# Compiladores

## Aula 19: Representação SSA (Static Single Assignment)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Maio de 2026

# Sumário

1. Objetivos
2. Motivação: O Problema da Ambiguidade
3. O Princípio SSA
4. A Função $\phi$ (Phi)
5. Conversão para SSA
6. Vantagens para Otimização
7. Exercício Guiado
8. Perguntas de Reflexão
9. Conclusão e Próximos Passos
10. Referências

# Objetivos da Aula

- Compreender o princípio da **Static Single Assignment (SSA)**.
- Entender por que SSA simplifica **análises e otimizações**.
- Dominar o conceito da **função** $\phi$ e onde ela é inserida.
- Conhecer o fluxo de conversão: IR convencional $\rightarrow$ SSA.
- Relacionar SSA com compiladores reais (GCC, LLVM).

# O Problema com IR Convencional

Na IR convencional (3AC), uma variável pode ser atribuída **múltiplas vezes**:

**IR convencional**

```
x = 1
x = 2
y = x // qual valor de x?
```

# O Problema com IR Convencional

Na IR convencional (3AC), uma variável pode ser atribuída **múltiplas vezes**:

**IR convencional**

```
x = 1
x = 2
y = x // qual valor de x?
```

- Qual definição de `x` chega até `y = x`?

# O Problema com IR Convencional

Na IR convencional (3AC), uma variável pode ser atribuída **múltiplas vezes**:

**IR convencional**

```
x = 1
x = 2
y = x // qual valor de x?
```

- Qual definição de `x` chega até `y = x`?
- Precisamos de **análise de fluxo de dados** (reaching definitions) para responder.

# O Problema com IR Convencional

Na IR convencional (3AC), uma variável pode ser atribuída **múltiplas vezes**:

**IR convencional**

```
x = 1
x = 2
y = x // qual valor de x?
```

- Qual definição de `x` chega até `y = x`?
- Precisamos de **análise de fluxo de dados** (reaching definitions) para responder.
- Essa análise é cara: ponto-fixo iterativo sobre o CFG.

# O Problema com IR Convencional

Na IR convencional (3AC), uma variável pode ser atribuída **múltiplas vezes**:

**IR convencional**

```
x = 1
x = 2
y = x // qual valor de x?
```

- Qual definição de `x` chega até `y = x`?
- Precisamos de **análise de fluxo de dados** (reaching definitions) para responder.
- Essa análise é cara: ponto-fixo iterativo sobre o CFG.

**Insight**

E se cada variável fosse atribuída **exatamente uma vez**? A resposta seria trivial!

# Quem Usa SSA?

| Compilador/VM | Uso de SSA |
| --- | --- |

# Quem Usa SSA?

| Compilador/VM | Uso de SSA |
| --- | --- |
| GCC (4.0+) | GIMPLE em SSA (middle-end) |

# Quem Usa SSA?

| **Compilador/VM** | **Uso de SSA** |
|---|---|
| GCC (4.0+) | GIMPLE em SSA (middle-end) |
| LLVM | LLVM IR é **nativamente** SSA |

# Quem Usa SSA?

| Compilador/VM | Uso de SSA |
|---|---|
| GCC (4.0+) | GIMPLE em SSA (middle-end) |
| LLVM | LLVM IR é **nativamente** SSA |
| HotSpot JVM | C2 (Sea of Nodes, SSA) |

# Quem Usa SSA?

| **Compilador/VM** | **Uso de SSA** |
|---|---|
| GCC (4.0+) | GIMPLE em SSA (middle-end) |
| LLVM | LLVM IR é **nativamente** SSA |
| HotSpot JVM | C2 (Sea of Nodes, SSA) |
| V8 (JavaScript) | Turbofan (SSA) |

# Quem Usa SSA?

| Compilador/VM | Uso de SSA |
|---|---|
| GCC (4.0+) | GIMPLE em SSA (middle-end) |
| LLVM | LLVM IR é **nativamente** SSA |
| HotSpot JVM | C2 (Sea of Nodes, SSA) |
| V8 (JavaScript) | Turbofan (SSA) |
| Rust (rustc) | Via LLVM IR (SSA) |

