Compiladores (CC3001)
Aula 13: Geração de código para funções

Mário Florido

DCC/FCUP

2024

[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

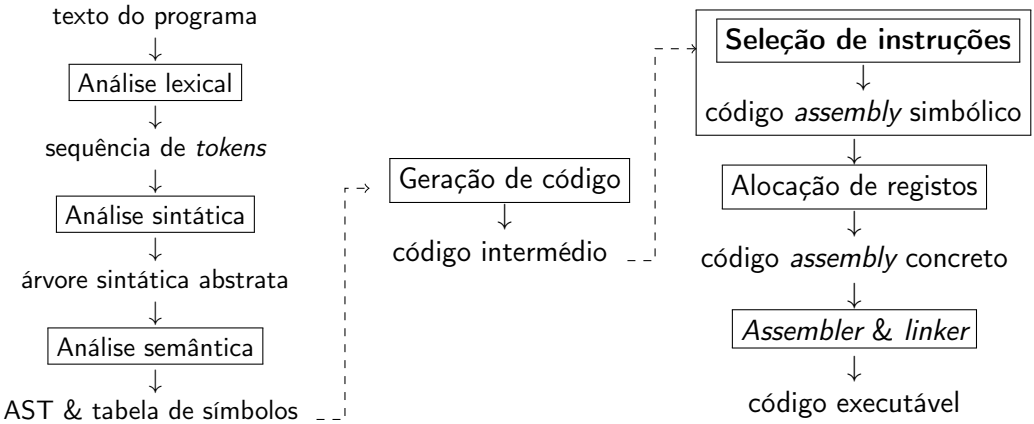
Esta aula

Chamada de funções

Registos de Ativação

Extras

Re-lembrar: fases dum compilador



Chamada de funções

Registos de Ativação

Extras

- Geramos código intermédio para cada função separadamente
- A chamada e retorno função foi tratada com duas instruções:

$$r := \texttt{CALL}\ f(t_1, t_2, \ldots, t_n)$$
$$\texttt{RETURN}\ t$$

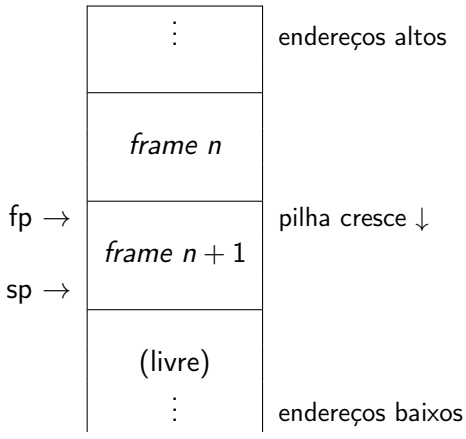
- Estas instruções não explicitam como implementar o controlo de fluxo e a passagem de parâmetros
- Vamos agora ver como implementar `CALL` e `RETURN` em código máquina

A pilha de execução mantém a informação que liga o ponto de chamada ("*caller*") e a função ("*callee*"):

- o endereço para retorno após execução da função;
- o conteúdo de registos antes da chamada (para ser restaurado após o retorno);
- possivelmente os parâmetros da função e valor de retorno;
- as variáveis locais à função que não cabem em registos (e.g. *arrays*).

Pilha de execução (cont.)

- Por razões históricas a pilha cresce de endereços mais altos para mais baixos
- O topo da pilha é indicado por `sp` (stack pointer)
- Endereços abaixo de `sp` são espaço livre
- A invocação de uma função reserva um registo de ativação (ou *frame*) contíguo
- O início do registo de ativação atual é indicado por `fp` (frame pointer)
- Quando a função termina, o seu registo de ativação é removido da pilha (*Last-In, First-Out*)



- A estrutura dos registos de ativação não é ditada pela arquitetura do processador
- Contudo: cada arquitetura ou sistema operativo define uma convenção de chamada que establece regras para passagem de argumentos e resultados
- Esta convenção permite combinar código gerado por compiladores e/ou linguagens diferentes
- No entanto: cada linguagem/compilador é livre de impletar uma convenção própria se tal for conveniente

Uma instrução CALL será traduzida por uma sequência de chamada:

- (possivelmente) guardar registos temporários na pilha;
- colocar argumentos da função na pilha e/ou registos;
- guardar o endereço de retorno;
- transferir a execução para o *callee*;
- repor eventuais registos guardados.

