Unit - IV

Code Optimization | Data Flow Analysis

Code Optimization

- Consideration for Optimization
- Scope of Optimization
- Local Optimization
- Loop Optimization
- Frequency Reduction
- Folding
- DAG Representation

Data Flow Analysis

- Flow Graph
- Data Flow Equation
- Global Optimization
- Redundant Sub Expression Elimination
- Induction Variable Elements
- Live Variable Analysis
- Copy Propagation

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Code Optimization

Code Optimization

- Consideration for Optimization
- Scope of Optimization
- Local Optimization
- Loop Optimization
- Frequency Reduction
- Folding
- DAG Representation

Optimization

- **Optimization** = transformation that improves the performance of the target code
- **Optimization**
  - must not change the output
  - must not cause errors that were not present in the original program
  - must be worth the effort (profiling often helps).
- Which optimizations are most important depends on the program, but generally, loop optimizations, register allocation and instruction scheduling are the most critical.
- Local optimizations : within Basic Blocks
- Superlocal optimizations : within Extended Basic Blocks
- Global optimizations: within Flow Graph

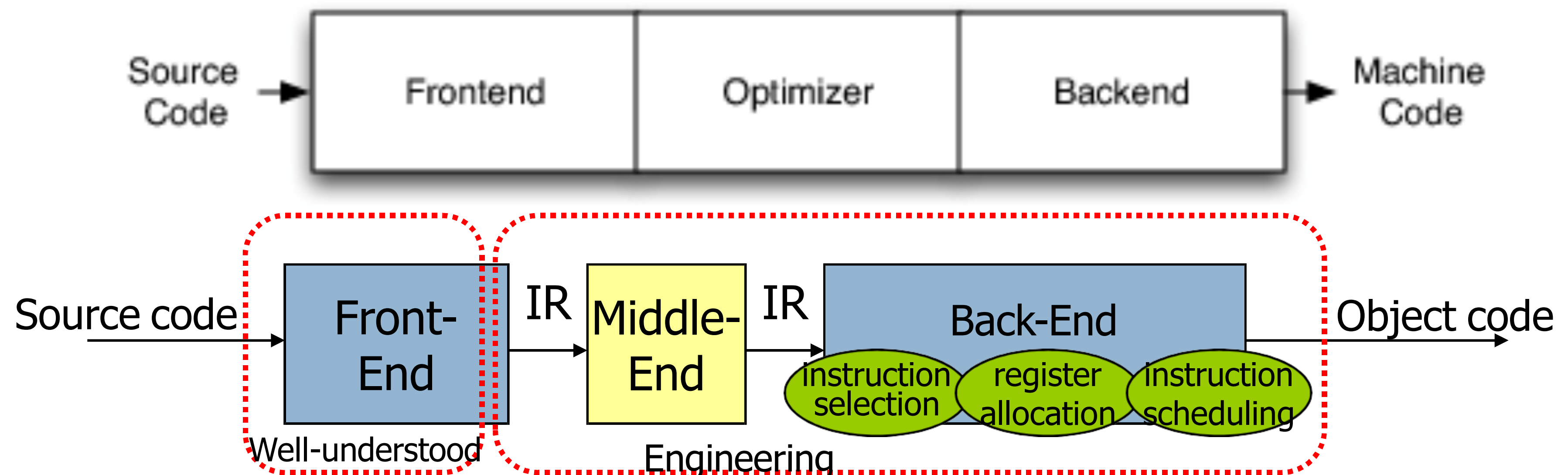
Code Optimization

Introduction:

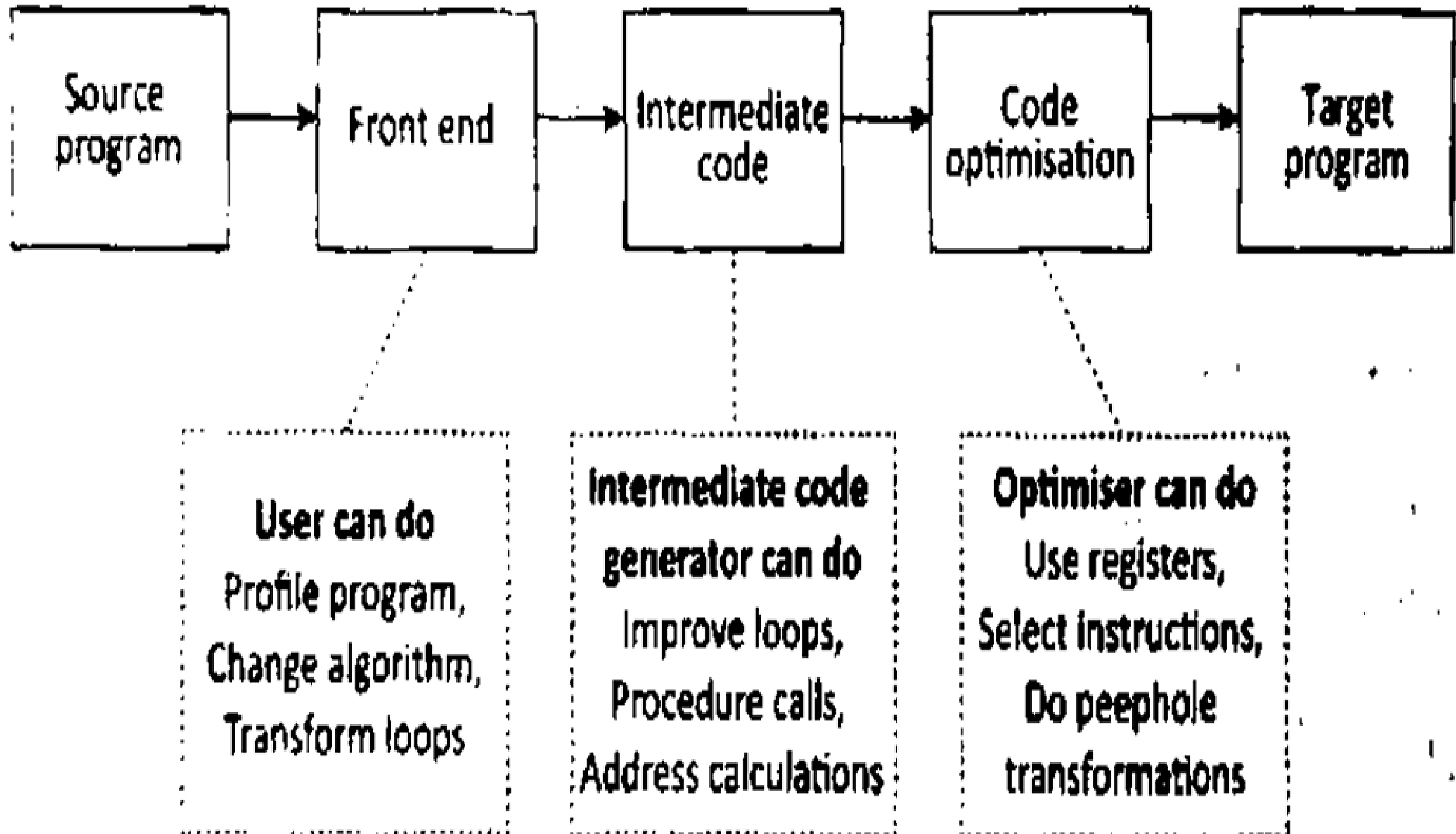
- Optimized code
  - Executes faster
  - efficient memory usage
  - yielding better performance.
- Compilers can be designed to provide code optimization.
- Users should only focus on optimizations not provided by the compiler such as choosing a faster and/or less memory intensive algorithm.

Code Optimization

- A Code optimizer sits between the front end and the code generator.
  - Works with intermediate code.
  - Can do control flow analysis.
  - Can do data flow analysis.
  - Does transformations to improve the intermediate code.



Code Optimization



Code Optimization

- Goal: improve program performance within some constraints.
  - (may also reduce size of the code, power consumption, etc…)
- Issues:
  - Legality: must preserve the meaning of the program.
  - Externally observable meaning may be sufficient/may need flexibility.
  - Benefit: must improve performance on average or common cases.
  - Predicting program performance is often non-trivial.
  - Compile-time cost justified: list of possible optimisations is huge.
  - Interprocedural optimisations.

Code Optimization Classification

- By Scope:
  - Local Optimization: within a single basic block.
  - Peephole Optimization : on a window of instructions (usually local)
  - Loop-level Optimization : on one or more loops or loop nests.
  - Global: for an entire procedure
  - Interprocedural: across multiple procedures or whole program.
- By machine information used:
  - Machine-independent versus machine-dependent.
- By effect on program structure:
  - Algebraic transformations (e.g., `x+0`, `x*1`, `3*z*4`, …)
  - Reordering transformations (change the order of 2 computations)
    - Loop transformations: loop-level reordering transformations.

Code Optimization Classification

- Machine-dependent optimization
  - It has some special machine properties that can reduce the amount of code or increases the execution time.
  - Ex: Register allocation and utilization of special machine instruction sequences.
- Machine-independent optimization
  - It depends only on the algorithms or arithmetic operations in the language and not on the target machine

Machine - Independent Optimizations

- Principle Sources of Optimization:
  1. Preserves the semantics of the original program
  2. Applies relatively low-level semantic transformations
- Data-Flow Analysis

Machine - Independent Optimizations

- Principle Sources of Optimization:
  1. Causes of Redundancy
  2. Semantics-Preserving Transformations
  3. Global Common Subexpressions
  4. Copy Propagation
  5. Dead-Code Elimination
  6. Code Motion
  7. Induction Variables and Reduction in Strength

Principle Sources of Optimization

1. **Causes of Redundancy**
   - Redundant operations are
     - at the source level
     - a side effect of having written the program in a high-level language
   - Each of high-level data-structure accesses expands into a number of low-level arithmetic operations
   - Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves.
   - By having a compiler eliminate the redundancies
     - The programs are both efficient and easy to maintain.

