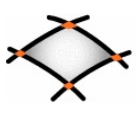
Archi'13,
Col de Porte,
25-29 mars 2013

Exécuter un programme en parallèle sur un processeur Many-core

Bernard Goossens et David Parello

Université de Perpignan Via Domitia
DALI, équipe-projet du LIRMM



UPVD
Université de Perpignan Via Domitia

DALI
Digits, Architectures et Logiciels Informatiques

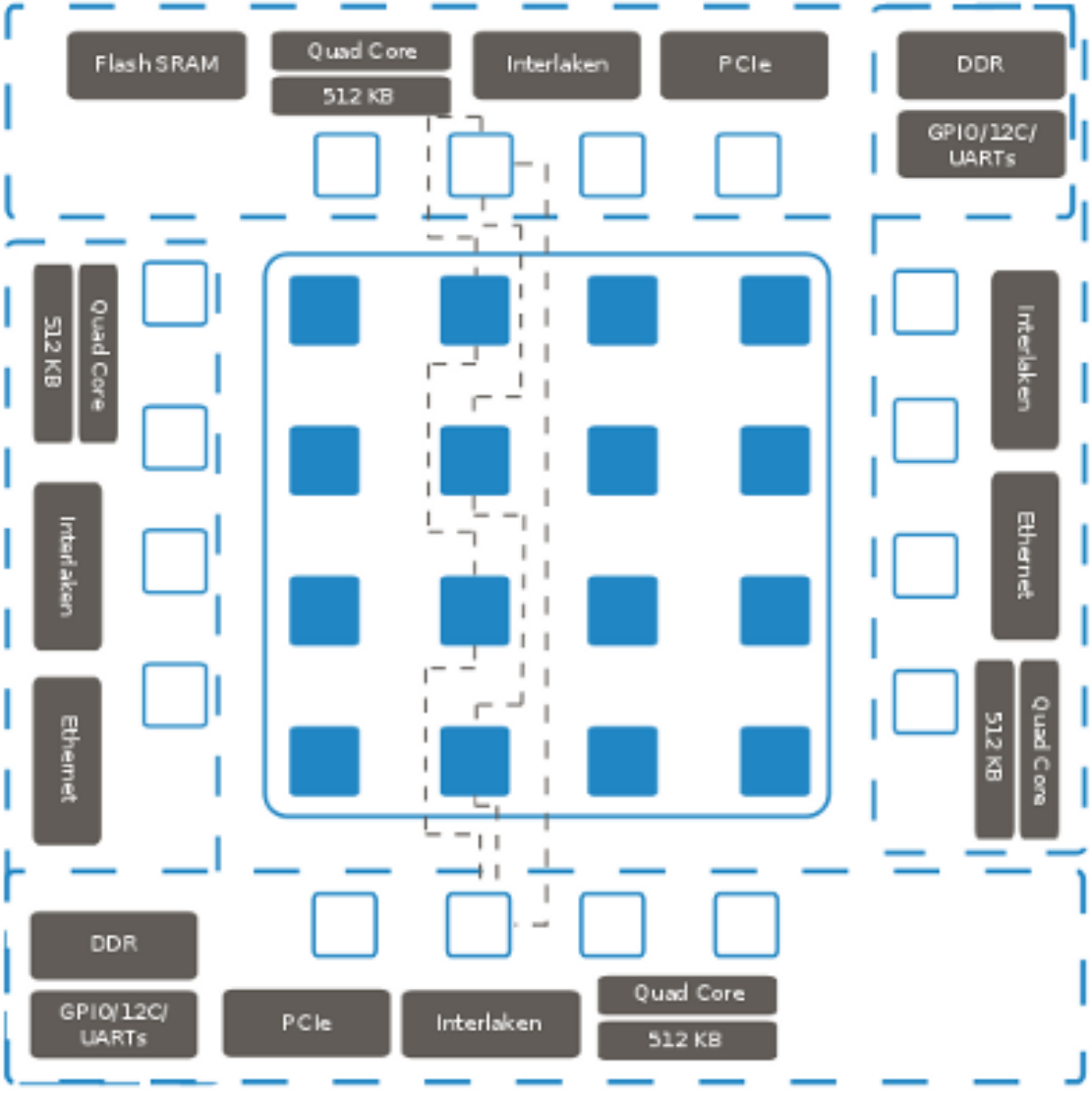
LIRMM

Plan de la présentation.

Un algorithme parallèle : la réduction de somme.

```
sum_reduce(t[0..n[) :
 if n=2 then t[0] + t[1]
 else sum_reduce(t[0..n/2[) + sum_reduce(t[n/2..n[)
```

Un processeur Many-core : le MPPA-256 de Kalray.



Comment exécuter un algorithme en parallèle sur un Many-core ?

- Implémentation pthread, MPI ...

Comment exécuter un algorithme en parallèle sur un Many-core ?

- Implémentation pthread, MPI ...
- Utiliser une bibliothèque de l'OS pour implémenter la distribution, la communication et la synchronisation.

Une implémentation Pthread.

```
typedef struct{int *t; int n;} ST;
void *sum_reduce(void *st){
 unsigned int s,s1,s2;
 ST str1,str2;
 pthread_t tid1, tid2;
 if (((ST *)st)->n==2){
  return ((ST *)st)->t[0] + ((ST *)st)->t[1];
 }
 str1.t=((ST *)st)->t; str1.n=((ST *)st)->n/2;
 pthread_create(&tid1,NULL,sum_reduce,(void *)&str1);
 str2.t=((ST *)st)->t + ((ST *)st)->n/2; str2.n=((ST *)st)->n/2;
 pthread_create(&tid2,NULL,sum_reduce,(void *)&str2);
 pthread_join(tid1,(void *)&s1); pthread_join(tid2,(void *)&s2);
 s=s1+s2; pthread_exit((void *)s);
}
```

Une implémentation MPI.

```
#include "mpi.h"
...
//i: local data element; s: recv, update (sum) and send s;
MPI_Reduce(&i, &s, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
...
```

Une implémentation pour GPU en CUDA.

```
__global__ void sum_reduce(int *t, int *sum){
 int tid = threadIdx.x;
 extern __shared__ s_data[];
 //to load the data from global memory to warp local shared memory
 s_data[tid] = ...;
 __syncthreads();
 for (int i=s_data_sizel; i>0; i/=2){
   if (tid < i){
     s_data[tid] = s_data[tid] + s_data[i+tid];
     __syncthreads();
   }
 }
 if (tid==0) sum[0]=s_data[0];
}
```

Une implémentation en C.

```
(0) unsigned int sum_reduce(unsigned int t[], unsigned int n){
(1)   if (n==2)
(2)     return t[0]+t[1];
(3)   else
(7)     return
(4)       sum_reduce(t,n/2)
(6)     +
(5)       sum_reduce(&t[n/2],n/2);
(8) }
```

Exécution du code C.

```
sum_reduce(&t[0], 8) =
  (1) (3) (4)
          sum_reduce(&t[0],4)
            (1) (3) (4)
                    sum_reduce(&t[0],2)
                      (1) (2)
            (5)
                    sum_reduce(&t[2],2)
                      (1) (2)
            (6) (7)
  (5)
          sum_reduce(&t[4],4)
            (1) (3) (4)
                    sum_reduce(&t[4],2)
                      (1) (2)
            (5)
                    sum_reduce(&t[6],2)
                      (1) (2)
            (6) (7)
  (6) (7)

trace = (1) (3) (4) (1) (3) (4) (1) (2) (5)
        (1) (2) (6) (7) (5) (1) (3) (4) (1)
        (2) (5) (1) (2) (6) (7) (6) (7)
```

Qu'est-ce qui séquentialise l'implémentation C ?

Qu'est-ce qui séquentialise l'implémentation C ?

- Ce n'est pas le code C.

Qu'est-ce qui séquentialise l'implémentation C ?

- Ce n'est pas le code C.
- C'est le modèle d'exécution employé : le modèle de Von Neumann.

Qu'est-ce qui séquentialise l'implémentation C ?

- Ce n'est pas le code C.
- C'est le modèle d'exécution employé : le modèle de Von Neumann.
- Ce modèle suppose l'existence d'un pointeur d'instruction IP contrôlant la phase d'extraction du programme enregistré en mémoire.

Traduction du code C pour une architecture Von Neumann.

```
(01) sum_reduce:   pushq %rbx                    #save t
(02)               pushq %rbp                    #save n
(03)               subq  $8, %rsp                #allocate tmp
(04)               cmpq  $2, %rbp                #if (n!=2)
(05)               jne   .L2                     #goto .L2
(06)               movq  8(%rbx), %rax           #a=t[1]
(07)               addq  (%rbx), %rax            #a+=t[0]
(08)               jmp   .L3                     #goto .L3
(09) .L2:          shrq  %rbp                    #n=n/2
(10)               call  sum_reduce              #a=sum(t,n/2)
(11)               movq  %rax, 0(%rsp)           #tmp=a
(12)               leaq  (%rbx,%rbp,8), %rbx     #t=&t[n/2]
(13)               call  sum_reduce              #a=sum(&t[n/2],n/2)
(14)               addq  0(%rsp), %rax           #a+=tmp
(15) .L3:          addq  $8, %rsp                #free tmp
(16)               popq  %rbp                    #restore n
(17)               popq  %rbx                    #restore t
(18)               ret                           #return a
```

Exécution du code assembleur.

```
sum_reduce(&t[0], 8) =
 7  (01)-(05), (09), (10)
          sum_reduce(&t[0],4)
 7            (01)-(05), (09), (10)
                    sum_reduce(&t[0],2)
12                        (01)-(08), (15)-(18)
 3            (11)-(13)
                    sum_reduce(&t[2],2)
12                        (01)-(08), (15)-(18)
 5            (14)-(18)
 3  (11)-(13)
          sum_reduce(&t[4],4)
 7            (01)-(05), (09), (10)
                    sum_reduce(&t[0],2)
12                        (01)-(08), (15)-(18)
            (11)-(13)
                    sum_reduce(&t[2],2)
12                        (01)-(08), (15)-(18)
 5            (14)-(18)
 5  (14)-(18)
90

trace = 90 instructions
```

Le parallélisme d'instruction ou ILP.

- ILP proche du pointeur d'instruction (centaines d'instructions).

Le parallélisme d'instruction ou ILP.

- ILP proche du pointeur d'instruction (centaines d'instructions).
- ILP distant, c-à-d K ou M ou G instructions.

L'ILP proche.

- Il est contrecarré par les dépendances.

L'ILP proche.

- Il est contrecarré par les dépendances.
- Vraies dépendances : producteur -> consommateur (données et contrôle).

L'ILP proche.

- Il est contrecarré par les dépendances.
- Vraies dépendances : producteur -> consommateur (données et contrôle).
- Fausses dépendances : ressources (registres et mémoire).

Vraies dépendances sur l'exemple.

```
(01) sum_reduce:   pushq %rbx                  # prod %rbx
(02)               pushq %rbp                  # 01, prod %rbp
(03)               subq  $8, %rsp              # 02
(04)               cmpq  $2, %rbp              # prod %rbp
(05)               jne   .L2                   # 04
(06)               movq  8(%rbx), %rax         # 05, prod %rbx, prod (%rbx)
(07)               addq  (%rbx), %rax          # 05, 06, prod %rbx, prod (%rbx
(08)               jmp   .L3                   # 05
(09) .L2:          shrq  %rbp                  # 05, prod %rbp
(10)               call  sum_reduce            # 05, 02
(11)               movq  %rax, 0(%rsp)         # 05, prod (%rsp), prod %rax
(12)               leaq  (%rbx,%rbp,8), %rbx   # 05, prod %rbx, prod %rbp
(13)               call  sum_reduce            # 05, prod %rsp
(14)               addq  0(%rsp), %rax         # 05, prod %rsp, prod (%rsp)
(15) .L3:          addq  $8, %rsp              # 02
(16)               popq  %rbp                  # 15, 02
(17)               popq  %rbx                  # 16, 01
(18)               ret                         # 17, prod (%rsp)
```

Fausses dépendances sur l'exemple.

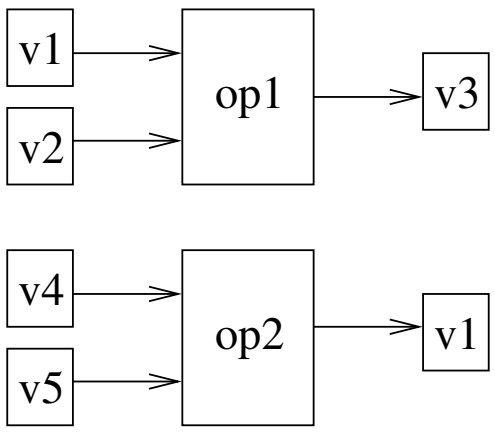
```
(01) sum_reduce:   pushq %rbx                  # ecr en (%rsp), lec en (%rsp)
(02)               pushq %rbp                  # ecr en (%rsp), lec en (%rsp)
(03)               subq  $8, %rsp              # 02
(04)               cmpq  $2, %rbp              # ecr en %eflags, lec en %eflag
(05)               jne   .L2                   #
(06)               movq  8(%rbx), %rax         # ecr en %rax, lec en %rax
(07)               addq  (%rbx), %rax          # 06
(08)               jmp   .L3                   #
(09) .L2:          shrq  %rbp                  # ecr en %rbp, lec en %rbp
(10)               call  sum_reduce            # ecr en %rsp, lec en %rsp
(11)               movq  %rax, 0(%rsp)         # ecr en (%rsp), lec en (%rsp)
(12)               leaq  (%rbx,%rbp,8), %rbx   # ecr en %rbx, lec en %rbx
(13)               call  sum_reduce            # ecr en %rsp, lec en %rsp
(14)               addq  0(%rsp), %rax         # ecr en %rax, lec en %rax
(15) .L3:          addq  $8, %rsp              # ecr en %rsp, lec en %rsp
(16)               popq  %rbp                  # ecr en %rbp, lec en %rbp
(17)               popq  %rbx                  # ecr en %rbx, lec en %rbx
(18)               ret                         #
```

La duplication de l'espace par le renommage. Dépendances EAL.

```
v3 = v1 op1 v2
v1 = v4 op2 v5
```

La duplication de l'espace par le renommage. Dépendances EAL.

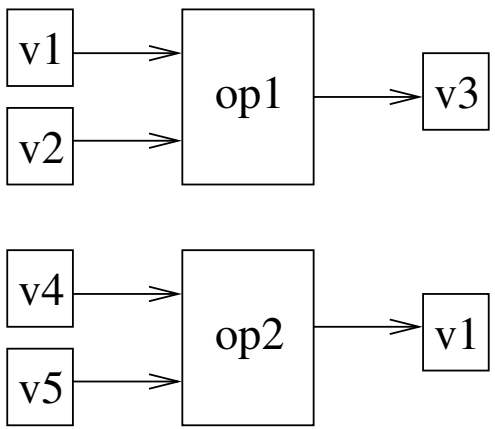
v3 = v1 op1 v2
v1 = v4 op2 v5



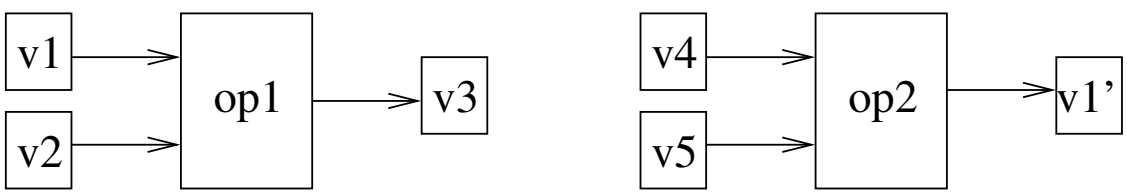
Sans renommage: 2 séquences

La duplication de l'espace par le renommage. Dépendances EAL.

```
v3 = v1 op1 v2
v1 = v4 op2 v5
```



Sans renommage: 2 séquences



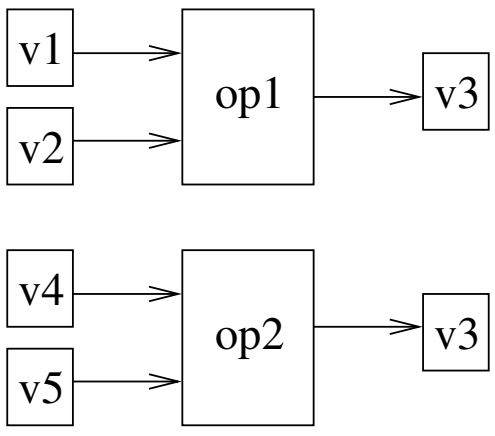
Avec renommage v1–>v1': 1 séquence

La duplication de l'espace par le renommage. Dépendances EAE.

```
v3 = v1 op1 v2
v3 = v4 op2 v5
```

La duplication de l'espace par le renommage. Dépendances EAE.

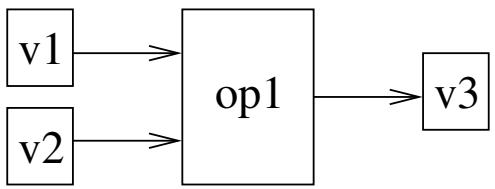
v3 = v1 op1 v2
v3 = v4 op2 v5

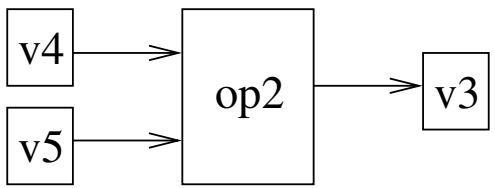


Sans renommage: 2 séquences

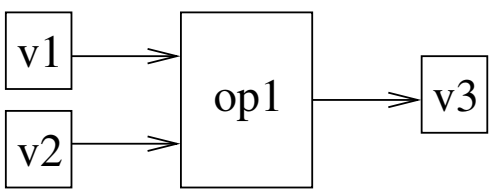
La duplication de l'espace par le renommage. Dépendances EAE.

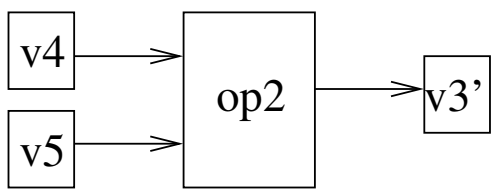
```
v3 = v1 op1 v2
v3 = v4 op2 v5
```





Sans renommage: 2 séquences





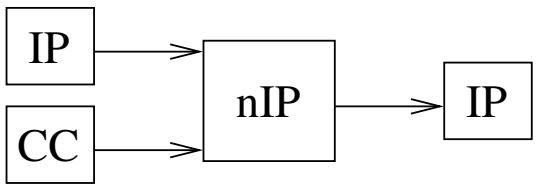
Avec renommage v3–>v3': 1 séquence

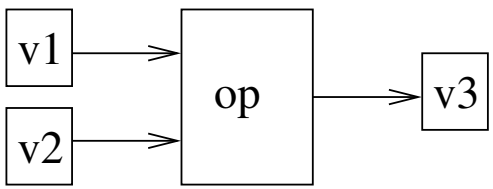
La spéculation de la source (IP) par la prédiction de saut. Dépendances de contrôle.

```
        jne .L1
.L1     v3 = v1 op v2
```

La spéculation de la source (IP) par la prédiction de saut. Dépendances de contrôle.

```
        jne .L1
.L1     v3 = v1 op v2
```

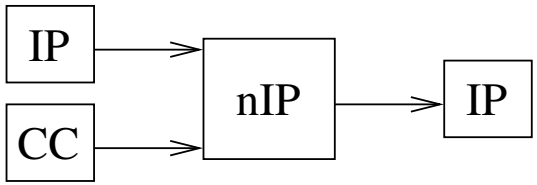


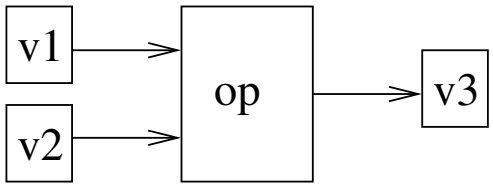


Sans prédiction: 2 séquences

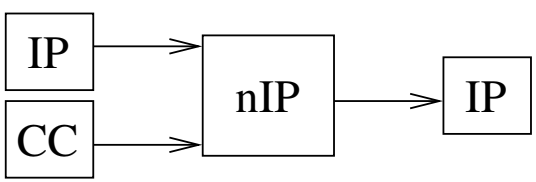
La spéculation de la source (IP) par la prédiction de saut. Dépendances de contrôle.

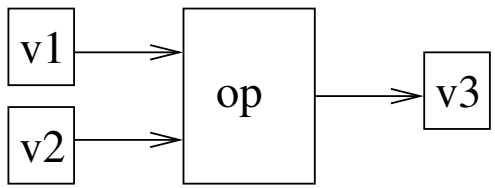
```
        jne .L1
.L1     v3 = v1 op v2
```





Sans prédiction: 2 séquences



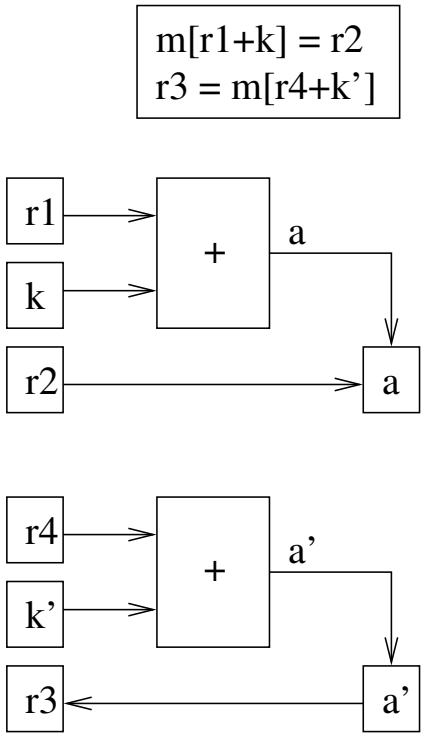


Avec prédiction: 1 séquence

La spéculation de la source par son chargement anticipé. Dépendances de mémoire.

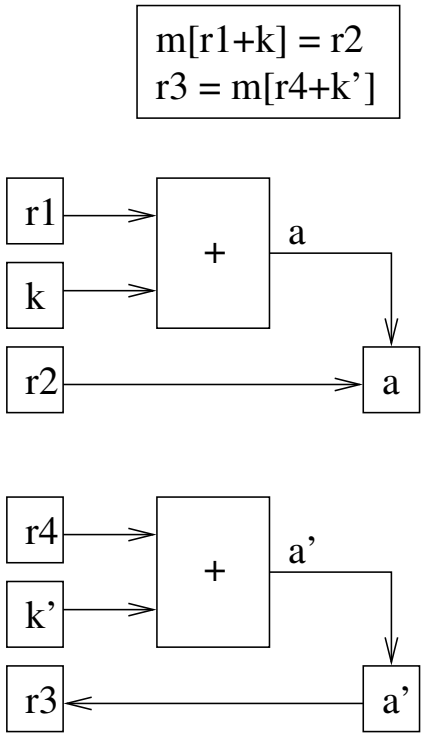
```
m[r1+k] = r2
r3 = m[r4+k']
```

La spéculation de la source par son chargement anticipé. Dépendances de mémoire.

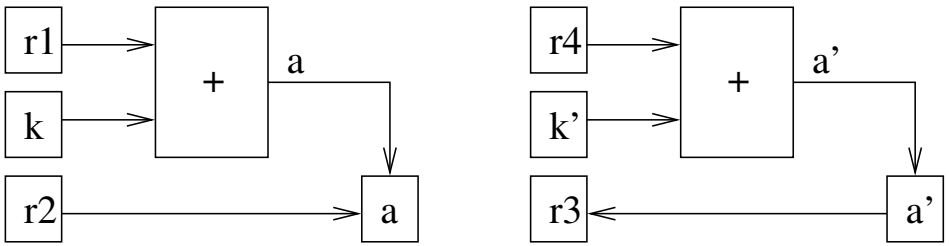


Sans spéculation: 2 séquences

La spéculation de la source par son chargement anticipé. Dépendances de mémoire.



Sans spéculation: 2 séquences

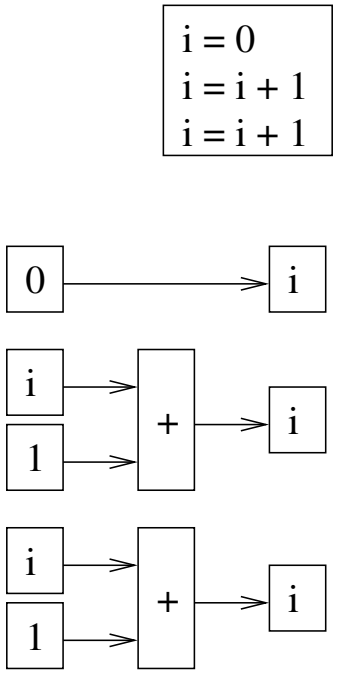


Avec spéculation: 1 séquence

Le calcul au renommage. Dépendances de données.

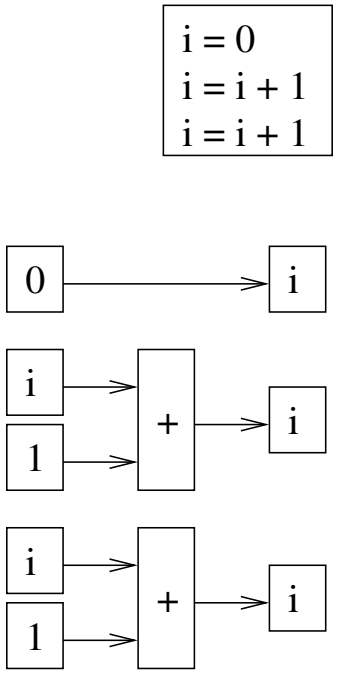
```
i = 0
i = i + 1
i = i + 1
```

Le calcul au renommage. Dépendances de données.

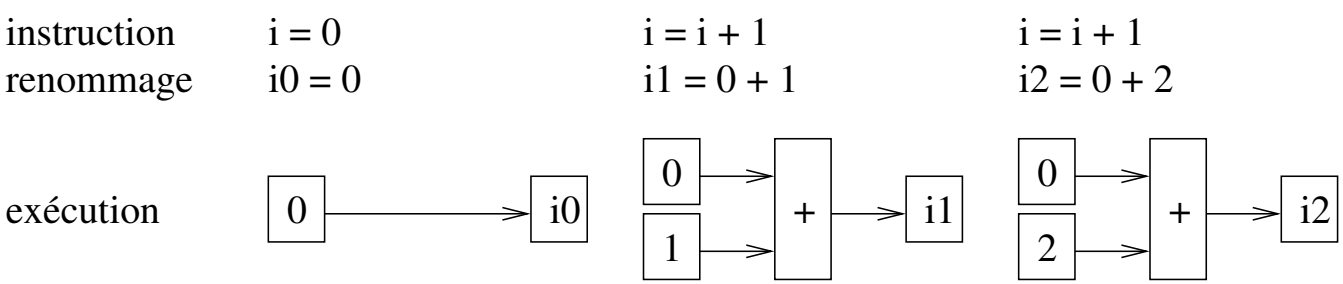


Sans précalcul: 3 séquences

Le calcul au renommage. Dépendances de données.



Sans précalcul: 3 séquences



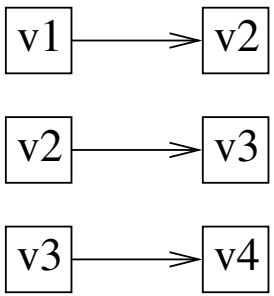
Avec précalcul: 1 séquence

Le partage de l'espace et l'élimination des copies. Dépendances de données.

```
mov v1, v2
mov v2, v3
mov v3, v4
```

Le partage de l'espace et l'élimination des copies. Dépendances de données.

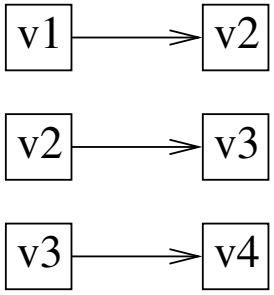
```
mov v1, v2
mov v2, v3
mov v3, v4
```



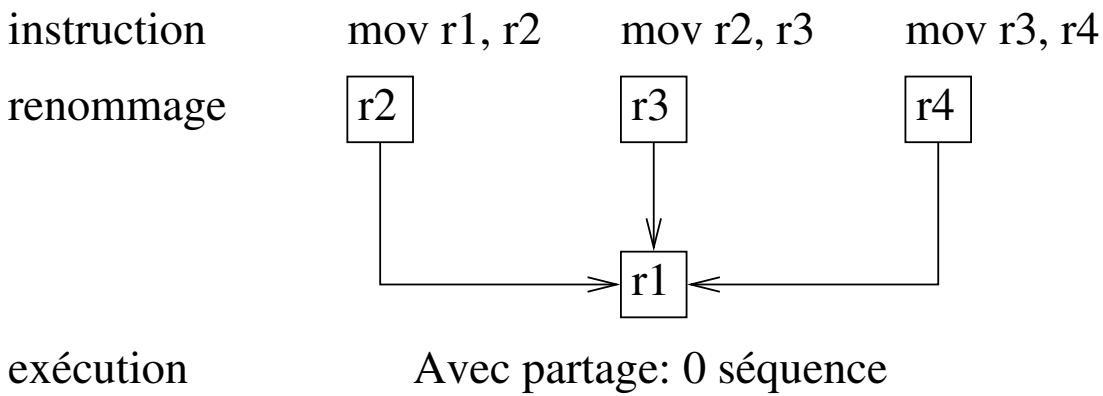
Sans partage: 3 séquences

Le partage de l'espace et l'élimination des copies. Dépendances de données.

```
mov v1, v2
mov v2, v3
mov v3, v4
```

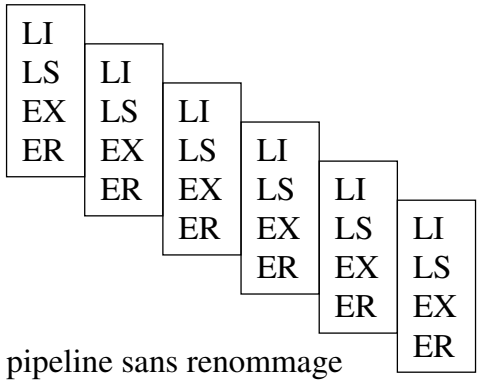


Sans partage: 3 séquences



Une phase de renommage.

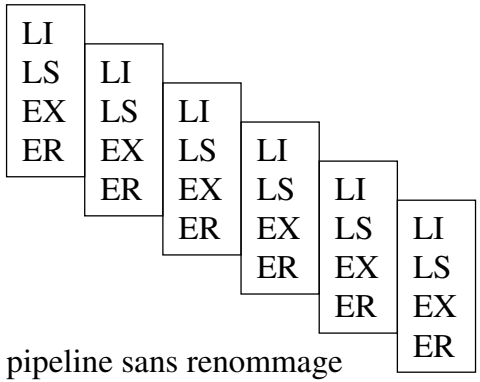
lecture d'une instruction
lecture de ses sources
exécution
écriture de ses résultats



pipeline sans renommage

Une phase de renommage.

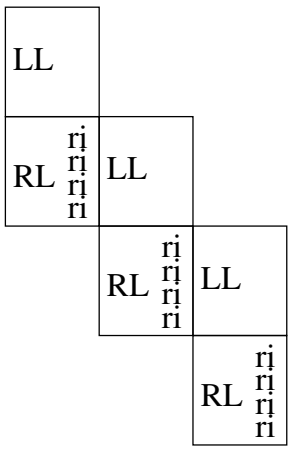
lecture d'une instruction
lecture de ses sources
exécution
écriture de ses résultats



pipeline sans renommage

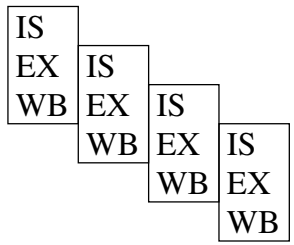
lecture d'une ligne d'instructions

renommage d'une instruction



pipeline de renommage

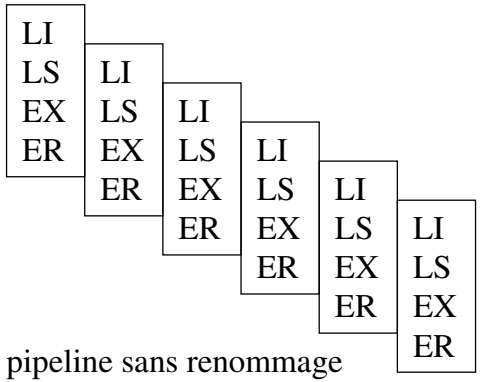
démarrage ooo des instructions prêtes
exécution en parallèle
terminaison



pipeline d'exécution ooo

Une phase de renommage.

