Compiler Optimizations

Unnikrishnan C

September 15, 2019

Optimizations

- Examples
  - Global common sub-expression elimination

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction
  - Loop unrolling and Function Inlining

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction
  - Loop unrolling and Function Inlining
  - Tail recursion removal.

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction
  - Loop unrolling and Function Inlining
  - Tail recursion removal.
  - Vectorization (for SIMD processor)

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction
  - Loop unrolling and Function Inlining
  - Tail recursion removal.
  - Vectorization (for SIMD processor)
  - Parallelization (for Multi-core processors)

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction
  - Loop unrolling and Function Inlining
  - Tail recursion removal.
  - Vectorization (for SIMD processor)
  - Parallelization (for Multi-core processors)
  - Loop Interchange

Optimizations

- Examples
  - Global common sub-expression elimination
  - Copy propagation
  - Constant propagation and constant folding
  - Induction variable elimination and strength reduction
  - Loop unrolling and Function Inlining
  - Tail recursion removal.
  - Vectorization (for SIMD processor)
  - Parallelization (for Multi-core processors)
  - Loop Interchange
- LLVM Assignment: Local Optimizations (within a BasicBlock)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)
  - Tail recursion (removing recursion)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)
  - Tail recursion (removing recursion)
  - Function inlining (to remove space and time overhead of function call)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)
  - Tail recursion (removing recursion)
  - Function inlining (to remove space and time overhead of function call)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)
  - Tail recursion (removing recursion)
  - Function inlining (to remove space and time overhead of function call)
  - Loop interchange (for cache locality)

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)
  - Tail recursion (removing recursion)
  - Function inlining (to remove space and time overhead of function call)
  - Loop interchange (for cache locality)
- LLVM Assignment: First three optimizations comes within local optimizations.

- Description
  - Copy propagation (remove a statement **x := y**, replace all uses of **x** by **y**)
  - Constant propagation (if a variable is having constant value, replace all uses with constant value)
  - Strength Reduction (e.g. Replace x= y*2 by x=y+y)
  - Induction variable Elimination (for loops)
  - Tail recursion (removing recursion)
  - Function inlining (to remove space and time overhead of function call)
  - Loop interchange (for cache locality)
- LLVM Assignment: First three optimizations comes within local optimizations.
- LLVM Assignment: local optimizations not limited to the first three optimizations.

Code Optimizations

- Code optimization needs following information

Code Optimizations

- Code optimization needs following information
  - Which definition reach a point.

Code Optimizations

- Code optimization needs following information
  - Which definition reach a point.
  - Which expression are recomputed.

Code Optimizations

- Code optimization needs following information
  - Which definition reach a point.
  - Which expression are recomputed.
  - Which copies and constants can be propogated, and many more.

Code Optimizations

- Code optimization needs following information
  - Which definition reach a point.
  - Which expression are recomputed.
  - Which copies and constants can be propogated, and many more.
  - *A defintion kills value of the defined variable.*

