Compiladores

Aula 26: Geração de Código de Máquina

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. Objetivos
2. O Back-End
3. Seleção e Escalonamento
4. Alocação de Registradores
5. Estudo de Caso
6. Referencias

Objetivos da Aula

- Compreender a Geração de Código como a fase final do *Back-End*.

Objetivos da Aula

- Compreender a Geração de Código como a fase final do *Back-End*.
- Entender o conceito de **Phase Ordering Problem** entre Seleção, Escalonamento e Alocação.

Objetivos da Aula

- Compreender a Geração de Código como a fase final do *Back-End*.
- Entender o conceito de **Phase Ordering Problem** entre Seleção, Escalonamento e Alocação.
- Explorar a **Seleção de Instruções** via algoritmos de *Tree Covering*.

Objetivos da Aula

- Compreender a Geração de Código como a fase final do *Back-End*.
- Entender o conceito de **Phase Ordering Problem** entre Seleção, Escalonamento e Alocação.
- Explorar a **Seleção de Instruções** via algoritmos de *Tree Covering*.
- Analisar o **Escalonamento de Instruções** para evitar *Hazards* no pipeline.

Objetivos da Aula

- Compreender a Geração de Código como a fase final do *Back-End*.
- Entender o conceito de **Phase Ordering Problem** entre Seleção, Escalonamento e Alocação.
- Explorar a **Seleção de Instruções** via algoritmos de *Tree Covering*.
- Analisar o **Escalonamento de Instruções** para evitar *Hazards* no pipeline.
- Estudar matematicamente a **Alocação de Registradores** e as heurísticas de Coloração de Chaitin-Briggs.

O Desafio do Back-End

A Geração de Código traduz a IR otimizada para a linguagem de montagem (*Assembly*) nativa. O Back-End é sustentado pela interação de três fases complexas:

1. **Seleção de Instruções:** Quais instruções de máquina usar?

O Desafio do Back-End

A Geração de Código traduz a IR otimizada para a linguagem de montagem (*Assembly*) nativa. O Back-End é sustentado pela interação de três fases complexas:

1. **Seleção de Instruções:** Quais instruções de máquina usar?
2. **Escalonamento de Instruções:** Em qual ordem emiti-las para maximizar o hardware?

O Desafio do Back-End

A Geração de Código traduz a IR otimizada para a linguagem de montagem (*Assembly*) nativa. O Back-End é sustentado pela interação de três fases complexas:

1. **Seleção de Instruções:** Quais instruções de máquina usar?
2. **Escalonamento de Instruções:** Em qual ordem emiti-las para maximizar o hardware?
3. **Alocação de Registradores:** Onde armazenar as variáveis físicas?

Phase Ordering Problem

O grande desafio teórico em compiladores é que essas três fases "brigam" entre si.

- **Escalonar antes de Alocar:**
  - Espalha o código para usar o pipeline da CPU.
  - Problema: Aumenta as variáveis simultaneamente ativas, estourando o limite de registradores (*Spilling* para a RAM).

Phase Ordering Problem

O grande desafio teórico em compiladores é que essas três fases "brigam" entre si.

- **Escalonar antes de Alocar:**
  - Espalha o código para usar o pipeline da CPU.
  - Problema: Aumenta as variáveis simultaneamente ativas, estourando o limite de registradores (*Spilling* para a RAM).
- **Alocar antes de Escalonar:**
  - Tenta espremer as variáveis no menor número de registradores físicos.
  - Problema: Reutiliza fisicamente registradores, criando falsas dependências que impedem o Escalonador de reordenar o código!

Seleção de Instruções

Mapear operações da IR para instruções da máquina afeta tamanho e velocidade.

- Exemplo: `a = b + (c * 8)` na IR.

Seleção de Instruções

Mapear operações da IR para instruções da máquina afeta tamanho e velocidade.

- Exemplo: `a = b + (c * 8)` na IR.
- Pode gerar três instruções *Assembly* simples: multiplicar, adicionar, mover.
- Mas no **x86**, podemos usar apenas uma instrução robusta: `lea rax, [rbx+rcx*8]`.

Seleção de Instruções

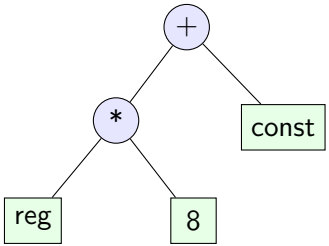
Mapear operações da IR para instruções da máquina afeta tamanho e velocidade.

