[A]rchitecture et [O]ptimisation de [C]ode pour microprocesseur hautes performances

Code Optimizations for Microarchitecture

Allen D. Malony

Department of Computer and Information Science
University of Oregon

Outline

- Introduction
- Intermediate representations and analysis
- Sequential model
- ILP model
- Scheduling
- Optimizations for memory hierarchy
- Optimizations for the SIMD

Origin of Course Materials

- Patrick Carribault, CEA
  - AOC, 2016

Introduction

- Optimizing a code
  - Minimize or maximize a cost function
  - Our objective is to increase IPC
- Which part of the code to optimize?
  - First need to find execution inefficiences (hot spots)
  - Profiling the code (e.g., with gprof) shows:
    - where time is spent (code locations)
    - performance metrics: time, hardware counters, …
  - Code parts: instruction, loops, basic blocks, …
- Algorithmic
  - Compiler does not improve program complexity
  - Just reduces the constants of this complexity

Concepts

- *Code analysis*
  - Analysis of program code to understand properties
  - Goal is to help in code optimization, bug detection, …
- *Code transformation* or *restructuring*
  - Rewriting of a program into another program
    - Should maintain execution semantics
  - Goal is to produce a new program with better qualities
    - improve its readability
    - enable optimization of performance, memory management, …
- *Code optimization*
  - Rewrite a program to generate a better version

Basic Blocks

- *Control flow* in a program
  - What determines which next instruction to execute
  - Instructions that alter the PC (e.g., jump, branch, subroutine call) can change the control flow
- A *base block* is a sequence of consecutive statements where the execution flow (control flow) enters at the beginning and leaves only at the end of the sequence
- The first instruction can be the destination of a branch in the program’s control flow
- Only the last instruction of a base block can be a branch

Algorithm for Partitioning in Basic Blocks

1. Identify the *leading instruction* of a basic block:

   Rule 1: First instruction of a program is a leading instruction

   Rule 2: Any instruction that is a branch destination (i.e., has a label) is a leading instruction

   Rule 3: Any instruction immediately following a branch is a leading instruction

2. Construct the basic block:
   - Starting with the leading instruction …
   - … include all subsequent instructions until the next leading statement is encountered

Basic Block Example

- Consider the Cartesian product of two vectors
- Look at the source code and the intermediate code
  - “Transformed” intermediate code is a *3-address code*

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i]
         i = i+ 1;
   end
   while i <= 20
end
```

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Basic Block Example (2)

- Apply basic block analysis to identify the leading instructions in the intermediate code

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i]
         i = i+ 1;
   end
   while i <= 20
end
```

Rule 1

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
(13)   …
```

Basic Block Example (3)

- Apply basic block analysis to identify the leading instructions in the intermediate code

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i]
         i = i+ 1;
   end
   while i <= 20
end
```

Rule 1

Rule 2

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
(13)   …
```

Basic Block Example (4)

- Apply basic block analysis to identify the leading instructions in the intermediate code

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i]
         i = i+ 1;
   end
   while i <= 20
end
```

Rule 1

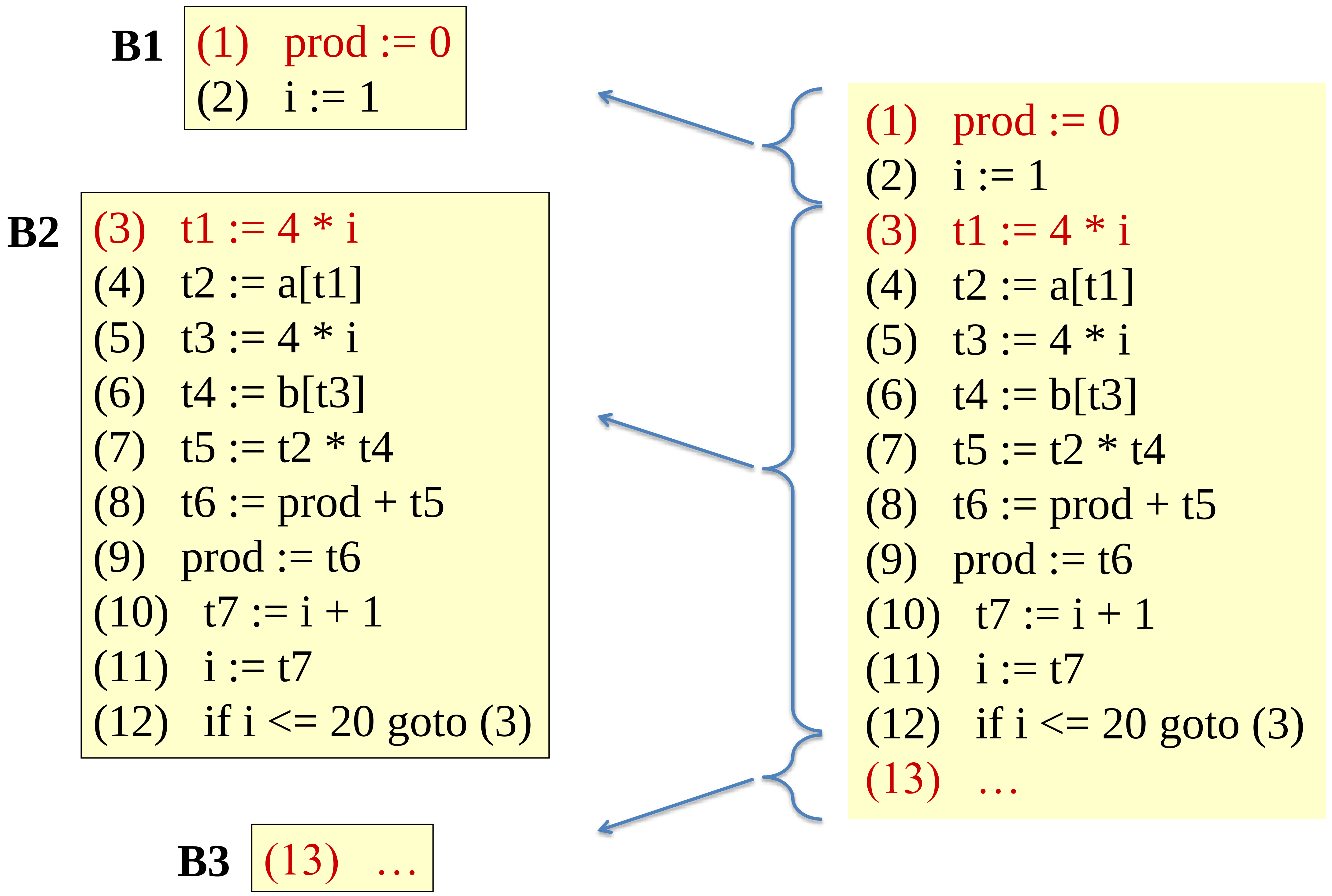
Rule 2

Rule 3

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
(13)   …
```

Basic Block Example (5)

- Now construct basic blocks



B1
```
(1)   prod := 0
(2)   i := 1
```

B2
```
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
```

B3
```
(13)   …
```

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
(13)   …
```

Control Flow Graph (CFG)

- A *control flow graph* (CFG) is a directed graph such that:
  - Nodes in the graph represent base blocks
  - Arcs (edges) in the graph represent control flow
    - possible out-of-order execution
- The starting node of the CFG is the node that contains the first instruction of the program
- There may be several final nodes because there can be several "exits" in the program

CFG Construction

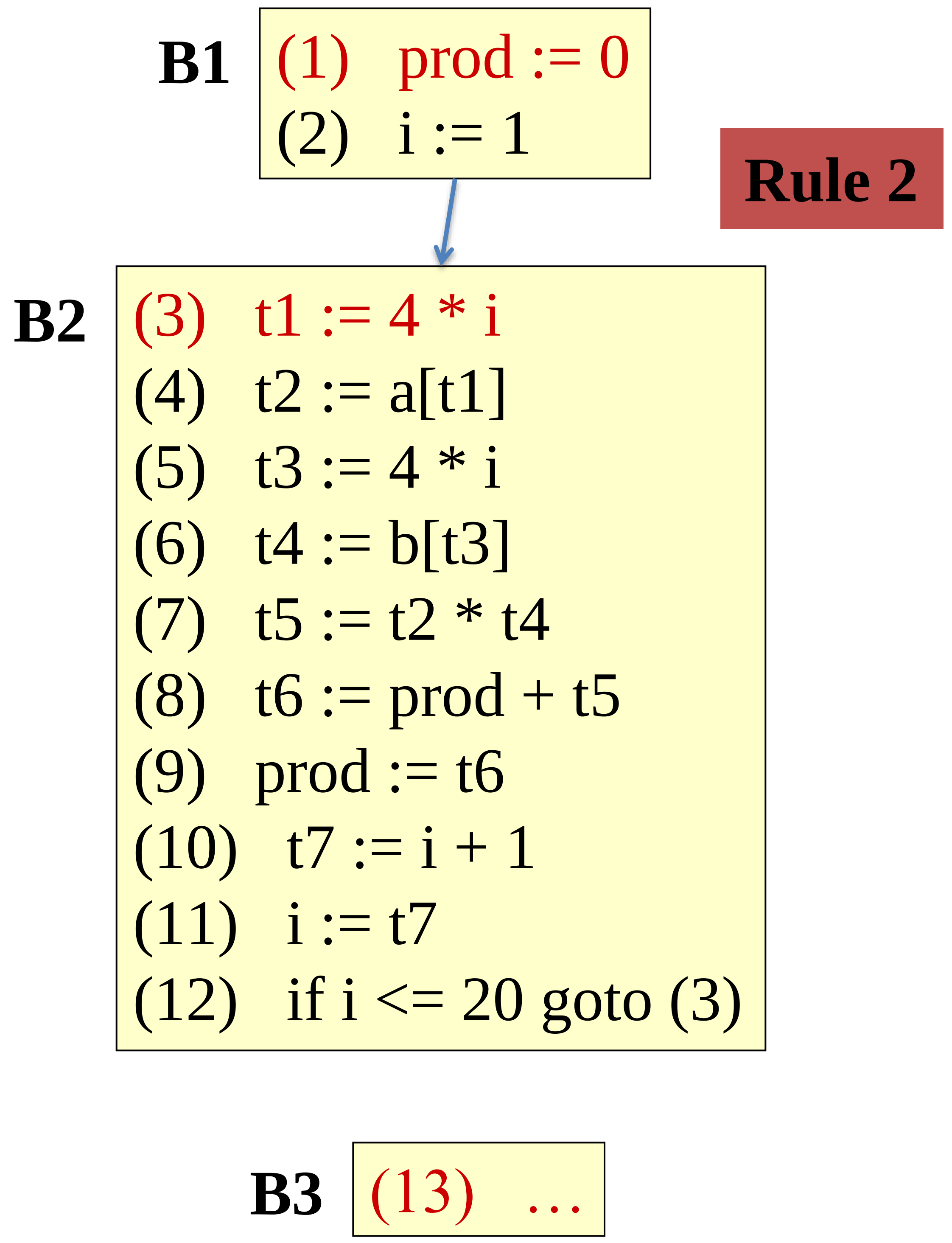
- Consider two basic blocks in a CFG: B1 and B2
- There is an oriented arc from basic block B1 to the basic block B2 in the CFG if:

  Rule 1: There is a branch of the last instruction from B1 to the B2 leading instruction, or

  Rule 2: B2 immediately follows B1 and B1 does not end with an unconditional branch

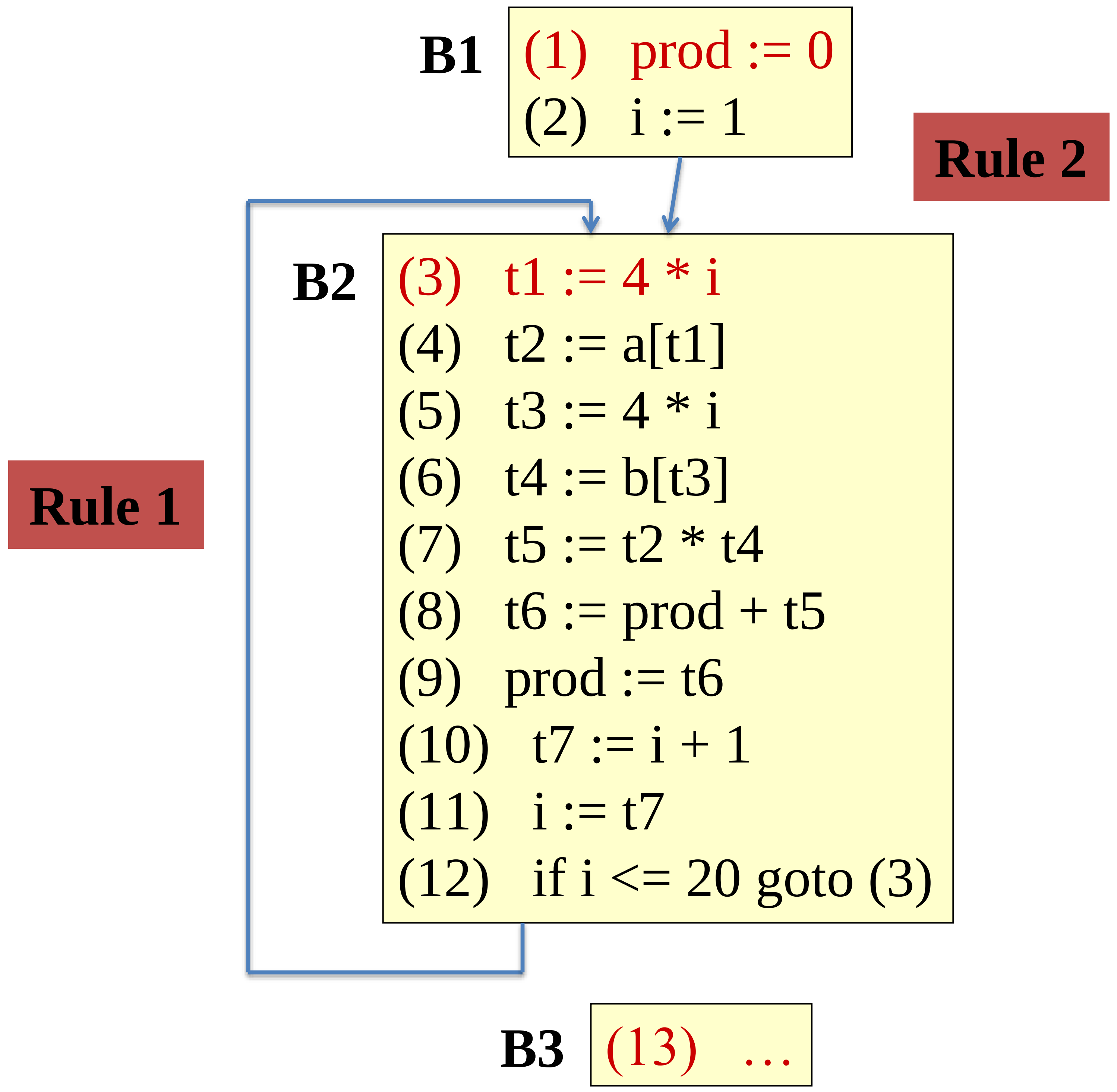
CFG Construction (2)

- Now construct basic blocks



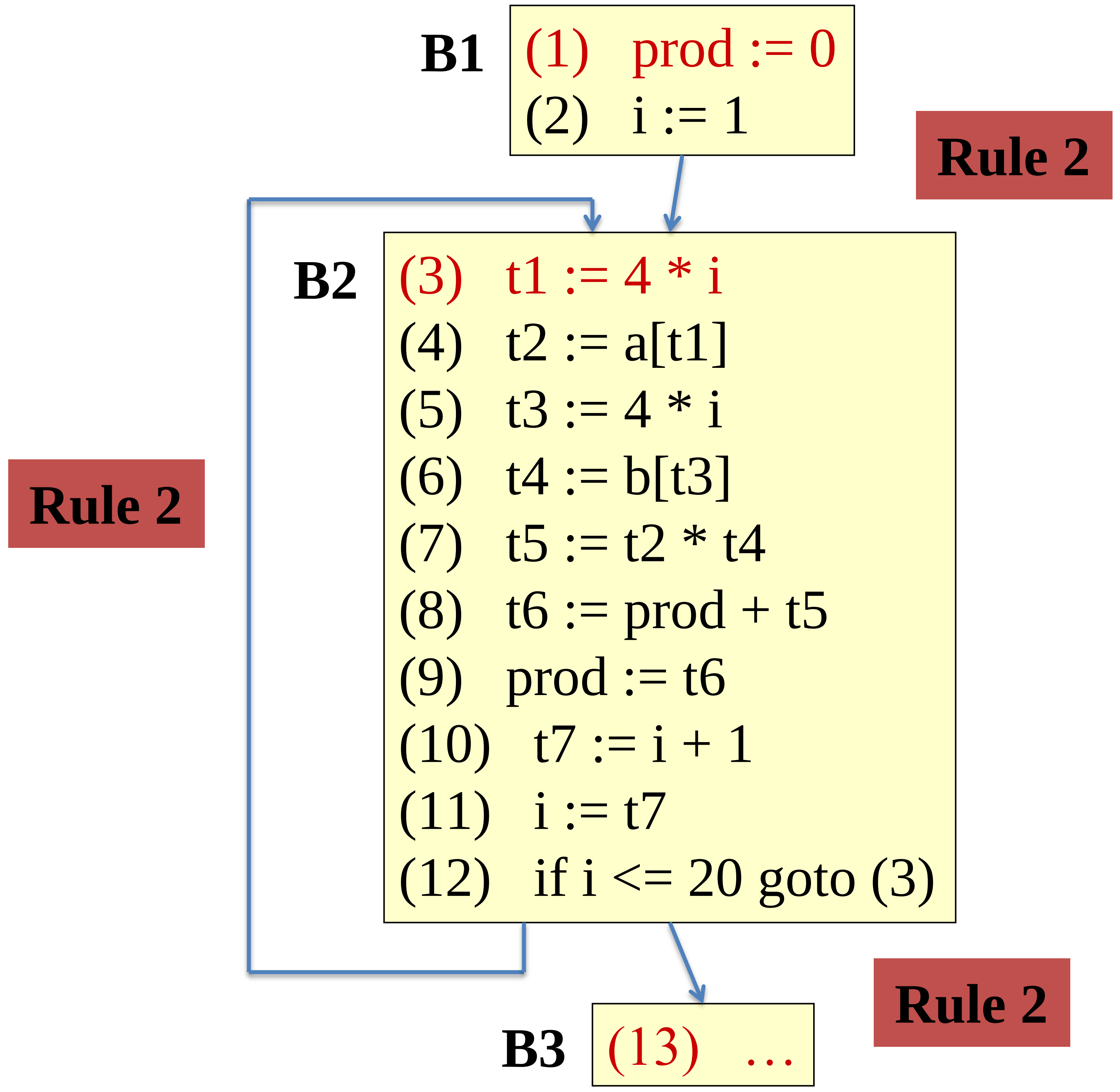
CFG Construction (3)

- Now construct basic blocks



CFG Construction (2)

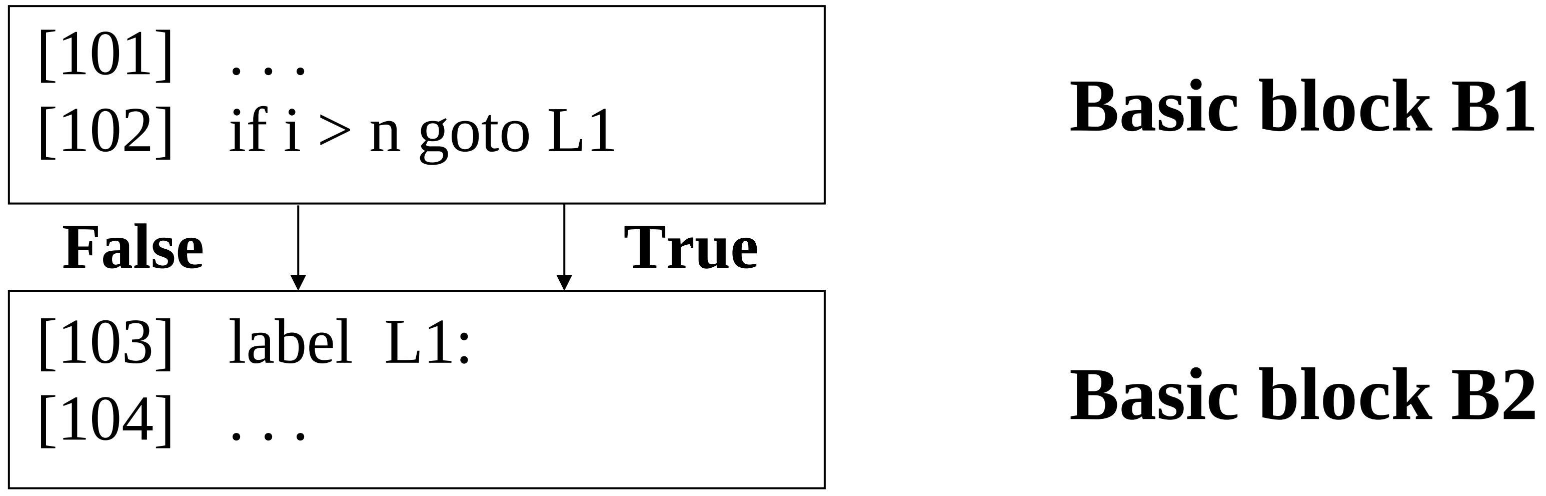
- Now construct basic blocks



$$\begin{tikzcd}B1 \arrow[d] \\ B2 \arrow[d] \arrow[loop right] \\ B3\end{tikzcd}$$

CFGs are Multigraphs

- There may be several arcs from one base block to another in a CFG
- So, in general, a CFG is a multi-graph
- Arcs can be distinguished by the conditions labels
- A trivial example below:



Function Call Graph

- A *function call graph* attempts represent the control flow in a program due to subroutines (functions)
  - It is a graph complementary to the CFG
  - At the function level
- Because functions organize code beyond basic blocks, function call graph shows a higher-level of application structure
- Function call graph structure
  - Each node represents a function
  - An arc exist from F1 to F2 if F1 calls F2

Function Call Graph (2)

- Consider the following code

```
int main (){
  …
  y=foo(5);
  …
}
```

```
int foo (int y){
  int x=y+3;
  if (x-5= 0)
    x=atoi(…);
  return x;
}
```

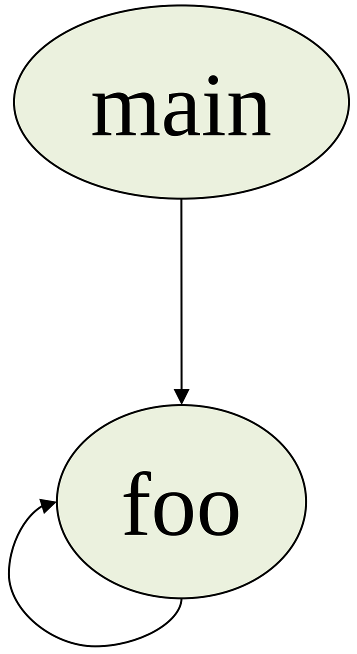
$$\begin{tikzcd} \text{main} \arrow[d] \\ \text{foo} \arrow[d] \\ \text{atoi} \end{tikzcd}$$

Function Call Graph (3)

- Consider the following code

```
int main (){
  …
  y=foo(5);
  …
}
```

```
int foo (int y){
  int x=y+3;
  if (x-5= 0)
    x=foo(x-1);
  return x;
}
```

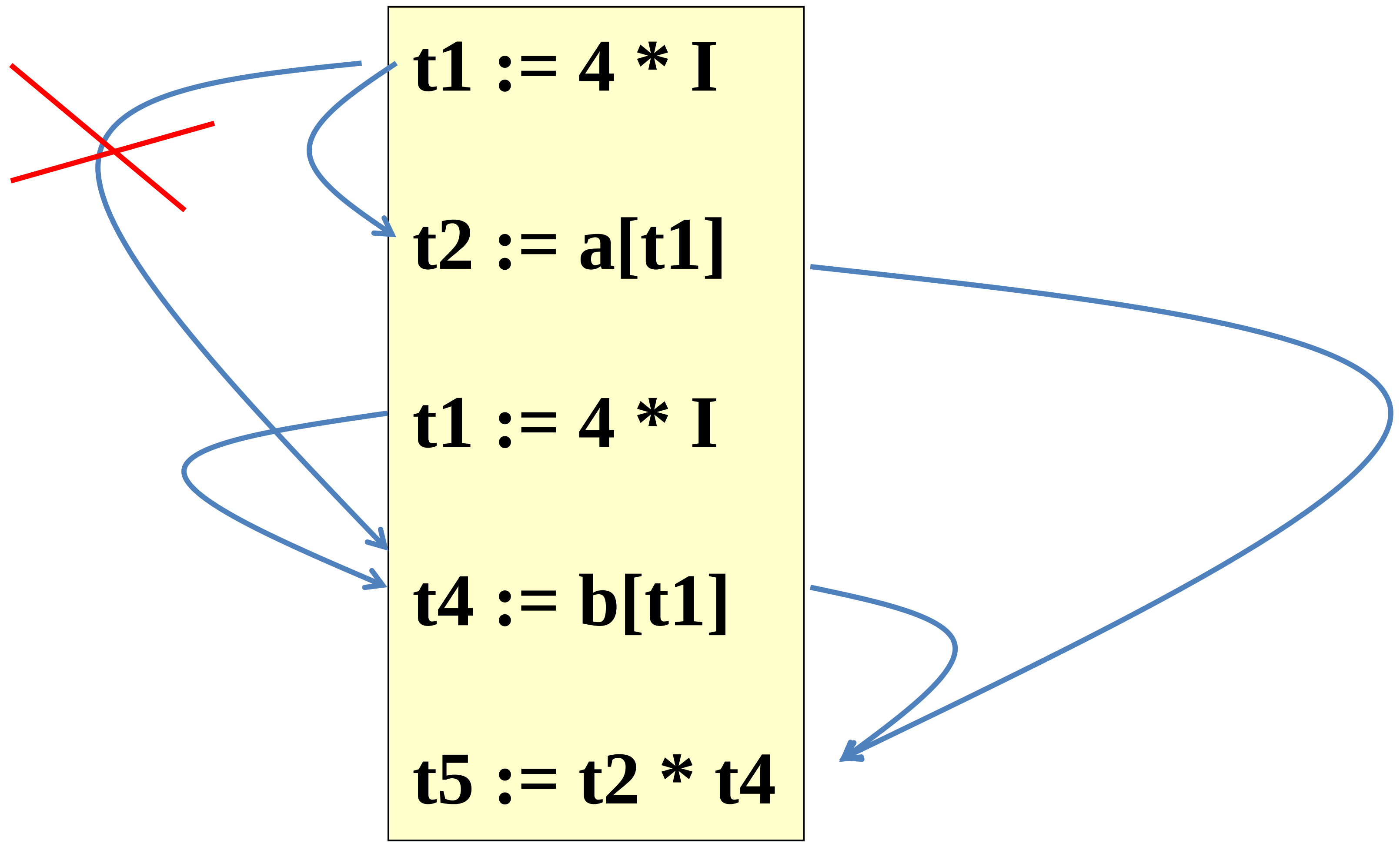


Data Flow

- It is said that there is a data flow dependency between an instruction *i1* and an instruction *i2* if *i1* produces a result read by *i2*
- Data dependency was seen in the previous course
- Detecting data flow dependencies is undecidable in the general case:
    - Existence of pointers (aliasing)
    - Existence of brnaches (conditions not known)

Data Flow (2)

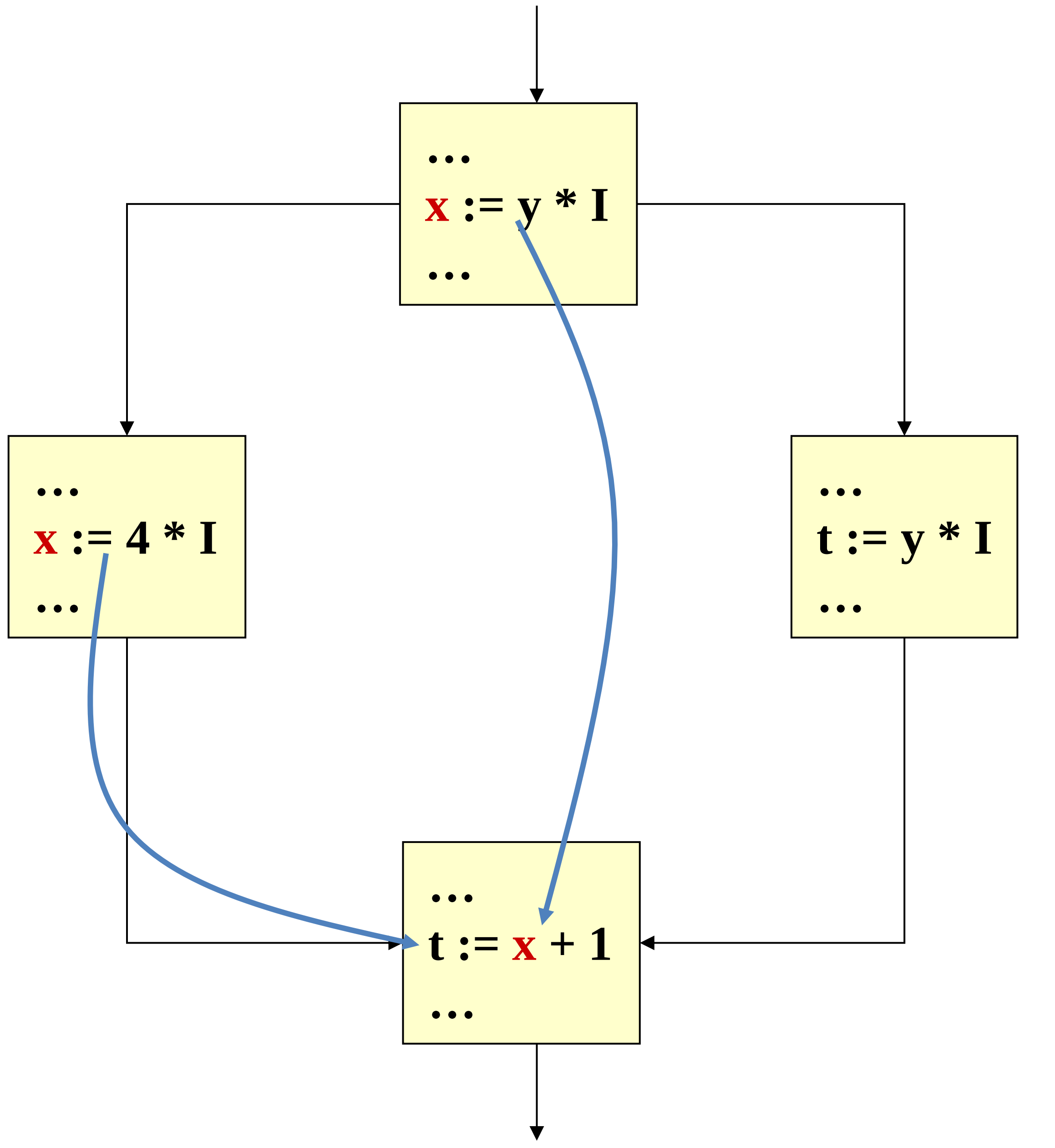
- Consider a sequence of statements below
- Where are the dependencies?

