Lecture 12: Introduction to Optimizing Compilers

- Motivation for using optimizing compilers
- Control flow analysis
  - Dominance analysis
  - Loop analysis
- Scalar optimizations on SSA Form
  - Copy propagation
  - Global value numbering
  - Partial redundancy elimination
  - Operator strength reduction
  - Constant propagation
  - Dead code elimination
- Instruction scheduling
- Register allocation

Motivation for Using Optimizing Compilers

- ***Execution time / energy reduction***: possible speedups due to compiler optimization depend on the application and the architecture (e.g. pipeline, SIMD, caches, multicore).
- Example: SPEC CPU2000 benchmark `gzip` on a Power machine: PowerMac Quad G5/2.5 GHz with similar cores to the IBM Power4 plus SIMD (the first multicore chip)

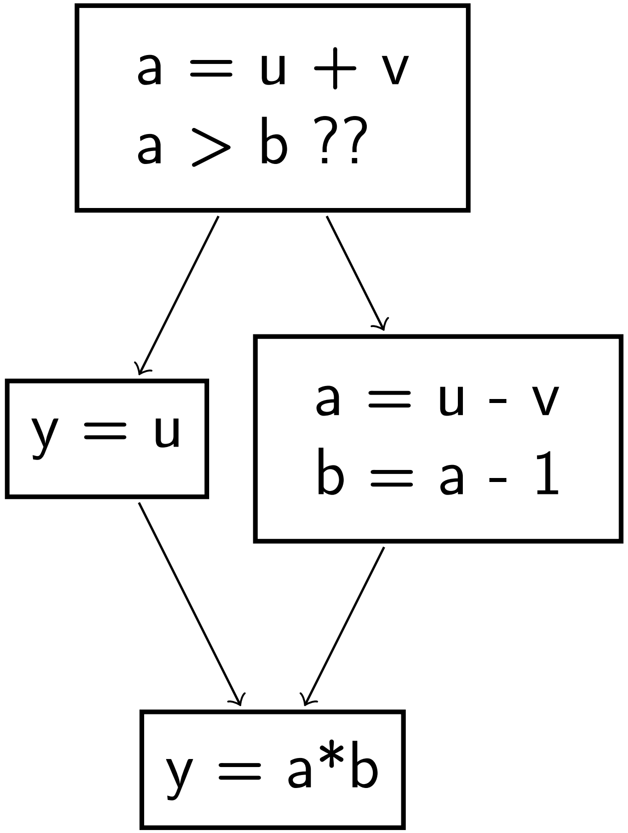
Compiler	Opt level	Execution time
IBM XL	max opt	135 s
GCC 4.7.2	max opt	145 s
GCC 4.7.2	no opt	494 s

- Increase ***programmer productivity*** by knowing
  - what the compiler can optimize faster and better than himself/herself, and
  - compilers' limitations and how to write code that helps them to do better automatic optimization.

Control-Flow Graph: Example C Code

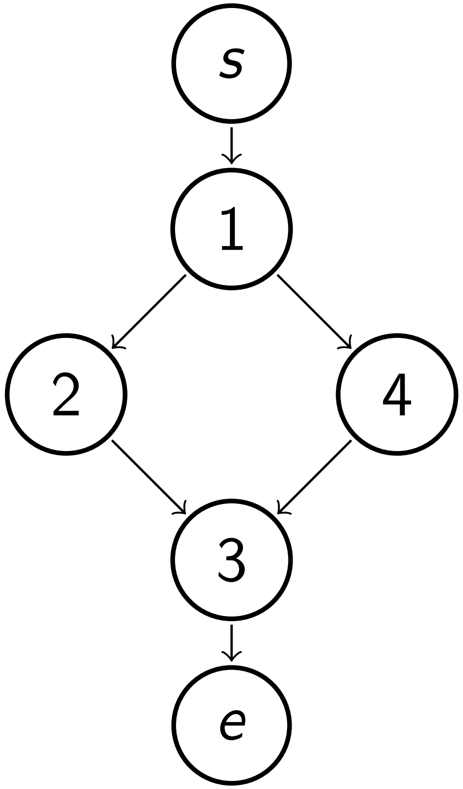
```
a = u + v;
if (a > b) {
           y = u;
} else {
           a = u - v;
           b = a - 1;
}
y = a * b;
```

Control-flow graph: Basic Blocks and Branches



Basic block: sequence of instructions with no label or branch
CFG: directed graph with basic blocks as nodes and branches as edges