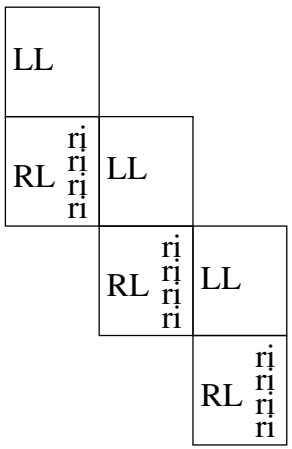
lecture d'une instruction
lecture de ses sources
exécution
écriture de ses résultats



pipeline sans renommage

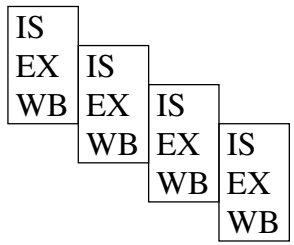
lecture d'une ligne d'instructions

renommage d'une instruction

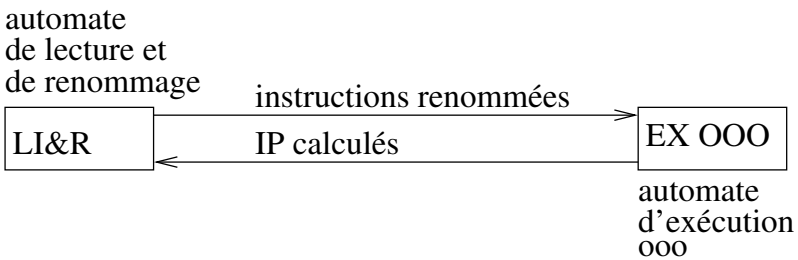


pipeline de renommage

démarrage ooo des instructions prêtes
exécution en parallèle
terminaison



pipeline d'exécution ooo



Un prédicteur de saut.

- Prédit les sauts conditionnels (direction et cible).

Un prédicteur de saut.

- Prédit les sauts conditionnels (direction et cible).
- Prédit les sauts inconditionnels indirects (cible).

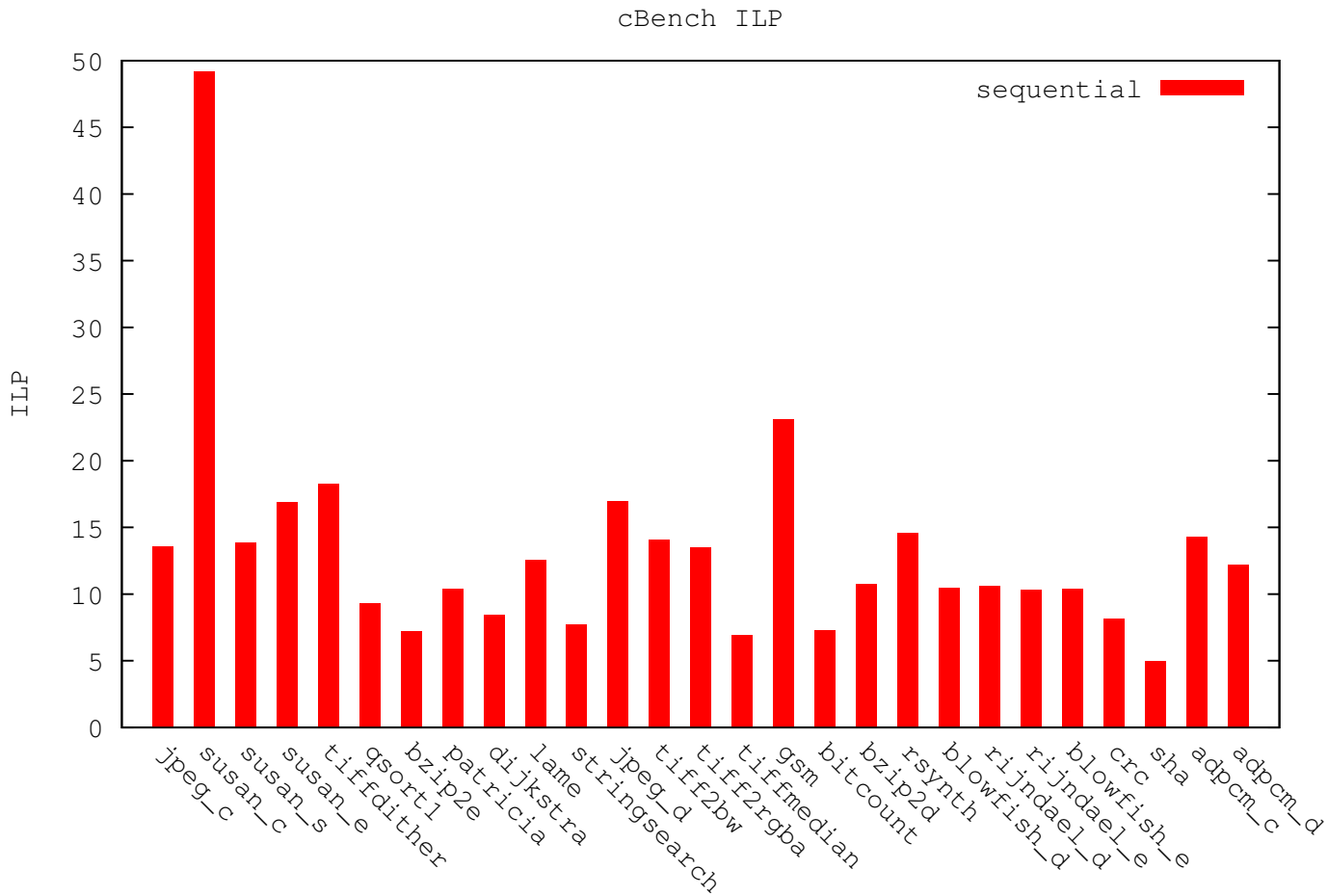
Un prédicteur de saut.

- Prédit les sauts conditionnels (direction et cible).
- Prédit les sauts inconditionnels indirects (cible).
- Prédit les retours de fonction.

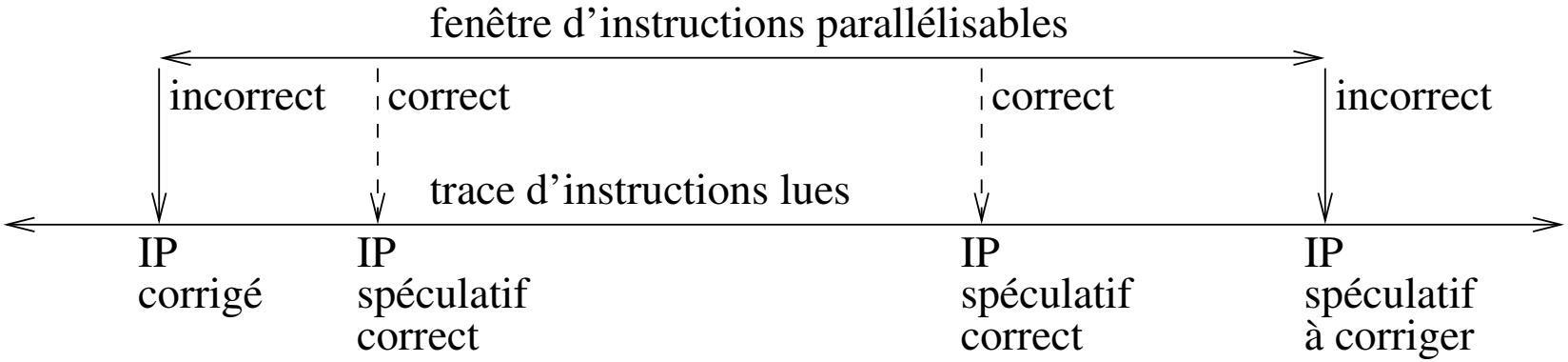
Un prédicteur de saut.

- Prédit les sauts conditionnels (direction et cible).
- Prédit les sauts inconditionnels indirects (cible).
- Prédit les retours de fonction.
- Taux de succès : 97%.

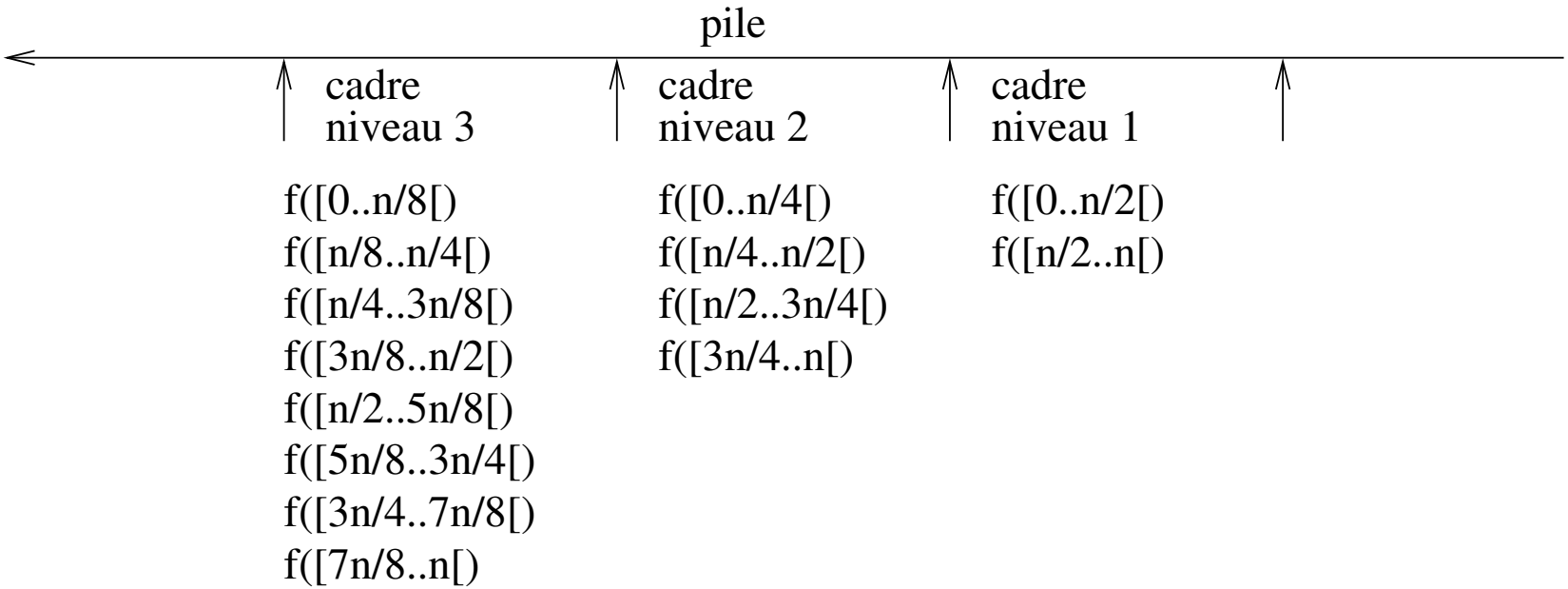
Impact de la parallélisation par le renommage mémoire et la prédiction de saut.



Premier verrou : séquentialisation par le contrôle.



Second verrou : séquentialisation par la gestion de la pile.



Comment lever le verrou sur le contrôle ?

- On ne sait pas prédire une trace au delà de quelques centaines d'instructions mais

Comment lever le verrou sur le contrôle ?

- On ne sait pas prédire une trace au delà de quelques centaines d'instructions mais

  on en connaît des points de passage sûrs tout au long du parcours.

Comment lever le verrou sur le contrôle ?

- On ne sait pas prédire une trace au delà de quelques centaines d'instructions mais

  on en connaît des points de passage sûrs tout au long du parcours.

- Le point de retour d'un appel de fonction.

Comment lever le verrou sur le contrôle ?

- On ne sait pas prédire une trace au delà de quelques centaines d'instructions mais

  on en connaît des points de passage sûrs tout au long du parcours.

- Le point de retour d'un appel de fonction.
- La sortie d'une boucle.

Comment lever le verrou sur la pile ?

- On ne sait pas positionner le cadre d'un appel de fonction dans la pile sans avoir positionné le cadre de la fonction appelante mais

Comment lever le verrou sur la pile ?

- On ne sait pas positionner le cadre d'un appel de fonction dans la pile sans avoir positionné le cadre de la fonction appelante mais

  la position du pointeur de pile au retour est la même qu'à l'appel.

Comment lever le verrou sur la pile ?

- On ne sait pas positionner le cadre d'un appel de fonction dans la pile sans avoir positionné le cadre de la fonction appelante mais

  la position du pointeur de pile au retour est la même qu'à l'appel.

- Spéculer sur la position du pointeur de pile.

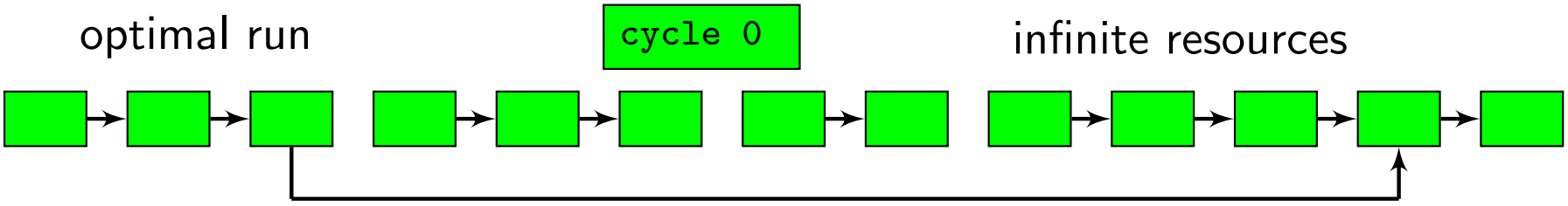
Comment lever le verrou sur la pile ?

- On ne sait pas positionner le cadre d'un appel de fonction dans la pile sans avoir positionné le cadre de la fonction appelante mais

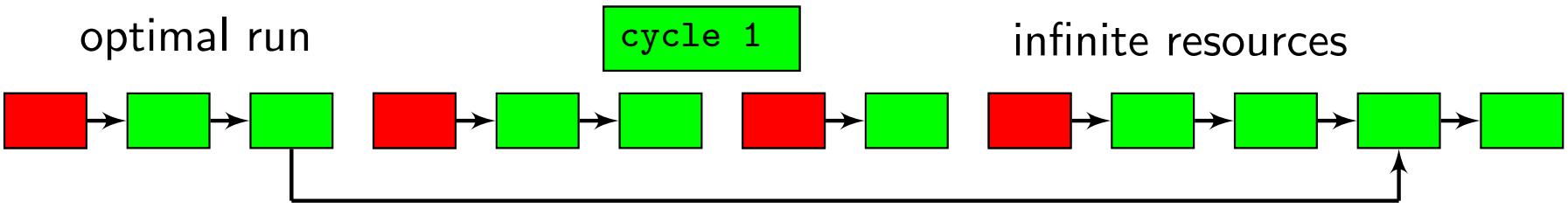
  la position du pointeur de pile au retour est la même qu'à l'appel.

- Spéculer sur la position du pointeur de pile.
- Distinguer les cadres d'appels de même niveau par renommage.

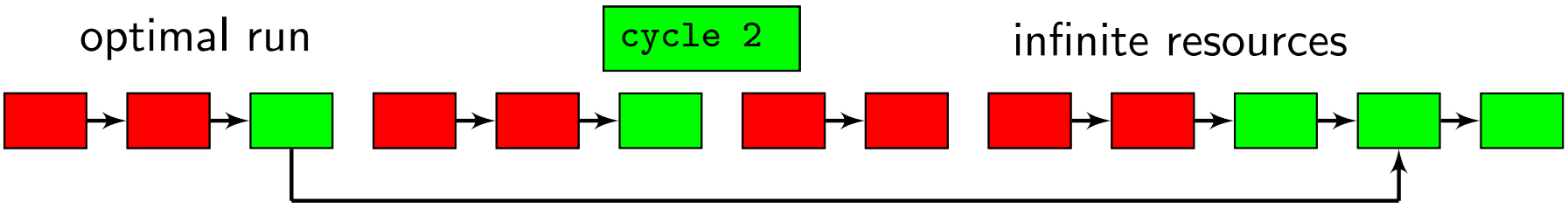
How to capture distant ILP.



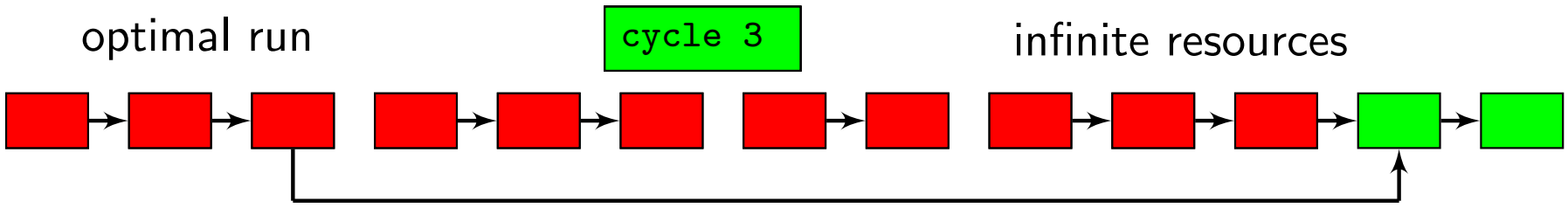
How to capture distant ILP.



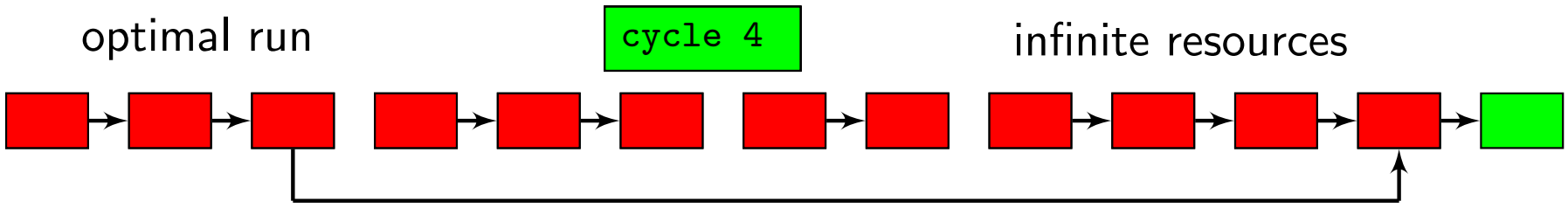
How to capture distant ILP.



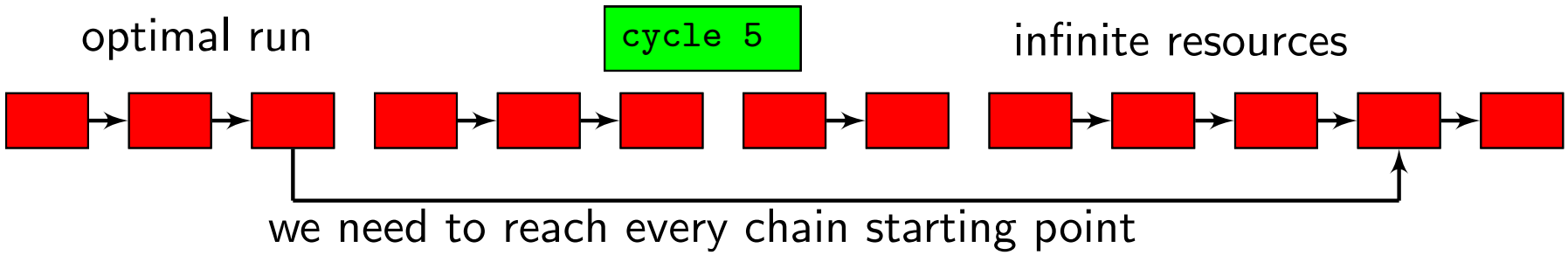
How to capture distant ILP.



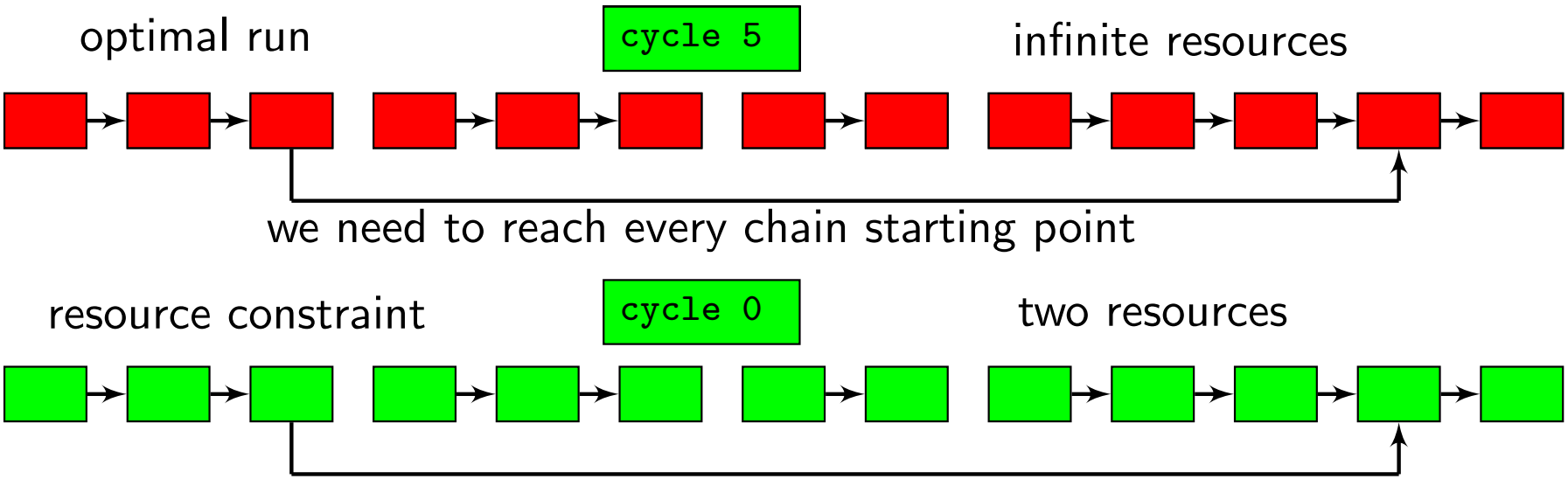
How to capture distant ILP.



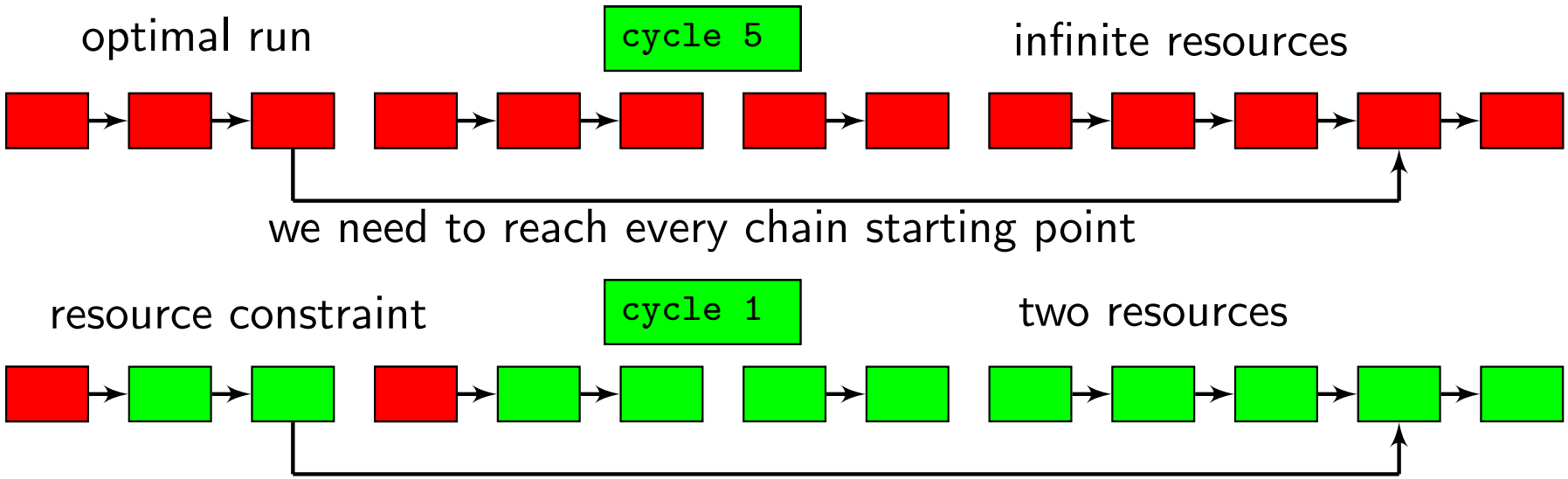
How to capture distant ILP.



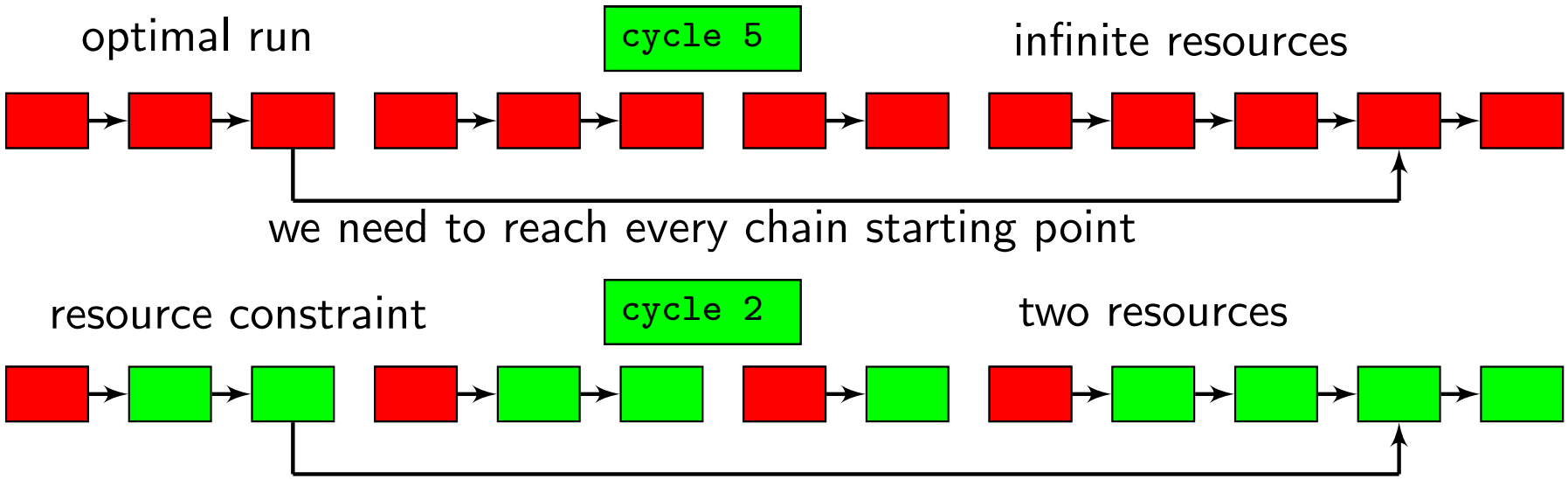
How to capture distant ILP.



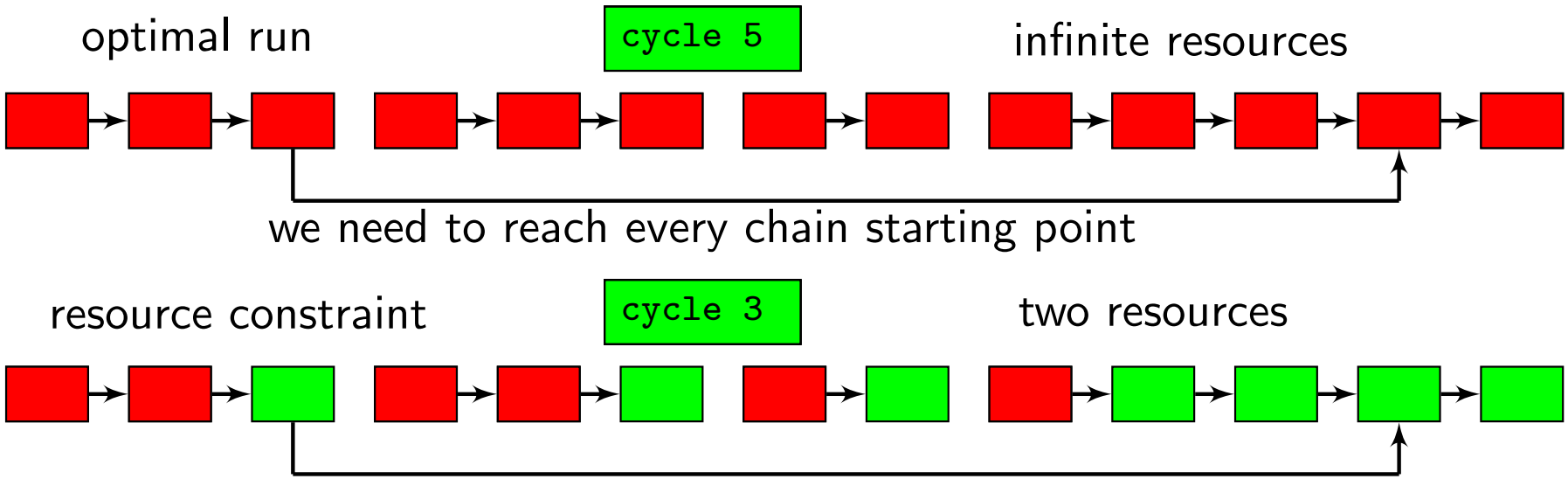
How to capture distant ILP.



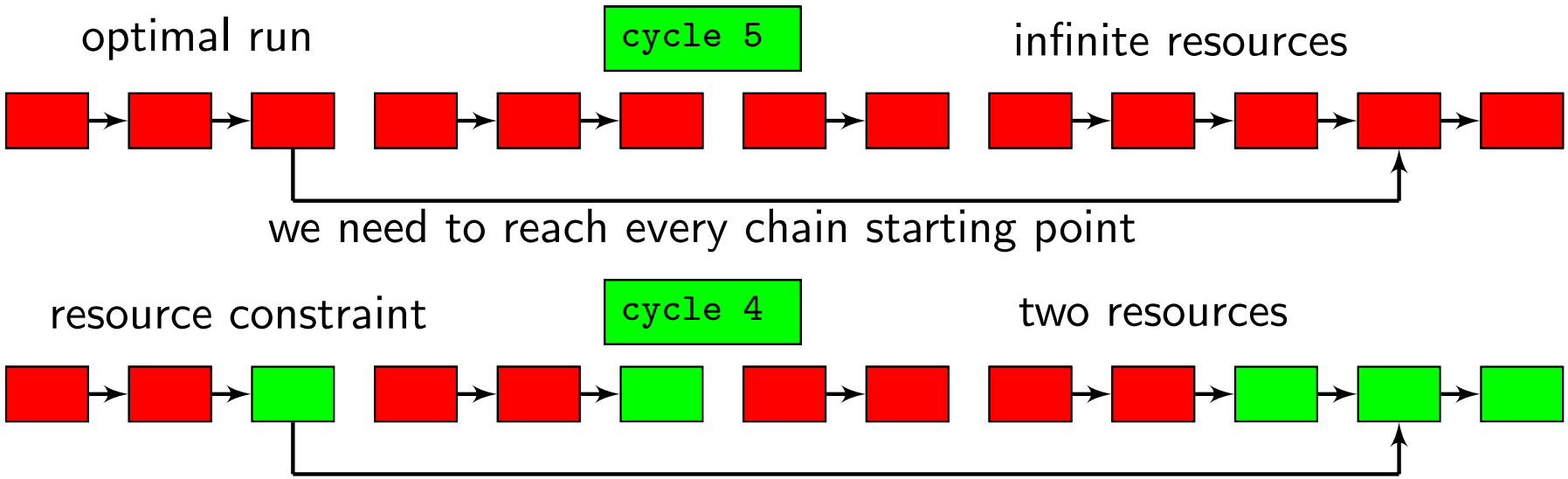
How to capture distant ILP.



