PROJETOS DIGITAIS E MICROPROCESSADORES MIPS: CONJUNTO DE INSTRUÇÕES CONTROLE DE FLUXO DE EXECUÇÃO

Marco A. Zanata Alves

CONTROLE DE FLUXO DE EXECUÇÃO (I)

Fluxo sequencial de execução:

- Próxima instrução é aquela em PC+4

Instruções para efetuar **desvios** `if(){ }` ou `while(){ }`

- `beq r1, r2, endereço` `# branchEqual desvia se r1 == r2`
- `bne r1, r2, endereço` `# branchNotEq desvia se r1 != r2`
- Se desvia PC ← endereço ; Senão PC ←PC+4

Instruções para efetuar **saltos** goto

- `j endereço` `# jump (salto incondicional)`
- `jr rt` `# jump register ; rt = endereço destino`
- PC ← endereço

Desvios são condicionais ; Saltos são incondicionais

CONTROLE DE FLUXO DE EXECUÇÃO (II)

Instruções para efetuar **desvios** if( ){ } while( ){ }

- `beq r1, r2, endereço` `# Branch Equal desvia se r1 == r2`
- `bne r1, r2, endereço` `# Branch Not Eq desvia se r1 != r2`

Instruções para comparação de magnitude

- `slt rd, r1, r2` `# Set on Less Than`
- rd ← 1 se r1 < r2 Em C: rd = ((r1 < r2) ? 1 : 0);

- `slt rd, r1, r2` `# rd ← 1 se ( r1 < r2 )`
- `slti rd, r1, const` `# rd ← 1 se ( r2 < ext(const) )`
- `sltu rd, r1, r2` `# subtração não gera exceção`
- `sltiu rd, r1, const` `# subtração não gera exceção`

CONTROLE DE FLUXO DE EXECUÇÃO (III)

```
beq r1, r2, endereço          # branchEqual desvia se r1 == r2
bne r1, r2, endereço          # branchNotEq desvia se r1 != r2
slt rd, r1, r2                # rd ← 1 se r1 < r2, senão 0
                              # em C, 1=TRUE e 0=FALSE
```

Sequência equivalente a blt (branch on less than)

```
slt r1, r2, r3                # r1 ← (r2 < r3) T ou F
bne r1, $zero, endereço       # salta se TRUE 6= FALSE
```

Sequência equivalente a bge (branch if greater or equal)

```
slt r1, r2, r3                # r1 ← (r2 < r3) T ou F
beq r1, $zero, endereço       # salta se FALSE = FALSE
```

DESVIOS: ENDEREÇO DE DESTINO

```
beq r1, r2, deslocamento
```

O endereço de destino quando o desvio for tomado é:

- PC ← (PC + 4) + extSinal(deslocamento) * 4
- Deslocamento é o número de instruções por saltar (relativo a PC+4)

```
    beq r1, r2, desloc # salta para L1
    add ...
    sub ...
    xor ...
L1: sw ...             # qual o valor em desloc?
```

DESVIOS: ENDEREÇO DE DESTINO (CONT)

Qual o efeito das sequências abaixo? nop = no-operation

```
L1: nop
L2: beq r1, r1, -1
L3: nop

L4: nop
L5: beq r1, r1, 0
L6: nop

L7: nop
L8: beq r1, r1, +1
L9: nop
```

DESVIOS E SALTOS (I)

```
if (i == j) goto L1;          beq $i, $j, L1
  f = g + h;                  add $f, $g, $h
L1:                     L1:   sub $f, $f, $i
f = f - i;
if (i == j)                   bne $i, $j, Else
  f = g + h;                  add $f, $g, $h
else                          j Exit      # salta else
  f = g - h;            Else: sub $f, $g, $h
                        Exit:
```

DESVIOS E SALTOS (II)

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

```
# i,j,k <-> $19,$20,$21
      la $7, save        # $7 ← &(save[0])
Loop: sll $9, $19, 2     # $9 ← i*4 (i<<2)
      add $9, $7, $9     # $9 ← (i*4 + save)
      lw $8, 0($9)       # $8 ← M[ i*4 + save ]
      bne $8, $21, Fim   # termina se save[i] ≠ k
      add $19, $19, $20  # i ← i + j
      j Loop             # repete
Fim:
```

MANIPULAÇÃO DE STRINGS

ESTRUTURAS DE DADOS EM C – STRINGS

Strings são vetores de caracteres terminados por '\0' = 0x00

Uma string é um vetor do tipo char, de tamanho "indefinido":

- O fim do vetor é a posição do '\0'
- No código fonte, strings são representadas entre aspas duplas,
- E caracteres representados entre aspas simples

No código fonte: "palavra" o '\0' não é mostrado

Quando se computa o tamanho da string, o '\0' é contado

Em memória, com a string alocada no endereço 30:

End.	30	31	32	33	34	35	36	37
	'p'	'a'	'l'	'a'	'v'	'r'	'a'	'\0'

CÓDIGO ASCII

Caracteres tipicamente são codificados em ASCII:

- Cada caractere é representado em 7 bits e armazenado num byte
- O dígito 0 é representado pelo caractere '0' que é 0x30, 1 é 0x31, 2 é 0x32, ... 9 é 0x39
- As maiúsculas vão de 0x41 = A até 0x5A = Z;
- As minúsculas vão de 0x61 = a até 0x7A = z;
- Espaço é 0x20, exclamação é 0x21, aspas duplas é 0x22, ...
- veja **`man ascii`** para a tabela completa

O código ASCII foi projetado para representar o idioma Inglês e portanto não provê acentuação;

- **`man iso 8859-1`** para codificação com acentos (Latin-1) em 8 bits

MANIPULAÇÃO DE STRINGS EM ASSEMBLY (I)

Trecho de código que copia strings:

```
char fte[16]="abcd-efgh-ijkl-";   // 16 contando '\0'
char dst[32];
int i;
i = 0;
while ( fte[i] != '\0' ) {        // terminou?
    dst[i] = fte[i];
    i = i + 1;
}
dst[i] = '\0';          // laço não copia o '\0'
```

MANIPULAÇÃO DE STRINGS EM ASSEMBLY (II)

```
      la r8, fte                    # r8 ← fte
      la r9, dst                    # r9 ← dst
      add r4, $zero, $zero          # i = 0;
# while (fte[i] != '\0') {
laco: add r18, r8, r4               # r18 ← fte+i
      lbu r5, 0(r18)                # r5 ← fte[i]
      beq r5, $zero, fim            # (r5 = '\0') ? fim
      add r19, r9, r4               # r19 ← dst+i
      sb r5, 0(r19);                # dst[i] ← fte[i]
      addi r4, r4, 1                # i = i + 1;
      j laco
# }
      add r19, r9, r4               # r19 ← dst+i
fim:  sb $zero, 0(r9)               # dst[i] = '\0';
```

EXERCÍCIOS

Traduza para assembly do MIPS o trecho de programa em C:

```
char fte[NN]; char dst[NN];
int i, num;
i = 0;
while (fte[i] != '\0') {
    i = i + 1;
}
num = i + 1;            // por que num←i+1 ??
for (i=0; num > 0; num--, i++) {
    dst[i] = fte[num - 1];
}
dst[i] = '\0';          // laço não copia o '\0'
```