Control-Flow Graph: the CFG View

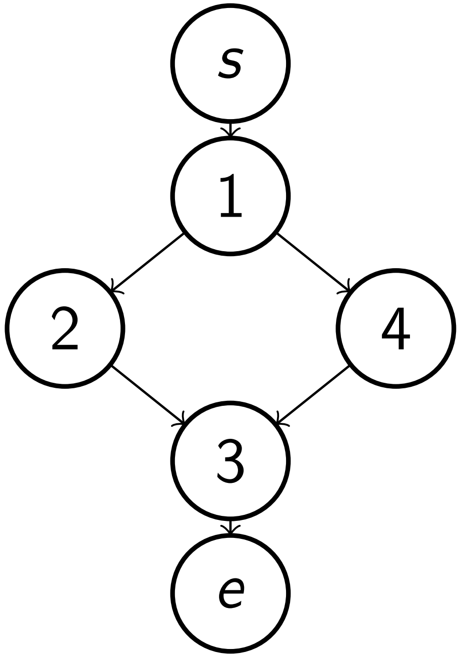


Special nodes:

- the first node is called s — start
- the last node is called e — exit

Definition of Dominance

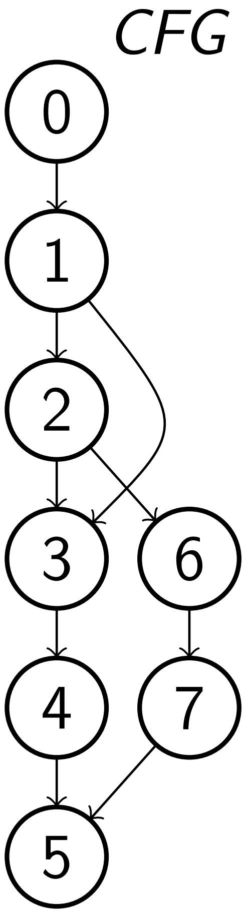
- Consider a control flow graph $G(V, E, s, e)$ and two vertices $u, v \in V$.
- If every path from s to v includes u then u **dominates** v, written $u \underline{\gg} v$.
- For example 1 dominates itself, 2, 3, 4, and e.



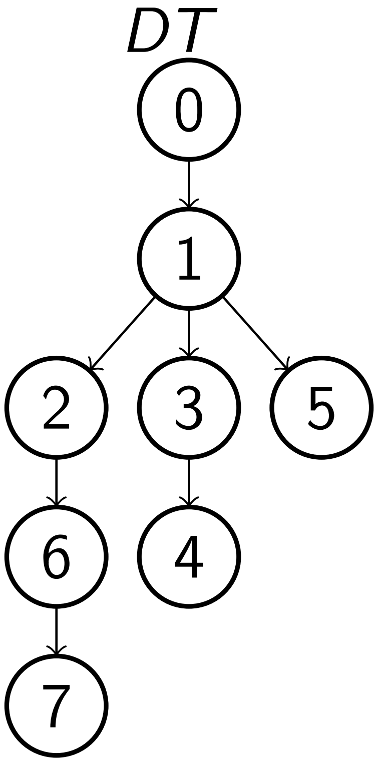
Immediate dominators

- The set $dom(w)$ is a total order.
- In other words: if $u, v \in dom(w)$ then either $u \gg\!\!\underline{\ } v$ or $v \gg\!\!\underline{\ } u$.
- We can order all vertices in $dom(w)$ to find the "closest" dominator of w.
- First let $S \leftarrow dom(w) - \{w\}$.
- Consider any two vertices in S.
- Remove from S the one which dominates the other. Repeat.
- The only remaining vertex in S is the **immediate dominator** of w.
- We write the immediate dominator of w as $idom(w)$.
- Every vertex, except s, has a unique immediate dominator.
- We can draw the immediate dominators in a tree called the **dominator tree**.

