Arquitetura de Computadores

Funções: função folha, função não folha, chamada, retorno, passagem de parâmetros, retorno de valores, convenções, estrutura *stack*
Bib: A – secção 6.3.7

Slides inspirados nos slides do prof. Tiago Dias

João Pedro Patriarca (jpatri@cc.isel.ipl.pt), Gabinete F.0.23 do edifício F

ISEL, ADEETC, LEIC

O que é uma função?

- Sequência de instruções que podem ser executadas mais do que uma vez ao longo do programa
- Uma função tem um ponto de entrada, corpo e um ou mais pontos de saída
- Uma função pode ter zero, um ou mais parâmetros
- Uma função pode retornar zero ou apenas um valor
- Diz-se que uma função é uma função folha se no seu corpo não for invocada outra função, caso contrário, diz-se que é uma função não folha

Enunciado

- Implementar a função `indexof_max` que retorna o índice do maior valor inteiro, em absoluto, de uma sequência de valores inteiros em memória.

```
uint8_t indexof_max(int16_t a[], uint8_t n);
```

- O parâmetro `a` representa o endereço base da sequência, ou seja, o endereço de memória do primeiro valor da sequência
- O parâmetro `n` representa o número de valores dentro da sequência
- A sequência de valores em memória é percorrida pelos índices `0` a `n-1`