Principle Sources of Optimization

1. **Causes of Redundancy: A Running Example (Quicksort)**

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Fig: C code for quicksort

Principle Sources of Optimization

1. **Causes of Redundancy: A Running Example (Quicksort)**

```
(1)   i = m-1
(2)   j = n
(3)   t1 = 4*n
(4)   v = a[t1]
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
(9)   j = j-1
(10)  t4 = 4*j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
(13)  if i>=j goto (23)
(14)  t6 = 4*i
(15)  x = a[t6]
(16)  t7 = 4*i
(17)  t8 = 4*j
(18)  t9 = a[t8]
(19)  a[t7] = t9
(20)  t10 = 4*j
(21)  a[t10] = x
(22)  goto (5)
(23)  t11 = 4*i
(24)  x = a[t11]
(25)  t12 = 4*i
(26)  t13 = 4*n
(27)  t14 = a[t13]
(28)  a[t12] = t14
(29)  t15 = 4*n
(30)  a[t15] = x
```

Fig: Three- address code for above C fragment

Principle Sources of Optimization

1. **Causes of Redundancy: A Running Example (Quicksort)**

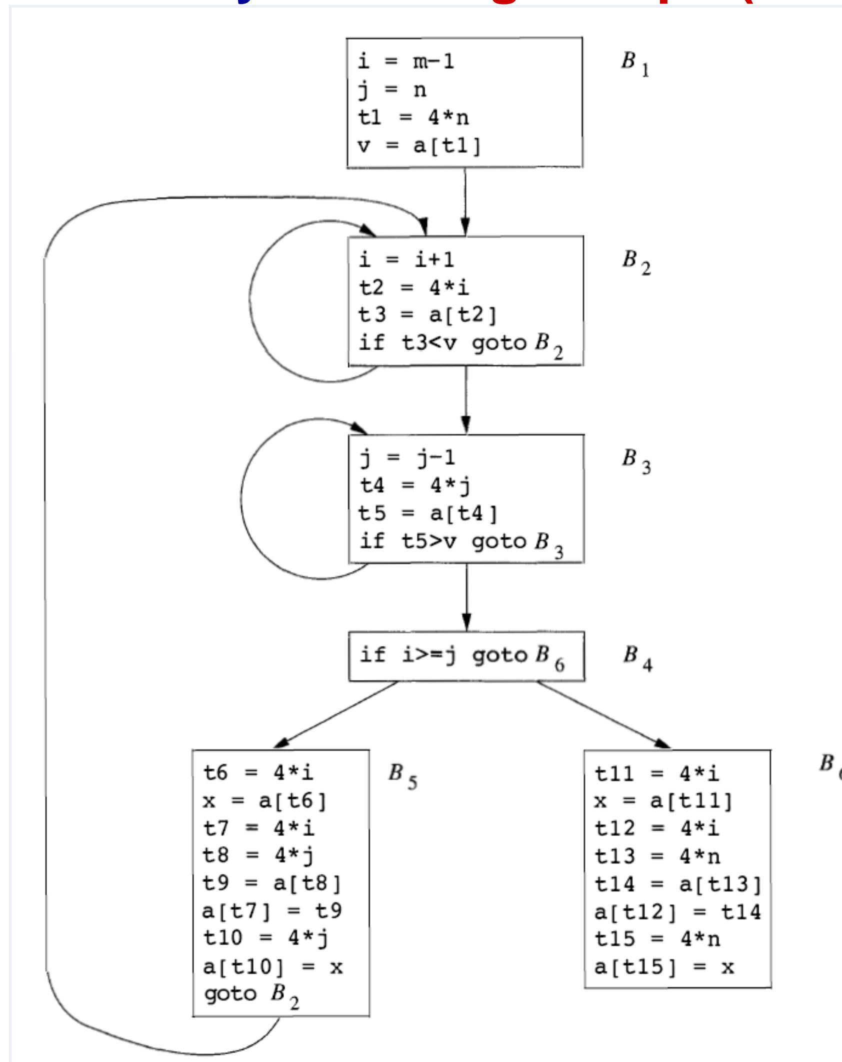


Fig: Flowgraph for above three-address code

Principle Sources of Optimization

2. **Semantics-Preserving Transformations**

- There are a number of ways in which a compiler can improve a program without changing the function it computes:
  - Common-sub expression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding

  are common examples of such function-preserving (or semantic-preserving) transformations.

Principle Sources of Optimization

3. **Common Subexpression Elimination**

- An occurrence of an expression E is called a *common subexpression* if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.

- An expression, say x+y, is redundant iff along every path from the procedure's entry it has been evaluated and its constituent subexpressions (x, y) have not been redefined.