The Dominator Tree



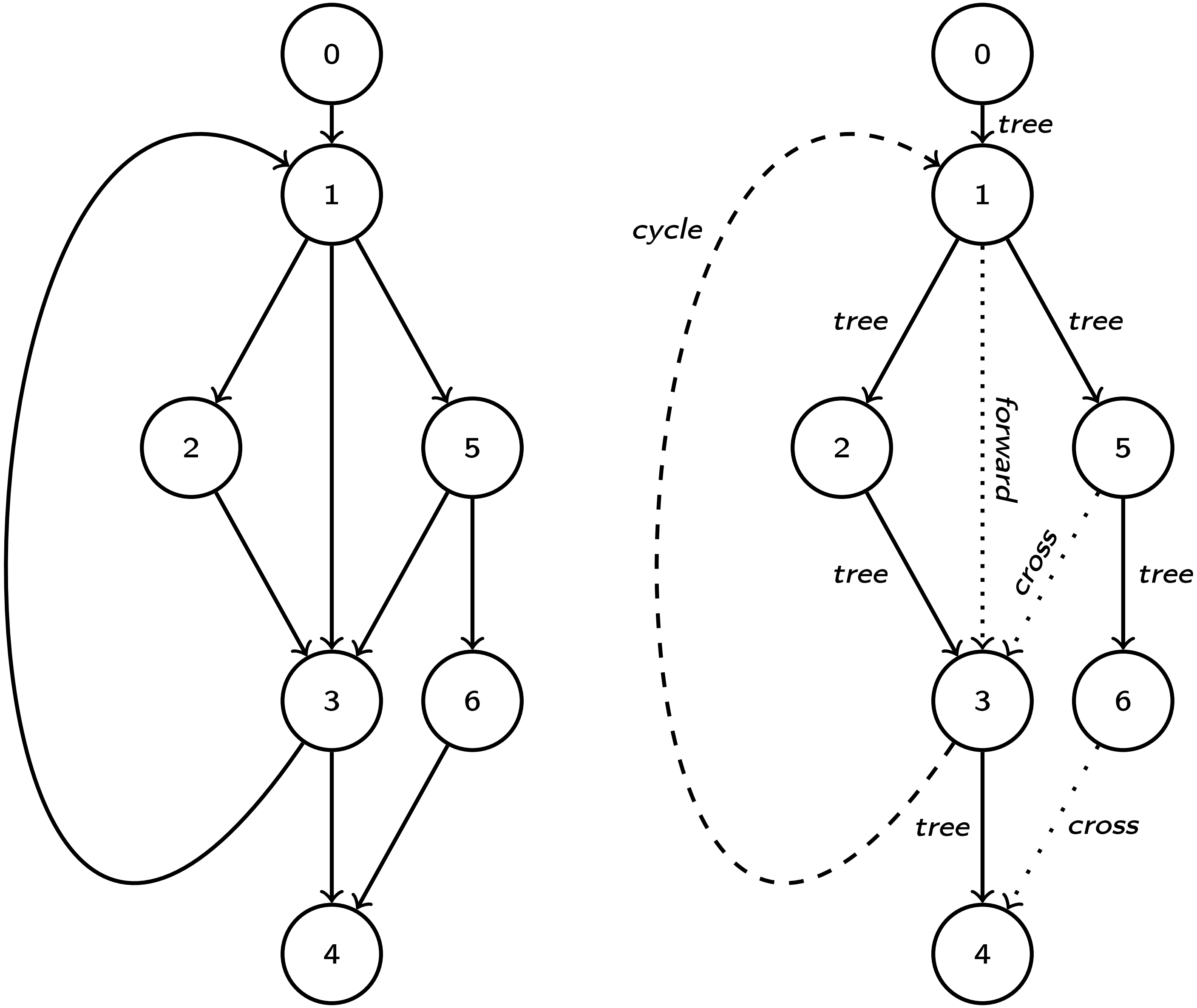
w	$idom(w)$
0	-
1	0
2	1
3	1
4	3
5	1
6	2
7	6



The Lengauer-Tarjan Algorithm

- The LT algorithm is the standard algorithm for computing the dominator tree.
- It was completed in 1979 by Robert Tarjan and his PhD student Thomas Lengauer at Stanford.
- Thomas Lengauer is the brother of Christian Lengauer whose group in Passau has developed many high order transformations which are now being implemented in clang and gcc.
- The LT algorithm calculates the immediate dominators in a clever way and is based on insights from depth first search.

Loop Analysis Using Dominance: More About DFS



Loop Analysis Using Dominance: Cycle Arcs

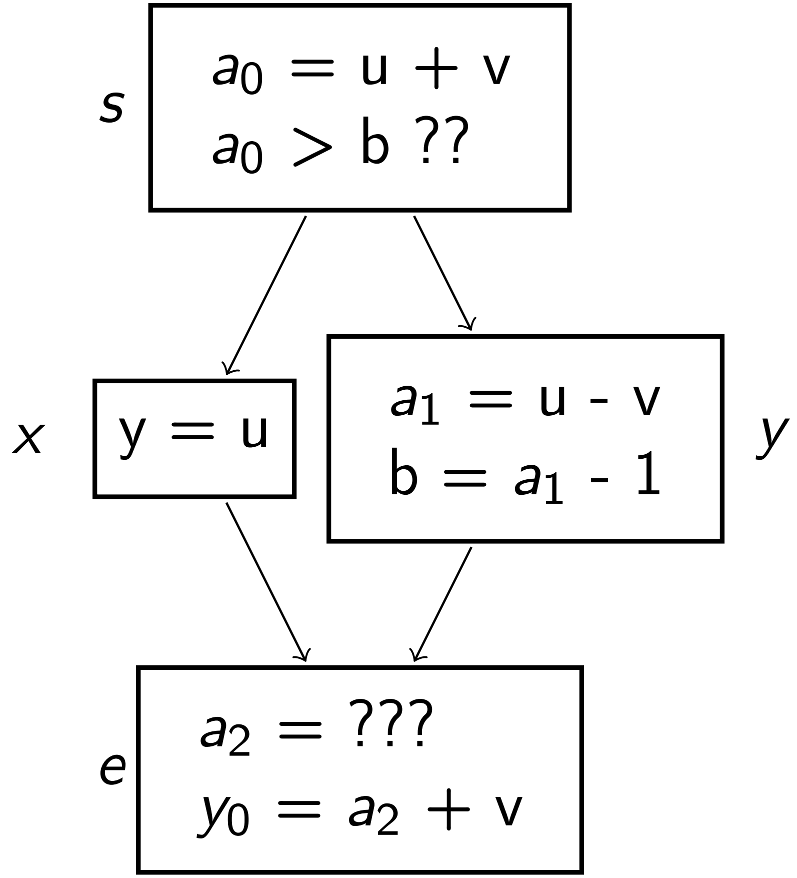
- Loops can be found by exploiting cycle arcs.
- In a **natural loop**, one vertex called the header dominates all vertices in the loop.
- Suppose there is a cycle arc (v, u) such as $(3, 1)$ above.
- Then, if $u \ggg v$ we know that u is a natural loop header.
- We can search backwards from v and include everything we find to the loop, stopping at u.
- Due to $u \ggg v$ we cannot go wrong and miss u.

