ÉCOLE
POLYTECHNIQUE
MONTRÉAL

LOG3210
Cours 10

Génération de code machine

Résumé chapitres 8.1 à 8.3

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le générateur de code transforme le code intermédiaire (IR) en code machine.
- L'architecture de la machine (RISC, CISC, stack-based) dicte le design du générateur de code.
- Les instructions machine ont un coût de performance variable et le générateur de code doit tenter de minimiser ce coût.
- Les registres sont disponibles en quantité limitée et le générateur de code doit les allouer de manière à maximiser les performances.
- Les tailles de variables, enregistrements, etc. que nous avons calculées lors de la génération de code intermédiaire nous permettent de générer les adresses mémoire des variables, de maintenir l'adresse du pointeur de la pile, etc.

Génération de code

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Si notre but est seulement de générer du code machine sémantiquement correct, une étape supplémentaire de traduction du code intermédiaire en code machine est suffisante.

- Il est toutefois attendu qu'un compilateur génère du code machine qui soit non seulement sémantiquement correct (prérequis non négociable), mais aussi « performant ».

- Le compilateur procédera donc à plusieurs phases d'optimisation dans le but d'éliminer des instructions inutiles, de mieux allouer les registres, de réduire la taille du code généré, d'exploiter le parallélisme, etc.

- Dans ce cours, nous aborderons différentes techniques d'optimisations « locales », effectuées lors de la génération de code machine.

Blocs de base

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les optimisations locales considèrent généralement des séquences d'instructions, appelées blocs de base, de telle sorte que:
  1. L'exécution d'un bloc débute toujours à la première instruction du bloc (pas d'étiquette à l'intérieur du bloc, sauf, peut-être, à la première instruction).
  2. L'exécution du bloc se termine toujours à la dernière instruction du bloc (pas de goto ou d'instruction de terminaison à l'intérieur du bloc, sauf, peut-être, à la dernière instruction.