# Quem Usa SSA?

| Compilador/VM | Uso de SSA |
|---|---|
| GCC (4.0+) | GIMPLE em SSA (middle-end) |
| LLVM | LLVM IR é **nativamente** SSA |
| HotSpot JVM | C2 (Sea of Nodes, SSA) |
| V8 (JavaScript) | Turbofan (SSA) |
| Rust (rustc) | Via LLVM IR (SSA) |
| Swift | Via LLVM IR + SIL (SSA) |

## Quem Usa SSA?

| Compilador/VM | Uso de SSA |
|---|---|
| GCC (4.0+) | GIMPLE em SSA (middle-end) |
| LLVM | LLVM IR é **nativamente** SSA |
| HotSpot JVM | C2 (Sea of Nodes, SSA) |
| V8 (JavaScript) | Turbofan (SSA) |
| Rust (rustc) | Via LLVM IR (SSA) |
| Swift | Via LLVM IR + SIL (SSA) |

**SSA é o estado da arte.** Praticamente todo compilador moderno de alto desempenho opera sobre SSA.

# Static Single Assignment: A Regra

**"Cada variável é definida exatamente uma vez no texto do programa."**

Para isso, **versionamos** as variáveis:

**IR convencional**

```
x = 1
x = 2
y = x
```

$\Rightarrow$

**IR em SSA**

$x_1 = 1$
$x_2 = 2$
$y_1 = x_2$

## "Cada variável é definida exatamente uma vez no texto do programa."

Para isso, **versionamos** as variáveis:

**IR convencional**

```
x = 1
x = 2
y = x
```

⇒

**IR em SSA**

$x_1$ = 1
$x_2$ = 2
$y_1$ = $x_2$

**Vantagem imediata**

Não existe ambiguidade: $y_1$ usa $x_2$. Cada uso aponta para **exatamente uma definição**. Análise de fluxo de dados se torna trivial.

# Exemplo Mais Complexo

## IR convencional

```
a = 5
b = a + 1
a = b * 2
c = a + b
```

# Exemplo Mais Complexo

## IR convencional

```
a = 5
b = a + 1
a = b * 2
c = a + b
```

## IR em SSA

```
a1 = 5
b1 = a1 + 1
a2 = b1 * 2
c1 = a2 + b1
```

# Exemplo Mais Complexo

## IR convencional

```
a = 5
b = a + 1
a = b * 2
c = a + b
```

## IR em SSA

```
a1 = 5
b1 = a1 + 1
a2 = b1 * 2
c1 = a2 + b1
```

**Observações:**

- a1 e a2 são variáveis **independentes**.
- Cada uso referencia a versão **exata** da definição.
- Se `a1` nunca é usado após `b1 = a1 + 1`, é fácil detectar que `a1 = 5` pode ser propagado.

# O Problema: Junções de Controle

SSA funciona perfeitamente em código **linear**. Mas e quando há **ramificação**?

**Código original**

```
if (cond)
  x = 1;
else
  x = 2;
print(x); // qual x?
```

# O Problema: Junções de Controle

SSA funciona perfeitamente em código **linear**. Mas e quando há **ramificação**?

**Código original**

```
if (cond)
  x = 1;
else
  x = 2;
print(x); // qual x?
```

- No bloco then: $x_1 = 1$
- No bloco else: $x_2 = 2$

# O Problema: Junções de Controle

SSA funciona perfeitamente em código **linear**. Mas e quando há **ramificação**?

**Código original**

```
if (cond)
  x = 1;
else
  x = 2;
print(x); // qual x?
```

- No bloco then: $\texttt{x}_1$ `= 1`
- No bloco else: $\texttt{x}_2$ `= 2`
- No bloco de junção: `print(???)`

# O Problema: Junções de Controle

SSA funciona perfeitamente em código **linear**. Mas e quando há **ramificação**?