- Exemplo: `a = b + (c * 8)` na IR.
- Pode gerar três instruções *Assembly* simples: multiplicar, adicionar, mover.
- Mas no **x86**, podemos usar apenas uma instrução robusta: `lea rax, [rbx+rcx*8]`.

Tree Covering (Cobertura de Árvore)
Convertendo a IR em florestas, cada instrução da CPU atua como um "ladrilho". O algoritmo **Maximal Munch** escolhe sempre gulozamente o maior ladrilho possível para cobrir a árvore, sendo rápido e eficiente.

Exemplo: Maximal Munch (Tree Covering)

Objetivo: Cobrir a árvore sintática da IR com ladrilhos (*tiles*) de instruções nativas de menor custo.



IR: `reg * 8 + const`

Abordagem Ingênua:

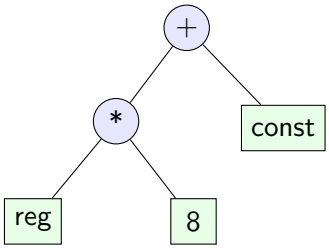
Gera uma instrução para cada nó isolado.

```
mul r1, r2, 8
add r1, r1, const
```

Custo: 2 instruções, maior tamanho de binário.

Exemplo: Maximal Munch (Tree Covering)

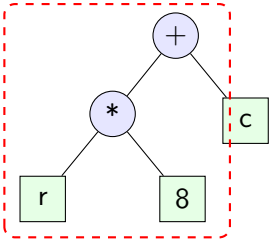
Objetivo: Cobrir a árvore sintática da IR com ladrilhos (*tiles*) de instruções nativas de menor custo.



IR: `reg * 8 + const`

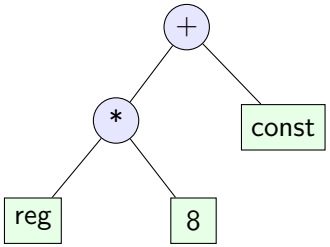
O Ladrilho x86 (`LEA`):

A arquitetura x86 possui a instrução `LEA` (Load Effective Address) que calcula nativamente o padrão `[reg * scale + disp]`!



Exemplo: Maximal Munch (Tree Covering)

Objetivo: Cobrir a árvore sintática da IR com ladrilhos (*tiles*) de instruções nativas de menor custo.



IR: `reg * 8 + const`

Resultado do Maximal Munch:

O algoritmo, ao descer pela raiz (+), encontra o ladrilho "gigante" do `LEA` e cobre a árvore inteira de uma só vez!

Código Gerado:

```
lea rax, [rbx * 8 + const]
```

Custo: 1 instrução ultra rápida. Algoritmos gulosos são excelentes para este tipo de cobertura!

Escalonamento de Instruções

A ordem de execução afeta drasticamente o desempenho devido a processadores *Superescalares* e *Pipelining*.

- **Hazards:** Se a instrução 2 precisa do resultado da 1 (que ainda está no pipeline), a CPU congela e cria uma bolha (*Stall*).

Escalonamento de Instruções

A ordem de execução afeta drasticamente o desempenho devido a processadores *Superescalares* e *Pipelining*.

- **Hazards:** Se a instrução 2 precisa do resultado da 1 (que ainda está no pipeline), a CPU congela e cria uma bolha (*Stall*).
- **List Scheduling:**
  1. Desenha um Grafo de Dependência.
  2. Identifica o **Caminho Crítico**.
  3. Escolhe heurísticas para priorizar instruções independentes e preencher as bolhas do pipeline!

Exemplo: Escalonamento para evitar Bolhas (Hazards)

Problema: Ler um dado da memória e usá-lo imediatamente trava o pipeline (*Load-Use Hazard*).

Ordem Original (Ruim):

```
1:  LOAD r1, [a]
2:  ADD r2, r1, 1 ← STALL!
3:  LOAD r3, [b]
4:  ADD r4, r3, 1 ← STALL!
```

O Grafo de Dependência:

A instrução 2 depende da 1.
A instrução 4 depende da 3.

Mas as cadeias (1,2) e (3,4) são **totalmente independentes** entre si!

Exemplo: Escalonamento para evitar Bolhas (Hazards)

Problema: Ler um dado da memória e usá-lo imediatamente trava o pipeline (*Load-Use Hazard*).

Ordem Original (Ruim):

```
1:  LOAD r1, [a]
2:  ADD r2, r1, 1 ← STALL!
3:  LOAD r3, [b]
4:  ADD r4, r3, 1 ← STALL!
```

Ação do List Scheduling:

O algoritmo olha para as instruções prontas para execução (dependências resolvidas).

