8 e guarda o resultado em R8:

0011 1101 0000 0000

OPERAÇÕES LÓGICAS

Considere que R9 contém

0000 1101 0000 0000

e R8 contém

0011 1100 0000 0000

A operação OR força bits a serem setados.

Por exemplo, os bits iguais a 0 em R8 foram setados aonde os bits correspondentes em R9 eram iguais a 1.

O nome *bis* vem de "bit set".

OPERAÇÕES LÓGICAS

Considere que R5 contém

0000 1101 0000 0000

e R7 contém

0011 1100 0000 0000

A instrução *and.w R5, R7* realiza a operação AND bit-a-bit de R5 com R7 e guarda o resultado em R7:

0000 1100 0000 0000

OPERAÇÕES LÓGICAS

Considere que R5 contém

0000 1101 0000 0000

e R7 contém

0011 1100 0000 0000

A instrução *and.w R5, R7* realiza a operação AND bit-a-bit de R5 com R7 e guarda o resultado em R7:

0000 1100 0000 0000

OPERAÇÕES LÓGICAS

Considere que R5 contém

0000 1101 0000 0000

e R7 contém

0011 1100 0000 0000

A operação AND força bits a serem zerados.

Por exemplo, os bits iguais a 1 em R7 foram zerados aonde os bits correspondentes em R5 eram iguais a 0.

0000 1100 0000 0000

OPERAÇÕES LÓGICAS

Considere que R5 contém

0000 1101 0000 0000

e R7 contém

0011 1100 0000 0000

A instrução *bic.w R5, R7* realiza a operação AND bit-a-bit de R7 **com o inverso de R5** e guarda o resultado em R7:

0011 0000 0000 0000

OPERAÇÕES LÓGICAS

Considere que R5 contém

0000 1101 0000 0000

e R7 contém

0011 1100 0000 0000

A instrução *bic.w R5, R7* realiza a operação AND bit-a-bit de R7 **com o inverso de R5** e guarda o resultado em R7:

0011 0000 0000 0000

OPERAÇÕES LÓGICAS

Considere que R5 contém

0000 1101 0000 0000

e R7 contém

0011 1100 0000 0000

A vantagem de usar *bic* ao invés de *and* é que *bic* usa 1s para selecionar quais bits zerar no registrador-destino, e *and* usa 0s para o mesmo fim.

0011 0000 0000 0000

OPERAÇÕES LÓGICAS

Considere que R12 contém

0000 1101 0000 0000

e R15 contém

0011 1100 0000 0000

A instrução *xor.w R12, R15* realiza a operação XOR bit-a-bit de R12 com R15 e guarda o resultado em R15:

0011 0001 0000 0000

OPERAÇÕES LÓGICAS

Considere que R12 contém

0000 1101 0000 0000

e R15 contém

0011 1100 0000 0000

A instrução *xor.w R12, R15* realiza a operação XOR bit-a-bit de R12 com R15 e guarda o resultado em R15:

0011 0001 0000 0000

OPERAÇÕES LÓGICAS

Considere que R12 contém

0000 1101 0000 0000

e R15 contém

0011 1100 0000 0000

A operação XOR força bits a serem invertidos.

Por exemplo, os bits em R15 foram invertidos aonde os bits correspondentes em R12 eram iguais a 1.

0011 0001 0000 0000

OPERAÇÕES LÓGICAS

Considere que R7 contém

0000 1101 0000 0000

A instrução *inv.w R7* inverte todos os bits de R7 e guarda o resultado em R7:

1111 0010 1111 1111

OPERAÇÕES LÓGICAS

Considere que R7 contém

0000 1101 0000 0000

A instrução *inv.w R7* inverte todos os bits de R7 e guarda

**Todas estas operações (*bis.w*, *bic.w*, *and.w*, *xor.w* e *inv.w*)
podem ser feitas considerando somente um byte
(*bis.b*, *bic.b*, *and.b*, *xor.b* e *inv.b*).**

DESVIO CONDICIONAL

Computadores não são meras calculadoras.

Eles precisam tomar decisões.

Por exemplo, fechar o programa quando
certo botão for clicado.

DESVIO CONDICIONAL

cmp R6, R8

seta ou reseta os bits N, Z, C e V no registrador SR (*status register*) de acordo com os valores de R6 e R8.

DESVIO CONDICIONAL

cmp R6, R8

seta ou reseta os bits N, Z, C e V no registrador SR (*status register*) de acordo com os valores de R6 e R8.

N = 1 se $R8 < R6$ ($R8 - R6 < 0$),
0 caso contrário ($R8 \geq R6$).

Z = 1 se $R8 = R6$ ($R8 - R6 = 0$),
0 caso contrário (R8 diferente de R6).

C = 1 se houve *carry* no cálculo $R8 - R6$, 0 caso contrário.

V = 1 se houve *overflow* no cálculo $R8 - R6$, 0 caso contrário.

DESVIO CONDICIONAL

cmp R6, R8

seta ou reseta os bits N, Z, C e V no registrador SR (*status register*) de acordo com os valores de R6 e R8.