**Código original**

```
if (cond)
  x = 1;
else
  x = 2;
print(x); // qual x?
```

- No bloco then: $\texttt{x}_1$ `= 1`
- No bloco else: $\texttt{x}_2$ `= 2`
- No bloco de junção: `print(???)`
- Precisamos de algo que "escolha" entre $\texttt{x}_1$ e $\texttt{x}_2$.

SSA funciona perfeitamente em código **linear**. Mas e quando há **ramificação**?

**Código original**

```
if (cond)
  x = 1;
else
  x = 2;
print(x); // qual x?
```

- No bloco then: $x_1$ = 1
- No bloco else: $x_2$ = 2
- No bloco de junção: `print(???)`
- Precisamos de algo que "escolha" entre $x_1$ e $x_2$.

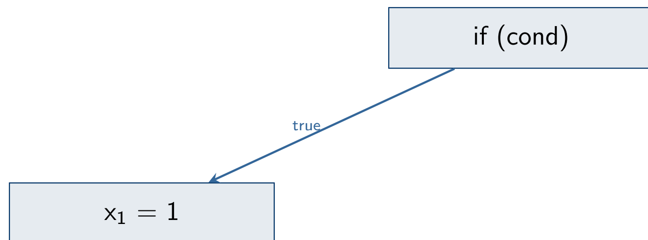
**Solução: a função $\phi$**

Uma instrução especial que "seleciona" o valor correto conforme o **predecessor** de onde o fluxo veio.

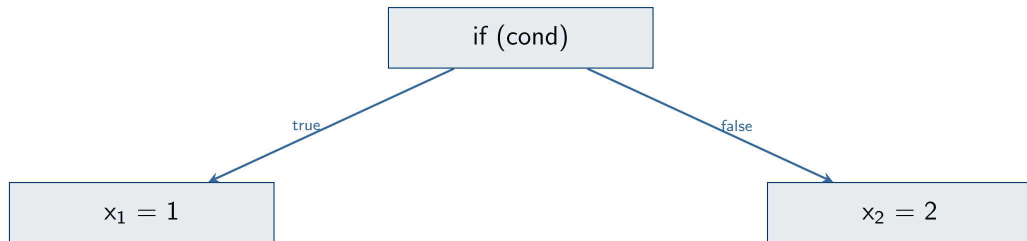
# A Função $\phi$ em Ação

if (cond)

