Compiladores (CC3001)
Aula 12: Geração de código máquina

Mário Florido

DCC/FCUP

2024

[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Esta aula

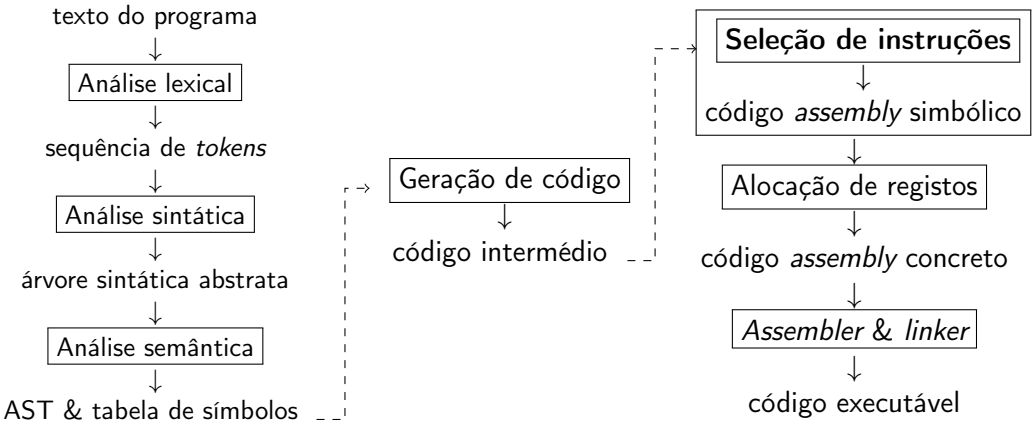
Código máquina simbólico

Código intermédio vs. código máquina

Padrões de instruções

Geração de código máquina

Re-lembrar: fases dum compilador



Esta aula

Código máquina simbólico

Código intermédio vs. código máquina

Padrões de instruções

Geração de código máquina

Código máquina simbólico (“assembly”)

- Representação textual de código máquina
- Mais legível do que código binário
- Etiquetas simbólicas para endereços e constantes
- Transformada em código máquina executável por um programa *assemblador*

- Arquitetura RISC
- 32 registos inteiros \$0 – \$31 de 32-bits
- Conjunto de instruções *load/store*
- Instruções de tamanho fixo (32-bits)
- Operações entre 3 registos ou registos e constantes (*immediate*)
- Relembrar Arquitetura de Computadores:
  `https://www.dcc.fc.up.pt/~ricroc/aulas/2324/ac/`

$0	sempre zero ($zero)
$1	temporário do *assembler*
$2–$3	valores de retorno ($v0–$v1)
$4–$7	argumentos ($a0-$a3)
$26–$27	reservados para *kernel*
$28	global pointer ($gp)
$29	stack pointer ($sp)
$30	frame pointer ($fp)
$31	return address ($ra)
$8–$25	temporários $t0–$t7, $s0–$s7 e $t8–$t9

Programa de exemplo

```
            .data               # dados
val:        .word 10, -14, 30   # array de constantes
str:        .asciiz "Hello!"    # string terminada em zero
_heap_:     .space 10000        # reservar 10k bytes
            .text               # instruções
            .global main        # main é visível do exterior
            la $gp, _heap_      # inicializar reg $28 = _heap_
            jal main            # jump and link (reg $31)
_stop_:
            li $2,10            # reg $2 = 10
            syscall             # syscall 10: exit
main:
            la $8, val          # reg $8 = val
            lw $9, 4($8)        # reg $9 = val[1]
            addi $9, $9, 4      # reg $9 = reg $9 + 4
            sw $9, 8($8)        # val[2] = reg $9
            jr $ra              # jump register $31 (return address)
```

Esta aula

Código máquina simbólico

Código intermédio vs. código máquina

Padrões de instruções

Geração de código máquina

1. O código máquina tem um número finito de registos
2. Não existe instrução CALL, i.e. tem de ser implementada com registos e pilha
3. As instruções de saltos condicionais têm só um destino
4. Pode ser necessário efetuar a comparação separadamente do salto condicional
5. Arquiteturas RISC só permitem instruções com constantes pequenas

O pontos 1 e 2 serão abordados separadamente (alocação de registos e implementação de chamada de funções).

Hoje vamos ver os pontos 3–5.