Sequência de valores de 16 bits em memória

- Endereço base: 0x1000

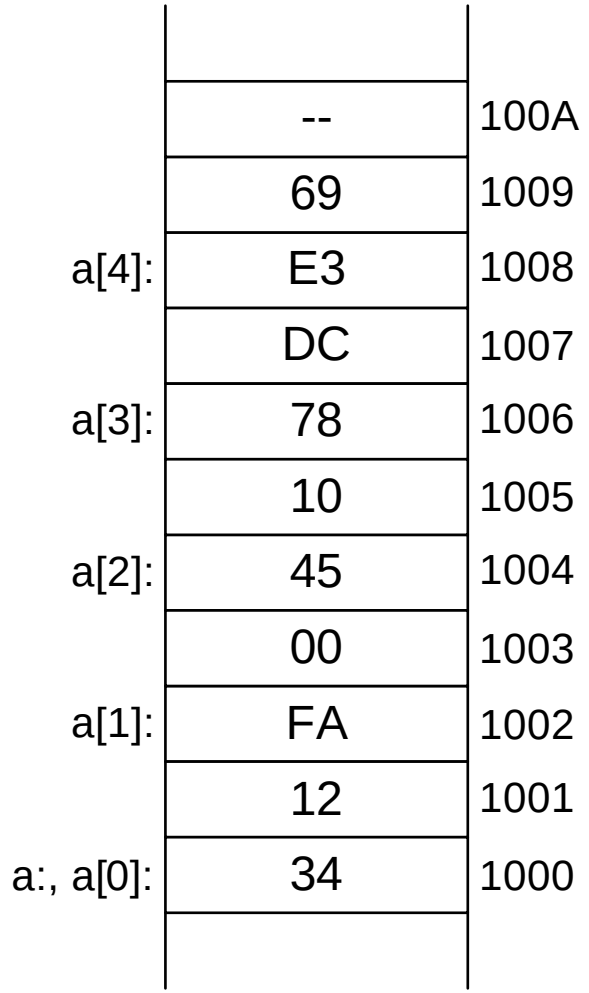
- Exemplo para 5 valores inteiros

  0x1234, 0x00FA, 0x1045, 0xDC78, 0x69E3

- Para a sequência de 5 inteiros em memória

  ```
  indexof_max(a, 5);
  ```

  a função retorna o índice 4

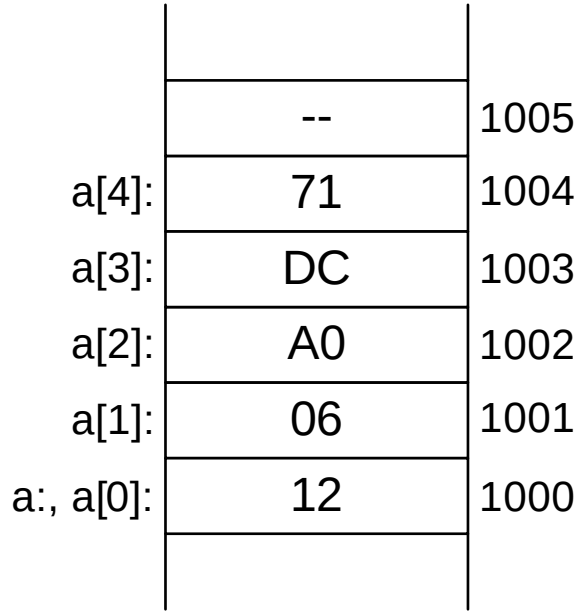


Sequência de valores de 8 bits em memória

- Endereço base: 0x1000

- Exemplo para 5 valores inteiros

  0x12, 0x06, 0xA0, 0xDC, 0x71



Algoritmo a implementar

```
uint8_t indexof_max(
        int16_t a[], uint8_t n) {
   uint8_t idx = 0, i = 0;
   uint16_t val = UINT16_MIN;
   while (i < n) {
      uint16_t tmp = abs(a[i]);
      if (val < tmp) {
         val = tmp;
         idx = i;
      }
      i += 1;
   }
   return idx;
}
```

```
/*-------------------------------------
 * A variável aux foi definida com o
 * objetivo de explorar a definição de
 * uma variável local no âmbito de uma
 * função folha
 * ------------------------------------*/

uint16_t abs(int16_t v) {
   uint16_t aux;
   if (v < 0)
      aux = -v;
   else
      aux = v;
   return aux;
}
```

Tipos convencionados

Naturais		Relativos	
uint8_t	8 bits sem sinal	int8_t	8 bits com sinal
uint16_t	16 bits sem sinal	int16_t	16 bits com sinal
uint32_t	32 bits sem sinal	int32_t	32 bits com sinal
uint64_t	64 bits sem sinal	int64_t	64 bits com sinal

- Mapeamento de tipos em registos
  - [u]int8_t: um registo nos bits de 0 a 7
  - [u]int16_t: um registo completo
  - [u]int32_t: dois registos completos
  - [u]int64_t: quatro registos completos
  - char: representa um carácter codificado em ASCII (8 bits)
- Dificilmente será usado o tipo de 64 bits no âmbito desta arquitetura

Problemas a resolver

- Como definir uma função?
- Como são passados parâmetros a uma função/como são recebidos parâmetros de uma função?
- Como são retornados valores de uma função?
- Como se invoca uma função?
- Como se retorna de uma função?
- Como são representadas as variáveis locais de uma função?

Definição de uma função

```
indexof_max:
    … ; corpo da função
    … ; retorno da função

abs:
    … ; corpo da função
    … ; retorno da função
```

Problemas a resolver

- Como definir uma função?
- Como são passados parâmetros a uma função/como são recebidos parâmetros de uma função?
- Como são retornados valores de uma função?
- Como se invoca uma função?
- Como se retorna de uma função?
- Como são representadas as variáveis locais de uma função?

Convenções na passagem de parâmetros

- Convenções definidas no âmbito desta arquitetura
- Parâmetros passados em registos do CPU e, se necessário, em memória
- Usados, no máximo, 4 registos (R0 a R3)
  - O primeiro parâmetro é passado em R0, o segundo em R1, e assim sucessivamente
  - Um parâmetro de dimensão inferior a 16 bits ocupa na mesma um registo por completo; o valor está codificado sempre nos bits de menor peso do registo
  - Um parâmetro de 32 bits usa dois registos: LSW no registo de índice inferior
  - Um *array*/*string*/… é passado como parâmetro através do seu endereço base
- Se forem necessários mais do que 4 registos, os restantes parâmetros são passados em memória, através da estrutura *stack*
- Os parâmetros são empilhados no *stack* da esquerda para a direita