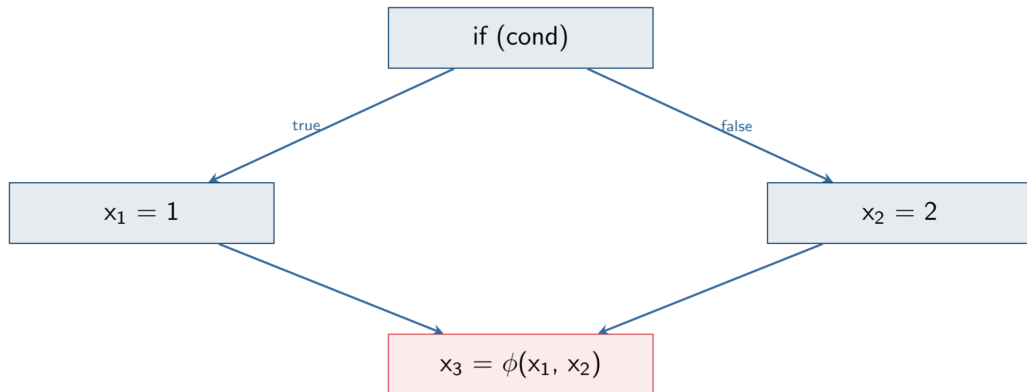
# A Função $\phi$ em Ação



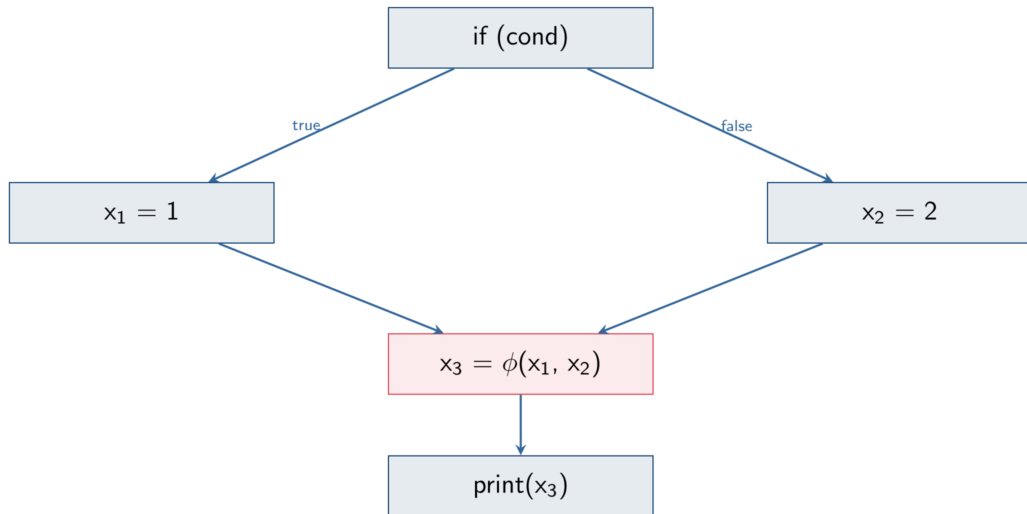
# A Função $\phi$ em Ação



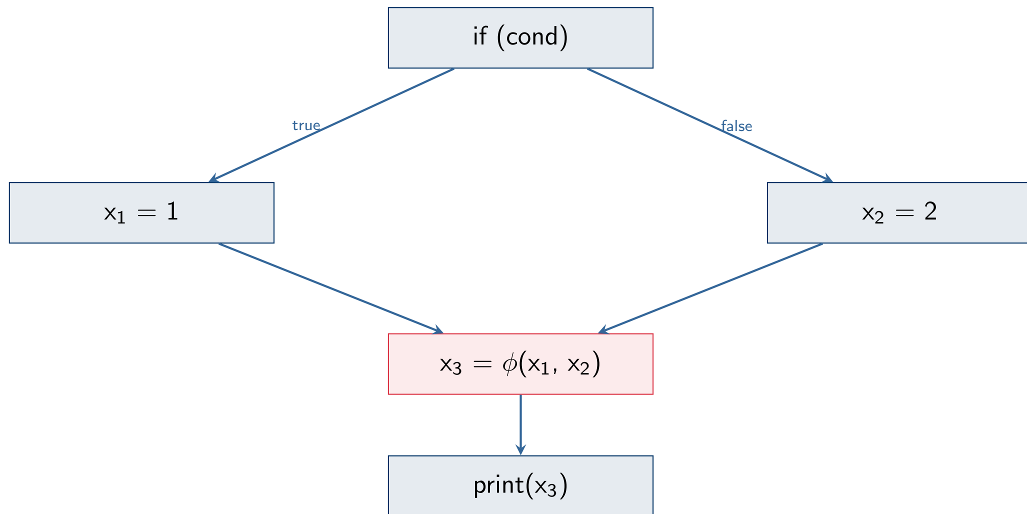
# A Função $\phi$ em Ação



# A Função $\phi$ em Ação



# A Função $\phi$ em Ação




A $\phi$ diz: "Se vim do then, $x_3 = x_1$. Se vim do else, $x_3 = x_2$."

## Propriedades da Função $\phi$

- $\phi$ tem **um argumento por predecessor** do bloco.

# Propriedades da Função $\phi$

- $\phi$ tem **um argumento por predecessor** do bloco.
- $\phi$ é uma instrução **fictícia** — não gera código diretamente.

# Propriedades da Função $\phi$

- $\phi$ tem **um argumento por predecessor** do bloco.
- $\phi$ é uma instrução **fictícia** — não gera código diretamente.
- Na geração de código real, vira **movimentação** ou é resolvida pelo **alocador de registradores**.

## Propriedades da Função $\phi$

- $\phi$ tem **um argumento por predecessor** do bloco.
- $\phi$ é uma instrução **fictícia** — não gera código diretamente.
- Na geração de código real, vira **movimentação** ou é resolvida pelo **alocador de registradores**.
- $\phi$ aparece **apenas no início** de blocos com $\geq 2$ predecessores.

