Dead Code Elimination & Constant Propagation on SSA form

This lecture is primarily based on Konstantinos Sagonas set of slides
(**Advanced Compiler Techniques**, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.
(In turn based on Keith Cooper's slides)

Dead Code Elimination Using SSA

Dead code elimination

- Conceptually similar to mark-sweep garbage collection:
  - Mark *useful* operations.
  - Everything not marked is useless.
- Need an efficient way to find and to mark useful operations.
  - Start with critical operations.
  - Work back up SSA edges to find their antecedents.
- Operations defined as critical:
  - I/O statements,
  - linkage code (*entry & exit blocks*),
  - return values,
  - calls to other procedures.

Algorithm will use post-dominators & reverse dominance frontiers.

Dead Code Elimination Using SSA

Mark

```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (WorkList ≠ Ø)
    remove i from WorkList
        (i has form "x←y op z")
    if def(y) is not marked then
        mark def(y)
        add def(y) to WorkList
    if def(z) is not marked then
        mark def(z)
        add def(z) to WorkList

    for each b ∈ RDF(block(i))
        mark the block-ending
            branch in b
        add it to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        if i is a branch then
            rewrite with a jump to
                i's nearest useful
                post-dominator
        if i is not a jump then
            delete i
```

Notes:

- Eliminates some branches.
- Reconnects dead branches to the remaining live code.
- Find useful post-dominator by walking post-dominator tree.
  - > Entry & exit nodes are useful

Dead Code Elimination Using SSA

Handling Branches

- When is a branch useful?
  - When another useful operation depends on its existence

In the CFG, *j* is control dependent on *i* if

1. ∃ a non-null path *p* from *i* to *j* such that *j* post-dominates every node on *p* after *i*

2. *j* does not strictly post-dominate *i*

- *j* control dependent on *i* ⇒ one path from *i* leads to *j*, one doesn't
- This is the reverse dominance frontier of *j* (RDF(j))

Algorithm uses RDF(*n*) to mark branches as live

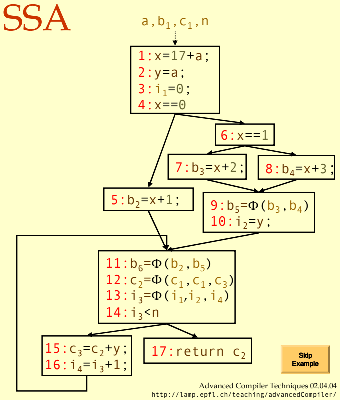
Dead Code Elimination Using SSA

Mark

```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (WorkList ≠ Ø)
    remove i from WorkList
        (i has form "x←y op z")
    if def(y) is not marked then
        mark def(y)
        add def(y) to WorkList
    if def(z) is not marked then
        mark def(z)
        add def(z) to WorkList

    for each b ∈ RDF(block(i))
        mark the block-ending
            branch in b
        add it to WorkList
```



Skip Example

Dead Code Elimination Using SSA

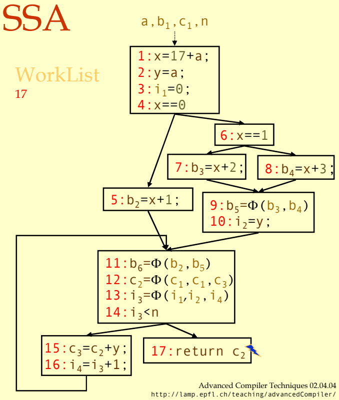
Mark

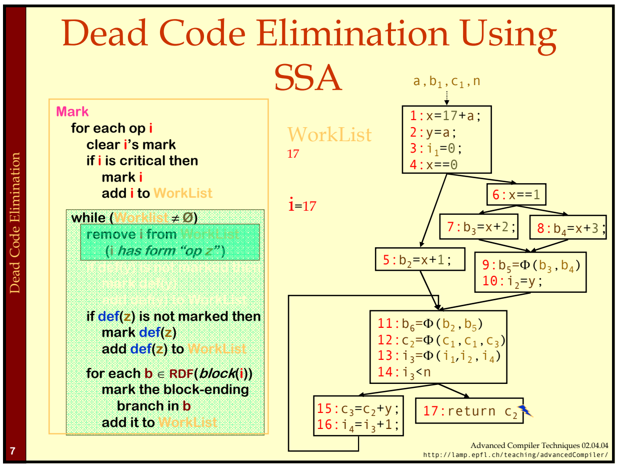
```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (WorkList ≠ Ø)
    remove i from WorkList
        (i has form "x←y op z")
    if def(y) is not marked then
        mark def(y)
        add def(y) to WorkList
    if def(z) is not marked then
        mark def(z)
        add def(z) to WorkList

    for each b ∈ RDF(block(i))
        mark the block-ending
            branch in b
        add it to WorkList
```

WorkList
17





Dead Code Elimination Using SSA

```
Mark
    for each op i
        clear i's mark
        if i is critical then
            mark i
            add i to WorkList
    while (Worklist ≠ Ø)
        remove i from WorkList
            (i has form "op z")
        if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
        if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
        for each b ∈ RDF(block(i))
            mark the block-ending
                branch in b
            add it to WorkList
```

WorkList
17

i=17

a, b_1, c_1, n

1: $x=17+a$;
2: $y=a$;
3: $i_1=0$;
4: $x==0$

5: $b_2=x+1$;

6: $x==1$

7: $b_3=x+2$;

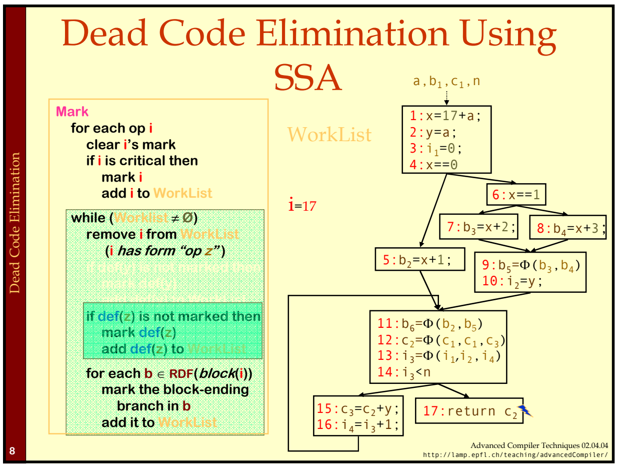
8: $b_4=x+3$;

9: $b_5=\Phi(b_3, b_4)$
10: $i_2=y$;

11: $b_6=\Phi(b_2, b_5)$
12: $c_2=\Phi(c_1, c_1, c_3)$
13: $i_3=\Phi(i_1, i_2, i_4)$
14: $i_3<n$

15: $c_3=c_2+y$;
16: $i_4=i_3+1$;

17: return c_2



Dead Code Elimination Using SSA

```
Mark
    for each op i
        clear i's mark
        if i is critical then
            mark i
            add i to WorkList
    while (Worklist ≠ Ø)
        remove i from WorkList
            (i has form "op z")
        if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
        if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
        for each b ∈ RDF(block(i))
            mark the block-ending
                branch in b
            add it to WorkList
```

WorkList

i=17

a, b_1, c_1, n

1: $x=17+a$;
2: $y=a$;
3: $i_1=0$;
4: $x==0$

5: $b_2=x+1$;

6: $x==1$

7: $b_3=x+2$;

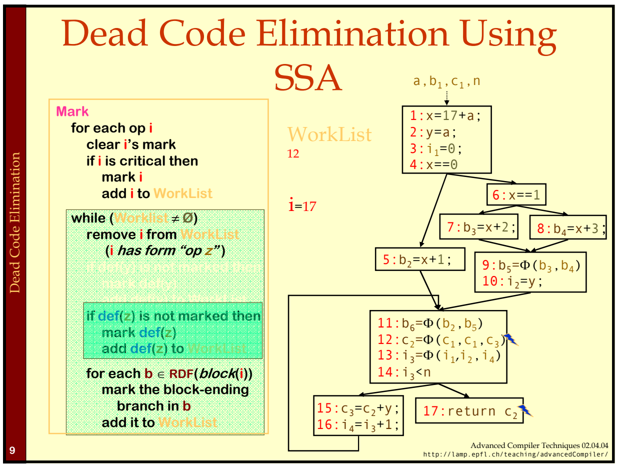
8: $b_4=x+3$;

9: $b_5=\Phi(b_3, b_4)$
10: $i_2=y$;

11: $b_6=\Phi(b_2, b_5)$
12: $c_2=\Phi(c_1, c_1, c_3)$
13: $i_3=\Phi(i_1, i_2, i_4)$
14: $i_3<n$

15: $c_3=c_2+y$;
16: $i_4=i_3+1$;

17: return c_2



Dead Code Elimination Using SSA

```
Mark
    for each op i
        clear i's mark
        if i is critical then
            mark i
            add i to WorkList
    while (Worklist ≠ Ø)
        remove i from WorkList
            (i has form "op z")
        if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
        if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
        for each b ∈ RDF(block(i))
            mark the block-ending
                branch in b
            add it to WorkList
```

WorkList
12

i=17

a, b_1, c_1, n

1: $x=17+a$;
2: $y=a$;
3: $i_1=0$;
4: $x==0$

5: $b_2=x+1$;

6: $x==1$

7: $b_3=x+2$;

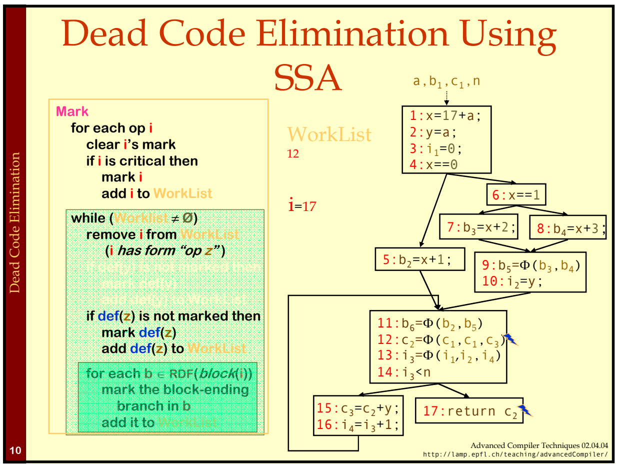
8: $b_4=x+3$;

9: $b_5=\Phi(b_3, b_4)$
10: $i_2=y$;

11: $b_6=\Phi(b_2, b_5)$
12: $c_2=\Phi(c_1, c_1, c_3)$
13: $i_3=\Phi(i_1, i_2, i_4)$
14: $i_3<n$