Exemplos de passagem de parâmetros

```
void f1(uint8_t a, int8_t b, int16_t c, uint16_t d);
           r0          r1         r2          r3

void f2(uint8_t a, uint16_t b, int8_t c, int16_t d, int8_t e, int16_t f);
           r0          r1         r2          r3        stack     stack

void f3(uint8_t a, uint32_t b, int8_t c);
           r0         r2:r1       r3

void f4(uint8_t array[], int8_t dim);
           r0                r1

void f5(int16_t array[], int8_t dim);
           r0                r1
```

Problemas a resolver

- Como definir uma função?
- Como são passados parâmetros a uma função/como são recebidos parâmetros de uma função?
- Como são retornados valores de uma função?
- Como se invoca uma função?
- Como se retorna de uma função?
- Como são representadas as variáveis locais de uma função?

Convenções no retorno de valores

- Uma função `void` não retorna qualquer valor
- Usado o registo R0 para retornar valores constituídos até 16 bits
  - O valor está codificado sempre nos bits de menor peso do registo
- Usado o par de registos R1:R0 para retornar valores constituídos entre 17 e 32 bits
  - LSW (*Least Significant Word*) no registo R0
- Para valores superiores a 32 bits, é usado um parâmetro adicional na função com o endereço base onde o valor a retornar deverá ser escrito em memória
- As convenções na passagem de parâmetros não influenciam as convenções no retorno de valores

Exemplos de retorno de valores

```
void f1(); // não retorna qualquer valor,
           // logo não usa qualquer registo para retorno


uint8_t f2();
   r0


int16_t f3();
   r0


int32_t f4();
 r1:r0
```

Problemas a resolver

- Como definir uma função?
- Como são passados parâmetros a uma função/como são recebidos parâmetros de uma função?
- Como são retornados valores de uma função?
- Como se invoca uma função?
- Como se retorna de uma função?
- Como são representadas as variáveis locais de uma função?

Chamada de uma função (#1)

- A chamada de uma função pode ser resolvida com a instrução *Branch* (B fx)?

`void f1() {`
`  …`
`  f2();`
`  …`
`}`
	`f1:`
	`  …`
	`  b f2`
	`f2_ret:`
	`  … ; retorno de f1`
		`f2:`
		`  … ; corpo da função`
		`  b f2_ret ; retorno de f2`

- Aparentemente, *parece* resolver…

Chamada de uma função (#2)

- Mas e se a função for chamada em vários sítios?

```
void f1() {
   …
   f2();
   …
   f2();
   …
}
```

```
f1:
   …
   b f2
f2_ret:
   …
   b f2
f2_ret2:
   … ; retorno de f1
```

```
f2:
   … ; corpo da função
   b f2_ret ; retorno de f2
```

- Com a instrução *Branch*, a função `f2` consegue retornar apenas para um endereço destino; não é possível alterar o *offset* da instrução `B` em `f2` em tempo de execução
- Relevante a retenção do valor do registo PC no momento da chamada para capacitar o retorno para a instrução a seguir à que provocou a respetiva chamada

Chamada de uma função (#3)

- Hipótese com instruções conhecidas

```
void f1() {
   …
   f2();
   …
   f2();
   …
}
```

```
f1:
   …
   mov r0, pc
   add r14, r0, #4
   b f2
   …
   mov r0, pc
   add r14, r0, #4
   b f2
   …
   mov pc, r14 ; retorno de f1
```

```
f2:
   … ; corpo da função
   mov pc, r14 ; retorno de f2
```

- A chamada é realizada à custa de 3 instruções
- Para retornar de f1 (função não folha), o valor do registo R14 precisa ser preservado

Chamada de uma função (#4)