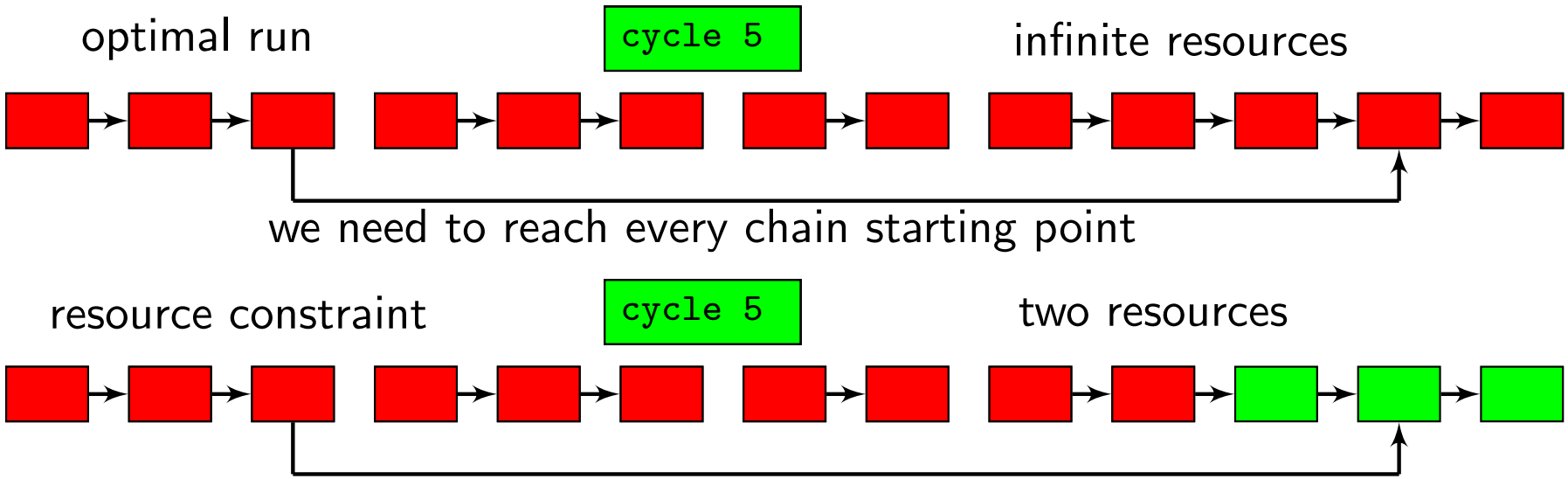
How to capture distant ILP.



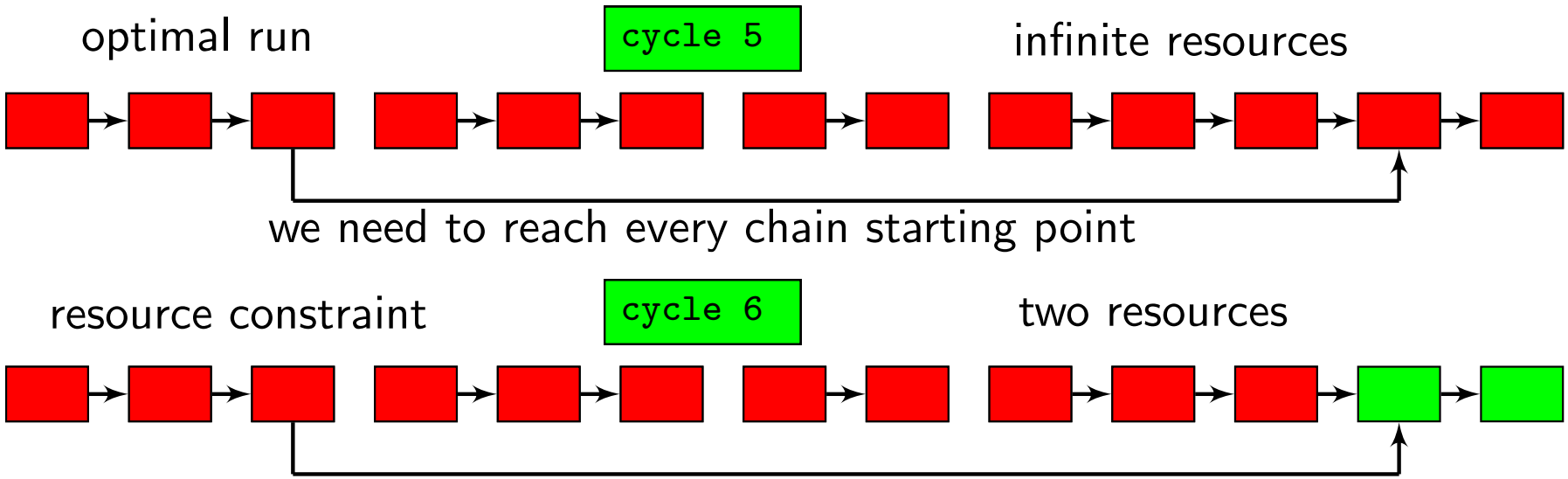
How to capture distant ILP.



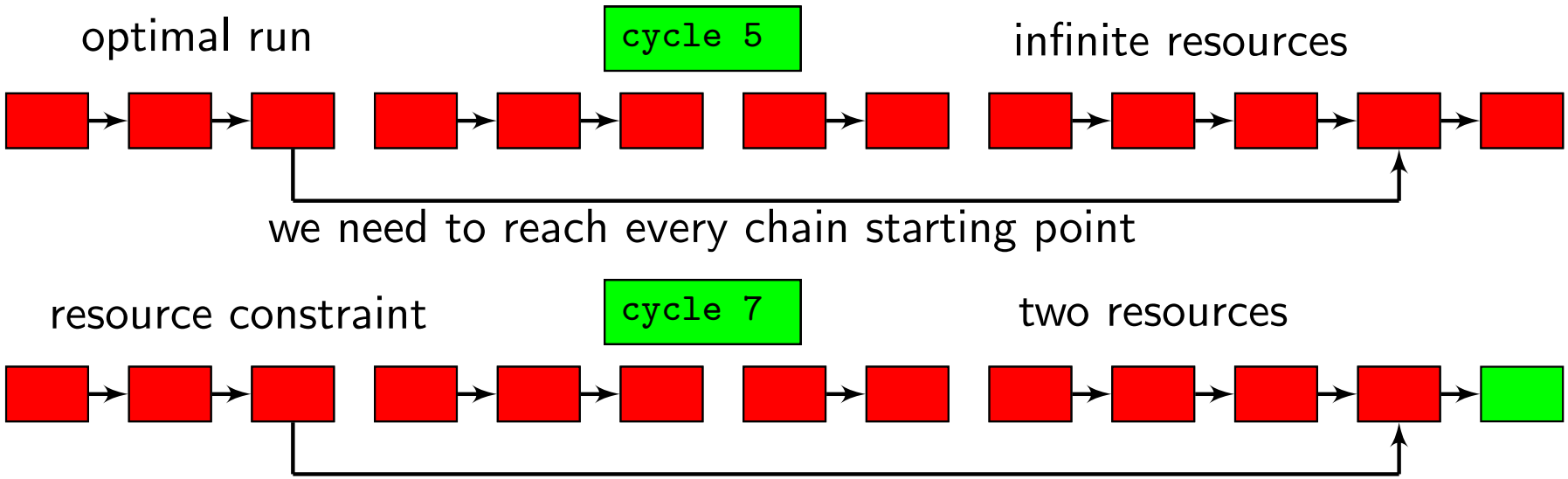
How to capture distant ILP.



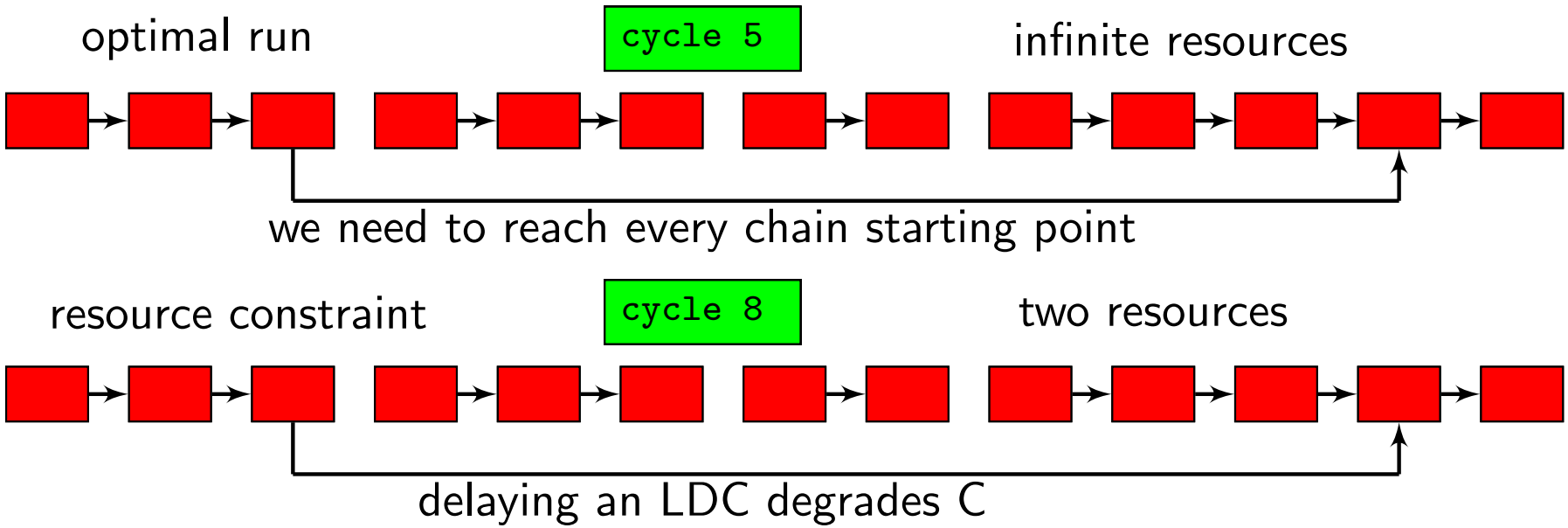
How to capture distant ILP.



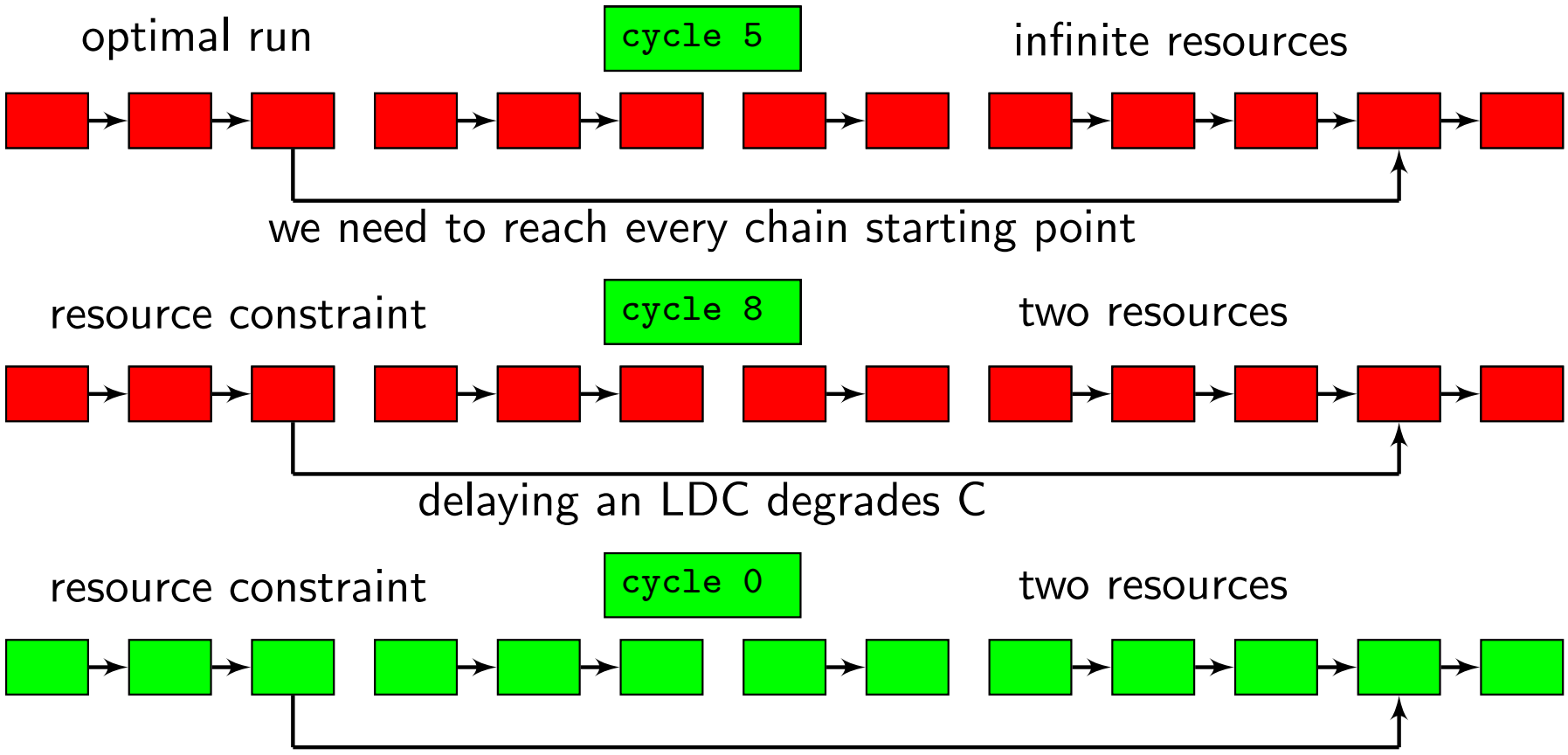
How to capture distant ILP.



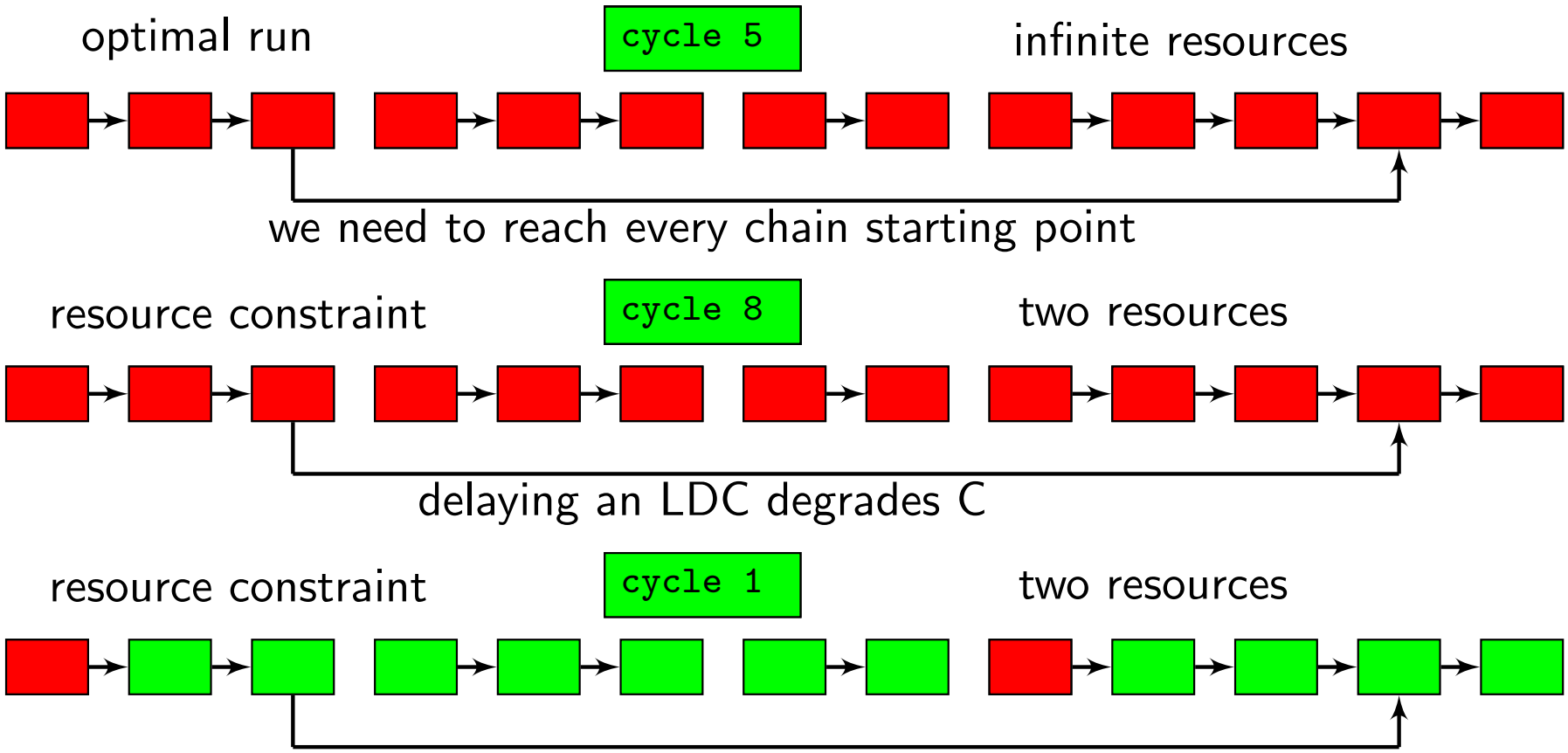
How to capture distant ILP.



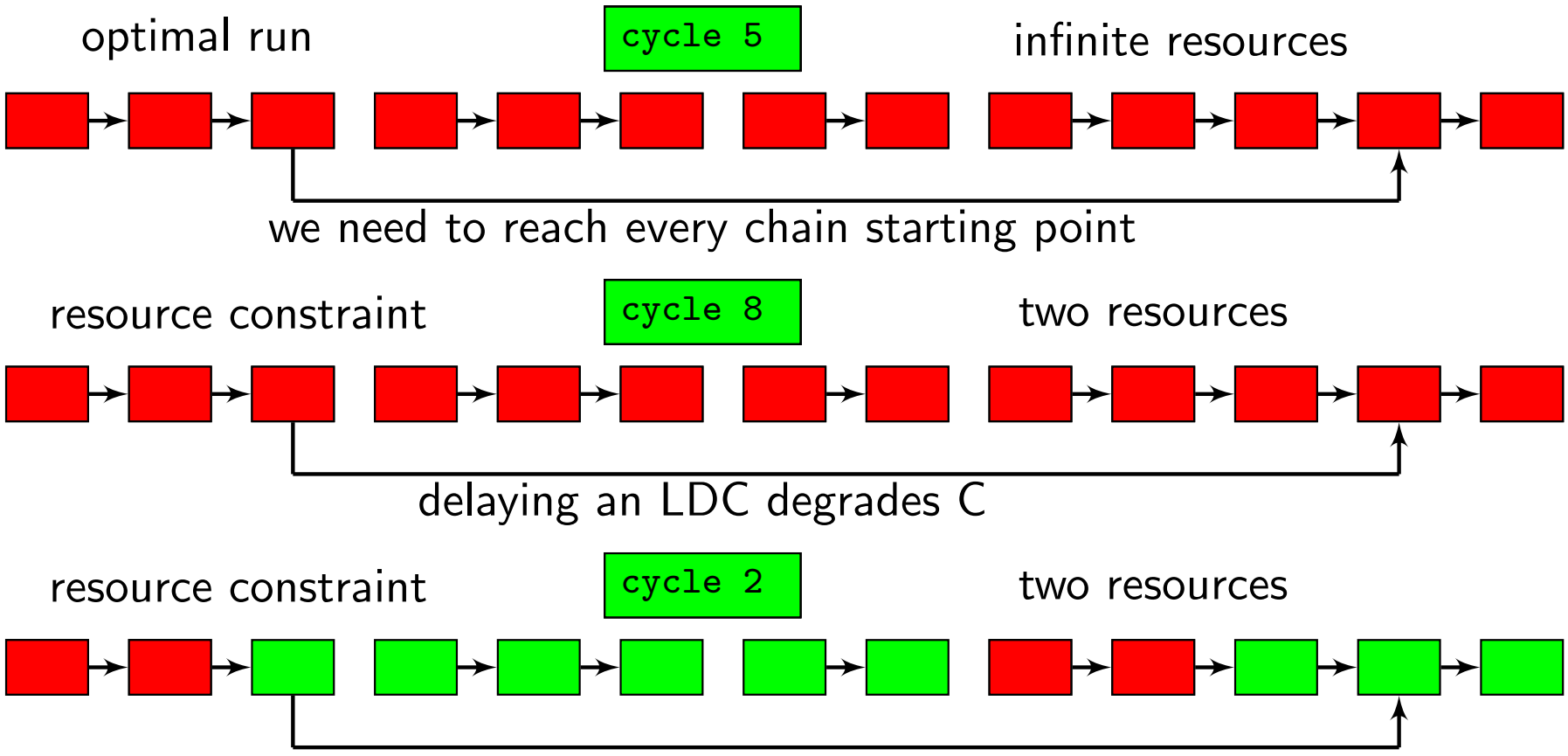
How to capture distant ILP.



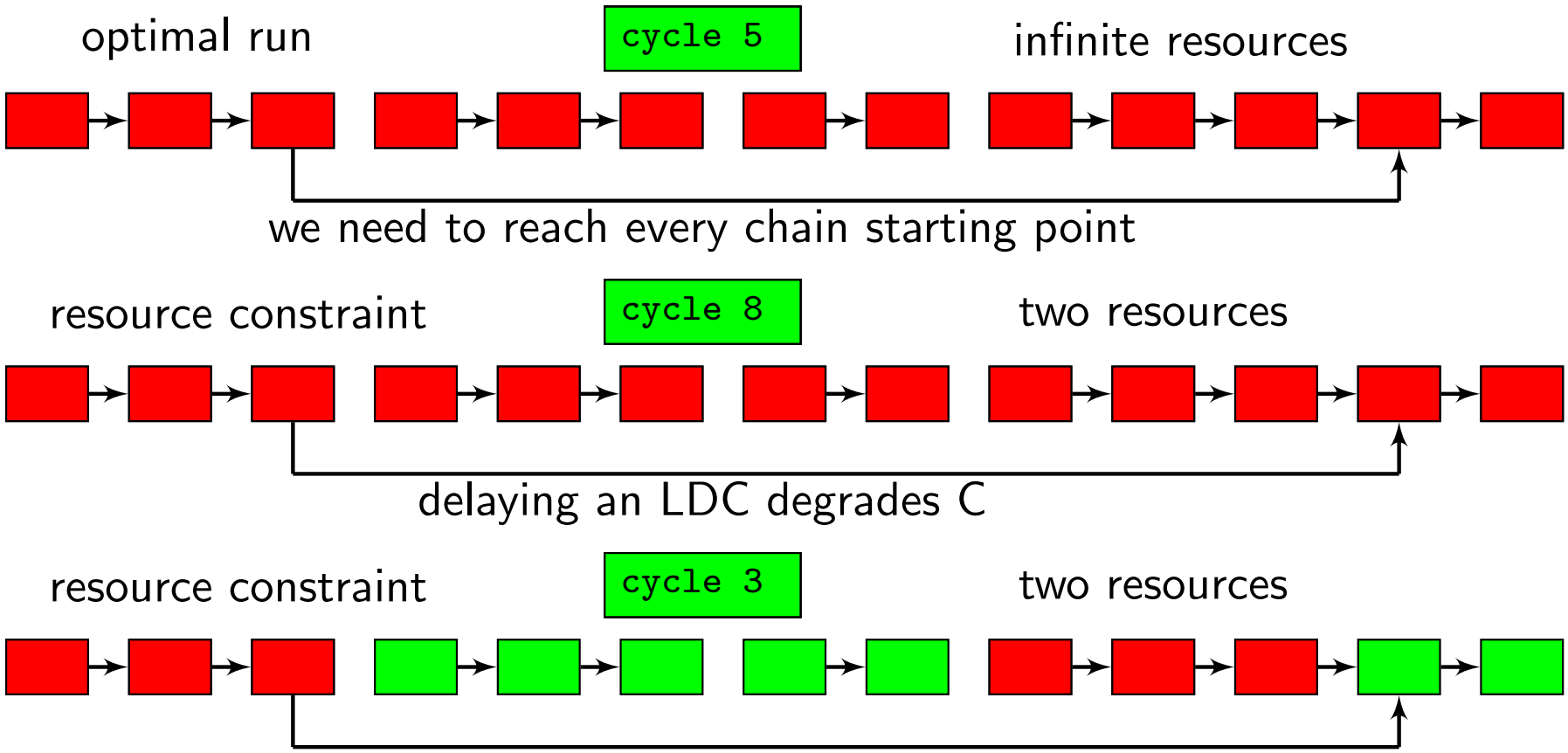
How to capture distant ILP.



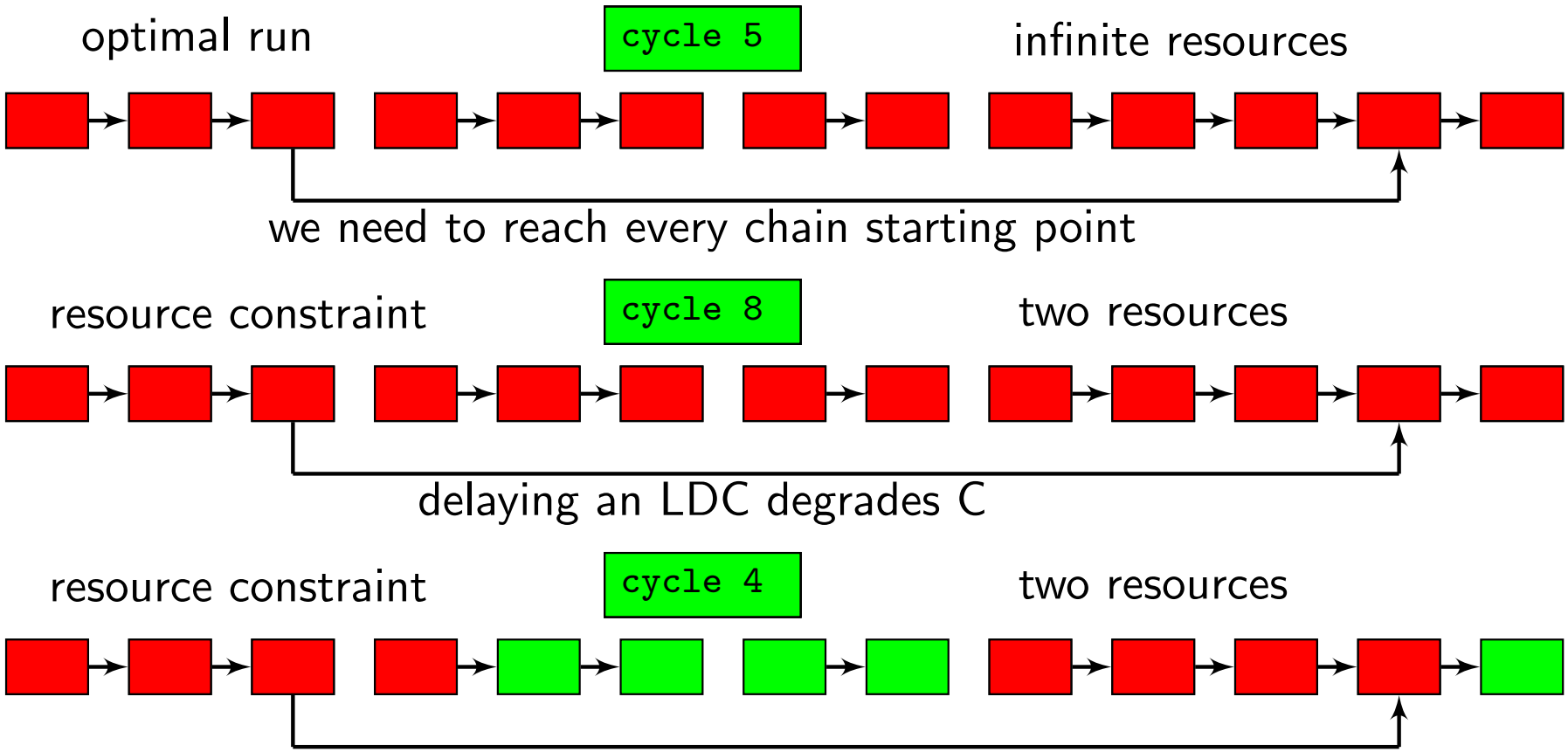
How to capture distant ILP.



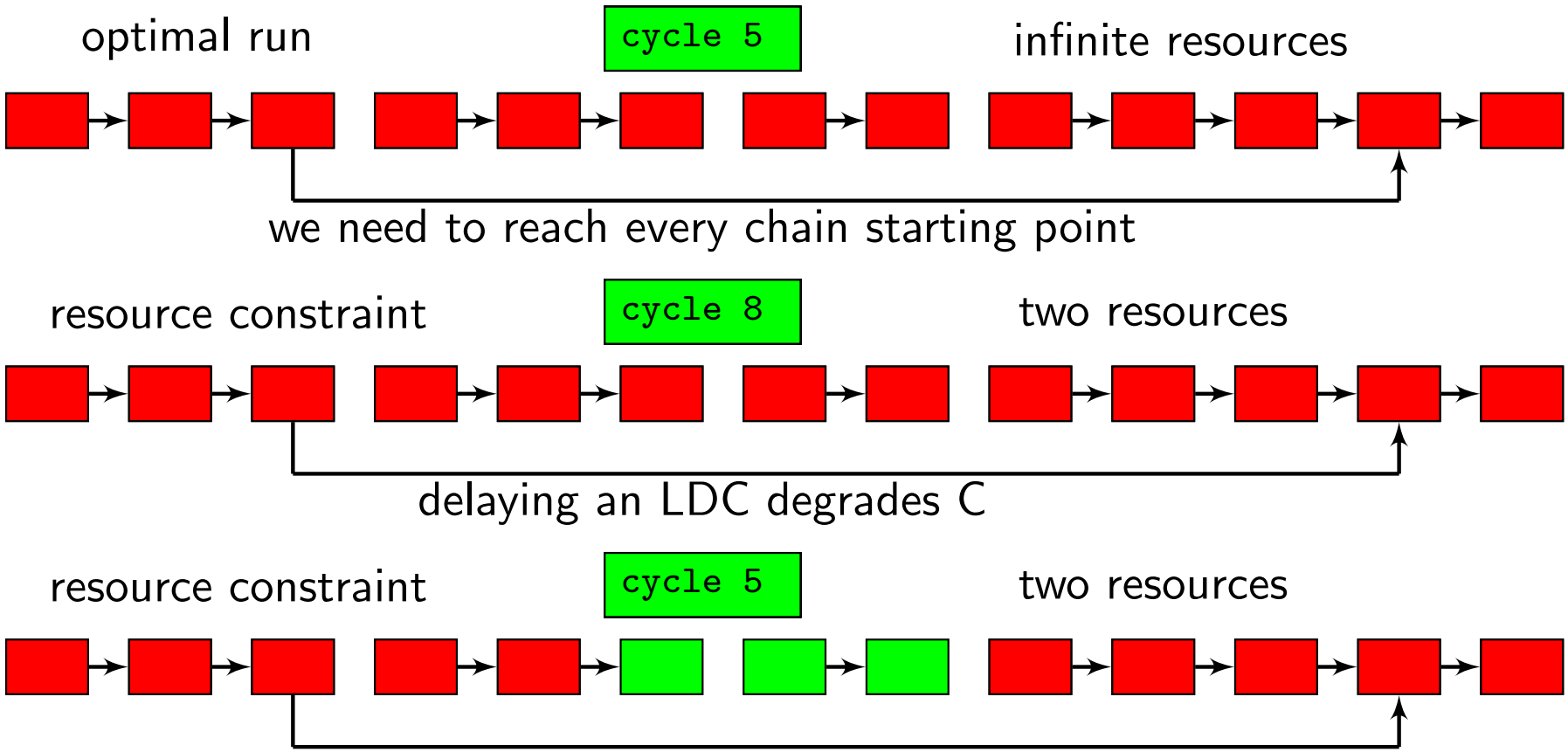
How to capture distant ILP.