15: $c_3=c_2+y$;
16: $i_4=i_3+1$;

17: return c_2



Dead Code Elimination Using SSA

```
Mark
    for each op i
        clear i's mark
        if i is critical then
            mark i
            add i to WorkList
    while (Worklist ≠ Ø)
        remove i from WorkList
            (i has form "op z")
        if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
        if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
        for each b ∈ RDF(block(i))
            mark the block-ending
                branch in b
            add it to WorkList
```

WorkList
12

i=17

a, b_1, c_1, n

1: $x=17+a$;
2: $y=a$;
3: $i_1=0$;
4: $x==0$

5: $b_2=x+1$;

6: $x==1$

7: $b_3=x+2$;

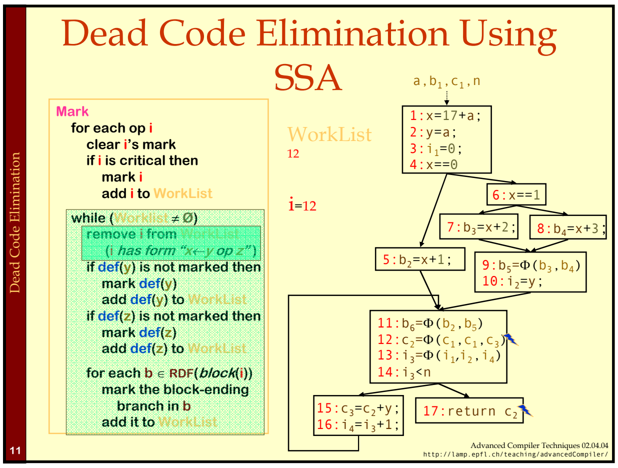
8: $b_4=x+3$;

9: $b_5=\Phi(b_3, b_4)$
10: $i_2=y$;

11: $b_6=\Phi(b_2, b_5)$
12: $c_2=\Phi(c_1, c_1, c_3)$
13: $i_3=\Phi(i_1, i_2, i_4)$
14: $i_3<n$

15: $c_3=c_2+y$;
16: $i_4=i_3+1$;

17: return c_2



Dead Code Elimination Using SSA

```
Mark
    for each op i
        clear i's mark
        if i is critical then
            mark i
            add i to WorkList
    while (Worklist ≠ Ø)
        remove i from WorkList
            (i has form "x←y op z")
        if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
        if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
        for each b ∈ RDF(block(i))
            mark the block-ending
                branch in b
            add it to WorkList
```

WorkList
12

i=12

a, b_1, c_1, n

1: $x=17+a$;
2: $y=a$;
3: $i_1=0$;
4: $x==0$

5: $b_2=x+1$;

6: $x==1$

7: $b_3=x+2$;

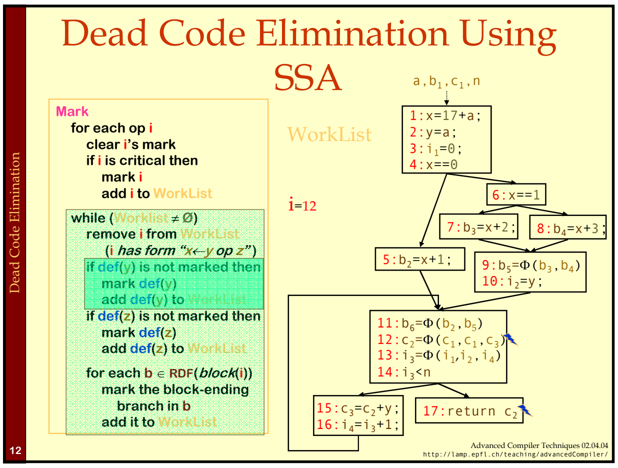
8: $b_4=x+3$;

9: $b_5=\Phi(b_3, b_4)$
10: $i_2=y$;

11: $b_6=\Phi(b_2, b_5)$
12: $c_2=\Phi(c_1, c_1, c_3)$
13: $i_3=\Phi(i_1, i_2, i_4)$
14: $i_3<n$

15: $c_3=c_2+y$;
16: $i_4=i_3+1$;

17: return c_2



Dead Code Elimination Using SSA

```
Mark
    for each op i
        clear i's mark
        if i is critical then
            mark i
            add i to WorkList
    while (Worklist ≠ Ø)
        remove i from WorkList
            (i has form "x←y op z")
        if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
        if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
        for each b ∈ RDF(block(i))
            mark the block-ending
                branch in b
            add it to WorkList
```

WorkList

i=12

a, b_1, c_1, n

1: $x=17+a$;
2: $y=a$;
3: $i_1=0$;
4: $x==0$

5: $b_2=x+1$;

6: $x==1$

7: $b_3=x+2$;

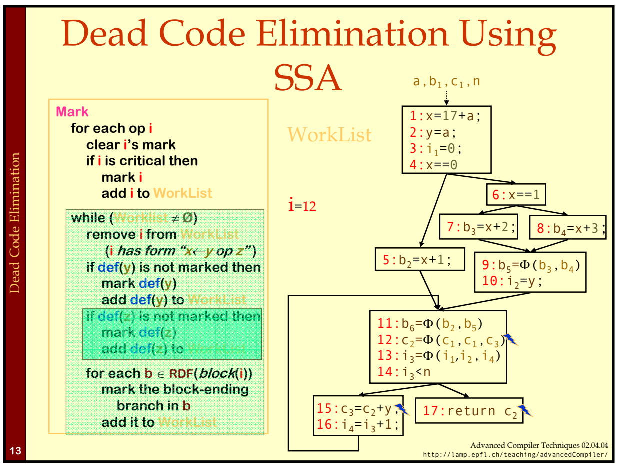
8: $b_4=x+3$;

9: $b_5=\Phi(b_3, b_4)$
10: $i_2=y$;

11: $b_6=\Phi(b_2, b_5)$
12: $c_2=\Phi(c_1, c_1, c_3)$
13: $i_3=\Phi(i_1, i_2, i_4)$
14: $i_3<n$

15: $c_3=c_2+y$;
16: $i_4=i_3+1$;

17: return c_2

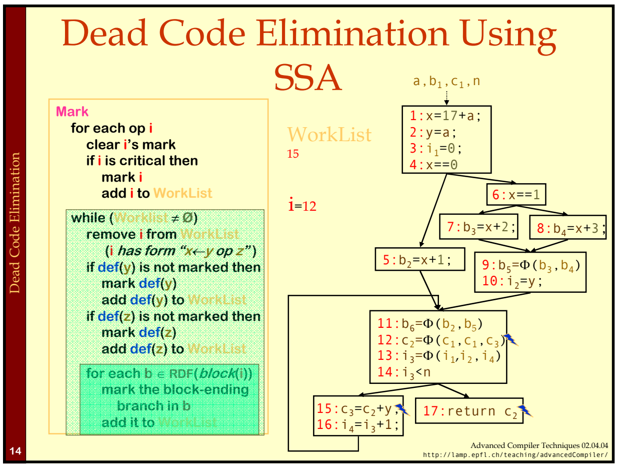


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

i=12



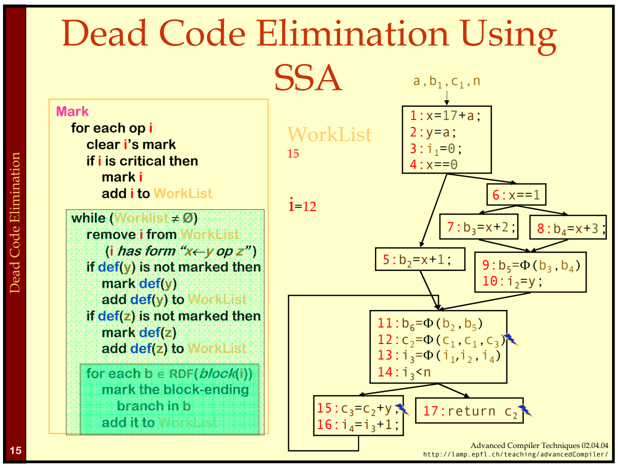
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

15

i=12



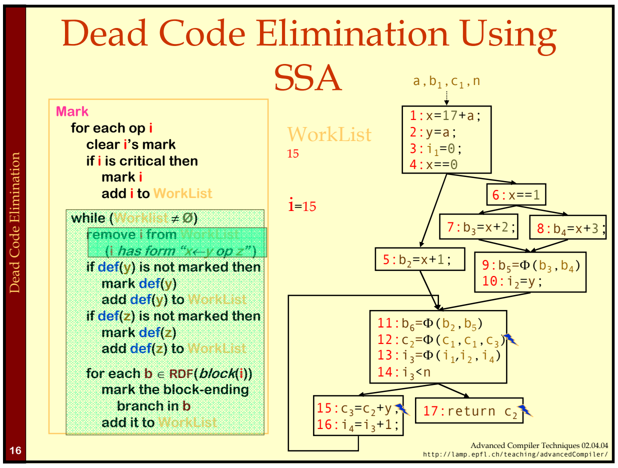
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

15

i=12



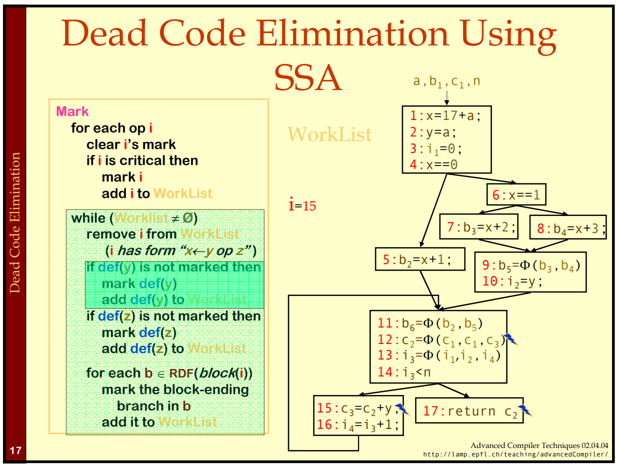
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