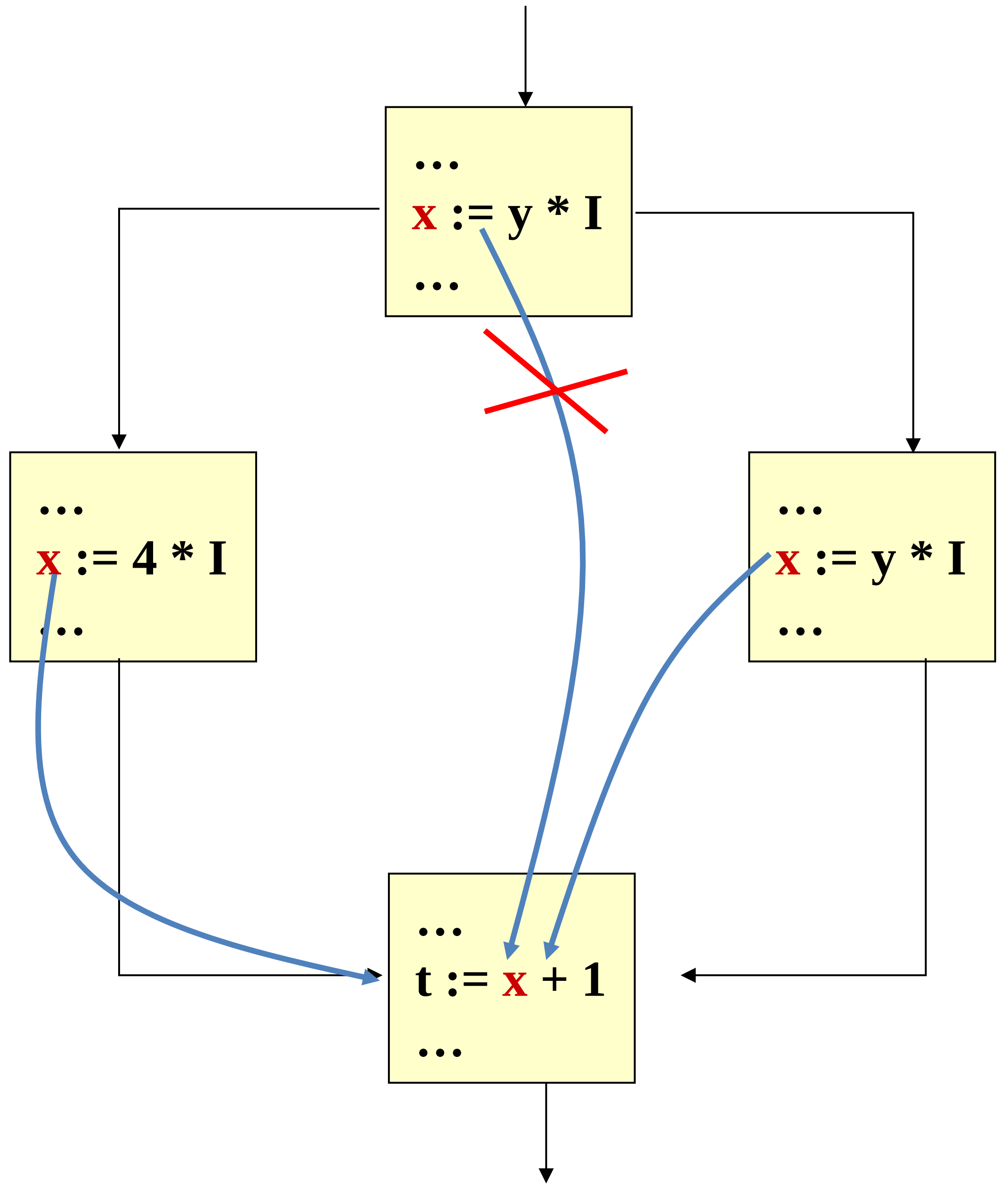


```
t1 := 4 * I
t2 := a[t1]
t1 := 4 * I
t4 := b[t1]
t5 := t2 * t4
```

Data Flow (3)

- What about scalars between basic blocks?





Examples

- Example 1
  ```
  S1: a=1;
  S2: b=1;
  ```
- Statements are independent

- Example 2
  ```
  S1: a=1;
  S2: b=a;
  ```
- Dependent (*true (flow) dependence*)
  - Second is dependent on first
  - Can you remove dependency?

- Example 3
  ```
  S1: a=f(x);
  S2: a=b;
  ```
- Dependent (*output dependence*)
  - Second is dependent on first
  - Can you remove dependency? How?

- Example 4
  ```
  S1: a=b;
  S2: b=1;
  ```
- Dependent (*anti-dependence*)
  - First is dependent on second
  - Can you remove dependency? How?

True Dependence and Anti-Dependence

- Given statements S1 and S2,

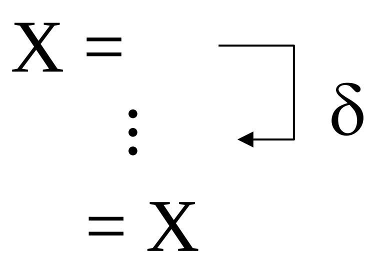
  S1;

  S2;

- S2 has a *true (flow) dependence* on S1

  if and only if

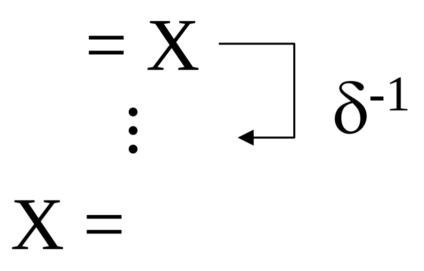
  S2 reads a value written by S1



- S2 has a *anti-dependence* on S1

  if and only if

  S2 writes a value read by S1



Output Dependence

- Given statements S1 and S2,

  S1;

  S2;

- S2 has an *output dependence* on S1

  if and only if

  S2 writes a variable written by S1

$$\begin{tikzcd}
X = \arrow[dd, bend left, "\delta^0"] \\
\vdots \\
X =
\end{tikzcd}$$

- Anti- and output dependences are “name” dependencies
  - Are they “true” dependences?
- How can you get rid of output dependences?
  - Are there cases where you can not?

Statement Dependency Graphs

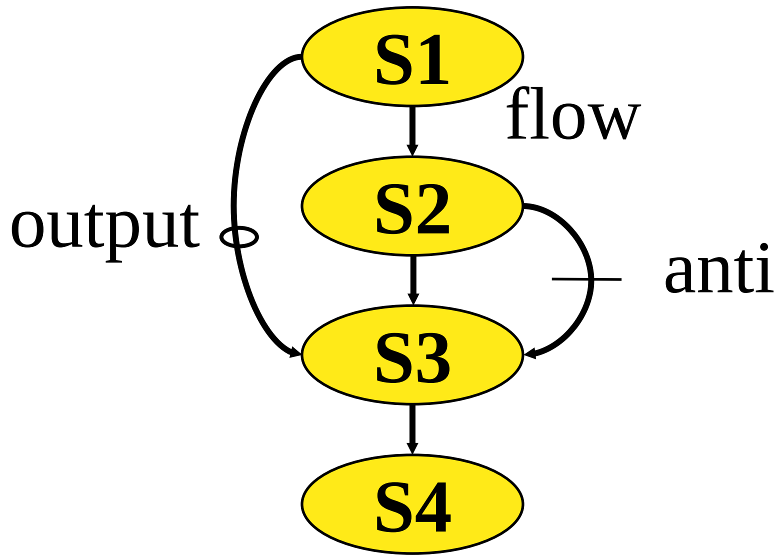
- Can use graphs to show dependence relationships
- Example

  S1: a=1;

  S2: b=a;

  S3: a=b+1;

  S4: c=a;



- $S_2 \ \delta \ \ S_3$: S_3 is flow-dependent on S_2
- $S_1 \ \delta^0 \ S_3$: S_3 is output-dependent on S_1
- $S_2 \ \delta^{-1} \ S_3$: S_3 is anti-dependent on S_2

When can statements execute in parallel?

- Statements S1 and S2 can execute in any order (i.e., at the same time) if and only if there are *no dependences* between S1 and S2
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be remove by modifying the program
  - Rearranging statements
  - Eliminating statements

How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
  - IN(S) : set of memory locations (variables) that may be used in S
  - OUT(S) : set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified
- As such, the sets can be conservatively large

IN / OUT Sets and Computing Dependence

- Assuming that there is a path from S1 to S2 , the following shows how to intersect the IN and OUT sets to test for data dependence

$$out(S_1) \cap in(S_2) \neq \emptyset \quad S_1\ \delta\ S_2 \quad \text{flow dependence}$$

$$in(S_1) \cap out(S_2) \neq \emptyset \quad S_1\ \delta^{-1}\ S_2 \quad \text{anti-dependence}$$

$$out(S_1) \cap out(S_2) \neq \emptyset \quad S_1\ \delta^{0}\ S_2 \quad \text{output dependence}$$

Nested Loops

- We are familiar with loops
  - Repetitive execution of code blocks
- *Nested loops*
  - Loops within loops
  - *Perfectly nested loops* are nested loops that do not break out of their natural loop execution
- Application performance
  - It is in loop nests that applications spend the most time
  - This is also where the data are most accessed

Dependency in a Loop Nest

- When dealing with nested loops, it can be challenging to determine dependencies

```
for (i=1; i<n; i++) {
    for (j=3; j<m-1; j++) {
        A[i,j+1]=…  ;
                                  ?
        …
        …= A[2j, i] ;
    }
}
```

Loop Unrolling

- Loops can be *unrolled* into separate statements / iterations to expose dependencies

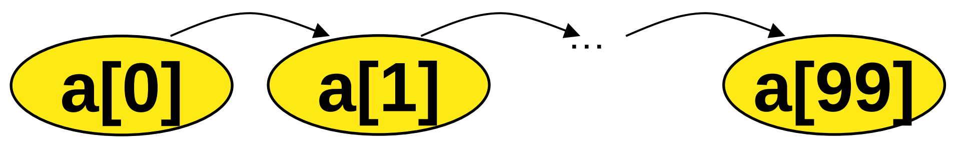
```
for (i=0; i<100; i++) {
    S1: a[i] = i;
}
Unrolled becomes:
a[0] = 0;
a[1] = 1;
…
a[99] = 99;
```

```
for (i=0; i<100; i++) {
    S1: a[i] = i;
    S2: b[i] = 2*i;
}
Unrolled becomes:
a[0] = 0; b[0] = 0;
a[1] = 1; b[1] = 2;
…
a[99] = 99; b[99] = 198;
```

Loops with Dependencies

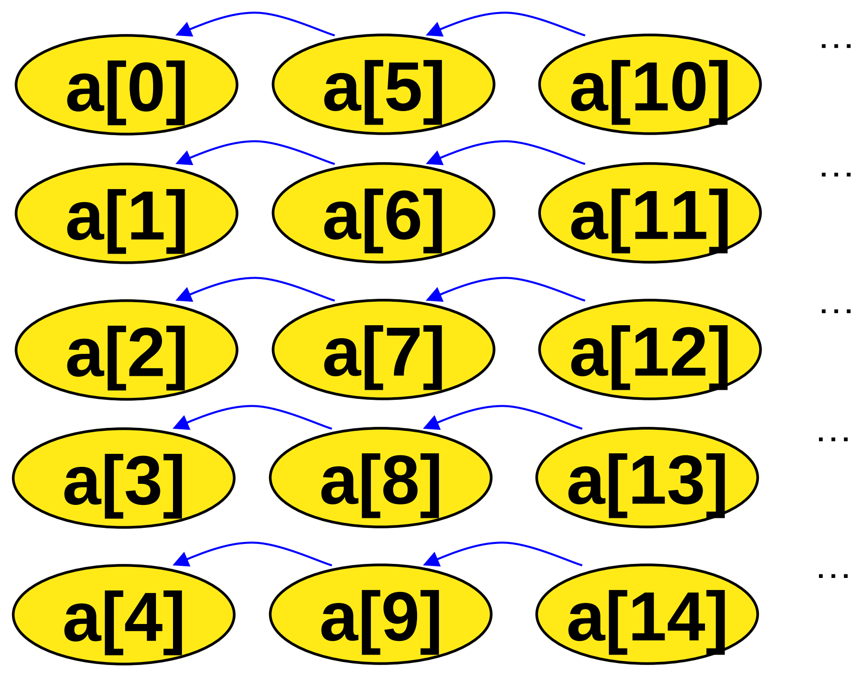
Case 1:

```
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;
```



Case 2:

```
for (i=5; i<100; i++)
    a[i-5] = a[i] + 100;
```



- Dependencies?
  - What type?
- *Iteration space* is the unrolled data flow graph

Another Loop Example

```
for (i=1; i<100; i++)
    a[i] = f(a[i-1]);
```

- Dependencies?
  - What type?
- Even though we do not know the return value of the function call, there is still a dependency from one iteration to the next

Loop Dependencies

- A *loop-carried* dependence is a dependence that is present only if the statements are part of the execution of a loop (i.e., between two statements instances in two different iterations of a loop)
- Otherwise, it is *loop-independent*, including between two statements instances in the same loop iteration
- Loop-carried dependences can prevent loop iteration parallelization
- The dependence is *lexically forward* if the source comes before the target or *lexically backward* otherwise
  - Use *loop unrolling* to see

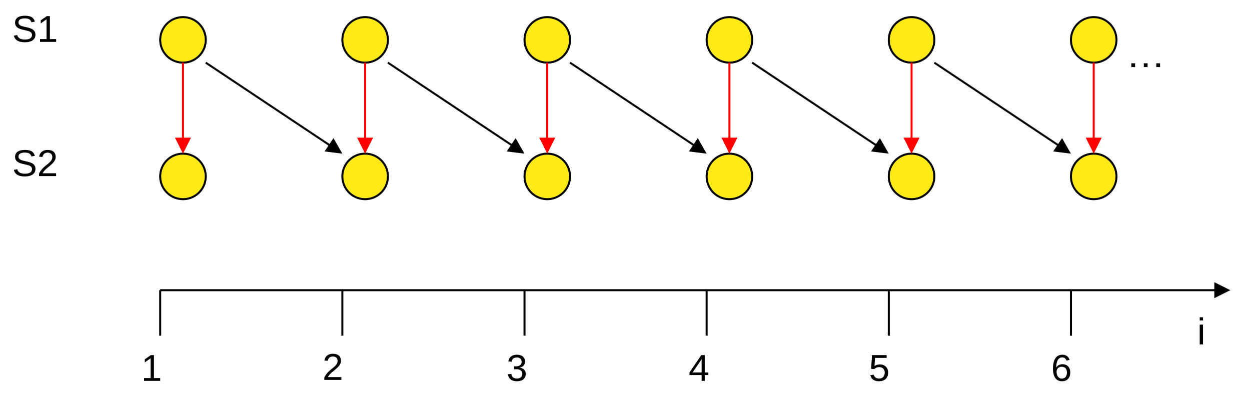
Loop Dependence Example

```
for (i=0; i<100; i++)
    a[i+10] = f(a[i]);
```

- Dependencies?
  - Between a[10], a[20], …
  - Between a[11], a[21], …
- These code blocks (sequences of statements) are separable

Dependences Between Iterations

```
for (i=1; i<100; i++) {
    S1: a[i] = …;
    S2: … = a[i-1];
}
```



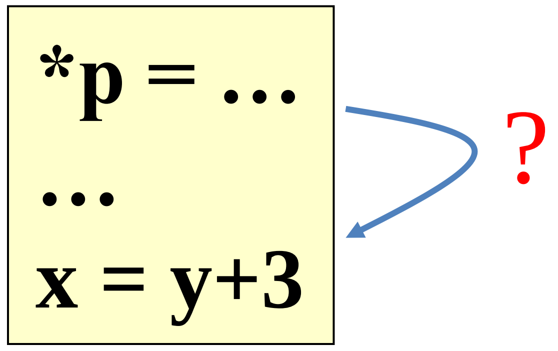
- Dependencies?
  - Between a[i] (S1) and a[i-1] (S2)

Data Dependencies in a Loop Nest

- In general, it is not easy to analyze dependencies in a loop nest because they can go between loops
- We can try to draw the dependency graph
- It is a cyclic graph that models dependencies between instructions of different iterations
- Arcs contain precise information to know which instances of statements are dependent
- The loop nest transformations must ensure that no dependency is violated

Data Dependencies with Pointers

- Using pointers in programs obfuscate data dependencies
- Pointers are aliases
- Do not know what they point to at the time of program analysis
- In the following example, if p points to y, then there is a flow dependency



```
*p = …
…
x = y+3
```

Sequential Model

- Let’s consider code optimization first with respect to a sequential execution model
  - Without advanced micro-architectural mechanisms
  - Remove from calculation or instructions
- Types of optimizations
  - Local optimizations
    - performs within a base block.
  - Peephole optimizations
    - small number of instructions
    - sliding window concept
  - Global optimizations

Elimination of Common Expressions

- There are places where expressions used in instructions can be optimized (eliminated) in the code by replacing the expression with the result
  - Previously stored in a variable or register
- Having one or more arithmetic expressions, try not to recalculate the same expression

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

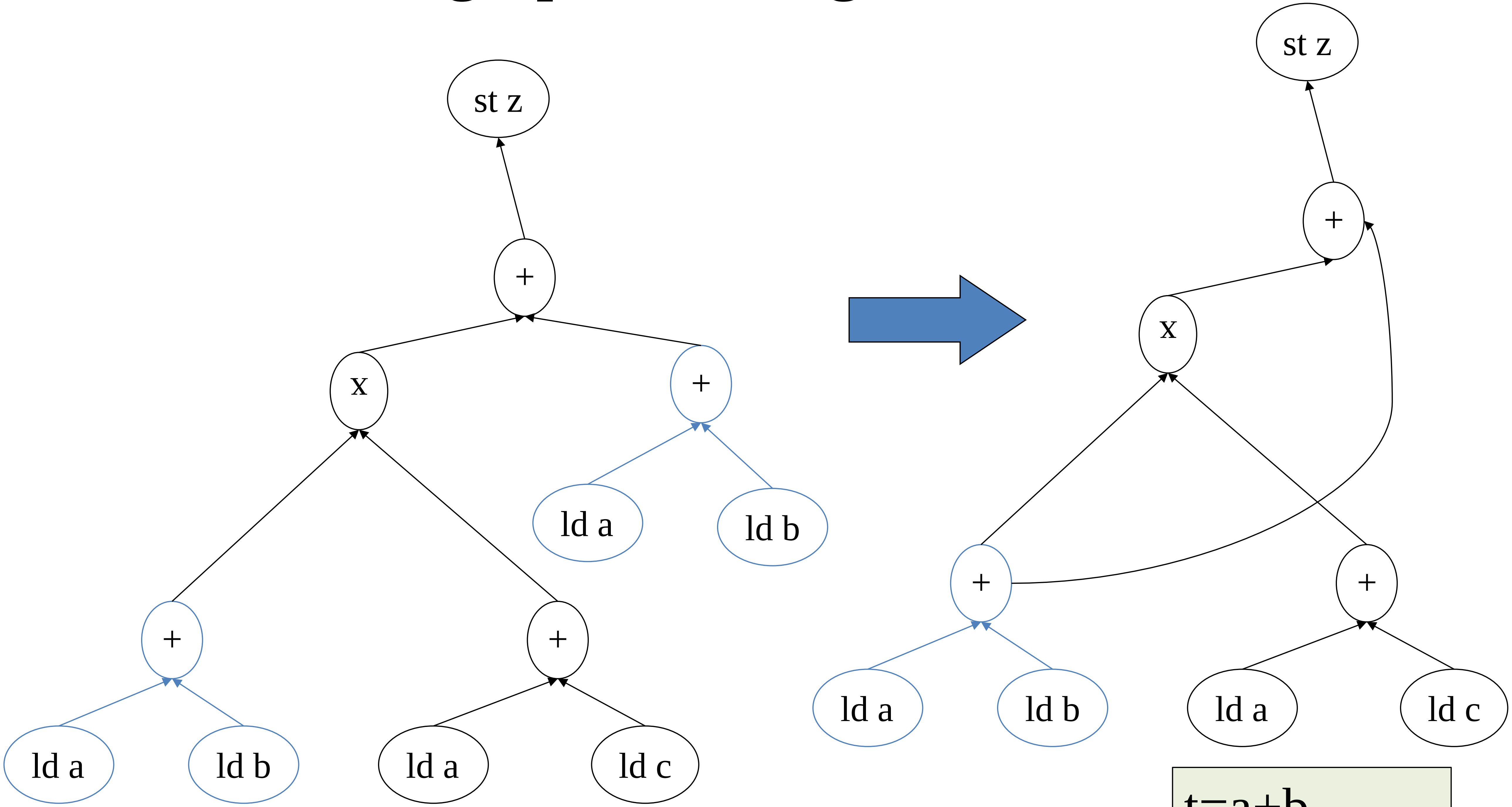
$\Rightarrow$

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

- Common expression elimination (CSE)

Elimination of Common Expressions (2)

- A data flow graph can be generated and modified



z=(a+b)*(a+c)+(a+b)

t=a+b
z=t*(a+c)+t

Elimination of Redundant Storage

- Sometimes usage of storage by instructions is inefficient
  - Identify the cases and eliminate
- In the following, if R0 is not modified between S1 and S2 then it is possible to eliminate S2

```
S1: LOAD   R0, a
/* … */
S2: STORE a, R0
```

$\Longrightarrow$

```
S1: LOAD R0, a
```

Constant Propagation

- If the value of a register is known and constant at a point in the code, the constant can be substituted for the register name

```
debug=0
…
if (debug==1) goto L1
goto L2
L1 : print informations
L2:…
```

$\Rightarrow$

```
if (0==1) goto L1
goto L2
L1 : print informations
L2:…
```

Algebraic Simplifications

- Sometimes it is the case that expressions can be analyzed to determine simpler forms
- Simple algebraic analysis can identify opportunities
- Consider the following

$$\boxed{\begin{array}{l} x+0 \rightarrow x \\ x*1 \rightarrow x \\ x*0 \rightarrow 0 \end{array}}$$

- If *x* is an integer, it is valid to make the transformation
- If *x* is a float, it is valid to make the transformation as long as we can guarantee that the float can not have as infinity value and NaN
  - One can not substitute □ * 0 by 0

Strength Reductions

- Here we consider the architectural operation used to implement the statements
- Principle: replace operations with other less expensive (architectural)
- Depends of course on architecture and micro-architecture
- Some examples

`x**2` $\rightarrow$ `x*x`	multiple less expensive than exponential
`x*2` $\rightarrow$ `x+x`	add less expensive than multiply
`x*2` $\rightarrow$ `x<<1`	shift less expensive than multiply (add)
`x/2` $\rightarrow$ `x>>1`	shift less expensive than divide

Control Flow Optimizations

- Idea is to find shortcuts in control paths

```
    goto L1
    . . .
L1: goto L2
```

$\Longrightarrow$

```
    goto L2
    . . .
L1: goto L2
```

```
    if a < b goto L1
    . . .
L1: goto L2
```

$\Longrightarrow$

```
    if a < b goto L2
    . . .
L1: goto L2
```

```
    goto L1
    . . .
L1: if a < b goto L2
L3:
```

$\Longrightarrow$

```
    if a < b goto L2
    goto L3
    . . .
L3:
```

Inaccessible Code Elimination

- In some cases, code analysis can determine that certain control paths are not possible
  - Can eliminate code along those paths
  - Must make sure they are not accessible on any path

```
#define DEBUG 0
if (DEBUG){
  print informations
}
```

```
if (0==1) goto L1
goto L2
L1 : print informations
L2:…
```

```
if (0==1) goto L1
goto L2
L1 : print informations
L2:…
```

```
#define DEBUG 0
if (DEBUG){
  print informations
}
```

```
if (0==1) goto L1
goto L2
L1 : print informations
L2:…
```

```
if (0==1) goto L1
goto L2
L1 : print informations
L2:…
```

Source code Intermediate code Optimized code

Elimination of Induction Variables

- An induction variable is a scalar variable in a loop that increments by one constant at each iteration
- Sometimes code optimizations can eliminate them

```
for (i1=1, i2=0 ; i1<n; i1++) {
   a[i1]=a[i2]+1;
   i2++;
}
```

```
for (i1=1; i1<n; i1++) {
   a[i1]=a[i1-1]+1;
}
```

Moving Loop Invariants

- Look for opportunities where there is not change in something that we can compute in advance
- If the result of an expression is not modified inside a loop, the expression can be moved outside

Case 1

```
for (i=0; i<n; i++){
 a[i]=x+y+z;
}
```

$\Rightarrow$

```
t=x+y+z
for (i=1; i<n; i++)
 a[i]=t
```

Case 2

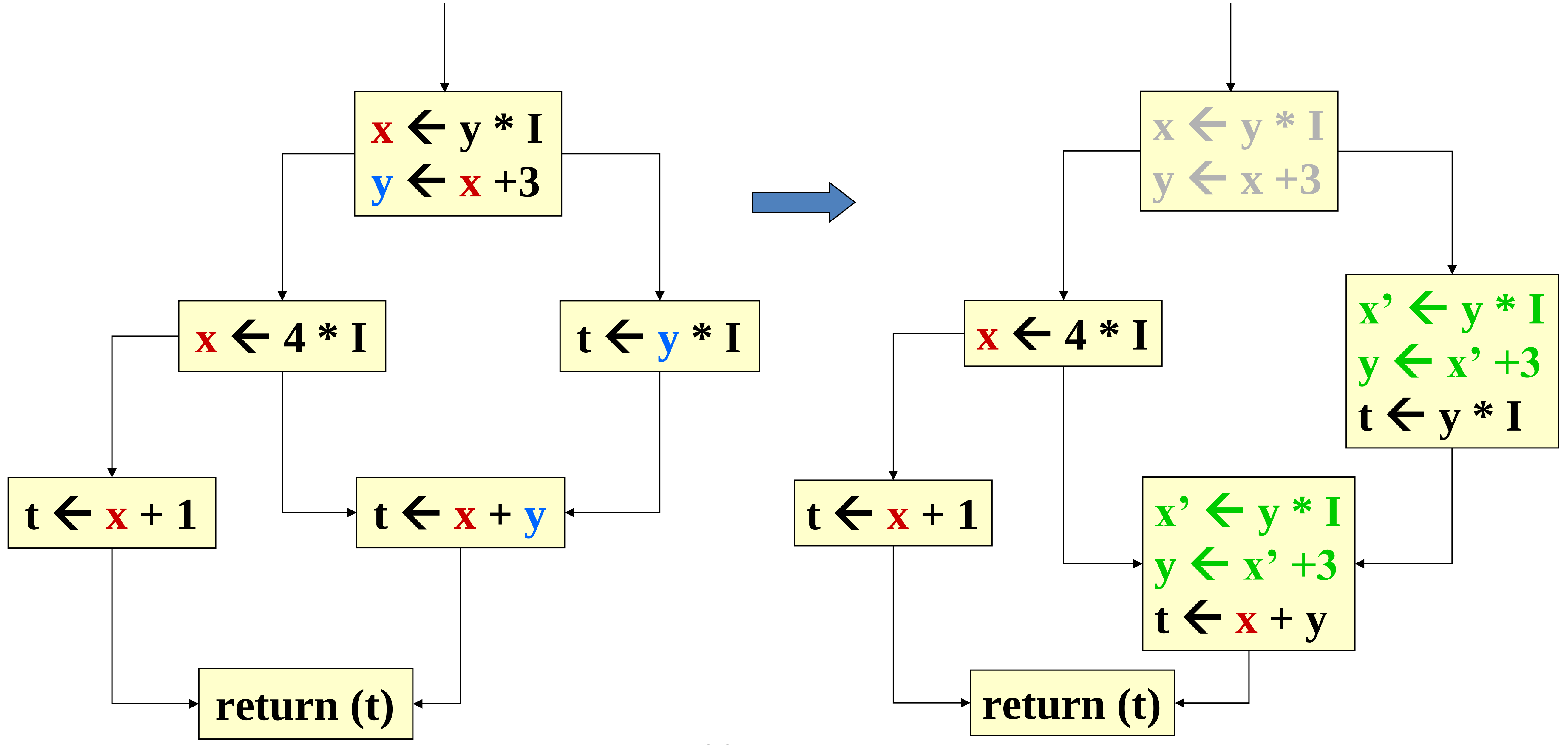
```
for (i=0; i<n; i++){
  if (x<3) a[i]=1;
}
```

$\Rightarrow$

```
if (x<3){
  for (i=1; i<n; i++)
  a[i]=1;
}
```

Code Movement between Basic Blocks

- Optimizations by moving code among basic blocks
  - Could potentially remove certain block altogether



Side effects?