Static Single Assignment For Form: SSA Form

- A variable is only assigned to by one unique instruction
- That instruction dominates all the uses of the assigned value
- We introduce a new variable name at each assignment
- SSA Form is the key to elegant and efficient scalar optimization algorithms
- Invented by IBM Research Yorktown Heights in New York

But what to do when paths from different assignments join???

Partial Translation to SSA Form



s

a_0 = u + v

a_0 > b ??

x

y = u

a_1 = u - v

b = a_1 - 1

y

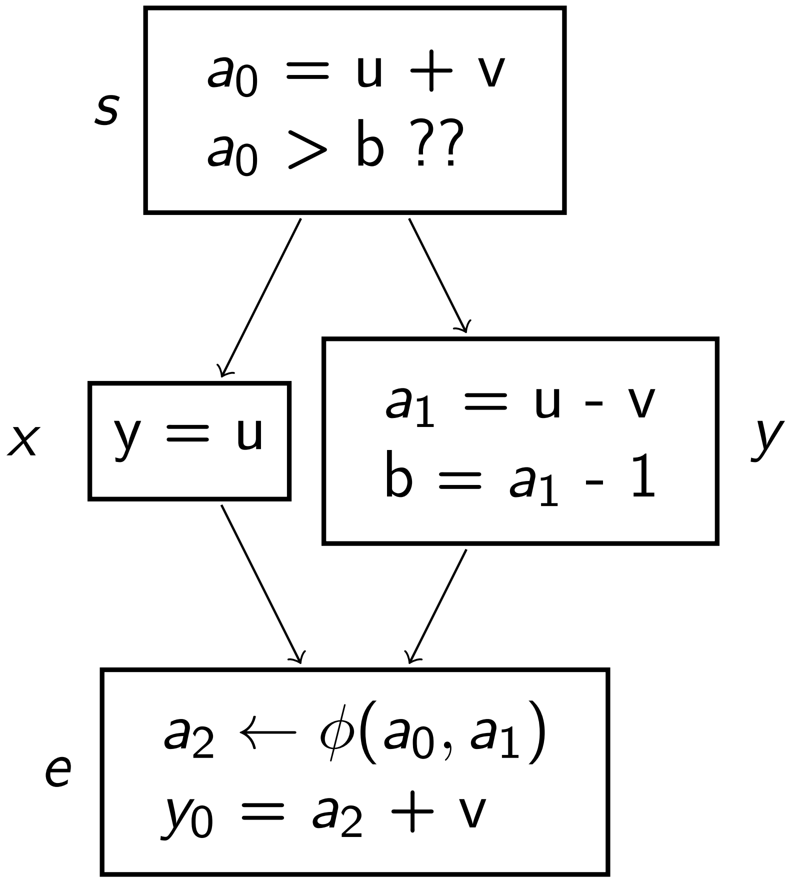
e

a_2 = ???

y_0 = a_2 + v

In node e: if we came from node x we let $a_2 \leftarrow a_0$ and if we came from node y we let $a_2 \leftarrow a_1$. This operation is called the ϕ-function.

Our Example Translated to SSA Form



A Function Translated to SSA Form

- We insert a ϕ-function where the paths from two different assignments of the same variable join
- With the ϕ-function, each definition dominates its uses

Copy Propagation

```
x0 = a0 + b0;
if (...) {
   ...;
}
y0 = x0;     /* COPY */
if (...) {
   ...;
}
c0 = y0 + 1; /* USE */
```

```
x0 = a0 + b0;
if (...) {
   ...;
}

if (...) {
   ...;
}
c0 = x0 + 1;
```

- With SSA Form we can know that it is correct to replace `y0` with `x0`
- The values of `x0` and `y0` do not change after the definition (in a static sense)

Constant Propagation with Iterative Dataflow Analysis

```
a = 1;
b = 2;
if (a < b)
        c = 3;
else
        c = 4;
put(c);
```

- Invented by Gary Kildall in 1973.
- Each variable can be either
  - Unknown
  - Constant
  - Non-constant
- Iterative dataflow analysis is performed to determine whether a variable is constant and in that case which constant.
- All branches (i.e. paths in a function) are assumed to be executable.
- Since c cannot be both 3 and 4 it's assumed to be nonconstant.