15

i=15

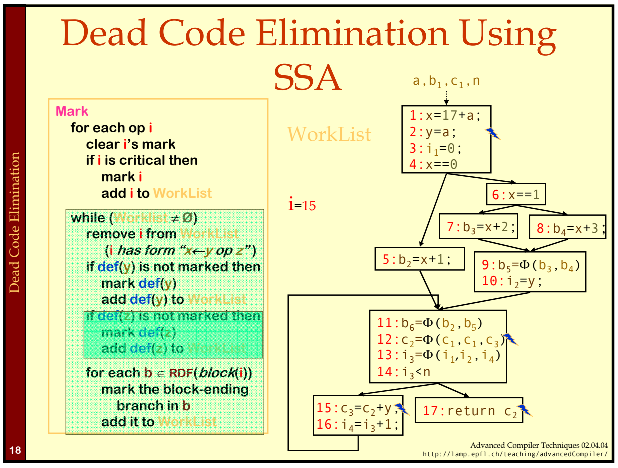


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

i=15

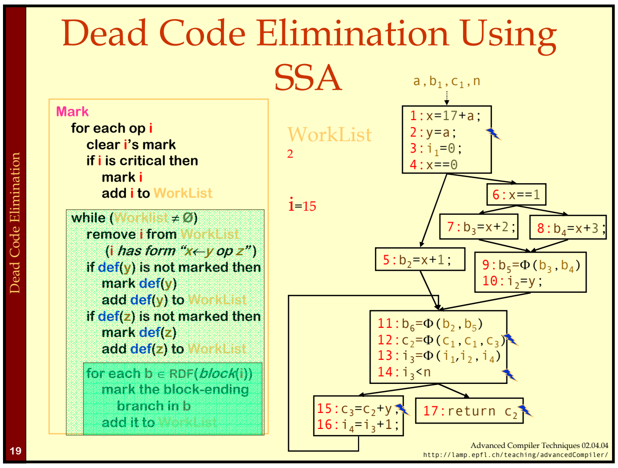


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

i=15



Dead Code Elimination Using SSA

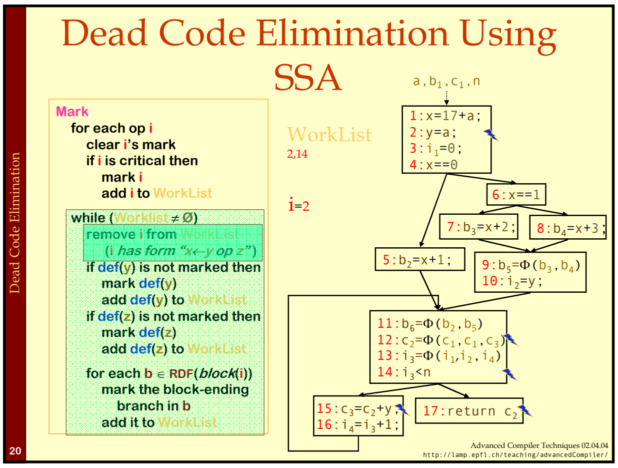
```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
      branch in b
      add it to WorkList
```

WorkList
2

i=15

Dead Code Elimination 19

Advanced Compiler Techniques 02.04.04
http://lamp.epfl.ch/teaching/advancedCompiler/



Dead Code Elimination Using SSA

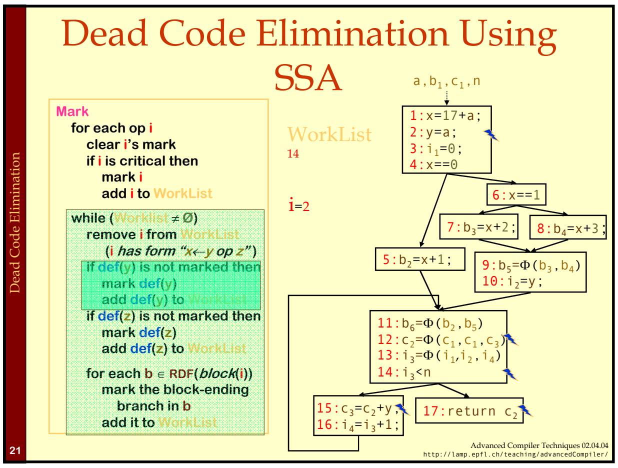
```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
      branch in b
      add it to WorkList
```

WorkList
2,14

i=2

Dead Code Elimination 20

Advanced Compiler Techniques 02.04.04
http://lamp.epfl.ch/teaching/advancedCompiler/



Dead Code Elimination Using SSA

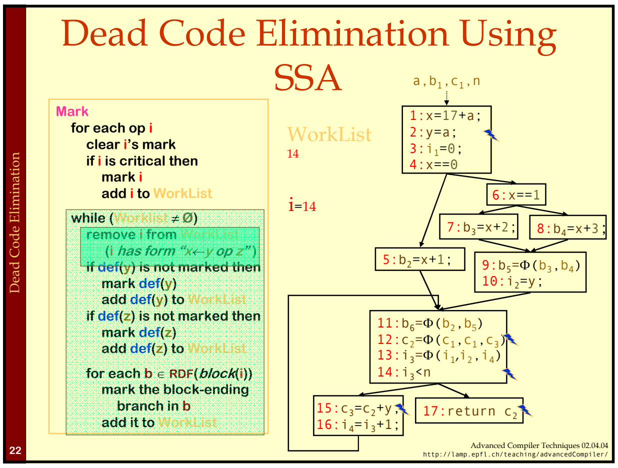
```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
      branch in b
      add it to WorkList
```

WorkList
14

i=2

Dead Code Elimination 21

Advanced Compiler Techniques 02.04.04
http://lamp.epfl.ch/teaching/advancedCompiler/



Dead Code Elimination Using SSA

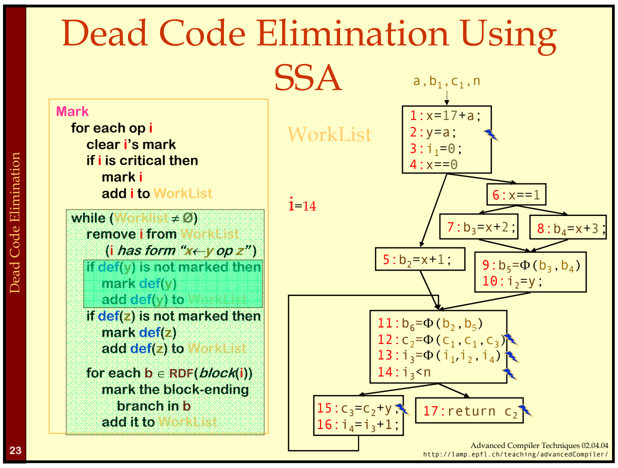
```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
      branch in b
      add it to WorkList
```

WorkList
14

i=14

Dead Code Elimination 22

Advanced Compiler Techniques 02.04.04
http://lamp.epfl.ch/teaching/advancedCompiler/



Dead Code Elimination Using SSA

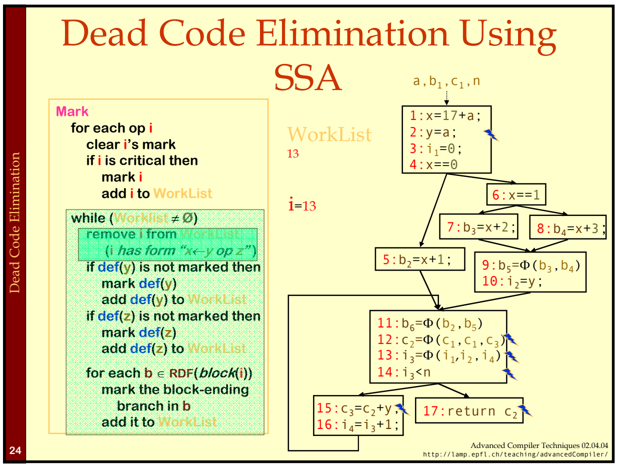
```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
      branch in b
      add it to WorkList
```

WorkList

i=14

Dead Code Elimination 23

Advanced Compiler Techniques 02.04.04
http://lamp.epfl.ch/teaching/advancedCompiler/



Dead Code Elimination Using SSA

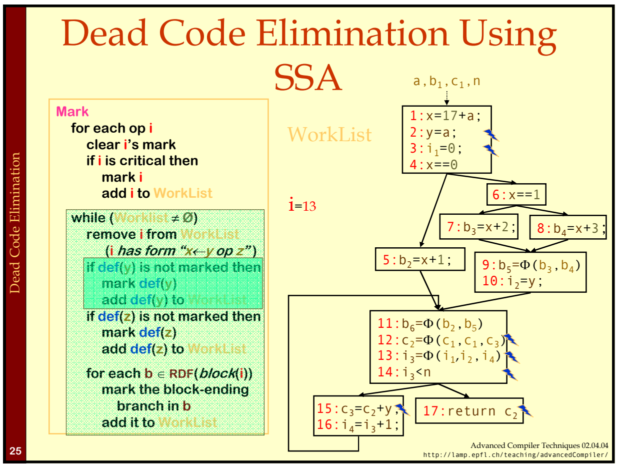
```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
      branch in b
      add it to WorkList
```

WorkList
13

i=13

Dead Code Elimination 24

Advanced Compiler Techniques 02.04.04
http://lamp.epfl.ch/teaching/advancedCompiler/

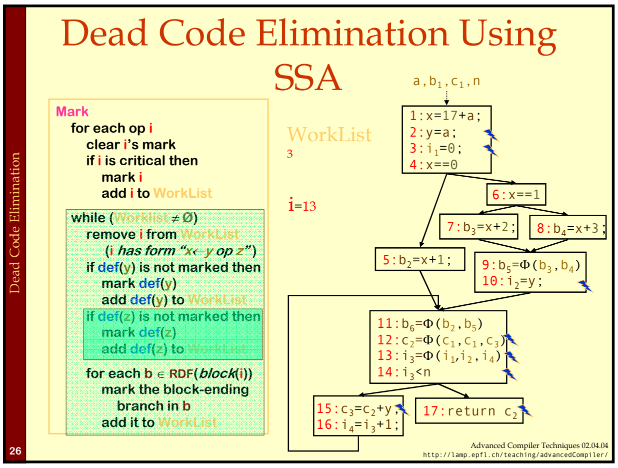


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

i=13



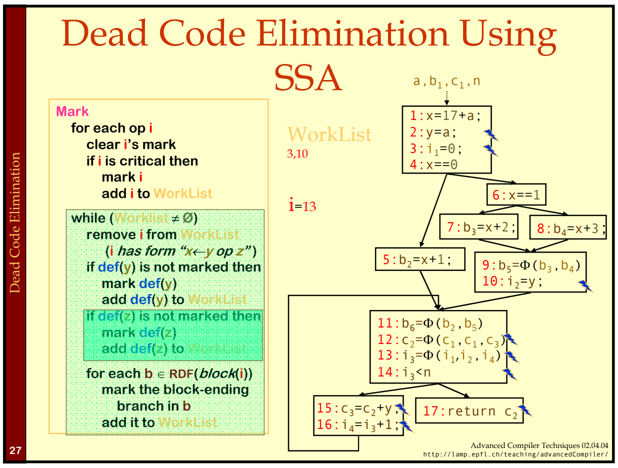
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