Partitionnement du code intermédiaire en blocs de base

ÉCOLE POLYTECHNIQUE MONTRÉAL

Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

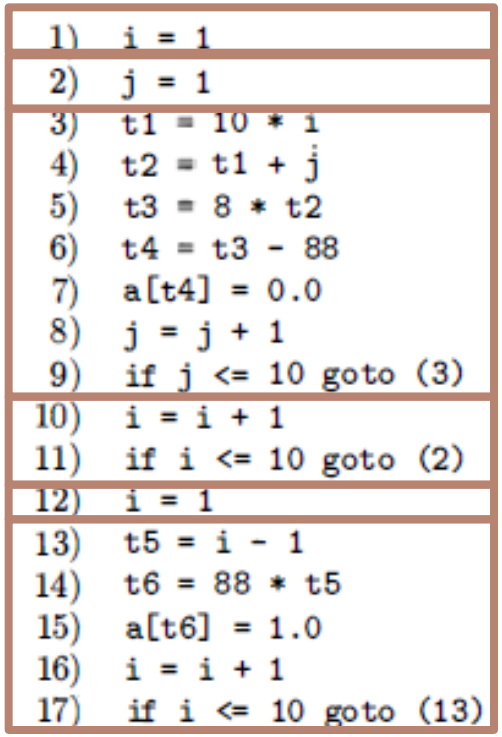
Exemple

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Exemple



```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Leaders
1, 2, 3, 10, 12, 13

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Next-Use

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Comme nous l'avons mentionné précédemment, les registres sont disponibles en quantité limitée.
- Lorsqu'un registre doit être libéré, il est préférable d'en libérer un qui contient une valeur qui ne sera plus utilisée ou à tout le moins qui ne sera pas utilisée bientôt.
- Le *next-use* d'une variable nous informe à savoir
  1. Si une variable sera encore utilisée (vie).
  2. Si oui, à quelle instruction elle sera utilisée (*next-use*).

Définition d'usage et vie d'une variable

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Afin de calculer le *next-use* d'une variable, nous devons définir deux termes: usage et vie d'une variable.

- **Usage d'une variable:** La *valeur* d'une variable *x*, définie à l'instruction *d* est **utilisée** à une instruction *j* si:
  1. Il existe un chemin d'exécution de *d* à *j* ne contenant pas de redéfinition de *x*.
  2. La variable *x* apparaît comme une *opérande* de l'instruction *j*.

- **Vie d'une variable:** Une variable *x*, est **vive** à une instruction *i* si elle peut être **utilisée** dans un chemin de *i* à la fin du programme.

Algorithme *next-use*

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Afin de déterminer la vie et l'usage d'une variable *x* à un point *i* il est nécessaire d'explorer les instructions suivant *i*, afin de trouver les futures usages de *x* par exemple.

- Parcourir les instructions en sens « avant » nous force donc à recourir à une forme de *backtracking*.

- En parcourant les instructions en sens inverse, il est possible de propager toute l'information nécessaire, sans avoir à recourir au *backtracking*.

Algorithme *next-use* (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

Instruction à 3 adresses générique

Algorithm 8.7: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

OUTPUT: At each statement i: $x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and y.
2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Algorithme *next-use* (exemple)

ÉCOLE POLYTECHNIQUE MONTRÉAL

Instructions	***Liveness***	***Next use***
Begin	b, c, d	
1) a = b + c	b, c, d	a: L3, b: L1, c: L1, d: L2
2) d = d - b	a, b, c, d	a: L3, c: L, b: L2, d: L2
3) e = a + c	a, b, c, d	a: L3, c: L3
End	a, b, c, d, e	

Algorithme *next-use* (exemple)

ÉCOLE POLYTECHNIQUE MONTRÉAL

Instructions	***Liveness***	***Next use***
Begin	`b, c, d`	
`1) a = b + c`	`b, c, d`	`a: L3, b: L1, c: L1, d: L2`
`2) d = d - b`	`a, b, c, d`	`a: L3, c: L, b: L2, d: L2`
`3) e = a + c`	`a, c`	`a: L3, c: L3`
End	–	

Algorithme *next-use* (exercice)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Calculer les *next-use* sur le programme suivant (après avoir séparer en blocs de bases).
- Les *next-use* à l'entrée du bloc suivant deviennent les *next-use* à l'entrée du bloc précédent.

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Graphes de flux de contrôle

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Une fois les blocs de base construits, le flux de contrôle entre les blocs peut être représenté par des arrêtes dirigées entre les blocs.

- Il y aura une arrête entre deux blocs *B* et *C* s'il est possible que la première instruction de *C* suive immédiatement la dernière instruction de *B*.

- Cette situation est possible lorsque:
  1. Il y a un saut (goto) de la fin de *B* au début de *C*.
  2. Les instructions du bloc *C* suivent immédiatement celles du bloc *B* dans le code à 3 adresses et *B* ne se termine pas par un saut non conditionnel.

Graphes de flux de contrôle (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Un peu de nomenclature…
- S'il existe une arrête *B* → *C*, alors *B* est le *prédécesseur* de *C* et *C* est le *successeur* de *B*.
- Afin de simplifier plusieurs analyses, on introduira souvent deux nœuds spéciaux *entry* et *exit* dans le graphe de flux de contrôle.
- *Entry* et *exit* ne représente aucune instruction. *Entry* est toujours le prédécesseur du premier bloc et exit est toujours le successeur du dernier bloc.

Boucles

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La vaste majorité des programmes passent la majeure partie de leur temps à exécuter des boucles et plusieurs optimisations visent spécifiquement ces structures.

- Dans un graphe de flux de contrôle, un ensemble de nœuds *L* forme une boucle si *L* contient un nœud e tel que:
  1. Le nœud e n'est pas le nœud *entry* du graphe de flux de contrôle.
  2. Outre e, aucun nœud en *L* n'a de prédécesseur en dehors de *L*.
  3. Pour tous nœuds *n* dans *L*, il existe un chemin non vide de *n* à e, contenant seulement des nœuds de *L*.

Optimisation des blocs de base

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Plusieurs techniques d'optimisation locale (de blocs) travaillent sur une représentation sous forme de graphe dirigé acyclique (DAG) des blocs.
- Il est possible de construire un DAG pour un bloc de base comme suit:
  1. Créer un nœud dans le DAG pour chaque *valeur initiale* des variables dans le bloc.
  2. Créer un nœud *N* pour chaque instruction *s* dans le bloc. Les enfants de *N* seront les nœuds correspondant aux définitions les plus récentes des opérandes utilisées dans *s*.
  3. Chaque nœud *N* est étiqueté avec l'opérateur de l'instruction *s*.
  4. Chaque nœud *N* est aussi étiqueté avec la liste des variables dont la dernière définition du bloc apparaît au nœud *N*.
  5. Certains nœuds seront définis comme nœuds de sortie. Ce sont les nœuds pour lesquels les variables sont vives à la *sortie* du bloc. (Nous n'avons pas encore vu comment les calculer).

Définitions récentes

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Pour construire le DAG d'un bloc, nous avons besoin de connaître les définitions les plus récentes de chaque opérandes et les dernières définitions présentes dans un bloc.

- Exemple:

```
1. a = b + c
2. b = a - d
3. c = b + c
4. d = a - d
```

a	b	c	d
a_0	b_0	c_0	d_0
1	b_0	c_0	d_0
1	2	c_0	d_0
1	2	3	d_0
1	2	3	4

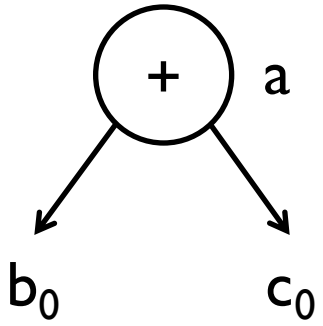
- Un simple parcours « avant » du bloc permet de calculer ces informations.

Exemple DAG d'un bloc

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
1. a = b + c
2. b = a - d
3. c = b + c
4. d = a - d
```