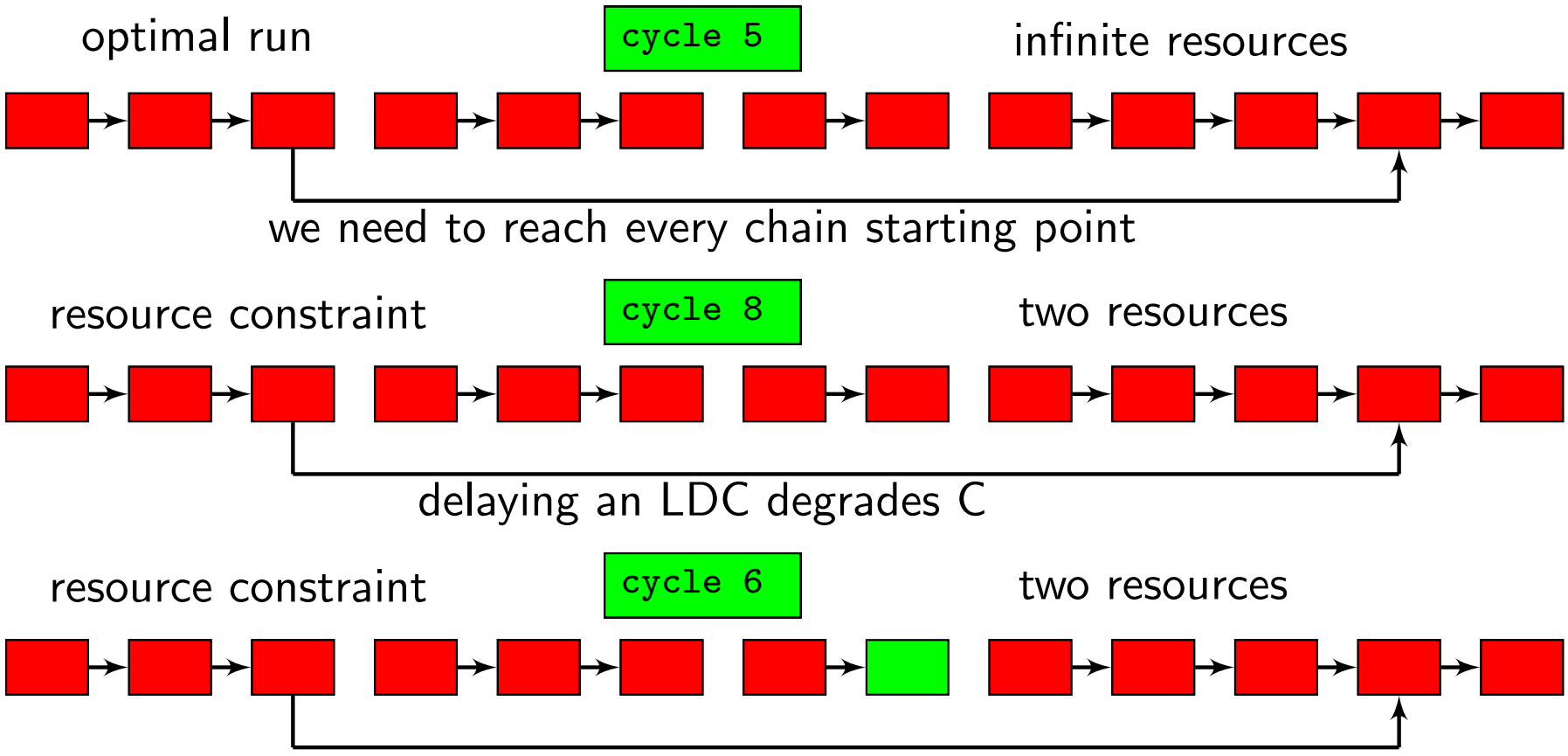
How to capture distant ILP.



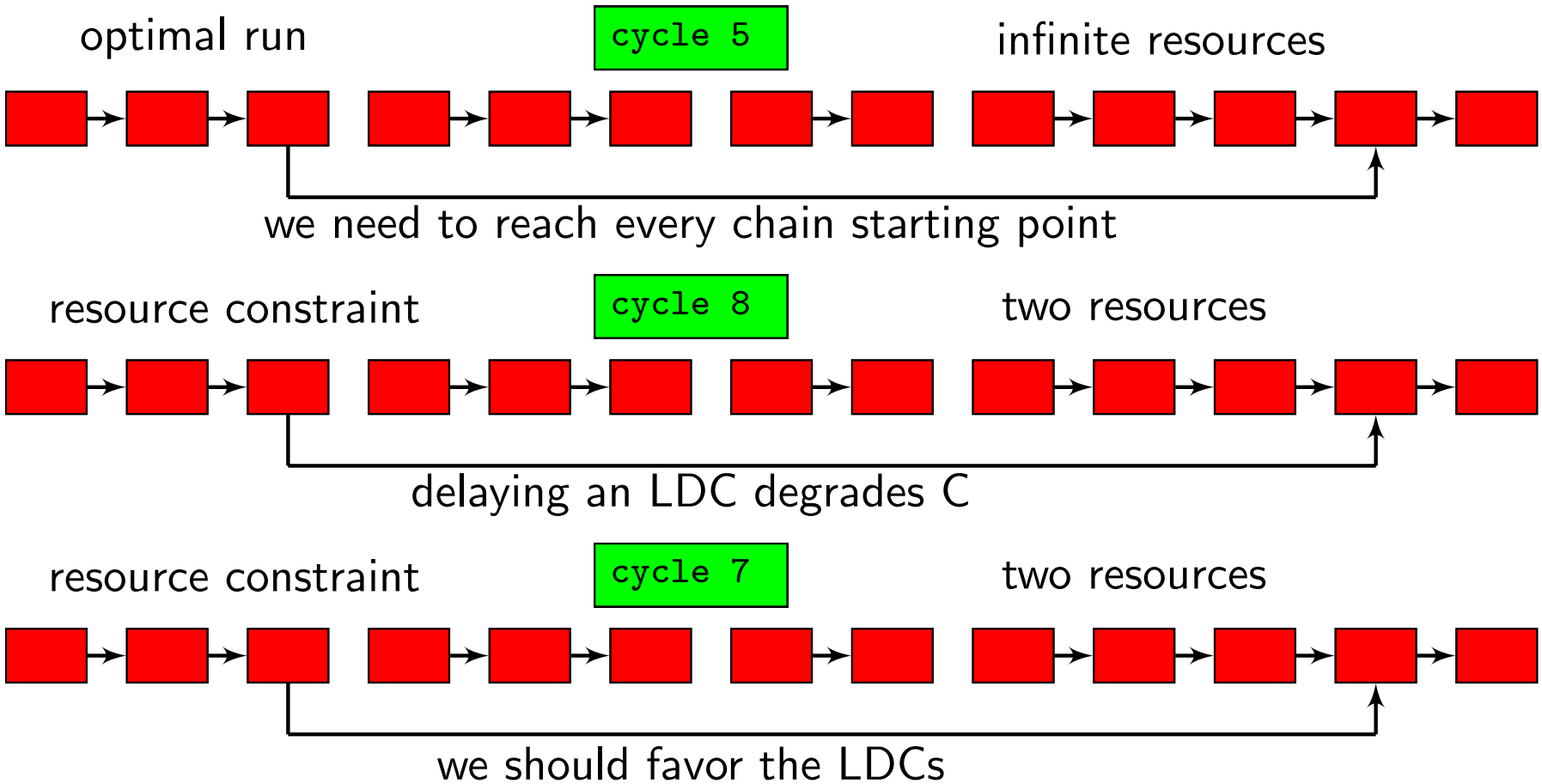
How to capture distant ILP.



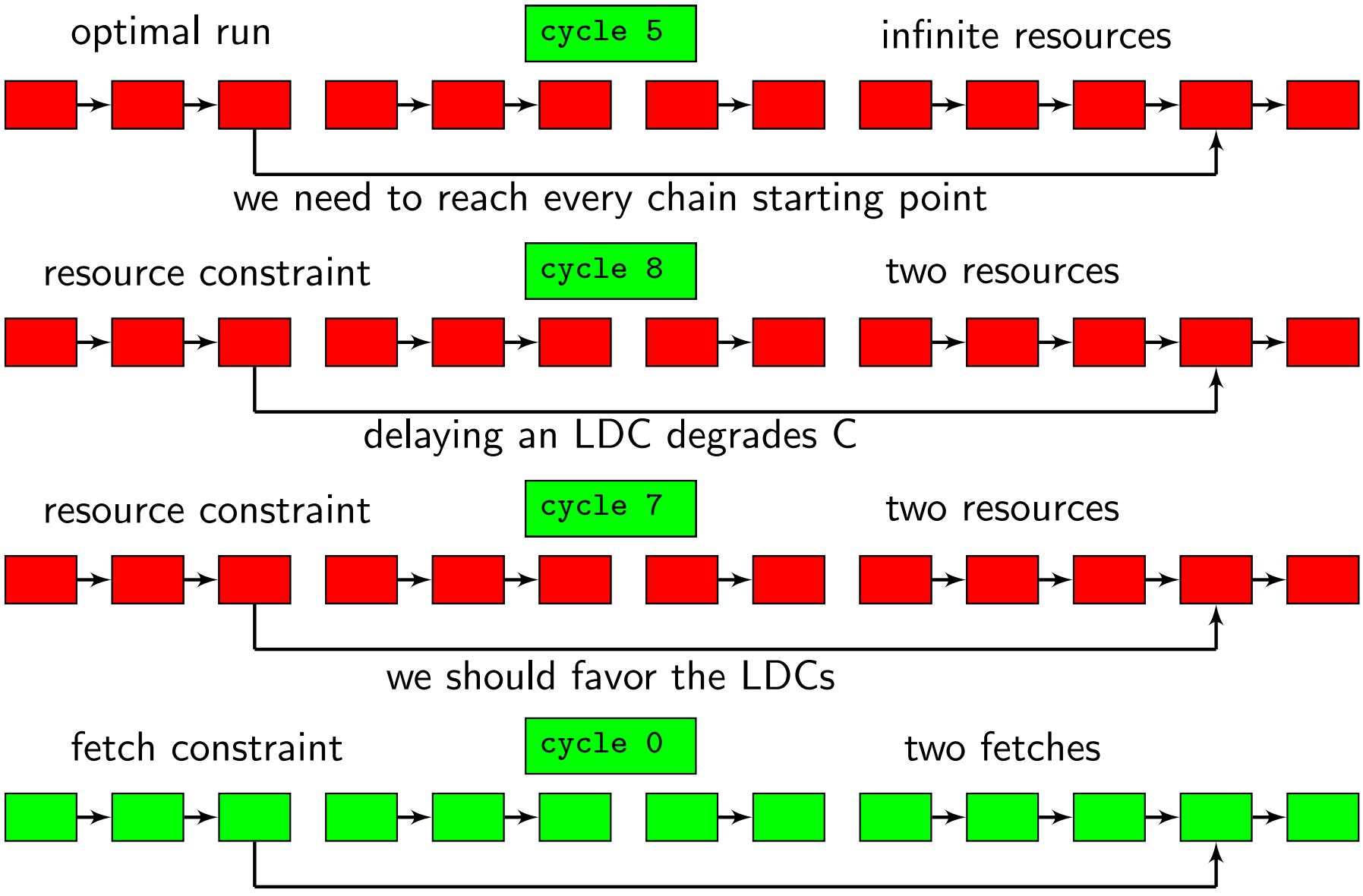
How to capture distant ILP.



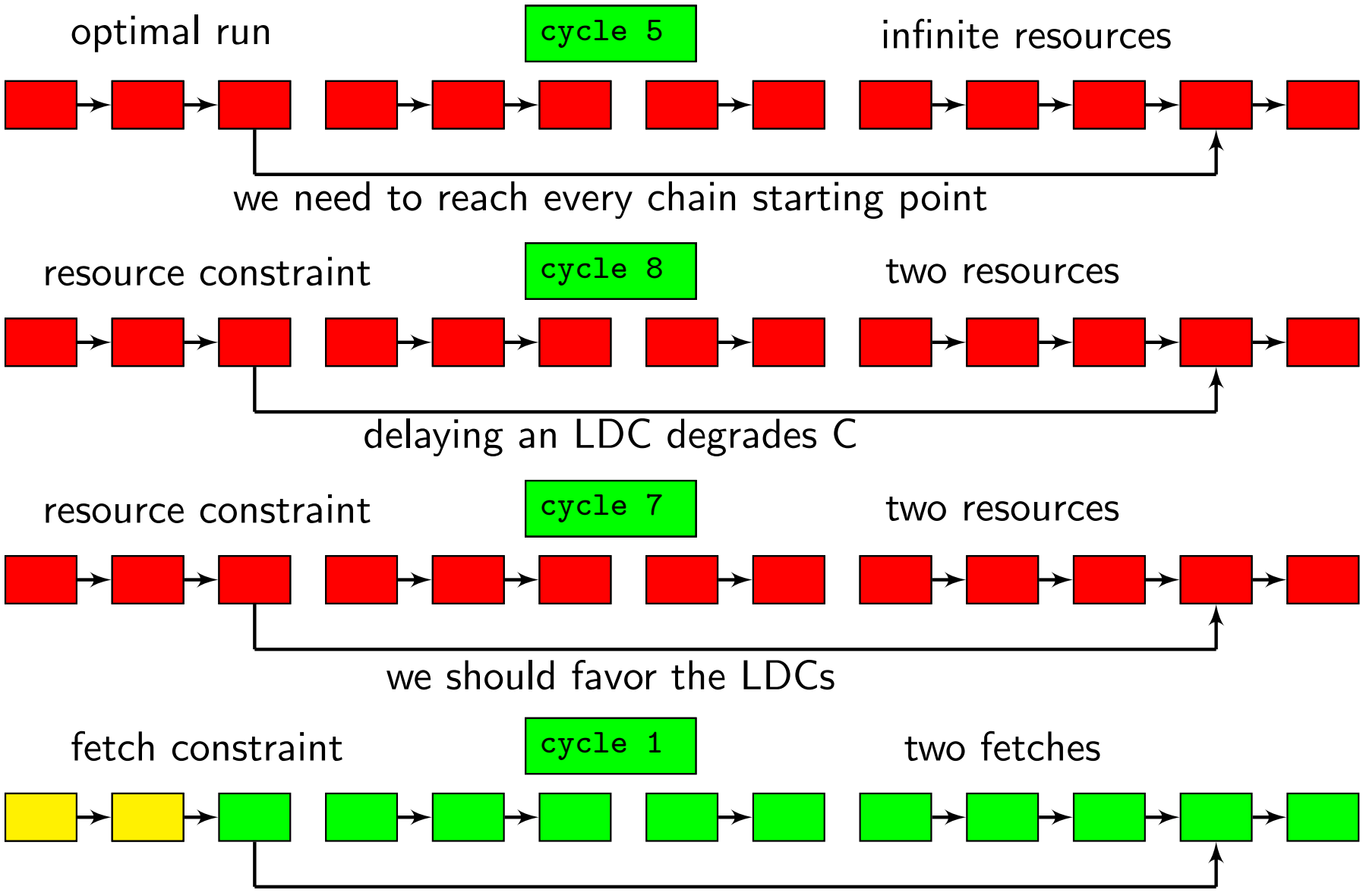
How to capture distant ILP.



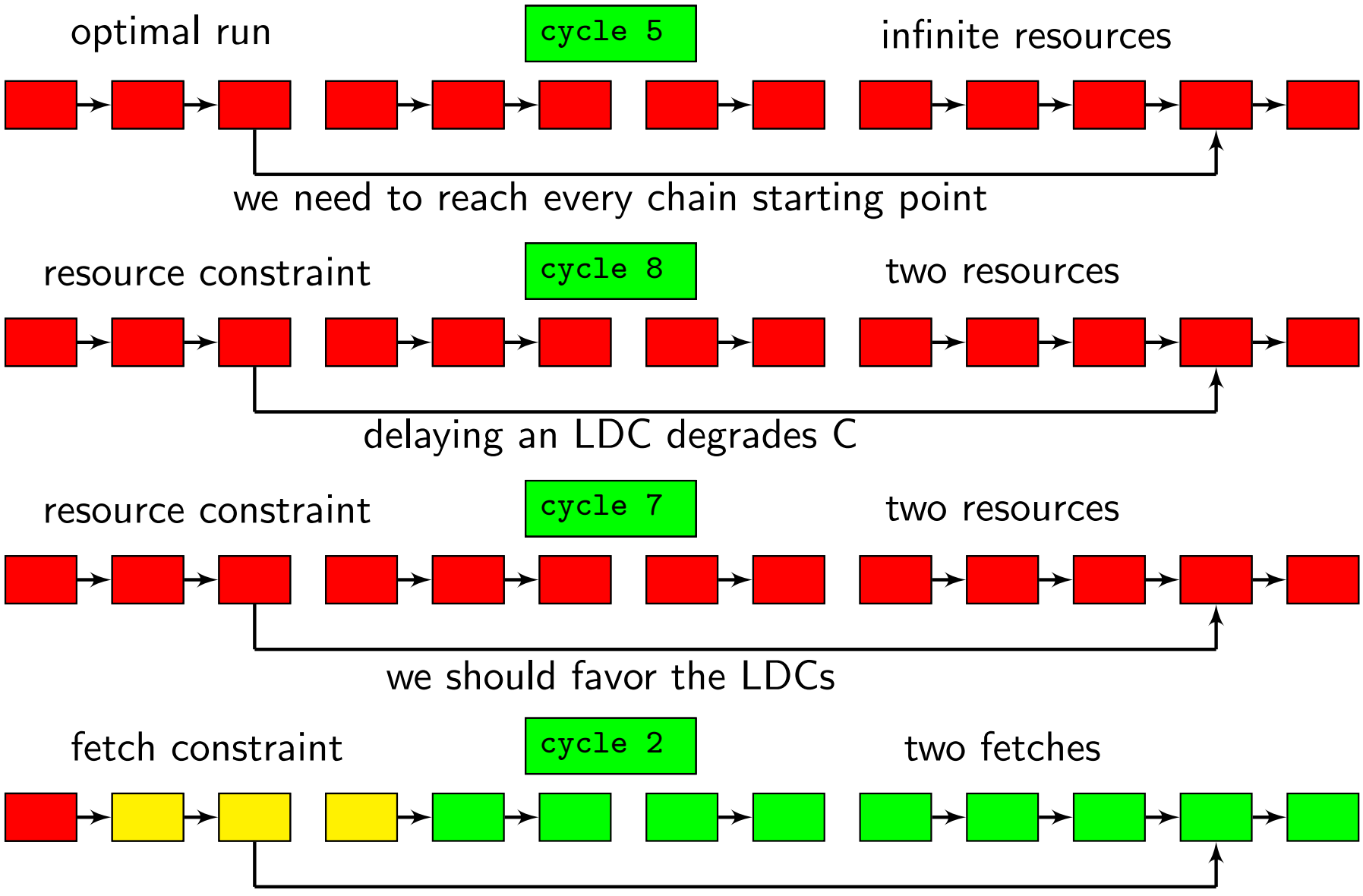
How to capture distant ILP.



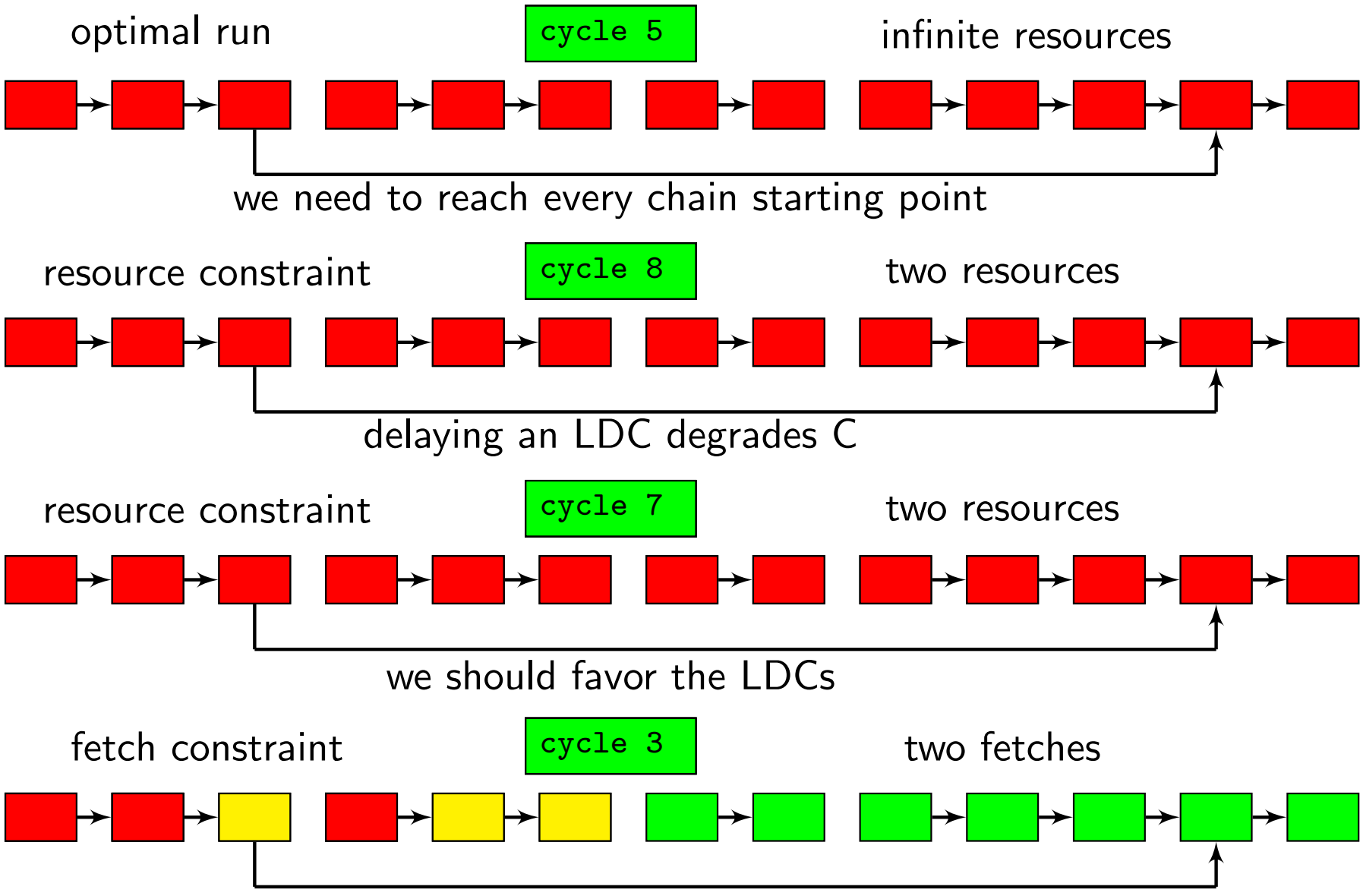
How to capture distant ILP.



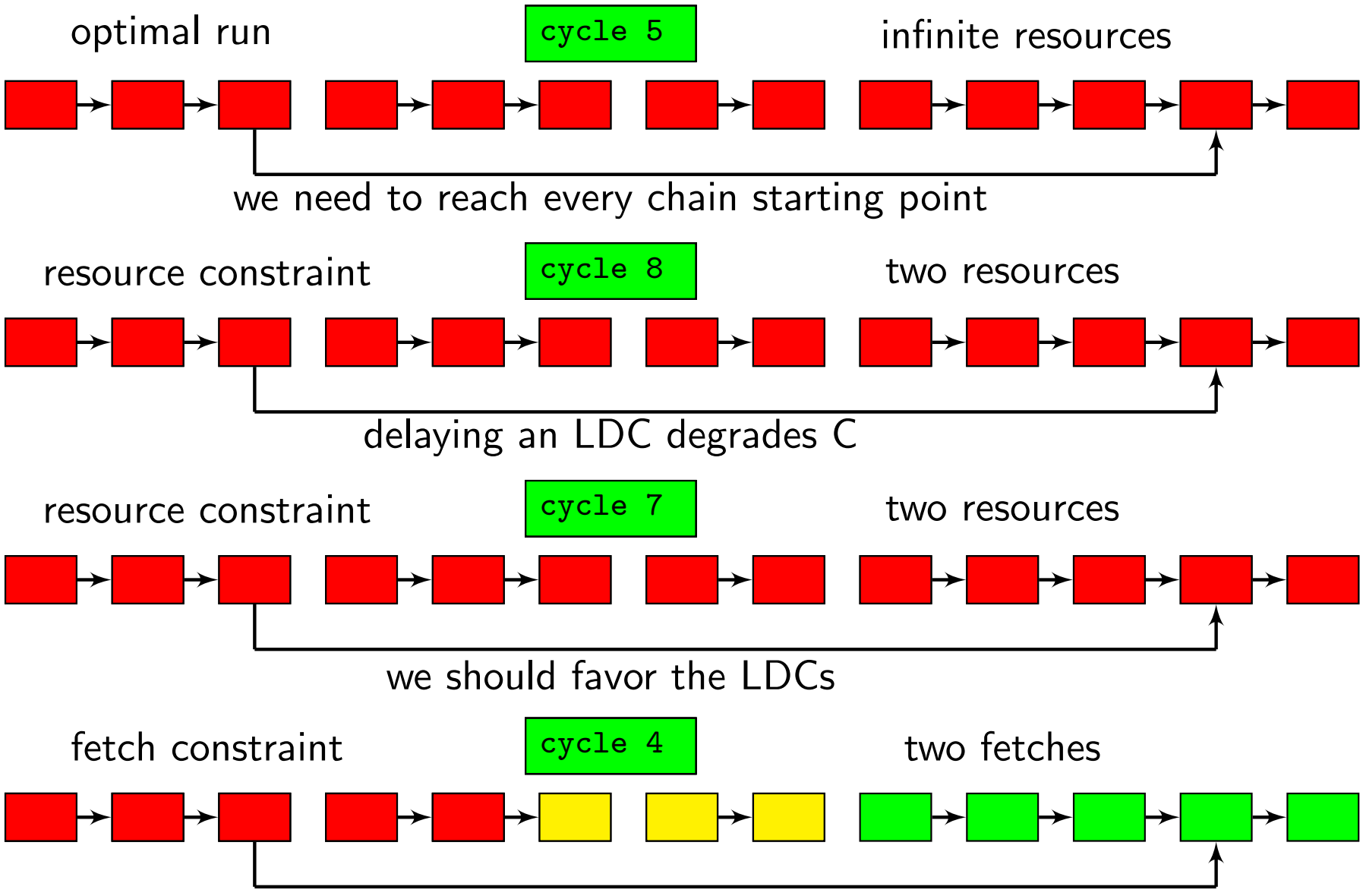
How to capture distant ILP.



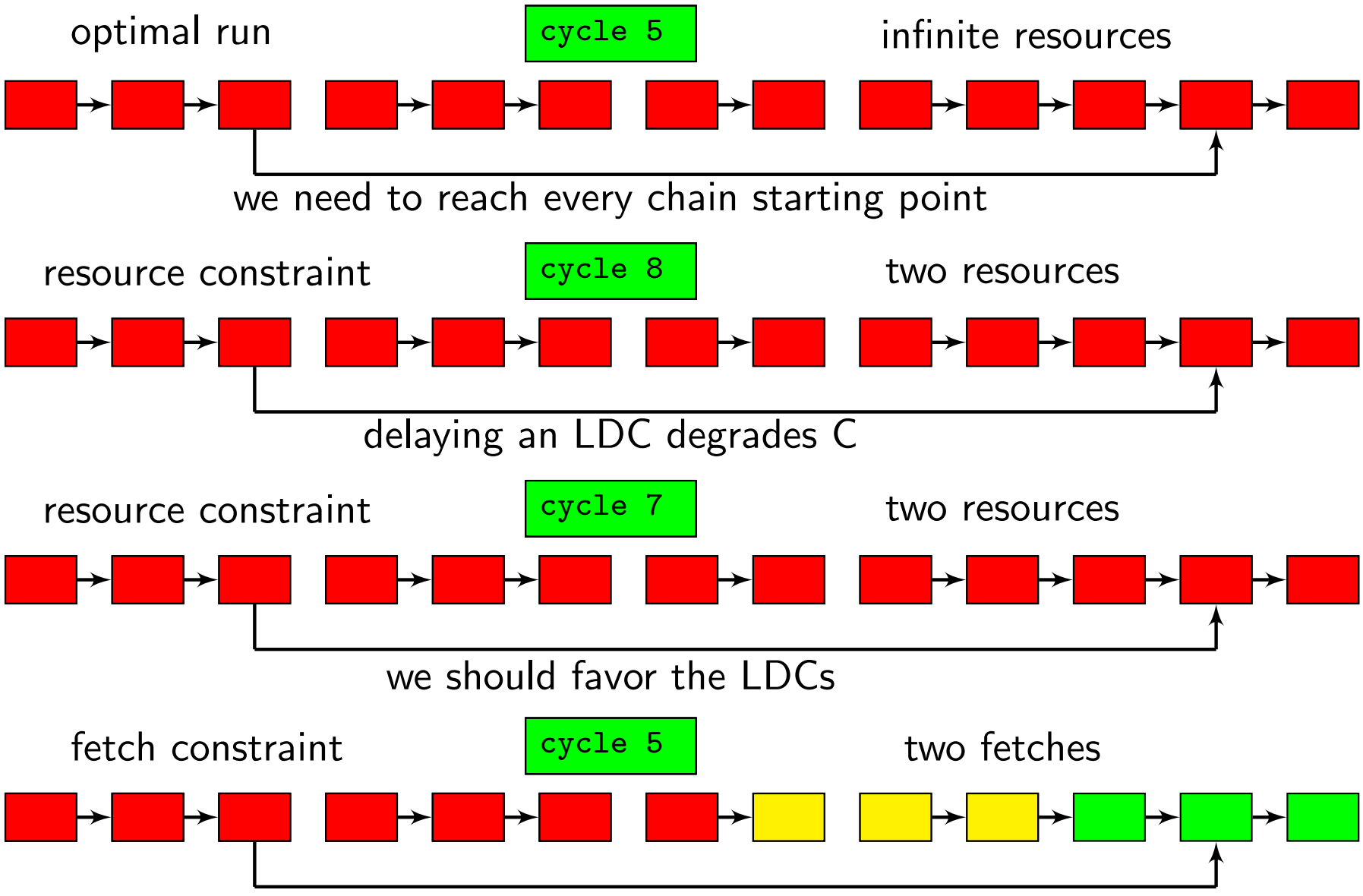
How to capture distant ILP.