Constant Propagation with Conditional Branches

```
a = 1;
b = 2;
if (a < b)
        c1 = 3;
else
        c2 = 4;
c3 = phi(c1, c2);
put(c3);
```

- Based on SSA Form.
- Invented at IBM Research and published 1991.
- Recall Kildall's algorithm assumed every branch was executable.
- This algorithm assumes that nothing is executable except the start vertex.
- The function is interpreted and the constant expressions are propagated.
- The interpretation proceeds until no new knowledge about constants can be found.

Key Idea with ϕ-functions

```
a = 1;
b = 2;
if (a < b)
        c1 = 3;
else
        c2 = 4;
c3 = phi(c1, c2);
put(c3);
```

- Thanks to SSA Form, one statement and variable is analyzed at a time.
- At a ϕ-function, if any operand is nonconstant the result is nonconstant, and if any two constants have different values the result also is nonconstant.
- However, operands corresponding to branches which we don't think will be executed can be ignored for the moment.
- While interpreting the program we may later realize that the branch in fact might be executed and then the ϕ-function will be re-evaluated.
- We can ignore c_2 and let c_3 be 3.

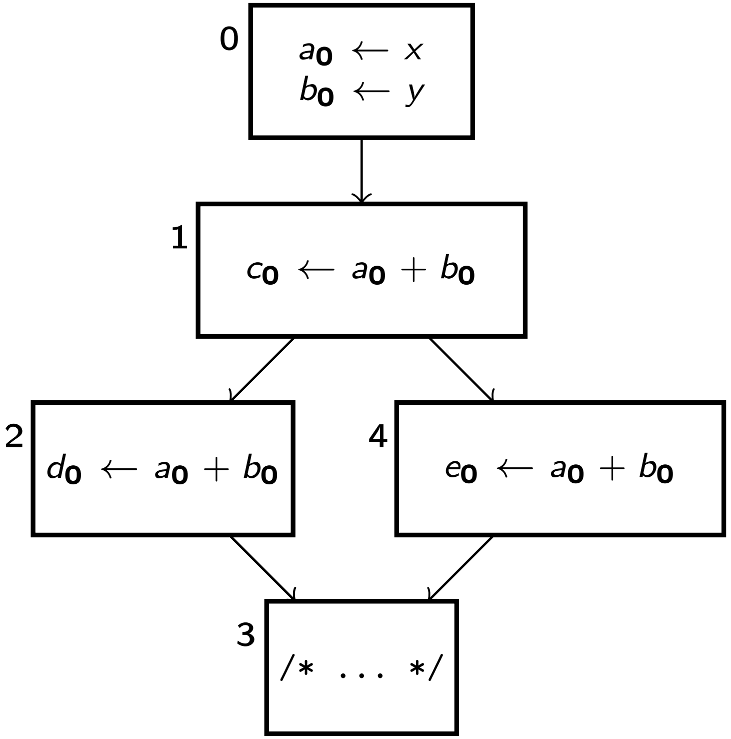
Redundancy Elimination

- An expression $a + b$ is **redundant** if it is evaluated multiple times with identical values of the operands.
- Eliminating redundant expressions is a very important optimization goal.
- There are different approaches to redundancy elimination, including
  1. Hash-Based Value Numbering
  2. Global Value Numbering
  3. Common Subexpression Elimination
  4. Code Motion out of Loops
  5. Partial Redundancy Elimination
- We will look at 1, 2, and 5.