a	b	c	d
a_0	b_0	c_0	d_0
1	b_0	c_0	d_0
1	2	c_0	d_0
1	2	3	d_0
1	2	3	4

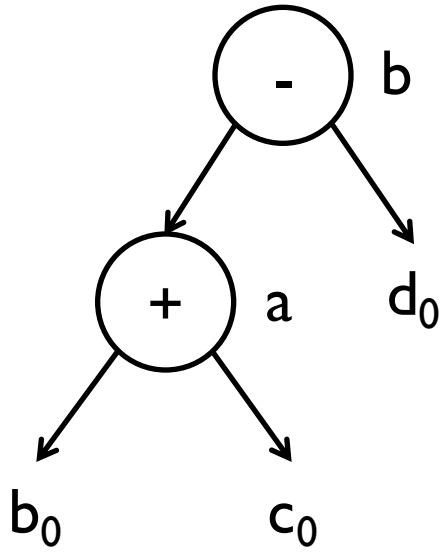


Exemple DAG d'un bloc

ÉCOLE POLYTECHNIQUE MONTRÉAL

1. a = b + c
2. **b = a - d**
3. c = b + c
4. d = a - d

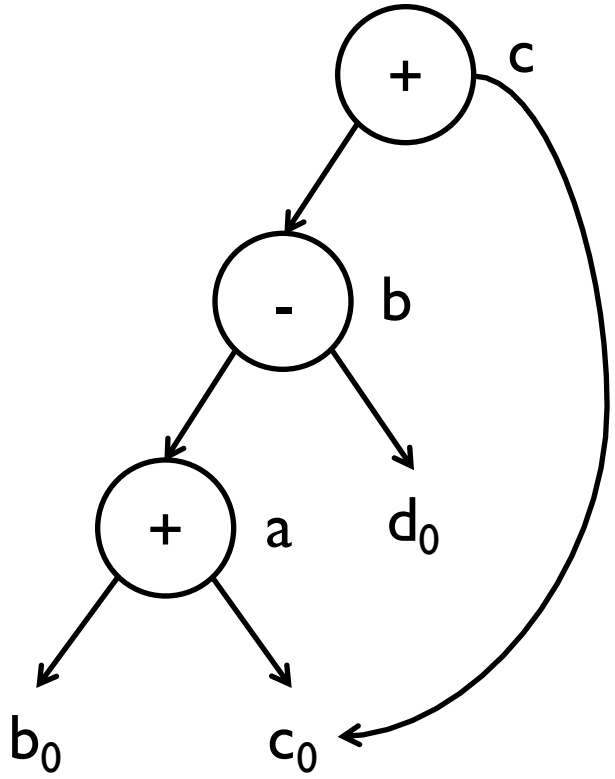
a	b	c	d
a_0	b_0	c_0	d_0
I	**b_0**	**c_0**	**d_0**
I	2	c_0	d_0
I	2	3	d_0
I	2	3	4



Exemple DAG d'un bloc

```
1. a = b + c
2. b = a - d
3. c = b + c
4. d = a - d
```

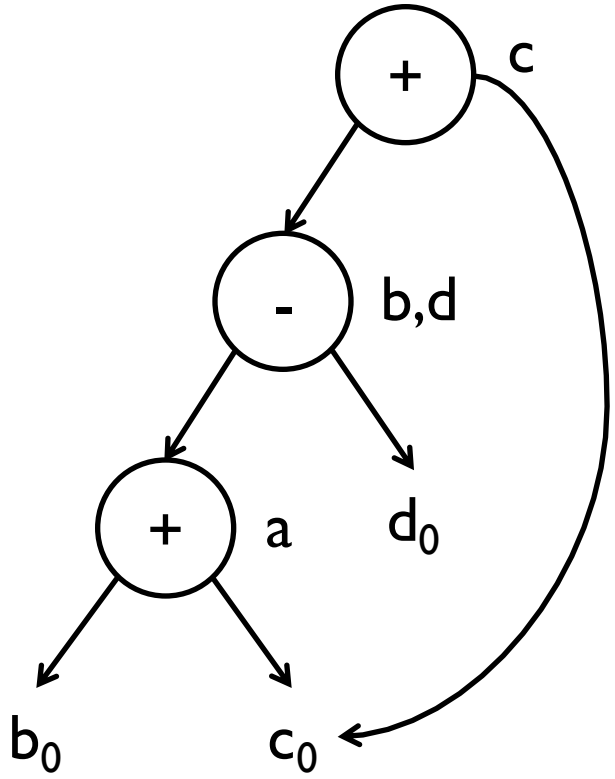
a	b	c	d
a_0	b_0	c_0	d_0
1	b_0	c_0	d_0
1	**2**	**c_0**	**d_0**
1	2	3	d_0
1	2	3	4