- Instrução para chamar uma função (*Branch and Link*)
  `BL offset; R14 (LR) = PC, PC = PC + offset*2`
- O registo R14 tem dupla funcionalidade: na chamada de uma função armazena o endereço de retorno
- R14 = LR = *Link Register*: o compilador traduz o símbolo LR pelo código 1110_2

```
void f1() {
   …
   f2();
   …
   f2();
   …
}
```

```
f1:
   …
   bl f2
   …
   bl f2
   …
   mov pc, lr ; retorno de f1
```

```
f2:
   … ; corpo da função
   mov pc, lr ; retorno de f2
```

- Para retornar de f1 (função não folha), o valor do registo LR precisa ser preservado

Problemas a resolver

- Como definir uma função?
- Como são passados parâmetros a uma função/como são recebidos parâmetros de uma função?
- Como são retornados valores de uma função?
- Como se invoca uma função?
- Como se retorna de uma função?
- Como são representadas as variáveis locais de uma função?

Retorno de uma função

- O retorno de uma função folha consiste em transferir o valor do registo LR para o registo PC (visto no contexto da chamada de uma função)
- Mas e se a função for uma função não folha? O valor do registo LR é esmagado por novo valor aquando a chamada da função interna
- É imperativo preservar o valor do registo LR antes de chamar a função interna. Onde?
  - Registos?
  - Memória?
- A compreensão do retorno de uma função não folha apenas será possível depois de serem tratados os seguintes temas:
  - Convenções na preservação de registos
  - Estrutura de dados *stack*

Convenções na preservação de registos

- Os valores dos registos R0 a R3 podem ser alterados no corpo da função sem preservar os valores que tinham à entrada da função
  - Na perspetiva de quem chama a função, após o retorno, os registos R0 a R3 podem trazer valores diferentes daqueles que tinham antes da respetiva chamada (o mesmo acontece para os registos CPSR e R14 (LR))
- Os valores dos registos R4 a R12 à entrada de uma função devem ser preservados de forma a que tenham os mesmos valores ao retornar da respetiva função
  - Na perspetiva de quem chama a função, após o retorno, os registos R4 a R12 trazem os mesmos valores que tinham antes da respetiva chamada
- Em resumo:
  - É da responsabilidade do chamador/*caller* guardar os valores dos registos R0 a R3, se necessário
  - É da responsabilidade do chamado/*callee* guardar os valores dos registo R4 a R12, se os usar
- A pergunta mantém-se: onde guardar o valor dos registos?

Estrutura de dados *Stack* (pilha)

- Estrutura de dados em memória que representa uma pilha de dados
- Útil para armazenar dados temporariamente: passagem de parâmetros; preservação de registos dentro de uma função; variáveis locais
- O acesso ao *stack* é realizado por uma das extremidades, denominado por Topo
  - Empilhar: adicionar um novo elemento ao topo da pilha
  - Desempilhar: remover o elemento do topo da pilha
  - A remoção de elementos da pilha acontece pela ordem inversa da inserção dos respetivos elementos (estrutura de dados do tipo LIFO – *Last In First Out*)
- Tipicamente, um registo do CPU é responsável por servir de ponteiro para o topo da pilha (no P16 esse registo é o R13 (SP = *Stack Pointer*))
- Filosofias na implementação de um *satck*:
  - *Full/Empty ascending* ou *Full/Empty descending*

Filosofias na implementação de um *satck*

- *Full*: o registo SP aponta para o elemento que está no topo
- *Empty*: o registo SP aponta para a primeira posição de memória a seguir ao topo (que está *vazia*)
- *Ascending*: empilhar implica crescer o topo para endereços maiores
- *Descending*: empilhar implica crescer o topo para endereços menores

Filosofia	Empilhar	Desempilhar
Full ascending	Mem[++SP] = val	val = Mem[SP--]
Empty ascending	Mem[SP++] = val	val = Mem[--SP]
Full descending	Mem[--SP] = val	val = Mem[SP++]
Empty descending	Mem[SP--] = val	val = Mem[++SP]