Code Optimizations

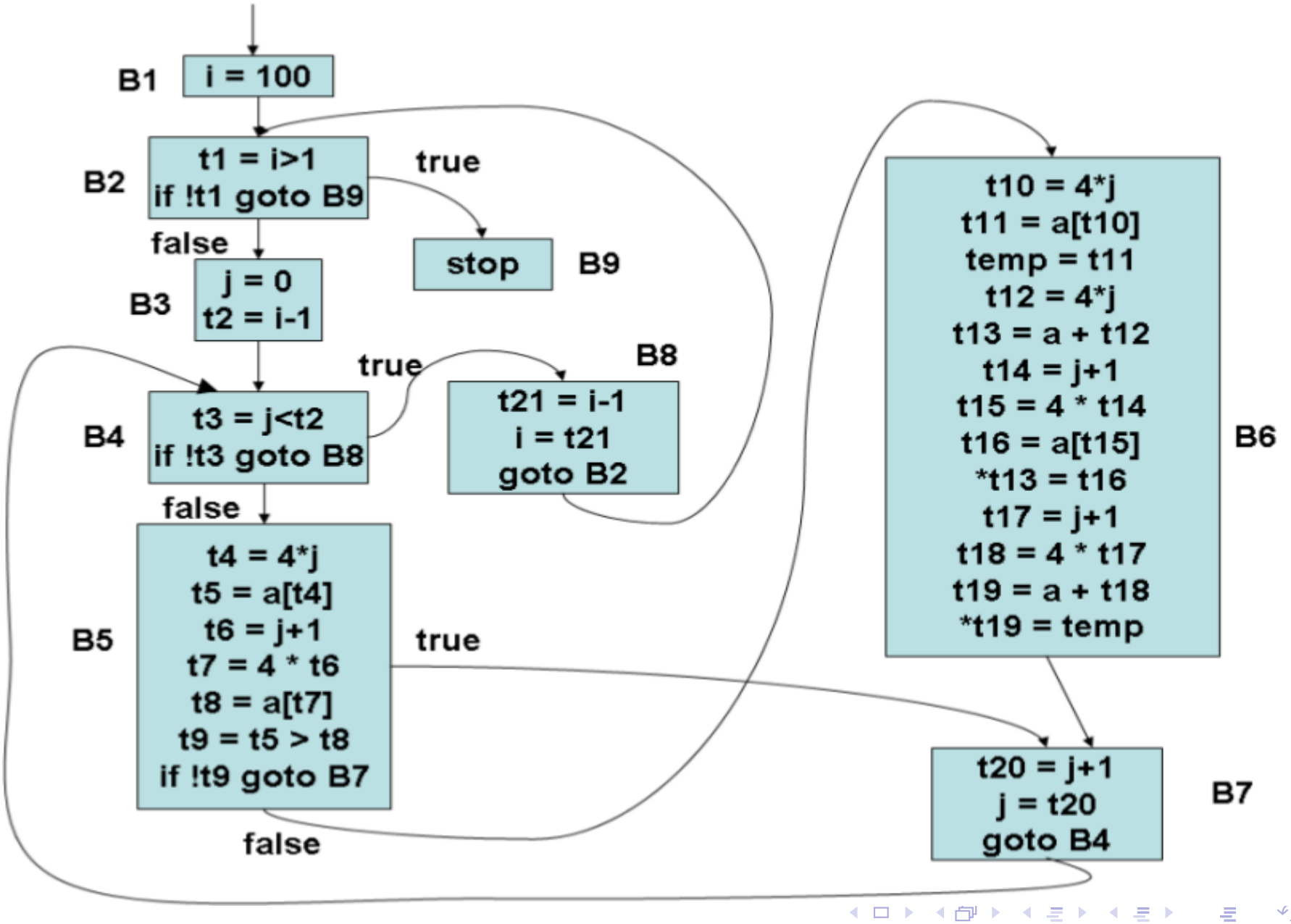
- Code optimization needs following information
  - Which definition reach a point.
  - Which expression are recomputed.
  - Which copies and constants can be propogated, and many more.
  - *A defintion kills value of the defined variable.*
- All the above information computed using Data Flow Analysis