3

i=13



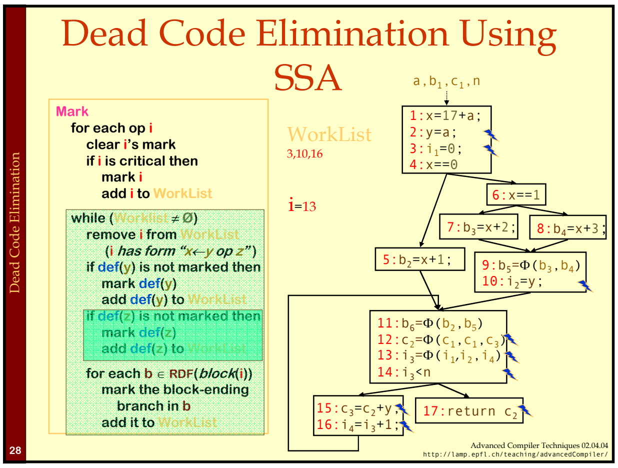
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

3,10

i=13



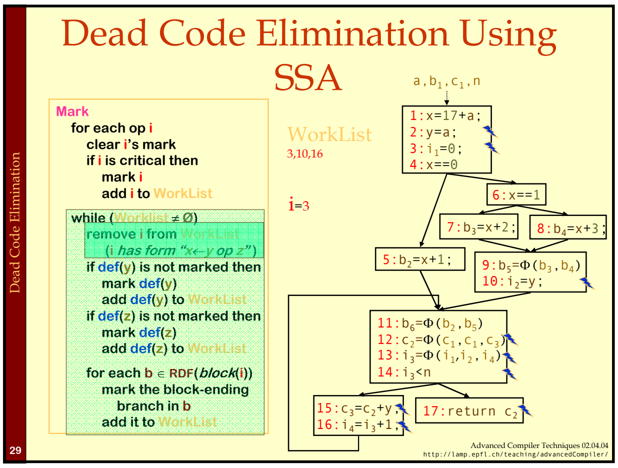
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

3,10,16

i=13



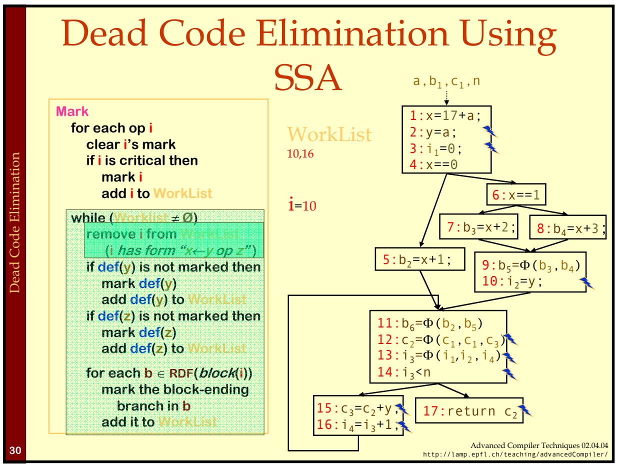
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

3,10,16

i=3



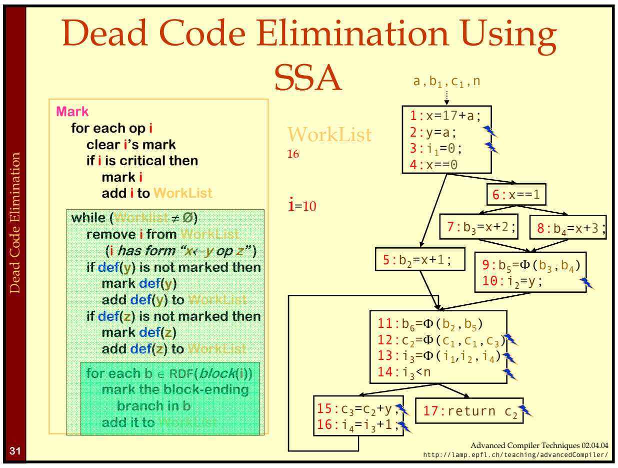
Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

10,16

i=10

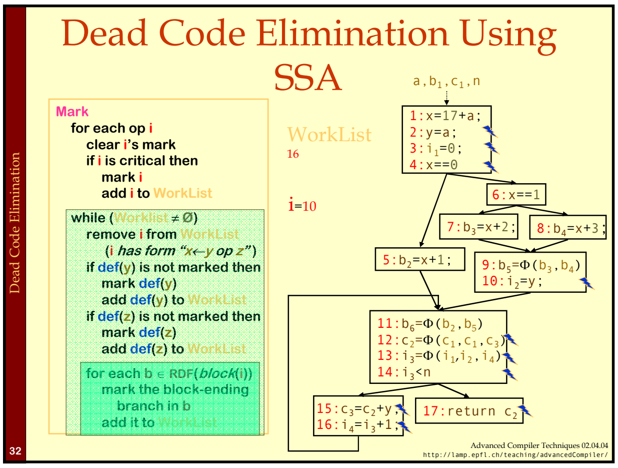


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form “x←y op z”)
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList
16

i=10

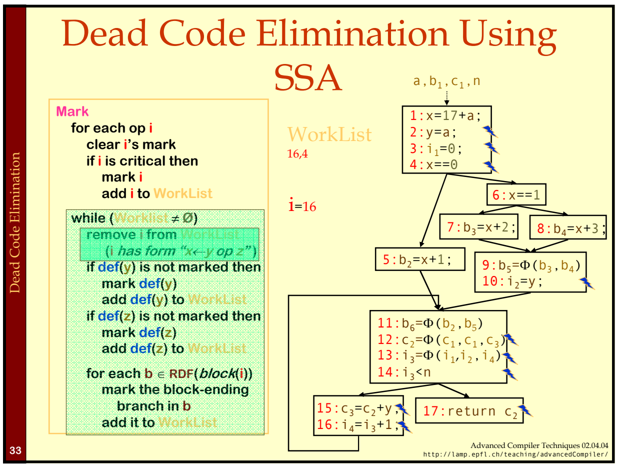


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form “x←y op z”)
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList
16

i=10

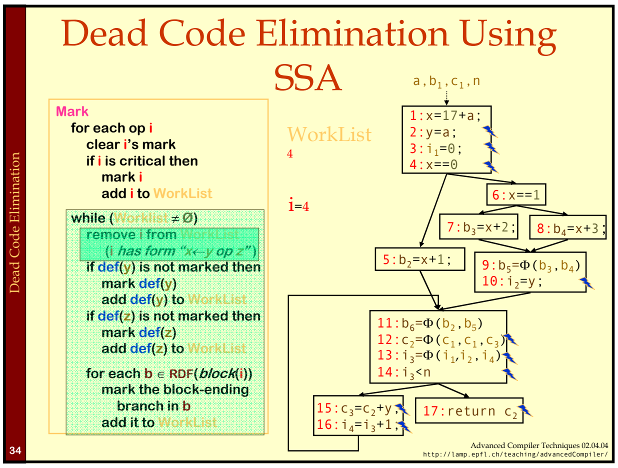


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form “x←y op z”)
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList
16,4

i=16

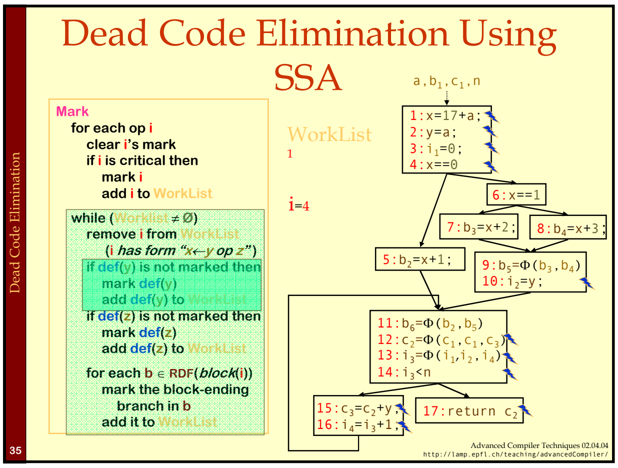


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form “x←y op z”)
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList
4

i=4

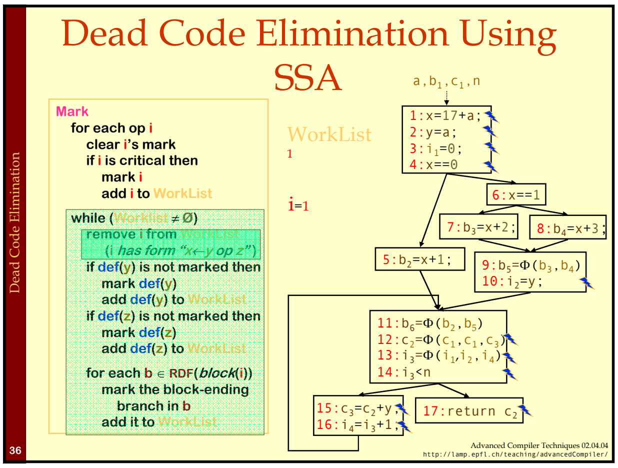


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form “x←y op z”)
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList
1

i=4

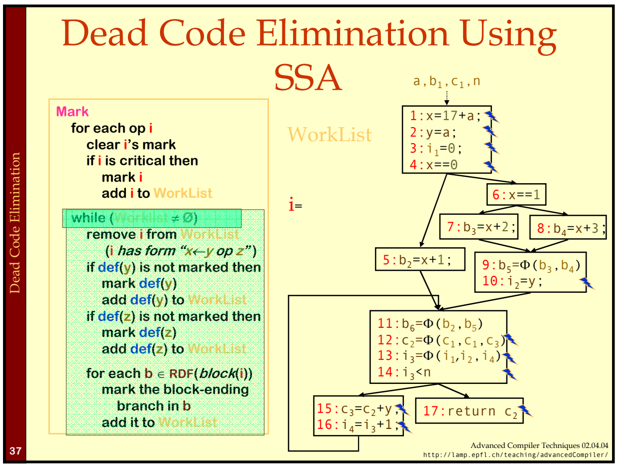


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form “x←y op z”)
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList
1

i=1

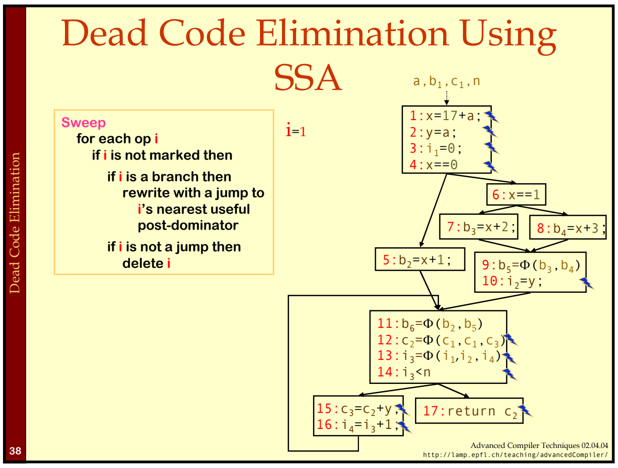


Dead Code Elimination Using SSA

```
Mark
  for each op i
    clear i's mark
    if i is critical then
      mark i
      add i to WorkList
  while (Worklist ≠ Ø)
    remove i from WorkList
      (i has form "x←y op z")
    if def(y) is not marked then
      mark def(y)
      add def(y) to WorkList
    if def(z) is not marked then
      mark def(z)
      add def(z) to WorkList
    for each b ∈ RDF(block(i))
      mark the block-ending
        branch in b
      add it to WorkList
```

