Instruções de escolha – operações condicionais

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of a computation…
Burks, Goldstine and von Neumann, 1947

Operações condicionais (***branch***)

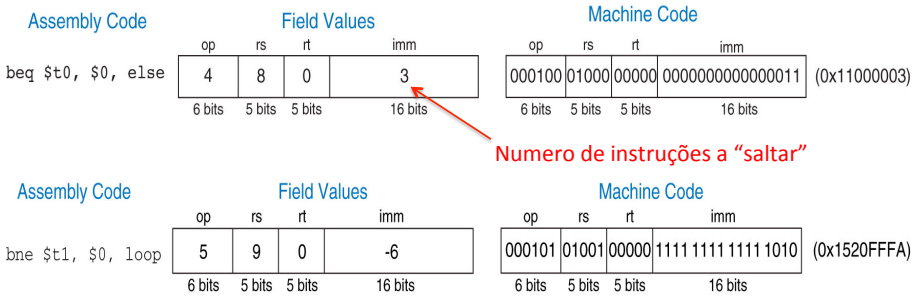
Salta para a instrução indicada se a condição é verdadeira
Senão, continua sequencialmente

`beq rs, rt, L1`
if (rs == rt) branch to instruction labeled L1;

```
…
beq $s0, $0, L1 # se ($s0) = 0 "saltar" para L1
…
…   } Instruções não executadas se ($s0) = 0
…
L1: add $s0, $t0, $s0
```

`bne rs, rt, L1`
if (rs != rt) branch to instruction labeled L1;

Formato das instruções de *branch* (Tipo-I)

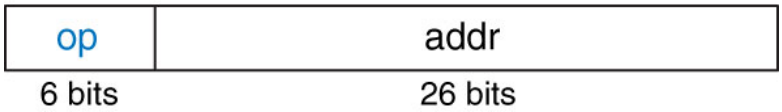


Salto incondicional (*GoTo*)

(*Go To Statement Considered Harmful, E.W.Dijkstra, 1968*)

`j L1`
unconditional jump to instruction labeled L1

J-type



Compilação de *if .. then*

Código C: **f** em $s0, **i** em $s3, **j** em $s4

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

Código Assembly MIPS:

Testa-se a condição negada

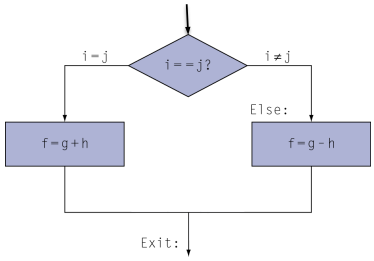
```
     bne $s3, $s4, L1
     add $s0, $s1, $s2
L1:  sub $s0, $s0, $s3
```


Compilação de *if .. then .. else*

- Código C:

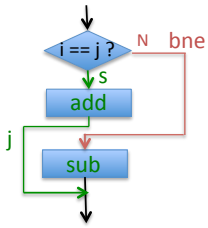
```
if (i==j) f = g+h;
else f = g-h;
```

f, g, …, j em $s0, $s1, …, $s4



- Código Compilado MIPS:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```



Assembler calcula endereços correspondentes aos labels

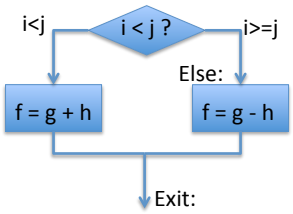
Outras operações condicionais

Como codificar outras condições para alem de = ou ≠ ?

- No MIPS isso é feito indiretamente, usando a instrução ***set on less than*** que compara os valores de 2 registos (instrução tipo R)

```
slt rd, rs, rt
```

  - if (rs < rt) rd = 1; else rd = 0;
- Usado em combinação com ***beq*** e ***bne***. Exemplo:



if (i < j) f = g+h;
else f = g-h; /* f, g, ..., j em $s0, $s1, ..., $s4

```
    slt $t0, $s3, $s4  # ($t0) = 1 if ($s3 < $s4)
    beq $t0, $zero, Else
    add $s0, $s1, $s2
    j   Exit
Else: sub $s0, $s1, $s2
Exit: ...
```


Comparações com constantes

slti - Set on less than immediate

compara o conteúdo de um registo com um imediato (constante) que figura no código de instrução

```
slti rt, rs, constant
```

➢ if (rs < constant) rt = 1; else rt = 0;

Comparações ***signed*** e ***unsigned***

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Exemplo:
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Outras Instruções de salto condicional