Function Inlining

- Significant performance gains can be obtained simply by replacing functions calls in a program
    - Expands the number of instructions in the code
    - Reduces the overheads and register management

```
int main (){
  …
  y=foo(5);
  …
}
```

```
int foo (int y){
  x=y+3;
  if (x-5 == 0)
    …
  return x;
}
```

→

```
int main (){
  …
  x= 5+3;
  if(x-5 == 0)
    …
  y = x; …
}
```

→

```
int main (){
  …
  x= 8;
  if(8 == 0) …
  y = 8;
  …
}
```

Function Specialization and Versioning

- Functions can be made more efficient by creating specialized versions based on parameters

```
int main (){
  …
  y=foo(5);
  …
}
```

```
int foo (int y){
  x=y+3;
  if (x-5= 0) …
  return x;
}
```

```
int main (){
  …
 y = foo_5();
  …
}
```

```
int foo_5 (){
  return 8;
}
```

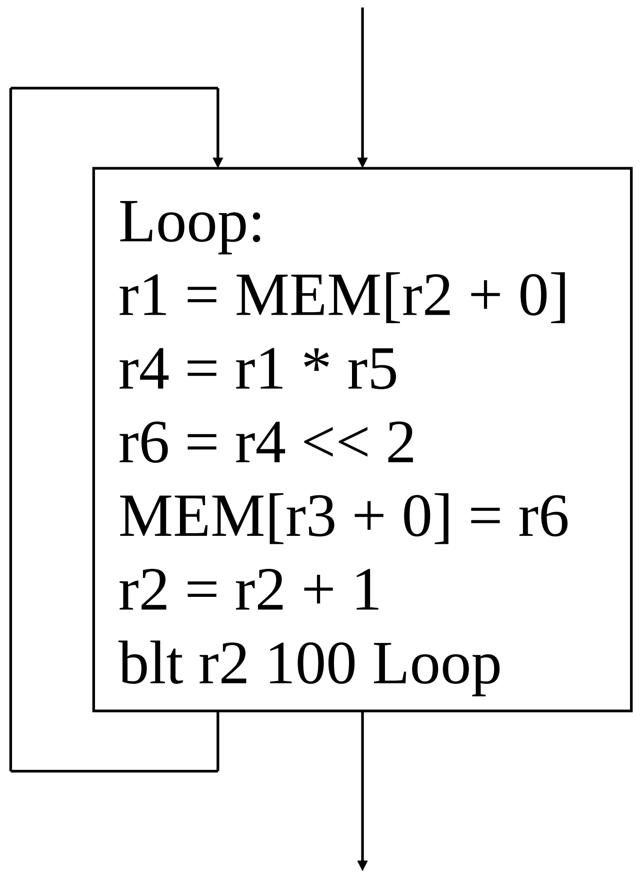
```
int main (){
  …
 y = 8;
  …
}
```

ILP Model

- Parallelism in instructions parallelism
    - Instruction-level parallelism (ILP)
- Granularity of parallelism is at the finest level visible to the architecture
    - Medium granularity: loops iterations, threads, process
    - Larger granularity: processes, applications
- List of transformations to expose more ILP
    - Allow more independent instructions
    - Impacts code generation (scheduling)
    - Depends on the micro-architecture

Loop Unwinding

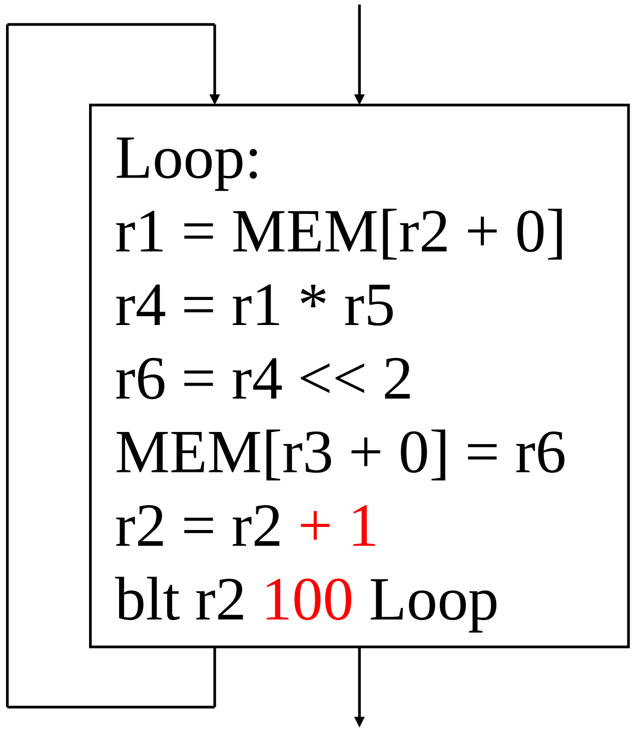
- Duplicate loop body
  - N copies
  - Loop is unrolled N times
  - Exposes parallel execution of independent operations of different iterations
  - Increases ILP extraction potential
- Three variants
  - Unroll a loop with a known number of iterations
  - Unroll a parameterized loop
  - Unwrap a while loop



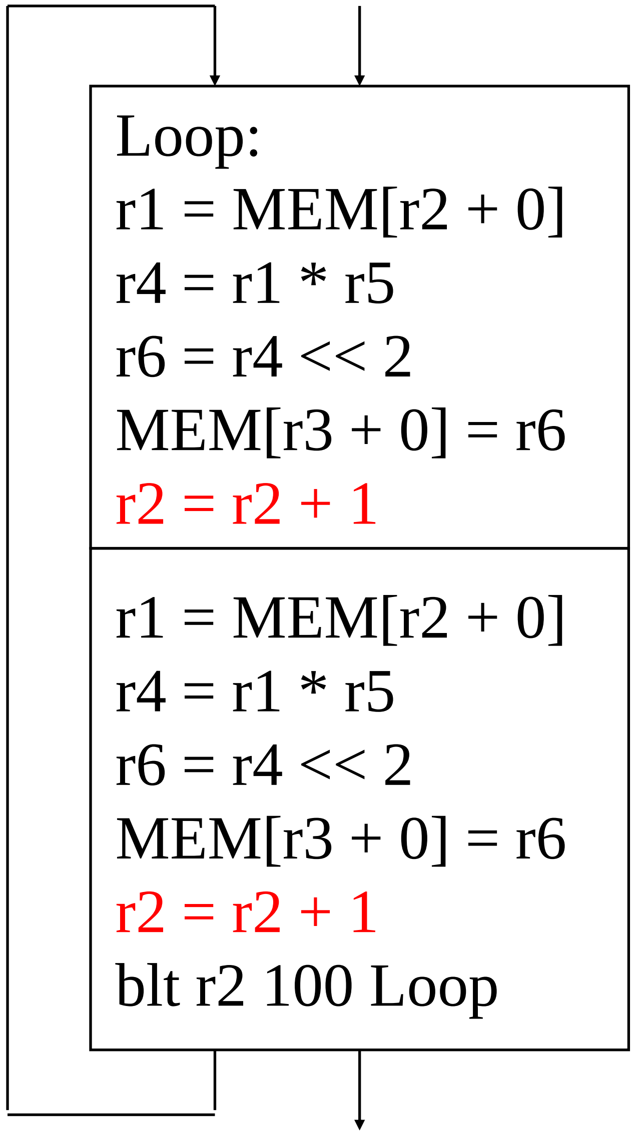
```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```

Loop Unwinding – Type 1

- Number of iterations is known
- R2 is the loop counter
  - Step is 1
  - Initial value 0
  - Final value 100
  - # iterations 100



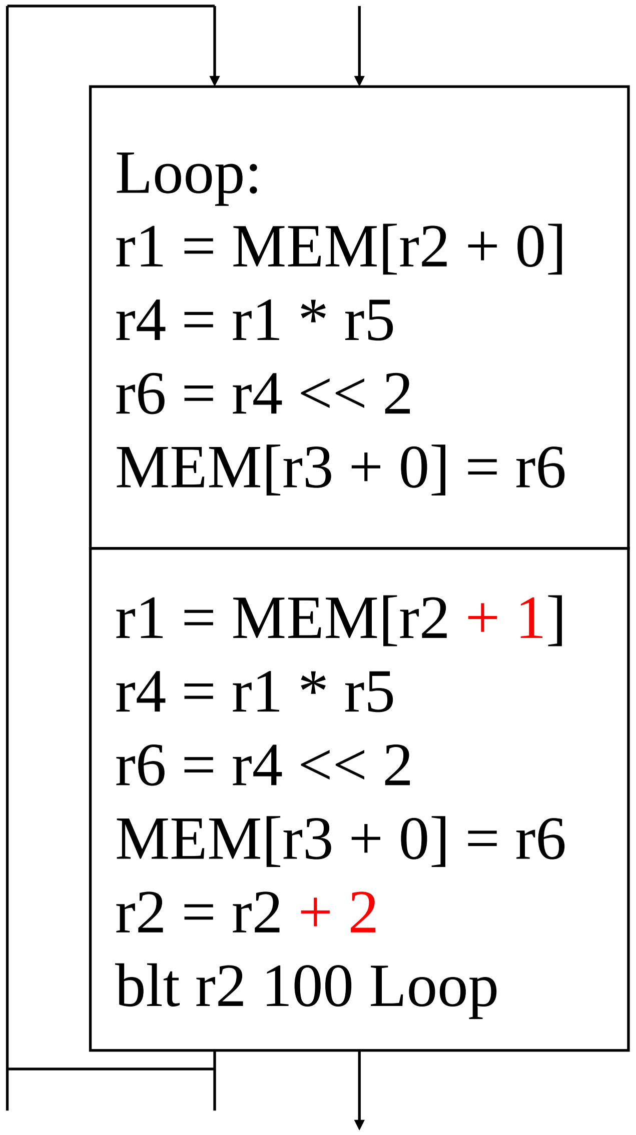
```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```



```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1

r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```

Remove the branch of N-1 first iterations



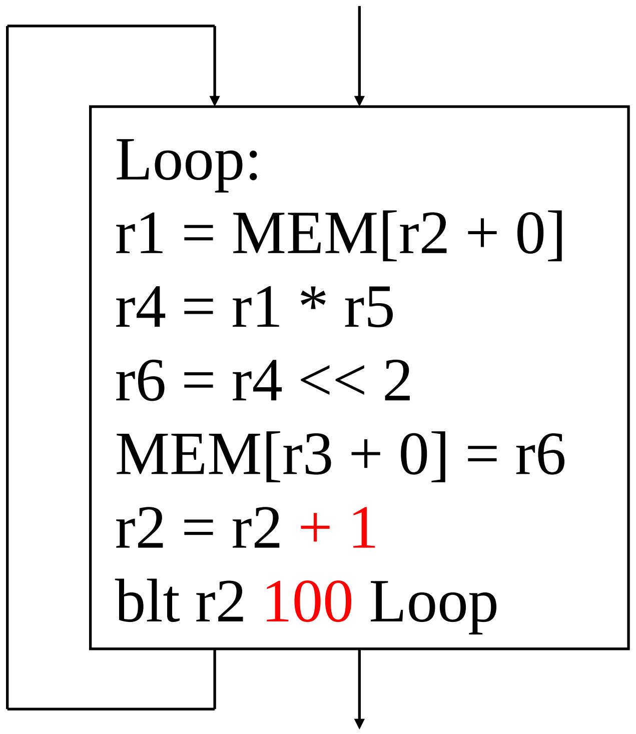
```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6

r1 = MEM[r2 + 1]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 2
blt r2 100 Loop
```

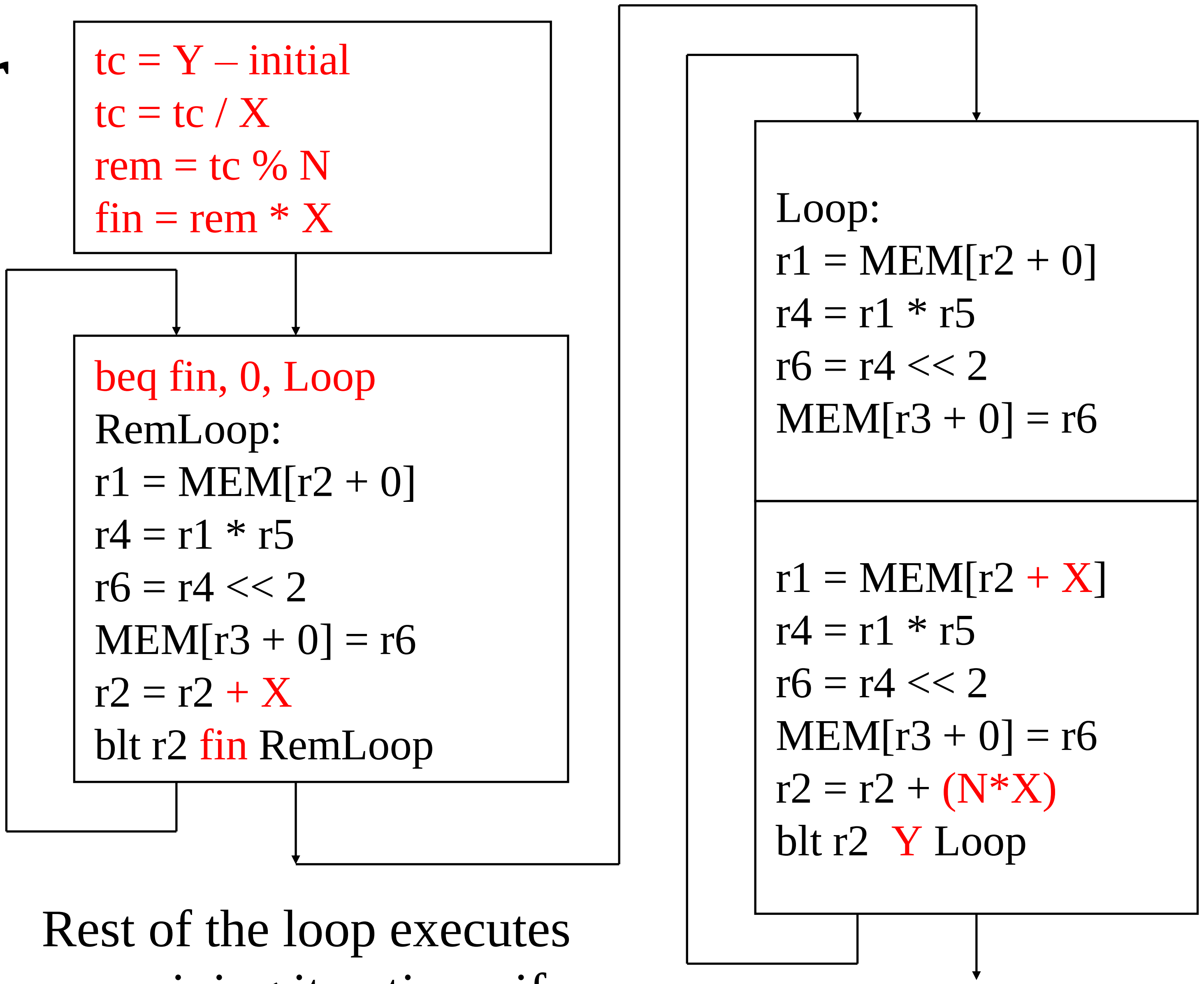
Removing the increment of r2 and changing the last increment

Loop Unwinding – Type 2

- Number of iterations is not known
- R2 is the loop counter
  - Increment = X?
  - Initial value =?
  - Final value = Y?
  - # iterations = ?



```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```



```
tc = Y – initial
tc = tc / X
rem = tc % N
fin = rem * X
```

```
beq fin, 0, Loop
RemLoop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + X
blt r2 fin RemLoop
```

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6

r1 = MEM[r2 + X]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + (N*X)
blt r2  Y Loop
```

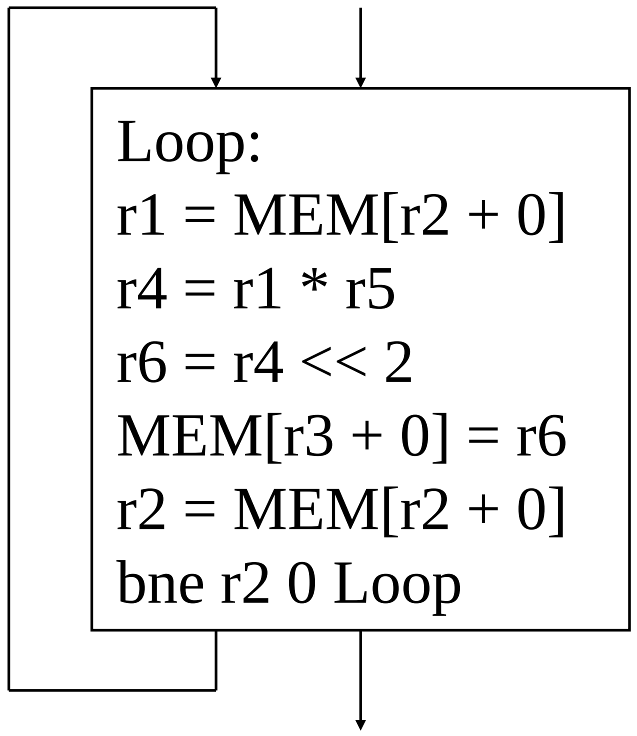
Rest of the loop executes remaining iterations, if the number of iterations is not a multiple of n

Loop unrolled as in Type 1, and necessarily executes a multiple of n

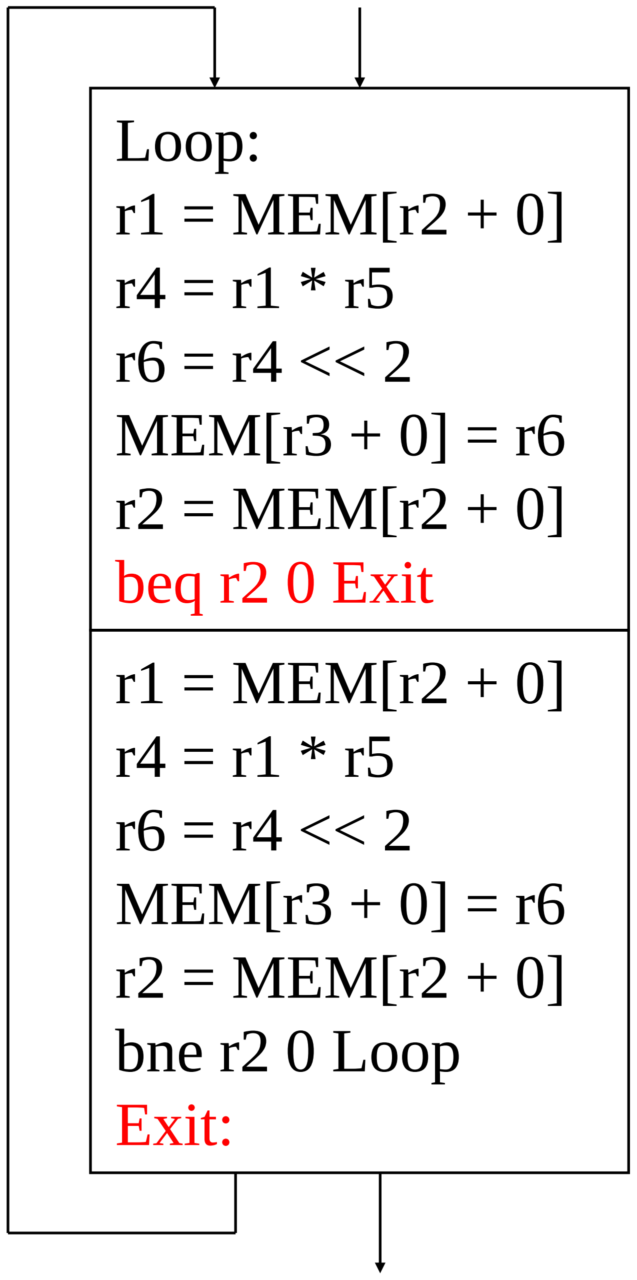
Loop Unwinding – Type 3

- Number of iterations is not known
- While loop
- Static

while (r2!=0)…



```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = MEM[r2 + 0]
bne r2 0 Loop
```



```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = MEM[r2 + 0]
beq r2 0 Exit
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = MEM[r2 + 0]
bne r2 0 Loop
Exit:
```

Duplicate loop bodies and keep internal branches, but convert into sort of breaks

Summary of Loop Unwinding

- Purpose of loop unwinding is to expose ILP in base blocks (multiple iterations instead of just one)
- Often followed by a local scheduling of instructions
  - Sometimes call *code compaction*
- How much unwinding to do?
  - Calculated empirically only
  - Too little degree of unwinding would not allow maximum exploitation of the ILP
  - Too much degree of unwinding increases code size and causes instruction cache defects

Loop Fusion

- Idea is to take several consecutive loops and merge them into a single loop
- Effect is an increase in the number of instructions in the body of the loop
  - Exposes more opportunities for finding independent instructions to improve ILP
  - Also possibly affects the locality of the data

```
for i=1 to N do
    A[i] = B[i] + D[i]
endfor
for i=1 to N do
    C[i] = B[i] / 2
endfor
```

$\Longrightarrow$

```
for i=1 to N do
    A[i] = B[i] + D[i]
    C[i] = B[i] / 2
endfor
```

Is this transformation legal?

Requirements for Loop Fusion

- In order to be merged, two loops must be structurally compatible
  - They have the same iteration space
  - They must be adjacent, or moved to become adjacent
  - One must be able to use a common loop counter
- Preliminary transformations could make two loops compatible
  - Loop peeling, cutting of iteration space, ...
- If two loops are compatible, the merge must also be legal
  - Neither violate nor add a dependency of data

Loop Peeling

- Remove the first or last iterations of the loop
- Put into a separate code

```
for (i=0;i<8;i++) {
    A[i] = (X+Y)*B[i]
}
```

$\Rightarrow$

```
A[0] = (X+Y)*B[0]
for (i=1;i<8;i++) {
    A[i] = (X+Y)*B[i]
}
```

```
for (i=0;i<N;i++) {
    A[i] = (X+Y)*B[i]
}
```

$\Rightarrow$

```
if (N >= 1) {
    A[0] = (X+Y)*B[0]
    for (i=1;i<N;i++) {
        A[i] = (X+Y)*B[i]
    }
}
```

Index Set Splitting (Split Iteration Space)

- Divide the iteration space into two parts

```
for i=1 to 100 do
    A[i] = B[i] + C[i]
    if i > 10 then
        D[i] = A[i] + A[i-10]
    endif
endfor
```

$\Rightarrow$

```
for i=1 to 10 do
    A[i] = B[i] + C[i]
endfor
for i=11 to 100 do
    A[i] = B[i] + C[i]
    D[i] = A[i] + A[i-10]
endfor
```

Loop Merging with Transformations

- Find opportunities to make the loops compatible
- Apply peeling, splitting, and other transformations

```
for i=1 to 99 do
  A[i] = B[i] + 1
endfor
for i=1 to 150 do
  C[i] = B[i+1] * 2
endfor
```

$\Rightarrow$

```
for i=1 to 99 do
   A[i] = B[i] + 1
endfor
for i=1 to 99 do
   C[i] = B[i+1] * 2
endfor
for i=100 to 150 do
   C[i] = B[i+1] * 2
endfor
```

Après découpage de l’espace d’itérations de la deuxième boucle

$\Rightarrow$

```
for i=1 to 99 do
   A[i] = B[i] + 1
   C[i] = B[i+1] * 2
endfor
for i=100 to 150 do
   C[i] = B[i+1] * 2
endfor
```

Après fusion des deux premières boucles

Loop Merging with Transformations (2)

- Find opportunities to make the loops compatible
- Apply peeling, splitting, and other transformations

```
for i=1 to 99 do
    A[i] = B[i] + 1
endfor
for i=2 to 99 do
    C[i] = C[i-1] * 2
endfor
```

```
A[1] = B[1] + 1
for i=2 to 99 do
    A[i] = B[i] + 1
endfor
for i=2 to 99 do
    C[i] = C[i-1] * 2
endfor
```

Après pelage de la première boucle

```
A[1] = B[1] + 1
for i=2 to 99 do
  A[i] = B[i] + 1
  C[i] = C[i-1] * 2
endfor
```

Après fusion de boucles

Illegal Loop Fusion

- Must always be careful to avoid violating dependencies that exist in the program
- Loop transformations must preserve dependencies

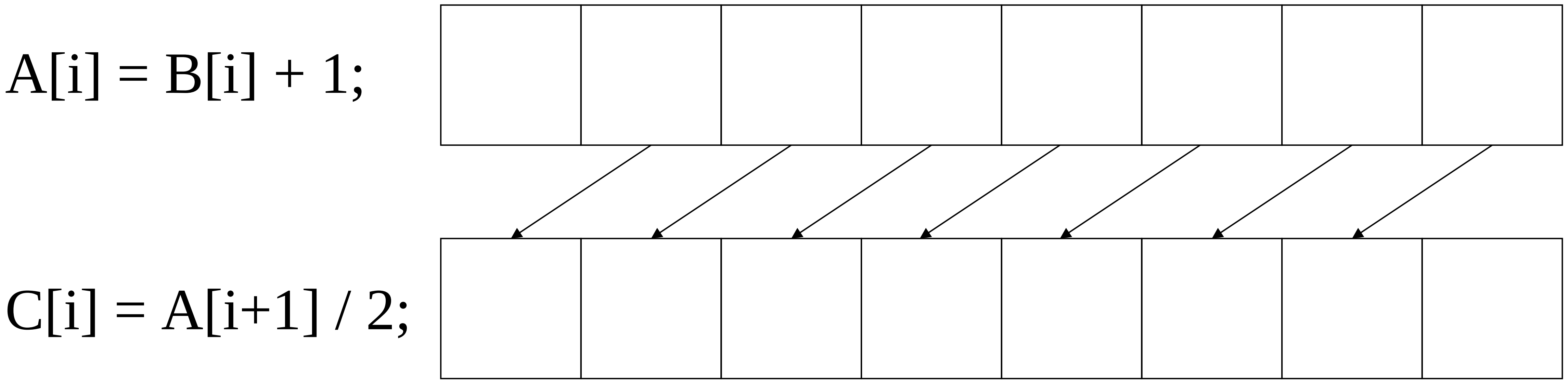
```
for i=1 to N do
    A[i] = B[i] + 1
endfor
for i=1 to N do
    C[i] = A[i+1] / 2
endfor
```

```
for i=1 to N do
    A[i] = B[i] + 1
    C[i] = A[i+1] / 2
endfor
```

The flow dependencies from the original loops are not preserved

Illegal Loop Fusion (2)

- Dependencies that prevent fusion are across loops



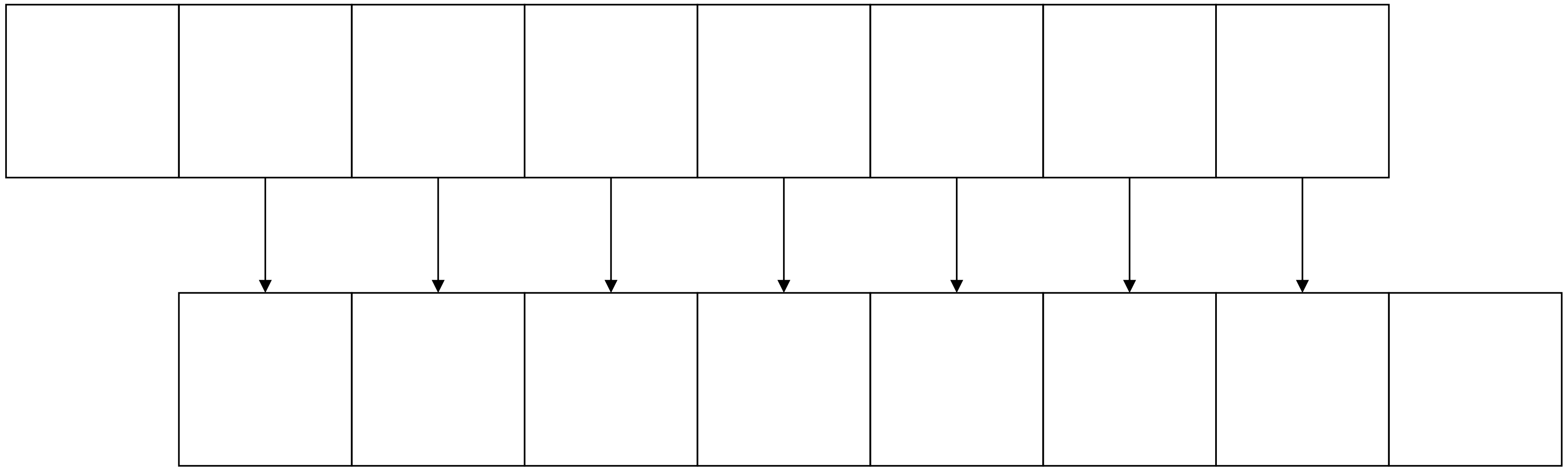
- Can anything be done?

Illegal Loop Fusion (3)

- Try to align the access patterns

A[i] = B[i] + 1;

C[i] = A[i+1] / 2;



- Does it help?

Legal Fusion after Preparations

```
for i=1 to N do
    A[i] = B[i] + 1
endfor
for i=1 to N do
    C[i] = A[i+1] / 2
endfor
```

Loop peeling

```
A[1]= B[1] + 1
for i=2 to N do
    A[i] = B[i] + 1
endfor
for i=1 to N-1 do
    C[i] = A[i+1] / 2
endfor
C[N] = A[N+1] / 2
```

Align interation spaces

```
A[1]= B[1] + 1
for i=1 to N-1 do
    A[i+1] = B[i+1] + 1
endfor
for i=1 to N-1 do
    C[i] = A[i+1] / 2
endfor
C[N] = A[N+1] / 2
```

Legal fusion possible