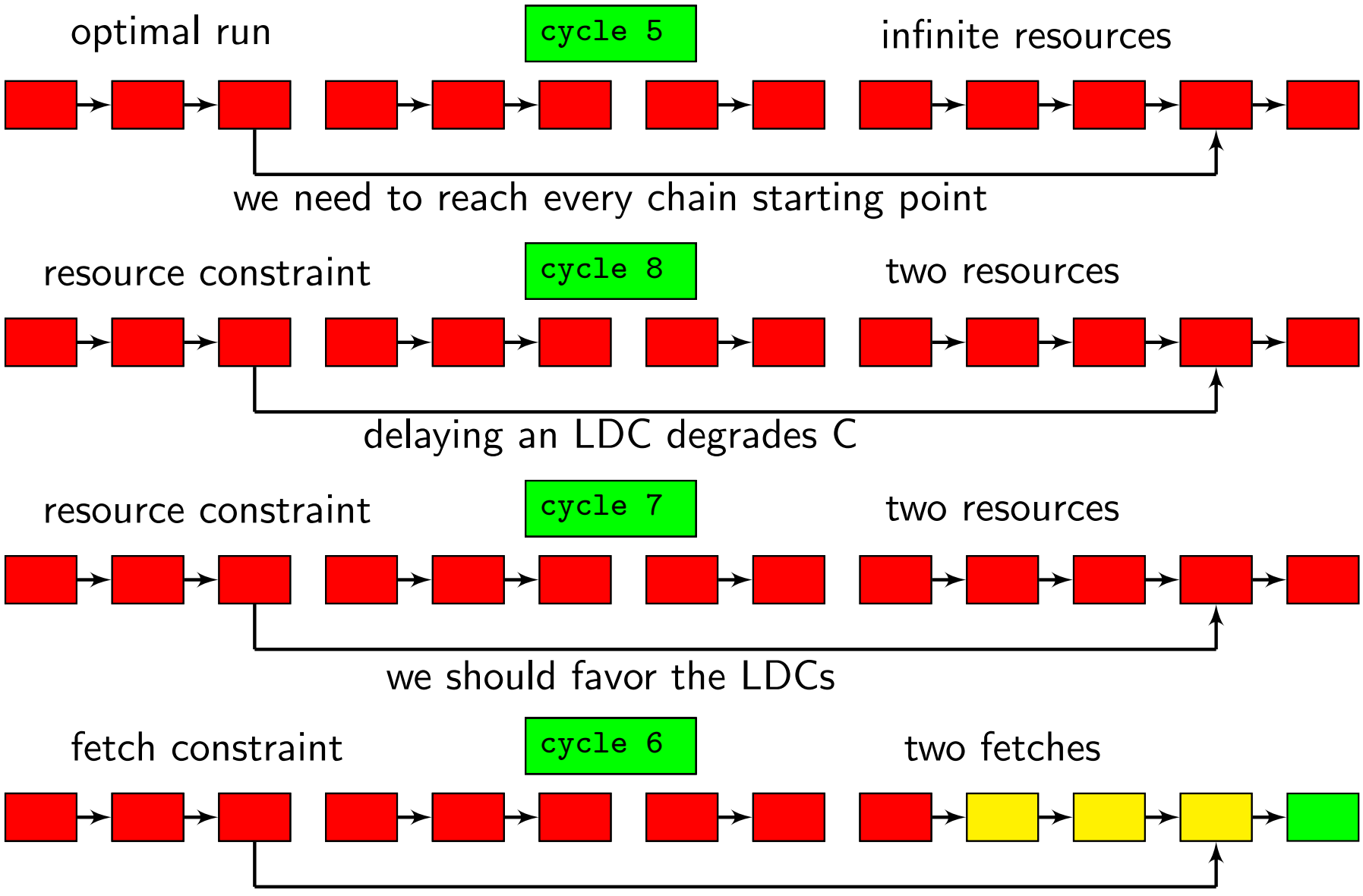
How to capture distant ILP.



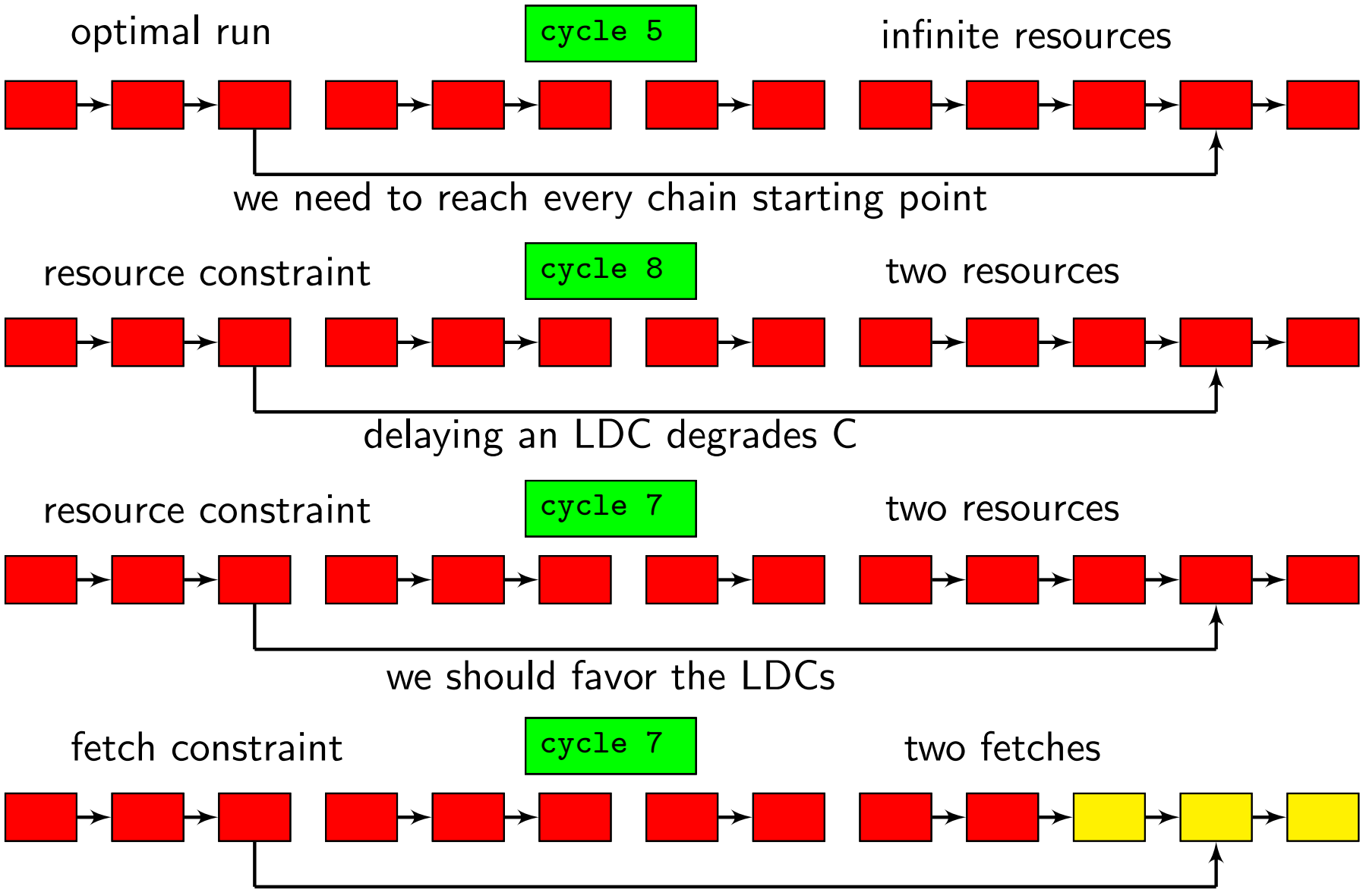
How to capture distant ILP.



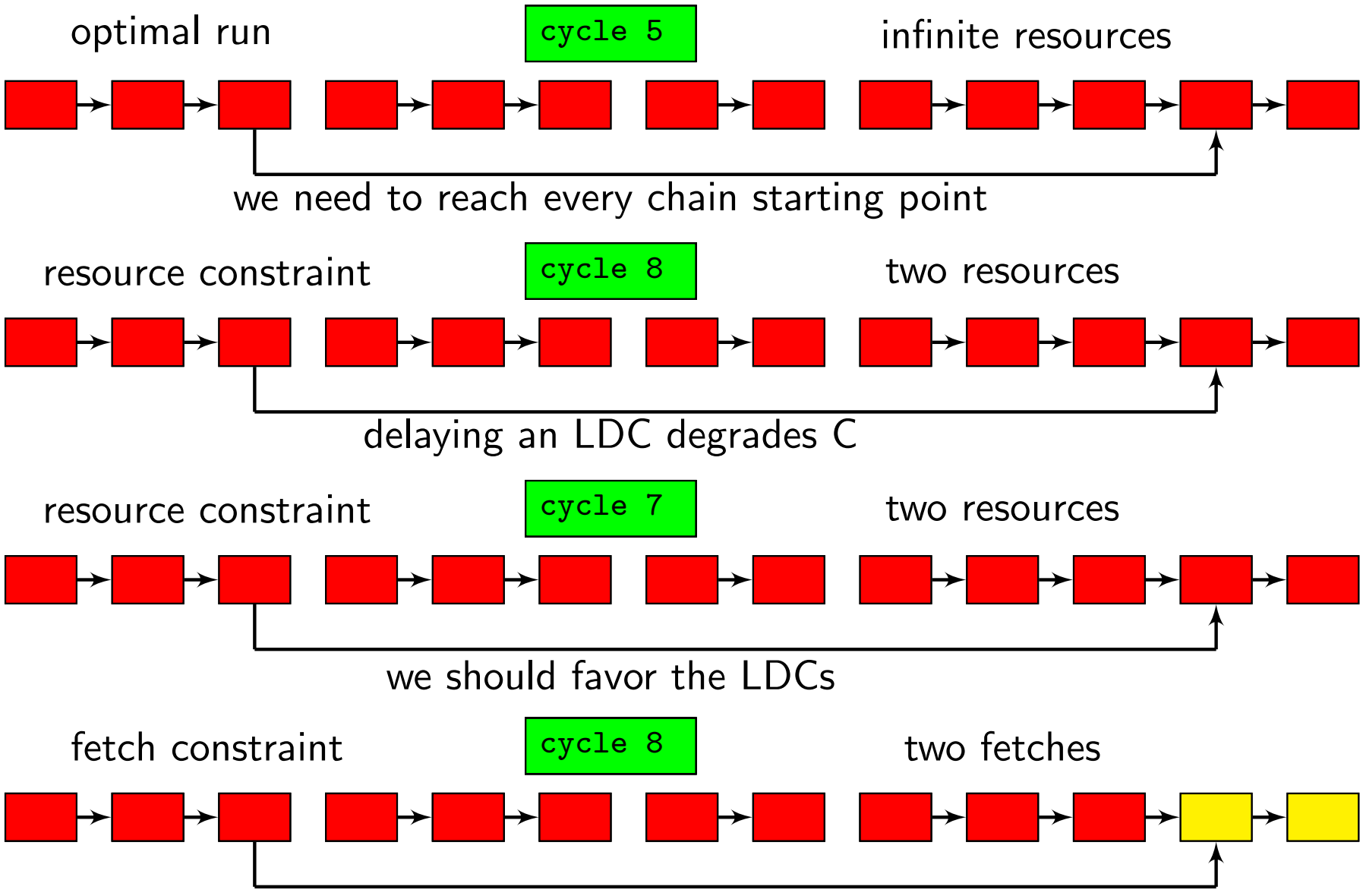
How to capture distant ILP.



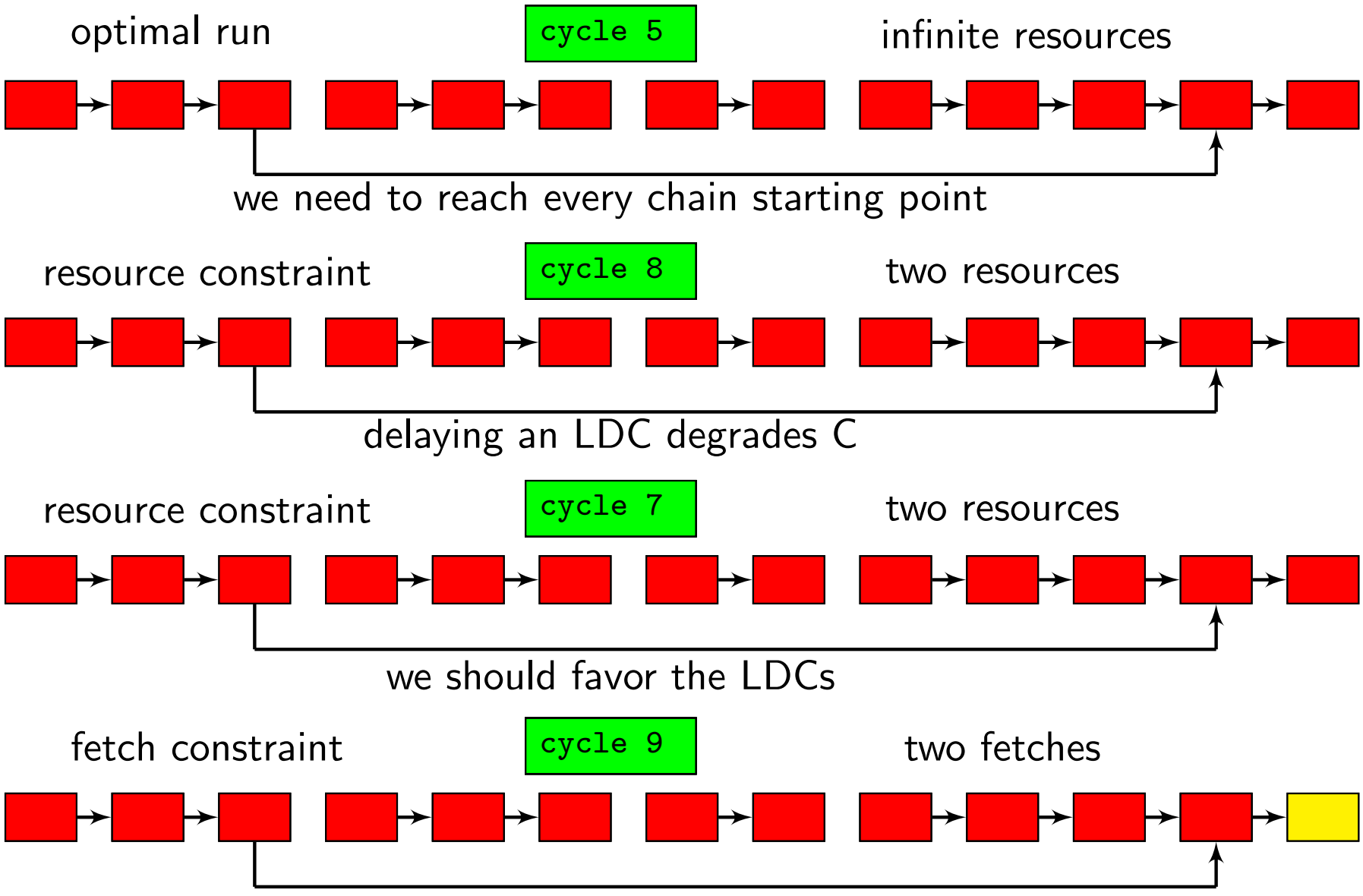
How to capture distant ILP.



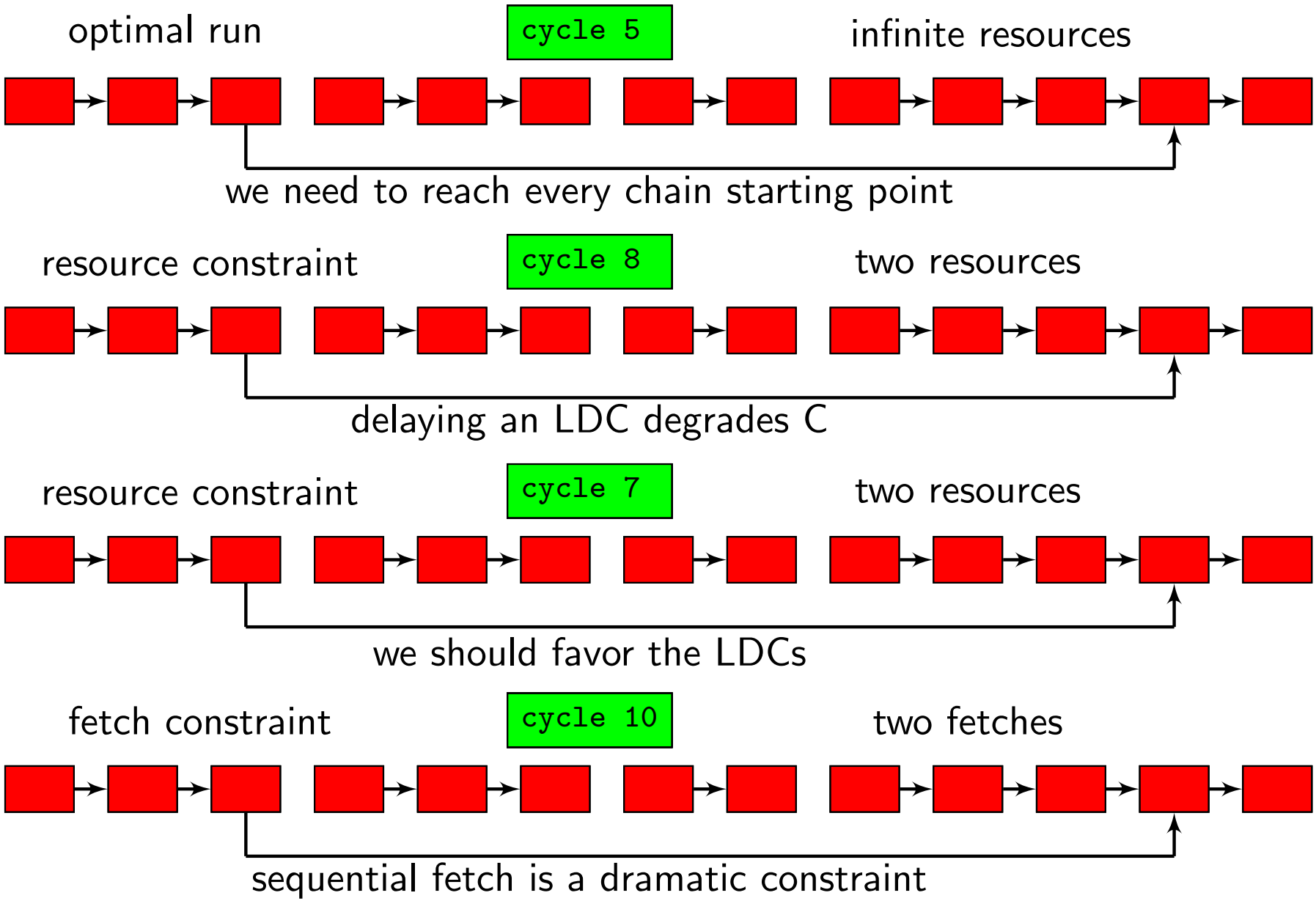
How to capture distant ILP.



How to capture distant ILP.



How to capture distant ILP.



Parallelizing a run.

Parallelizing a run.

- We must reach ASAP the most distant points in the trace to start late LDCs with a minimal delay.

Parallelizing a run.

- We must reach ASAP the most distant points in the trace to start late LDCs with a minimal delay.
- Use the return address to fork fetch at every call instruction.

Parallelizing a run.

- We must reach ASAP the most distant points in the trace to start late LDCs with a minimal delay.
- Use the return address to fork fetch at every call instruction.
- We must rename dynamic memory places.

Parallelizing a run.

- We must reach ASAP the most distant points in the trace to start late LDCs with a minimal delay.
- Use the return address to fork fetch at every call instruction.
- We must rename dynamic memory places.
- Memory renaming is ideal to match producer/consumer dependent store/load pairs and to parallelize independent accesses.

ILP in for loops : control dependency chain.

ILP in for loops : control dependency chain.

- The unrolled loop control builds a dependency chain (i++ ; i<LIMIT ; branch if true).

ILP in for loops : control dependency chain.

- The unrolled loop control builds a dependency chain (i++ ; i<LIMIT ; branch if true).
- Because of the i++ instruction self-dependency, a new iteration starts at best every cycle.

ILP in for loops : control dependency chain.

- The unrolled loop control builds a dependency chain (i++ ; i<LIMIT ; branch if true).
- Because of the i++ instruction self-dependency, a new iteration starts at best every cycle.
- The control chain has length $n + 2$, n being the number of iterations.

ILP in for loops : control dependency chain.

- The unrolled loop control builds a dependency chain (i++ ; i<LIMIT ; branch if true).
- Because of the i++ instruction self-dependency, a new iteration starts at best every cycle.
- The control chain has length $n + 2$, n being the number of iterations.
- The control chain starts its run when :
  - The i variable initial value has been set (favor constants).
  - Fetch point has reached the loop (fast forward fetch for late loops).

ILP in for loops : control dependency chain.

- The unrolled loop control builds a dependency chain (i++ ; i<LIMIT ; branch if true).
- Because of the i++ instruction self-dependency, a new iteration starts at best every cycle.
- The control chain has length $n + 2$, n being the number of iterations.
- The control chain starts its run when :
  - The i variable initial value has been set (favor constants).
  - Fetch point has reached the loop (fast forward fetch for late loops).
- The test and conditional branch in the control are delayed if the LIMIT is not a constant.

ILP in for loops.

```
//ILP = 3
for (i=0; i<1024; i++){     //from cycle 1 to 1024, produce i
   ...                      //from cycle 2 to 1025, test i
}                           //from cycle 3 to 1026, cond branch
```

ILP in for loops.

```
//ILP = 3
for (i=0; i<1024; i++){      //from cycle 1 to 1024, produce i
   ...                       //from cycle 2 to 1025, test i
}                            //from cycle 3 to 1026, cond branch

//ILP = 3n/(p+1) <3
//produce n at cycle p-1>n
for (i=0; i<n; i++){         //from cycle 1 to n, produce i
   ...                       //at cycle p, 1024 tests of i
}                            //at cycle p+1, 1024 cond branch
```

ILP in for loops.

```
//ILP = 3
for (i=0; i<1024; i++){     //from cycle 1 to 1024, produce i
   ...                      //from cycle 2 to 1025, test i
}                           //from cycle 3 to 1026, cond branch

//ILP = 3n/(p+1) <3
//produce n at cycle p-1>n
for (i=0; i<n; i++){        //from cycle 1 to n, produce i
   ...                      //at cycle p, 1024 tests of i
}                           //at cycle p+1, 1024 cond branch

//ILP = 3n/(p+1) <3
//produce j at cycle q-1>n
//produce n at cycle p-1>q+1024
for (i=j; i<n; i++){        //from cycle q to q+1024, produce i
   ...                      //at cycle p, 1024 tests of i
}                           //at cycle p+1, 1024 cond branch
```

ILP in for loops.

```
//ILP = 3
for (i=0; i<1024; i++){     //from cycle 1 to 1024, produce i
  ...                       //from cycle 2 to 1025, test i
}                           //from cycle 3 to 1026, cond branch

//ILP = 3n/(p+1) <3
//produce n at cycle p-1>n
for (i=0; i<n; i++){        //from cycle 1 to n, produce i
  ...                       //at cycle p, 1024 tests of i
}                           //at cycle p+1, 1024 cond branch

//ILP = 3n/(p+1) <3
//produce j at cycle q-1>n
//produce n at cycle p-1>q+1024
for (i=j; i<n; i++){        //from cycle q to q+1024, produce i
  ...                       //at cycle p, 1024 tests of i
}                           //at cycle p+1, 1024 cond branch

//ILP = 3*1024
//nested loops are vectorized
for (i=0; i<1024; i++){     //from cycle 1 to 1024, produce i
  for (j=0; j<1024; j++){   //from cycle 1 to 1024, produce 1024 j
    ...                     //from cycle 2 to 1024, test 1024 j
  }                         //from cycle 3 to 1026, 1024 cond branch
  ...                       //from cycle 2 to 1024, test i
}                           //from cycle 3 to 1026, cond branch
```

Loops parallelism.

Loops parallelism.

- Nest long loops, even artificially, with an internal loop constant start value.

Loops parallelism.

- Nest long loops, even artificially, with an internal loop constant start value.
- Favor constant start values and constant increment.

Loops parallelism.

- Nest long loops, even artificially, with an internal loop constant start value.
- Favor constant start values and constant increment.
- The loop body should be built to allow a maximum overlapping of successive iterations (i.e. one cycle shift).

Loops parallelism.

- Nest long loops, even artificially, with an internal loop constant start value.
- Favor constant start values and constant increment.
- The loop body should be built to allow a maximum overlapping of successive iterations (i.e. one cycle shift).
- The body ILP is bounded by the body instruction number. To increase the body ILP, unroll the loop.

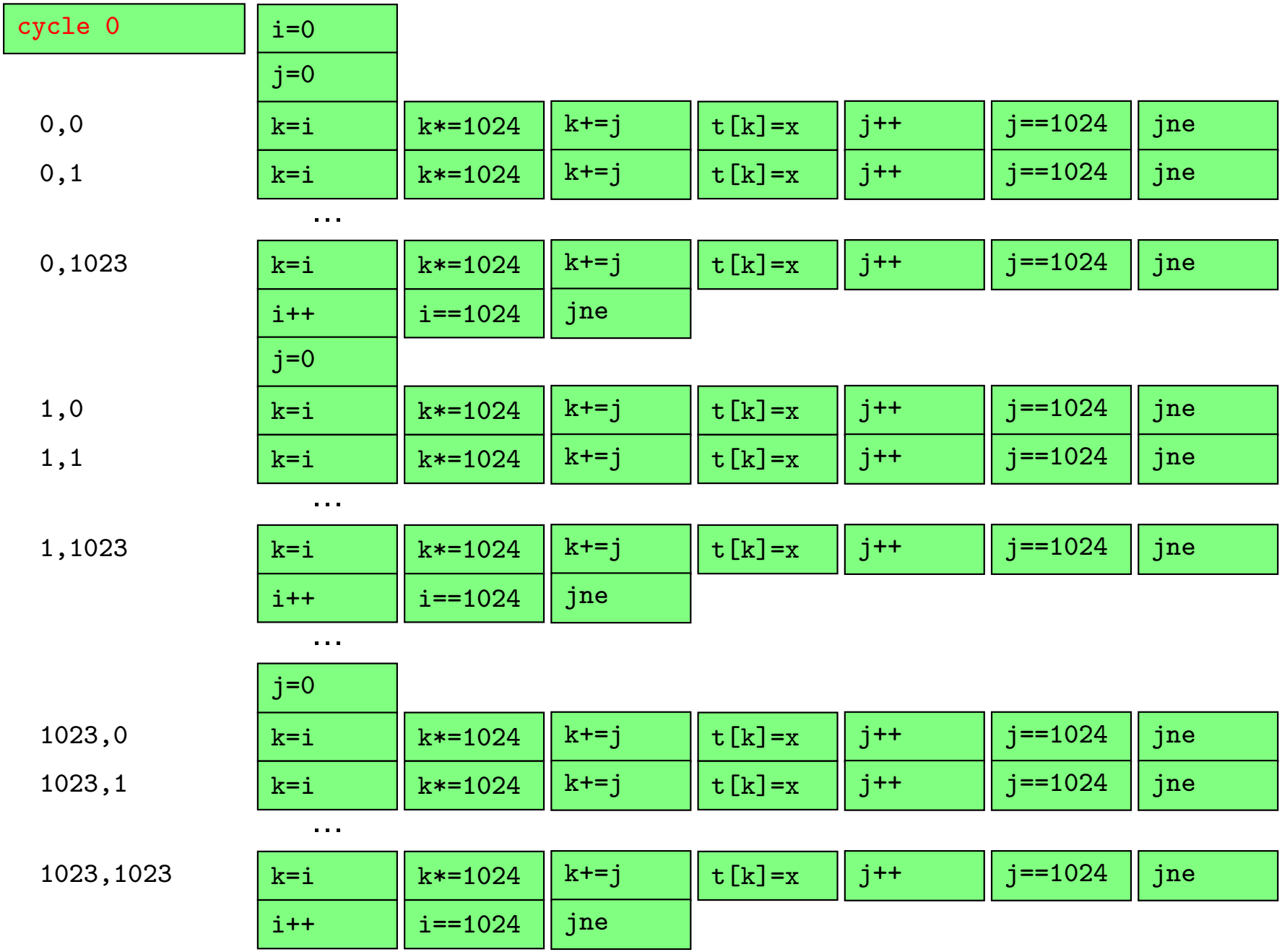
Loops parallelism.

- Nest long loops, even artificially, with an internal loop constant start value.
- Favor constant start values and constant increment.
- The loop body should be built to allow a maximum overlapping of successive iterations (i.e. one cycle shift).
- The body ILP is bounded by the body instruction number. To increase the body ILP, unroll the loop.
- Loops controls are often independent and can be run in parallel.