Exemple DAG d'un bloc

1. a = b + c
2. b = a - d
3. c = b + c
4. **d = a - d**

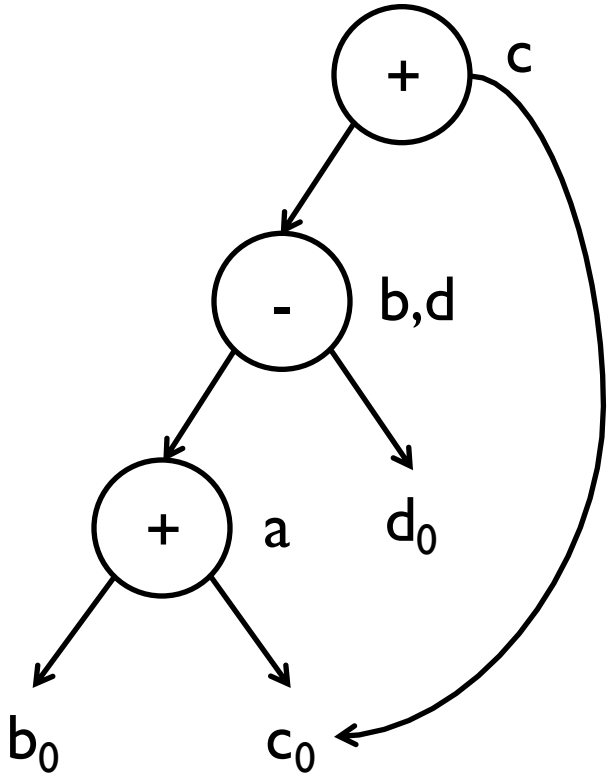
a	b	c	d
a_0	b_0	c_0	d_0
1	b_0	c_0	d_0
1	2	c_0	d_0
1	**2**	**3**	**d_0**
1	2	3	4



Élimination des sous-expressions communes

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Dans l'exemple précédent, la dernière instruction (d = a – d) n'ajoutait pas de nœud au DAG.
- Dans ce cas particulier, si *b* ou *d* ne sont pas vives à la sortie, une des deux instructions représentée par le nœud peut être éliminée.
- Si *b* et *d* sont vives à la sortie, on peut éliminer une des deux instructions et ajouter une instruction de copie à la fin.



Élimination des sous-expressions communes (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

b et d sont vives

```
a = b + c
b = a - d
c = b + c
d = b
```

b est non vive, d est vive

```
a = b + c
d = a - d
c = d + c
```

Dans le cas général, dès qu'une instruction n'introduit pas un nouveau nœud dans le DAG, il y une sous-expression commune qui peut être éliminée.

Élimination du code mort

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Étant donné le DAG d'un bloc, il est possible d'éliminer le code mort (inutile) en retirant du DAG les racines qui ne sont pas étiquetées avec une variable vive.

- Cette opération est répétée tant et aussi longtemps que des nœuds peuvent être éliminés.

Élimination du code mort (Exemple)