- No P16 é implementada a filosofia *Full descending*

Instruções para manipulação de dados no *stack*

- Empilhar no topo do *stack*

  `PUSH Rs ; ≡ SUB sp, sp, 2 (primeiro) e STR Rs, [sp] (segundo)`

- Desempilhar do topo do *stack*

  `POP Rd ; ≡ LDR Rd, [sp] (primeiro) e ADD sp, sp , 2 (segundo)`

- As instruções `PUSH` e `POP` atualizam implicitamente o registo SP
- O registo R13 (SP) está comprometido com a manipulação de dados no *stack*
- No P16, a granularidade dos dados a empilhar/desempilhar no topo do *stack* é sempre *Word*

Problemas a resolver

- Como definir uma função?
- Como são passados parâmetros a uma função/como são recebidos parâmetros de uma função?
- Como são retornados valores de uma função?
- Como se invoca uma função?
- Como se retorna de uma função?
- Como são representadas as variáveis locais de uma função?

Variáveis locais

- Deve-se dar preferência à utilização de registos do CPU para mapear variáveis locais
- Diferentes estratégias função de se tratar de uma função folha ou de uma função não folha

Função folha	Função não folha
➢ dar preferência aos registos R0 a R3 ➢ usar apenas os registos R4 a R12 quando os registos anteriores estiverem esgotados (porque R4 a R12 precisam ser preservados)	➢ usar registos R0 a R3 se durante o tempo de vida das variáveis que representam não existam chamadas a funções ➢ usar registos R4 a R12 se durante o tempo de vida das variáveis que representam existirem chamadas a funções, principalmente se enquadradas num ciclo

- Usar memória (*stack*) apenas quando todos os registos estiverem esgotados

Finalmente! Retorno de função não folha

```
void f1() {
   …
   f2();
   …
   f2();
   …
}
```

```
f1:
   push lr ; preserva LR no
           ; topo da pilha
   …
   bl f2
   …
   bl f2
   …
   pop pc ; retorno de f1:
          ; remove endereço de
          ; retorno do topo da
          ; pilha e afeta PC
```

```
f2:
   … ; corpo da função
   mov pc, lr ; retorno de f2
```

Finalmente! Variáveis locais e preservação de registos
Função folha

- Neste caso não é necessário preservar qualquer registo
  - Parâmetro v em R0
  - Variável local aux em R1
  - Retorno em R0
  - O LR não precisa ser guardado por se tratar de uma função folha

```
/*-------------------------------------
 * A variável aux foi definida com o
 * objetivo de explorar a definição de
 * uma variável local no âmbito de uma
 * função folha
 * ------------------------------------*/

uint16_t abs(int16_t v) {
   uint16_t aux;
   if (v < 0)
      aux = -v;
   else
      aux = v;
   return aux;
}
```

Finalmente! Variáveis locais e preservação de registos
Função não folha

- Variáveis a mapear de R4 a R12
  - `a`, `n`, `ixd`, `i`, `val` porque é invocada a função `abs` durante os seus tempos de vida
  - Os registos usados para mapear estas variáveis devem ser empilhados no *stack* no prólogo da função antes de serem iniciados com os valores das variáveis
  - Os mesmos registos devem ser desempilhados do topo do *stack* no epílogo da função pela ordem inversa
- Variáveis a mapear de R0 a R3
  - `tmp`, porque durante o seu tempo de vida não está envolvida qualquer chamada a função
- O LR deve ser igualmente empilhado no *stack*
  - Sendo o último valor a ser desempilhado para retornar da função, deve ser o primeiro registo a ser empilhado
- Parâmetro `a` em R0 e `n` em R1
  - Transferir os valores de R0 e R1 para os registos que mapeiam a e n depois de empilhados no *stack*
- Retorno em R0
  - Transferir o valor do registo que mapeia `idx` para R0 depois do ciclo `while` ser quebrado

```
uint8_t indexof_max(
        int16_t a[], uint8_t n) {
   uint8_t idx = 0, i = 0;
   uint16_t val = UINT16_MIN;
   while (i < n) {
      uint16_t tmp = abs(a[i]);
      if (val < tmp) {
         val = tmp;
         idx = i;
      }
      i += 1;
   }
   return idx;
}
```