Value Numbering

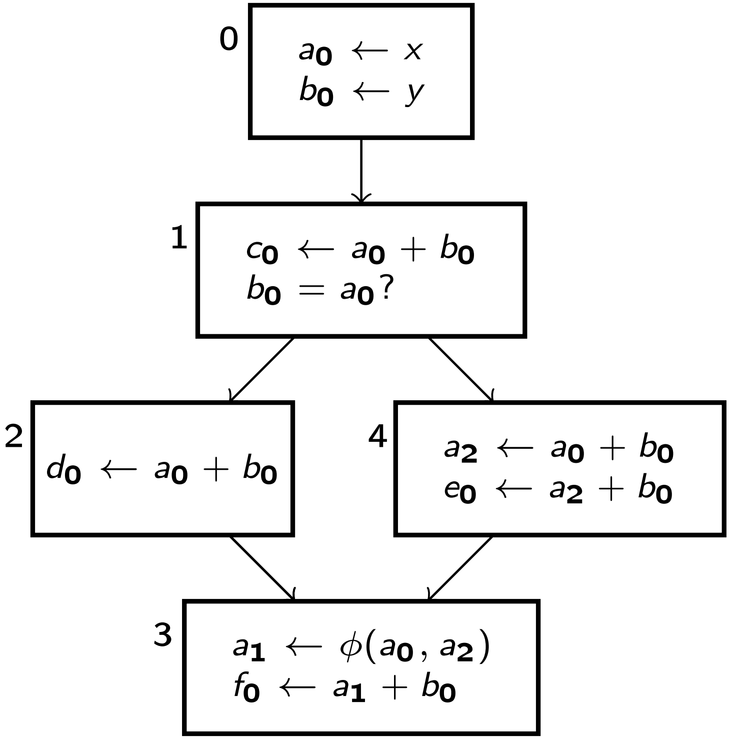
- The name is due to each expression, e.g. $t_i \leftarrow a + b$, is given a number, essentially a hash-table index.
- In subsequent occurrences of $t_j \leftarrow a + b$ it is checked whether the statement can be changed to $t_j \leftarrow t_i$.
- This is a very old optimization technique with one version that is performed during translation to SSA Form and other versions when the code already is on SSA Form.
- There are obviously older versions used before SSA Form but we will not look at them.

Example 1


0
$a_0 \leftarrow x$
$b_0 \leftarrow y$
1
$c_0 \leftarrow a_0 + b_0$
2
$d_0 \leftarrow a_0 + b_0$
4
$e_0 \leftarrow a_0 + b_0$
3
/* ... */

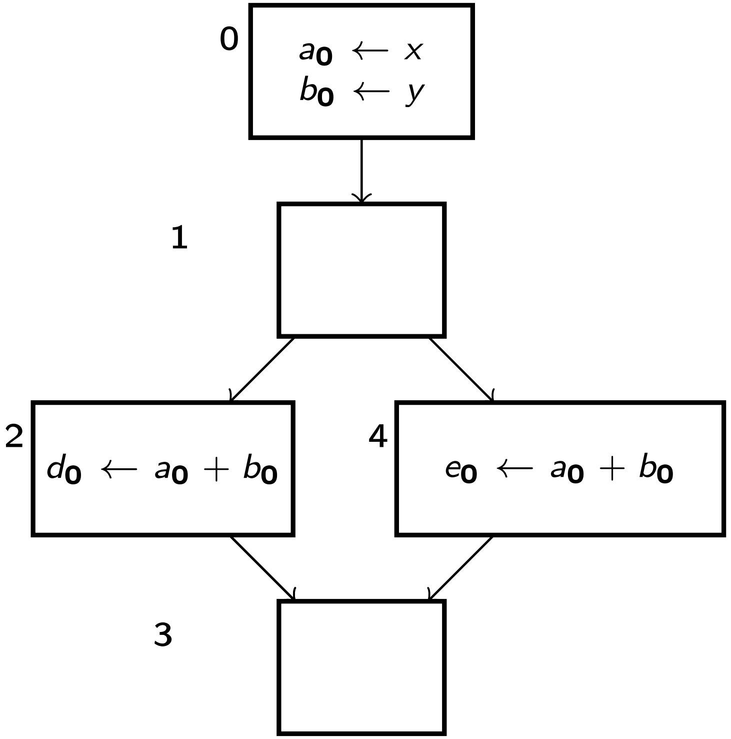
- In vertex 1 the expression $a_0 + b_0$ is first computed.
- The redundant occurrences of $a_0 + b_0$ can easily be removed.
- On SSA Form we simply check that the variable versions are the same in the current and previous occurrence.

Example 2


0
$a_0 \leftarrow x$
$b_0 \leftarrow y$
1
$c_0 \leftarrow a_0 + b_0$
$b_0 = a_0?$
2
$d_0 \leftarrow a_0 + b_0$
4
$a_2 \leftarrow a_0 + b_0$
$e_0 \leftarrow a_2 + b_0$
3
$a_1 \leftarrow \phi(a_0, a_2)$
$f_0 \leftarrow a_1 + b_0$

- The occurrences in vertices 3 and 4 cannot mistakenly be regarded as useful due to mismatching variable versions.

Example 3



- Obviously there are no redundant expressions here.
- We could perhaps save memory by computing $a_0 + b_0$ in vertex 1 but that is not a goal for redundancy elimination.
- Which data structure should we use for performing value numbering during translation to SSA Form?