- Todas as outras condições para além de **=** e **≠** podem ser expressas por uma combinação de **slt** e **beq** ou **bne**.
- (Pseudo)Instruções que permitem exprimir essas outras condições diretamente:

  beqz **-** Branch on equal to zero
  bnez **-** Branch on not equal to zero
  bge **-** Branch on greater or equal
  bgeu **-** Branch on greater or equal unsigned
  bgt **-** Branch on greater than
  bgtu **-** Branch on greater than unsigned
  ble **-** Branch on less or equal
  bleu **-** Branch on less or equal unsigned
  blt **-** Branch on less than
  bltu **-** Branch on less than unsigned

Case / Switch

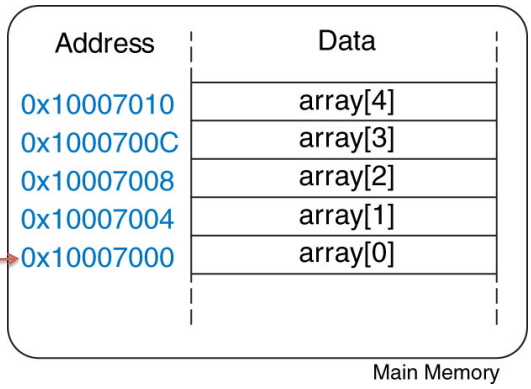
- Escolha de uma entre várias alternativas. Como codificar em *assembly* esta construção?
- Duas possibilidades:
  1. Codificar *switch como uma sequência* de *if-then-else*
  2. Codificar as alternativas como uma tabela de endereços das sequências alternativas de instruções – *jump address table* ou simplesmente *jump table.*

jump table – array de *words* contendo os endereços que correspondem aos *labels* do programa

Arrays em memória

Array de inteiros com 5 elementos

Endereço do array



- Elementos do array são armazenados em localizações contíguas na memória
- Cada elemento do array é identificado pelo respetivo índice
- C: o primeiro elemento do array tem o índice 0
- O "endereço do array" é o endereço do seu primeiro elemento

Compilação de ciclos: ***while***

Código C:

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

i em $s3, k em $s5, endereço de save em $s6

Tradução para *assembly*:

1. Colocar o elemento índice i de save num registo temporário
   a. Obter o endereço de save [i]: &save[i] = &save[0] + 4*i

```
Loop: sll  $t1, $s3, 2      #  4*i em $t1
      add  $t1, $t1, $s6    # endereço de save[i] em $t1
```

Necessário o label para iniciar aqui cada nova iteração do ciclo

   b. Transferir save[i] para $t0

```
lw   $t0, 0($t1)
```


Compilação de ciclos: ***while*** (2)

2. Testar a condição (save[i] == k)

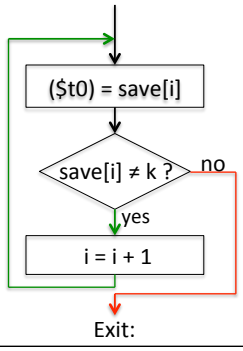
```
bne  $t0, $s5, Exit      # se save[i] ≠ k terminar ciclo
```

3. Incrementar **i** e iniciar nova iteração

```
addi $s3, $s3, 1         # (i+1) em $s3
j    Loop
```

- Código Compilado MIPS:

```
Loop:   sll  $t1, $s3, 2      # ($t1) = 4*($s3)
        add  $t1, $t1, $s6
        lw   $t0, 0($t1)
        bne  $t0, $s5, Exit
        addi $s3, $s3, 1
        j    Loop
Exit: …
```



Compilação de ciclos: *for*

- Exemplo:

```
for (k = 0; k < 50; k++)
{
    sum = sum + y[k];
}
...
```

- Usando *while*:

```
k = 0;
while (k < 50)
{
    sum = sum + y[k];
    k++;
}
...
```

Inicialização
(antes de iniciar o ciclo)

Condição de continuação de execução do ciclo

Última instrução do ciclo:
atualização da variável de controle

for (2)

C:

```
for (k = 0; k < 50; k++)
    {
        sum = sum + y[k];
    }
    ...
```

- Traduzido para *while*:

```
k = 0;
while (k < 50)
{
    sum = sum + y[k];
    k++;
}
...
```