O código de cada função deve conter:

- uma sequência inicial (prelúdio) que atualiza *stack/frame pointer*, retira os argumentos da pilha para registos, e (possivelmente) preserva registos na pilha.
- uma sequência final (epílogo) que restaura registos guardados, coloca o valor de retorno (na pilha ou registos) e transfere execução para o *caller*.

A instrução `RETURN` pode apenas colocar o resultado num registo apropriado e saltar para o epílogo da função.

Supondo que dentro da função `g(...)` chamamos a função `f(...)`.

Quem deve guardar valores de registos na pilha entre chamada?

Supondo que dentro da função `g(...)` chamamos a função `f(...)`.

Quem deve guardar valores de registos na pilha entre chamada?

Duas estratégias:

Caller-saved `g` deve guardar os registos de contêm variáveis vivas antes de chamar `f`

Callee-saved `f` deve guardar os registos modificados no corpo da função antes de retornar a `g`

Supondo que dentro da função `g(...)` chamamos a função `f(...)`.

Quem deve guardar valores de registos na pilha entre chamada?

Duas estratégias:

Caller-saved `g` deve guardar os registos de contêm variáveis vivas antes de chamar `f`

Callee-saved `f` deve guardar os registos modificados no corpo da função antes de retornar a `g`

Podemos usar uma estratégia mista: alguns registos são *caller-saved* e outros são *callee-saved*.

A passagem de parâmetros e resultados pela pilha obriga a transferências frequentes entre registos e memória (lentas).

Para evitar isso, as convenções de chamada permitem passar alguns argumentos e resultados em registos:

- um subconjunto de registos *caller-saved* para os primeiros 4–8 argumentos; os restantes argumentos (se existirem) são passados na pilha;
- alguns registos (possivelmente os mesmos) para retornar resultados;
- frequentemente o endereço de retorno é também passado num registo.

- registos $a0–$a3 passam os primeiros 4 argumentos da função (restantes na pilha)
- registos $v0–$v1 passam o resultado do *callee*
- registo $ra passa o endereço de retorno
- registos $t0-$t9 são *caller-saved* (podem ser escritos pelo *callee*)
- registos $s0-$s7 são *callee-saved* (se forem usados pelo *callee* devem ser guardados na pilha e repostos)

Nesta apresentação vamos simplificar a convenção e assumir que todos os argumentos são passados na pilha.

(Podemos usar os registos $a0–$a3 como temporários no *callee*.)

Esta aula

Chamada de funções

Registos de Ativação

Extras

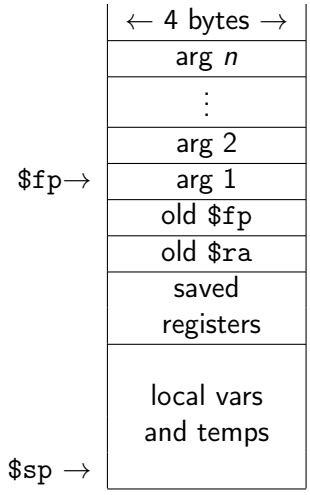
Layout do registo de ativação

- Os argumentos têm deslocamentos não-negativos relativos a `$fp`

  arg_1 está em `0($fp)`;
  arg_2 está em `4($fp)`;
  arg_3 está em `8($fp)`;
  ...

- Registos *callee-saved* e variáveis locais têm deslocamentos negativos:

  `$fp` anterior em `-4($fp)`
  `$ra` anterior em `-8($fp)`
  outros registos e variáveis locais em `-12($fp)`, `-16($fp)`, etc.



Uma instrução de código intermédio

$$t := \texttt{CALL}\ F(r_1, \ldots, r_n)$$

será traduzida pela sequência:

```
⋮                                    # save registers $t0–$t9 (if needed)
sw      r_n,      -4($sp)            # store arg n
sw      r_{n-1},  -8($sp)            # store arg n − 1
⋮                                    ⋮
sw      r_1       -k($sp)            # store arg 1 (k = 4n)
addiu   $sp,      $sp, -k            # grow stack
jal     label_F                      # jump and link ($ra points to next instruction)
addiu   $sp,      $sp, k             # shrink stack
⋮                                    # restore registers $t0–$t9
move    t,        $v0                # save result
```

A definição de uma função

$$F(\ldots)\ [$$
$$\vdots$$
$$]$$

será traduzida por um bloco de código:

```
label_F :                                  # entry label for F
        sw      $fp,  -4($sp)              # save old $fp
        sw      $ra,  -8($sp)              # save return address
        move    $fp,  $sp                  # setup frame pointer
        addiu   $sp,  $sp, -n              # stack space for temps
        ⋮                                  # save registers $s0–$s7 (if needed)
        ⋮                                  # function code
```