```
A[1]= B[1] + 1
for i=1 to N-1 do
    A[i+1] = B[i+1] + 1
    C[i] = A[i+1] / 2
endfor
C[N] = A[N+1] / 2
```

Dependencies are preserved

Limitations with Loop Transformations

- They work well on regular (intensive) computational codes, with a large number of iterations

- Limitations:
  - Nests of non-perfect loops
  - Some "short" loops (few iterations)

Scheduling Techniques

- Principle
  - Optimize for back-end of compilation
  - Generation of final instructions with a particular order (linearization)
  - Support better instruction execution and memory usage
- Field of research
  - Acyclic scheduling of tasks
  - Represent as a directed acyclic graph (DAG)
- Complexity issues
  - Having a DAG, find a minimal scheduling under resource constraints
  - NP-complete problem
- There are many heuristics
  - Best known are the variants of the scheduling by list

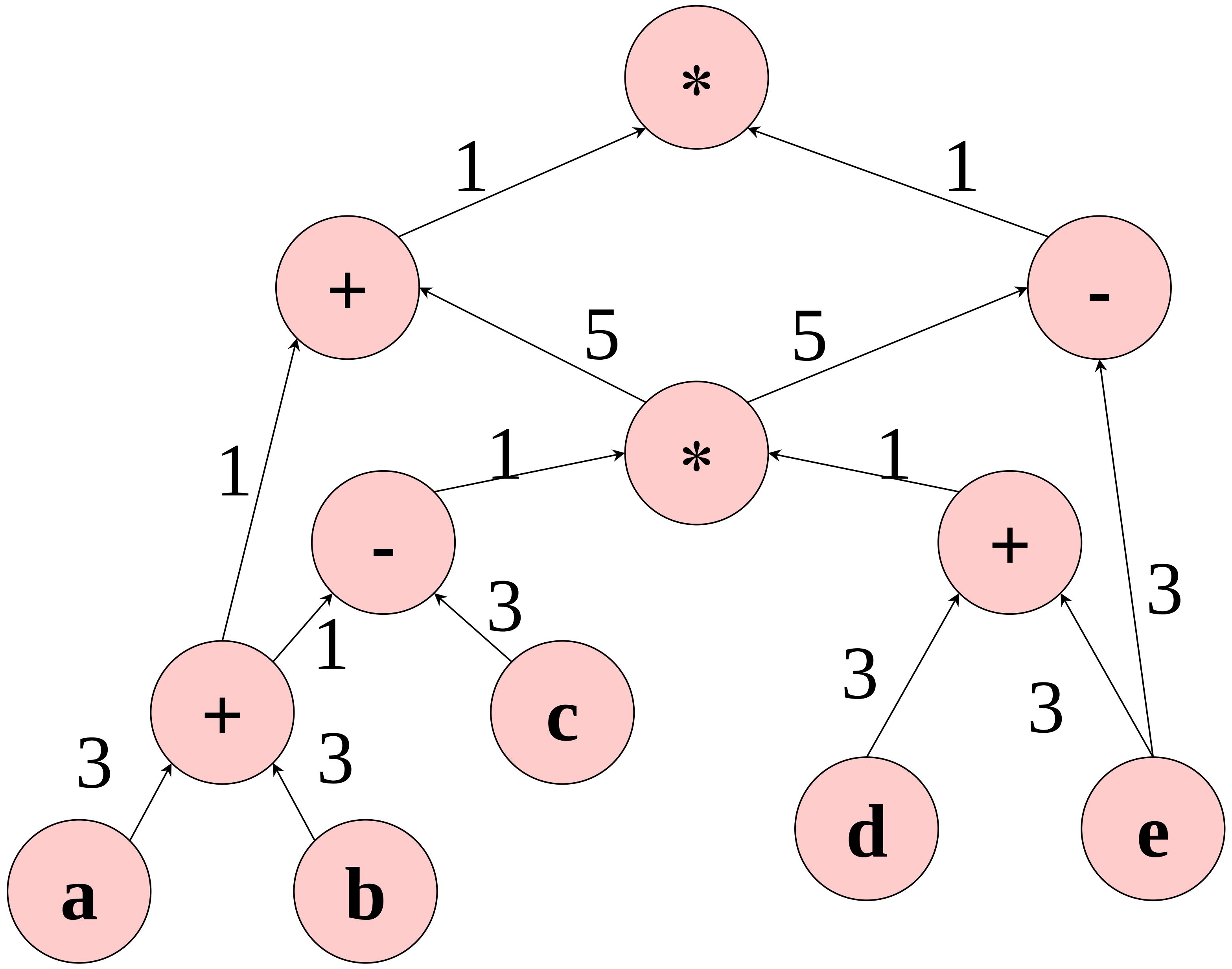
Scheduling by List

- Objective is to keep processor busy with work
- A list contains all the tasks (nodes) that are *ready*
  - Have all of the operands needed
  - Initially, this list contains the beginning nodes
  - This list is sorted by a priority function
- **Step 1**: Choose a ready task to run at the current time
  - Which task to choose?
  - Several variants of *priority*
    - one that has more / fewer leads
    - one that is on the critical path
    - one that consumes a critical resource
- **Step 2**: If the resource that calculates the priority task is free, then schedule the task on the resource at the current time
- **Step 3**: Update Task List Ready
- Mathematical property of heuristic (any variant)
  - Scheduling by list produces a result at worst 2x the optimum

Scheduling by List Example

- Consider a DAG
  - A node is an operation
  - An arc is a data dependency
  - Arc labeled with the time between two tasks
- Consider a processor
  - Degrees of maximum parallelism (issue width): 4 instructions per cycle
  - Resource constraints (per cycle): 1 MEM, 2 ALU, 1 branch
  - Latencies: MEM (3cycles), add / under (1cycle), mult (5cycles)

Scheduling by List Example (2)



1 VLIW = 1 cycle static

Number of operations = 12

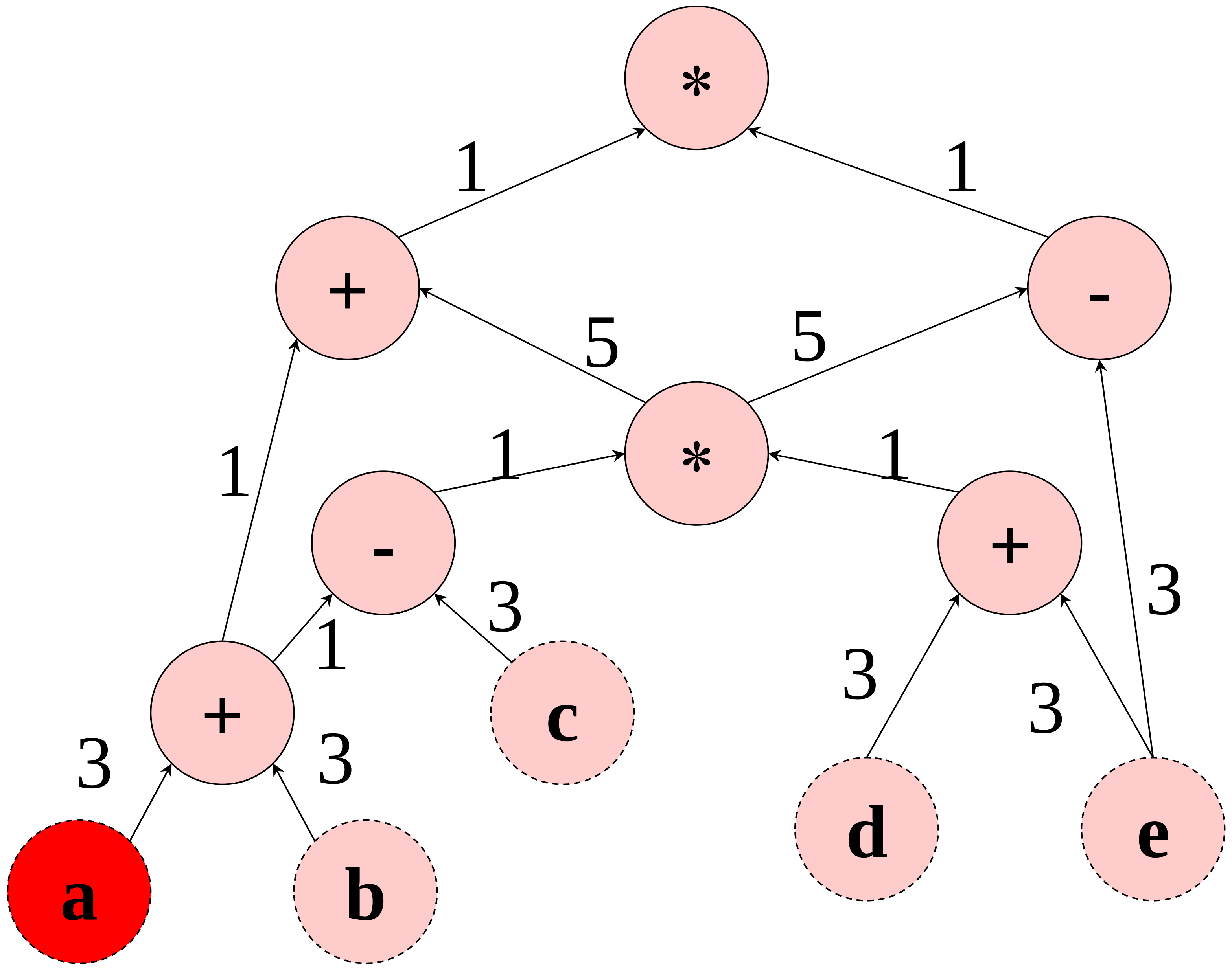
Critical path = 11

Maximum CPI = 12/11

Maximum Antichain = 5

List of ready tasks: a,b,c,d,e

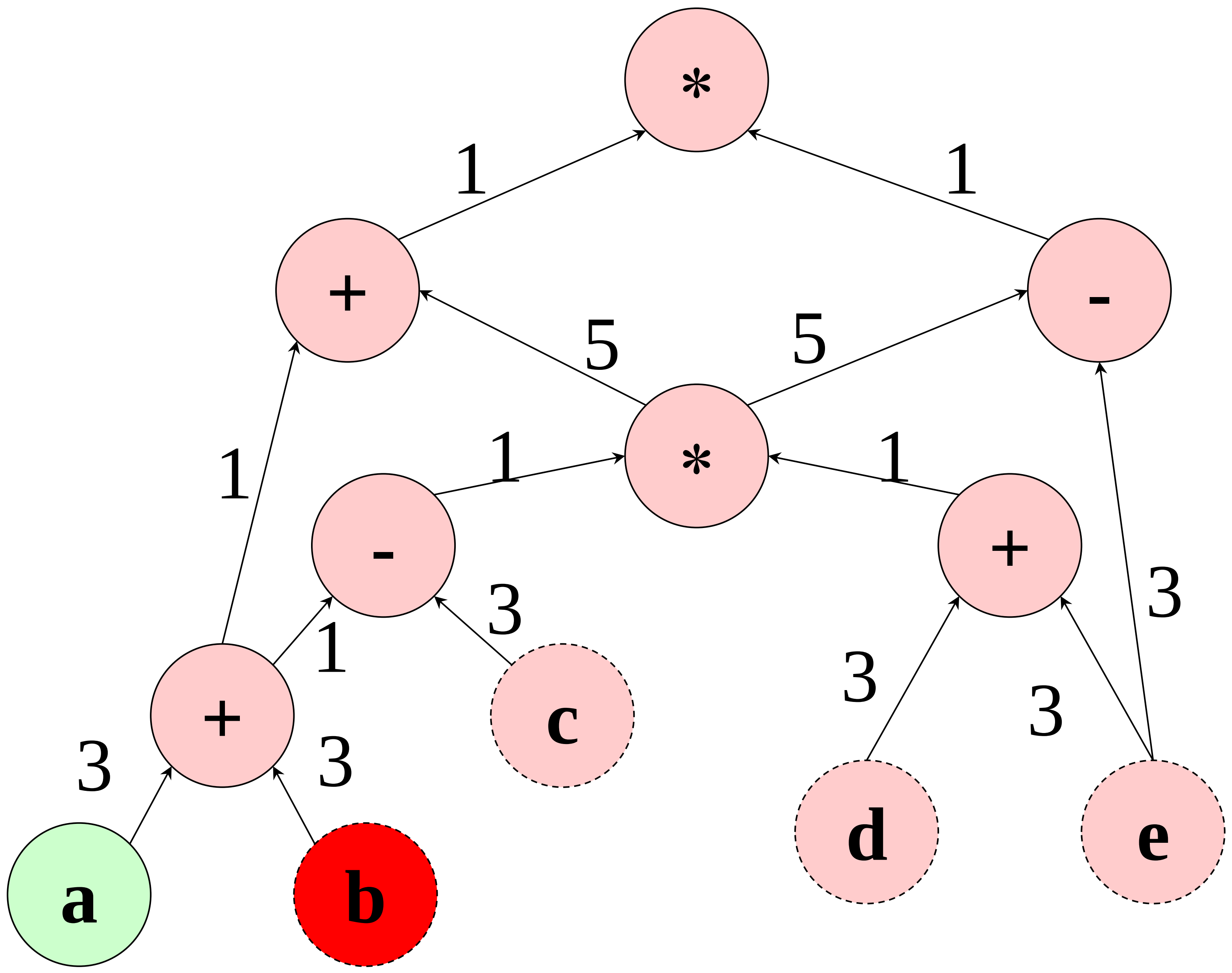
Scheduling by List Example (3)



```
vliw 0: ld R0, a;
```

List of ready tasks: a,b,c,d,e

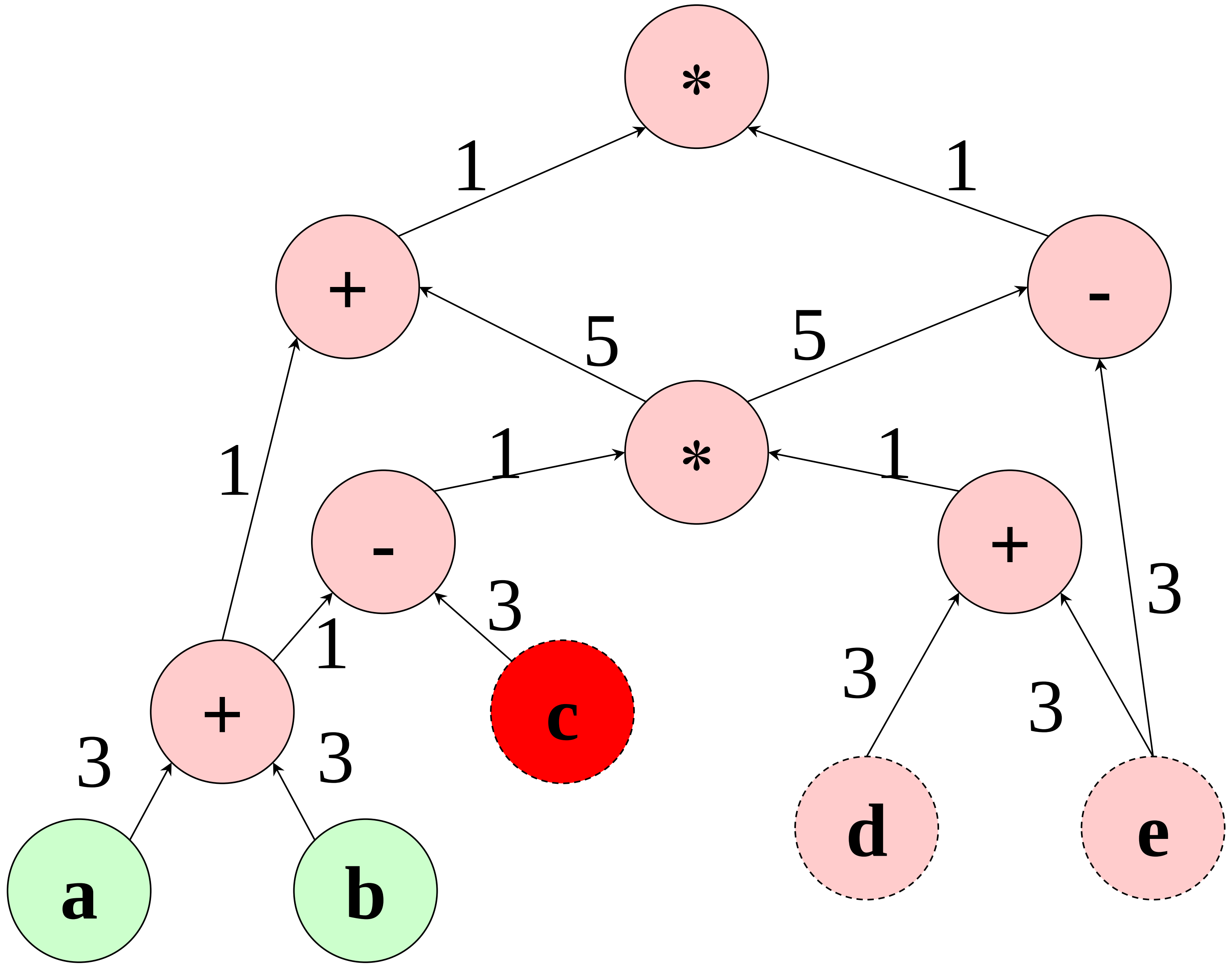
Scheduling by List Example (4)



```
vliw 0: ld R0, a;
vliw 1: ld R1, b;
```

List of ready tasks: b,c,d,e

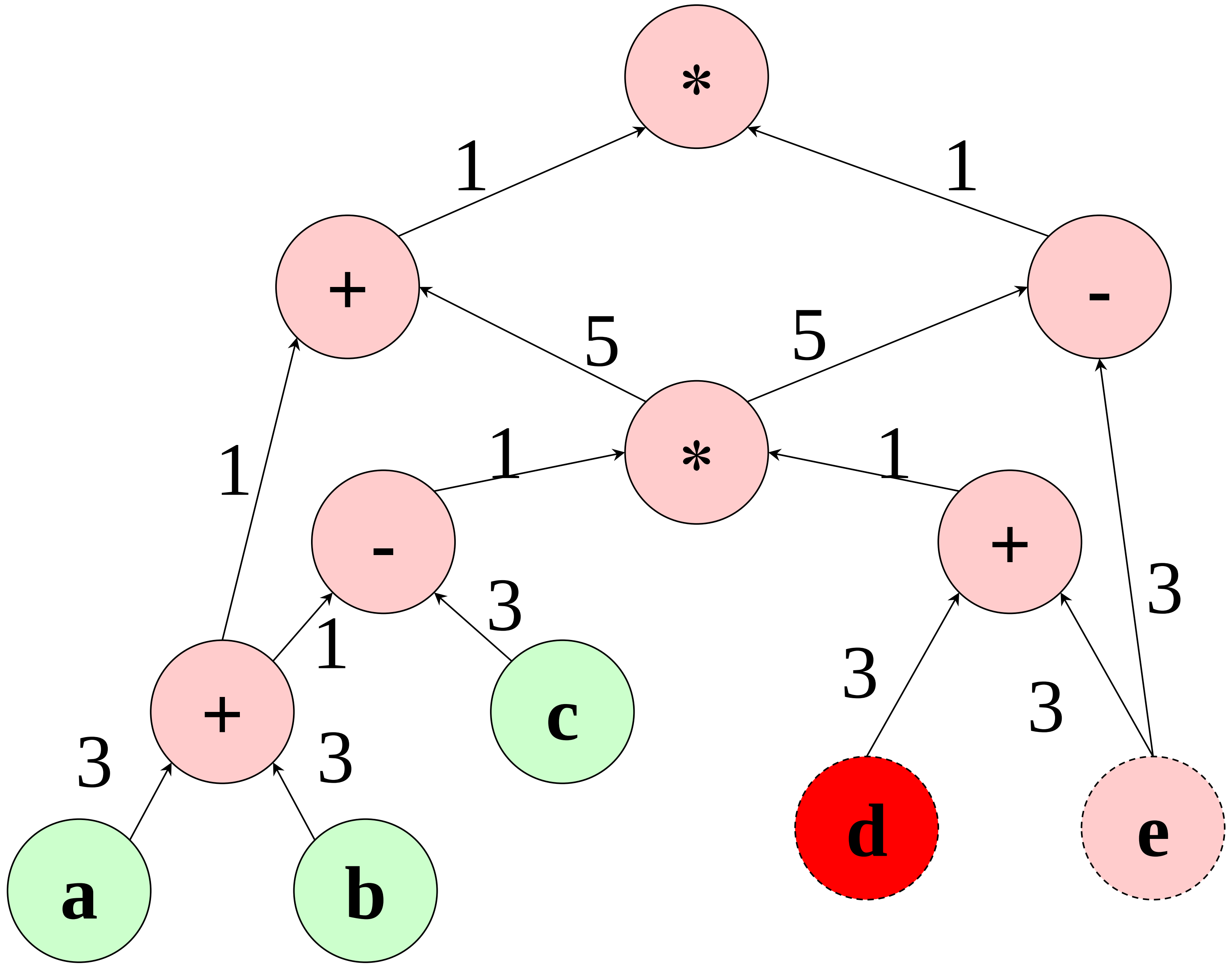
Scheduling by List Example (5)



```
vliw 0: ld R0, a;
vliw 1: ld R1, b;
vliw 2: ld R2, c;
```

List of ready tasks: c,d,e

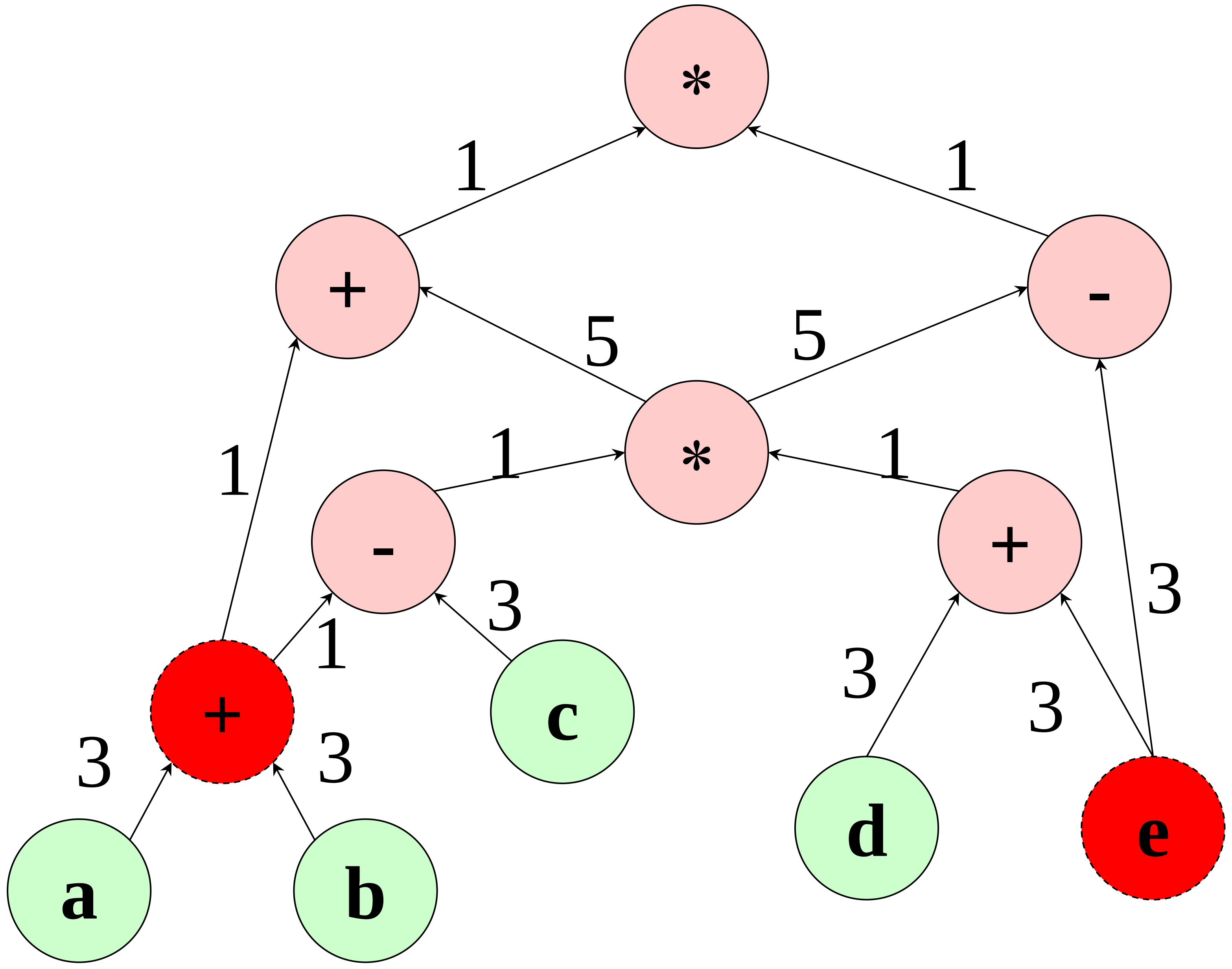
Scheduling by List Example (6)



```
vliw 0: ld R0, a;
vliw 1: ld R1, b;
vliw 2: ld R2, c;
vliw 3: ld R3, d;
```

List of ready tasks: d,e

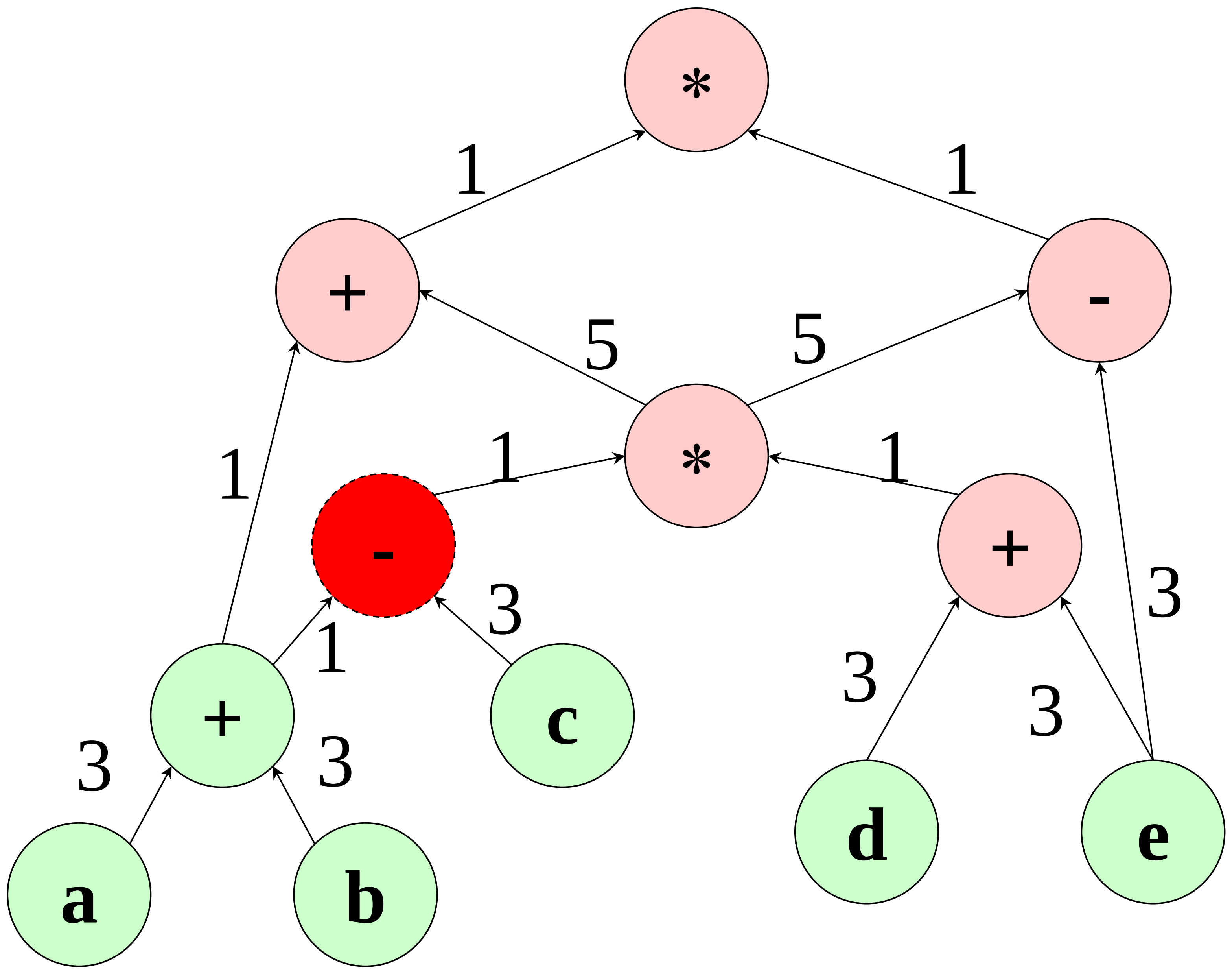
Scheduling by List Example (7)



```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
```

List of ready tasks: e, add

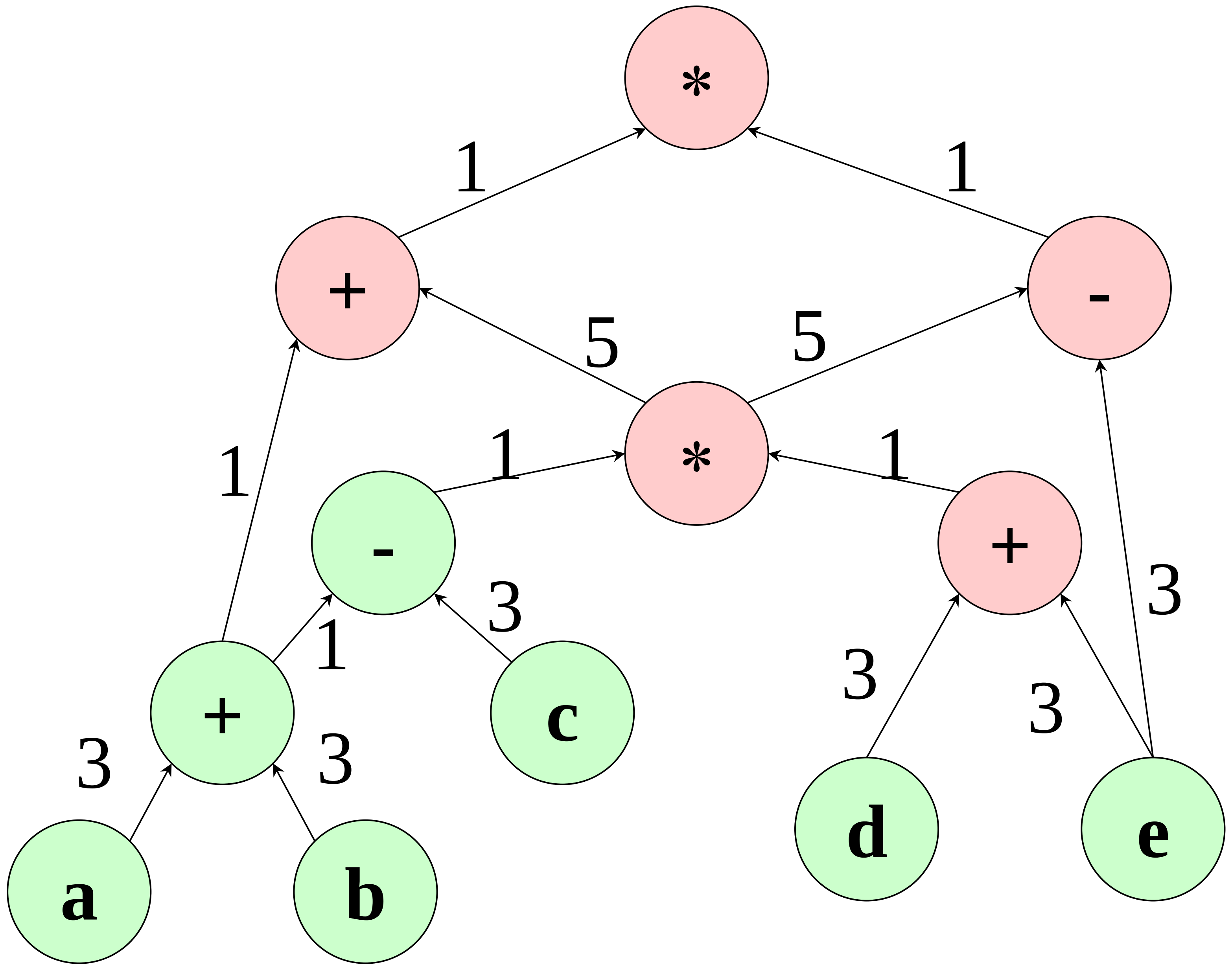
Scheduling by List Example (8)



```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;
vliw 5: sub R2,R1,R2;
```

List of ready tasks: sub

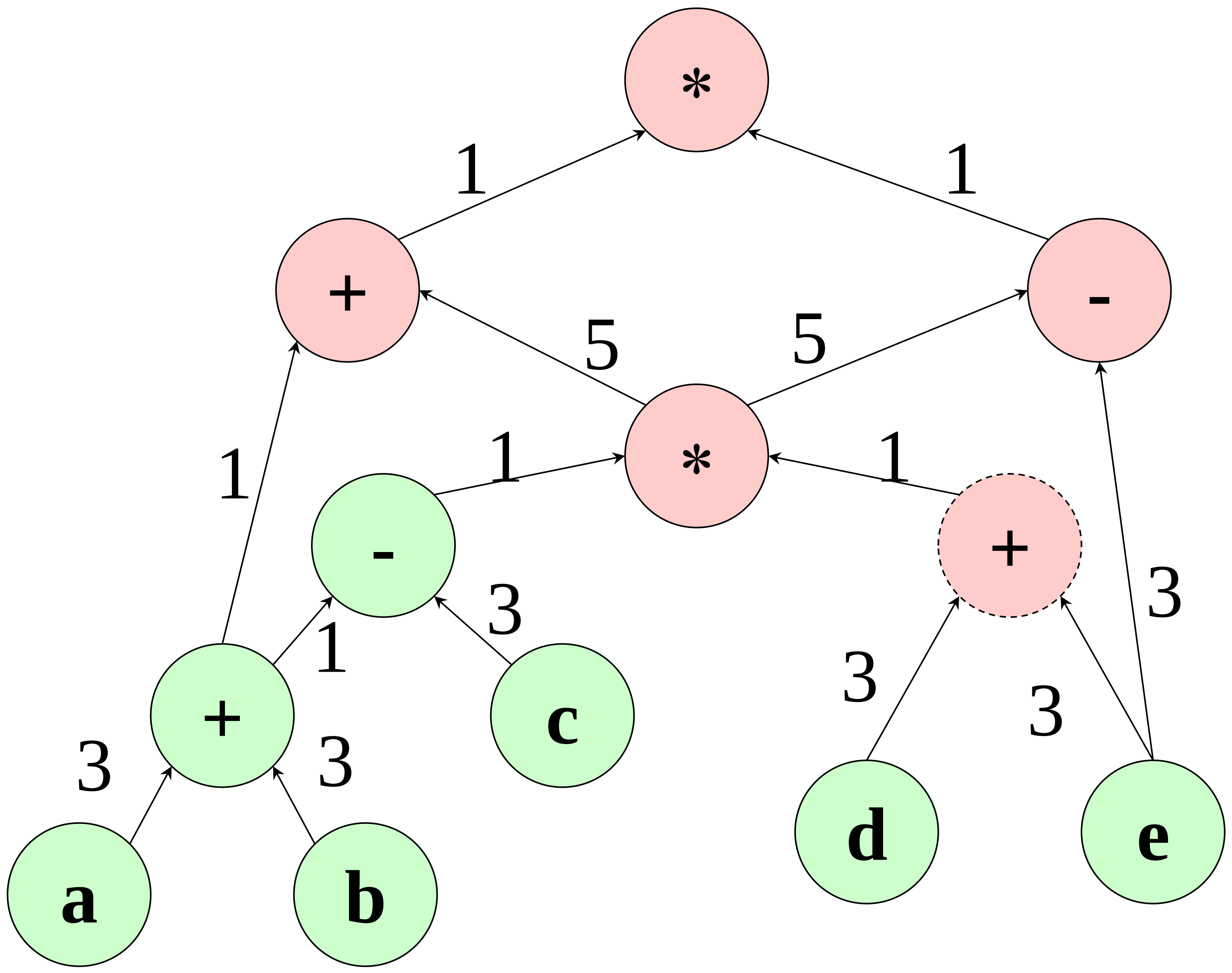
Scheduling by List Example (9)