WorkList

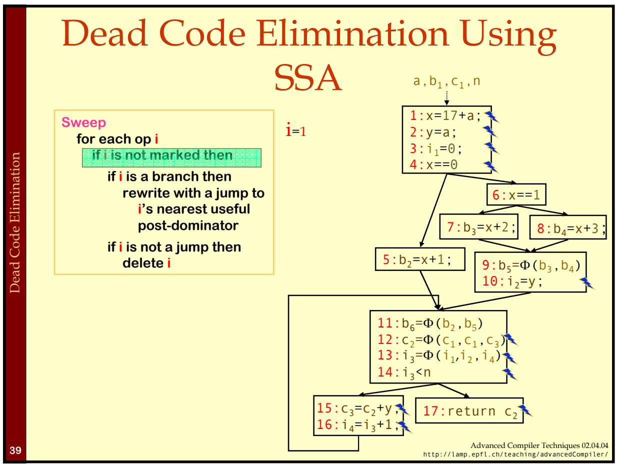
i=



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

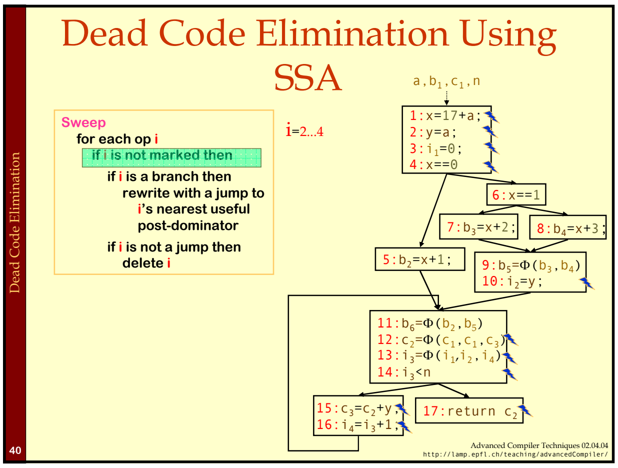
i=1



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

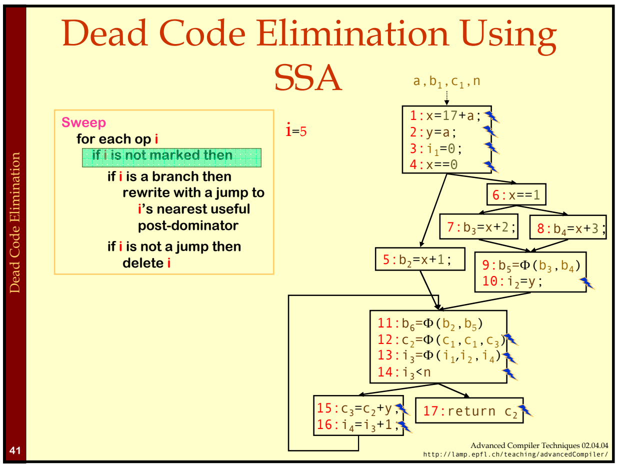
i=1



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

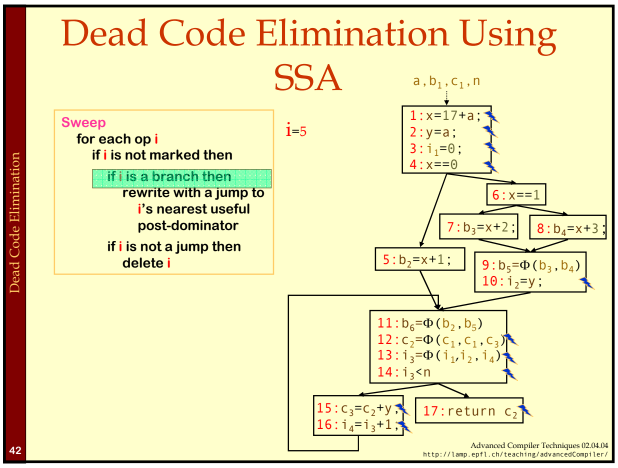
i=2...4



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

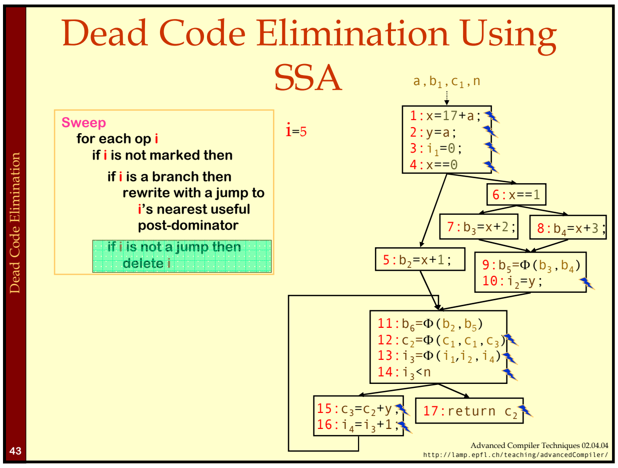
i=5



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=5



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=5

a,b_1,c_1,n

1:x=17+a;
2:y=a;
3:i_1=0;
4:x==0

6:x==1

7:b_3=x+2;
8:b_4=x+3;

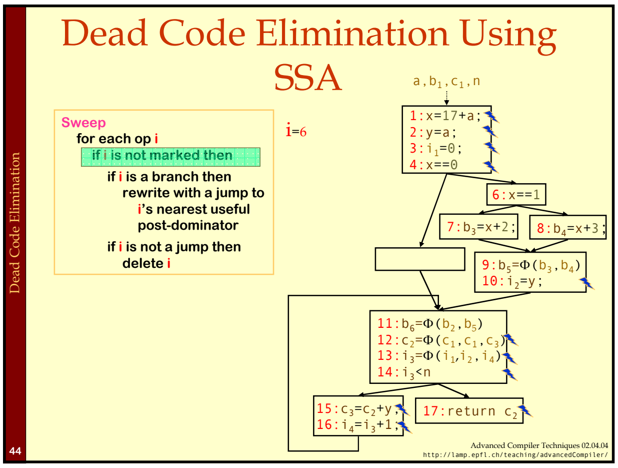
5:b_2=x+1;

9:b_5=Φ(b_3,b_4)
10:i_2=y;

11:b_6=Φ(b_2,b_5)
12:c_2=Φ(c_1,c_1,c_3)
13:i_3=Φ(i_1,i_2,i_4)
14:i_3<n

15:c_3=c_2+y;
16:i_4=i_3+1;

17:return c_2



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=6

a,b_1,c_1,n

1:x=17+a;
2:y=a;
3:i_1=0;
4:x==0

6:x==1

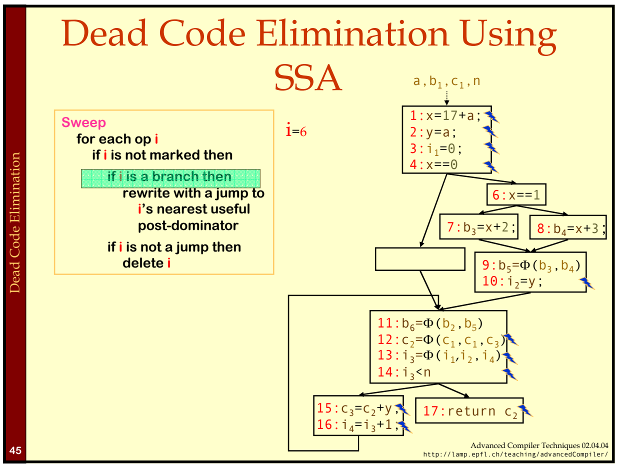
7:b_3=x+2;
8:b_4=x+3;

9:b_5=Φ(b_3,b_4)
10:i_2=y;

11:b_6=Φ(b_2,b_5)
12:c_2=Φ(c_1,c_1,c_3)
13:i_3=Φ(i_1,i_2,i_4)
14:i_3<n

15:c_3=c_2+y;
16:i_4=i_3+1;

17:return c_2



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=6

a,b_1,c_1,n

1:x=17+a;
2:y=a;
3:i_1=0;
4:x==0

6:x==1

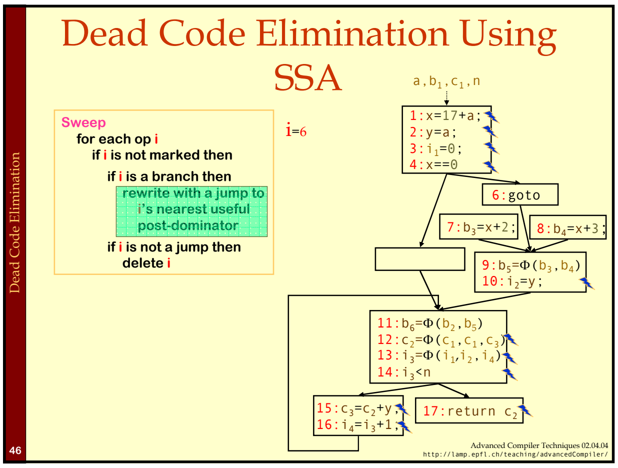
7:b_3=x+2;
8:b_4=x+3;