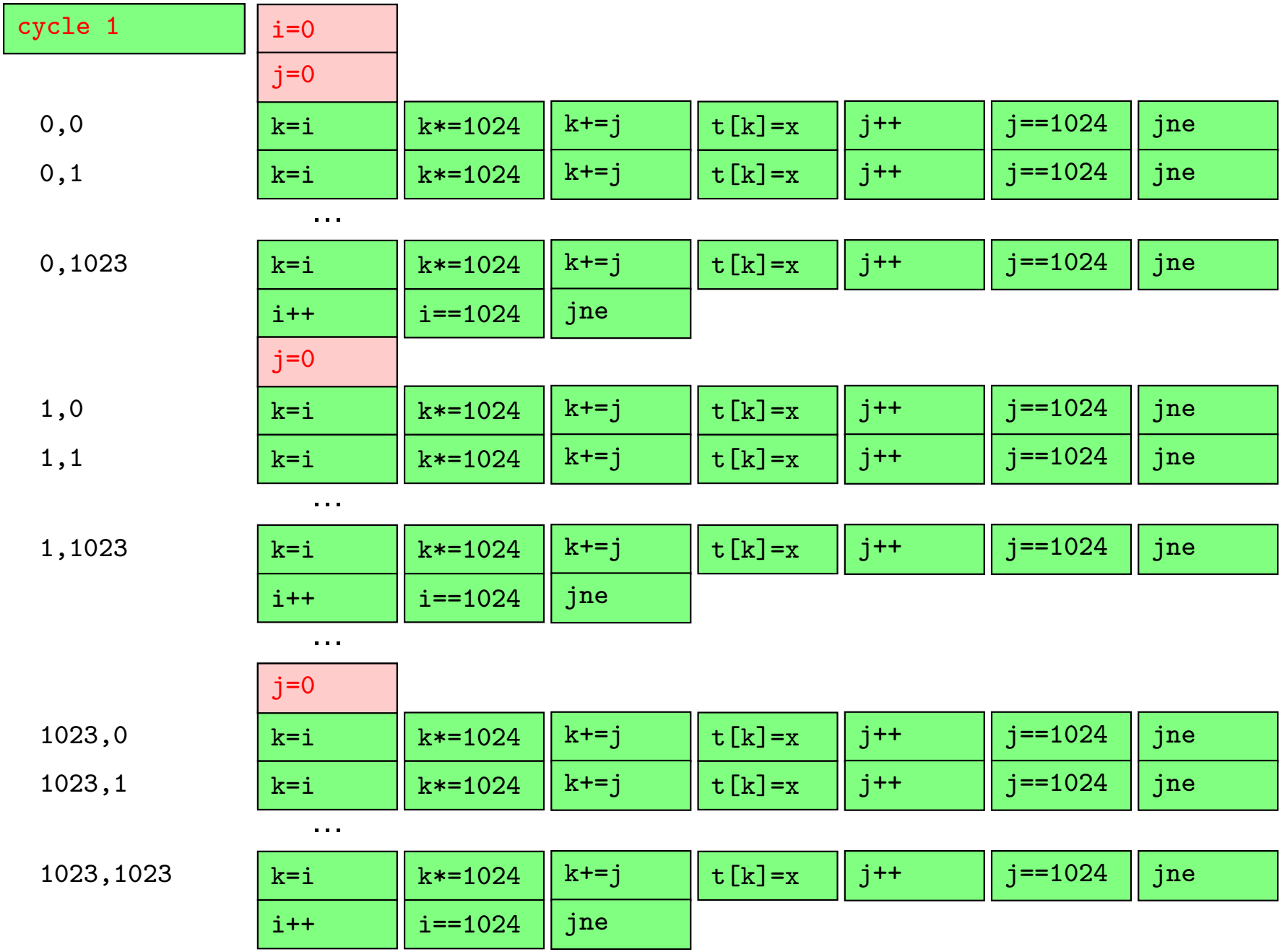
Nested loops.

```
        ...                 ; x computed in al      x=f(...);
        movl $0, %esi       ; i=0                   for (i=0;i<1024;i++)
.L5:    ; external loop start                         for (j=0;j<1024;j++)
        movl $0, %edi       ; j=0                       t[i][j]=x;
.L4:    ; internal loop start
        movl %esi, %ecx     ; ecx=i
        sall $10, %ecx      ; ecx=ecx*1024
        addl %edi, %ecx     ; ecx=ecx+j
        movb %al, t(%ecx)   ; t[i][j]=x
        addl $1, %edi       ; j++
        cmpl $1024, %edi    ; (j == 1024)
        jne  .L4            ; if (j!=1024) goto .L4
        ; internal loop end
        addl $1, %esi       ; i++
        cmpl $1024, %esi    ; (i == 1024)
        jne  .L5            ; if (i!=1024) goto .L5
        ; external loop end
```

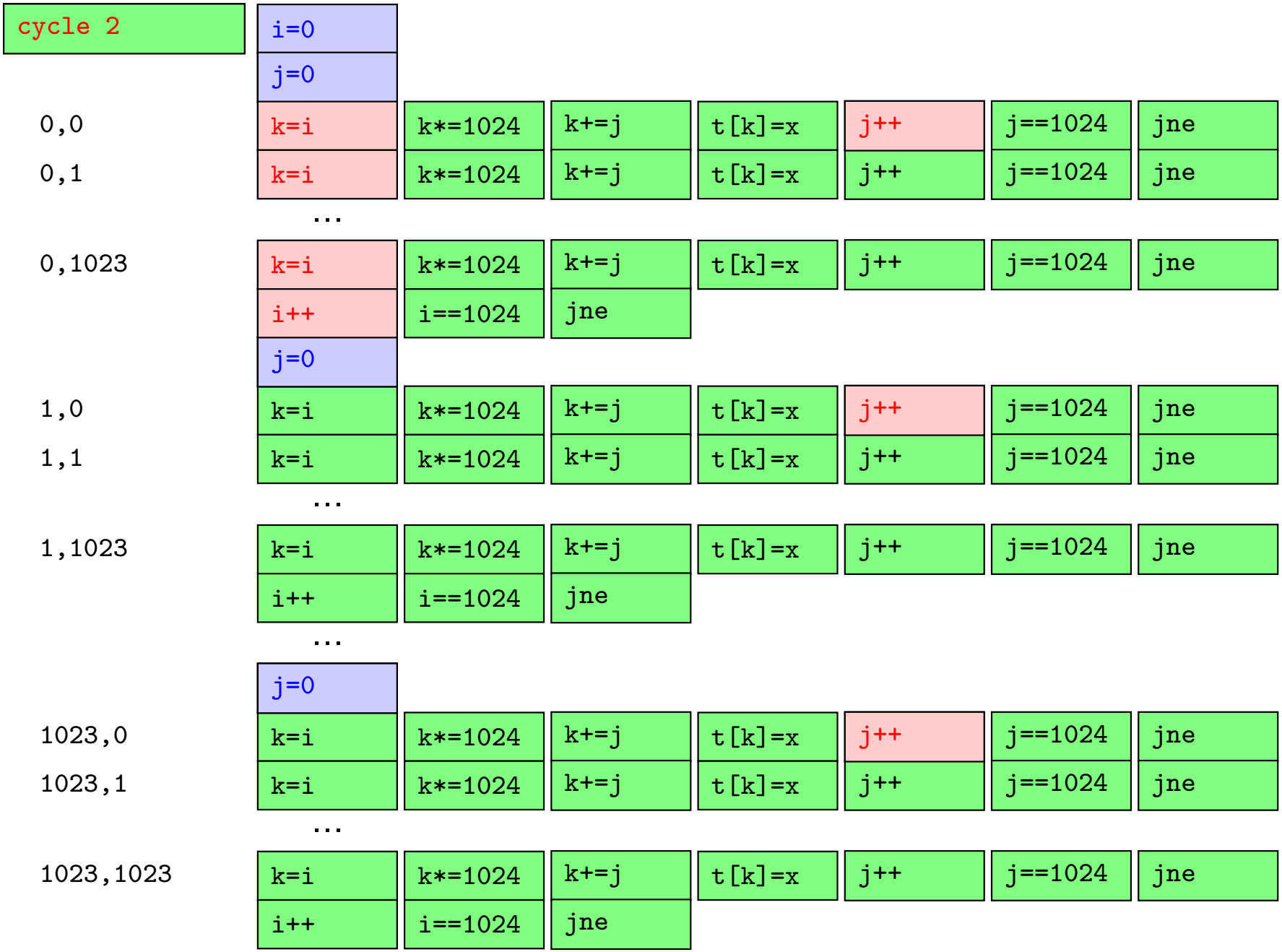
Nested loops.



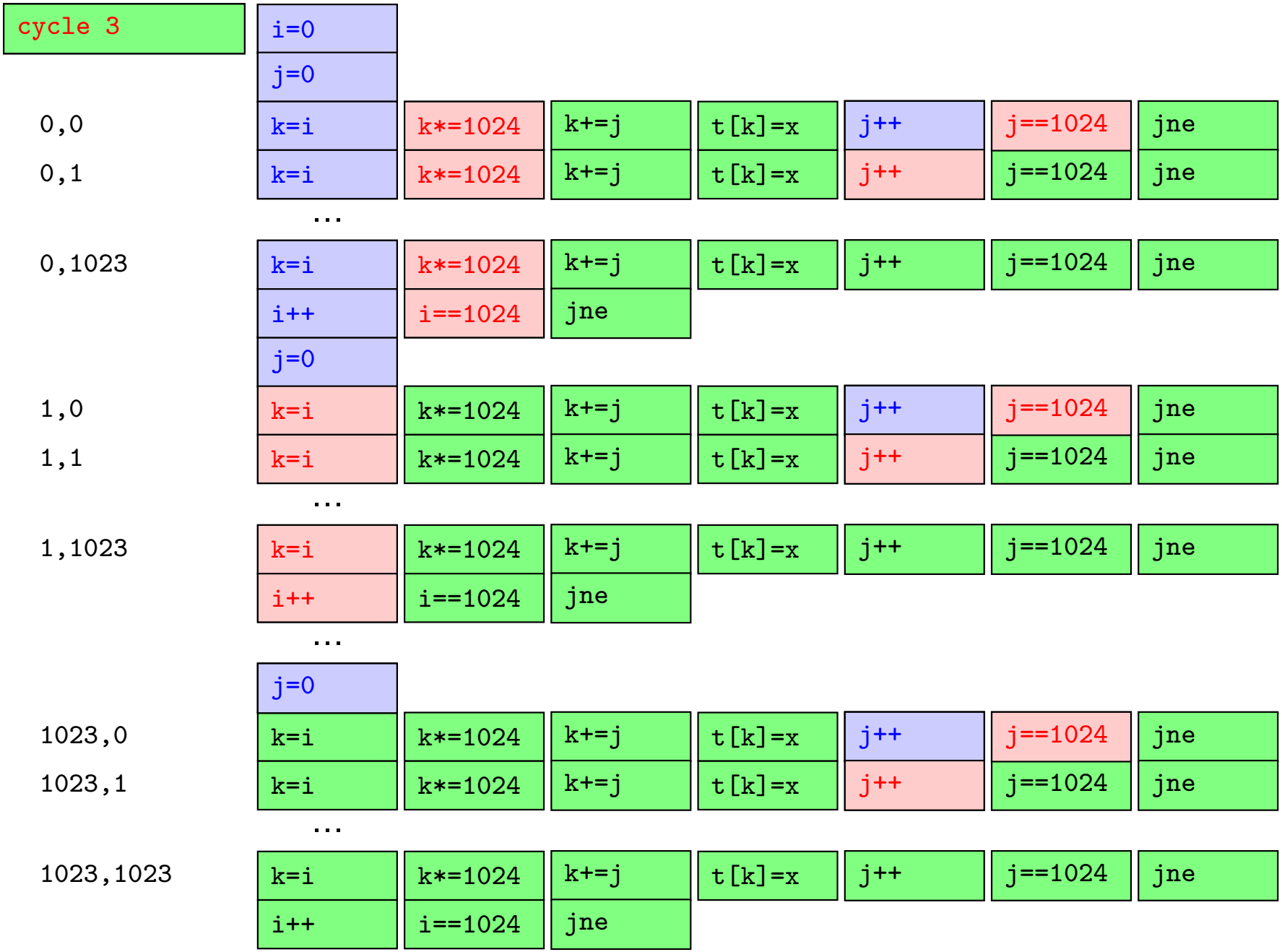
Nested loops.



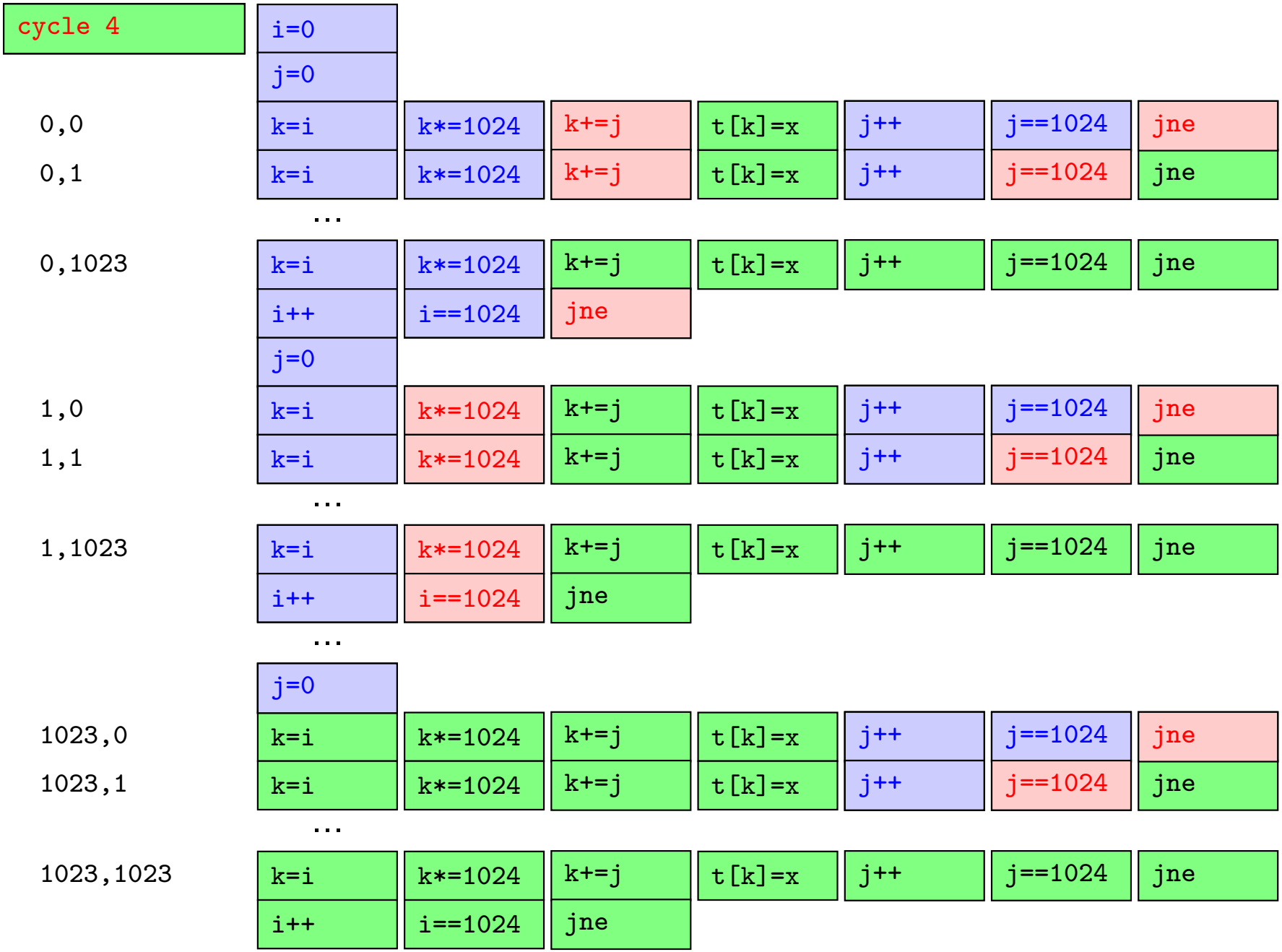
Nested loops.



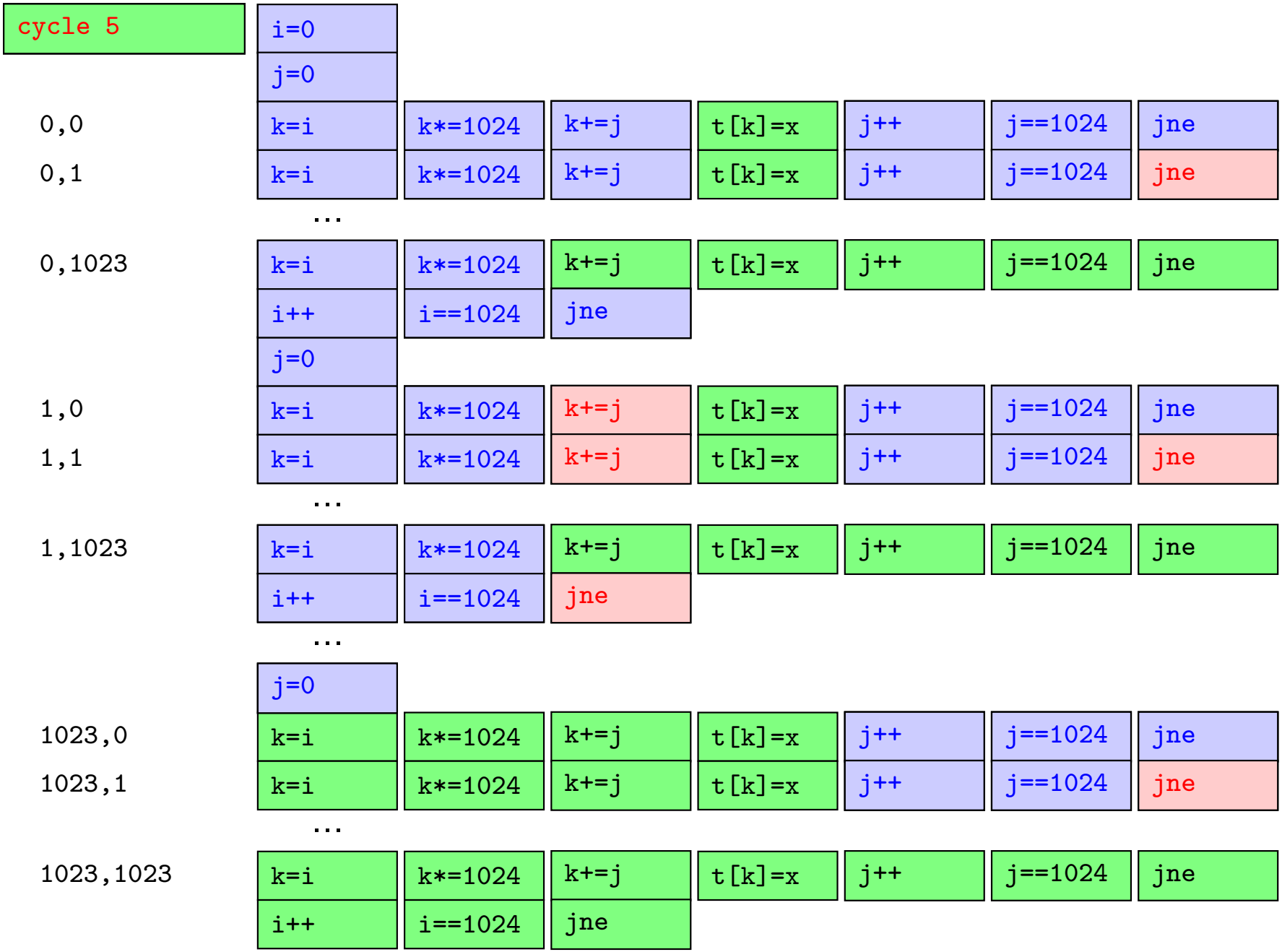
Nested loops.



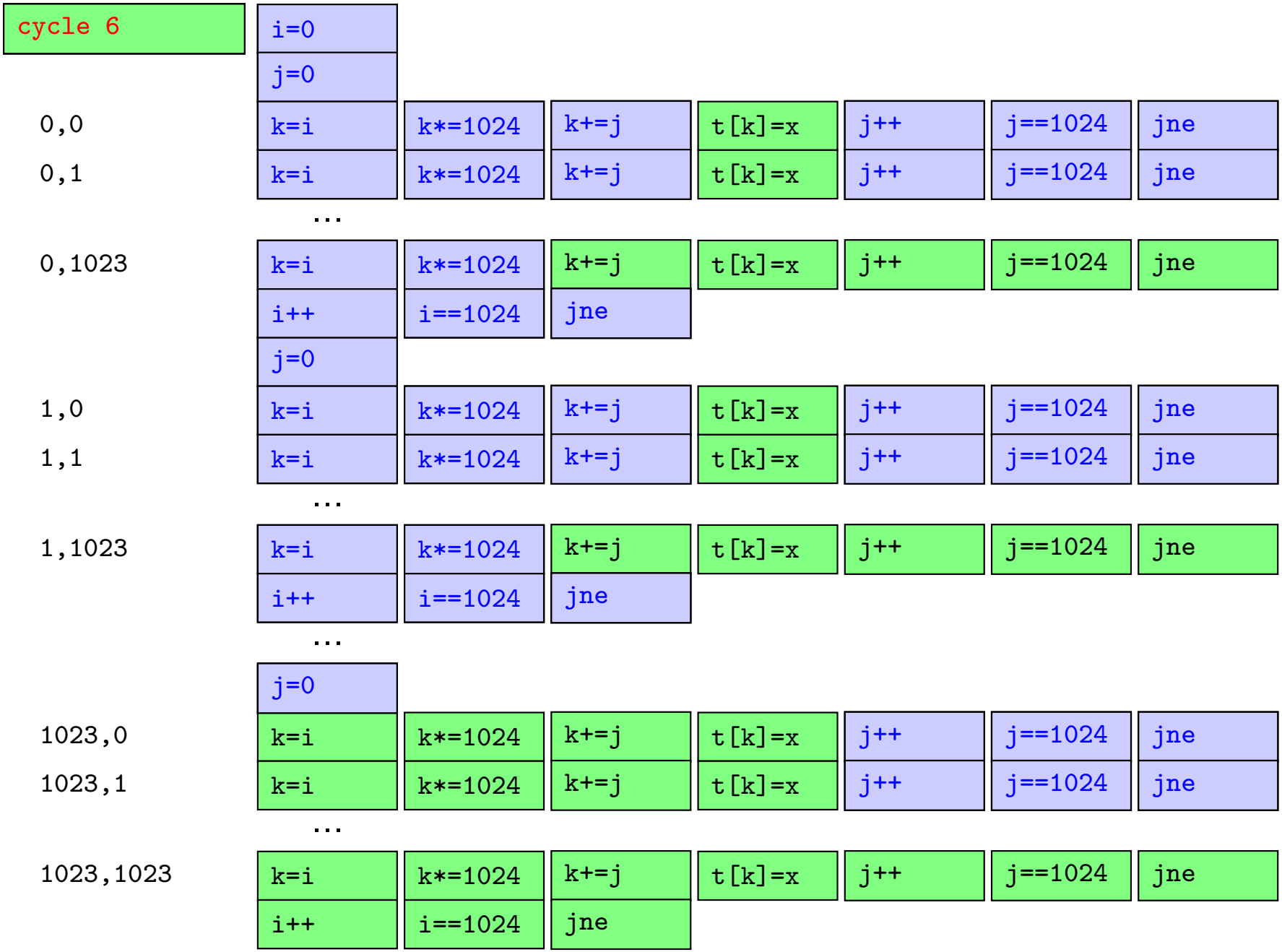
Nested loops.



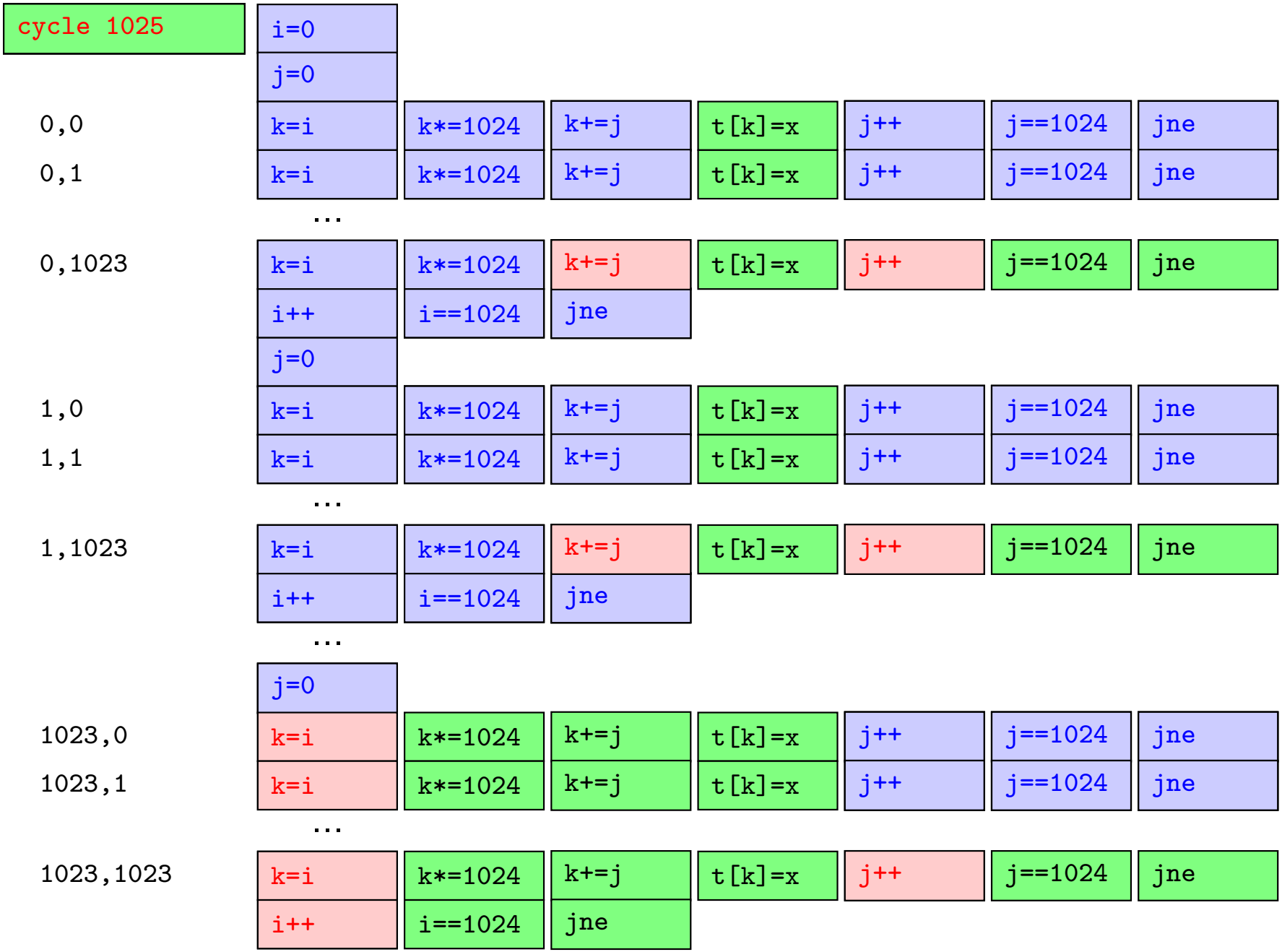
Nested loops.



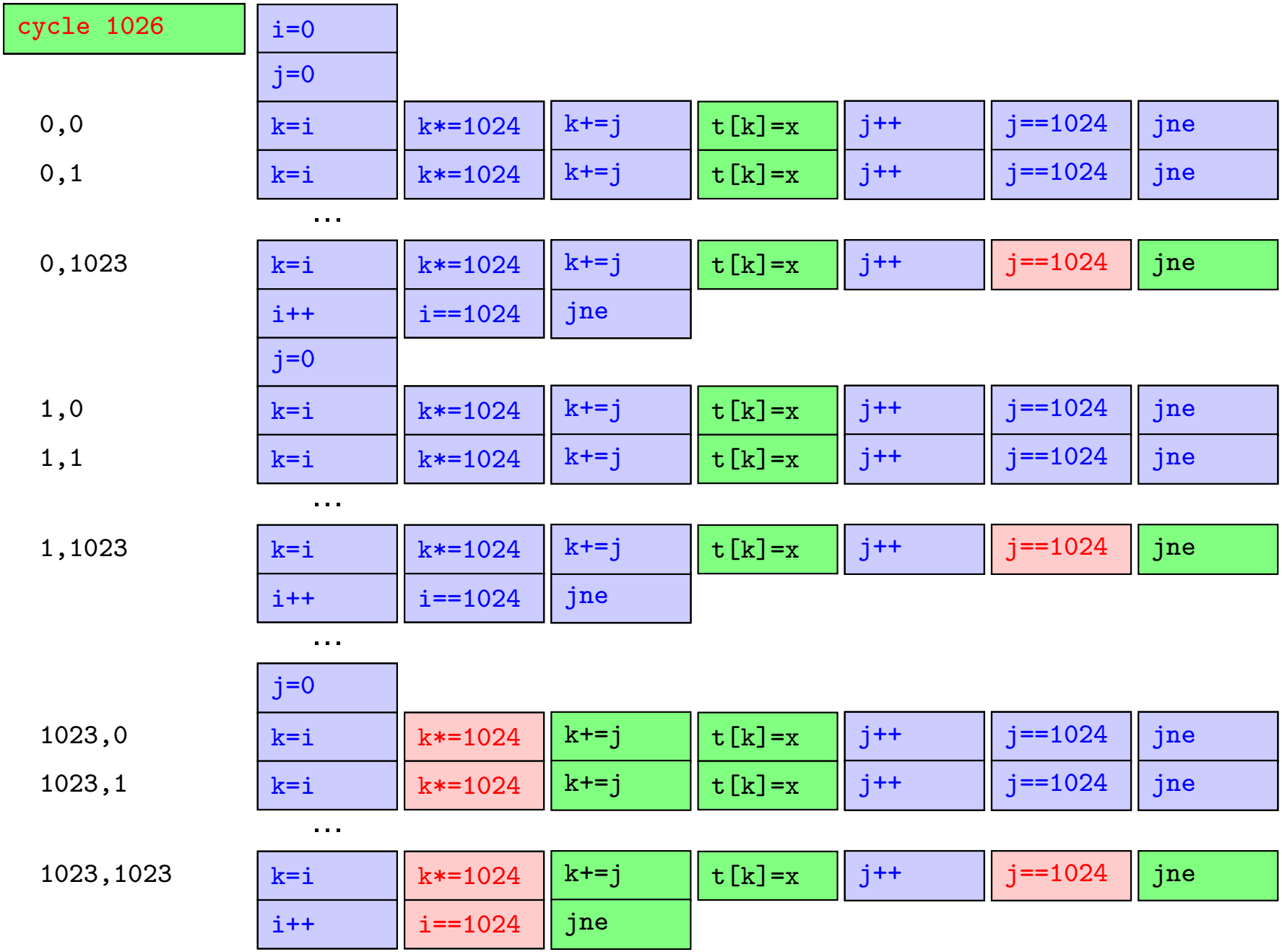
Nested loops.



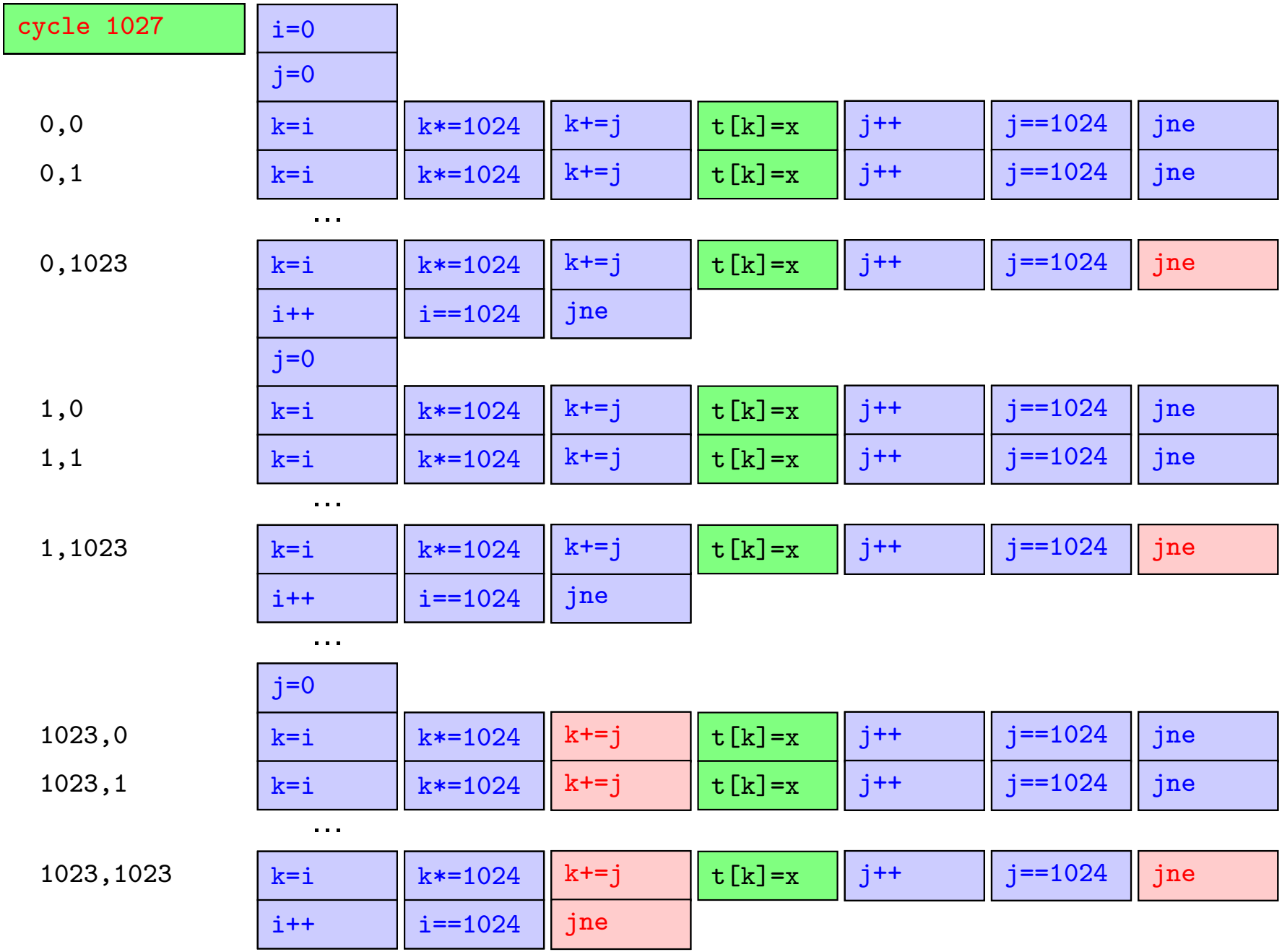
Nested loops.