```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
```

List of ready tasks:

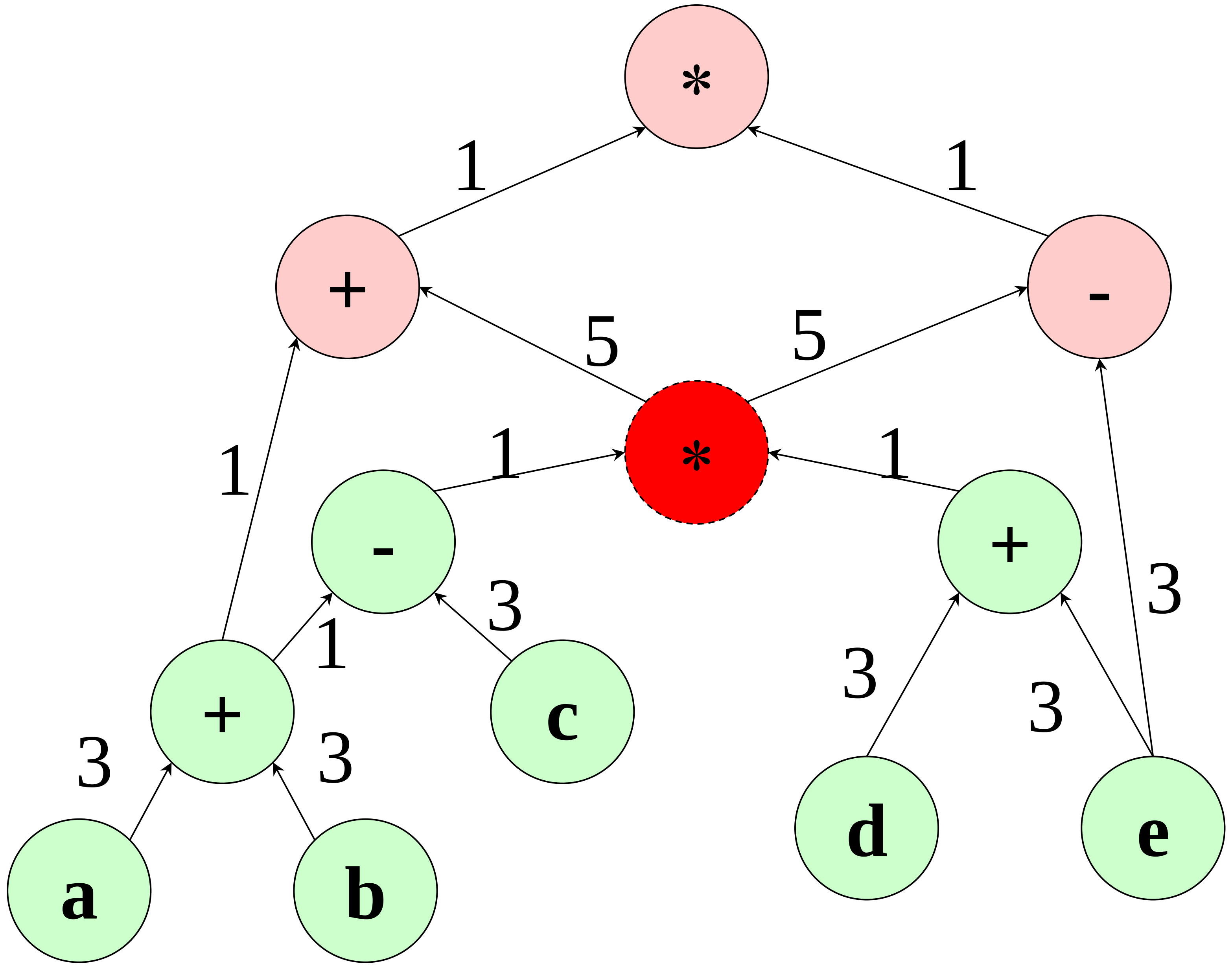
Scheduling by List Example (10)



```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
vliw 7: add R0,R3,R4;
```

List of ready tasks: add

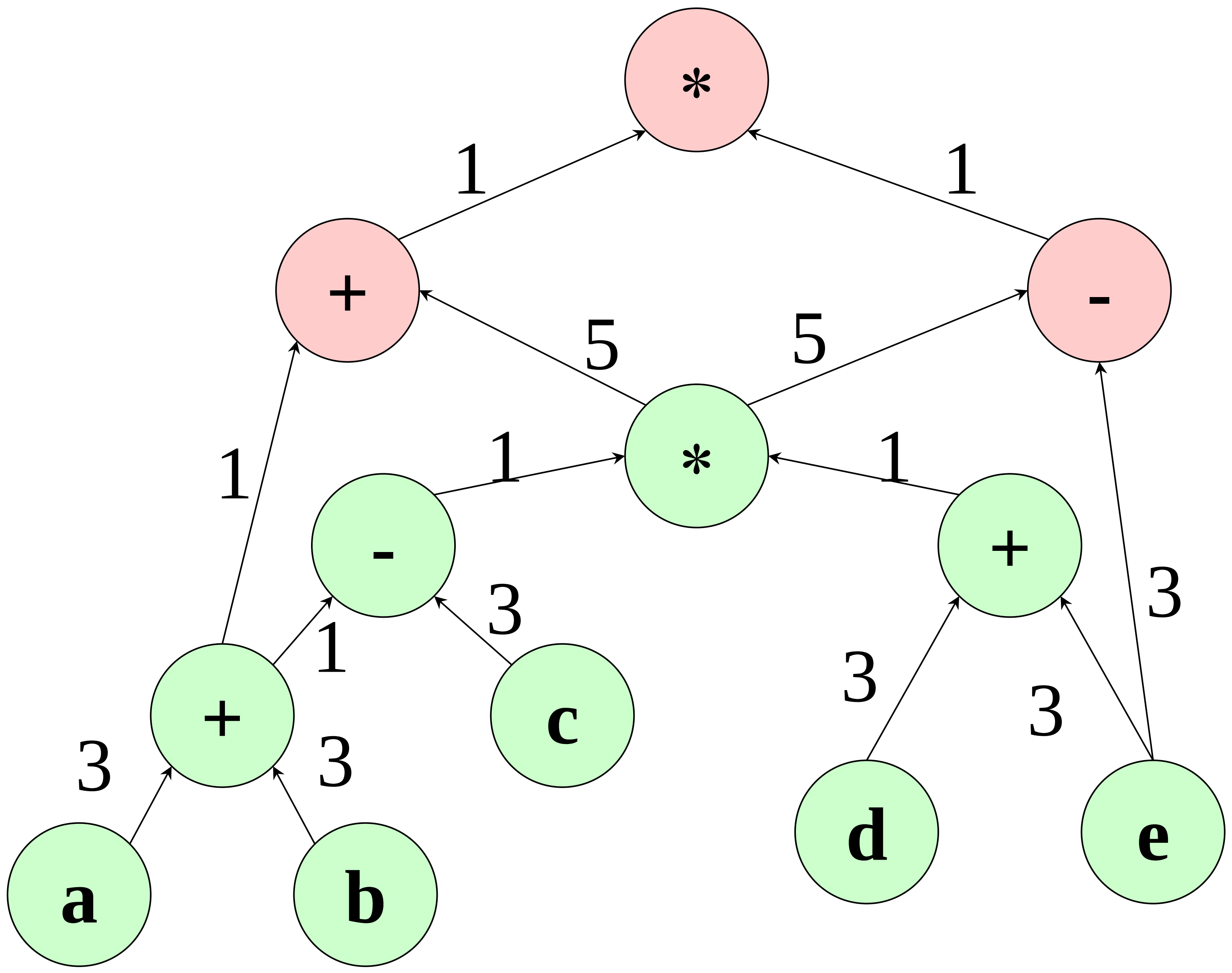
Scheduling by List Example (11)



```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
vliw 7: add R0,R3,R4;
vliw 8: mult R0,R2,R0;
```

List of ready tasks: mult

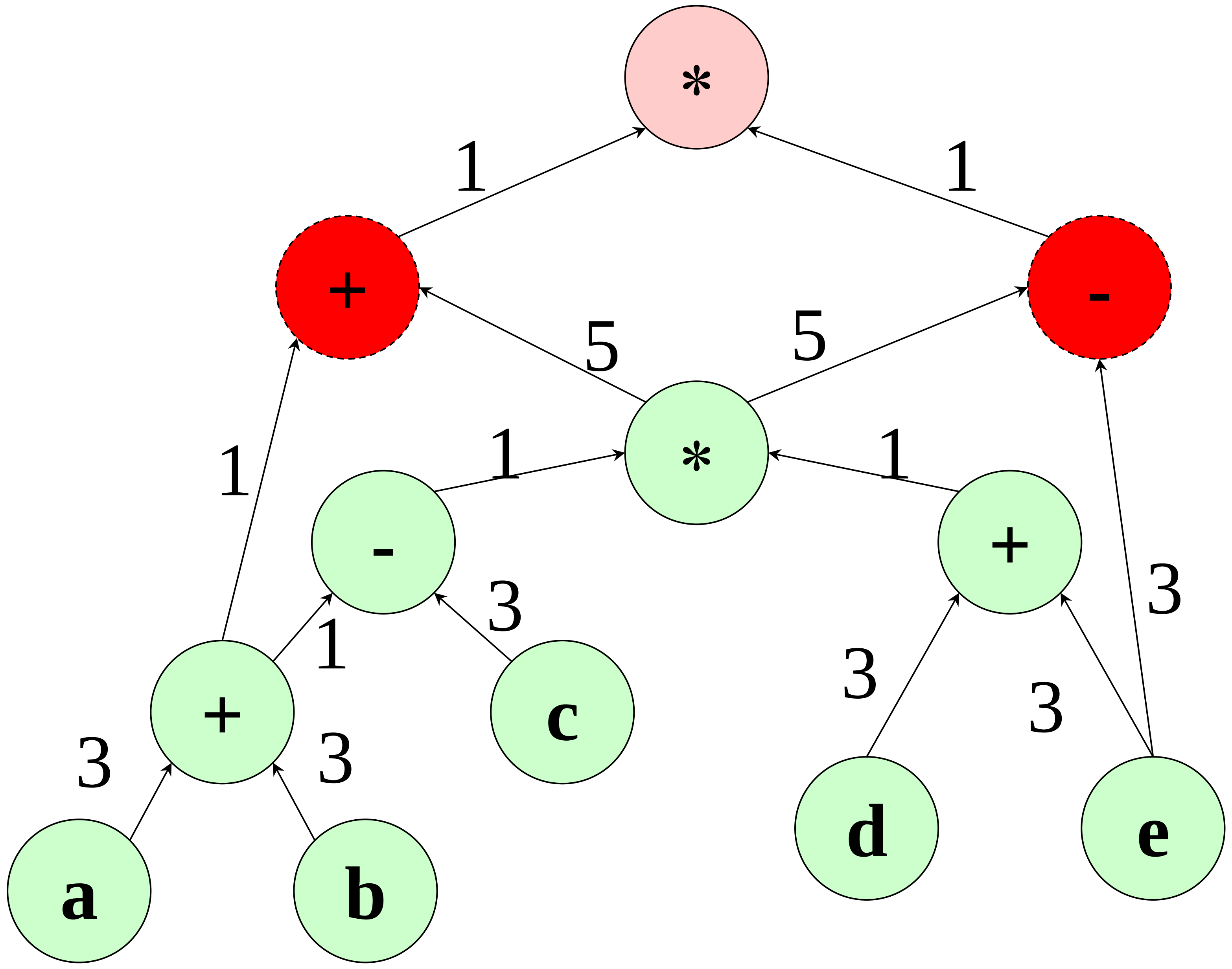
Scheduling by List Example (12)



```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
vliw 7: add R0,R3,R4;
vliw 8: mult R0,R2,R0;
vliw 9: nop;
vliw 10: nop;
vliw 11: nop;
vliw 12: nop;
vliw 13: nop;
```

List of ready tasks:

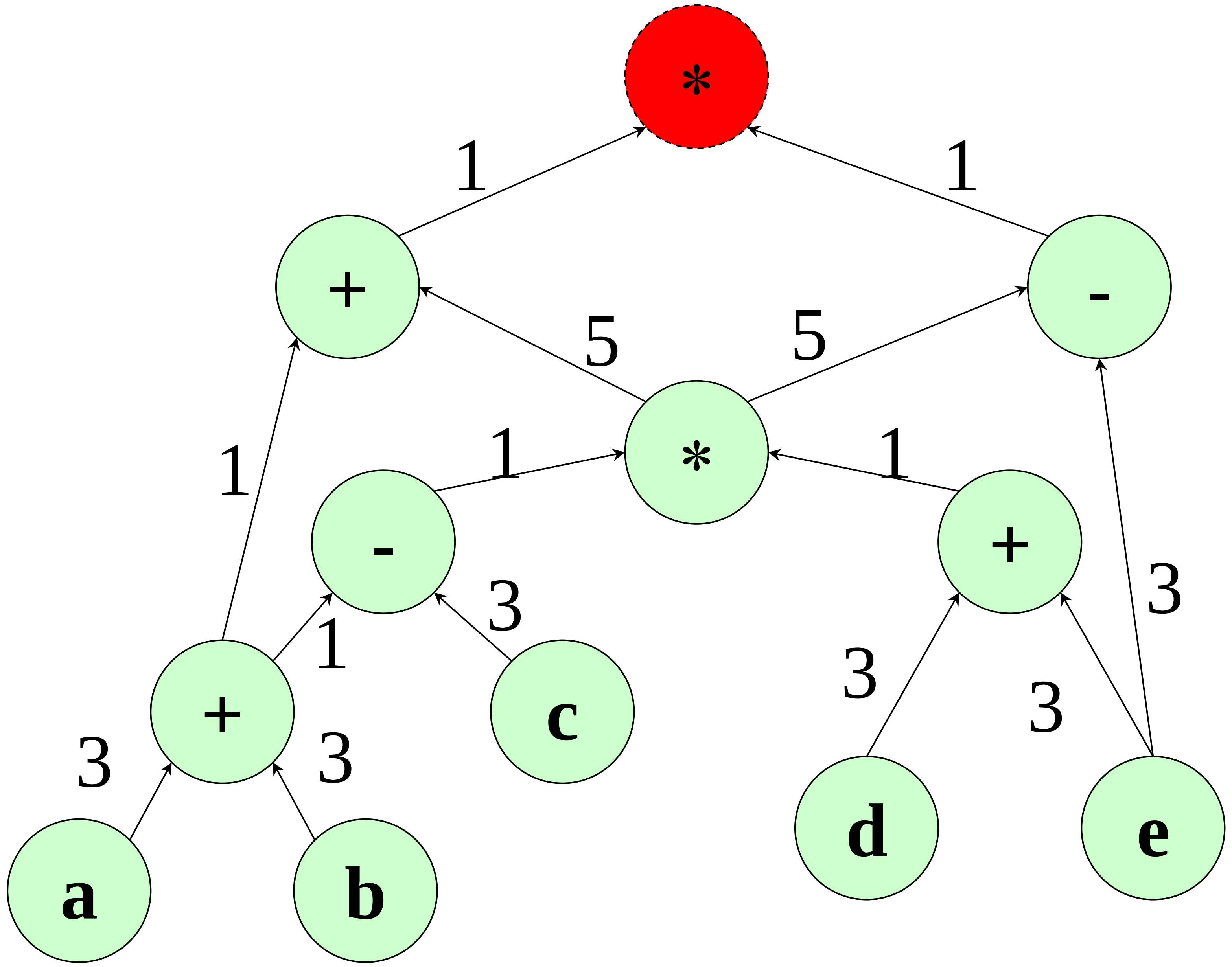
Scheduling by List Example (13)



List of ready tasks: add, sub

```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
vliw 7: add R0,R3,R4;
vliw 8: mult R2,R2,R0;
vliw 9: nop;
vliw 10: nop;
vliw 11: nop;
vliw 12: nop;
vliw 13: nop;
vliw 14: add R0,R0,R2;
         sub R1,R2,R4;;
```

Scheduling by List Example (13)



List of ready tasks: mult

```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
vliw 7: add R0,R3,R4;
vliw 8: mult R2,R2,R0;
vliw 9: nop;
vliw 10: nop;
vliw 11: nop;
vliw 12: nop;
vliw 13: nop;
vliw 14: add R0,R0,R2;
         sub R1,R2,R4;;
vliw 15:mult R0,R0,R1;
```

Scheduling by List Example (13)

- Number of static cycles: 16
- Static IPC: 12/16 <1
- Processor allows IPC of 4
- DAG did not exhibit much parallelism
- Resources of the processors (MEM) limited parallelism