9:b_5=Φ(b_3,b_4)
10:i_2=y;

11:b_6=Φ(b_2,b_5)
12:c_2=Φ(c_1,c_1,c_3)
13:i_3=Φ(i_1,i_2,i_4)
14:i_3<n

15:c_3=c_2+y;
16:i_4=i_3+1;

17:return c_2



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=6

a,b_1,c_1,n

1:x=17+a;
2:y=a;
3:i_1=0;
4:x==0

6:goto

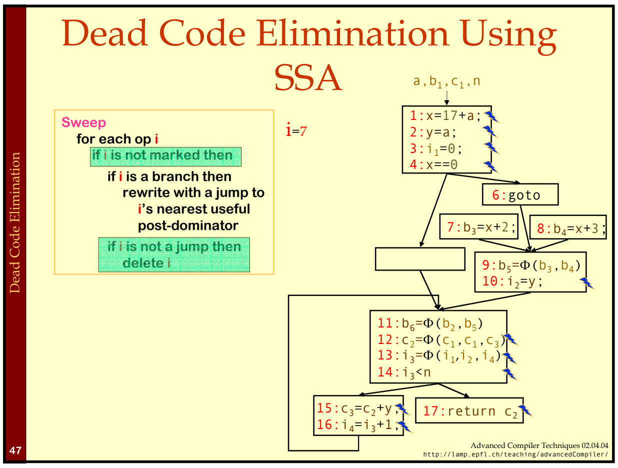
7:b_3=x+2;
8:b_4=x+3;

9:b_5=Φ(b_3,b_4)
10:i_2=y;

11:b_6=Φ(b_2,b_5)
12:c_2=Φ(c_1,c_1,c_3)
13:i_3=Φ(i_1,i_2,i_4)
14:i_3<n

15:c_3=c_2+y;
16:i_4=i_3+1;

17:return c_2



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=7

a,b_1,c_1,n

1:x=17+a;
2:y=a;
3:i_1=0;
4:x==0

6:goto

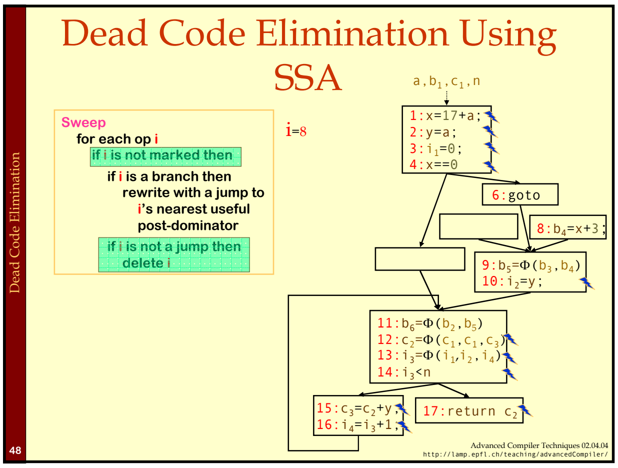
7:b_3=x+2;
8:b_4=x+3;

9:b_5=Φ(b_3,b_4)
10:i_2=y;

11:b_6=Φ(b_2,b_5)
12:c_2=Φ(c_1,c_1,c_3)
13:i_3=Φ(i_1,i_2,i_4)
14:i_3<n

15:c_3=c_2+y;
16:i_4=i_3+1;

17:return c_2



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=8

a,b_1,c_1,n

1:x=17+a;
2:y=a;
3:i_1=0;
4:x==0

6:goto

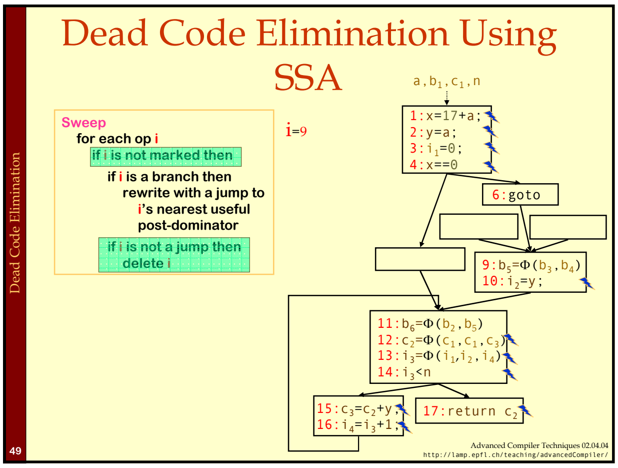
8:b_4=x+3;

9:b_5=Φ(b_3,b_4)
10:i_2=y;

11:b_6=Φ(b_2,b_5)
12:c_2=Φ(c_1,c_1,c_3)
13:i_3=Φ(i_1,i_2,i_4)
14:i_3<n

15:c_3=c_2+y;
16:i_4=i_3+1;

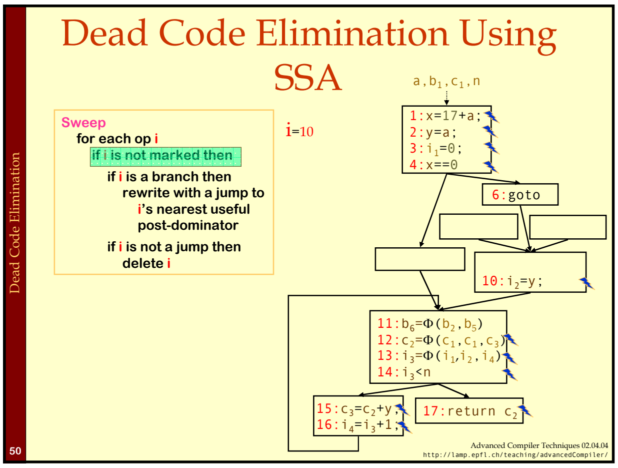
17:return c_2



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

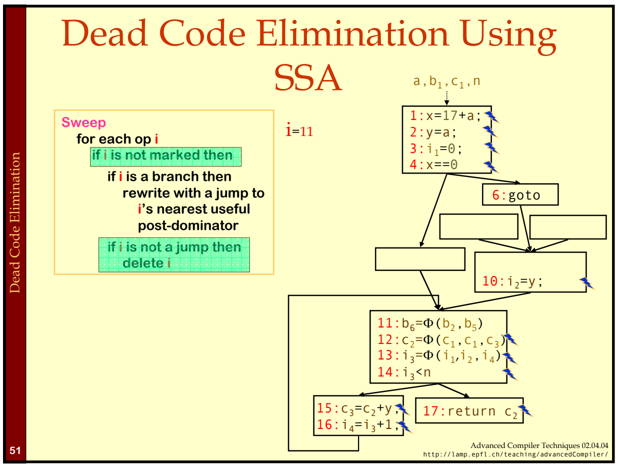
i=9



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

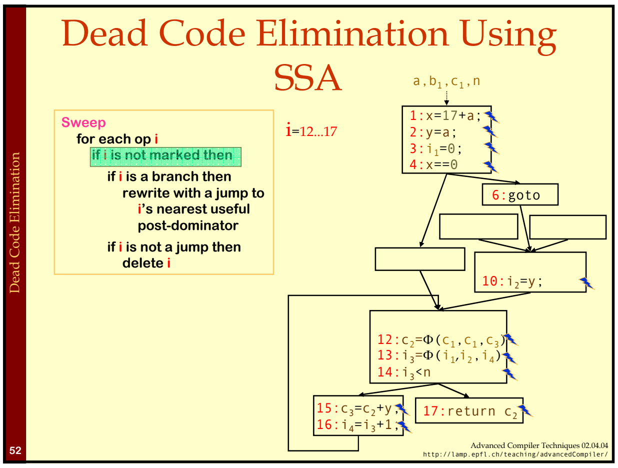
i=10



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

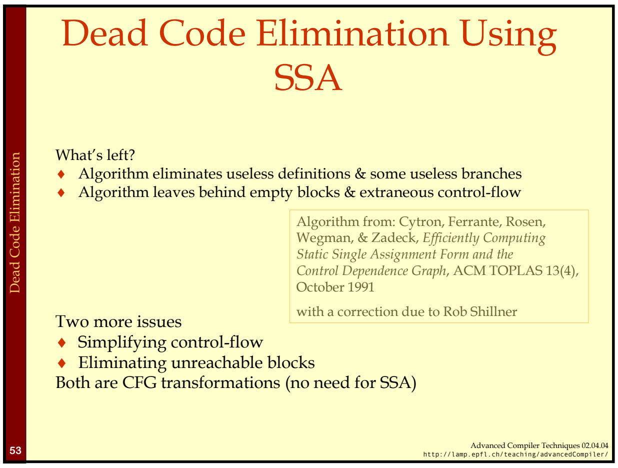
i=11



Dead Code Elimination Using SSA

```
Sweep
  for each op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to
          i's nearest useful
          post-dominator
      if i is not a jump then
        delete i
```

i=12…17



Dead Code Elimination Using SSA

What's left?