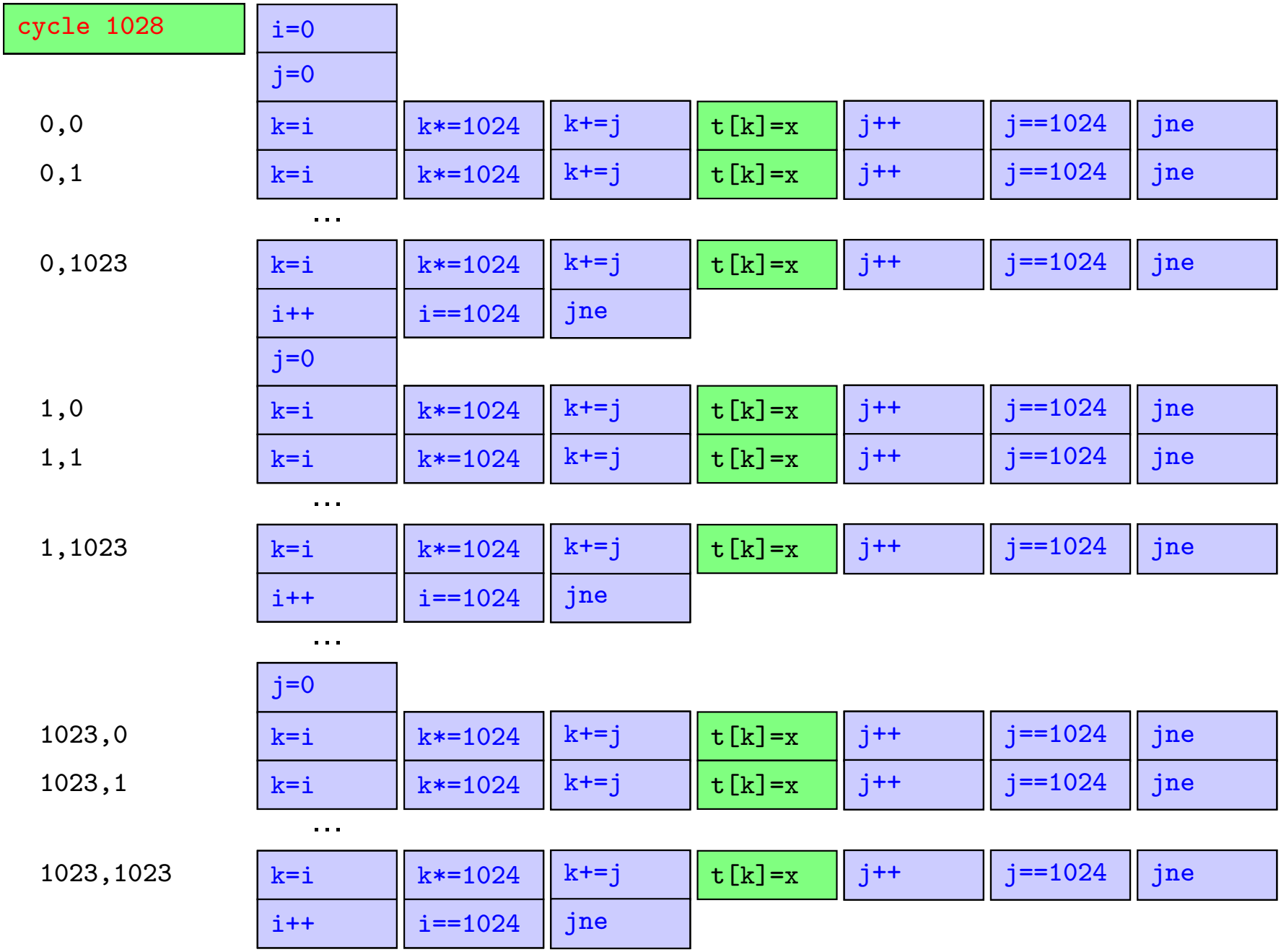
Nested loops.



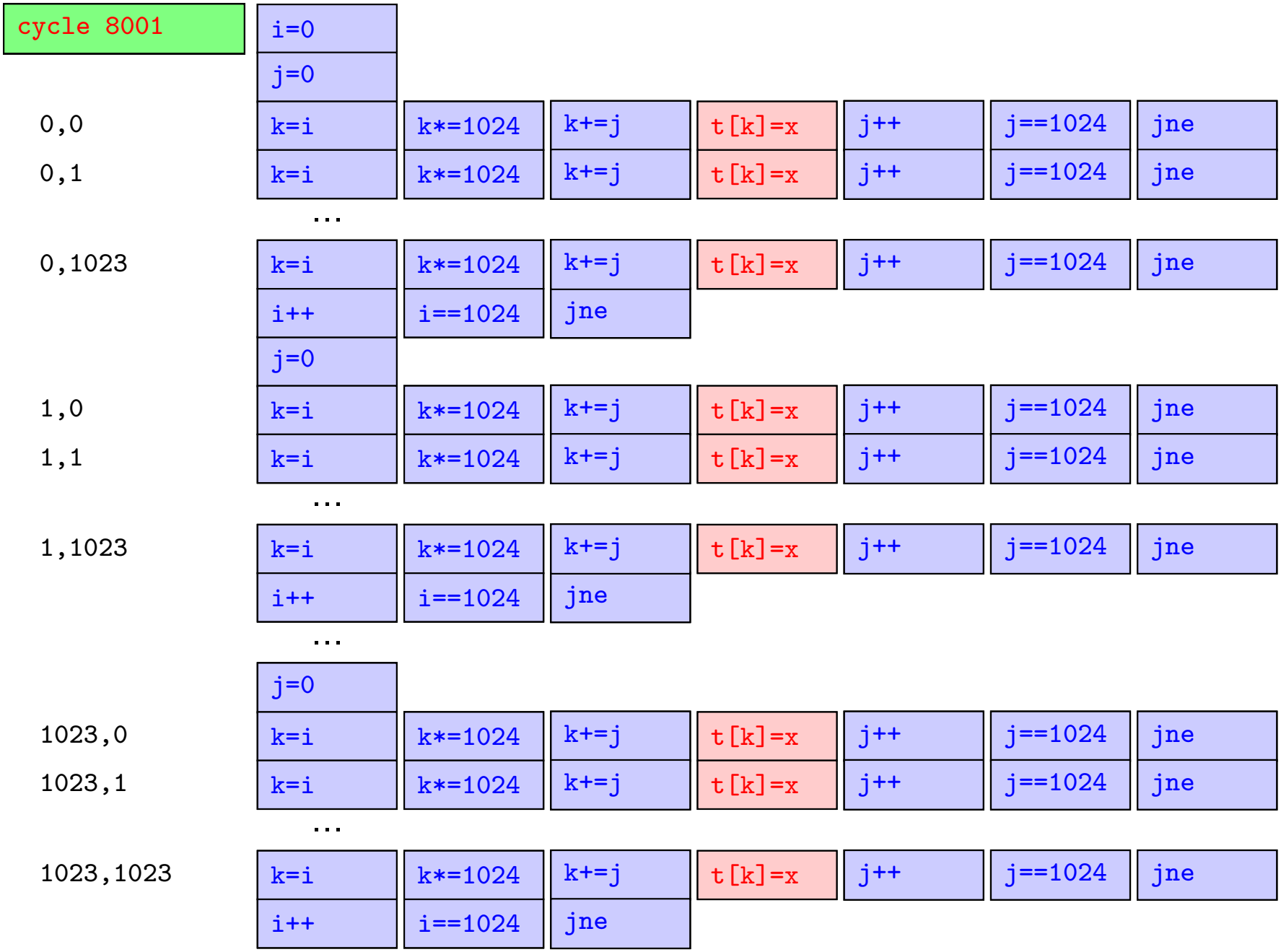
Nested loops.



Nested loops.



Nested loops.



ILP in functions.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.
- This creates artificial WAR and WAW dependencies between possibly independent function calls.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.
- This creates artificial WAR and WAW dependencies between possibly independent function calls.
- The callee may save some result in the stack.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.
- This creates artificial WAR and WAW dependencies between possibly independent function calls.
- The callee may save some result in the stack.
- The caller reads stacked results in its stack frame.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.
- This creates artificial WAR and WAW dependencies between possibly independent function calls.
- The callee may save some result in the stack.
- The caller reads stacked results in its stack frame.
- This is a producer/consumer callee/caller dependency.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.
- This creates artificial WAR and WAW dependencies between possibly independent function calls.
- The callee may save some result in the stack.
- The caller reads stacked results in its stack frame.
- This is a producer/consumer callee/caller dependency.
- The callee may push the caller's return address in the stack and later pop it.

ILP in functions.

- A run is a succession of function calls with a tree organization, a caller being the father of its callees.
- A run is the function calls tree traversal.
- A caller may move some effective arguments on top of the stack.
- The callee reads stacked formal arguments in its stack frame.
- This is a producer/consumer caller/callee dependency, even if no computation occurs.
- The callee uses its stack frame to initialize, read and write its locals.
- All the functions at the same hierarchical level in the tree share the same portion of the stack for their arguments and locals.
- This creates artificial WAR and WAW dependencies between possibly independent function calls.
- The callee may save some result in the stack.
- The caller reads stacked results in its stack frame.
- This is a producer/consumer callee/caller dependency.
- The callee may push the caller's return address in the stack and later pop it.
- This creates a tree traversal dependency chain.

Parallelizing functions.

Parallelizing functions.

- Eliminate false dependencies between independent calls with memory renaming.

Parallelizing functions.

- Eliminate false dependencies between independent calls with memory renaming.
- Avoid simple move argument transmissions (global variables, argument/result computations).

Parallelizing functions.

- Eliminate false dependencies between independent calls with memory renaming.
- Avoid simple move argument transmissions (global variables, argument/result computations).
- Don't save the return address in the stack.

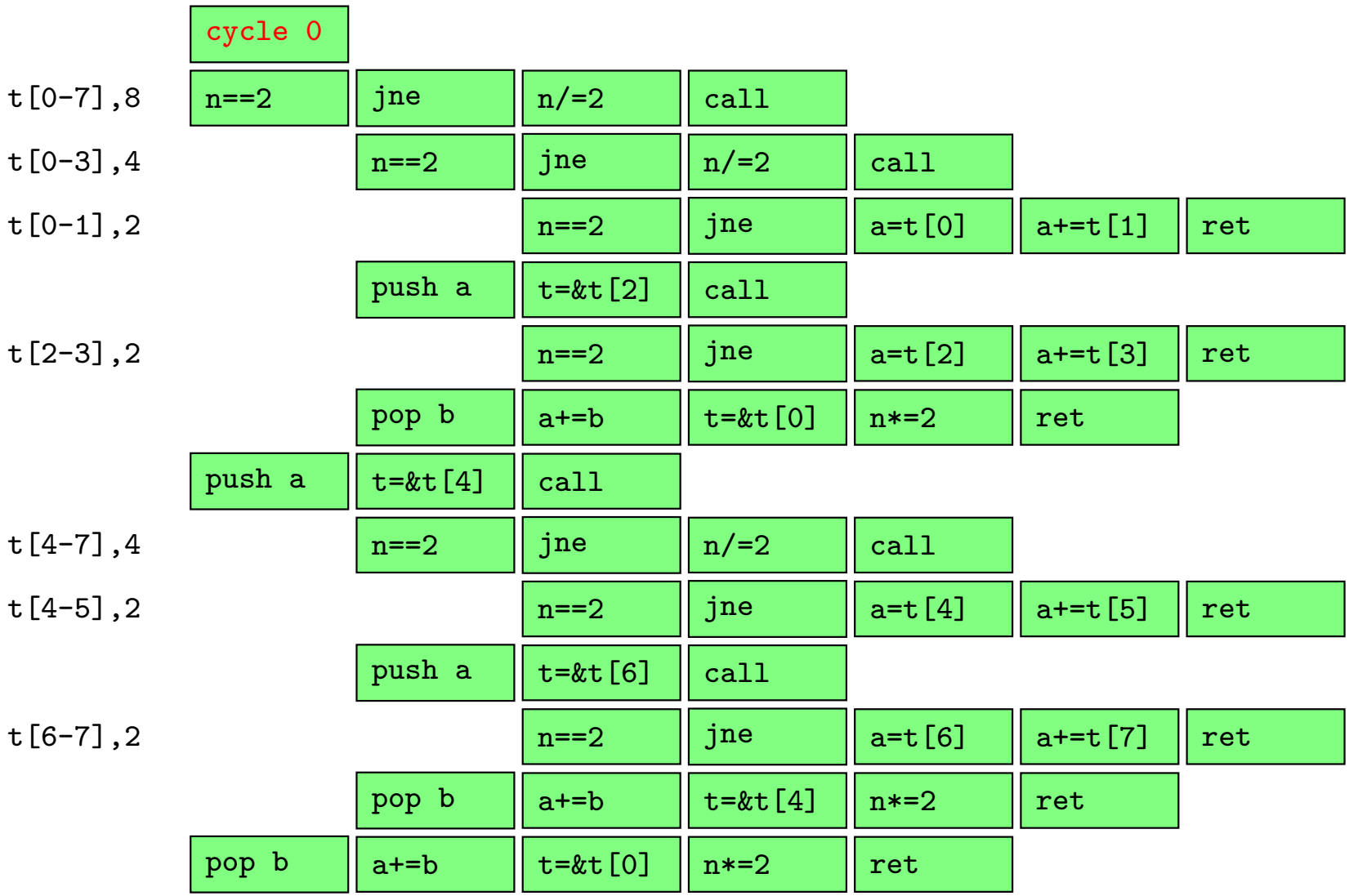
Parallelizing functions.

- Eliminate false dependencies between independent calls with memory renaming.
- Avoid simple move argument transmissions (global variables, argument/result computations).
- Don't save the return address in the stack.
- Only true dependencies should remain, linking a producing function to a consuming one.

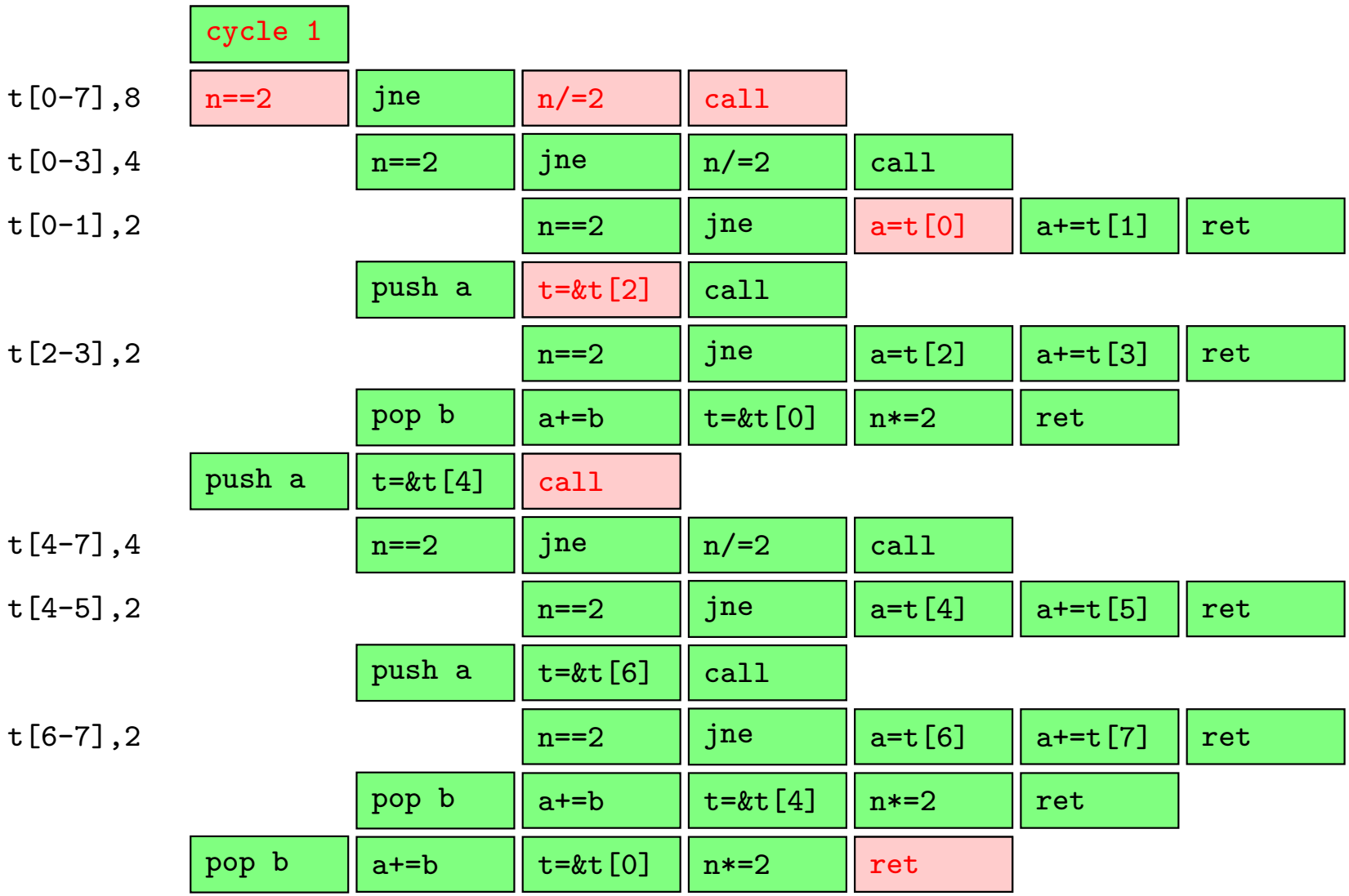
Parallelizing functions.

- Eliminate false dependencies between independent calls with memory renaming.
- Avoid simple move argument transmissions (global variables, argument/result computations).
- Don't save the return address in the stack.
- Only true dependencies should remain, linking a producing function to a consuming one.
- Functions controls are often independent and can be run in parallel.

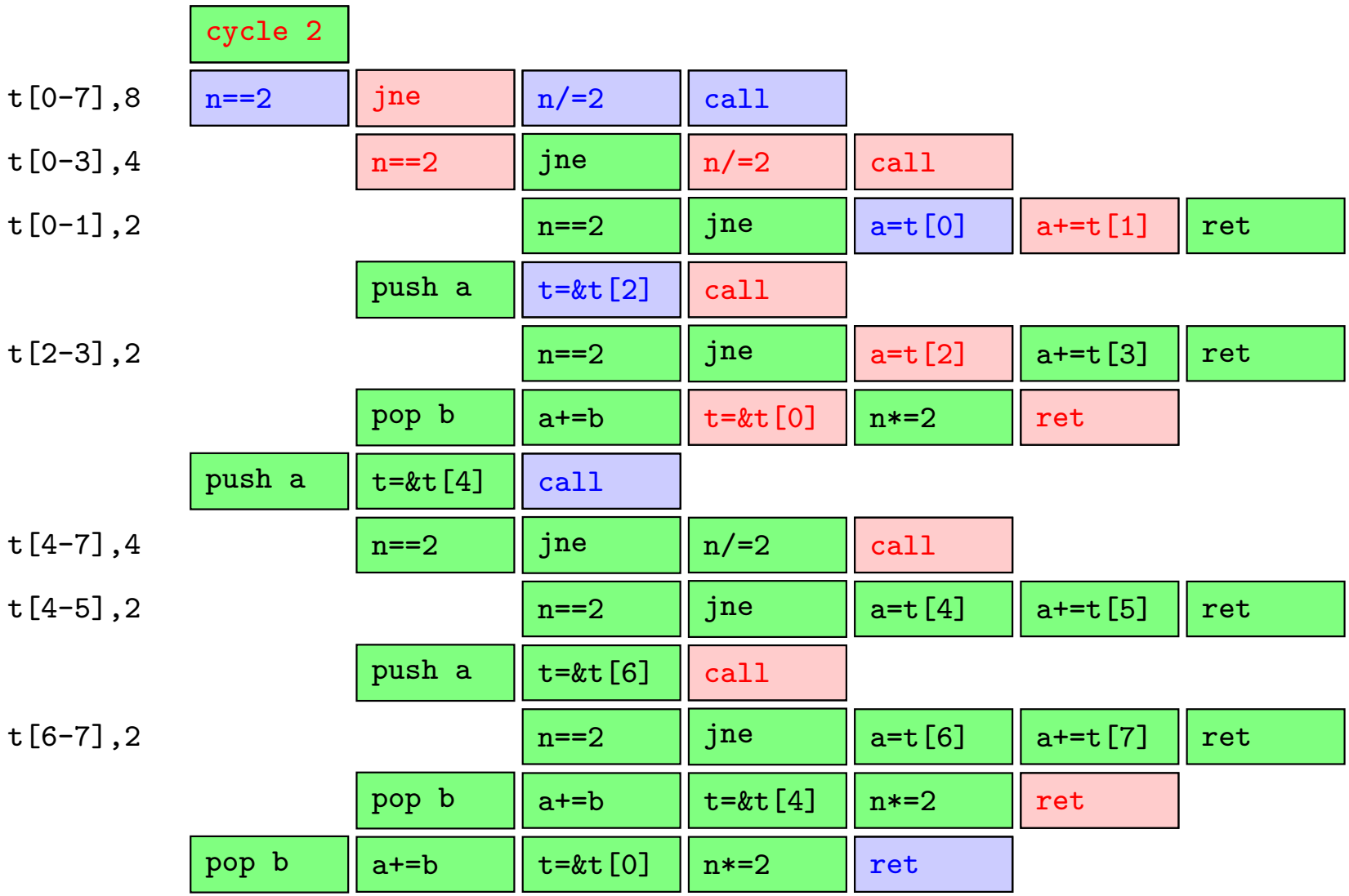
Running sum_reduce(t[0,7],8) in parallel.



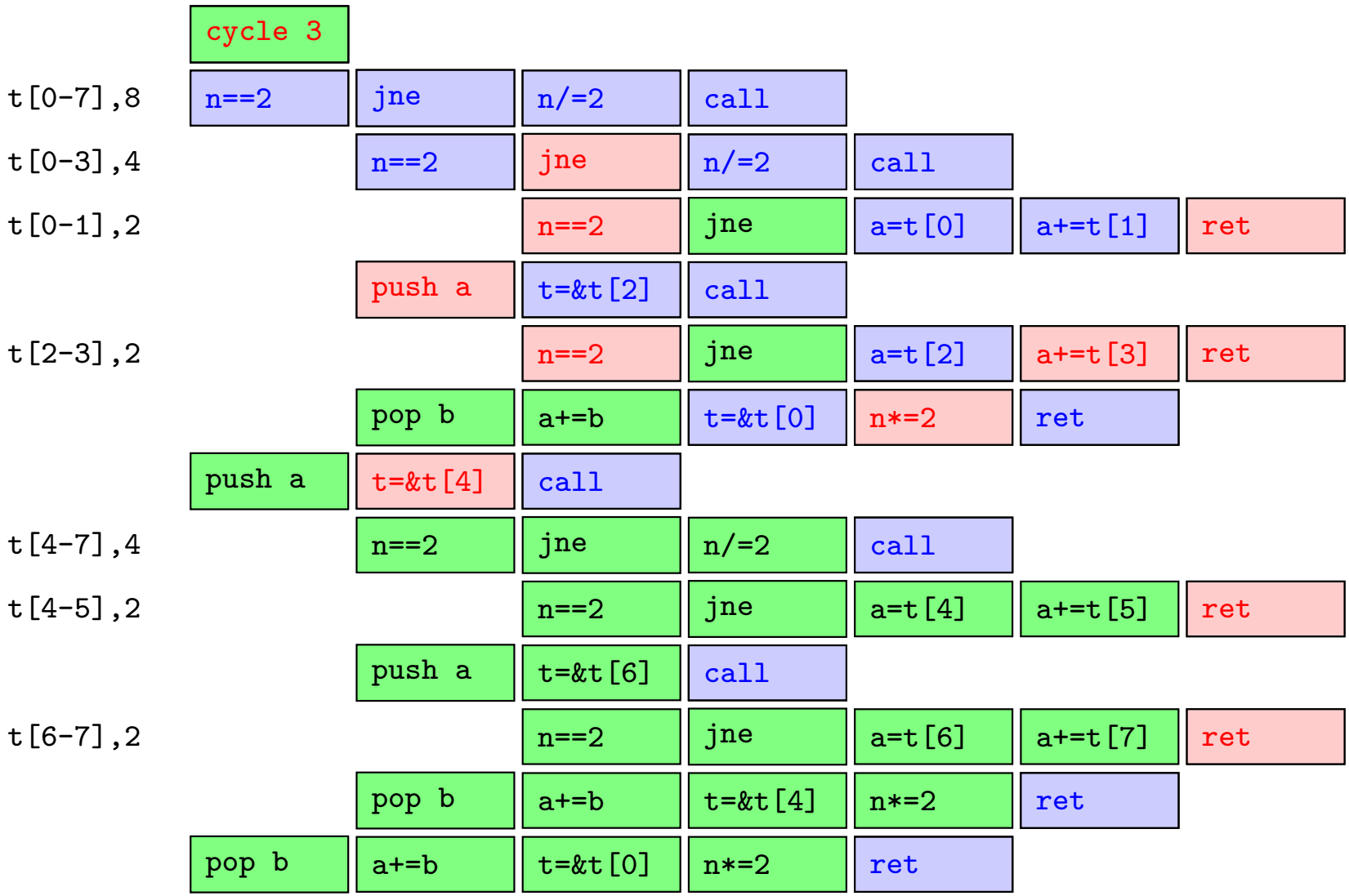
Running sum_reduce(t[0,7],8) in parallel.



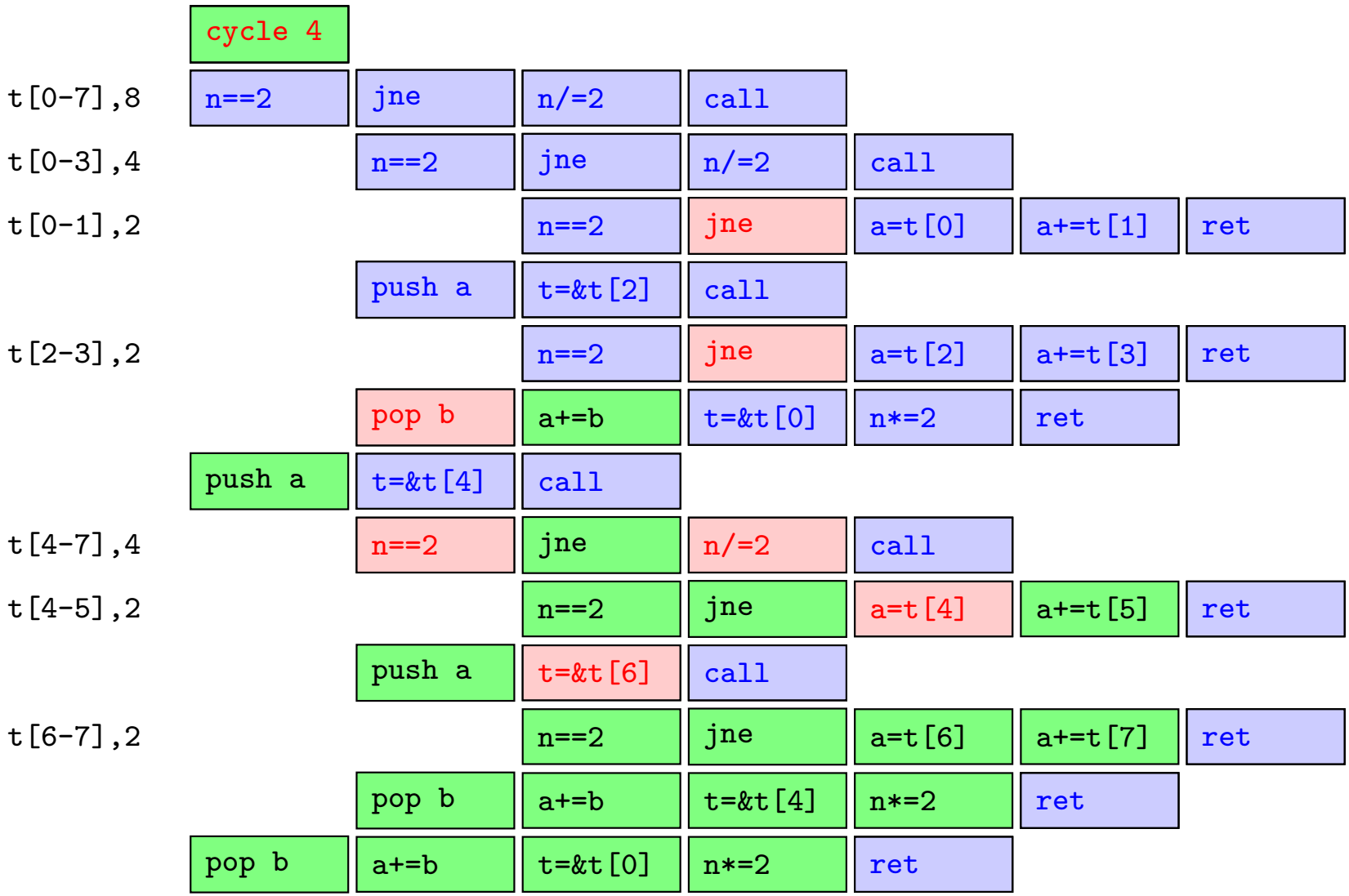
Running sum_reduce(t[0,7],8) in parallel.



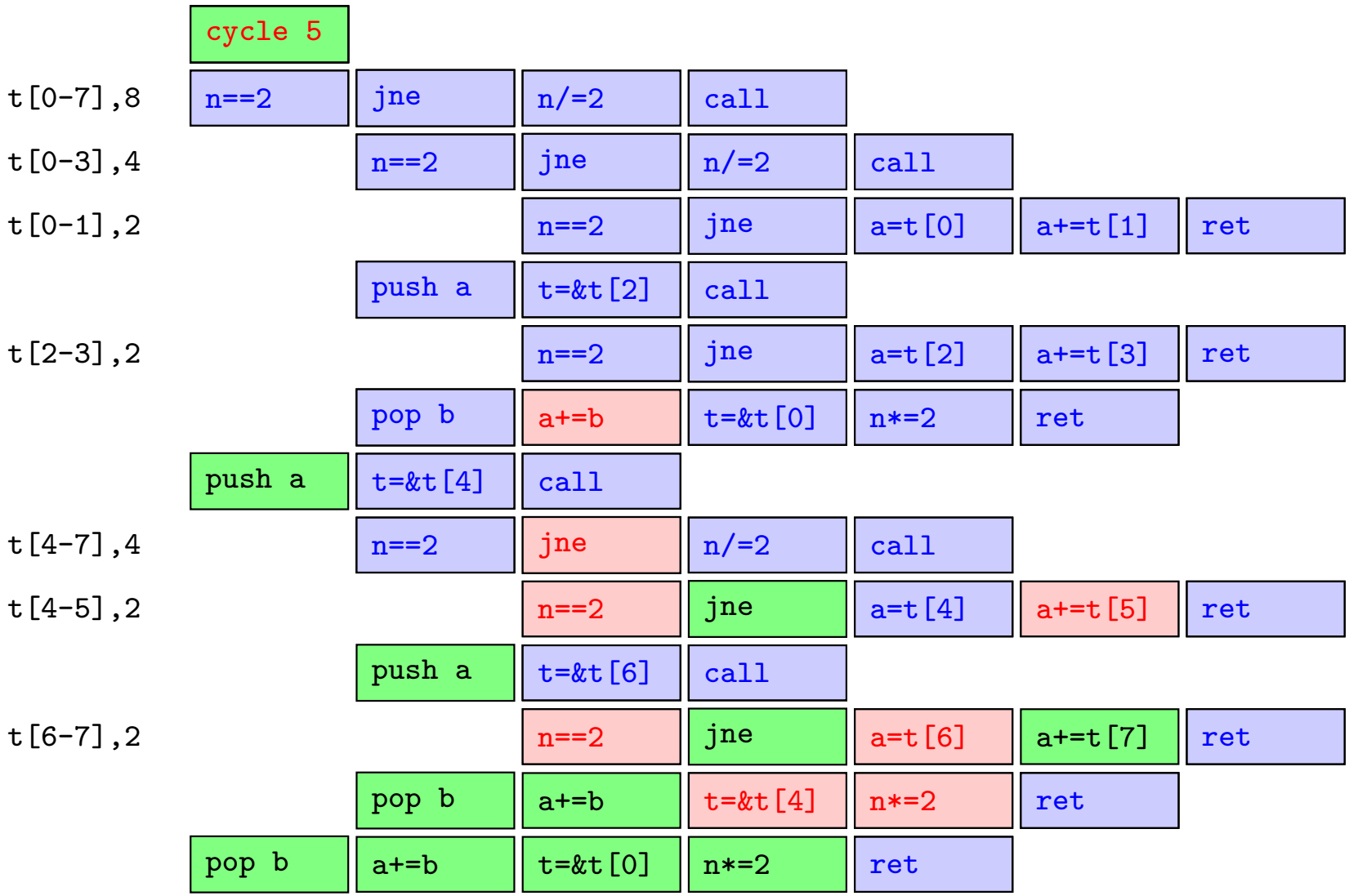
Running sum_reduce(t[0,7],8) in parallel.