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
a = b + c
b = b - d
c = c + d
e = b + c
```

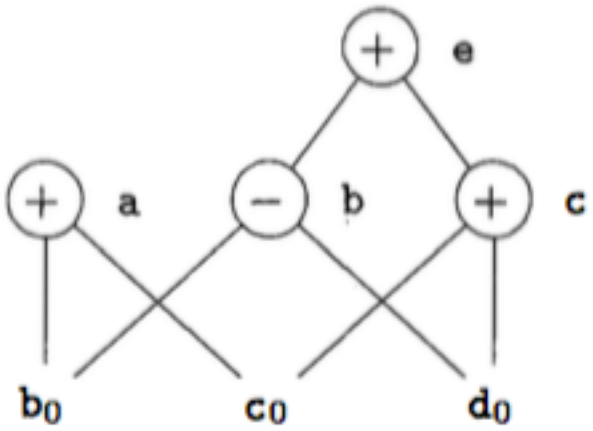


Figure 8.13: DAG for basic block in Example 8.11

Élimination du code mort (Exemple)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- a = b + c
  b = b − d
  ~~c = c + d~~
  ~~e = b + c~~

Si *c* et *e* ne sont pas vives à la fin du bloc.

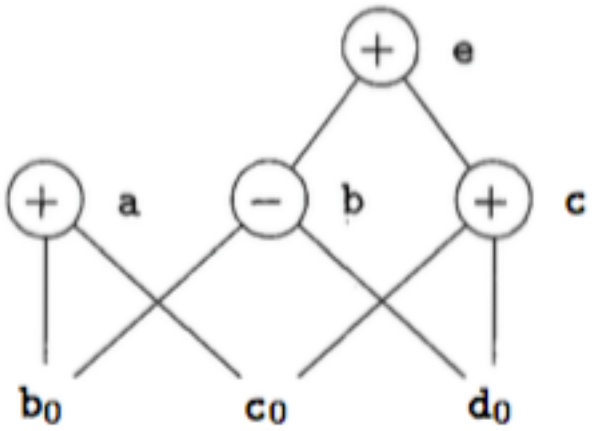


Figure 8.13: DAG for basic block in Example 8.11

Identités algébriques

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Certaines identités algébriques peuvent nous aider à éliminer des instructions.

- Identités arithmétiques:
  - $x + 0 = 0 + x = x$ $\qquad$ $x - 0 = x$
  - $x \times 1 = 1 \times x = x$ $\qquad$ $x / 1 = x$

- Réduction en force (*reduction in strength*) où certaines opérations s'exécutent plus rapidement :
  - $x^2 \quad = \quad x \times x$
  - $2 \times x \quad = \quad x + x$
  - $x / 2 \quad = \quad x \times 0.5$ (*shift*)