- Algorithm eliminates useless definitions & some useless branches
- Algorithm leaves behind empty blocks & extraneous control-flow

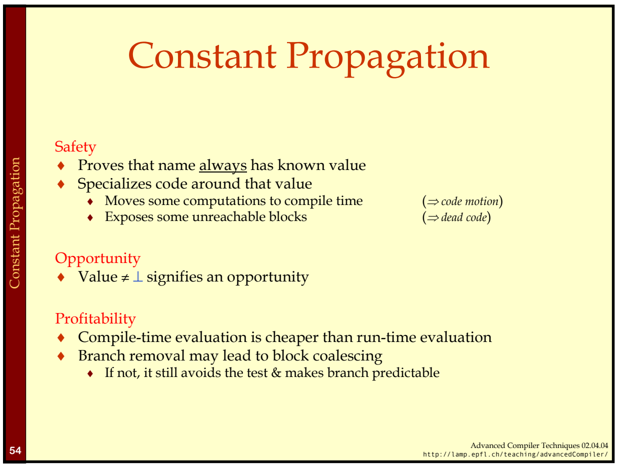
Algorithm from: Cytron, Ferrante, Rosen, Wegman, & Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM TOPLAS 13(4), October 1991

with a correction due to Rob Shillner

Two more issues

- Simplifying control-flow
- Eliminating unreachable blocks

Both are CFG transformations (no need for SSA)



Constant Propagation

Safety

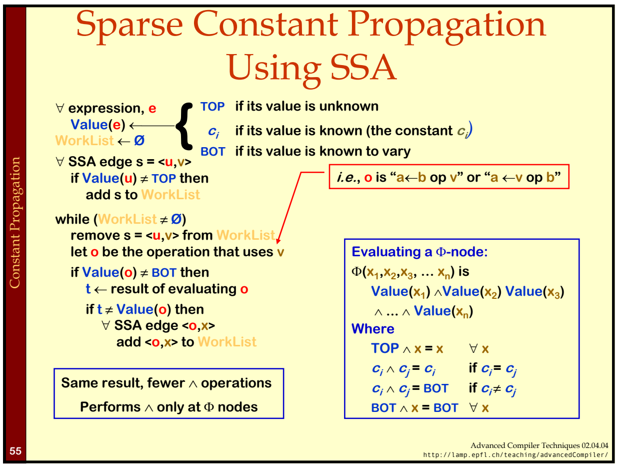
- Proves that name <u>always</u> has known value
- Specializes code around that value
  - Moves some computations to compile time (⇒ *code motion*)
  - Exposes some unreachable blocks (⇒ *dead code*)

Opportunity

- Value ≠ ⊥ signifies an opportunity

Profitability

- Compile-time evaluation is cheaper than run-time evaluation
- Branch removal may lead to block coalescing
  - If not, it still avoids the test & makes branch predictable



Sparse Constant Propagation Using SSA

```
∀ expression, e
   Value(e) ← { TOP  if its value is unknown
                c_i  if its value is known (the constant c_i)
                BOT  if its value is known to vary
WorkList ← Ø
∀ SSA edge s = <u,v>
   if Value(u) ≠ TOP then
      add s to WorkList

while (WorkList ≠ Ø)
   remove s = <u,v> from WorkList
   let o be the operation that uses v
   if Value(o) ≠ BOT then
      t ← result of evaluating o
      if t ≠ Value(o) then
         ∀ SSA edge <o,x>
            add <o,x> to WorkList
```

i.e., o is "a←b op v" or "a ←v op b"

Evaluating a Φ-node:

$\Phi(x_1,x_2,x_3, \ldots x_n)$ is

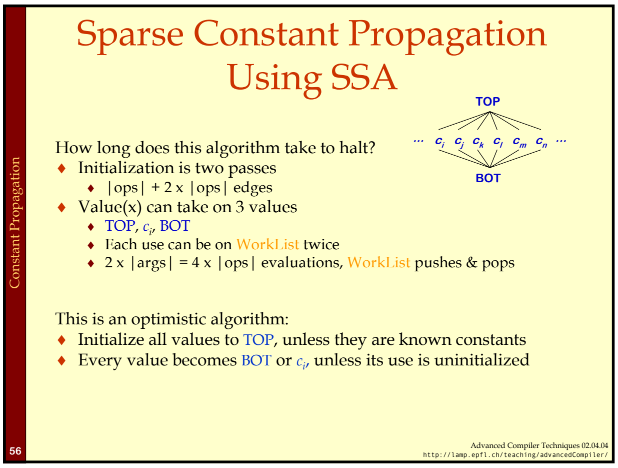
Value(x_1) ∧Value(x_2) Value(x_3) ∧ … ∧ Value(x_n)

Where

TOP ∧ x = x	∀ x
$c_i \wedge c_j = c_i$	if $c_i = c_j$
$c_i \wedge c_j$ = BOT	if $c_i \neq c_j$
BOT ∧ x = BOT	∀ x

Same result, fewer ∧ operations

Performs ∧ only at Φ nodes



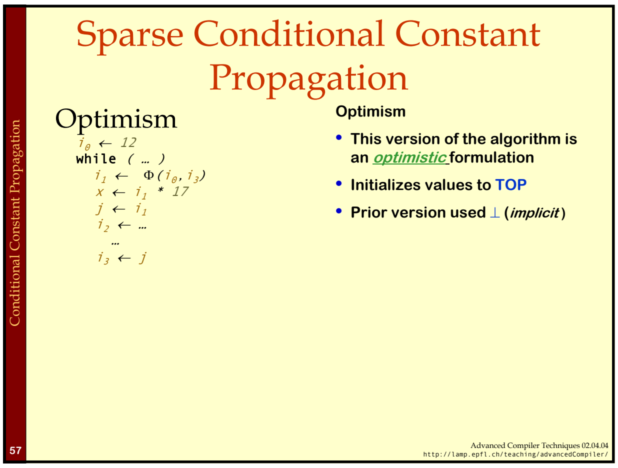
Sparse Constant Propagation Using SSA

How long does this algorithm take to halt?

- Initialization is two passes
  - |ops| + 2 x |ops| edges
- Value(x) can take on 3 values
  - TOP, c_i, BOT
  - Each use can be on WorkList twice
  - 2 x |args| = 4 x |ops| evaluations, WorkList pushes & pops

This is an optimistic algorithm:

- Initialize all values to TOP, unless they are known constants
- Every value becomes BOT or c_i, unless its use is uninitialized



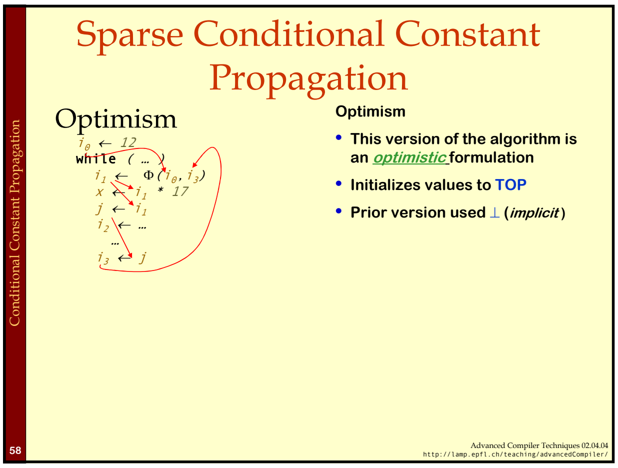
Sparse Conditional Constant Propagation

Optimism

```
i0 ← 12
while ( … )
   i1 ←  Φ(i0,i3)
   x ← i1 * 17
   j ← i1
   i2 ← …
    …
   i3 ← j
```

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)



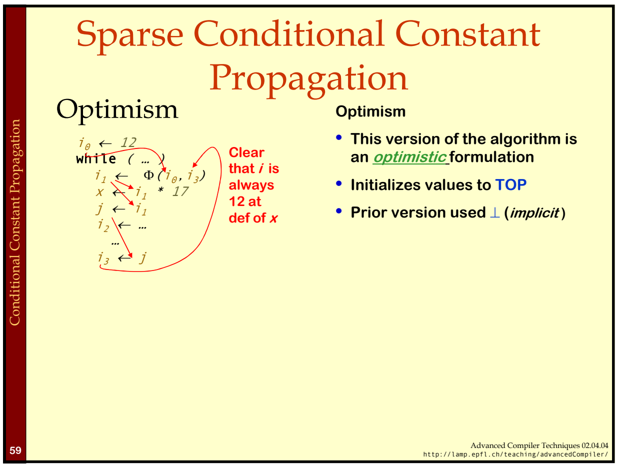
Sparse Conditional Constant Propagation

Optimism

```
i0 ← 12
while ( … )
   i1 ←  Φ(i0,i3)
   x ← i1 * 17
   j ← i1
   i2 ← …
    …
   i3 ← j
```

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)



Sparse Conditional Constant Propagation

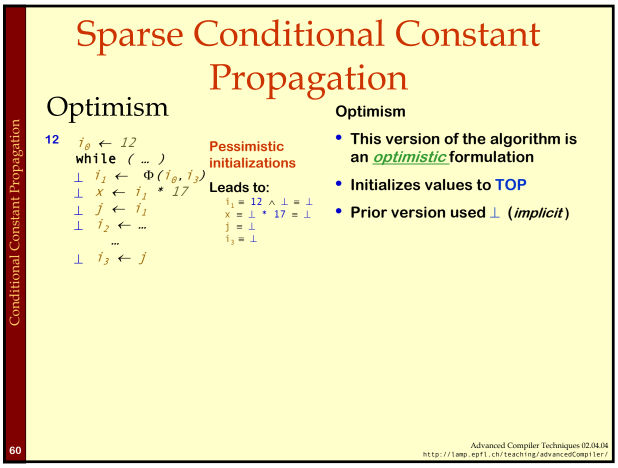
Optimism

```
i0 ← 12
while ( … )
   i1 ←  Φ(i0,i3)
   x ← i1 * 17
   j ← i1
   i2 ← …
    …
   i3 ← j
```

Clear that *i* is always 12 at def of *x*

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)



Sparse Conditional Constant Propagation

Optimism

```
12   i0 ← 12
     while ( … )
⊥    i1 ←  Φ(i0,i3)
⊥    x ← i1 * 17
⊥    j ← i1
⊥    i2 ← …
      …
⊥    i3 ← j
```

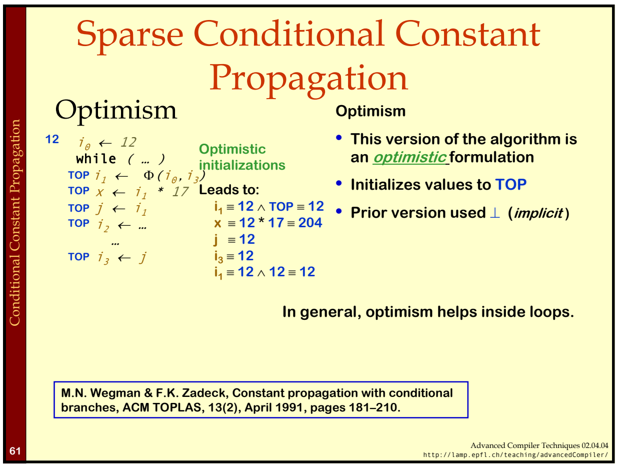
Pessimistic initializations

Leads to:

```
i1 ≡ 12 ∧ ⊥ ≡ ⊥
x ≡ ⊥ * 17 ≡ ⊥
j ≡ ⊥
i3 ≡ ⊥
```

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)



Sparse Conditional Constant Propagation

Optimism

```
12    i0 ← 12
      while ( … )
  TOP i1 ←  Φ(i0, i3)
  TOP x ← i1 * 17
  TOP j ← i1
  TOP i2 ← …
        …
  TOP i3 ← j
```

Optimistic initializations

Leads to:

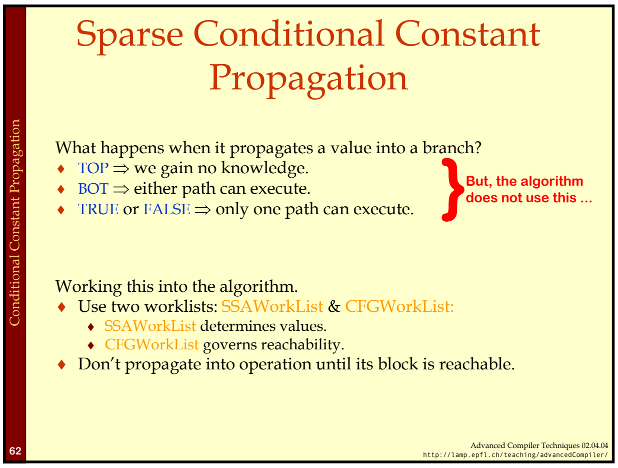
$i_1 \equiv 12 \wedge \text{TOP} \equiv 12$
$x \equiv 12 * 17 \equiv 204$
$j \equiv 12$
$i_3 \equiv 12$
$i_1 \equiv 12 \wedge 12 \equiv 12$

Optimism

- This version of the algorithm is an *optimistic* formulation
- Initializes values to TOP
- Prior version used ⊥ (*implicit*)

In general, optimism helps inside loops.