Bubble Sort Example

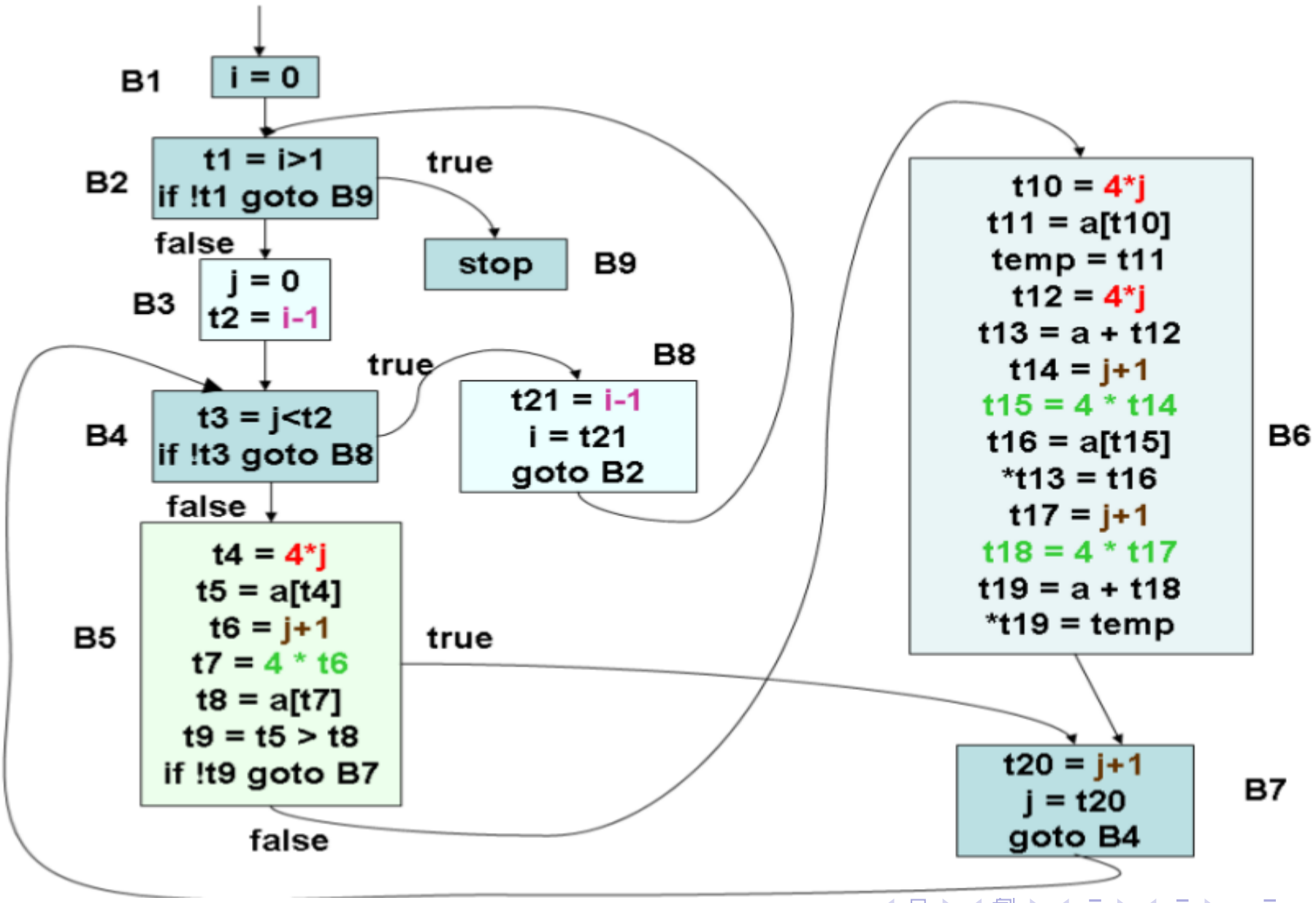
```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

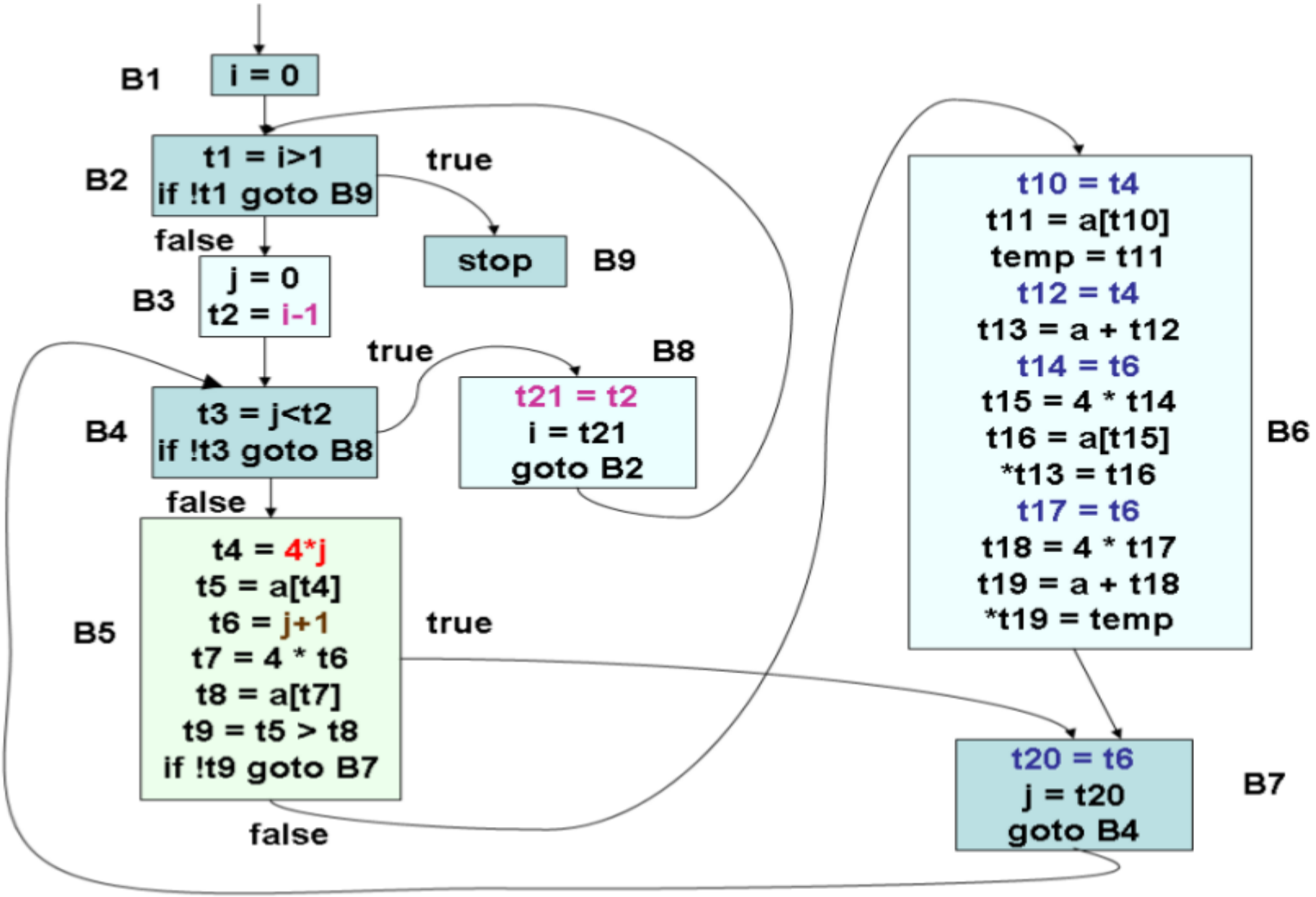
Control Flow Graph (CFG) of Bubble Sort