Principle Sources of Optimization

3. Common Subexpression Elimination

Local:

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B_5

(a) Before.

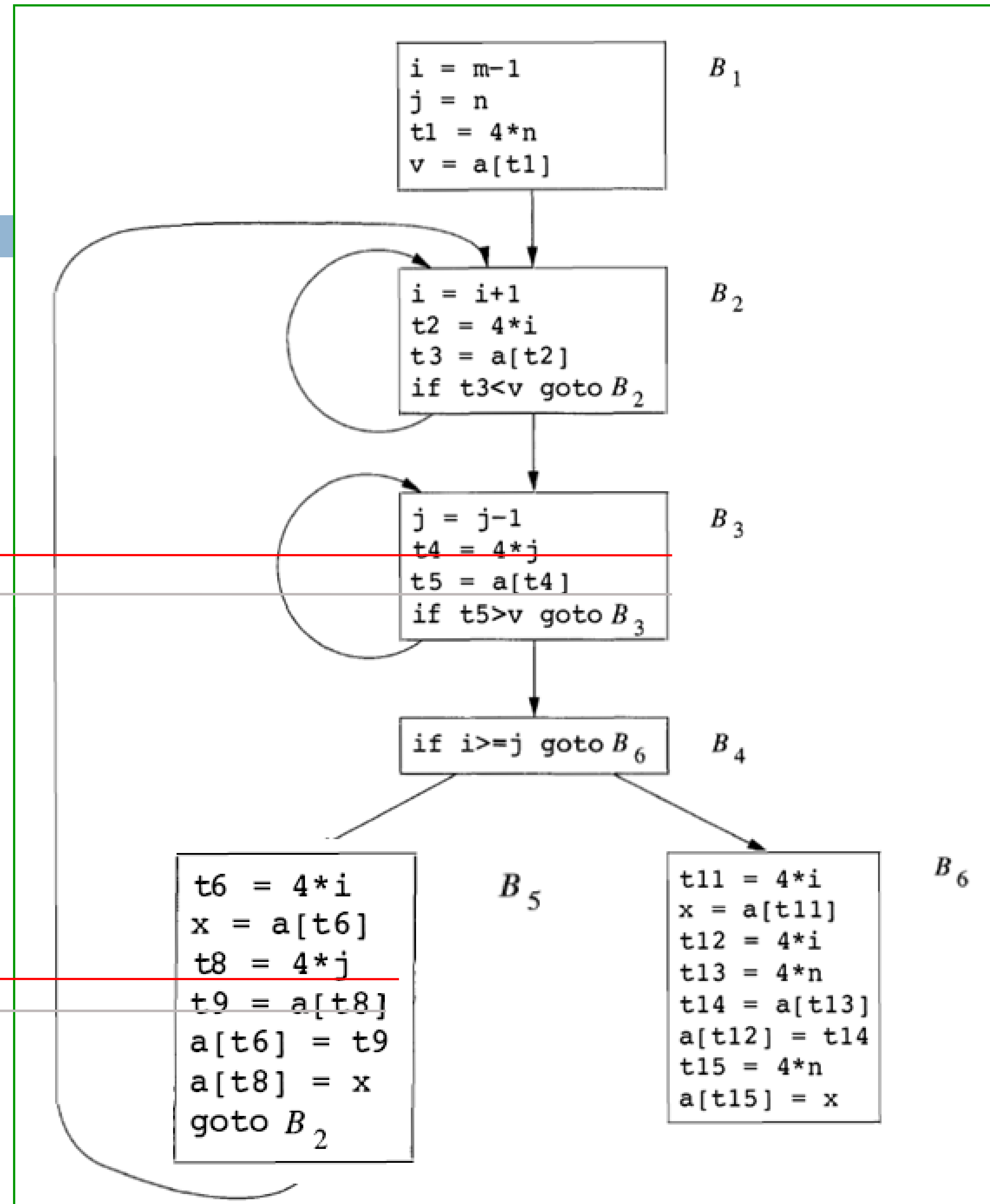
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

(b) After.