Após emitir a instrução 1, a 2 não está pronta fisicamente (o dado não chegou da RAM). Então ele busca e emite a instrução 3!

Exemplo: Escalonamento para evitar Bolhas (Hazards)

Problema: Ler um dado da memória e usá-lo imediatamente trava o pipeline (*Load-Use Hazard*).

Ordem Original (Ruim):

```
1:  LOAD r1, [a]
2:  ADD r2, r1, 1 ← STALL!
3:  LOAD r3, [b]
4:  ADD r4, r3, 1 ← STALL!
```

Ordem Escalonada (Ótima):

```
1:  LOAD r1, [a]
3:  LOAD r3, [b] ← Preenche a bolha
2:  ADD r2, r1, 1 ← r1 já chegou!
4:  ADD r4, r3, 1 ← r3 já chegou!
```

O código agora executa sem paralisações, aproveitando o tempo que o hardware estaria ocioso.

Alocação de Registradores (Tempo de Vida)

Devemos mapear infinitos registradores virtuais em finitos (ex: 16) registradores físicos. Para compartilhar a mesma "casa", as variáveis não podem coexistir no tempo.

Cálculo Matemático de Liveness (De Baixo para Cima)

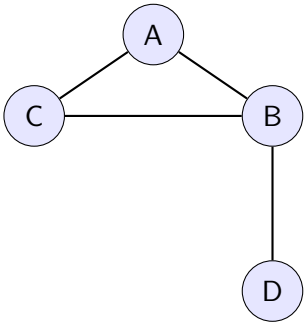
Para cada bloco B:

$$OUT(B) = \bigcup_{S \in Succ(B)} IN(S)$$

$$IN(B) = USE(B) \cup (OUT(B) - DEF(B))$$

Coloração de Grafos: Passo a Passo

Objetivo: Alocar registradores físicos limitados baseando-se em interferência temporal.



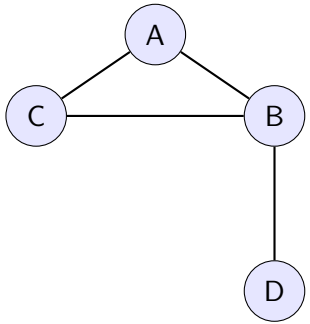
Grafo de Interferência
Aresta indica: "Estão ativas juntas".

O Hardware:
Temos apenas **3 Cores** (registradores físicos $K = 3$).

O nó B tem Grau 3, mas $K = 3$. Pela heurística clássica de **Chaitin**, B teria que sofrer *Spill* imediatamente para a RAM!

Coloração de Grafos: Passo a Passo

Objetivo: Alocar registradores físicos limitados baseando-se em interferência temporal.



Grafo de Interferência

Aresta indica: "Estão ativas juntas".

Heurística Otimista de Briggs:

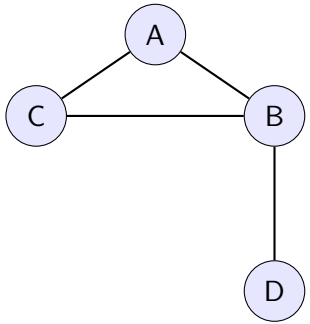
Não declare *Spill* ainda! Vamos empilhar os nós com Grau $< K$ e ver o que sobra.

- Removemos D (Grau 1).
- Removemos C (Grau 2).
- Removemos A (Grau 1 restante).

Agora B está sozinho e o empilhamos também!

Coloração de Grafos: Passo a Passo

Objetivo: Alocar registradores físicos limitados baseando-se em interferência temporal.



Grafo de Interferência
Aresta indica: "Estão ativas juntas".

Desempilhando e Colorindo:
Ao desempilhar, B volta para o grafo. Percebemos que **A** e **D** não têm aresta entre si! Eles podem compartilhar a mesma cor.

- **A** = Vermelho (R_1)
- **C** = Verde (R_2)
- **B** = Azul (R_3)
- **D** = Vermelho (R_1)

O "milagre" aconteceu: todos foram alocados e nenhum foi para a RAM!

Referencias

- Aho, et al. *Compilers: Principles, Techniques, and Tools*. Cap. 8.
- Chaitin, G. J. *Register allocation & spilling via graph coloring.*
- Briggs, P. *Improvements to Graph Coloring Register Allocation.*

Duvidas?

Obrigado!

Email: alessio@cefetmg.br