Saltos condicionais com dois destinos

`COND` *c* *labelt* *labelf* pode ser traduzido como

```
branch_if_c    labelt
jump           labelf
```

`COND` *c* *labelt* *labelf* pode ser traduzido como

```
        branch_if_c    labelt
        jump           labelf
```

Se a próxima instrução for *labelt* ou *labelf* podemos eliminar um salto:

```
  COND c labelt labelf
  LABEL labelt
  ...
```

pode ser traduzido como

```
         branch_if_not_c labelf
labelt: ....
```

Saltos condicionais com dois destinos

`COND` *c* *labelt* *labelf* pode ser traduzido como

```
        branch_if_c    labelt
        jump           labelf
```

Se a próxima instrução for *labelt* ou *labelf* podemos eliminar um salto:

```
  COND c labelt labelf
  LABEL labelt
  ...
```

pode ser traduzido como

```
         branch_if_not_c labelf
labelt: ....
```

(Mantemos *labelt* porque pode ser referido noutra instrução.)

Em muitas arquiteturas as comparações devem ser efetuadas separadamente do salto: primeiro efetuamos a comparação e depois efetuamos salto.

Exemplos:

- Em MIPS comparações $=$ e $\neq$ são diretas (`beq` e `bne`) mas $<$, $>$ etc. devem calcular a condição num registo (`slt`, `sgt`, etc.)
- Em ARM e X86 as operações aritméticas marcam *flags* (zero, negativo, *overflow*, *carry*,...)

Comparação $=$ em MIPS:

```
beq $t1, $t2, ltrue    # se $t1==$t2 ir para ltrue
...                    # caso contrário
```

Comparação $<$ em MIPS:

```
slt $1, $t1, $t2        # comparar $t1 < $t2 e guardar em $1
bne $1, $zero, ltrue    # se $1 diferente de zero ir para ltrue
...                     # caso contrário
```

(Usamos o registo $1 como temporário.)

Frequentemente as instruções de código máquina permitem apenas constantes de tamanho limitado:

- MIPS suporta apenas constantes de 16 bits; para constantes maiores usamos `lui` para carregar 16 bits na metade superior de um registo
- ARM suporta apenas constantes de 8 bits que podem ser colocadas em diferentes posições de um registo

A geração de código deve decompor em várias instruções se necessário.

Por exemplo, a atribuição

```
t0 := 0x87fe000
```

corresponde a 2 instruções MIPS (usando $1 como temporário):

```
lui $1,0x87            # $1 = 0x870000
ori $t0, $1, 0xfe00    # $t0 = 0x87fe00
```

- MIPS suporta *pseudo-instruções* que facilitam a geração de código, e.g.
  - `blt`, `bgt`, `ble`, `bge` (*branch*) com condições $<$, $>$, $\leq$, $\geq$
  - `li` (*load immediate*) com constantes $>$ 16 bits
  - `move` entre registos
- Não são implementadas em *hardware*
- Traduzidas para sequências de instruções básicas pelo assemblador; e.g.:

pseudo-instrução	tradução
`move $t1, $t2`	`addu $t1, $zero, $t2`
`li $t0, 0x87fe00`	`lui $1,0x87` `ori $t0, $1, 0xfe00`
`blt $t0, $t1, label`	`slt $1, $t0, $t1` `bne $1, $zero, label`

Esta aula

Código máquina simbólico

Código intermédio vs. código máquina

Padrões de instruções

Geração de código máquina

- Vamos traduzir padrões de instruções intermédias em instruções de código máquina
- Preferimos padrões mais longos se possível (mais vantajoso)
- Se não for possível: padrões para instruções individuais

Tradução de padrões para MIPS (1)

COND $r_s = r_t$ $label_t$ $label_f$, LABEL $label_f$		beq	r_s, r_t, $label_t$
	$label_f$:		
COND $r_s = r_t$ $label_t$ $label_f$, LABEL $label_t$		bne	r_s, r_t, $label_f$
	$label_t$:		
COND $r_s = r_t$ $label_t$ $label_f$		beq	r_s, r_t, $label_t$
		j	$label_f$
COND $r_s < r_t$ $label_t$ $label_f$, LABEL $label_f$		blt*	r_s, r_t, $label_t$
	$label_f$:		
COND $r_s < r_t$ $label_t$ $label_f$, LABEL $label_t$		bge*	r_s, r_t, $label_f$
	$label_t$:		
COND $r_s < r_t$ $label_t$ $label_f$		blt*	r_s, r_t, $label_t$
		j	$label_f$
LABEL $label$	$label$:		
JUMP $label$		j	$label$

(* são *pseudo*-instruções.)

Tradução de padrões para MIPS (2)

$r_d := r_s + r_t$	`add` r_d, r_s, r_t
$r_d := r_s + k$	`addi` r_d, r_s, k
$r_d := r_s * r_t$	`mul` r_d, r_s, r_t
$r_d := r_t$	`move` r_d, r_t
$r_d := k$	`li` r_d, k
$r_t := M[r_s]$	`lw` r_t, $0(r_s)$
$M[r_s] := r_t$	`sw` r_t, $0(r_s)$

É necessário tratar todas as possibilidades do codigo intermédio (e.g. as restantes operações aritméticas).