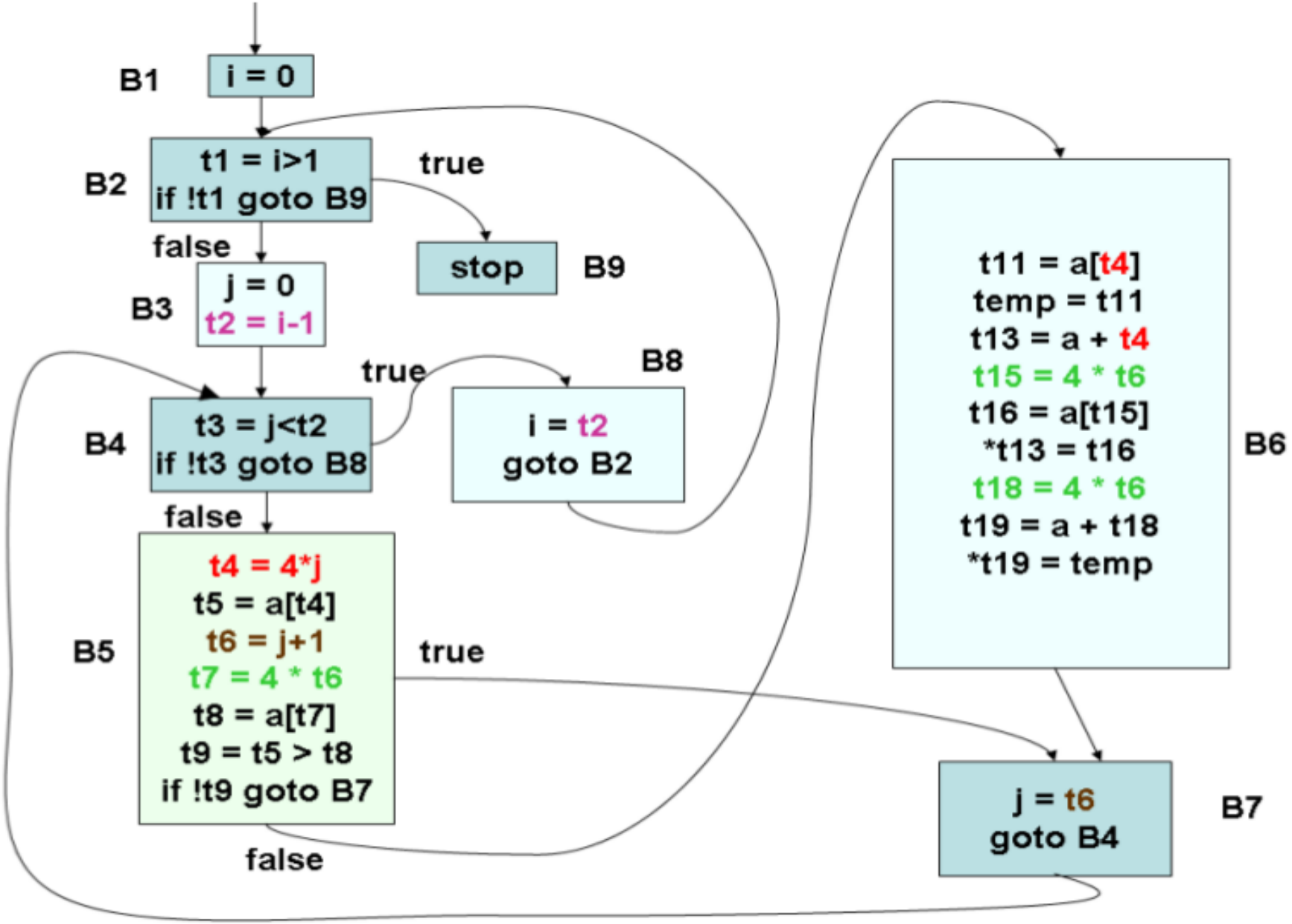
Global Common Sub Expression (GCSE)-one



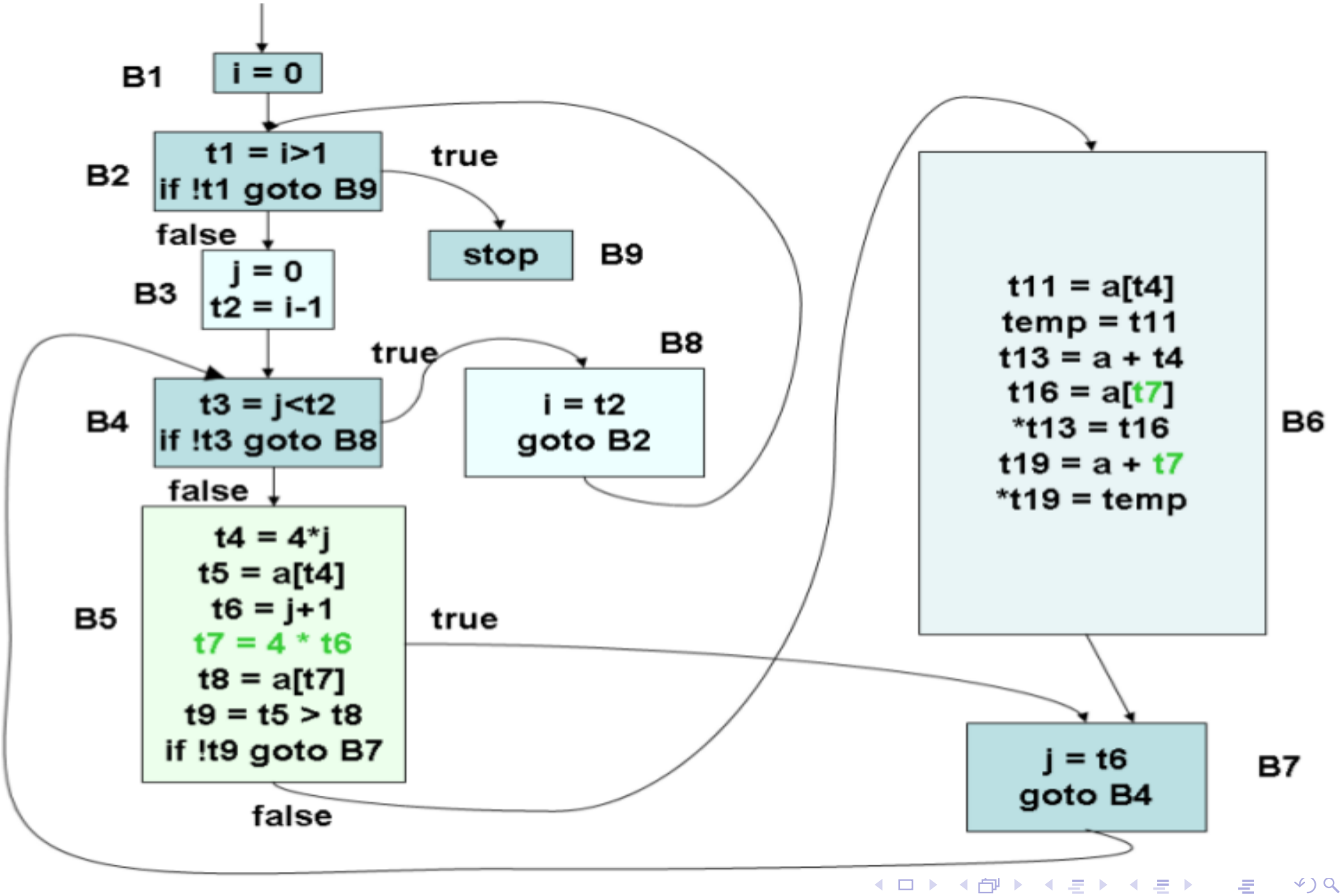
GCSE-two


B1
i = 0
B2
t1 = i>1
if !t1 goto B9
true
false
stop
B9
B3
j = 0
t2 = i-1