M.N. Wegman & F.K. Zadeck, Constant propagation with conditional branches, ACM TOPLAS, 13(2), April 1991, pages 181–210.

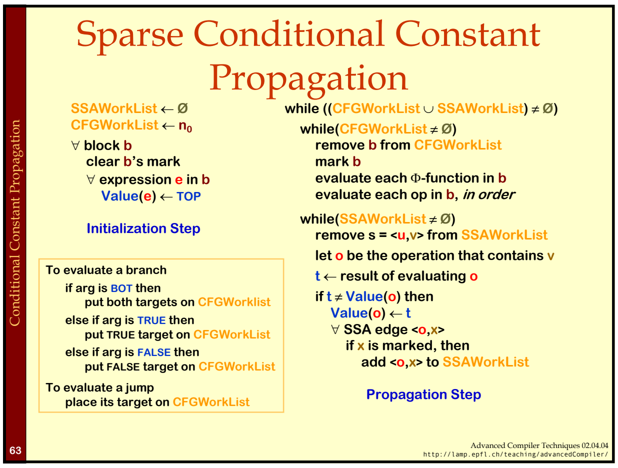


Sparse Conditional Constant Propagation

What happens when it propagates a value into a branch?
- TOP ⇒ we gain no knowledge.
- BOT ⇒ either path can execute.
- TRUE or FALSE ⇒ only one path can execute.

} But, the algorithm does not use this ...

Working this into the algorithm.
- Use two worklists: SSAWorkList & CFGWorkList:
  - SSAWorkList determines values.
  - CFGWorkList governs reachability.
- Don't propagate into operation until its block is reachable.



Sparse Conditional Constant Propagation

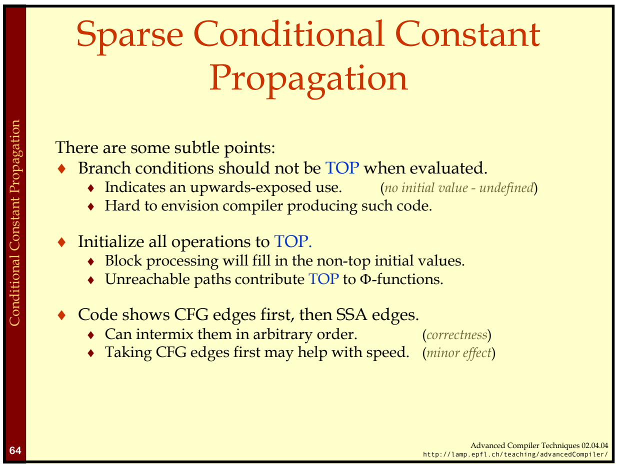
```
SSAWorkList ← Ø
CFGWorkList ← n0
∀ block b
   clear b's mark
   ∀ expression e in b
      Value(e) ← TOP
```

Initialization Step

```
To evaluate a branch
   if arg is BOT then
      put both targets on CFGWorklist
   else if arg is TRUE then
      put TRUE target on CFGWorkList
   else if arg is FALSE then
      put FALSE target on CFGWorkList
To evaluate a jump
   place its target on CFGWorkList
```

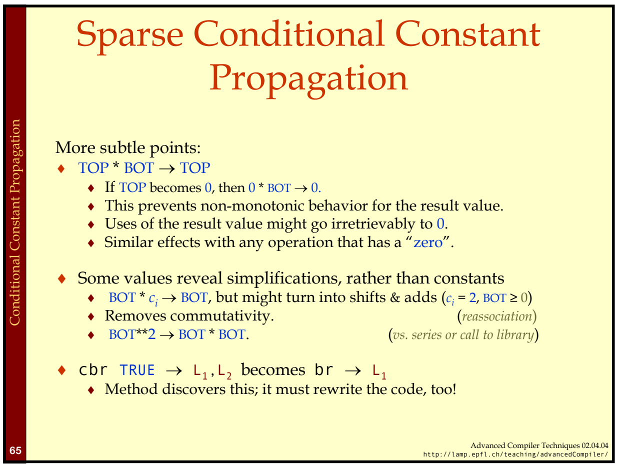
```
while ((CFGWorkList ∪ SSAWorkList) ≠ Ø)
   while(CFGWorkList ≠ Ø)
      remove b from CFGWorkList
      mark b
      evaluate each Φ-function in b
      evaluate each op in b, in order
   while(SSAWorkList ≠ Ø)
      remove s = <u,v> from SSAWorkList
      let o be the operation that contains v
      t ← result of evaluating o
      if t ≠ Value(o) then
         Value(o) ← t
         ∀ SSA edge <o,x>
            if x is marked, then
               add <o,x> to SSAWorkList
```

Propagation Step



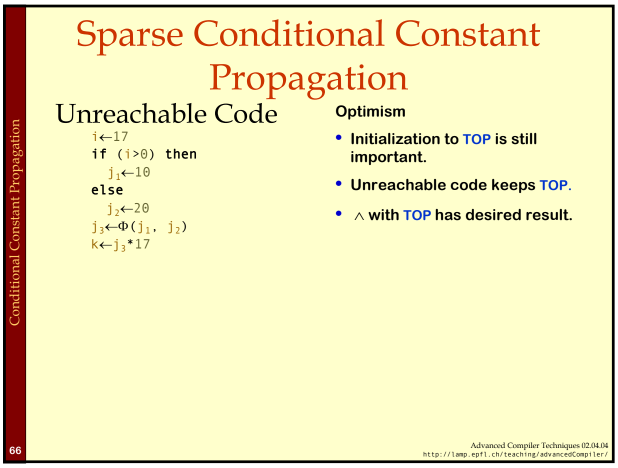
Sparse Conditional Constant Propagation

There are some subtle points:
- Branch conditions should not be TOP when evaluated.
  - Indicates an upwards-exposed use. (*no initial value - undefined*)
  - Hard to envision compiler producing such code.
- Initialize all operations to TOP.
  - Block processing will fill in the non-top initial values.
  - Unreachable paths contribute TOP to Φ-functions.
- Code shows CFG edges first, then SSA edges.
  - Can intermix them in arbitrary order. (*correctness*)
  - Taking CFG edges first may help with speed. (*minor effect*)



Sparse Conditional Constant Propagation

More subtle points:
- TOP * BOT → TOP
  - If TOP becomes 0, then 0 * BOT → 0.
  - This prevents non-monotonic behavior for the result value.
  - Uses of the result value might go irretrievably to 0.
  - Similar effects with any operation that has a "zero".
- Some values reveal simplifications, rather than constants
  - BOT * c_i → BOT, but might turn into shifts & adds (c_i = 2, BOT ≥ 0)
  - Removes commutativity. (*reassociation*)
  - BOT**2 → BOT * BOT. (*vs. series or call to library*)
- `cbr TRUE → L1,L2` becomes `br → L1`
  - Method discovers this; it must rewrite the code, too!



Sparse Conditional Constant Propagation

Unreachable Code

```
i←17
if (i>0) then
  j1←10
else
  j2←20
j3←Φ(j1, j2)
k←j3*17
```

Optimism

- Initialization to TOP is still important.
- Unreachable code keeps TOP.
- ∧ with TOP has desired result.

Sparse Conditional Constant Propagation

Unreachable Code

```
17    i←17
      if (i>0) then
10      j_1←10
      else
20      j_2←20
⊥     j_3←Φ(j_1, j_2)
⊥     k←j_3*17
```

All paths execute

Optimism

- Initialization to TOP is still important.
- Unreachable code keeps TOP.
- ∧ with TOP has desired result.

Sparse Conditional Constant Propagation

Unreachable Code

```
17    i←17
      if (i>0) then
TOP     j_1←10
      else
TOP     j_2←20
TOP   j_3←Φ(j_1, j_2)
170   k←j_3*17
```

With SCC marking blocks

Optimism

- Initialization to TOP is still important.
- Unreachable code keeps TOP.
- ∧ with TOP has desired result.

Sparse Conditional Constant Propagation

Unreachable Code

```
17    i←17
      if (i>0) then
10      j_1←10
      else
TOP     ~~j_2←20~~
10    j_3←Φ(j_1, j_2)
170   k←j_3*17
```

With SCC marking blocks

Optimism

- Initialization to TOP is still important.
- Unreachable code keeps TOP.
- ∧ with TOP has desired result.

Cannot get this any other way:

- DEAD code cannot test (i > 0).
- DEAD marks j_2 as useful.

Sparse Conditional Constant Propagation

Unreachable Code

```
17    i←17
      if (i>0) then
10      j_1←10
      else
TOP     ~~j_2←20~~
10    j_3←Φ(j_1, j_2)
170   k←j_3*17
```

With SCC marking blocks

Optimism

- Initialization to TOP is still important.
- Unreachable code keeps TOP.
- ∧ with TOP has desired result.

In general, combining two optimizations can lead to answers that cannot be produced by any combination of running them separately.

This algorithm is one example of that general principle.

Combining register allocation & instruction scheduling is another …

Using SSA Form for Optimizations

In general, using SSA conversion leads to:

- Cleaner formulations.
- Better results.
- Faster algorithms.

We've seen two SSA-based algorithms.

- Dead-code elimination.
- Sparse conditional constant propagation.