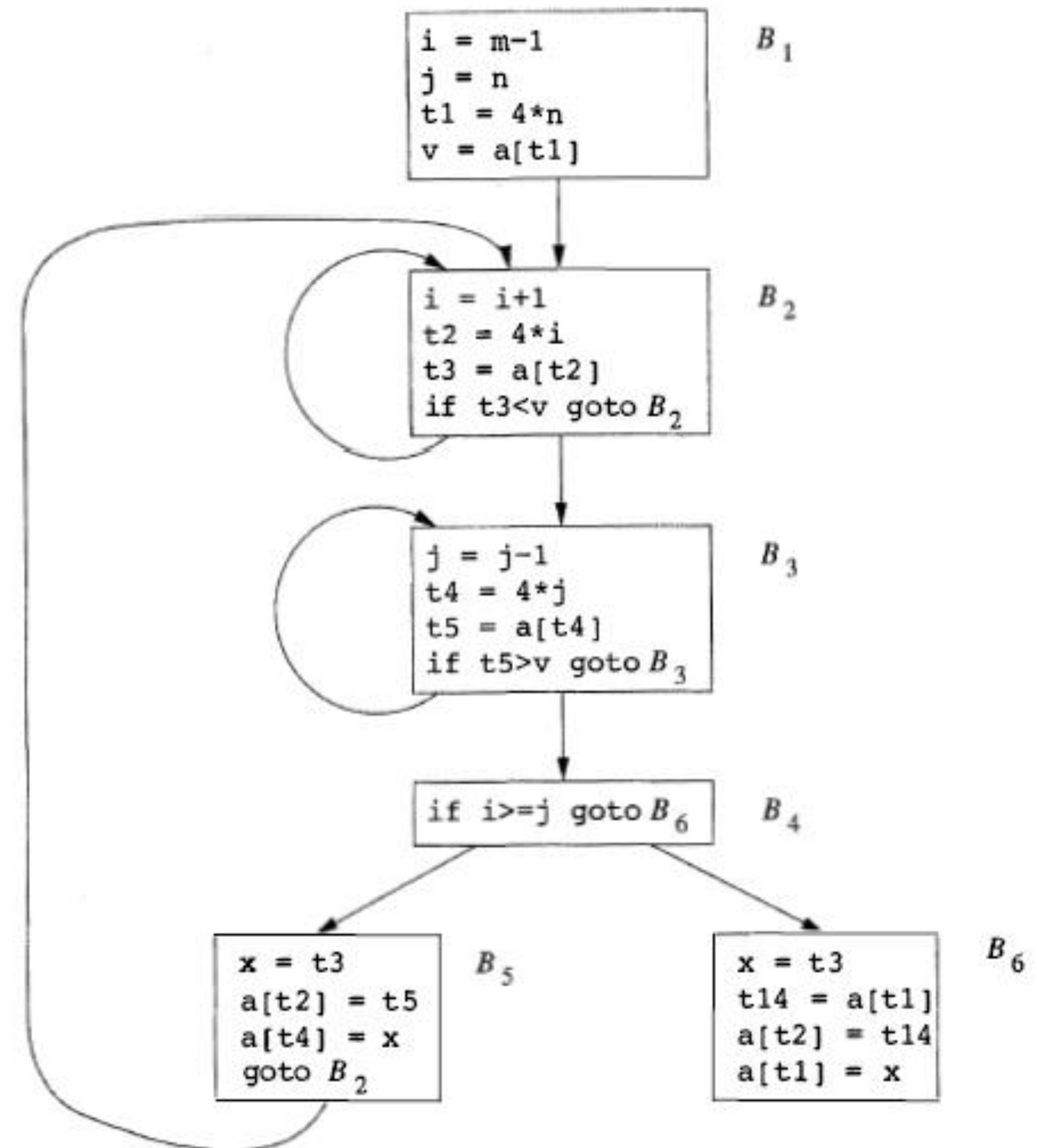
3. Common Subexpression Elimination

Global:



Principle Sources of Optimization

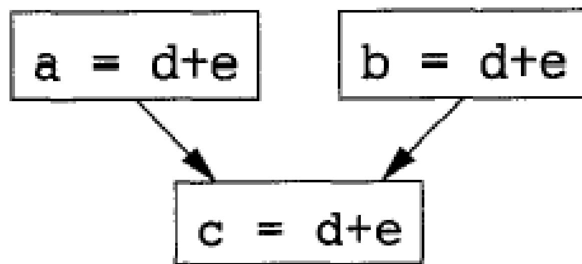
3. **Common Subexpression Elimination**



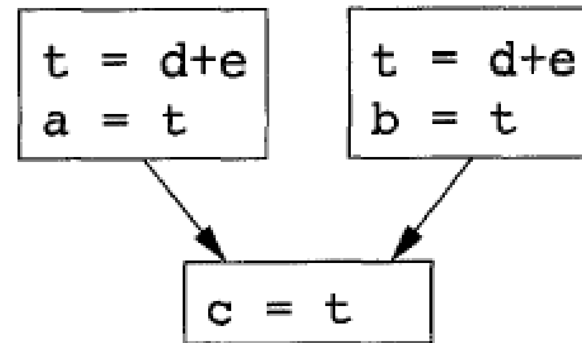
Principle Sources of Optimization

4. **Copy Propagation**

- Copy statements or Copies
  - u = v



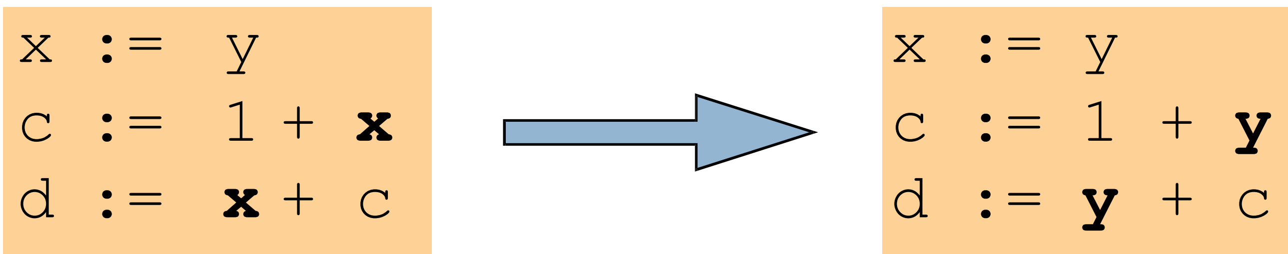
(a)



(b)

Principle Sources of Optimization

- for a statement x := y
- replace later uses of x with y, if x and y have not been changed.



Analysis needed, as y and x can be assigned more than once!

Principle Sources of Optimization

5. **Dead-Code Elimination**

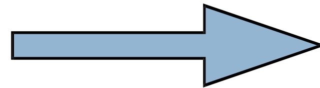
- *Live variable*
  - A variable is live at a point in a program if its value can be used subsequently;
  - otherwise, it is dead at that point.
- *Constant folding*
  - Deducing at compile time that the value of an expression is a constant and using the constant instead

Principle Sources of Optimization

5. Dead-Code Elimination

- Remove unnecessary code
- e.g. variables assigned but never read

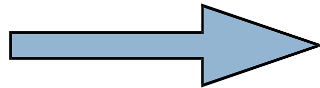
```
b := 3
c := 1 + 3
d := 3 + c
```



```
c := 1 + 3
d := 3 + c
```

- Remove code never reached

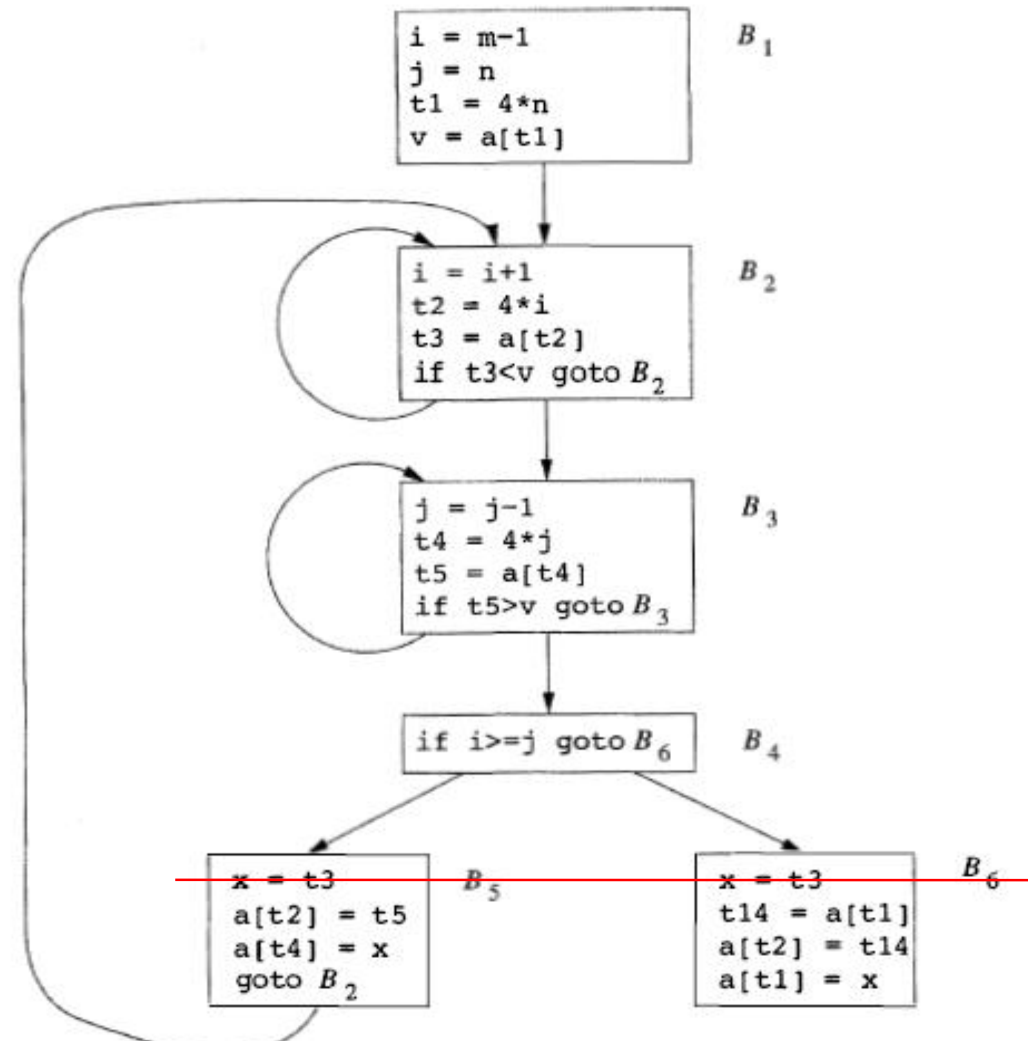
```
if (false)
{a := 5}
```



```
if (false)
{}
```