N = 1 se $R8 < R6$ ($R8 - R6 < 0$),
0 caso contrário ($R8 \geq R6$).

Z = 1 se $R8 = R6$ ($R8 - R6 = 0$),
0 caso contrário (R8 diferente de R6).

C = 1 se houve *carry* no cálculo $R8 - R6$, 0 caso contrário.

V = 1 se houve *overflow* no cálculo $R8 - R6$, 0 caso contrário.

Ao final da instrução, R6 e R8 não se alteram.

DESVIO CONDICIONAL

tst RII

seta ou reseta os bits N, Z, C e V no registrador SR (*status register*) de acordo com os valores de RII.

$N = 1$ se $RII < 0$,
0 caso contrário.

$Z = 1$ se $RII = 0$,
0 caso contrário.

$C = 1$

$V = 0$

Ao final da instrução, RII não se altera.

DESVIO CONDICIONAL

```
cmp R6, R8  
jeq LABEL_1
```

A primeira instrução (*cmp*) compara R6 e R8.

Se R6 for igual a R8, a segunda instrução (*jeq*) faz com que a CPU comece a executar instruções a partir da linha de código marcada com o *label LABEL_1*.

Se R6 for diferente de R8, a próxima instrução é executada normalmente.

DESVIO CONDICIONAL

```
cmp R6, R8  
jne LABEL_2
```

A primeira instrução (*cmp*) compara R6 e R8.

Se R6 for diferente a R8, a segunda instrução (*jne*) faz com que a CPU comece a executar instruções a partir da linha de código marcada com o *label LABEL_2*.

Se R6 for igual a R8, a próxima instrução é executada normalmente.

DESVIO CONDICIONAL

jmp LABEL_3

Esta instrução faz com que a CPU comece a executar instruções a partir da linha de código marcada com o *label LABEL_3*, **independente de qualquer resultado anterior.**

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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


OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

```
cmp R7, R8          ; R7 = i, R8 = j
jne ELSE
mov.w R5, R4         ; R4 = f, R5 = g
sub.w R6, R4         ; f = g + h (R4 = R5 + R6)
jmp EXIT
ELSE:
    mov.w R5, R4     ; R4 = f, R5 = g
    sub.w R6, R4     ; f = g - h (R4 = R5 - R6)
EXIT:
...                  # próximas instruções do código
```

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

```
cmp R7, R8  
jne ELSE  
mov.w R5, R4  
sub.w R6, R4  
jmp EXIT  
ELSE:  
    mov.w R5, R4  
    sub.w R6, R4  
EXIT:  
...
```

; R7 = i, R8 = j

jne funciona como um goto condicional. Ele executa o oposto de $if(i==j)$

; R4 = f, R5 = g

; f = g - h (R4 = R5 - R6)

próximas instruções do código

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

```
cmp R7, R8           ; R7 = i, R8 = j
jne ELSE
mov.w R5, R4
sub.w R6, R4
jmp EXIT
ELSE:
  mov.w R5, R4
  sub.w R6, R4        ; f = g - h (R4 = R5 - R6)
EXIT:
...
```

Esse salto incondicional é necessário para evitar que a instrução $f = g - h$; seja executada quando $i == j$.