```
vliw 0: ld R0,a;
vliw 1: ld R1,b;
vliw 2: ld R2,c;
vliw 3: ld R3,d;
vliw 4: ld R4,e;
        add R0,R0,R1;;
vliw 5: sub R2,R1,R2;
vliw 6: nop;
vliw 7: add R0,R3,R4;
vliw 8: mult R2,R2,R0;
vliw 9: nop;
vliw 10: nop;
vliw 11: nop;
vliw 12: nop;
vliw 13: nop;
vliw 14: add R0,R0,R2;
         sub R1,R2,R4;;
vliw 15:mult R0,R0,R1;
```

Experimental Findings

- Local scheduling (within a base block) is limited
  - There are not enough parallel operations to fill all holes

- Look for parallel operations beyond the base block
- One technique is software pipelining

Resource Constraints

- What are resource constraints:
  - Number of resources
  - Use of Resources
  - Conflict of simultaneous use
  - Latency
- Aim for considering resource constraints
  - Test if two instructions can cause resource assignment problems (functional units, pipeline floor, ...)
  - Need to account for these conflicts several cycles in advance (latency > 1)
- Representations
  - Reservation table
  - Machines

Reservation Tables

- Make a list of the needs of an instruction with a *resource utilization matrix*
  - Intuitive approach
- Semantics
  - Lines: record instruction latency (in cycles)
  - Columns: Available resources on the target architecture
  - Cell(i, j): marked if the instruction needs the resource *j* during its *ith* execution cycle
  - Binary tables
- Ability to have multiple tables per instruction
  - Concept of alternatives or options

Reservation Table Example

- Consider an example with perfectly pipelined resources
  - 2 ALU: ALU0 and ALU1
  - 2 instructions: ADD and MUL
- Constraints:
  - ADD can be executed on ALU0 or ALU1
  - MUL can only be executed on ALU1
- What are the reservation tables?

Table for ADD

	ALU0	ALU1
0	X	

	ALU0	ALU1
0		X

Table for MUL

	ALU0	ALU1
0		X

Reservation Table Example (2)

- Will the following sequences create resource conflicts?
  - ADD | ADD ?
  - ADD | MUL ?
  - MUL | MUL ?
  - ADD; ADD ?
  - ADD | MUL; MUL ?

Table for ADD

	ALU0	ALU1
0	X	

	ALU0	ALU1
0		X

Table for MUL

	ALU0	ALU1
0		X

Reservation Table Example (3)

- Consider another example with perfectly pipelined resources:
  - 3 instructions: ADD, SUB, LD
  - 2 resources: ALU and LD / ST
- Constraints:
  - ADD latency: 1 cycle
  - SUB latency: 2 cycles
  - LD: 1 cycle on ALU then 1 cycle on LD / ST
- What are the reservation tables?

- Will the following sequences create resource conflicts
  - ADD | SUB ?
  - ADD | ADD ?
  - SUB | LD ?
  - LD; ADD ?
  - LD; SUB ?
  - SUB; LD ?
  - ADD; SUB; LD ?
  - LD; ADD; SUB ?

Reservation Table Summary

- Reservation table use
  - Binary AND to check if an instruction can be scheduled
  - Binary OR to update resource utilization state
- Advantages
  - Intuitive presentation
  - Reduced storage
- Disadvantages
  - Many tests required (simultaneously)
  - Redundant information

Reservation Tables and State Machines

- Objective
  - Provide for all opportunities for concurrent use of resources
  - Modeling using a finite state machine
- Semantics
  - State is equivalent to a resource assignment
  - Transitions between states are equivalent to scheduling an instruction in the current cycle
- Transitions label
  - Instruction that can be scheduled at the current cycle
  - Special label: NOP instruction

Rservation Table and State Machine

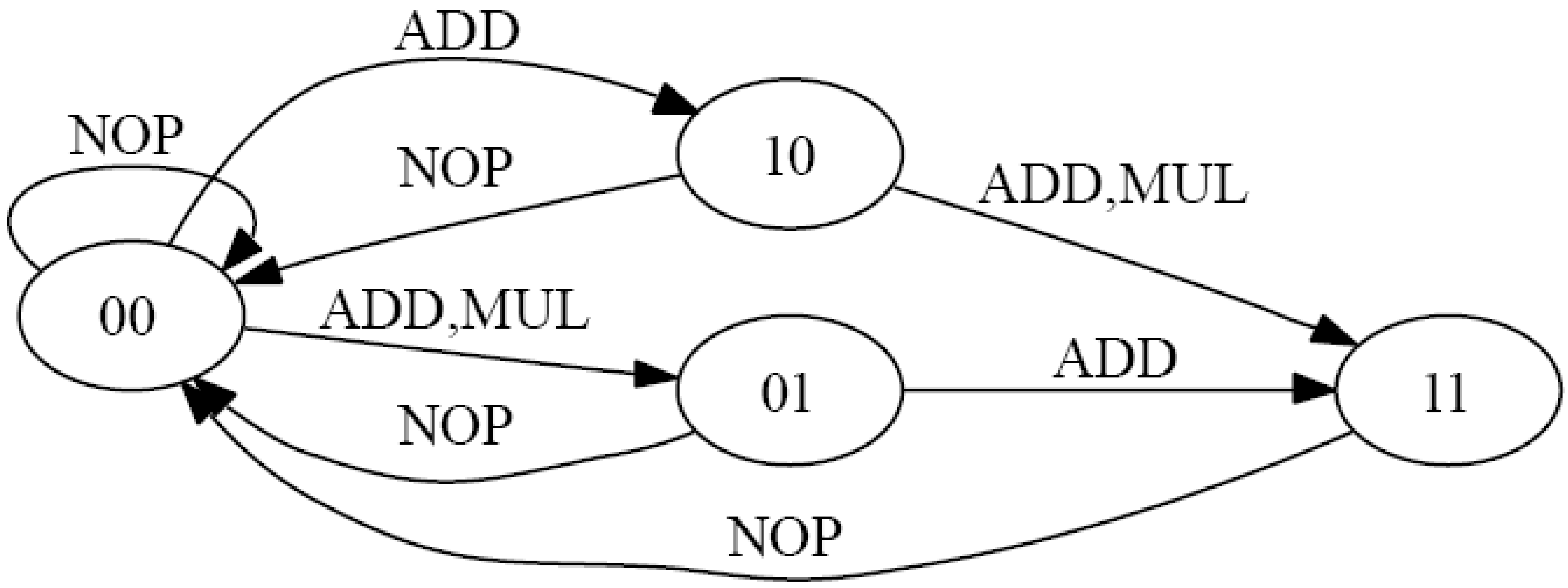
Table for ADD

	ALU0	ALU1
0	X	

	ALU0	ALU1
0		X

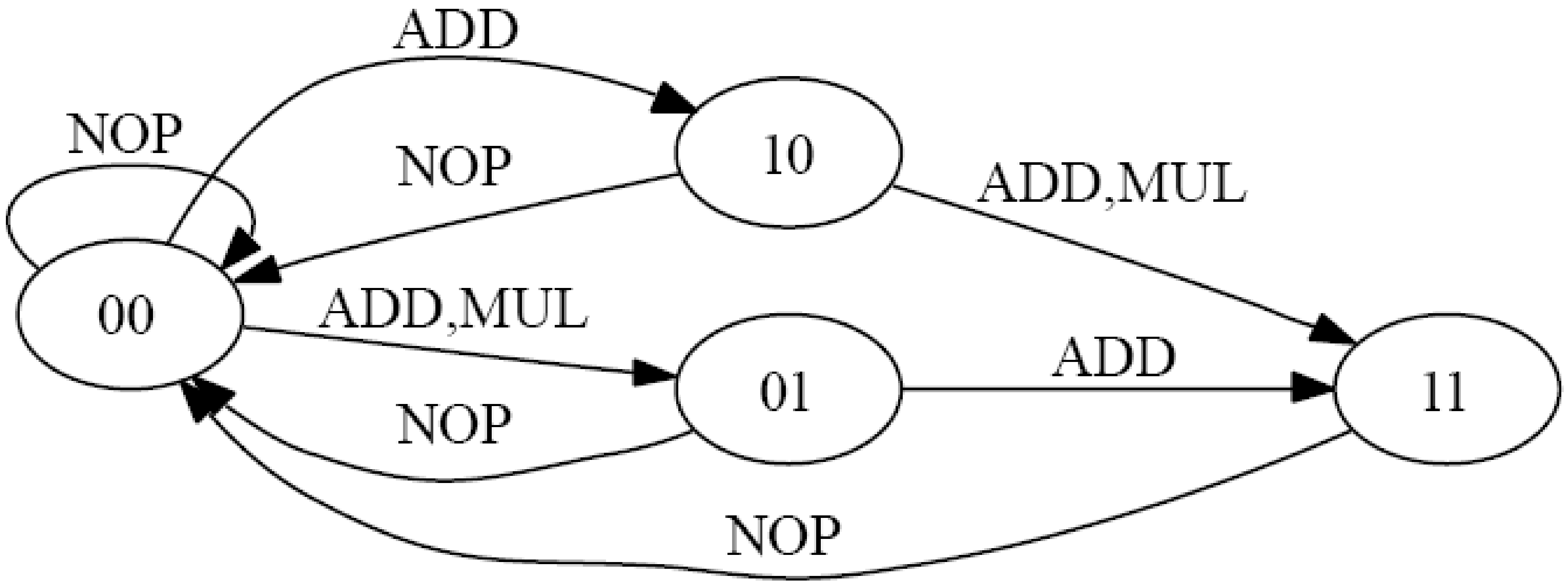
Table for MUL

	ALU0	ALU1
0		X



Rservation Table and State Machine (2)

- Following sequences:
  - ADD | ADD ?
  - ADD | MUL ?
  - MUL | MUL ?
  - ADD; ADD ?
  - ADD | MUL; MUL ?



Rservation Table and State Machine (3)

ADD instruction:

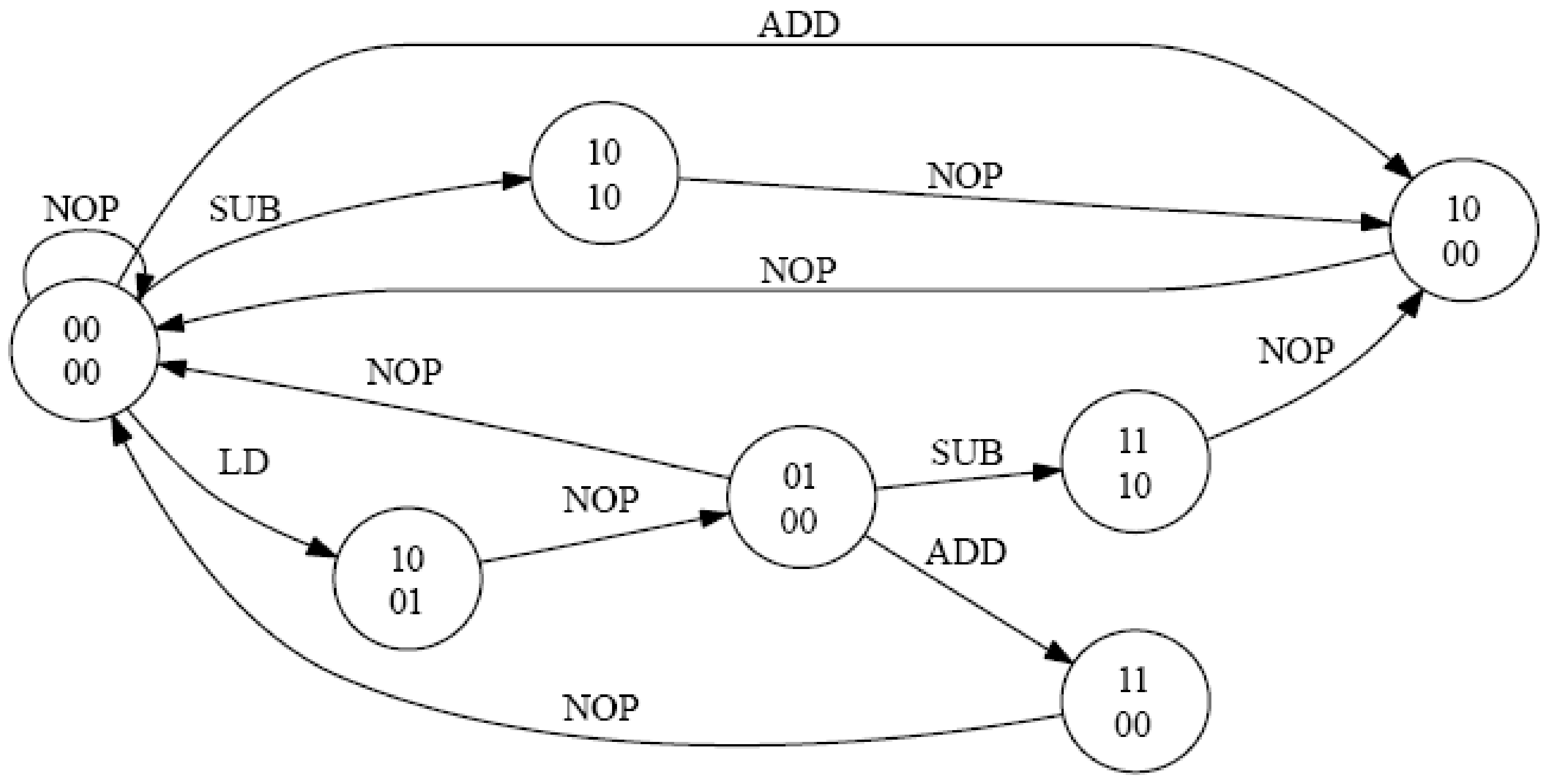
	ALU	LD/ST
0	X	

SUB instruction:

	ALU	LD/ST
0	X	
1	X	

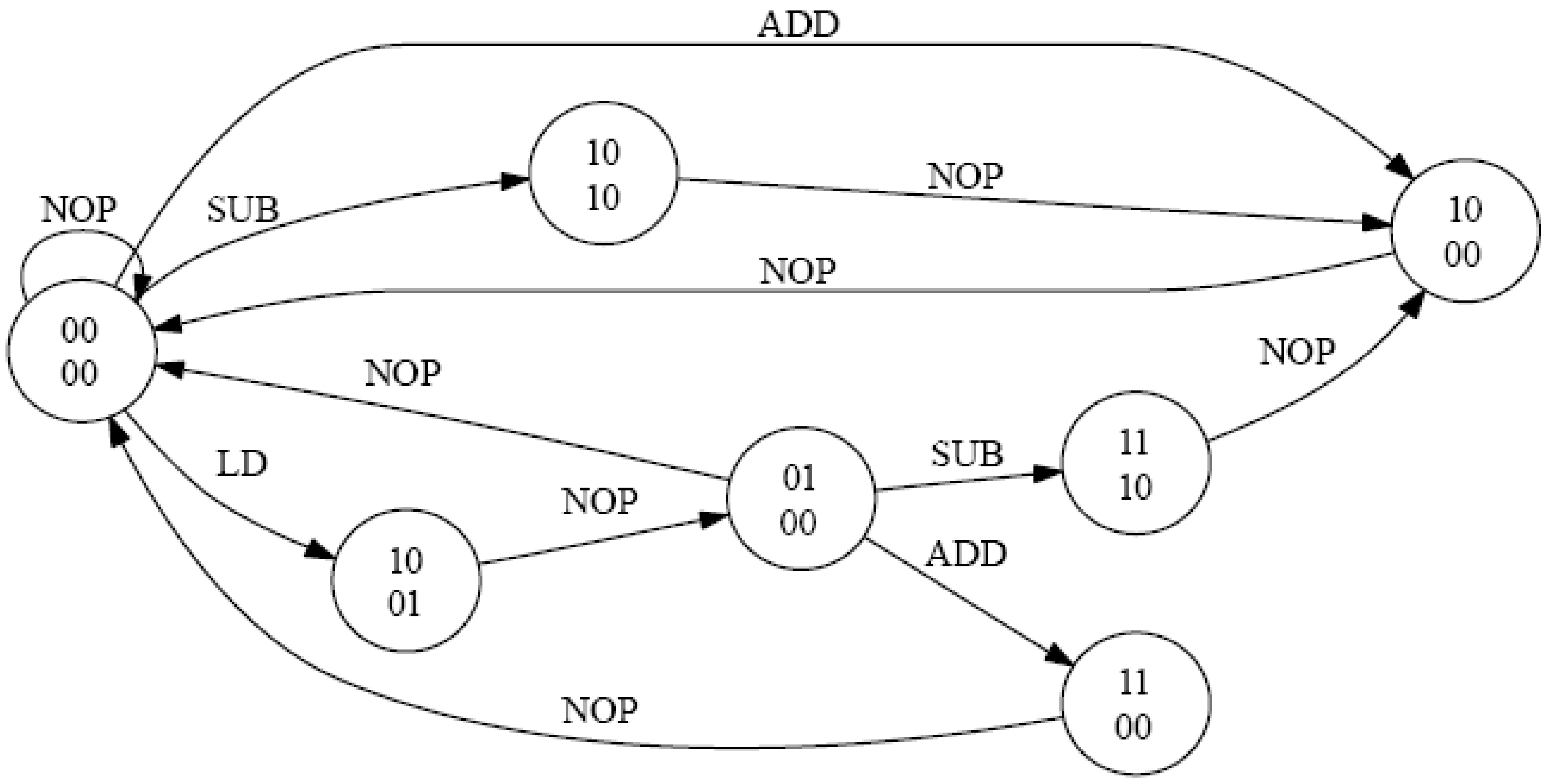
LD instruction:

	ALU	LD/ST
0	X	
1		X



Rservation Table and State Machine (4)

- Following sequences:
  - ADD | SUB ?
  - ADD | ADD ?
  - SUB | LD ?
  - LD ; ADD ?
  - LD ; SUB ?
  - SUB ; LD ?
  - ADD ; SUB ; LD ?
  - LD ; ADD ; SUB ?



State Machine Summary

- State machine use
  - An instruction *I* can be scheduled in a given state if there is an outgoing arc labeled *I*
  - Update the current state by following this arc
- Advantages
  - Quick test
  - Fast update
- Disadvantages
  - Pre-processing time to determine
  - Storage of the state machine (possibly separate)
  - Reduced flexibility
    - difficult to schedule instructions at any cycle

Resource Scheduling and Constraints

- Scheduling strategy
  - High part: main heuristic taking into account data dependencies
  - Low part: storage of resource uses
- Scheduling Process
  - Beginning in the upper part
  - Heuristic selects the next instruction it wants to schedule
  - Asks the lower part if the constraints of the resource allow it
    - if so, proceed to the next instruction
    - otherwise, on backtrack

Introduction to Software Pipelining

- Loop Scheduling
  - List scheduling on the loop body
  - We ignore the dependencies with a non-zero distance
  - Only the available ILP is exploited in an iteration
- Take advantage of parallelism between iterations?
  - Unrolling before scheduling (or during scheduling)
  - Overlap of iterations in a continuous stream

Software Pipelining Example

$$\begin{tikzcd} A \arrow[r, "{1,0}"] & B \arrow[r, "{1,0}"] & C \end{tikzcd}$$

A	r0	r1	r2
0	X		

B	r0	r1	r2
0		X	

C	r0	r1	r2
0			X

- Scheduling valid?
  - Using a list schedule
- We obtain a scheduling of size 3
  - But inter-iteration parallelism is available

Software Pipelining Example (2)

$$\begin{tikzcd} A \arrow[r, "{1,0}"] & B \arrow[r, "{1,0}"] & C \end{tikzcd}$$

A	r0	r1	r2
0	X		

B	r0	r1	r2
0		X	

C	r0	r1	r2
0			X

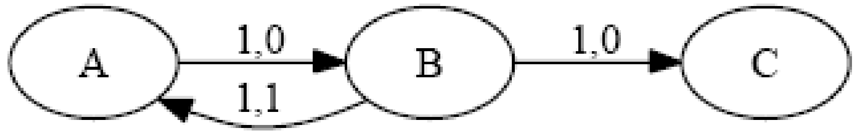
- Scheduling on multiple iterations?

Cycle	Schedule			
	0	1	2	3
0	A			
1	B	A		
2	C	B	A	
3		C	B	A

Cycle	r0	r1	r2
0	X		
1	X	X	
2	X	X	X
3	X	X	X

- Core of 1 cycle and depth of 3 iterations

Software Pipelining Example (3)



A	r0	r1	r2
0	X		

B	r0	r1	r2
0		X	

C	r0	r1	r2
0			X

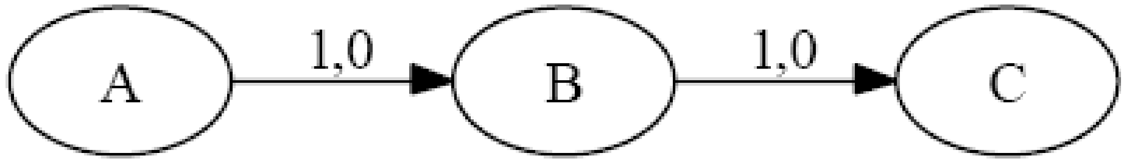
- Scheduling on multiple iterations?

Cycle	Schedule			
	0	1	2	3
0	A			
1	B			
2	C	A		
3		B		
4		C	A	

Cycle	r0	r1	r2
0	X		
1		X	
2	X		X
3		X	
4	X		X

- Core of 2 cycles and depth of 2 iterations

Software Pipelining Example (4)



A	r0	r1
0	X	

B	r0	r1
0	X	

C	r0	r1
0		X

- Scheduling on multiple iterations?

Cycle	Schedule			
	0	1	2	3
0	A			
1	B			
2	C	A		
3		B		
4		C	A	

Cycle	r0	r1
0	X	
1	X	
2	X	X
3	X	
4	X	X

- Core of 2 cycles and depth of 2 iterations

Concept of Software Pipelining

- Approach
  - Start scheduling an iteration before the end of previous one
  - Constant time between beginning of 2 consecutive iterations (called the *initiation interval (II))*
  - Respect of constraints (dependence of data, resources, ...)
  - Schema: *prologue* / *core* / *epilogue*
- Scheduling
  - Several iterations are in-life in the kernel (called the *pipeline depth*)
  - Scheduling an instruction for a cycle and iteration number (called *two-dimensional scheduling*)

Software Pipelining Parameters

- Performance
  - Performance P in cycles for n iterations
  - $P = (n-1) \times II + M$
  - Linear in II
  - Minimize the value of II
- Settings
  - II - initiation interval
  - D - depth
  - M - makespan

Interval Initiation

- Time in cycles between the launch of two consecutive iterations
  - Corresponds to the size of the kernel
  - Describes performance
- Parameters influencing *II*
  - Data Dependencies: *RecMII*
  - Resource constraints: *ResMII*

$$RecMII = \max_{\forall circuits\, \theta} \left\lceil \frac{\mathrm{latency}(\theta)}{\mathrm{distance}(\theta)} \right\rceil$$

- Minimum value: *MII = Max (ResMII, RecMII)*

Data Dependencies (RecMII)

- Constraints of dependencies
- Intra-iteration constraint between *a* and *b*
  - σ scheduling function
  - *l* latency of a dependency
  - *d* distance of a dependency
- Inter-iteration constraint between *a* and *b*

$$\sigma(a) + l(a,b) \le \sigma(b)$$

- Dependency on a cycle

$$\sigma(a) + l(a,b) \le \sigma(b) + II.d(a,b)$$

Depth and Makespan

- *Depth*
  - Number of Iiterations in the kernel
  - Secondary parameter
  - Influences the size of the *prologue* / *epilogue*
  - Complex relationship with *II*
- *Makespan*
  - Time to run a full iteration in the kernel
  - Secondary parameter
  - Linked to *depth* and *II*
  - Influences the lifetime of variables

Software Pipelining Approaches

- Principal approaches
  - Exact algorithms
    - list all possibilities
    - NP-Complete
  - Heuristics
    - Choosing a production compiler
- Heuristic family
  - Modulo Scheduling
    - most used solution in practice
  - Kernel Recognition

Modulo Scheduling

- Principle
  - Sets an iteration so that its repetition is valid every *II* cycles
  - Need to fix *II* before scheduling this iteration
  - If this is not possible, release the constraint by increasing *II*
- Main algorithm
  - Sort nodes by priority
  - Calculate *MII*
  - *II = MII*
  - As long as the scheduling is not valid
    - schedule an iteration with *II*
    - if the scheduling is not valid, the increase *II* by 1