true
B8
t21 = t2
i = t21
goto B2
B4
t3 = j<t2
if !t3 goto B8
false
B5
t4 = 4*j
t5 = a[t4]
t6 = j+1
t7 = 4 * t6
t8 = a[t7]
t9 = t5 > t8
if !t9 goto B7
true
false
B6
t10 = t4
t11 = a[t10]
temp = t11
t12 = t4
t13 = a + t12
t14 = t6
t15 = 4 * t14
t16 = a[t15]
*t13 = t16
t17 = t6
t18 = 4 * t17
t19 = a + t18
*t19 = temp
B7
t20 = t6
j = t20
goto B4

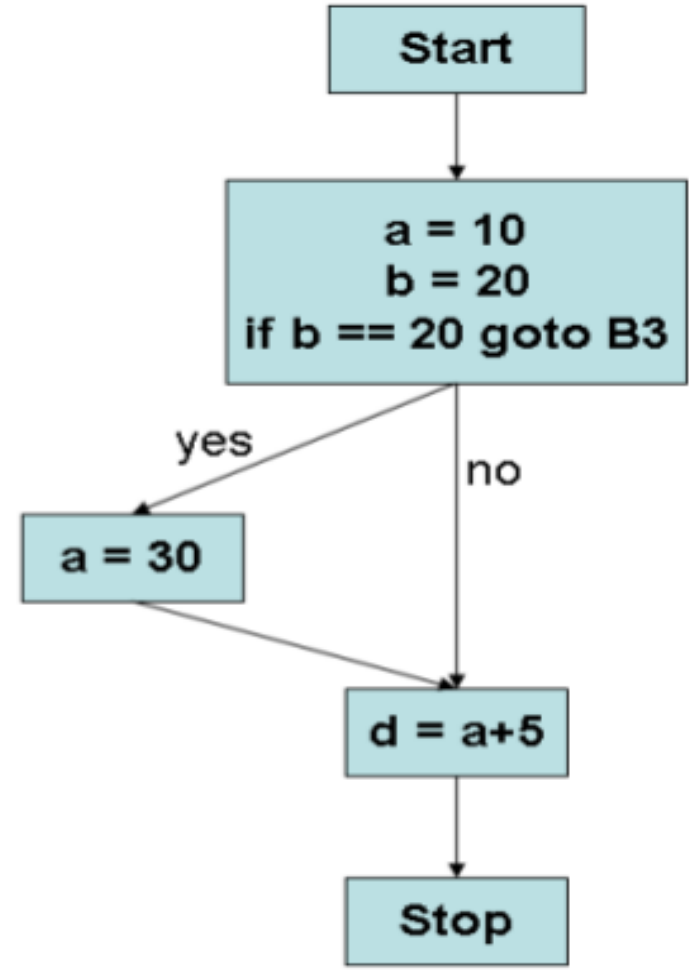
Copy Propagation