The Power of Global Value Numbering

```
int h(int a, int b)
{
    int x, y;

    x = 1;
    y = 1;
    do {
        a = a + b;
        x = x + a;
        y = y + a;
    } while (a > 0);
    return x + y;
}
```

```
int h(int a, int b)
{
    int x, y;

    x = 1;
    do {
        a = a + b;
        x = x + a;
    } while (a > 0);
    return x + x;
}
```

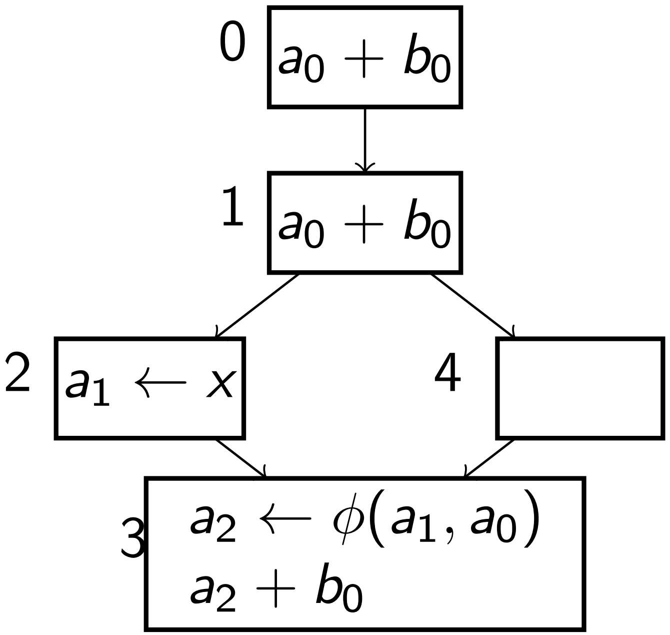
Purpose of Partial Redundancy Elimination

- Partial Redundancy Elimination, or **PRE**, can eliminate both **full** and **partial** redundancies.
- Full redundancies: when the expression is available from all predecessor basic blocks.
- Partial redundancies: when the expression is only available from some but not all predecessor basic blocks.
- Partial redundancies also covers loops, i.e. PRE can move code out from loops.

Partial Redundancy Elimination History

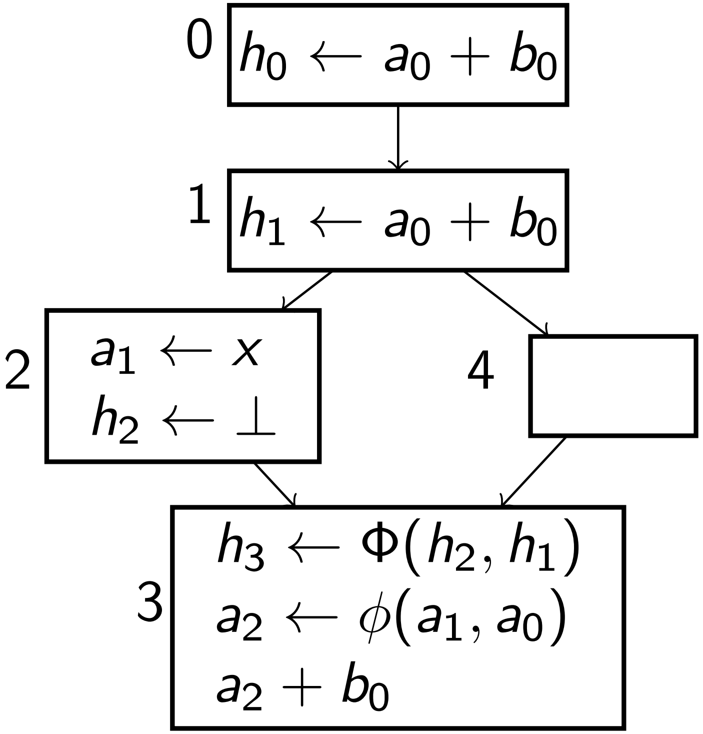
- PRE was invented by Morel and Renvoise in 1979.
- Then Fred Chow in his PhD thesis at Stanford from 1983 (with John Hennessy as supervisor) improved it.
- In 1992 Knoop et al. published a version of PRE which is optimal in the sense of minimizing register pressure. They called their algorithm **Lazy Code Motion**.
- It was stated by a famous researcher that PRE cannot be done on SSA Form since SSA Form involves variables while PRE involves expressions.
- ”Cannot” is dangerous to state in public...
- In 1999 Kennedy and Chow and others at SGI published the SSA formulation of Lazy Code Motion and called it **SSAPRE**.

Limitations of Value Numbering



- Both hash-based and global value numbering can optimize the full redundancy in vertex 1.
- None of them can optimize the partial redundancy in vertex 3.

The Key Idea of SSAPRE


0
$h_0 \leftarrow a_0 + b_0$
1
$h_1 \leftarrow a_0 + b_0$
2
$a_1 \leftarrow x$
$h_2 \leftarrow \perp$
4
3
$h_3 \leftarrow \Phi(h_2, h_1)$
$a_2 \leftarrow \phi(a_1, a_0)$
$a_2 + b_0$

- We create Φ-functions for the hypothetical variable h.
- After SSAPRE, Φ-functions become normal ϕ-functions and they are really the same (different notation to distinguish between them only).
- By inserting the expression $a + b$ at Φ-operands with the value $\perp$ (”bottom”), the partial redundancy in vertex 3 becomes a full redundancy and can be eliminated.