# Propriedades da Função $\phi$

- $\phi$ tem **um argumento por predecessor** do bloco.
- $\phi$ é uma instrução **fictícia** — não gera código diretamente.
- Na geração de código real, vira **movimentação** ou é resolvida pelo **alocador de registradores**.
- $\phi$ aparece **apenas no início** de blocos com $\geq 2$ predecessores.
- Pode haver **múltiplas** $\phi$ no mesmo bloco (uma por variável que precisa de merge).

# Propriedades da Função $\phi$

- $\phi$ tem **um argumento por predecessor** do bloco.
- $\phi$ é uma instrução **fictícia** — não gera código diretamente.
- Na geração de código real, vira **movimentação** ou é resolvida pelo **alocador de registradores**.
- $\phi$ aparece **apenas no início** de blocos com $\geq 2$ predecessores.
- Pode haver **múltiplas** $\phi$ no mesmo bloco (uma por variável que precisa de merge).

**Na prática**

$\texttt{x}_3 = \phi(\texttt{x}_1\texttt{:B1},\ \texttt{x}_2\texttt{:B2})$ — cada argumento é anotado com o bloco predecessor de onde vem.

## $\phi$ em Loops

Loops também geram $\phi$, pois o cabeçalho do loop tem dois predecessores (entrada + back-edge):

**Código original**

```
i = 0
while (i < 10)
  i = i + 1
```

```
     i1 = 0
     goto L1
L1:  i2 = phi(i1:entry, i3:L2)
     if i2 < 10 goto L2
     goto L3
L2:  i3 = i2 + 1
     goto L1
L3:  ...
```

# $\phi$ em Loops

Loops também geram $\phi$, pois o cabeçalho do loop tem dois predecessores (entrada + back-edge):

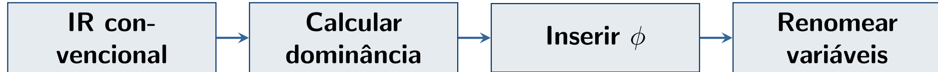
**Código original**

```
i = 0
while (i < 10)
  i = i + 1
```

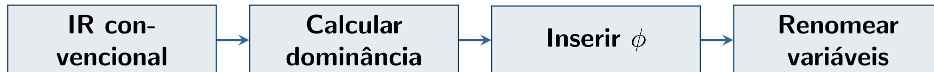
```
      i1 = 0
      goto L1
L1:   i2 = phi(i1:entry, i3:L2)
      if i2 < 10 goto L2
      goto L3
L2:   i3 = i2 + 1
      goto L1
L3:   ...
```

$\mathtt{i}_2$ recebe $\mathtt{i}_1$ na primeira iteração e $\mathtt{i}_3$ nas seguintes.

# Fluxo de Conversão: IR $\rightarrow$ SSA

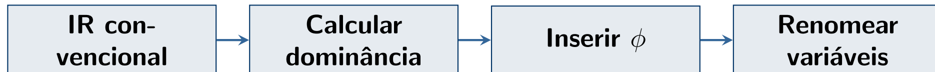


# Fluxo de Conversão: IR $\rightarrow$ SSA



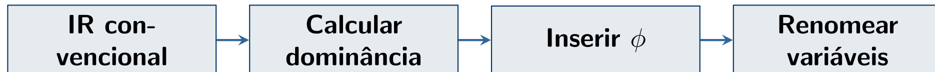

1. **Dominância:** bloco $B$ domina $B'$ se todo caminho da entrada até $B'$ passa por $B$. Define **fronteiras de dominância** (pontos de merge).

## Fluxo de Conversão: IR $\rightarrow$ SSA



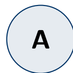

1. **Dominância:** bloco $B$ domina $B'$ se todo caminho da entrada até $B'$ passa por $B$. Define **fronteiras de dominância** (pontos de merge).
2. **Inserir $\phi$:** para cada variável com definições em $\geq 2$ caminhos, inserir $\phi$ nas fronteiras de dominância.