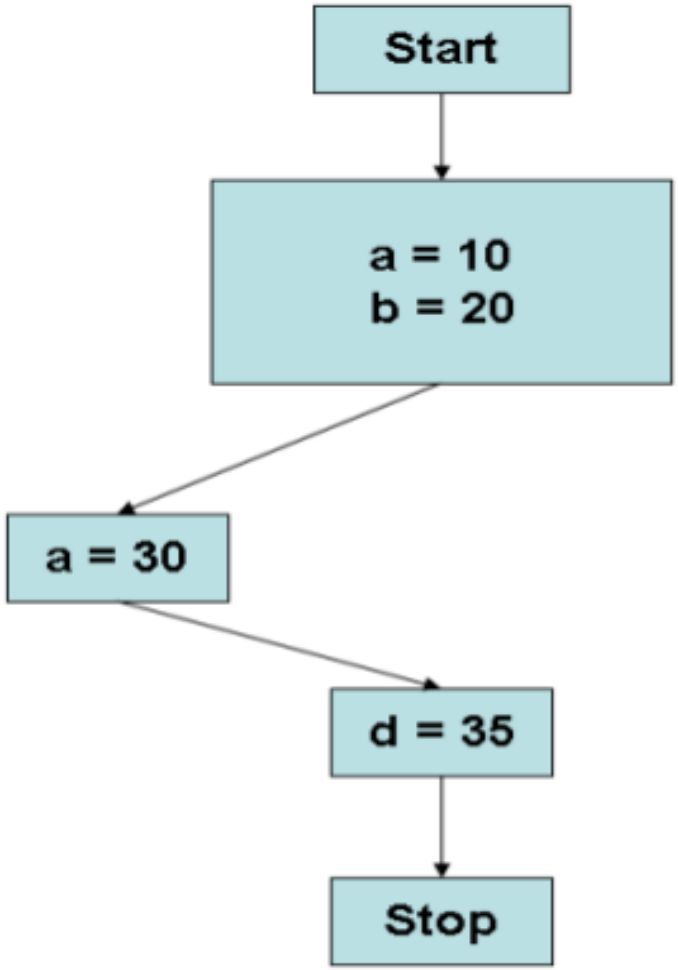
GCSE and Copy Propagation



Constant Propagation and Folding Example



Before constant propagation



After constant propagation and folding

Loop Invariant Code Motion