Identités algébriques (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

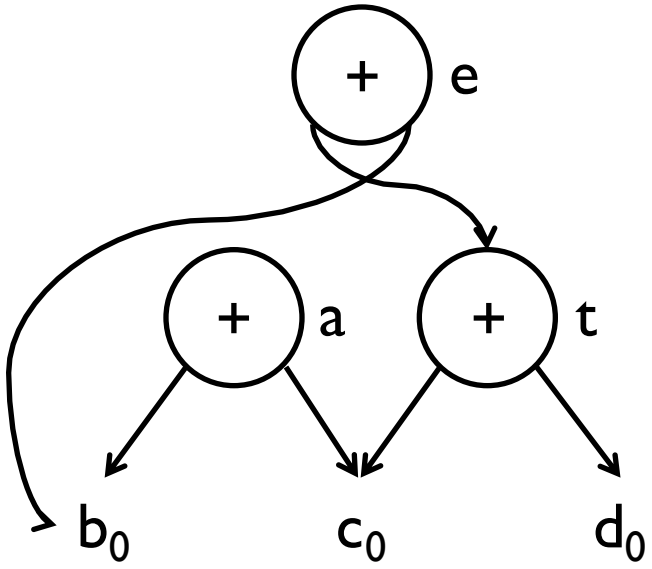
- Commutativité:
  - Par exemple, si le langage spécifie que x * y = y * x, avant de créer un nouveau nœud pour x * y, on peut vérifier s'il existe déjà un nœud pour y * x.

- Commutativité + Associativité:
  (`a = b + c; e = c + d + b;`)

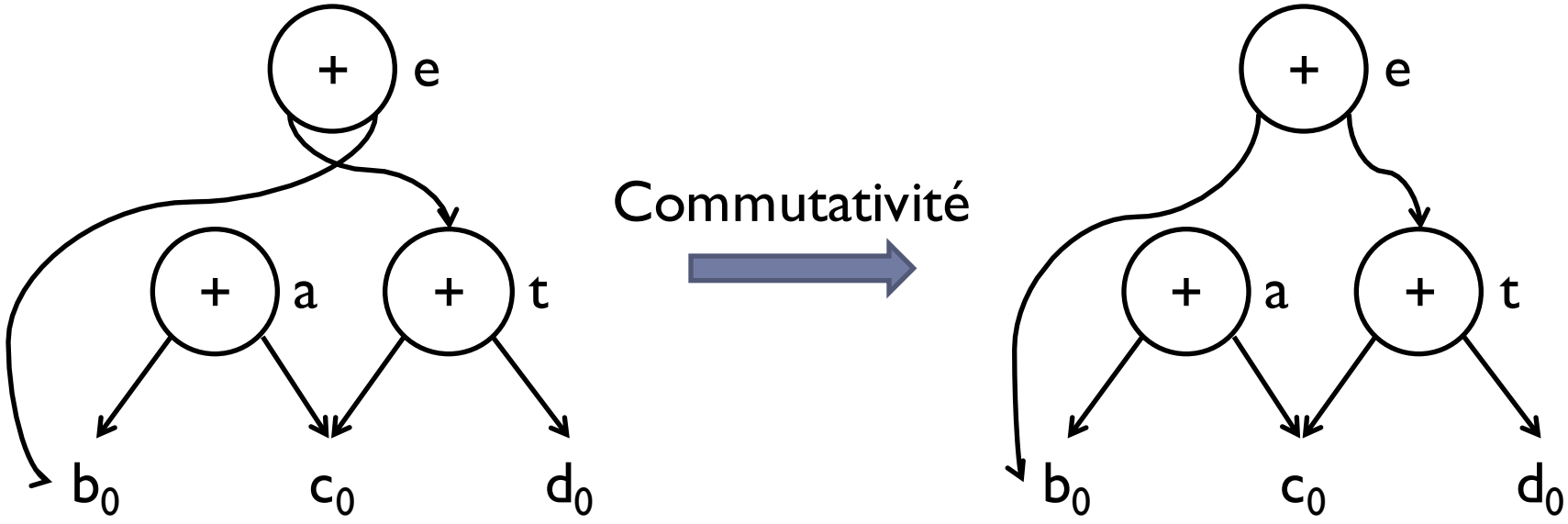
```
a = b + c          a = b + c
t = c + d    ➔     e = a + d
e = t + b
```

Commutativité et associativité

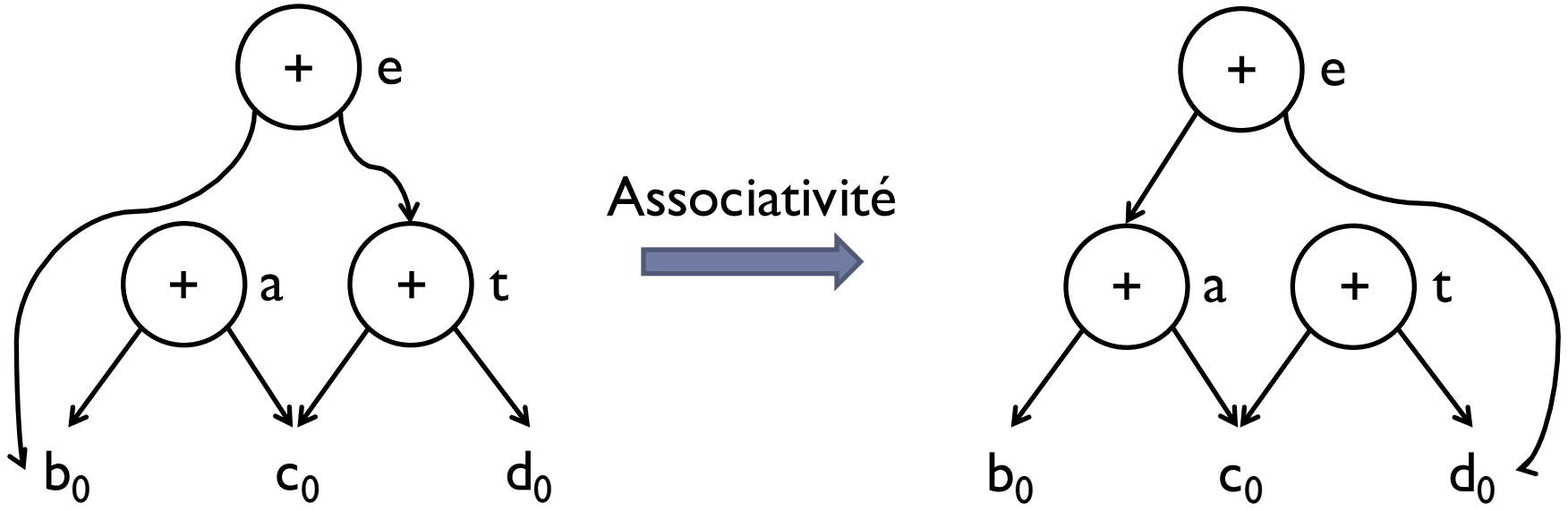
```
a = b + c
t = c + d
e = t + b
```



Commutativité et associativité (suite)



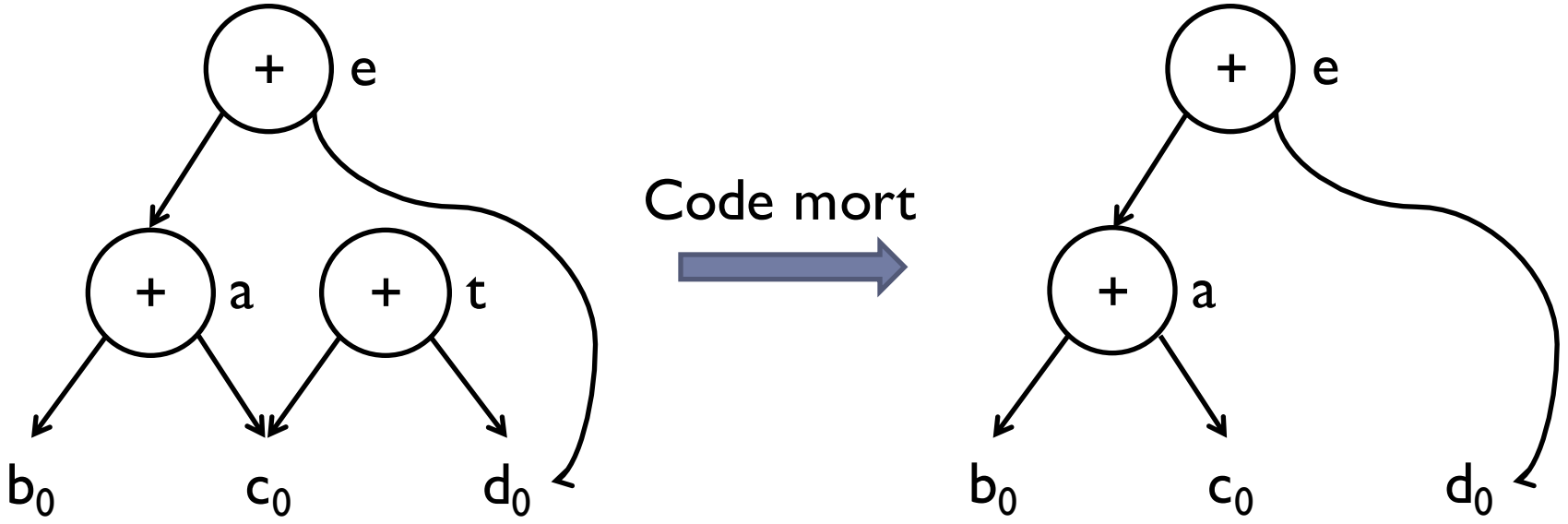
Commutativité et associativité (suite)



$e = b_0 + (c_0 + d_0)$

$e = (b_0 + c_0) + d_0$

Commutativité et associativité (suite)



Références à un tableau

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les références à un tableau doivent être traitées avec prudence. Considérons l'exemple suivant:

```
x = a[i]
a[j] = y
z = a[i]
```

- Est-il légitime de remplacer z = a[i] par z = x? Par exemple, si *i* == *j* ?

Références à un tableau (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- La manière de créer les nœuds du DAG pour les références à un tableau est la suivante:
  1. Les assignations à partir d'un tableau, comme x = a[i], génère un nœud =[] et deux enfants représentant la valeur du tableau (a) et l'index (i). Les étiquettes de variable sont placées comme avant.
  2. Les assignations à un tableau, comme a[j] = y, sont représentées par un nœud []= et possèdent trois enfants représentant a, j et y. Il n'y a pas d'étiquette de variable.
  3. Finalement, les nœuds []= *tuent* les nœuds qui dépendent de leur tableau. Un nœud tué ne peut plus recevoir d'étiquette de variable et ne peut donc pas devenir une sous-expression commune.

Références à un tableau (suite)

```
x = a[i]
a[j] = y
z = a[i]
```

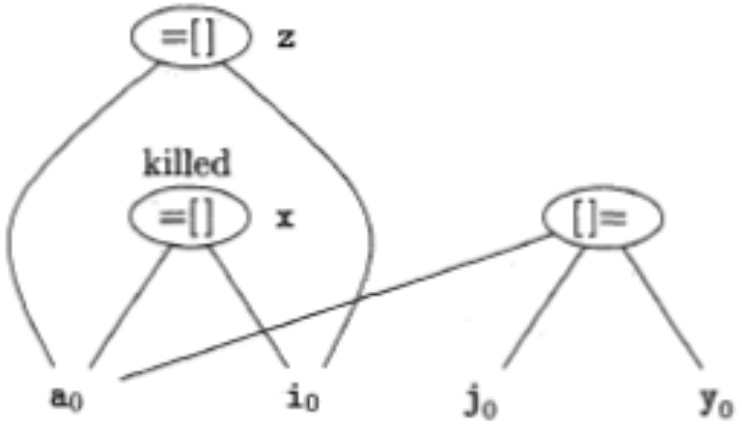


Figure 8.14: The DAG for a sequence of array assignments

Assignation à pointeurs

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Lors d'une assignation par déréférentiation de pointeurs, il n'est souvent pas possible de savoir à quelle variable un pointeur pointe.

- Par exemple, dans:

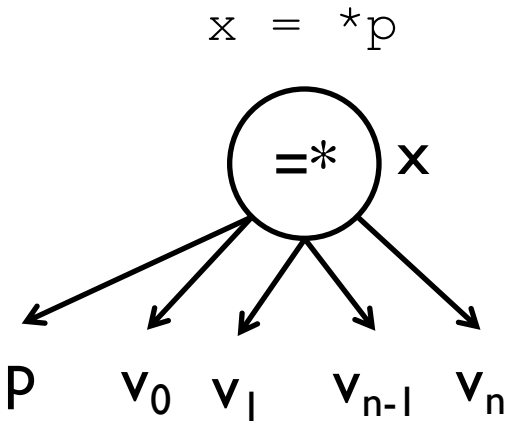
```
x = *p
*q = y
```

nous ne savons pas à quoi p et q pointent.

Assignation à pointeurs (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Conséquences sur le DAG:



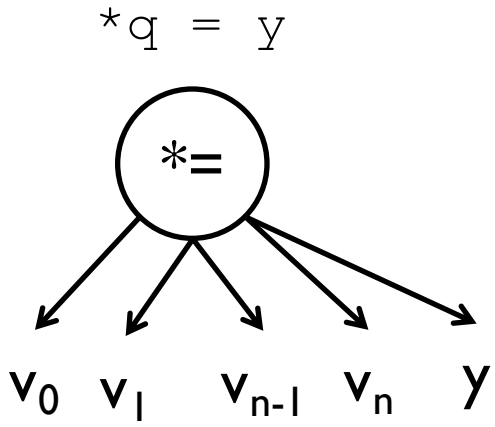
L'opérateur =* utilise potentiellement toutes les variables.

Il faudrait modifier la 3ième instruction de l'algorithme *next-use* afin de faire en sorte que l'opérateur =* rende vive **toutes** les variables (impacts sur l'élimination de code mort).

Assignation à pointeurs (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Conséquences sur le DAG:



L'opérateur *= défini potentiellement toutes les variables.

L'opérateur *= tue **tous** les nœuds présentement dans le DAG. Les nœuds tués ne peuvent plus faire partie de sous-expressions communes.

Générateur de code simple

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Considérons un générateur de code simplifié. Ses tâches principales seront:
  1. Gérer les assignations aux registres de manière efficace.
  2. Générer les instructions pour charger le contenu d'une adresse mémoire dans un registre (LD *reg, mem*).
  3. Générer les instructions pour stocker le contenu d'un registre dans une adresse mémoire (ST *mem, reg*).
  4. Générer les instructions qui feront des opérations. On suppose que les opérandes doivent être dans des registres et que le résultat doit être stocké dans un registre (OP *reg, reg, reg*).

Générateur de code simple (suite)

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Voir l'algorithme de génération de code dans les sections 8.6.1 et 8.6.2 du manuel.

- En résumé:
  1. Le générateur lit le code à 3 adresses une instruction à la fois.
  2. Si les opérandes sont déjà en registres, il génère l'instruction pour faire l'opération (OP), sinon il charge d'abord les opérandes (LD) dans des registres.
  3. Tout au long de la génération, le générateur maintient une structure de données qui indique, pour chaque variable, dans quel(s) registres et dans quelle(s) adresses mémoire la variable est stockée.
  4. À la fin d'un bloc, le générateur s'assure que toutes les variables vives soient stockées dans une adresse mémoire afin de libérer les registres pour le prochain bloc.

Générateur de code simple (exemple)

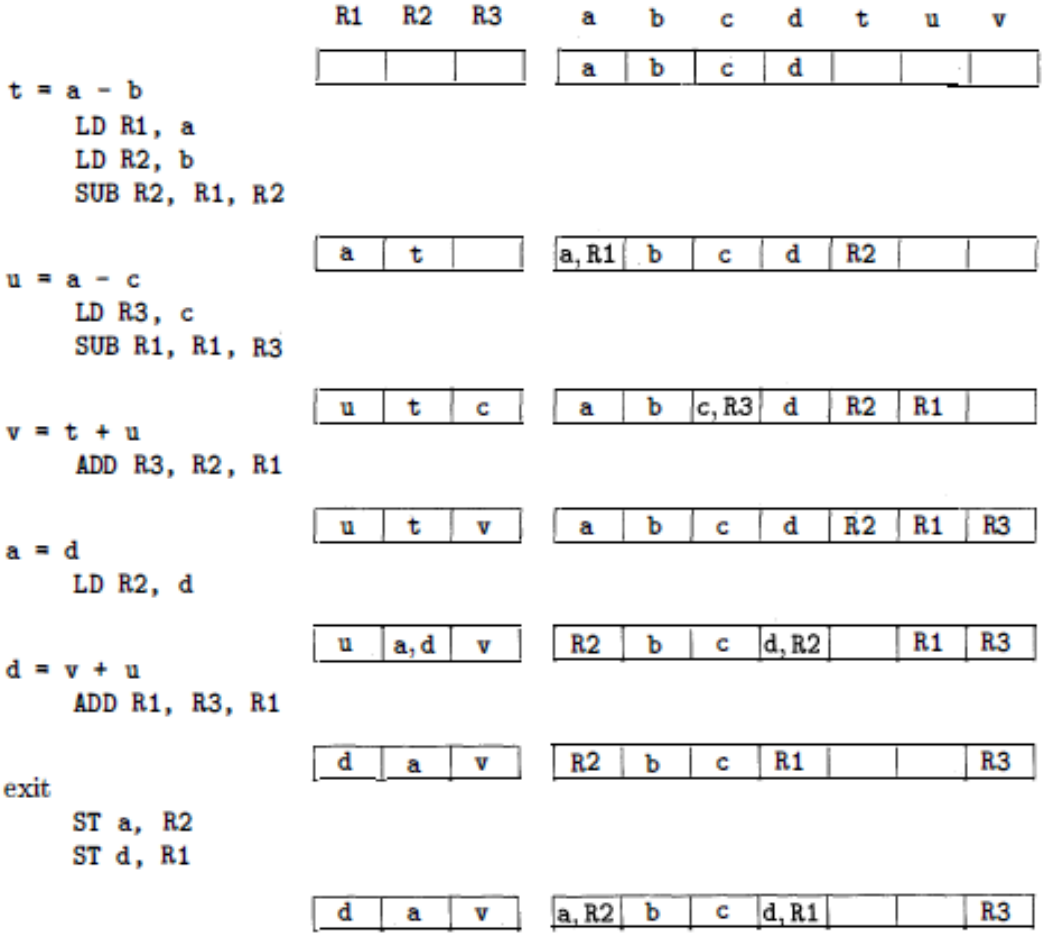


Figure 8.16: Instructions generated and the changes in the register and address descriptors