# Fluxo de Conversão: IR $\rightarrow$ SSA

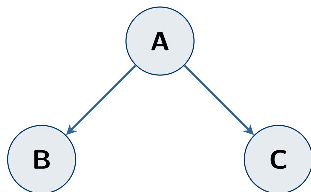



1. **Dominância:** bloco $B$ domina $B'$ se todo caminho da entrada até $B'$ passa por $B$. Define **fronteiras de dominância** (pontos de merge).
2. **Inserir $\phi$:** para cada variável com definições em $\geq 2$ caminhos, inserir $\phi$ nas fronteiras de dominância.
3. **Renomear:** cada definição recebe um índice novo ($x \rightarrow x_1, x_2, \ldots$); usos apontam para a versão dominante.

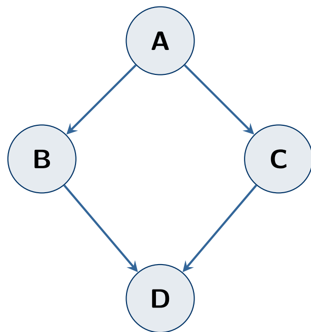
# Dominância: Conceito Visual



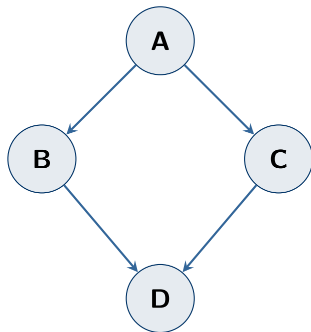
# Dominância: Conceito Visual



# Dominância: Conceito Visual

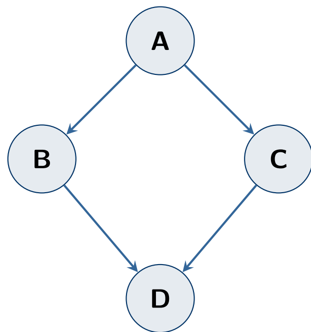


# Dominância: Conceito Visual



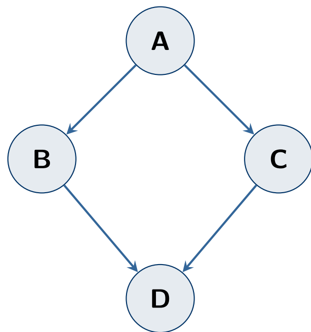

- A domina {A, B, C, D} (tudo passa por A).

# Dominância: Conceito Visual




- A domina {A, B, C, D} (tudo passa por A).
- B **não** domina D (existe caminho A→C→D que não passa por B).

# Dominância: Conceito Visual




- A domina {A, B, C, D} (tudo passa por A).
- B **não** domina D (existe caminho A→C→D que não passa por B).
- D é **fronteira de dominância** de B e C ⇒ $\phi$ são inseridas em D.

# Exemplo Prático: Passo a Passo

## Código Original

```
x = 0
y = x + 1
x = 2
z = x + y
x = z
```

## Exemplo Prático: Passo a Passo

**Código Original**

```
x = 0
y = x + 1
x = 2
z = x + y
x = z
```

**Passo 1**

$x_1$ = 0
$y_1$ = $x_1$ + 1
x = 2
z = x + y
x = z

- **Passo 1:** Renomear as primeiras definições de `x` e `y`.

# Exemplo Prático: Passo a Passo

**Código Original**

```
x = 0
y = x + 1
x = 2
z = x + y
x = z
```

**Passo 1**

$x_1$ = 0
$y_1$ = $x_1$ + 1
x = 2
z = x + y
x = z

**Passo 2**

$x_1$ = 0
$y_1$ = $x_1$ + 1
$x_2$ = 2
z = $x_2$ + $y_1$
x = z

- **Passo 1:** Renomear as primeiras definições de `x` e `y`.
- **Passo 2:** Atualizar usos e renomear nova definição de `x`.