```
      t1 = 202
      i = 1
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t3 = addr(a)
      t4 = t3 - 4
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before LIV code motion

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

After LIV code motion

Strength Reduction

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before strength reduction for t5

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

After strength reduction for t5 and copy propagation

Induction Variable Elimintation

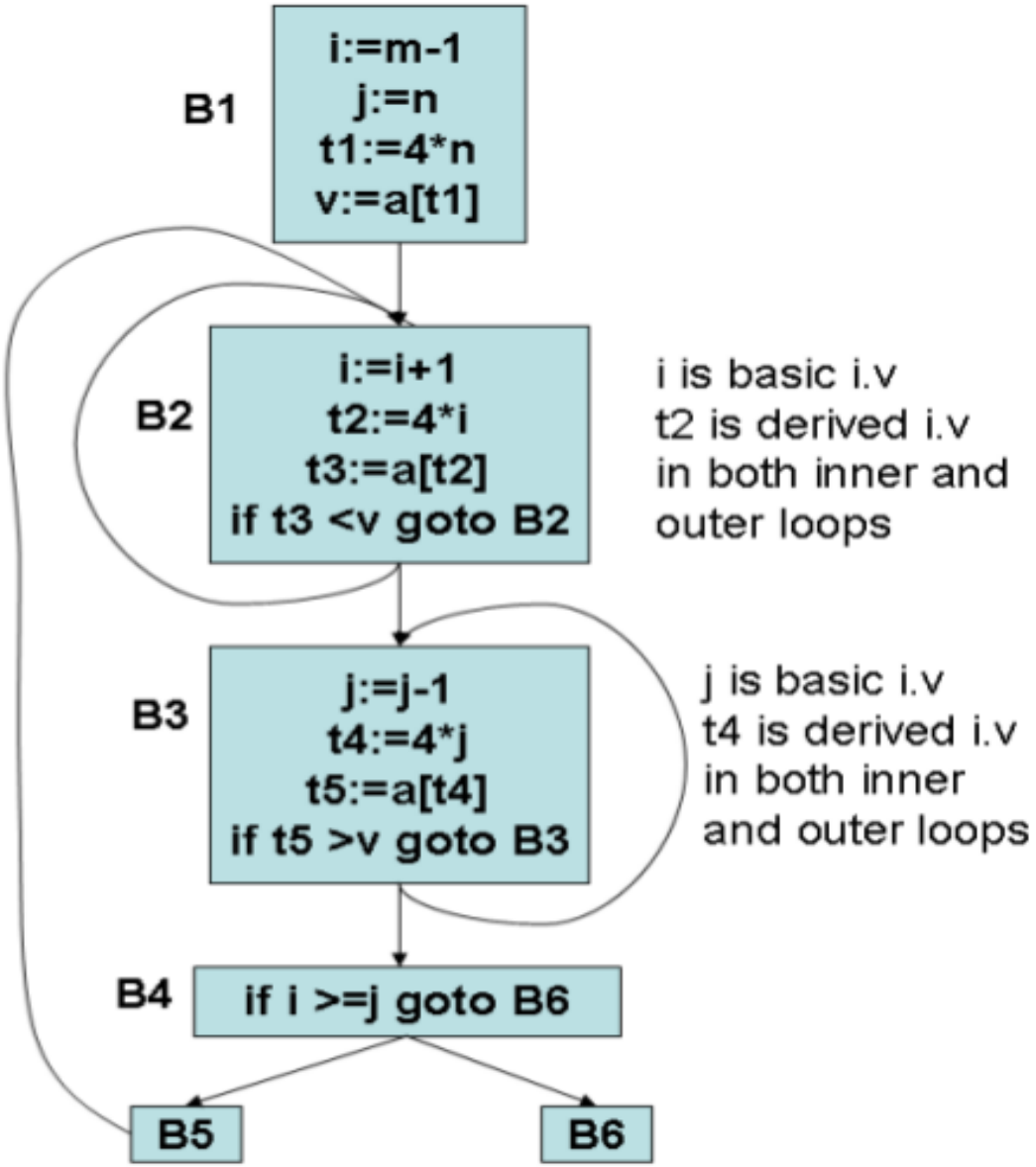
```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

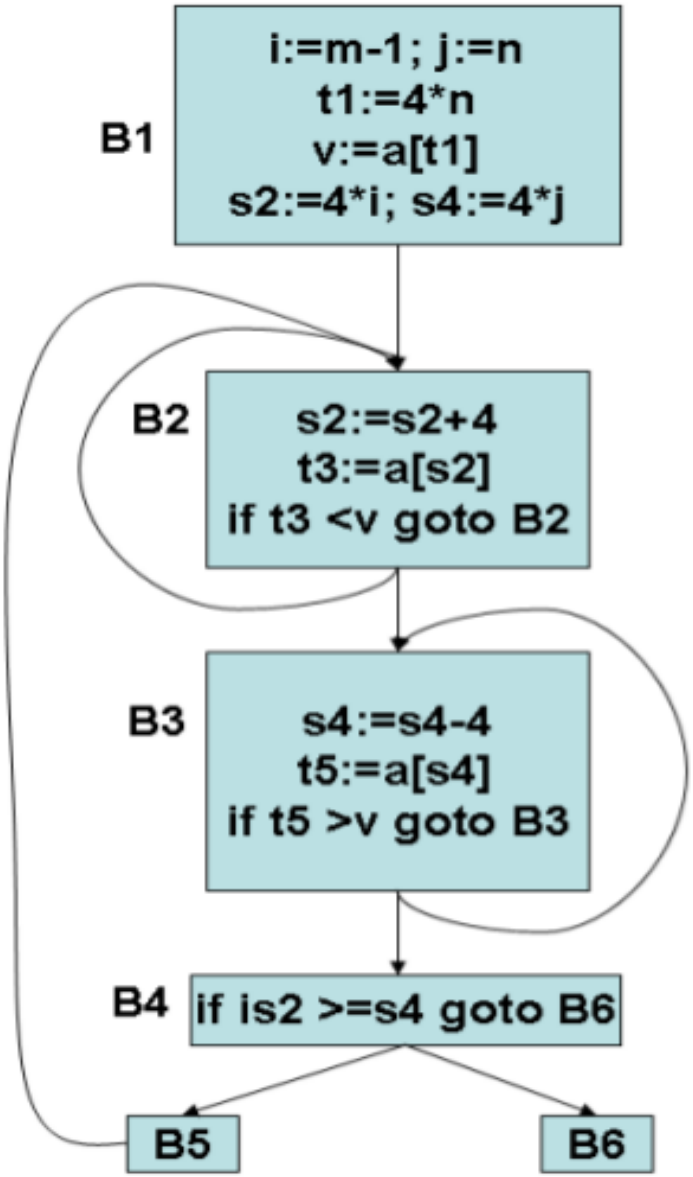
Before induction variable elimination (i)

```
      t1 = 202
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = t7 >400
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      t7 = t7 + 4
      goto L1
L2:
```

After eliminating i and replacing it with t7

Induction Variable Elimintation and Strength Reduction





- Topics: Loop Interchange, Parallelization Vectorization will be covered in coming classes.

- Topics: Loop Interchange, Parallelization Vectorization will be covered in coming classes.
- Read online on Data Flow Analysis and Abstract Interpretation (upto you).