Implementação da função **abs**

```
; In: R0=v
; Out: R0=abs(v)
abs:
   mov  r1, #0
   cmp  r0, r1
   bge  abs_endif
   sub  r0, r1, r0
abs_endif:
   mov  pc, lr
```

```
uint16_t abs(int16_t v) {
   uint16_t aux;
   if (v < 0)
      aux = -v;
   else
      aux = v;
   return aux;
}
```

Implementação da função `indexof_max` (#1)

```
; In: R0=a, R1=n
; Out: R0=indíce do maior absoluto
indexof_max:
   ; prólogo
   push lr
   push r4
   push r5
   push r6
   push r7
   push r8
   … ; iniciação de variáveis locais
   … ; corpo da função
   … ; epílogo e retorno
```

```
uint8_t indexof_max(
        int16_t a[], uint8_t n) {
   uint8_t idx = 0, i = 0;
   uint16_t val = UINT16_MIN;
   while (i < n) {
      uint16_t tmp = abs(a[i]);
      if (val < tmp) {
         val = tmp;
         idx = i;
      }
      i += 1;
   }
   return idx;
}
```

Implementação da função `indexof_max` (#2)

```
   .equ UINT16_MIN, 0
; In: R0=a, R1=n
; Out: R0=indíce do maior absoluto
indexof_max:
   … ; prólogo
   ; iniciação de variáveis locais
   mov  r4, r0    ; R4=a
   lsl  r8, r1, 1 ; R8=n
   mov  r6, #0    ; R6=idx
   mov  r7, #0    ; R7=i
   mov  r5, #UINT16_MIN & 0xFF; R5=val
   movt r5, #UINT16_MIN >> 8;
   … ; corpo da função
   … ; epílogo e retorno
```

```
uint8_t indexof_max(
        int16_t a[], uint8_t n) {
   uint8_t idx = 0, i = 0;
   uint16_t val = UINT16_MIN;
   while (i < n) {
      uint16_t tmp = abs(a[i]);
      if (val < tmp) {
         val = tmp;
         idx = i;
      }
      i += 1;
   }
   return idx;
}
```

Implementação da função `indexof_max` (#3)

```
indexof_max: … ; prólogo + variáveis locais
   b    idxofmax_whilecond
idxofmax_whilebody:
   ldr  r0, [r4, r7]
   bl   abs
   cmp  r5, r0
   bhs  idxofmax_endif
   mov  r5, r0
   mov  r6, r7
idxofmax_endif:
   add  r7, r7, #2
idxofmax_whilecond:
   cmp  r7, r8
   blo  idxofmax_whilebody
   lsr  r0, r6, #1
   … ; epílogo e retorno
```

```
uint8_t indexof_max(
        int16_t a[], uint8_t n) {
   uint8_t idx = 0, i = 0;
   uint16_t val = UINT16_MIN;
   while (i < n) {
      uint16_t tmp = abs(a[i]);
      if (val < tmp) {
         val = tmp;
         idx = i;
      }
      i += 1;
   }
   return idx;
}
```

Implementação da função `indexof_max` (#4)

```
; In: R0=a, R1=n
; Out: R0=indíce do maior absoluto
indexof_max:
   … ; prólogo
   … ; iniciação de variáveis locais
   … ; corpo da função
   ; epílogo e retorno
   pop  r8
   pop  r7
   pop  r6
   pop  r5
   pop  r4
   pop  pc
```

```
uint8_t indexof_max(
        int16_t a[], uint8_t n) {
   uint8_t idx = 0, i = 0;
   uint16_t val = UINT16_MIN;
   while (i < n) {
      uint16_t tmp = abs(a[i]);
      if (val < tmp) {
         val = tmp;
         idx = i;
      }
      i += 1;
   }
   return idx;
}
```