# Exemplo Prático: Passo a Passo

**Código Original**

```
x = 0
y = x + 1
x = 2
z = x + y
x = z
```

- **Passo 1:** Renomear as primeiras definições de x e y.
- **Passo 2:** Atualizar usos e renomear nova definição de x.
- **Passo 3:** Atualizar usos restantes.

**Passo 1**

$x_1 = 0$
$y_1 = x_1 + 1$
$x = 2$
$z = x + y$
$x = z$

**Passo 2**

$x_1 = 0$
$y_1 = x_1 + 1$
$x_2 = 2$
$z = x_2 + y_1$
$x = z$

**Passo 3**

$x_1 = 0$
$y_1 = x_1 + 1$
$x_2 = 2$
$z_1 = x_2 + y_1$
$x_3 = z_1$

**Propagação de constantes:**

# Otimizações Simplificadas por SSA

**Propagação de constantes:**

- Se a única def que alcança um uso é constante, substituir.
- Em SSA: cada uso tem **exatamente uma** definição $\Rightarrow$ trivial.

# Otimizações Simplificadas por SSA

**Propagação de constantes:**

- Se a única def que alcança um uso é constante, substituir.
- Em SSA: cada uso tem **exatamente uma** definição $\Rightarrow$ trivial.

**Eliminação de código morto:**

- Def cujo valor nunca é usado $\Rightarrow$ remover.
- Em SSA: basta verificar se a def tem **zero usos**.

# Otimizações Simplificadas por SSA

**Propagação de constantes:**

- Se a única def que alcança um uso é constante, substituir.
- Em SSA: cada uso tem **exatamente uma** definição ⇒ trivial.

**Eliminação de código morto:**

- Def cujo valor nunca é usado ⇒ remover.
- Em SSA: basta verificar se a def tem **zero usos**.

**Propagação de cópias:**

- $y_1 = x_2$ ⇒ substituir todos os usos de $y_1$ por $x_2$.

# Otimizações Simplificadas por SSA

**Propagação de constantes:**

- Se a única def que alcança um uso é constante, substituir.
- Em SSA: cada uso tem **exatamente uma** definição ⇒ trivial.

**Eliminação de código morto:**

- Def cujo valor nunca é usado ⇒ remover.
- Em SSA: basta verificar se a def tem **zero usos**.

**Propagação de cópias:**

- $y_1 = x_2$ ⇒ substituir todos os usos de $y_1$ por $x_2$.

**Alocação de registradores:**

- Intervalos de vida mais claros.
- LLVM aloca registradores diretamente sobre nomes SSA.

# Exemplo Prático: Otimizações com SSA

## IR em SSA

```
x1 = 10
y1 = x1
z1 = y1 + 5
```

# Exemplo Prático: Otimizações com SSA

## IR em SSA

```
x1 = 10
y1 = x1
z1 = y1 + 5
```

## 1. Propagação de Cópias

```
x1 = 10
y1 = x1
z1 = x1 + 5
```

Substituímos `y1` por `x1` em `z1`.

# Exemplo Prático: Otimizações com SSA

## IR em SSA

```
x1 = 10
y1 = x1
z1 = y1 + 5
```

## 1. Propagação de Cópias

```
x1 = 10
y1 = x1
z1 = x1 + 5
```

Substituímos `y1` por `x1` em `z1`.

## 2. Propagação de Constantes

```
x1 = 10
y1 = x1
z1 = 10 + 5
```

Como `x1` é constante, substituímos na expressão.

# Exemplo Prático: Otimizações com SSA

## IR em SSA

```
x1 = 10
y1 = x1
z1 = y1 + 5
```

## 1. Propagação de Cópias

```
x1 = 10
y1 = x1
z1 = x1 + 5
```

Substituímos `y1` por `x1` em `z1`.

## 2. Propagação de Constantes

```
x1 = 10
y1 = x1
z1 = 10 + 5
```

Como `x1` é constante, substituímos na expressão.

## 3. Constant Folding & Dead Code

```
z1 = 15
```

Avaliamos $10 + 5$. Se `x1` e `y1` não tiverem mais usos, são eliminados (Dead Code).