Iterative Modulo Scheduling (IMS)

- Principle
  - Bob Rau, MICRO-27 (1994)
  - Extension of scheduling by list
  - Notion of budget
- Implementation in production compilers

Iterative Modulo Scheduling (IMS) (2)

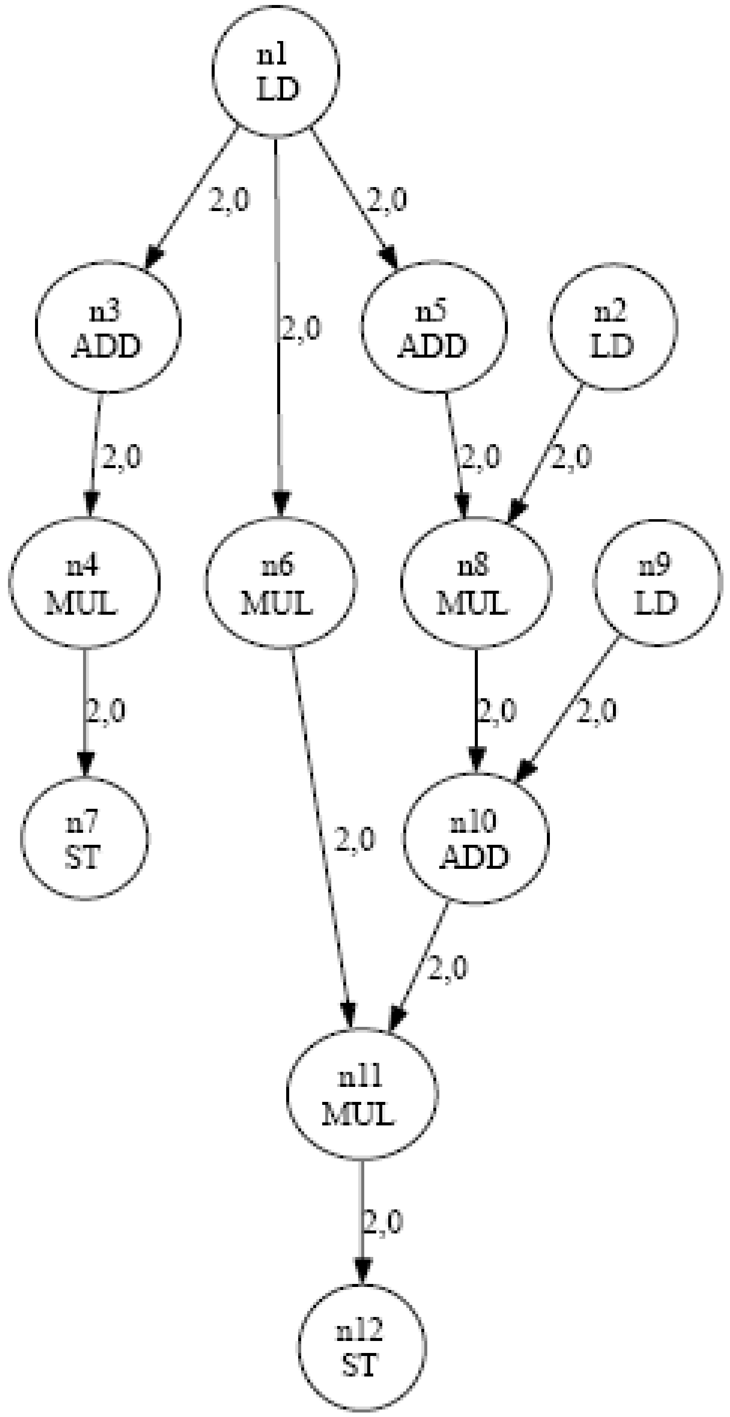
- Choose next instruction in order of decreasing priority
  - *H* is the priority
  - How to calculate *H*
- Calculation of the possible scheduling interval
  - [Estart, Estart + II-1]

$$H(P) = \begin{cases} 0 & \text{if P is a leaf} \\ \max_{Q \in Succ(P)} (H(Q) + L(P,Q) - II \times D(P,Q)) & \text{otherwise} \end{cases}$$

- Scheduling test
  - If unsuccessful, force scheduling (remove another statement)
  - Involves the notion of budget to avoid entering into a scheduling / disordering cycle

$$Estart(P) = \max_{Q \in Pred(P)} \begin{cases} 0 & \text{if Q is unscheduled} \\ \max(0, \sigma(Q) + L(Q,P) - II \times D(Q,P)) & \text{otherwise} \end{cases}$$

IMS Example 1



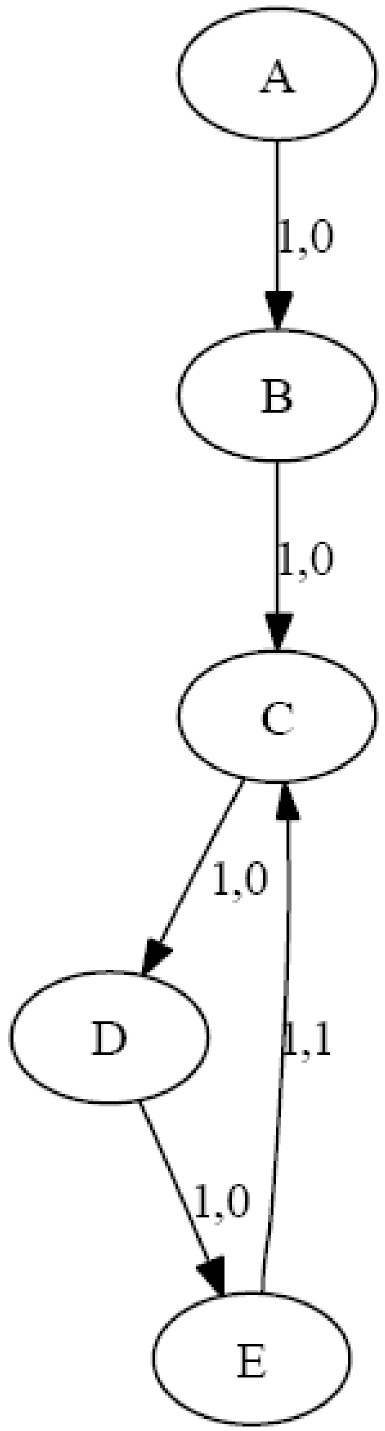
LD/ST	r0	r1	r2	r3
0			X	
1			X	

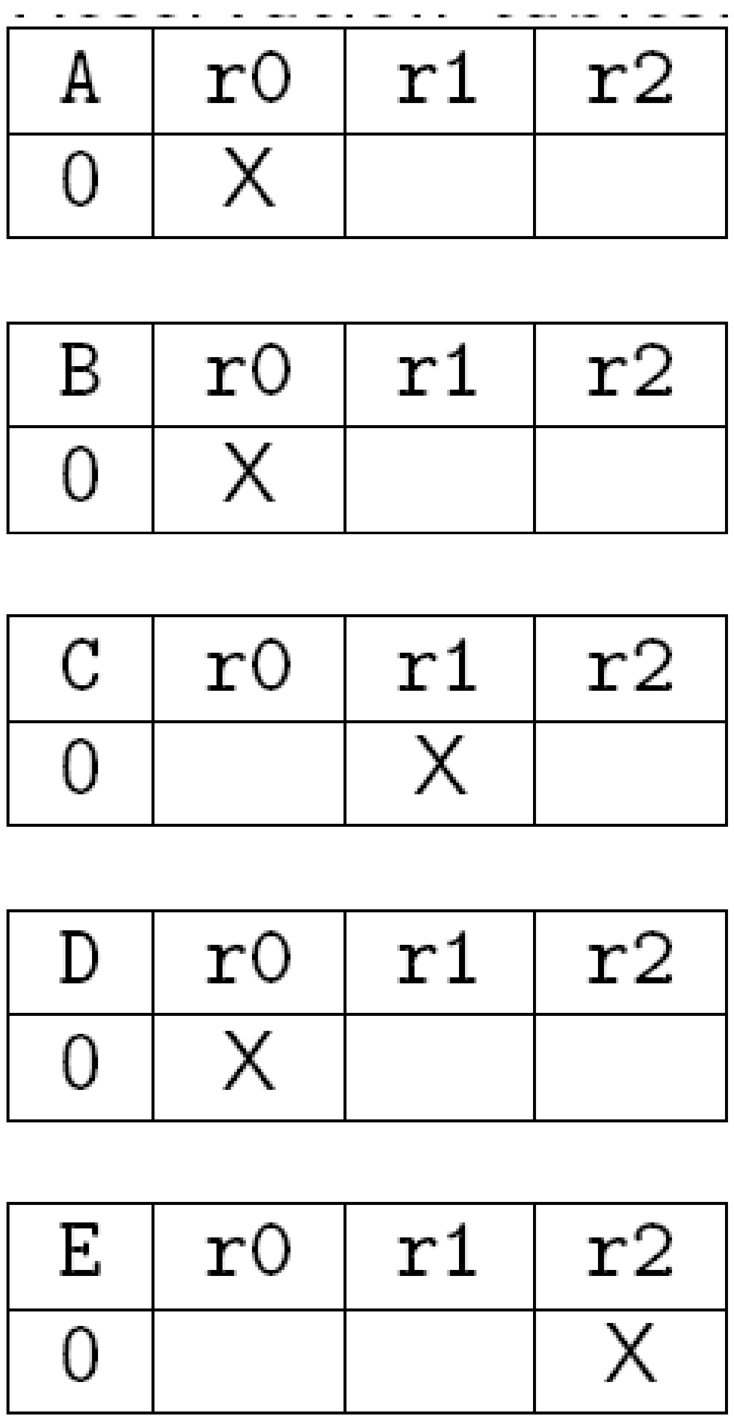
LD/ST	r0	r1	r2	r3
0				X
1				X

ADD	r0	r1	r2	r3
0	X			
1	X			

MUL	r0	r1	r2	r3
0		X		
1		X		

IMS Example 2





A	r0	r1	r2
0	X		

B	r0	r1	r2
0	X		

C	r0	r1	r2
0		X	

D	r0	r1	r2
0	X		

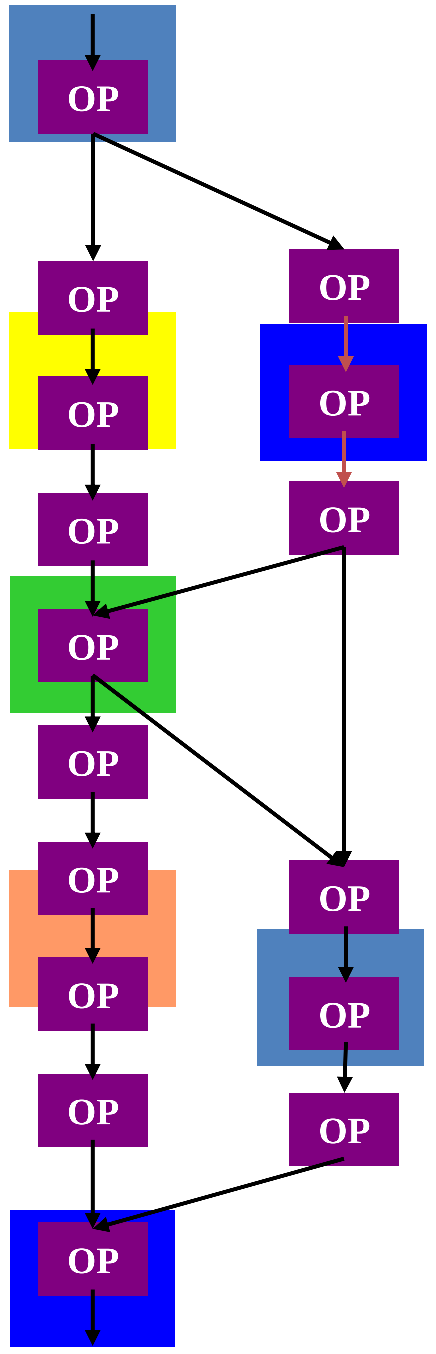
E	r0	r1	r2
0			X

IMS Exemple 2 (2)

- Calculate *MII*
    - *RecMII* = 3, *ResMII* = 3
    - ➔ *MII* = 3
- Calculate the priority function *H*
    - *H(A)=4*
    - *H(B)=3*
    - *H(C)=2*
    - *H(D)=1*
    - *H(E)=0*
- Test with II=MII=3
- Success: *II=3*, *D=3*, *M=7*

Limits of ILP in a Basic Block

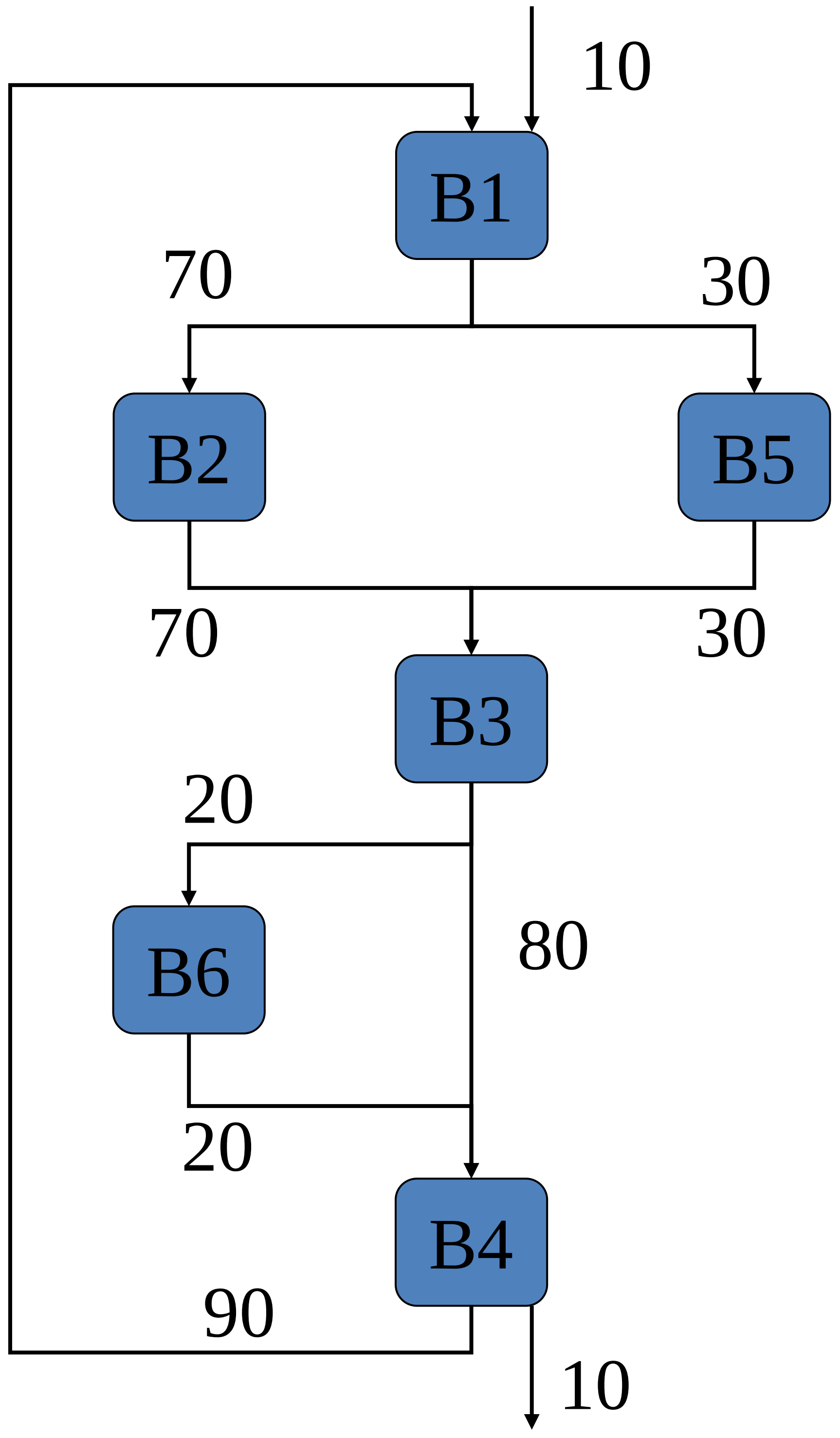
- Initially ('70 -'80), the code was locally optimized in each basic block (BB)
  - Some techniques for specific VLIW processors required operation movements from one BB to another
- Empirically, ILP was limited within a BB
  - Not enough operations
- Find independent operations in other BBs
- Scheduling of instructions is no longer done at the level of BB
  - Done at the level of "regions”



Source: Josh Fisher

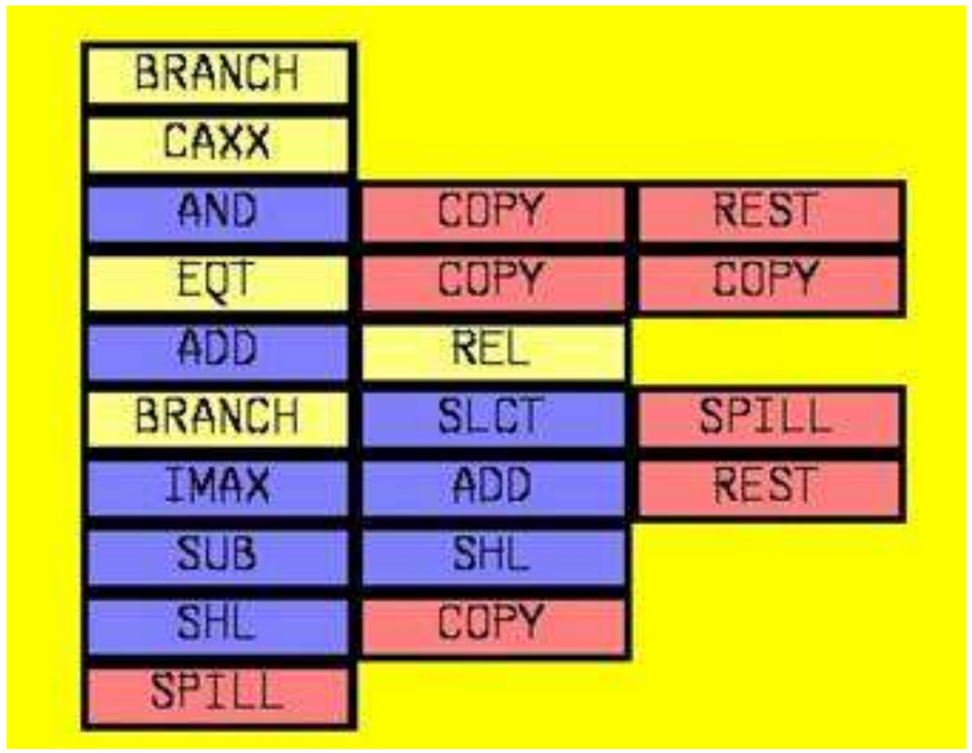
Profile the Application for Information

- Take a representative dataset
- Execute the application with a profiling tool to get information on runtime frequencies of base blocks
- In a CFG, the arcs now contain the execution frequencies or probabilities
- The sum of the outgoing frequencies of a BB is equal to 100%
- A good heuristic of ILP extraction within a function relies on profiling

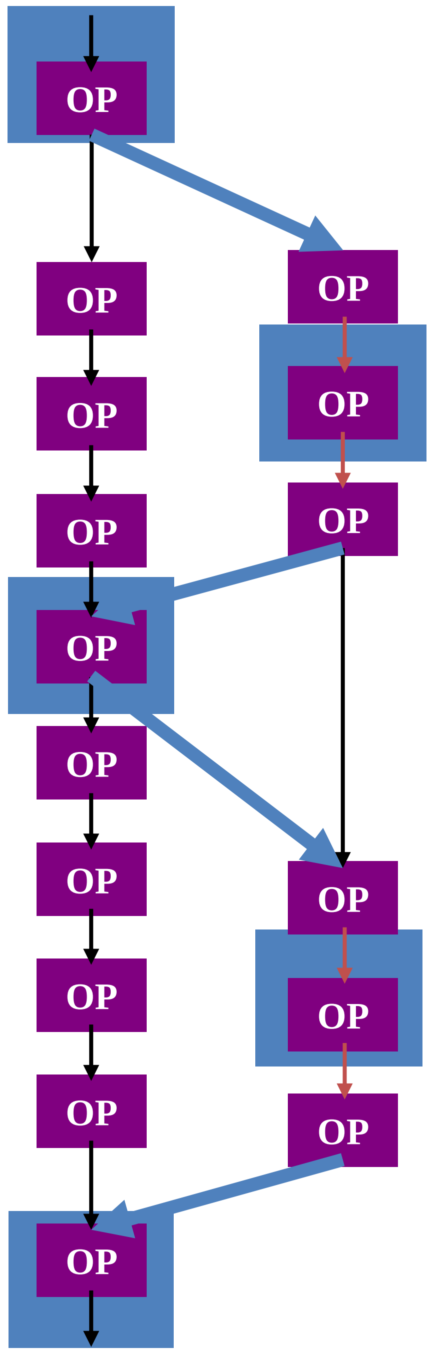


Trace Scheduling History

- The DAGs are now constructed in a "region" and then scheduled as in a conventional base block
- Bring rules to introduce compensation code to move operations from one BB to another



BRANCH		
CAXX		
AND	COPY	REST
EQT	COPY	COPY
ADD	REL	
BRANCH	SLCT	SPILL
IMAX	ADD	REST
SUB	SHL	
SHL	COPY	
SPILL		



Source: Josh Fisher

Speculative and Predicated Execution

- **Speculative execution**
  - Execution of an instruction before knowing if its execution was necessary
- **Predicate execution**
  - Architectural support for the conditional execution of an instruction based on the value of a Boolean operand, called the *predicate* of the statement
- **If-conversion**
  - Algorithm that automatically converts a code with conditional branching into a code with predicates

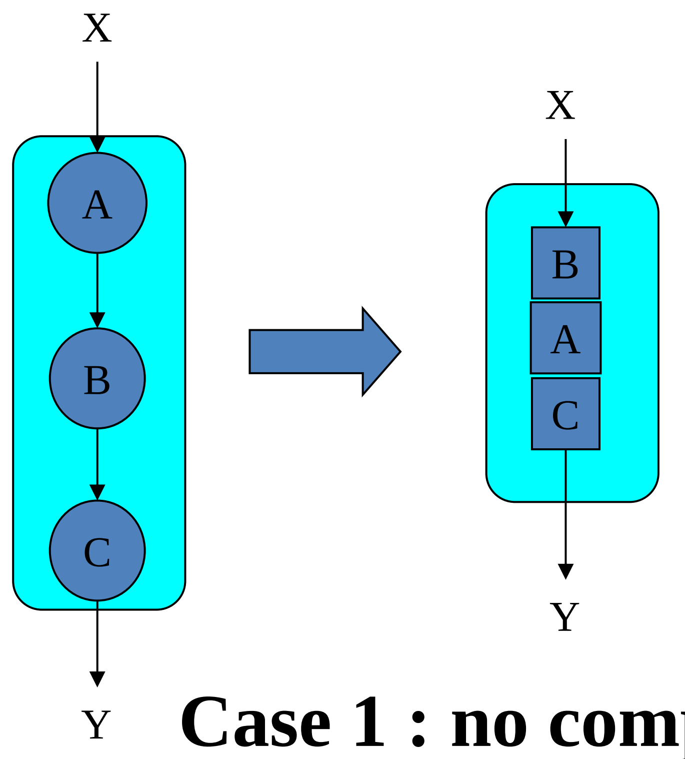
Trace Scheduling (Fisher, 1981)

- Some optimization and scheduling decisions can decrease the execution time of a path and increase the execution time of another path
- Therefore ILP scheduling decisions should favor the most commonly executed execution paths to improve the overall performance of the application
- Trace scheduling divides a function into a set of frequently executed paths, called *traces*

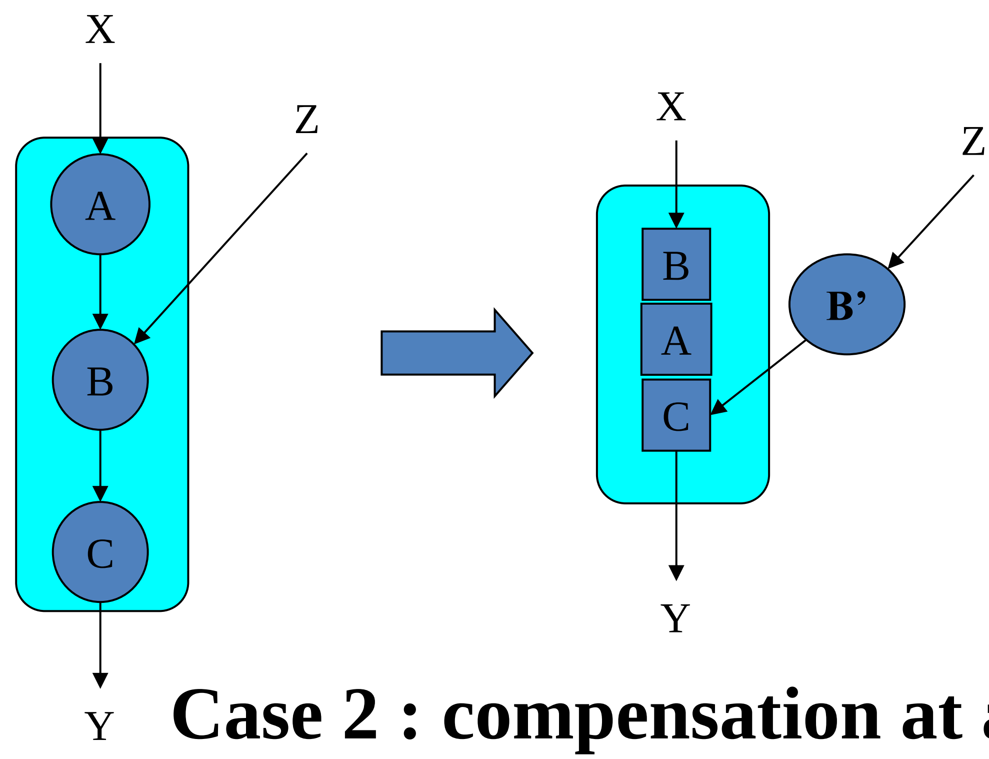
Trace Scheduling (2)

- A trace may contain conditional connections in the middle (*output points*)
- A trace can also contain labels allowing the connection of another trace in the middle (*entry points*)
- These control flow transitions are ignored during trace scheduling
- Once the ILP scheduling is done, a compensation code is introduced to correct the data flow

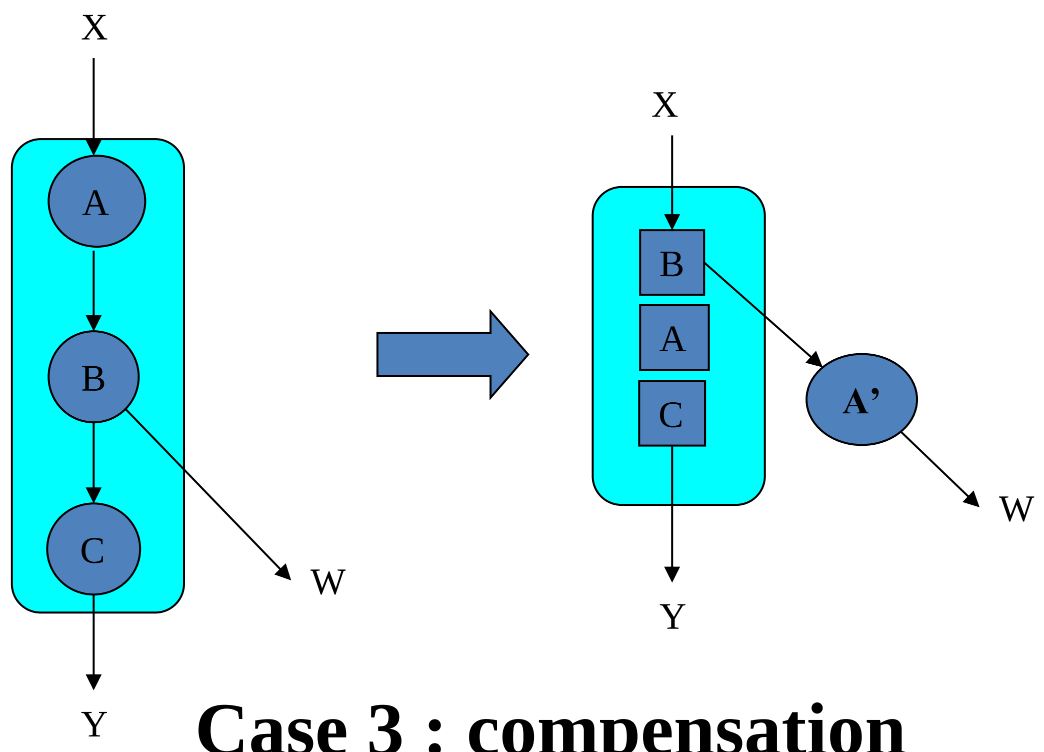
Four Scenarios for Compensation Code



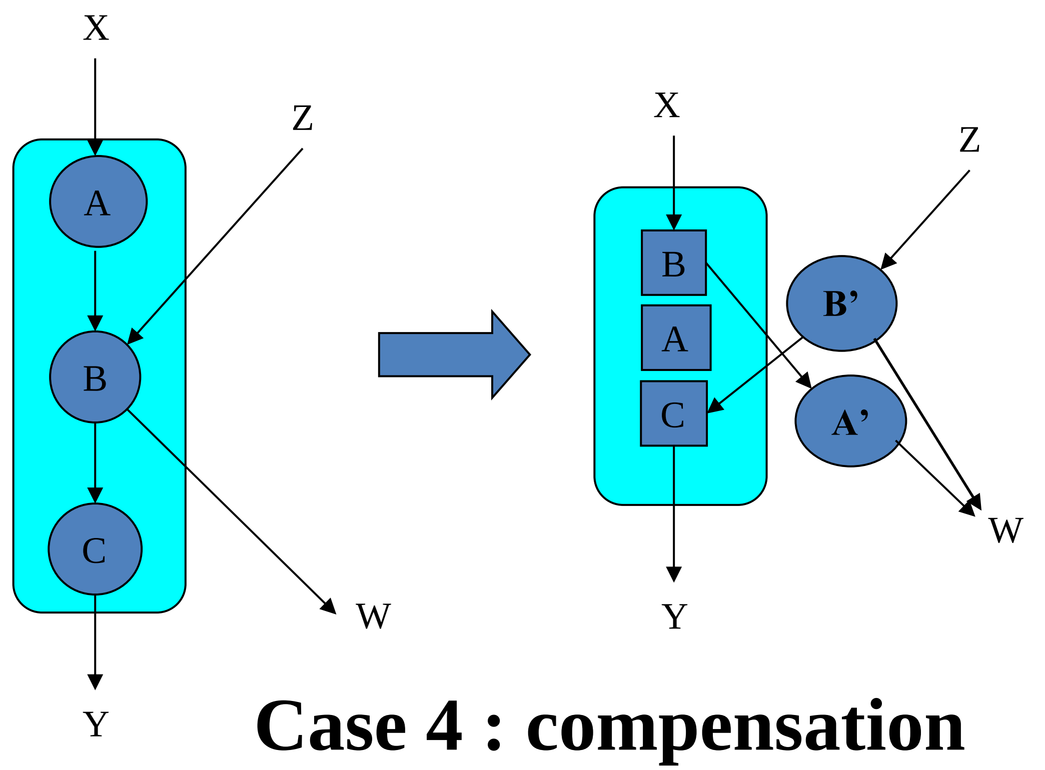
Case 1 : no compensation



Case 2 : compensation at a joint



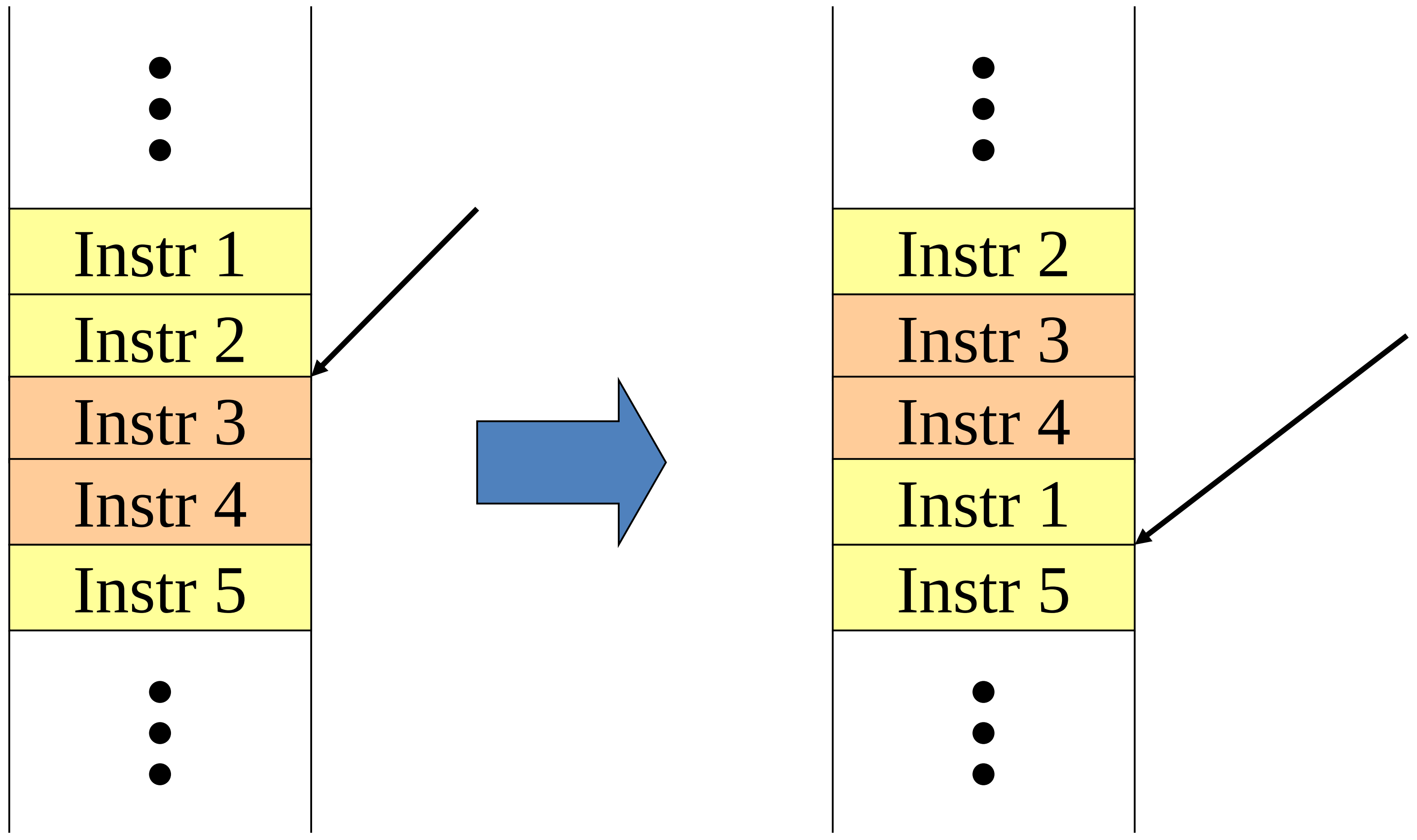
Case 3 : compensation during a fork



Case 4 : compensation during a fork-joint

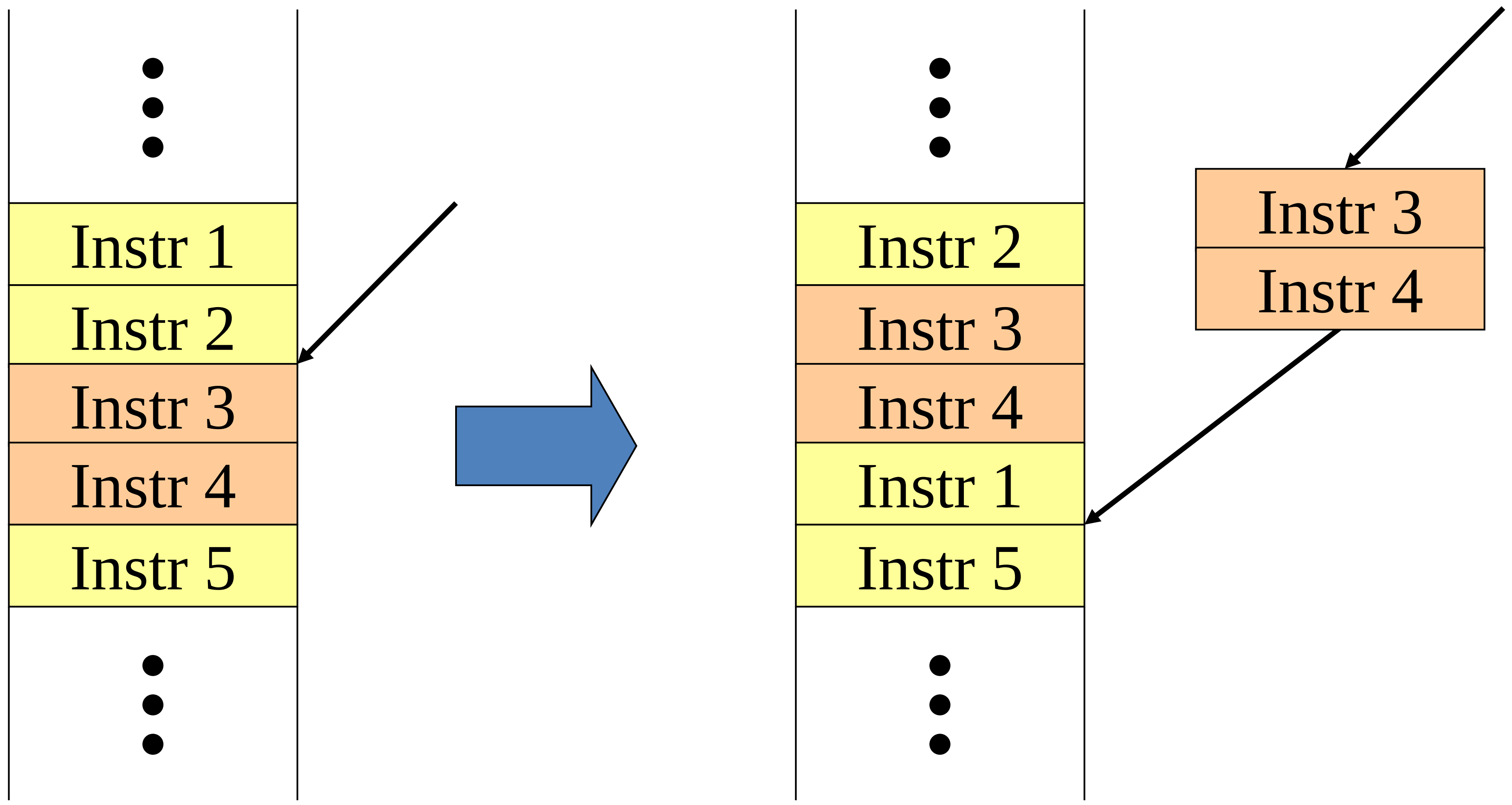
Compensation Code

- Which compensation code should be entered when Instr 3 and Instr 4 are moved above the entry point?



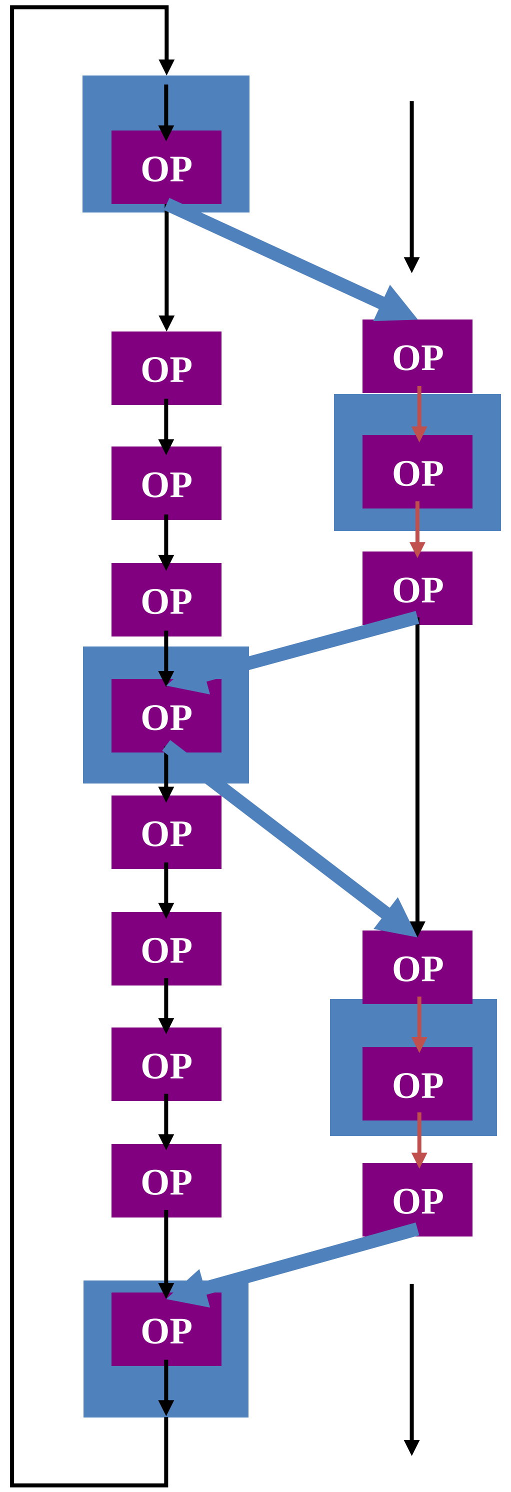
Compensation Code (2)

- Which compensation code should be entered when Instr 3 and Instr 4 are moved above the entry point?



Problems with Traces

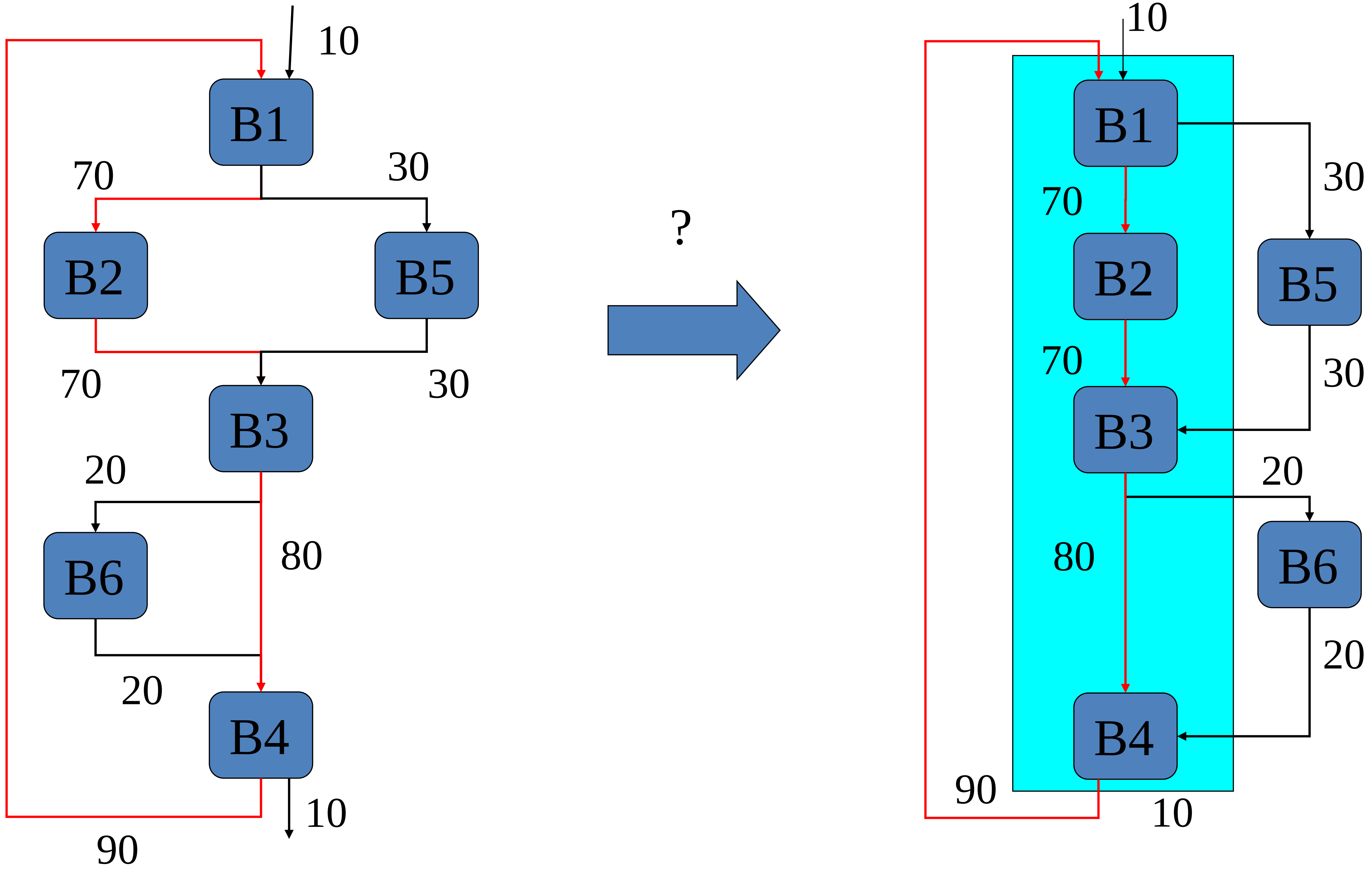
- Traces are complex regions
  - Multiple-entry, multiple-exit
  - Number of distinct paths quickly becomes very large
- Introduction of the clearing code becomes very complex, and the profit is not guaranteed



Super Blocks

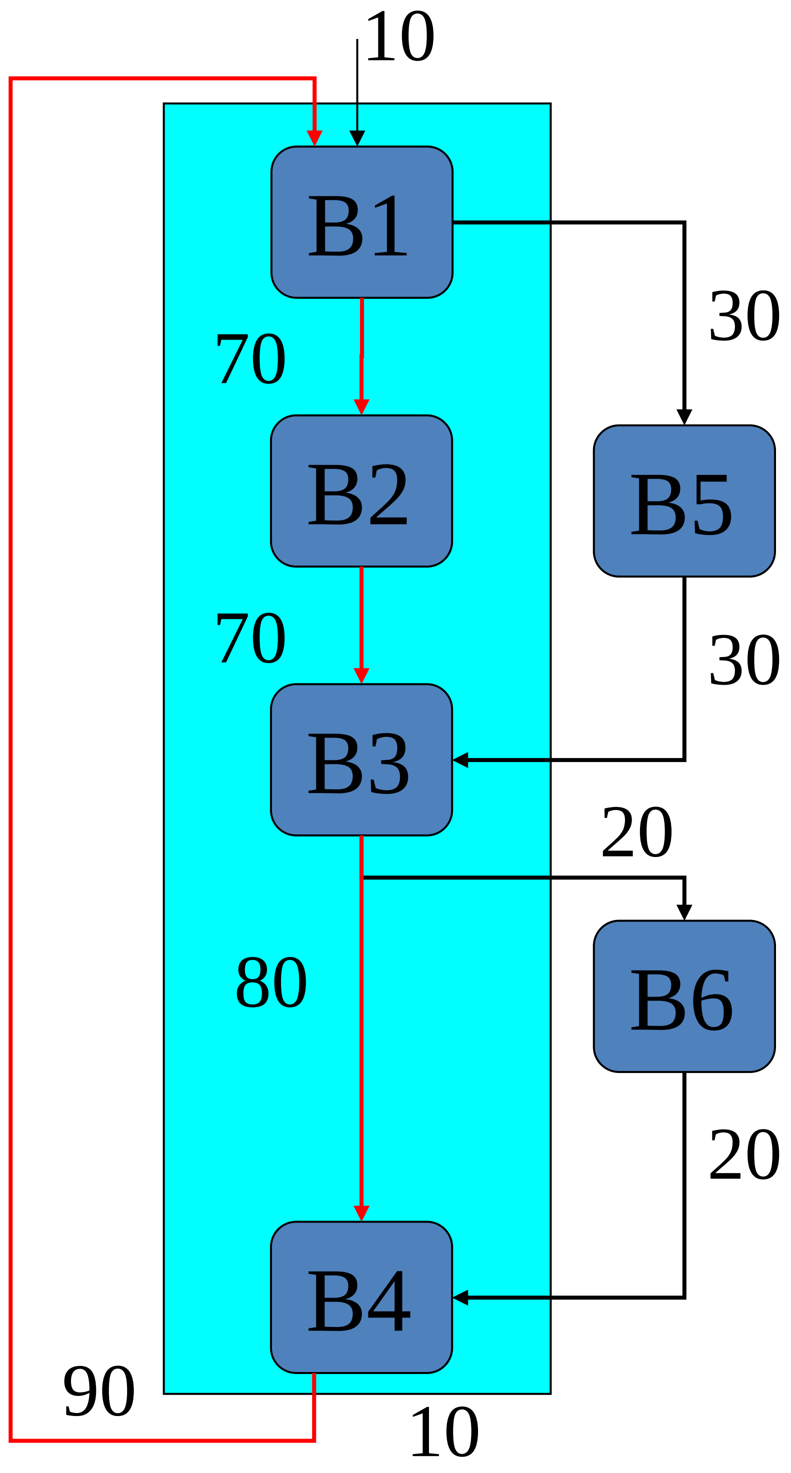
- A super block is a trace with no entry points in the middle
  - Control only enters from the top, but can leave at multiple exit points
  - This is a single-entry trace, multiple-exit
- The formation of the super blocks makes it possible to avoid the complexity of the compensation codes of any trace
  - ILP scheduling heuristic in the super block region is simplified
  - Also, this opens up new optimization opportunities beyond scheduling

Example Formation of a Super Block



Example Formation of a Super Block (2)

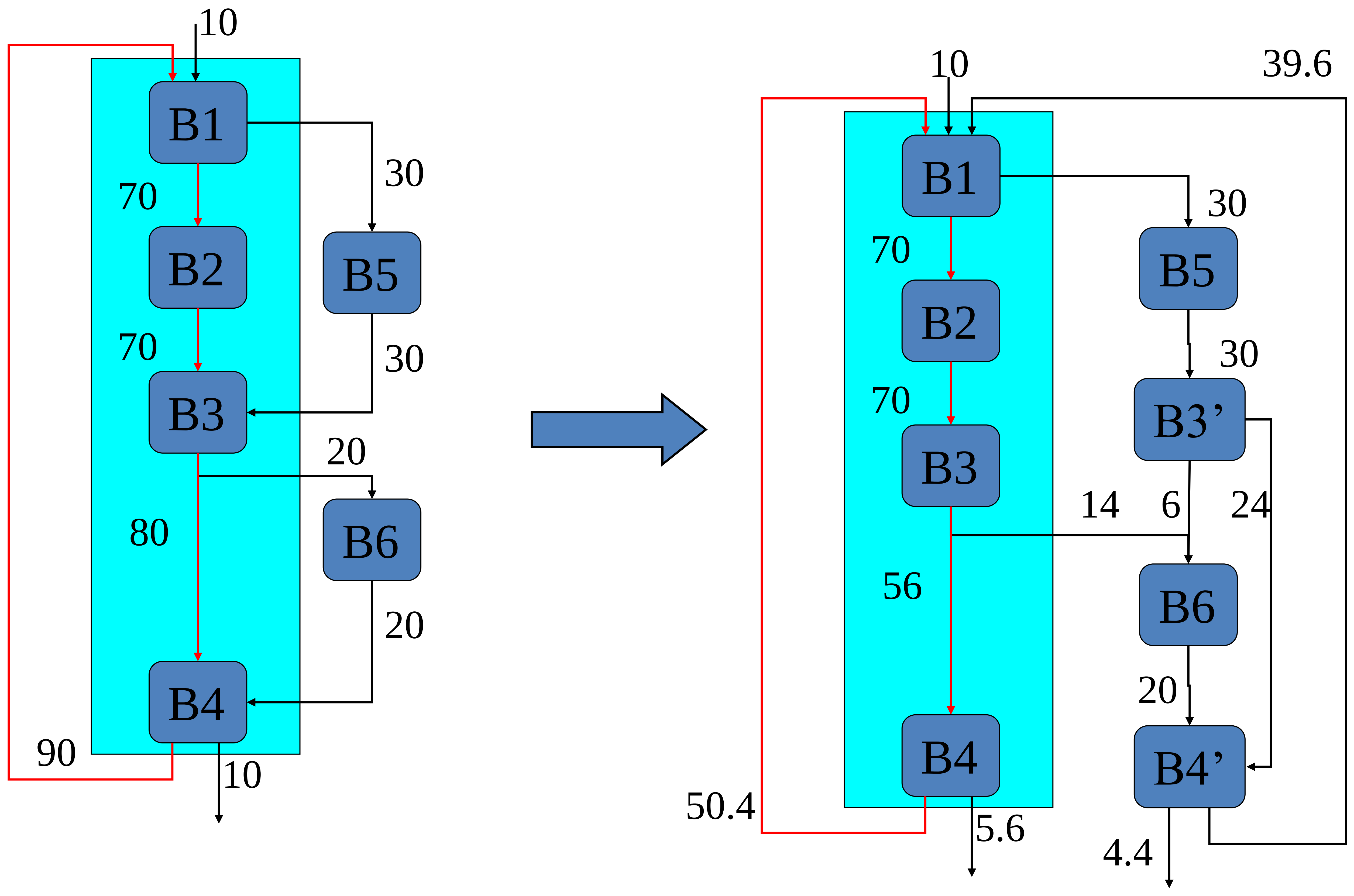
- Is it a super block?
- No, a super block has only one entry point (at the beginning), whereas this one has two additional entries in the middle
  - Entry at nodes B3 and B4
- How to convert this trace to super block?



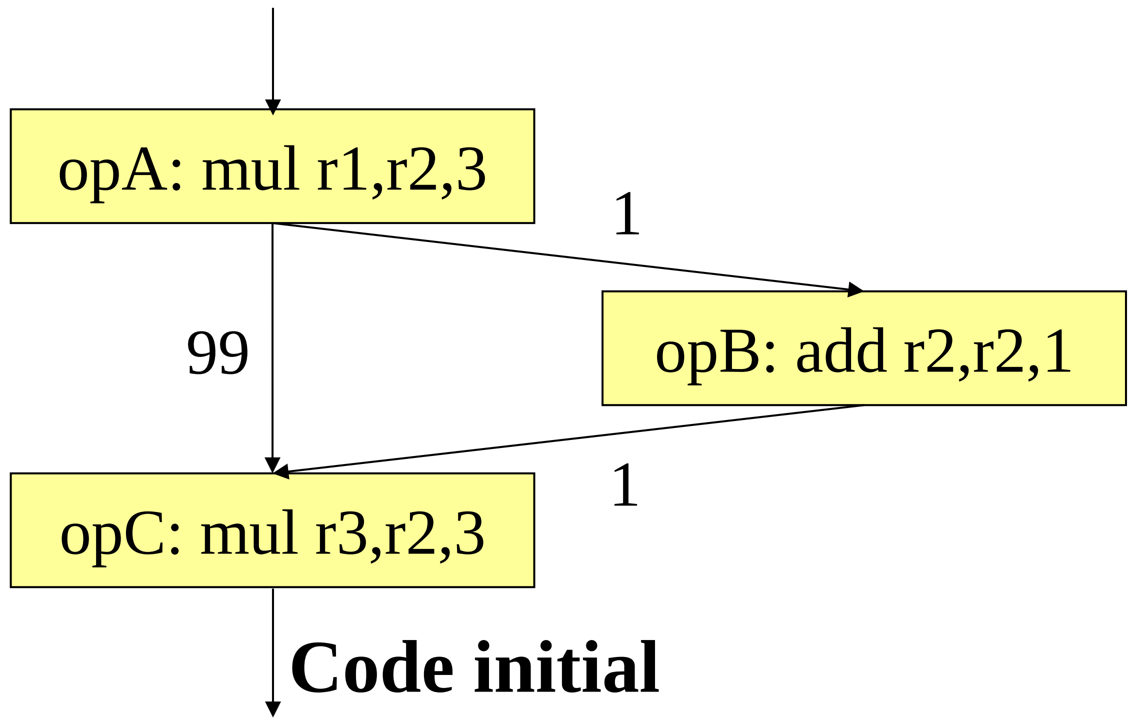
Super Block from Tail Duplication

- Tail duplication is a duplication of base blocks that are a destination of an unwanted entry point to form a super block
- Be careful, the code size can double!
- In a sense, tail duplication is similar to the notion of code clearing in a trace!

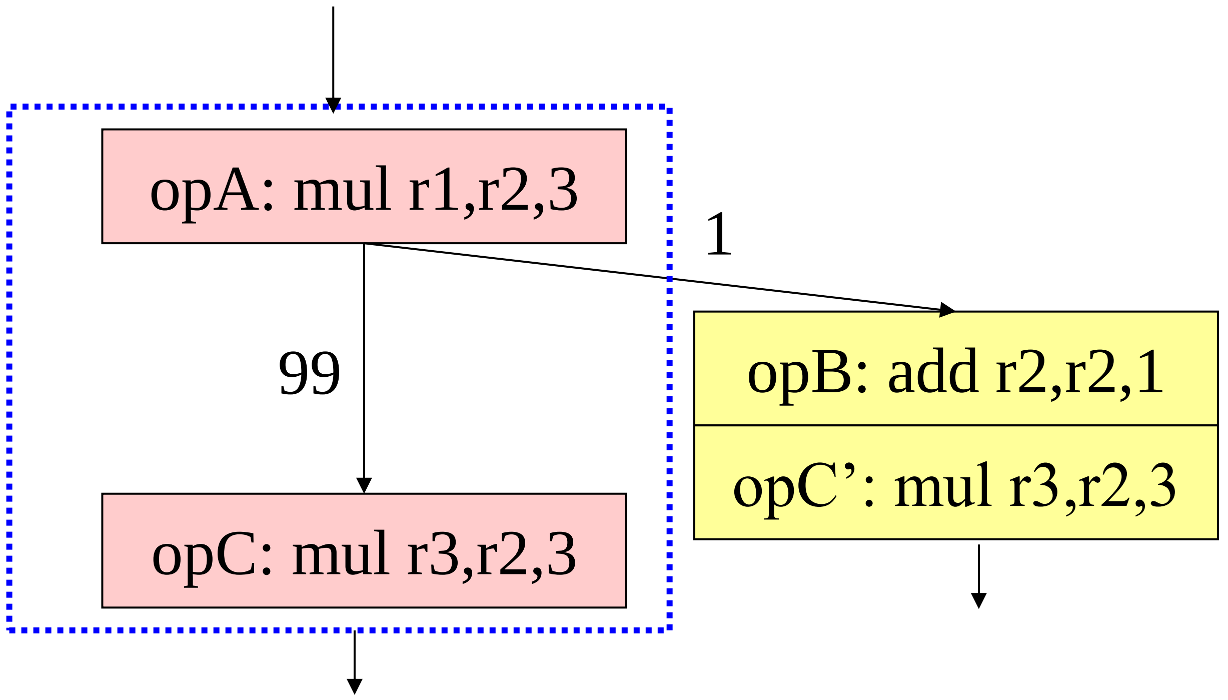
Super Block from Tail Duplication (2)



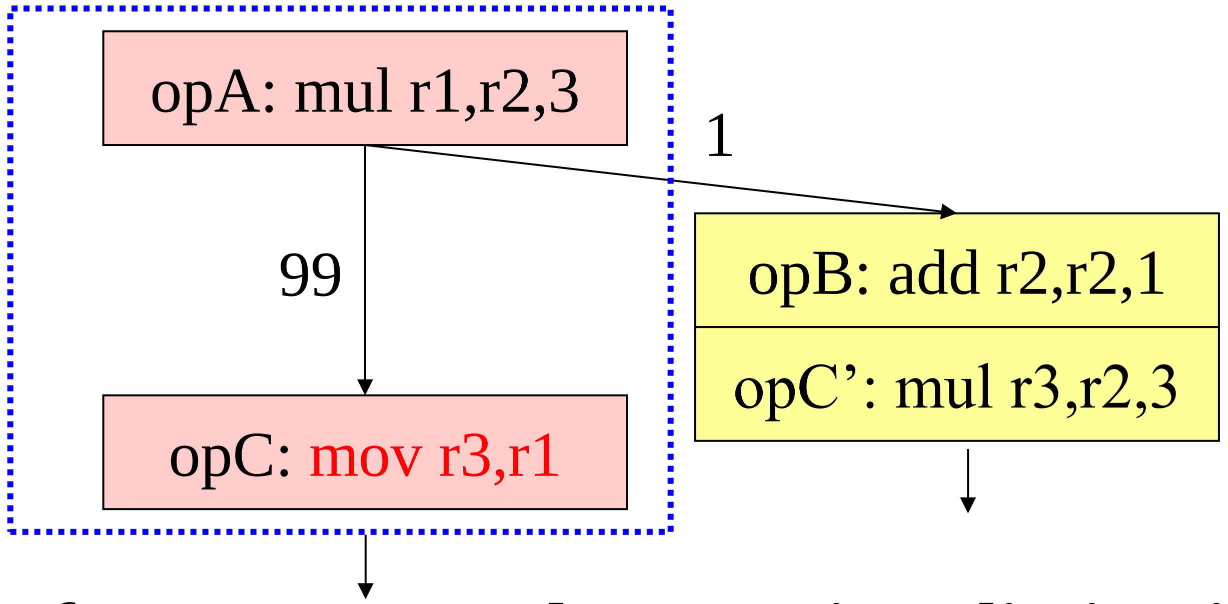
Elimination of Common Subexpressions



Code initial



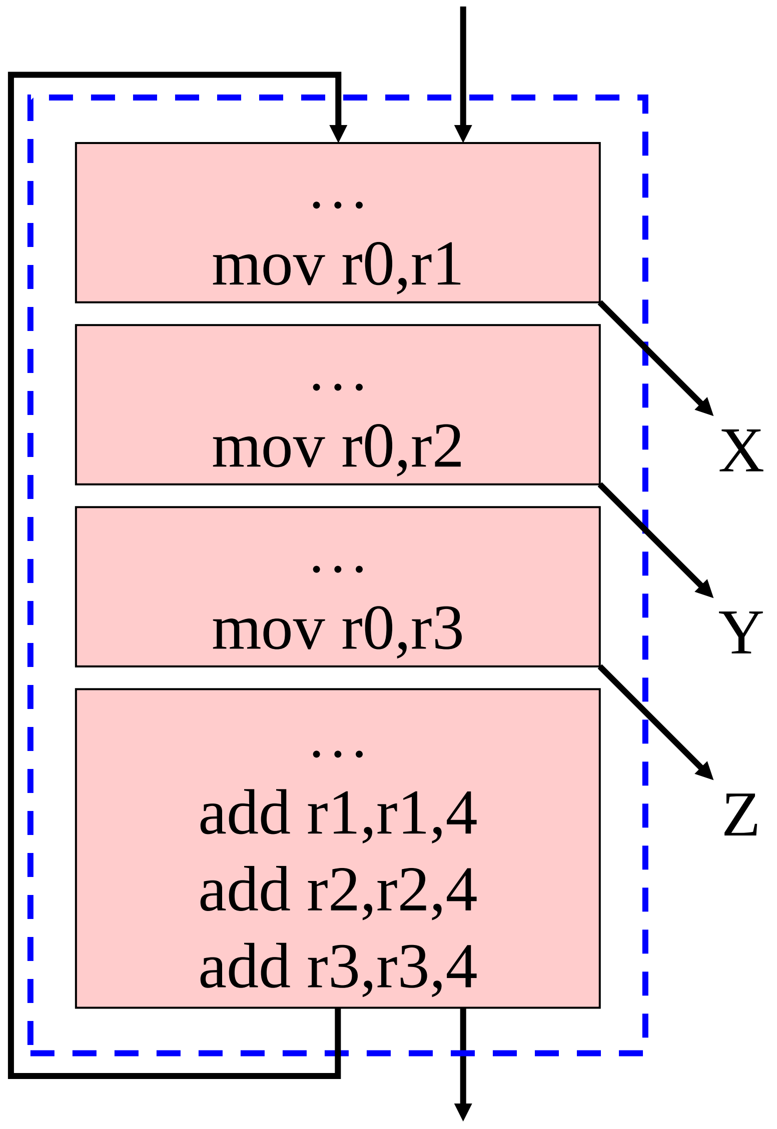
Code after formation of super block



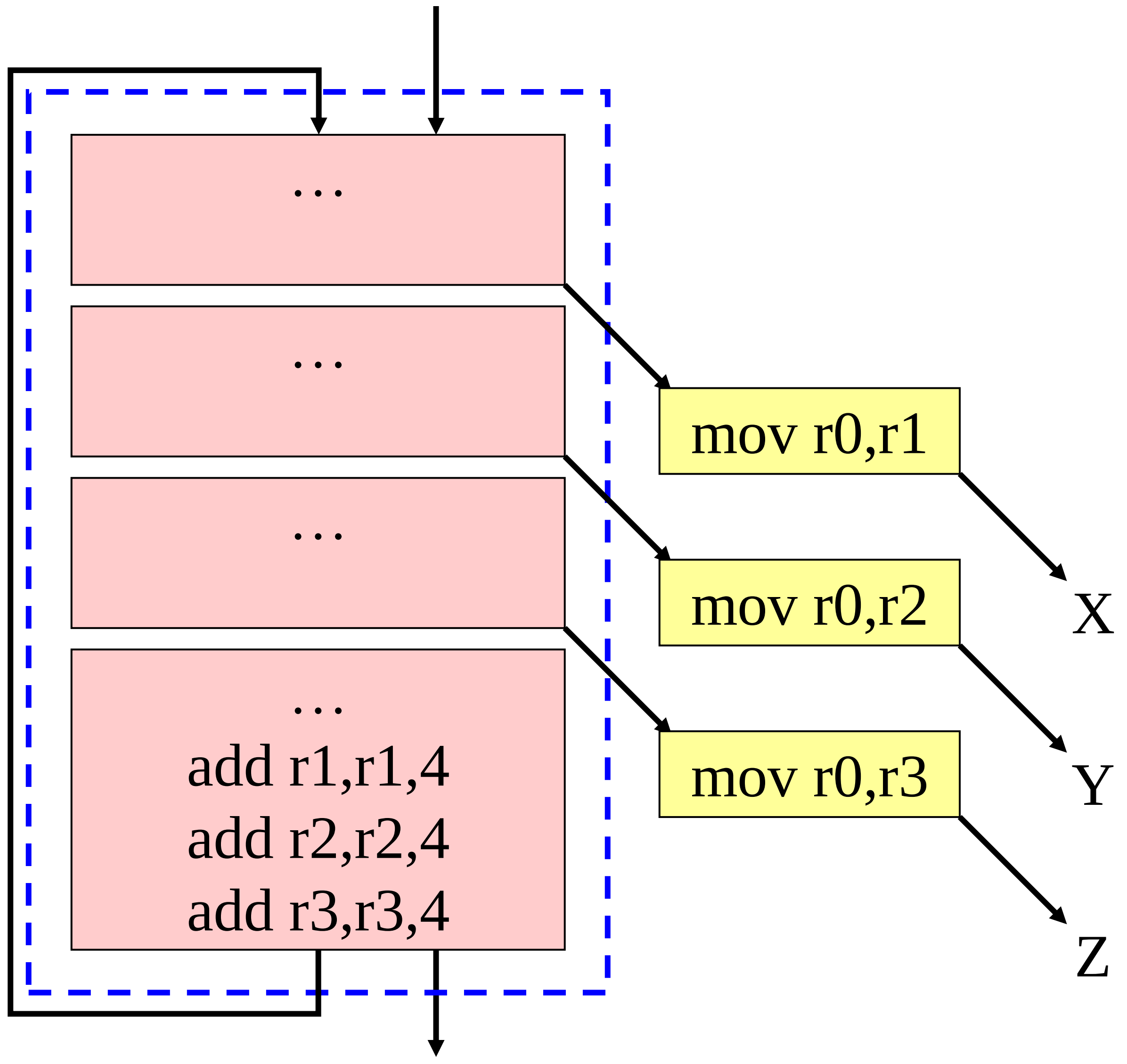
Code after common subexpression elimination

Source: J.N. Amaral

Moving Operations in a Super Block



Code original



Code after operation movement

Source: J.N. Amaral

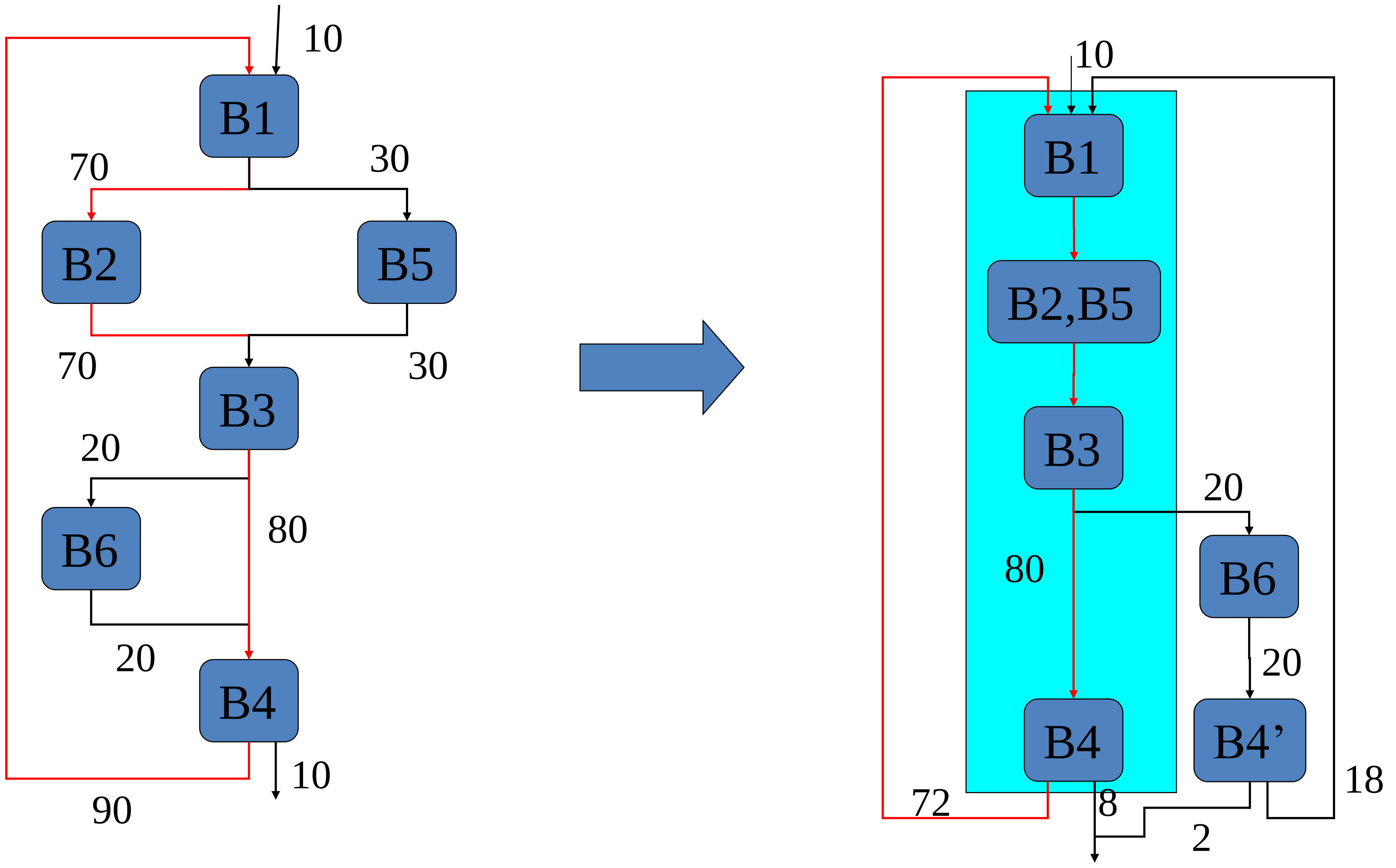
Optimizations with Super Blocks

- Existence of many operations in a super-block can take advantage of several ad-hoc optimizations
- Thus, experimental research has gone towards an enlargement of super blocks
  - In theory, the bigger the super block, the more optimization opportunities
- Disadvantages of large super blocks:
  - Code size
  - Instruction cache
  - Compilation time

Hyper Blocks

- It is a single-entry region with multiple exit points with an internal flow of control
- It is a variant of super blocks using prediction to fold several control paths into a single super block

Formation of a Hyper Block

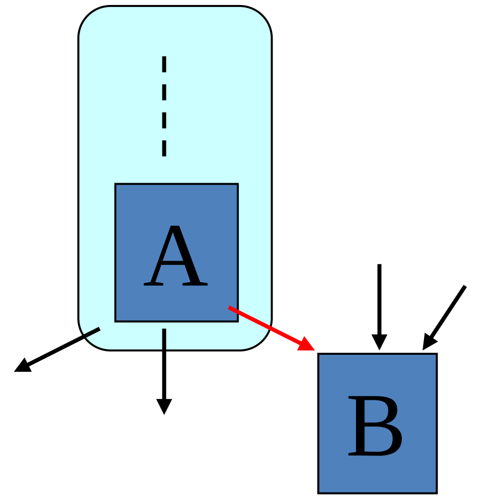


Global ILP scheduling to a region

- A function is decomposed into regions
- Scheduling the DAG of each region, as if it were a base block
- Enter the compensation code (possibly followed by a size optimization pass for this extra code)
- How to build a region?

Heuristic Training Regions of a Function

- Measure the execution frequencies of each basic block via profiling
- Shapes of the previous regions have a single point of entry
- Start forming a region from a head
  - For each new base block *A* added in the region, examine its successors *Bi*
  - If *Bi* is the most frequent successor of *A*, and if *A* is the most frequent predecessor of *Bi*, then the pair *(A, Bi)* is said to be mutually frequent
  - Add *Bi* to the trace
- Iterate until one of the following conditions is true:
  - Construction of a cycle
  - No more mutually frequent pair *(A, Bi)*
  - Other heuristic variant



Memory Hierarchy

- Until now
  - Decrease in number of instructions / calculations
  - Transformation to generate code with parallelism of instructions
- Need to consider the memory hierarchy
  - Cache levels (micro-architecture)
  - Notion of pagination and TLB (architecture / micro-architecture)
- Cache
  - Transfer memory to cache with an entire line (64o)
  - Reuse of the same or similar data
- TLB
  - Cache for pages and physical / virtual correspondence
  - Need to minimize the memory footprint of a piece of code to avoid TLB defects

Optimization for Memory Hierarchy

- List of optimizations for the memory hierarchy
  - Loop fusion
  - Loop fission
  - Switching loops (loop interchange)
  - Tiling
  - Preloading
- More generally
  - Manipulation of data structures

Fission or Distribution of Loops

- Consider the following loop