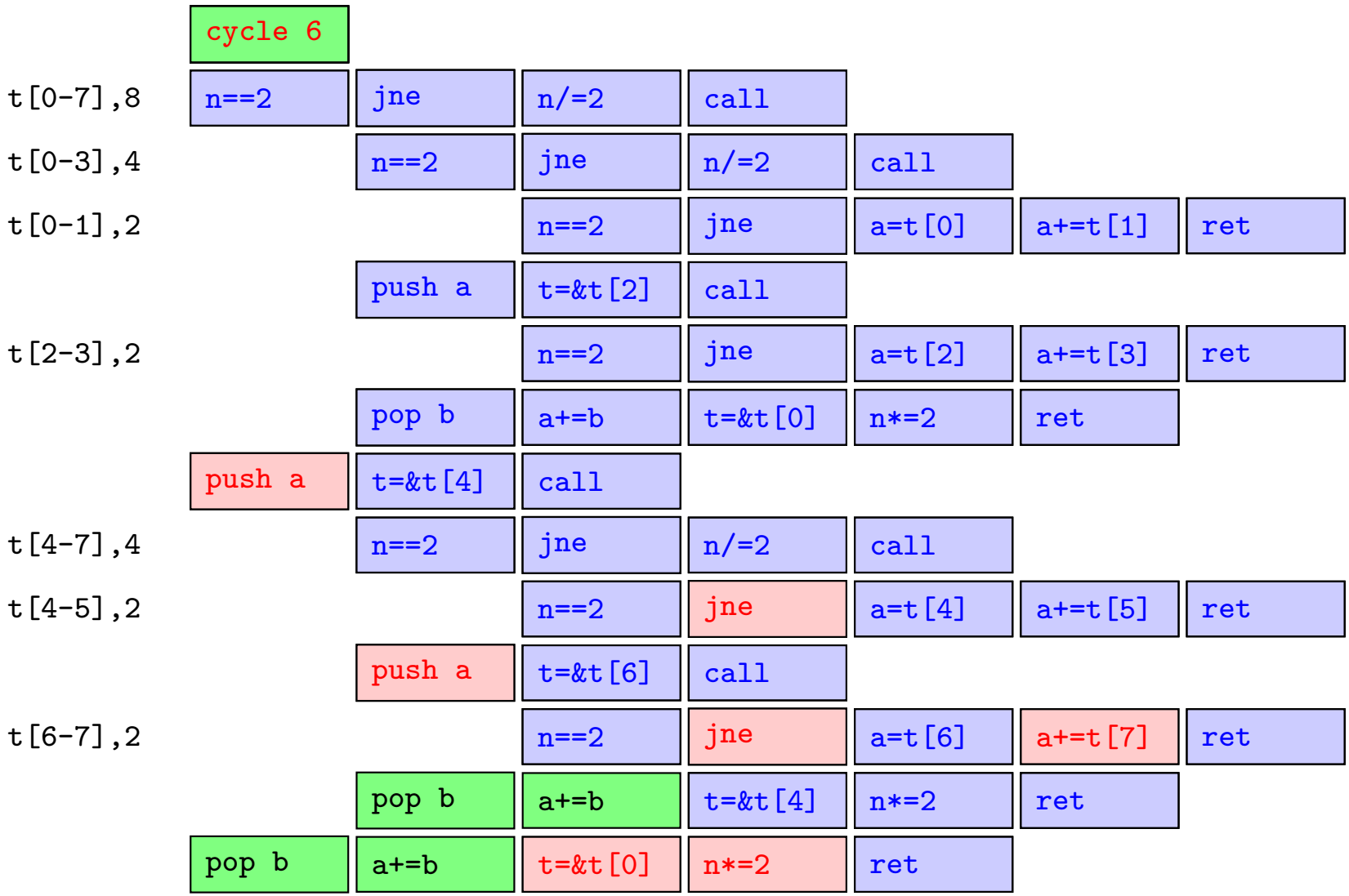
Running sum_reduce(t[0,7],8) in parallel.



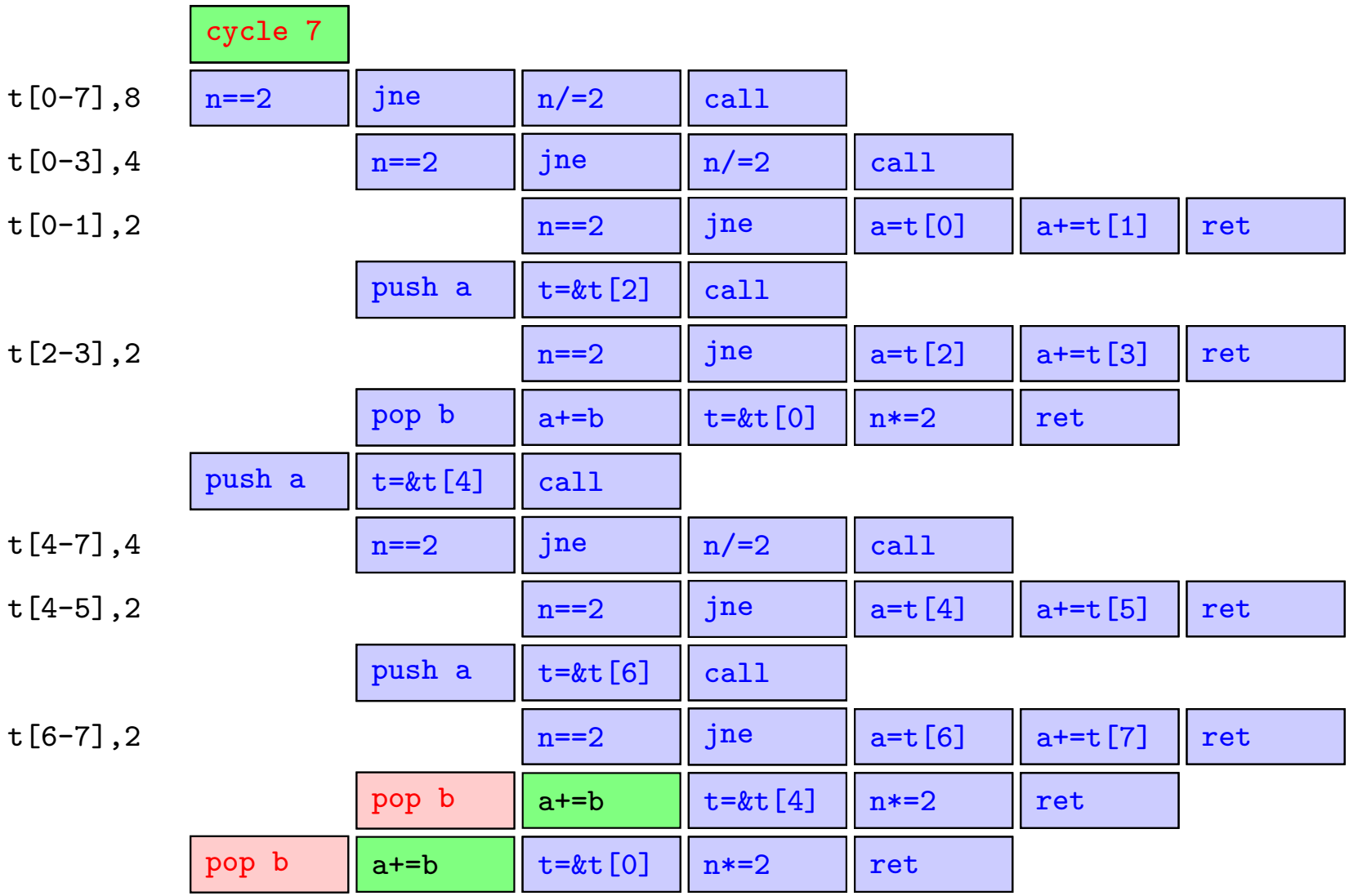
Running sum_reduce(t[0,7],8) in parallel.



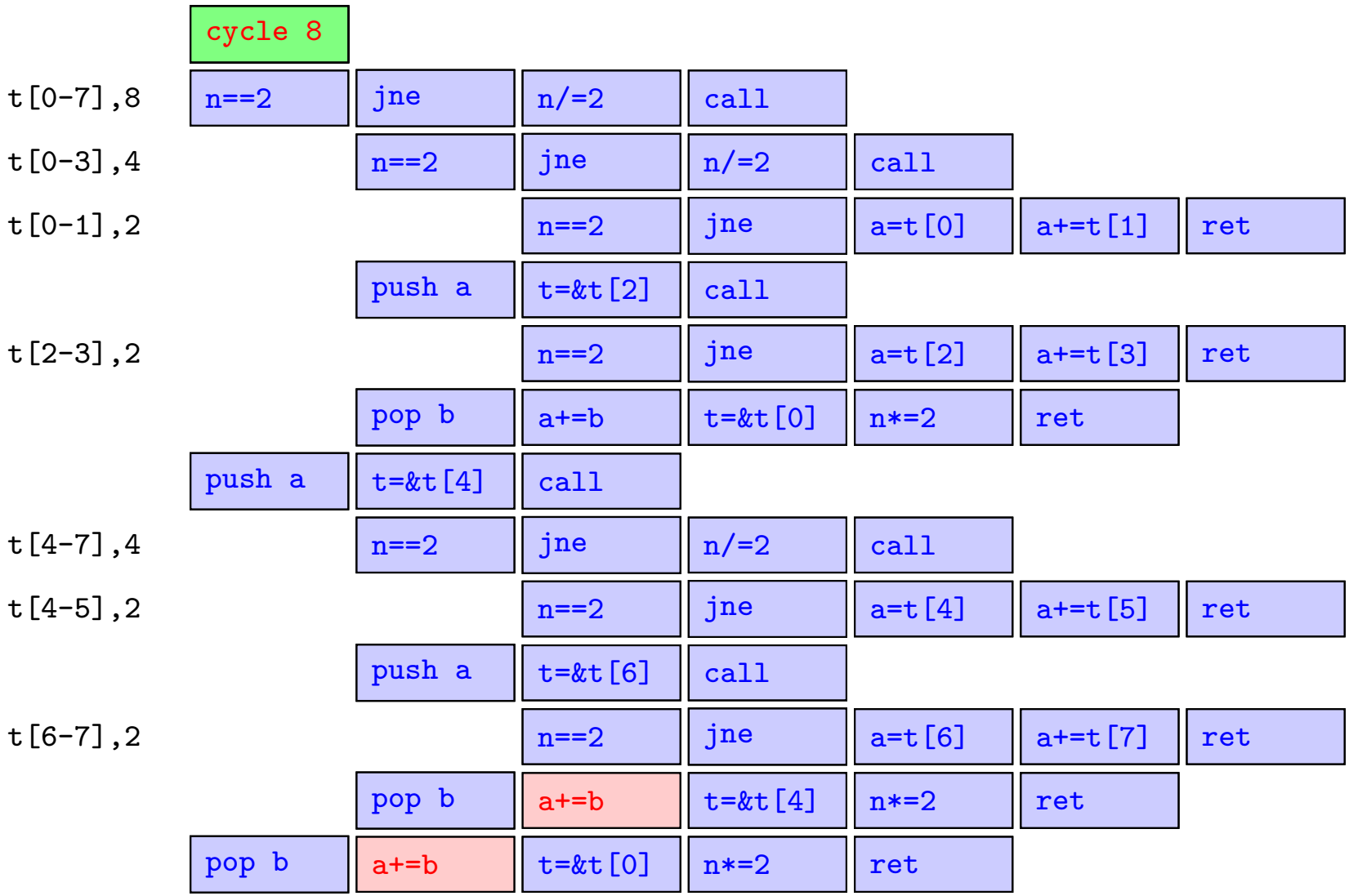
Running sum_reduce(t[0,7],8) in parallel.



Running sum_reduce(t[0,7],8) in parallel.



Running sum_reduce(t[0,7],8) in parallel.



Parallelizing programs.

Parallelizing programs.

- Use a multicore processor.

Parallelizing programs.

- Use a multicore processor.
- Fork at call, join at return.

Parallelizing programs.

- Use a multicore processor.
- Fork at call, join at return.
- Parallelize fetch, parallelize rename, parallelize run.

Parallelizing programs.

- Use a multicore processor.
- Fork at call, join at return.
- Parallelize fetch, parallelize rename, parallelize run.
- Map memory on physical registers and generalize renaming.

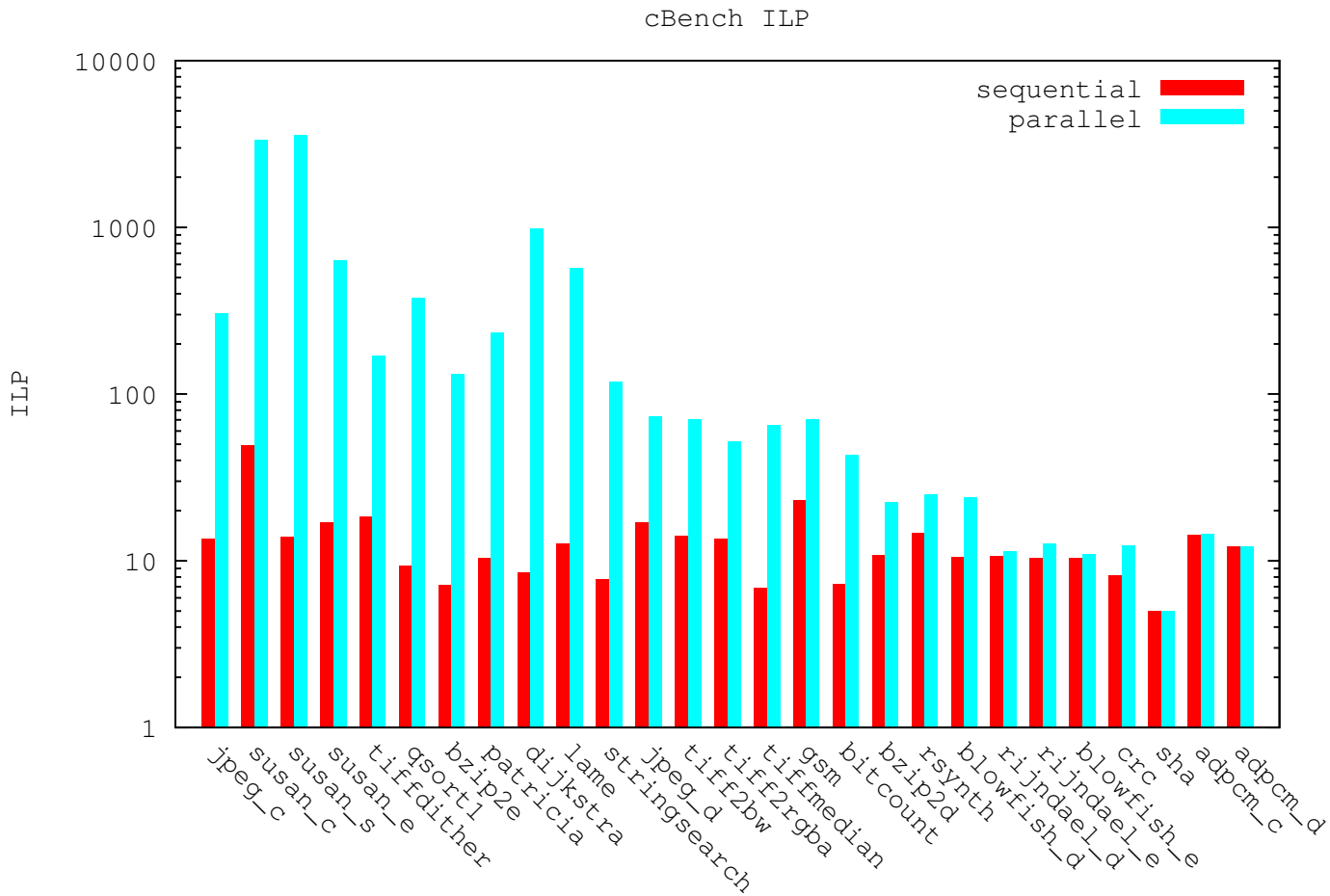
Parallelizing programs.

- Use a multicore processor.
- Fork at call, join at return.
- Parallelize fetch, parallelize rename, parallelize run.
- Map memory on physical registers and generalize renaming.
- Import distant values from producing cores.

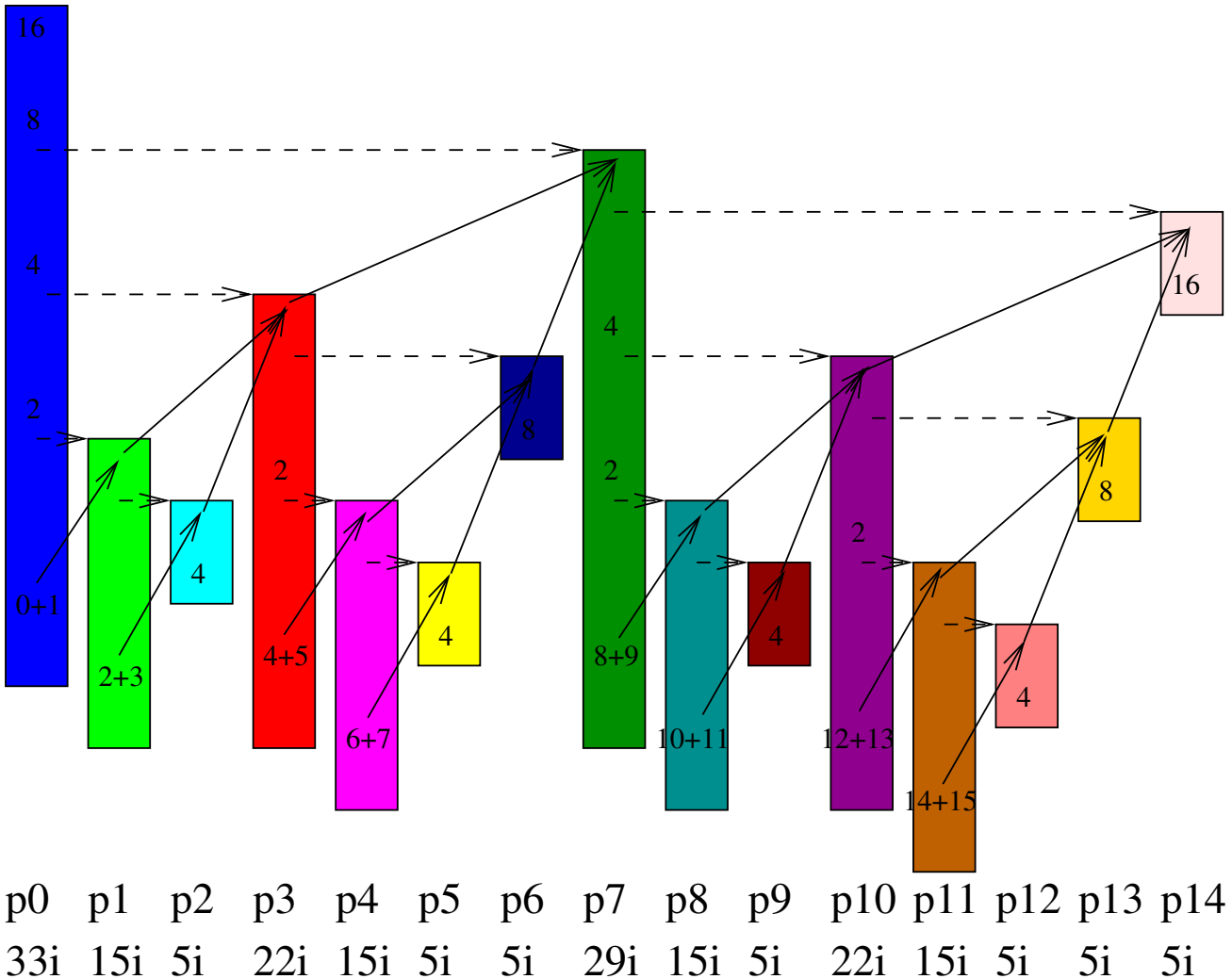
Parallelizing programs.

- Use a multicore processor.
- Fork at call, join at return.
- Parallelize fetch, parallelize rename, parallelize run.
- Map memory on physical registers and generalize renaming.
- Import distant values from producing cores.
- Optionally, by renaming constants we can precompute loop control at rename phase to allow vectorization at run phase.

Impact de la parallélisation de l'extraction, du renommage et de l'exécution.

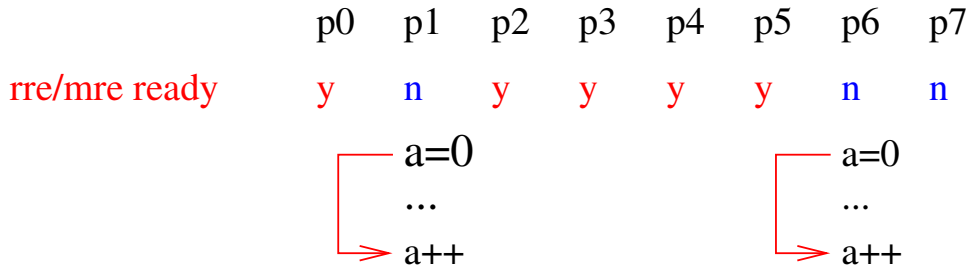


Le modèle d'exécution parallèle appliqué à sum_reduce(t,16).

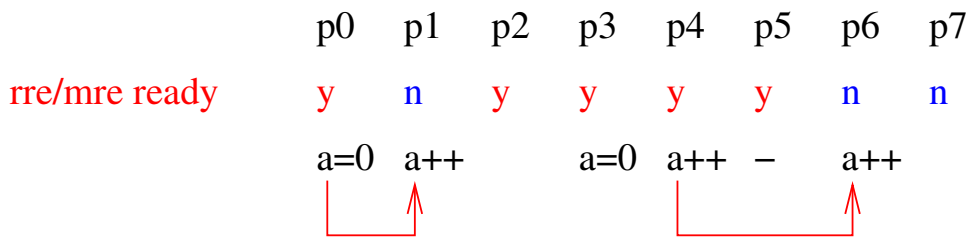


Le renommage en parallèle.

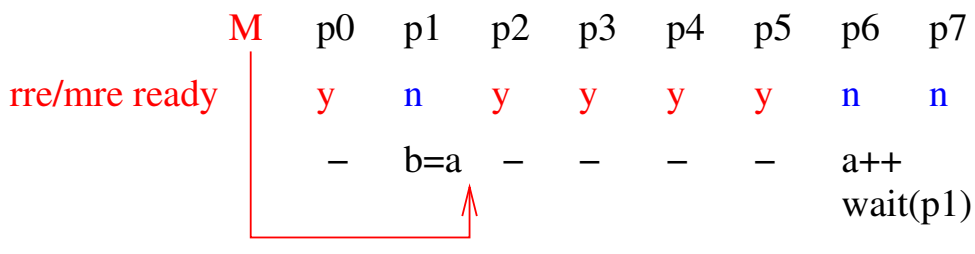
(a) parallel local imports



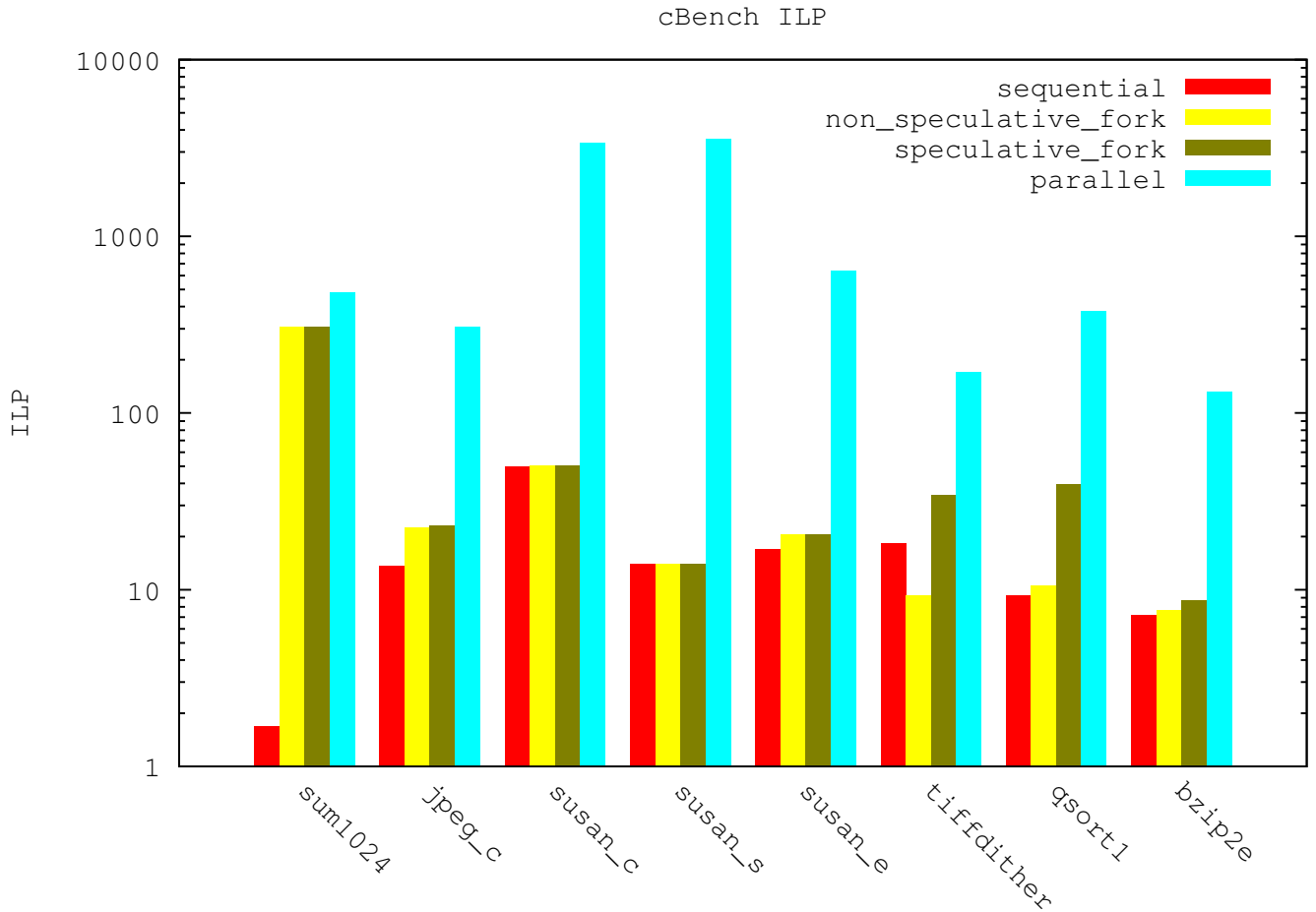
(b) parallel global imports



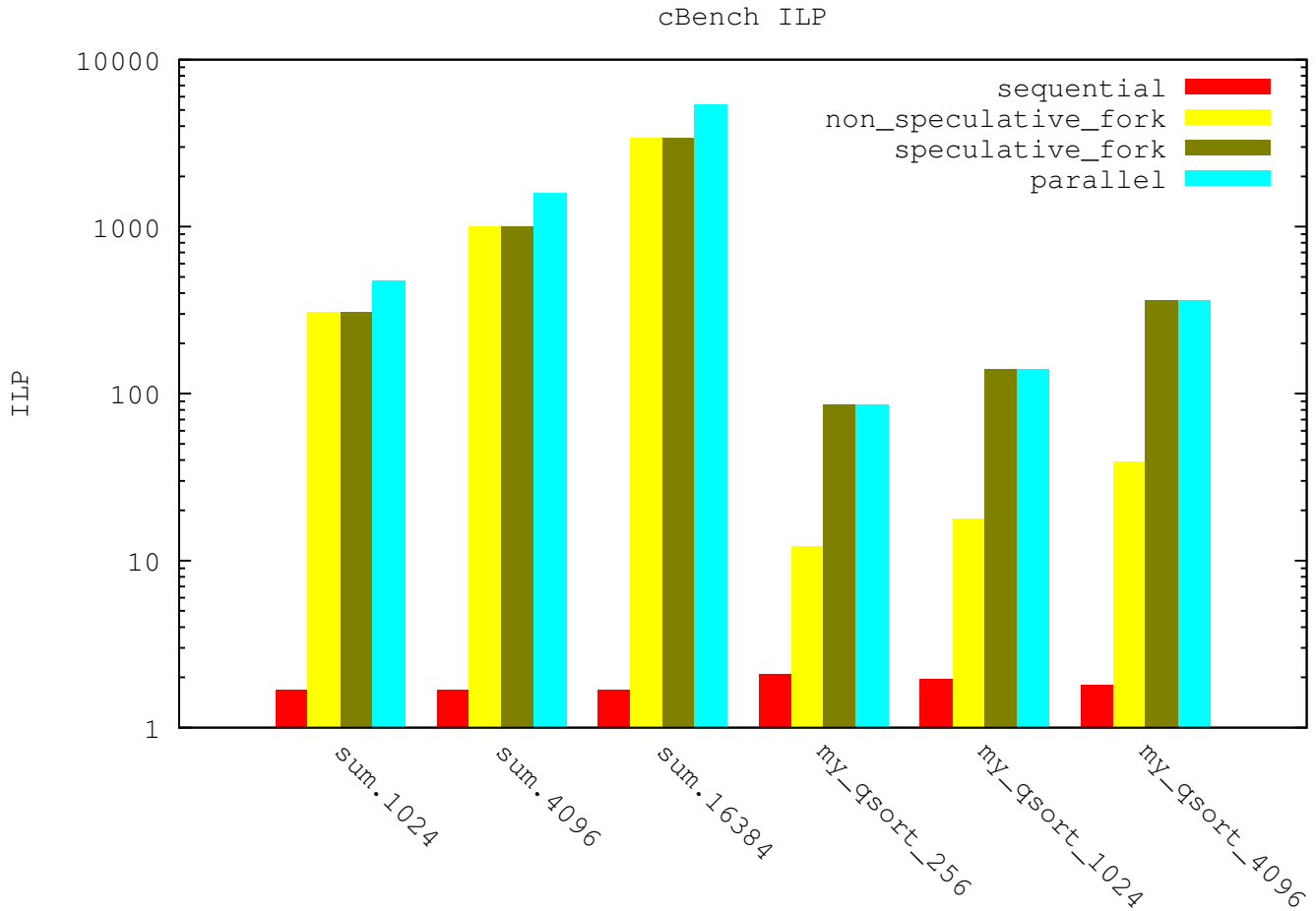
(c) memory import and import wait



La capture d'ILP avec fork spéculatif et non spéculatif.



La capture d'ILP sur deux algorithmes parallèles.



Conclusion.

Conclusion.

- Parallelism is in the algorithm.

Conclusion.

- Parallelism is in the algorithm.
- The hardware implementation sequentializes.

Conclusion.

- Parallelism is in the algorithm.
- The hardware implementation sequentializes.
- We re-parallelize through extended renaming and forking.

Conclusion.

- Parallelism is in the algorithm.
- The hardware implementation sequentializes.
- We re-parallelize through extended renaming and forking.
- The processor moves quickly along the fetch path thanks to the forking mechanism.

Conclusion.

- Parallelism is in the algorithm.
- The hardware implementation sequentializes.
- We re-parallelize through extended renaming and forking.
- The processor moves quickly along the fetch path thanks to the forking mechanism.
- This fast forwarding feeds the cores with independent instructions.

Conclusion.

- Parallelism is in the algorithm.
- The hardware implementation sequentializes.
- We re-parallelize through extended renaming and forking.
- The processor moves quickly along the fetch path thanks to the forking mechanism.
- This fast forwarding feeds the cores with independent instructions.
- No need to rewrite the C version of the algorithm which can be interpreted as parallel.

Conclusion.

- Parallelism is in the algorithm.
- The hardware implementation sequentializes.
- We re-parallelize through extended renaming and forking.
- The processor moves quickly along the fetch path thanks to the forking mechanism.
- This fast forwarding feeds the cores with independent instructions.
- No need to rewrite the C version of the algorithm which can be interpreted as parallel.
- What we usually call sequential programs contain a lot of parallelism unless the algorithms they come from are themselves sequential.