# Exercício: Conversão para SSA

Converta o seguinte código 3AC para SSA:

**IR convencional**

```
    a = 5
    b = a + 1
    if b > 3 goto L1
    a = 10
    goto L2
L1: a = 20
L2: c = a + b
```

**Perguntas:**

1. Quantas versões de `a` existem?
2. Onde inserir $\phi$ para `a`?
3. Precisa de $\phi$ para `b`? Por quê?
4. Escreva o código completo em SSA.

# Solução do Exercício

```
        a1 = 5
        b1 = a1 + 1
        if b1 > 3 goto L1
        a2 = 10
        goto L2
L1:     a3 = 20
L2:     a4 = phi(a2, a3)
        c1 = a4 + b1
```

# Solução do Exercício

```
      a1 = 5
      b1 = a1 + 1
      if b1 > 3 goto L1
      a2 = 10
      goto L2
L1:   a3 = 20
L2:   a4 = phi(a2, a3)
      c1 = a4 + b1
```

- **3 versões** de a: $a_1$, $a_2$, $a_3$ ($+$ $a_4$ da $\phi$).

# Solução do Exercício

```
      a1 = 5
      b1 = a1 + 1
      if b1 > 3 goto L1
      a2 = 10
      goto L2
L1:   a3 = 20
L2:   a4 = phi(a2, a3)
      c1 = a4 + b1
```

- **3 versões** de a: $a_1$, $a_2$, $a_3$ ($+$ $a_4$ da $\phi$).
- $\phi$ em L2 porque a é definida em ambos os caminhos.

# Solução do Exercício

```
      a1 = 5
      b1 = a1 + 1
      if b1 > 3 goto L1
      a2 = 10
      goto L2
L1:   a3 = 20
L2:   a4 = phi(a2, a3)
      c1 = a4 + b1
```

- **3 versões** de a: $a_1$, $a_2$, $a_3$ ($+$ $a_4$ da $\phi$).
- $\phi$ em L2 porque a é definida em ambos os caminhos.
- **Não** precisa de $\phi$ para b: apenas uma definição ($b_1$) alcança todos os usos.

## Para Pensar...

1. Se SSA requer $\phi$ em junções, e loops criam junções, um programa com muitos loops aninhados terá muitas $\phi$. Isso é um problema na prática?

# Para Pensar...

1. Se SSA requer $\phi$ em junções, e loops criam junções, um programa com muitos loops aninhados terá muitas $\phi$. Isso é um problema na prática?
2. Como a "saída de SSA" (desconversão) funciona antes da geração de código final? O que acontece com as $\phi$?

# Para Pensar...

1. Se SSA requer $\phi$ em junções, e loops criam junções, um programa com muitos loops aninhados terá muitas $\phi$. Isso é um problema na prática?
2. Como a "saída de SSA" (desconversão) funciona antes da geração de código final? O que acontece com as $\phi$?
3. A LLVM IR é **nativamente** SSA. Isso significa que o frontend (Clang) precisa gerar código já em SSA?

# Resumo da Aula

- **SSA:** cada variável é definida exatamente uma vez; versões $(x_1, x_2, \ldots)$ eliminam ambiguidade.
- $\phi$**:** nos merges do CFG, seleciona o valor correto conforme o predecessor.
- **Conversão:** dominância $\rightarrow$ inserir $\phi$ $\rightarrow$ renomear variáveis.
- SSA simplifica **propagação de constantes**, **eliminação de código morto**, **propagação de cópias** e **alocação de registradores**.

**Próxima Aula:** Introdução ao LLVM IR — a IR em SSA mais usada no mundo, anatomia de um módulo LLVM, e ferramentas `clang`/`opt`/`llc`.

# Referências


- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição. Cap. 9 (Otimizações).
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição. Cap. 9 (Data-Flow Analysis), Cap. 10 (SSA).
- **CYTRON, R. et al.** “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.” *ACM TOPLAS*, 1991.
- **Documentação LLVM:** `https://llvm.org/docs/LangRef.html`

# Dúvidas?

**Obrigado!**

**Email:** alessio@cefetmg.br
**Web:** alessiojr.com