***Assembly*:**

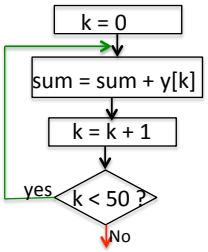
- **k** em $s1, Endereço do array **y** em $s2, **sum** em $s3

```
        add $s1, $0, $0 # k= 0
        addi $t1, $0, 50
while:  bge $s1, $t1, endciclo  # (se k ≥ 50 fim ciclo)
        sll $t0, $s1, 2     # ($t0) = 4 * k
        add $t0, $s2, $t0  # ($t0) = 4*k + &y[0]
        lw $t0, 0($t0)    # ($t0) = y[k]
        add $s3, $s3, $t0  # ($s3) = ($s3) + y[k]
        addi $s1, $s1, 1    # k = k+1
        j while
Endciclo: …
```

do … while

C:

```
k = 0;
do
{
  sum = sum + y[k];
     k++;
} while (k < 50)
…
```



***Assembly*:**

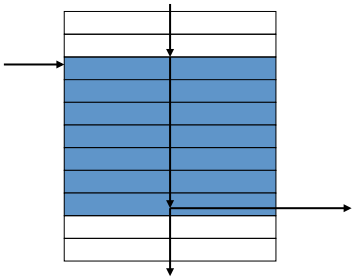
- **k** em $s1, endereço do array **y** em $s2, **x** em $s3

```
      addi    $s1, $s0, $s0     # k= 0
do:  sll $t0, $s1, 2    # ($t0) = 4 * k
     add $t0, $s2, $t0  # ($t0) = 4*k + Ender. y[0]
     lw $t0, 0($t0)     # ($t0) = y[k]
     add $s3, $s3, $t0  # ($s3) = ($s3) + y[k]
     addi $s1, $s1, 1     # k = k+1
     blt $s1, 50, do      # while (k < 50)
     …
```

teste no fim do ciclo

Basic Blocks

- basic block - sequência de instruções sem
  - saltos (exceto no fim)
  - alvos de instruções de salto (exceto no início)

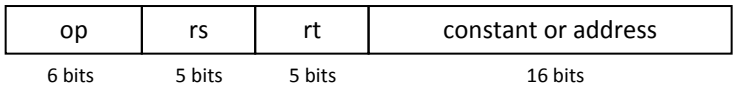


- O compilador identifica *basic blocks* para optimização
- O processador pode acelerar a execução dos *basic blocks*

12. Modos de endereçamento no MIPS

Endereçagem de ***branch***

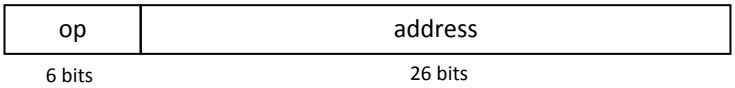
- As instruções de *branch* especificam
  - Opcode, dois registos, target address
- A maioria dos branch saltam para instruções próximas
  - Forward or backward



- ***PC-relative addressing***
  - Target address = PC + offset × 4
  - PC já a apontar para a instrução seguinte (previamente incrementado de 4)

Endereçagem de *jump*

- Destinos de ***Jump*** (`j` and `jal`) podem estar em qualquer posição no segmento ***text***
  - Endereço completo codificado na instrução (instrução Tipo J)



- (Pseudo)**Direct addressing**
  - Target address = $PC_{31...28}$: (address × 4)

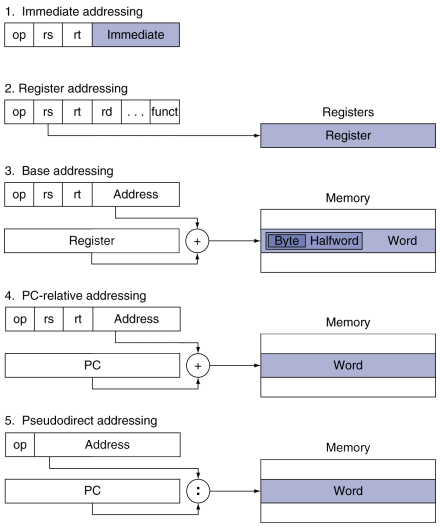
Target Addressing: exemplo

- Assume-se que Loop em 80000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	4	0	
0	9	22	9	0	32	
35	9	8	0			
5	8	21	2			
8	19	19	1			
2	20000					

Modos de Endereçamento: sumário



addi
andi, ori

add, sub
and, or, xor

lw, sw
lb, sb
lh, sh

beq, bne

j

Instruções MIPS já vistas

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

Constantes de 32-bits

- Carregar constante de 16 bits (0-extended) num registo: ***ori* $s0, $0, immediate**
- Carregar constante de 32 bits num registo:

  ***lui* $s0, immediate_16MSB**

  ***ori* $s0, $s0, immediate_16LSB**