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
```

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
```

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
```

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
```

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
```

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
    move $t3, $t1
```

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
    move $t3, $t1
label3:
```

Exemplo

Código intermédio

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

Código MIPS

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
    move $t3, $t1
label3:
```

- Seguimos um algoritmo guloso:
  1. traduzir o maior padrão que encaixa o prefixo de código;
  2. repetir para o restante
- Alternativa: atribuir custos a cada instrução de máquina; encontrar o encaixe que minimiza o custo total usando programação dinâmica
- A primeira abordagem é mais simples e funciona bem em arquiteturas RISC

- O cálculo de endereços necessita de multiplicar índices pelo tamanho dos elementos (e.g. 4 bytes para inteiros 32-bits)
- Se o tamanho for potência de 2: em vez de multiplicação podemos usar uma operação de *left shift* (mais eficiente)
- Exemplo: podemos traduzir `t1 := t1 * 8` por

  ```
  sll $t1, $t1, 3
  ```

  (porque $8 = 2^3$)
- Caso geral: traduzimos

  $$r_d := r_s * 2^k$$

  por

  $$\texttt{sll}\ r_d,\ r_s,\ k$$

Exemplo

Consider a seguinte atribuição em C:

```
int x[N], i;
...
x[i] = 5;
```

(Assumindo $[\mathtt{x} \mapsto t_0, \mathtt{i} \mapsto t_1]$.)

Código intermédio:

```
t2 := t1
t2 := t2*4
t2 := t2 + t0
t4 := 5
M[t2] := t4
```

Código MIPS:

```
move $t2, $t1
sll $t2, $t2, 2
add $t2, $t2, t0
li $t4, 5
sw $t4, 0($t2)
```

Muitas arquiteturas permitem combinar várias operações numa só instrução de máquina:

- operações *load/store* que efetuam cálculos de endereços;
- operações aritméticas combinando *add* e *shifts*;
- instruções que transferem multiplos registos de/para memória

Podemos traduzir várias instruções intermédias $\rightarrow$ uma instrução de máquina.

As duas instruções intermédias:

$$t2 := t1 + 8$$
$$t3 := M[t2]$$

podem ser combinadas numa só instrução MIPS:

```
lw $t3, 8($t1)
```

As duas instruções intermédias:

$$t2 := t1 + 8$$
$$t3 := M[t2]$$

podem ser combinadas numa só instrução MIPS:

```
lw $t3, 8($t1)
```

Mas: apenas se o valor de `t2` não for mais usado!

Podemos descobrir essa informação por meio de análise semântica de *liveness* das variáveis (mais tarde).

Esta aula

Código máquina simbólico

Código intermédio vs. código máquina

Padrões de instruções

Geração de código máquina

- Nos padrões de código intermédios apresentados assumimos que as variáveis temporárias correspondem aos registos
- Para atribuir registos a temporários de forma ótima é necessária uma análise separada (alocação de registos)
- Alternativa mais simples:
  - re-utilizar temporários durante geração de código intermédio
  - atribuir cada temporário a um registo MIPS diferente
  - pode não compilar programas com expressões muito longas...
  - ...mas é suficiente para o projeto laboratorial (porque a arquitetura MIPS tem bastantes registos)

- O esquema de tradução para código intermédio introduz sempre variáveis intermédias para sub-expressões
- Mas sabemos que essas variáveis não serão mais necessárias
- Logo: podemos re-utilizar em expressões seguintes

transExp (*expr*, *table*, *dest*) = case *expr* of	
⋮	
e_1 **binop** e_2	$t_1 = newTemp()$ $t_2 = newTemp()$ $code_1 = transExp(e_1, table, t_1)$ $code_2 = transExp(e_2, table, t_2)$ $popTemp(2)$ (t_1, t_2 não vão ser mais necessários) return $code_1 ++ code_2 ++ [dest := t_1$ **binop** $t_2]$

Re-utilizar temporários (cont.)

A operação popTemp apenas decrementa o contador de temporários:

```
int temp_count = 0;  // contador global

int newTemp() {
    return temp_count ++;
}

void popTemp(int k) {
   temp_count -= k;
}
```

O número de temporários usados numa função será a profundidade máxima das expressões aritméticas (independente do número de instruções).

Exemplo

```
int f (int x, int y) {
  return 3*x + 2*y;
}
```

```
f(a0, a1) [
t3 := 3
t4 := a0
t1 := t3 * t4
t3 := 2
t4 := a1
t2 := t3 * t4
t0 := t1 + t2
RETURN t0
]
```

- Na tradução para código máquina podemos usar registos $t0–$t4 para temporários
- Na próxima aula: veremos como tratar argumentos e valores de retorno