A instrução de código intermédio

RETURN *r*

será traduzida pela sequencia:

```
                move  $v0,  r          # store result
return_F :
                ⋮                      # restore registers $s0–$s7 (if needed)
                move  $sp,  $fp        # restore stack pointer
                lw    $ra,  -8($sp)    # restore return address
                lw    $fp,  -4($sp)    # restore frame pointer
                jr    $ra              # return
```

Se a função tiver múltiplos RETURN: podemos partilhar o código de $return_F$.

Exemplo

```
int add(int x, int y) {
  return x+y+3;
}
```

```
add:                        # prelúdio
      sw $fp, -4($sp)
      sw $ra, -8($sp)
      move $fp, $sp
      addiu $sp, $sp, -8

      lw $a0, 0($fp)        # $a0 := x
      lw $a1, 4($fp)        # $a1 := y
      add $t0, $a0, $a1     # $t0 := $a0 + $a1
      addi $t0, $t0, 3      # $t0 := $t0 + 3
                            # epílogo
      move $v0, $t0         # RETURN $t0
      move $sp, $fp
      lw $ra, -8($sp)
      lw $fp, -4($sp)
      jr $ra
```

Esta aula

Chamada de funções

Registos de Ativação

Extras

A *chamada* pode usar a convenção normal

A *implementação* pode ser diretamente em código máquina

Exemplo: funções de I/O podem ser implementadas usando *syscalls* MIPS.

```
print_int:                  # print_int(n):
        li $v0, 1           # syscall 1
        lw $a0, 0($sp)      # fetch argument from stack
        syscall
        jr $ra              # return

read_int:                   # read_int():
        li $v0, 5           # syscall 5
        syscall             # result in $v0
        jr $ra              # return
```

O *caller* só necessita de guardar registos cujos valores vão ser necessários (veremos "liveness analysis")

O *callee*
- só necessita de guardar $ra se chamar outras funções
- se não necessitar de espaço na pilha pode mesmo omitir o código para criar a *frame*

Exemplo:

```
int add(int x, int y) {
    return x+y+3;
}
```

Esta função não chama outras e usa apenas registos temporários — podemos omitir a criação da *frame* (slide seguinte).

Optimizações de chamada (cont.)


```
add:
        sw $fp, -4($sp)
        sw $ra, -8($sp)
        move $fp, $sp
        addiu $sp, $sp, -8

        lw $a0, 0($fp)
        lw $a1, 4($fp)
        add $t0, $a0, $a1
        addi $t0, $t0, 3
        move $v0, $t0

        move $sp, $fp
        lw $ra, -8($sp)
        lw $fp, -4($sp)
        jr $ra
```

```
add:
        sw $fp, -4($sp)
        sw $ra, -8($sp)
        move $fp, $sp
        addiu $sp, $sp, -8

        lw $a0, 0($sp)
        lw $a1, 4($sp)
        add $t0, $a0, $a1
        addi $t0, $t0, 3
        move $v0, $t0

        move $sp, $fp
        lw $ra, -8($sp)
        lw $fp, -4($sp)
        jr $ra
```

O espaço para *arrays* locais deve ser reservado no registo de ativação da função.

```
int f(...) {
   int x[5];
   ...
}
```

```
labelF:  ...
         addiu $sp, $sp, -20  # reservar 5*4 bytes
         move $t1, $sp        # x : reg $t1
         # x[0] -> 0($t1)
         # x[1] -> 4($t1)
         # x[2] -> 8($t1)
         # etc.
```

Arrays locais (cont.)

O tamanho do *array* também pode ser calculado dinamicamente.

```
int f(int n) {
   int x[n];
   ...
}
```

```
labelF:                          # supondo n : reg $a0
        move $t0, $a0
        sll $t0, $t0, 2          # shift left 2
        sub $sp, $t0, $sp        # reservar 4*n bytes
        move $t1, $sp            # x : reg $t1
        # x[0] -> 0($t1)
        # x[1] -> 4($t1)
        # etc.
```

- O registo de ativação da função é removido no retorno
- Assim *arrays* locais não sobrevivem à invocação da função
- Exemplo: o seguinte código é errado (retorna um "dangling pointer").

```
int[] f(...) {
   int x[5];
   ...
   return x;
}
```