```
(1) for i=1 to N do
(2)     A[i] = A[i] + B[i-1]
(3)     B[i] = C[i-1]*X + Z
(4)     C[i] = 1/B[i]
(5)     D[i] = sqrt(C[i])
(6) endfor
```

Initial Loop

$$\begin{tikzcd}
S2 \\
S3 \arrow[u, bend left] \arrow[d] \\
S4 \arrow[u, bend left] \arrow[d] \\
S5
\end{tikzcd}$$

Dependence graph

Fission or Distribution of Loops (2)

- Breaking a "big" loop into small loops

```
(1) for i=1 to N do
(2)     A[i] = A[i] + B[i-1]
(3)     B[i] = C[i-1]*X + Z
(4)     C[i] = 1/B[i]
(5)     D[i] = sqrt(C[i])
(6) endfor
```

Initial loops

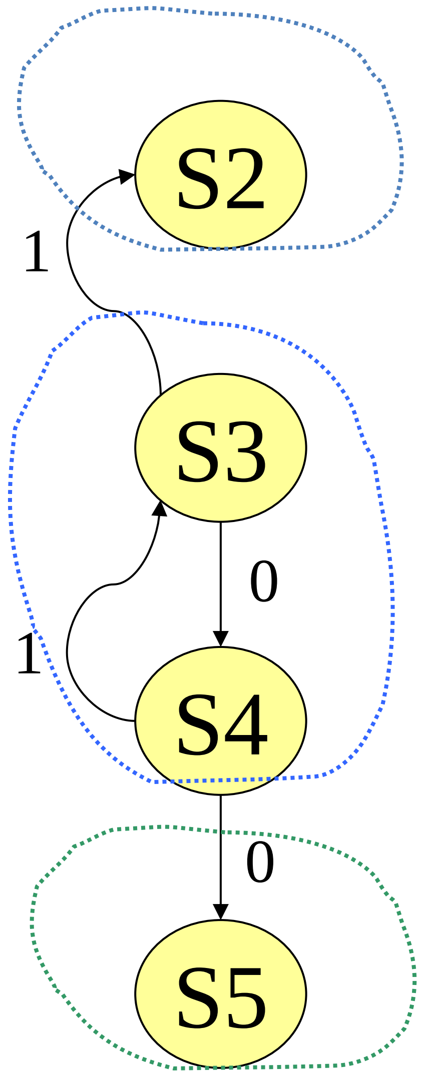
```
(1) for i=1 to N do
(3)     B[i] = C[i-1]*X + Z
(4)     C[i] = 1/B[i]
(6) endfor
(1) for i=1 to N do
(2)     A[i] = A[i] + B[i-1]
(6) endfor
(1) for i=1 to N do
(5)     D[i] = sqrt(C[i])
(6) endfor
```

After loop fission

Legal Loop Fission

- It is necessary to decompose the data dependence graph (DDG) into strongly related components
- Each loop resulting from the fusion corresponds to a strongly connected component of the DDG
- Order of the fission loops is defined by the order of the strongly connected components of the DDG

Legal Loop Fission (2)



Data dependence graph

```
(1) for i=1 to N do
(2)     A[i] = A[i] + B[i-1]
(3)     B[i] = C[i-1]*X + Z
(4)     C[i] = 1/B[i]
(5)     D[i] = sqrt(C[i])
(6) endfor
```

Initial loop

```
(1) for i=1 to N do
(3)     B[i] = C[i-1]*X + Z
(4)     C[i] = 1/B[i]
(6) endfor
(1) for i=1 to N do
(2)     A[i] = A[i] + B[i-1]
(6) endfor
(1) for i=1 to N do
(5)     D[i] = sqrt(C[i])
(6) endfor
```

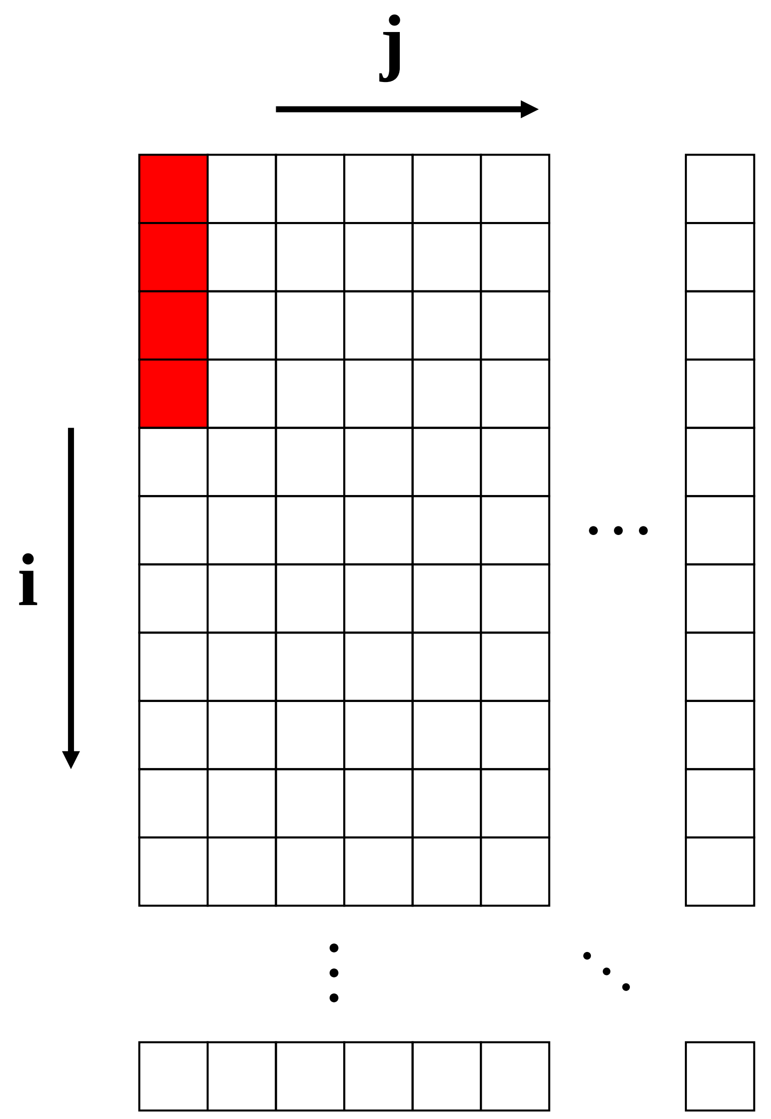
After loop fission

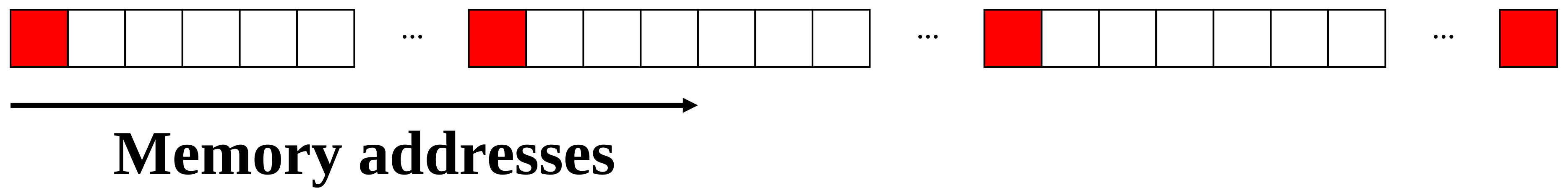
Switching Loops (Loop Interchange)

- Permutation of two nested loops can improve the spatial locality of the memory accesses
- Consider the following code

```
for (j = 0; j < 100; j = j+1)
   for (i = 0; i < 5000; i = i+1)
         x[i][j] = 2 * x[i][j];
```

  - Assume column-major addressing
  - All accesses to x [:] [j] will result in cache misses





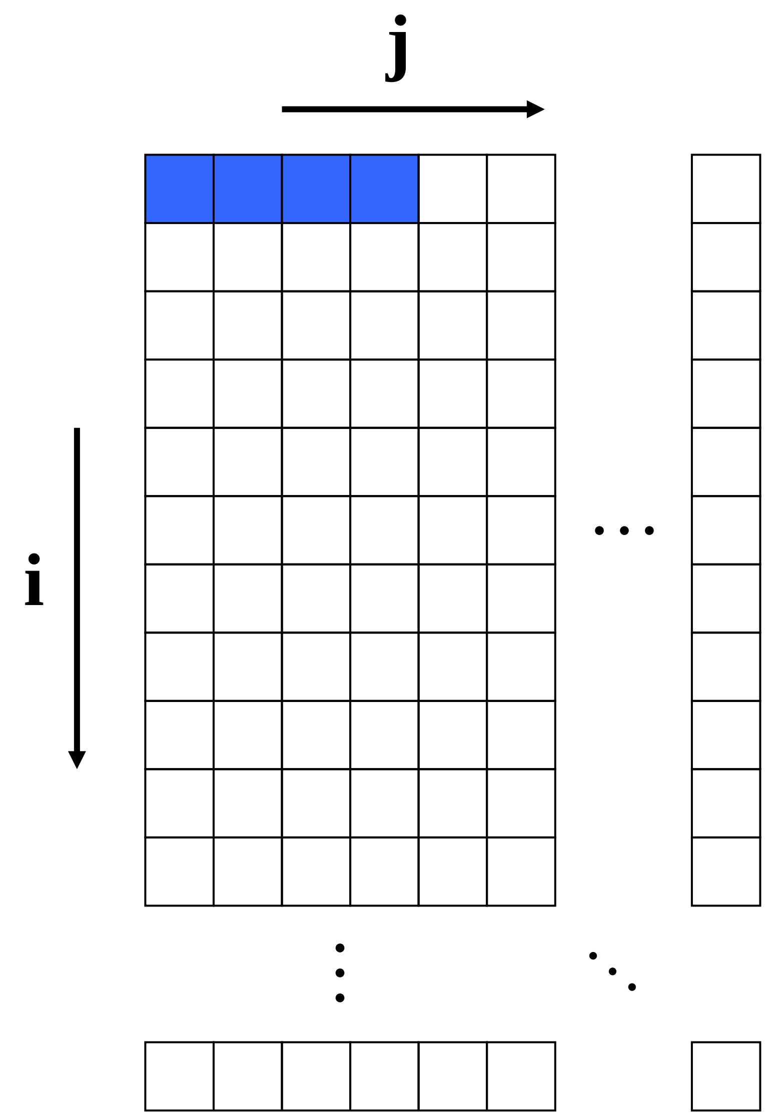
Matrice x

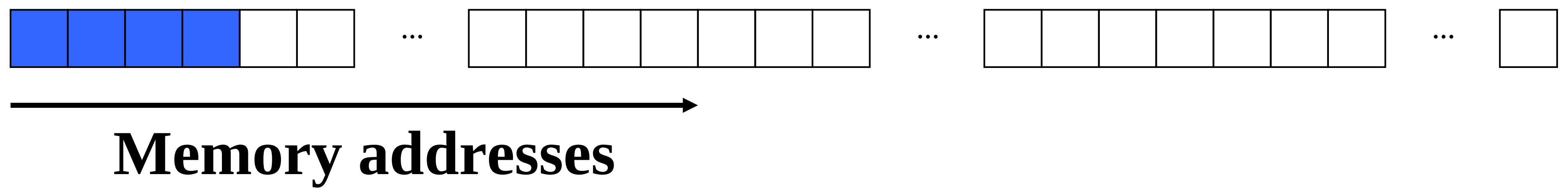
Switching Loops (Loop Interchange) (2)

- Permutation of two nested loops can improve the spatial locality of the memory accesses
- Permute the i and j loops

```
for (i = 0; i < 5000; i = i+1)
  for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

  - Assume cache block of 4
  - Now, every accesses to x [i] [j mod 4 = 0] will be a miss





Tiling

- Principle
  - Reduce the workspace on a particular block to take advantage of data reuse
  - Create a working *tile*
  - Iterate on this tile for all the iterations
- Example with 2D filter

```
for ( i = 0 ; i < N ; i++) {
  for ( j = 0 ; j < M ; j++ ) {
    a[i][j] = b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1] ;
  }
}
```

Tiling (2)

- Required Transformations
  - Strip Mining (or blocking) on the external loop
    - Strip mining on the inner loop
- Switching middle loops
- Settings
  - Size of tile
  - Depends on data cache
  - Depends on memory access in the body of the loop nest

Preloading

- Idea is for the processor to fetch data in advance
  - To avoid cache defects
  - Need to recognize data streams (flows)
  - Launch preloading of data flow
- Should not go beyond a memory page (why?)
- Ability to do this at the software level
  - Advance Loading Instructions
  - Special pre-loading instructions
- Advantages disadvantages ?

SIMD

- Architecture trend is to give the process more instructions to process SIMD operations
  - Operation of the SIMD model
  - Extension of vector units (e.g., 512bits on Xeon Phi)
- Architectural Mechanism
  - New instructions in the ISA
- How to exploit it?
  - Within a BB?
  - Between BB?
  - Between function?
  - …

Exploiting SIMD

- Idea
  - Matching of similar operations on different data
  - Need to identify independent arithmetic operations
- Within a basic block
  - Matching local operations
- In a loop
  - Ability to enjoy several consecutive iterations
  - Use unwinding to expose several iterations
  - Use loop next fusion (unroll & jam) to help bring closer the instructions
- Example with GCC

Conclusion

- A set of mechanisms to help execute codes on a calculation core
  - Micro-architecture helps, but need to consider it
  - Architecture offers the possibility to the compiler / developer to optimize the program
- Wide spectrum of transformations to optimize the program
  - Analysis: data, control
  - Transformations: enabling transformations to reformulate the code
  - Optimization
- Combination of these complex transformations!