Principle Sources of Optimization

5. **Dead-Code Elimination**



Principle Sources of Optimization

6. **Code Motion**

- An important modification that decreases the amount of code in a loop
- *Loop-invariant computation*
  - An expression that yields the same result independent of the number of times a loop is executed
- Code Motion takes loop-invariant computation before its loop

```
while (i <= limit-2)
```

```
t = limit -2 while
(i <= t)
```

Principle Sources of Optimization

7. **Induction Variables and Reduction in Strength**

- Induction variable
  - For an induction variable x, there is a positive or negative constant c such that each time x is assigned, its value increases by c
- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration
- Strength reduction
  - The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition
- Induction variables lead to
  - strength reduction
  - eliminate computation

Principle Sources of Optimization

Strength Reduction

- Replace expensive operations with simpler ones
- Example: Multiplications replaced by additions

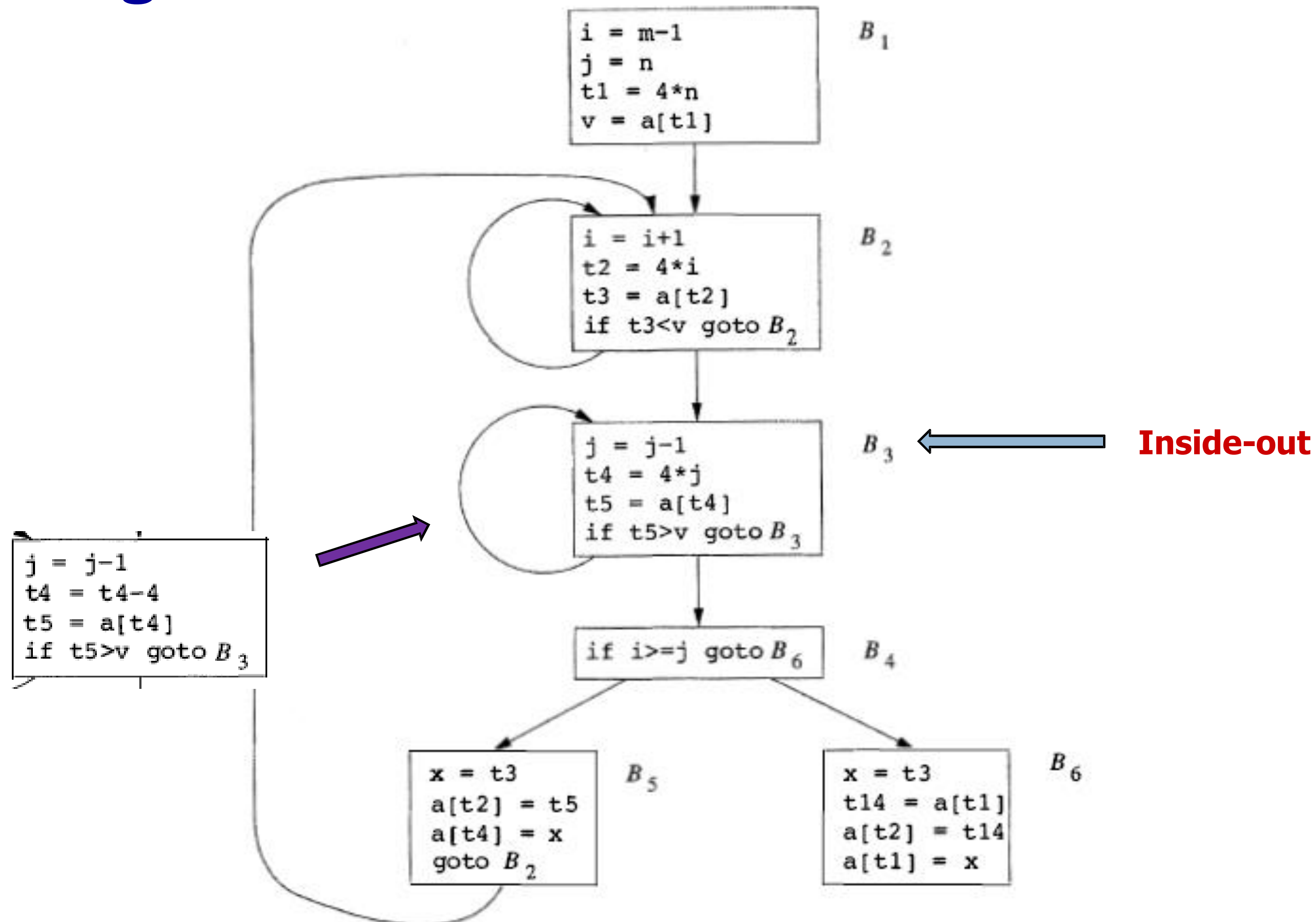
```
y := x * 2
```

```
y := x + x
```

Peephole optimizations are often strength reductions

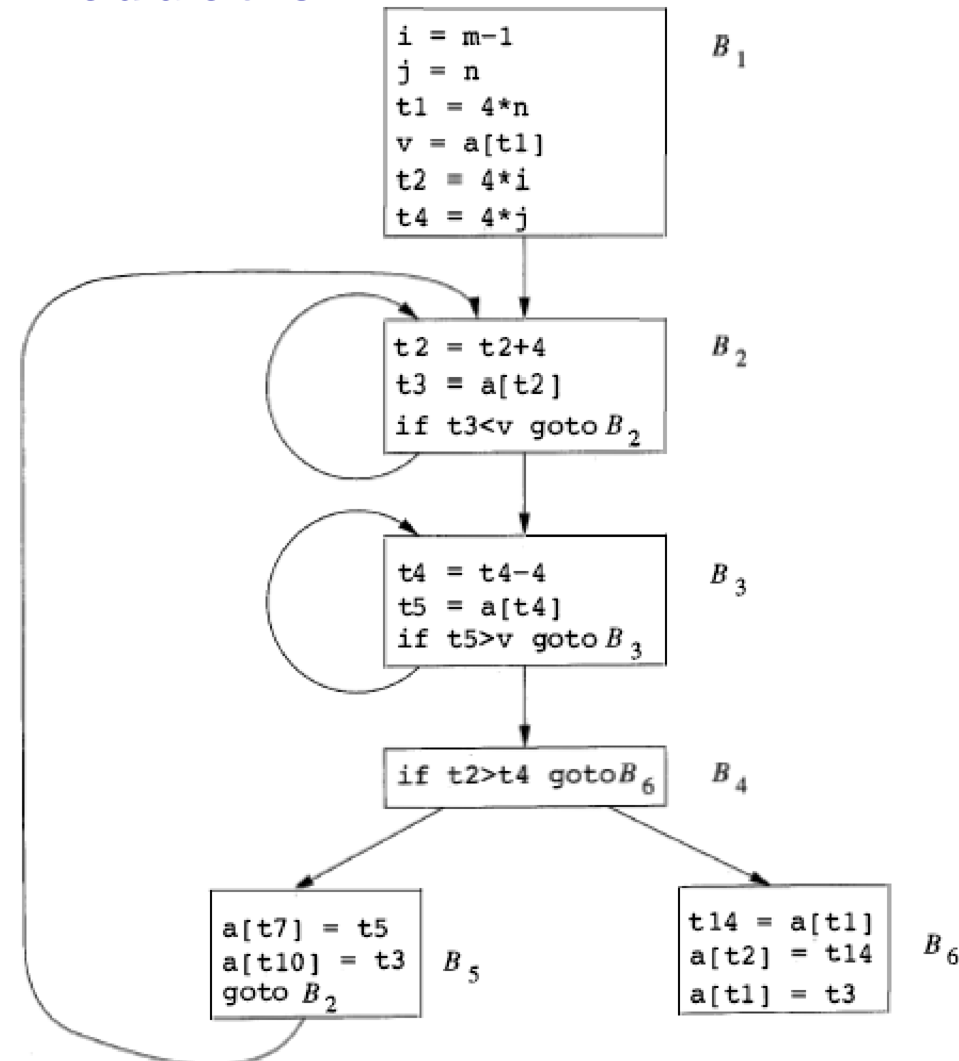
Principle Sources of Optimization

Strength Reduction



Principle Sources of Optimization

Strength Reduction



Principle sources of optimizations: Examples

Before optimisation	After optimisation
// Common subexpression elimination `A[I,I*2+10]=B[I,I*2+10]+5`	`tmp=I*2+10` `A[I,tmp]=B[I,tmp]+5`

// Copy propagation
`t=I*4   s=t`
`a[s]=a[s]+4`
	`t=I*4   s=t`
	`a[t]=a[t]+4`

// Constant propagation
`N=64`
`c=2`
`for (I=0;I<N;I++)`
`a[I]=a[I]+c`
	`N=64`
	`c=2`
	`for (I=0;I<64;I++)`
	`a[I]=a[I]+2`

// Constant folding `tmp=5*3+8-12/2`	`tmp=17`

// Dead-code elimination
`if (3>7) then { … }`
	`// removed (some of the`
	`// above optimisations may`
	`` // create `useless' code…) ``

// Reduction in strength `x*2+x*1024`	`x+x+(x<<10)`