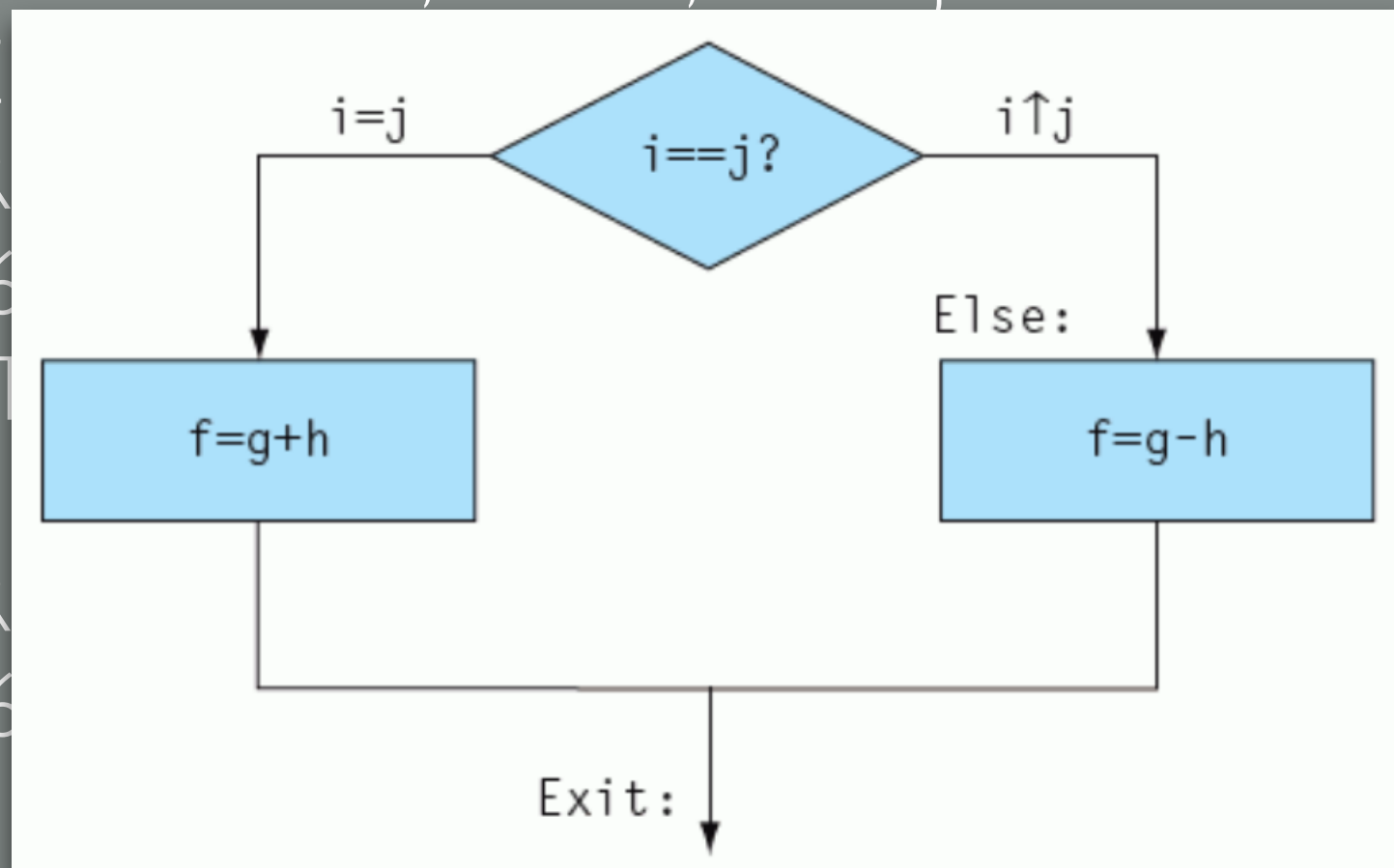
próximas instruções do código

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

```
cmp R7, R8      ; R7 = i, R8 = j
jne ELSE
mov.w R6, R5     f = g + h
sub.w R6, R5
jmp EXIT
ELSE:
mov.w R6, R5     f = g - h
sub.w R6, R5
EXIT:
...
```



próximas instruções do código

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

```
LOOP: mov.w R7, R12 ; R7 = i, R12 = temporário
      rla R12       ; R12 = 2*i
      add.w R10, R12 ; R10 = save, R12 = save + 2*i = &save[i]
      cmp 0(R12), R9 ; Compara save[i] com k (R9)
      jne EXIT      ; save[i] == k ?
      inc.w R7       ; i += 1;
      jmp LOOP
```

EXIT:

...

próximas instruções do código

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

LOOP: mov.w R7, R12 · R7 = i R12 = temporário

rla R12

add.w R10, R12

cmp 0(R12), R9

jne EXIT

inc.w R7

jmp LOOP

EXIT:

...

O vetor save[] é do tipo inteiro, portanto é necessário multiplicar i por 2 para andar pelas posições do vetor.

= &save[i]

próximas instruções do código

OPERAÇÕES ARITMÉTICAS

Como a seguinte linha em C é compilada para o MSP430?

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

```
LOOP: mov.w R7, R12 ; R7
      rla R12      ; R12
      add.w R10, R12 ; R12
      cmp 0(R12), R9 ; Co
      jne EXIT     ; sav
      inc.w R7      ; i += 1;
      jmp LOOP
```

EXIT:

...

próximas instruções do código

O vetor save[] está na memória, exigindo a definição do endereço de save[i] antes de sua leitura.

DESVIO CONDICIONAL

jne e *jeq* lidam somente com igualdades e desigualdades, mas também é necessário identificar valores maiores ou menores.

DESVIO CONDICIONAL

jne e *jeq* lidam somente com igualdades e desigualdades, mas também é necessário identificar valores maiores ou menores.

```
    cmp R5, R6  
    jl  OUTRO_LABEL_1
```

Se $R6 < R5$, a CPU começa a executar instruções a partir da linha de código marcada com o *label OUTRO_LABEL_1*.

DESVIO CONDICIONAL

jne e *jbe* são instruções de desvio condicional, mas *cmp* *B*, *A* calcula *A-B*, setando os bits *N* (*negative*) e *V* (*overflow*) de acordo com o resultado.

jl causa o jump se *N XOR V* for igual a 1, o que garante que o o jump será feito independentemente do tipo das variáveis *A* e *B* (*signed* ou *unsigned*).

DESVIO CONDICIONAL

```
cmp R7, R4  
jge OUTRO_LABEL_2
```

Se $R4 \geq R7$, a CPU começa a executar instruções a partir da linha de código marcada com o *label OUTRO_LABEL_2*.

DESVIO CONDICIONAL

jge faz o inverso de *jl*.

Ou seja, o jump é feito se $N \text{ XOR } V$ for igual a 0, independentemente do tipo das variáveis A e B (*signed* ou *unsigned*) usadas na instrução anterior *cmp B, A*.

Se
da

rtir
2.