Operator Strength Reduction

```
double  a[N];

for (i = 0; i < N; ++i)
        x += a[i];
```

```
double*     p = a;
double*     end = &a[N];

while (p < end)
        x += *p++;
```

- The most important purpose is to rewrite the code to the left into the code to the right.
- C/C++ compilers are required to make it possible to use the address of the array element **after** the last declared element.
- Typically, in total one extra byte might be wasted in memory due to this.
- It's **not** one extra byte per array but rather per memory segment.

Invalid C Code

```
double  a[N];

for (i = N-1; i >= 0; --i)
        x += a[i];
```

```
double*     p = &a[N];

while (--p >= a)
       x += *p;
```

- In the last iteration `p == &a[-1]` in the comparison.
- The compiler is not required to make that address valid.
- The code to the right triggers undefined behavior if performed by the programmer.

Another Name for OSR

OSR is also known as Induction Variable Elimination

```
do {
    x = x + a[i];
    i = i + 1;
} while (i < N);
```

```
do {
    s = i * 4;
    t = load a+s;
    x = x + t;
    i = i + 1;
} while (i < N);
```

Basic och dependent IV

The primary goal is to get rid of the multiplication

```
do {
    s = i * 4;
    t = load a+s;
    x = x + t;
    i = i + 1;
} while (i < N);
```

- i is a *basic* induction variable
- Classes of *dependent* induction variables: $j \leftarrow b \times i + c$, i is a basic IV
- $s \leftarrow 4 \times i + 0$

Strength reduction

```
do {
    s = i * 4;
    t = load a+s;
    x = x + t;
    i = i + 1;

} while (i < N);
```

```
s = 4 * i;
do {

    t = load a+s;
    x = x + t;
    i = i + 1;
    s = s + 4;
} while (i < N);
```

- Initialize the dependent IV before the loop
- Increment the dependent IV just after the basic IV is incremented
- Maybe we can get rid of the basic IV now?

Linear function test replacement

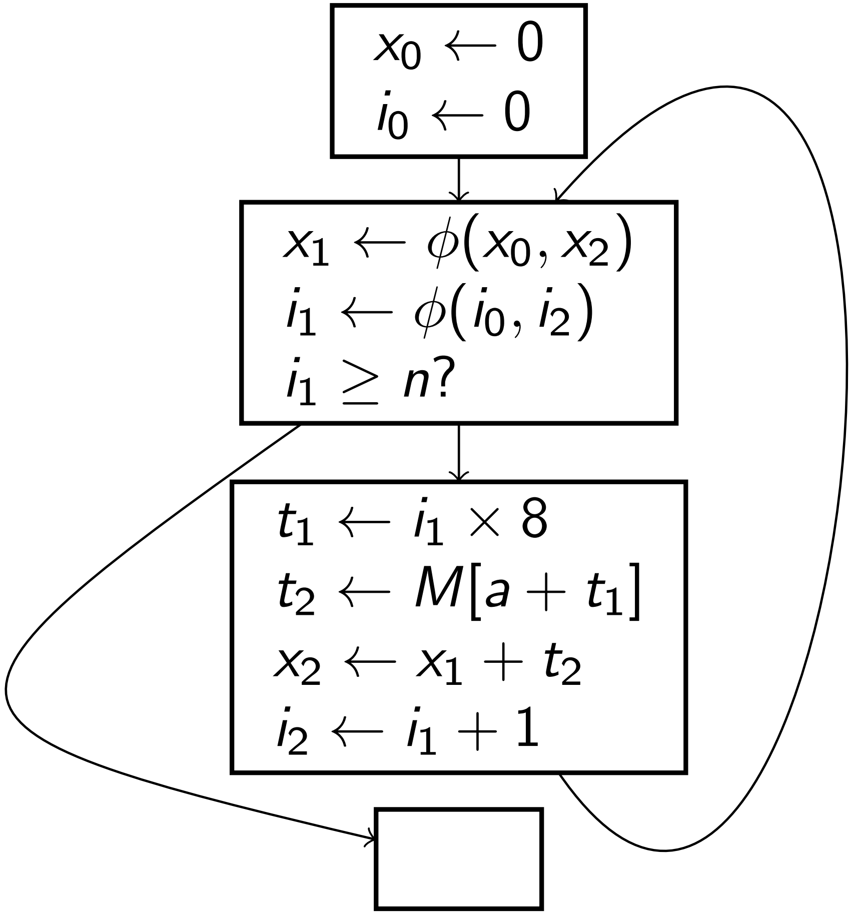
```
s = 4 * i;
do {
    t = load a+s;
    x = x + t;
    i = i + 1;
    s = s + 4;
} while (i < N);
```

```
m = 4 * N;
s = 4 * i;
do {
    t = load a+s;
    x = x + t;
    s = s + 4;
 } while (s < m);
```

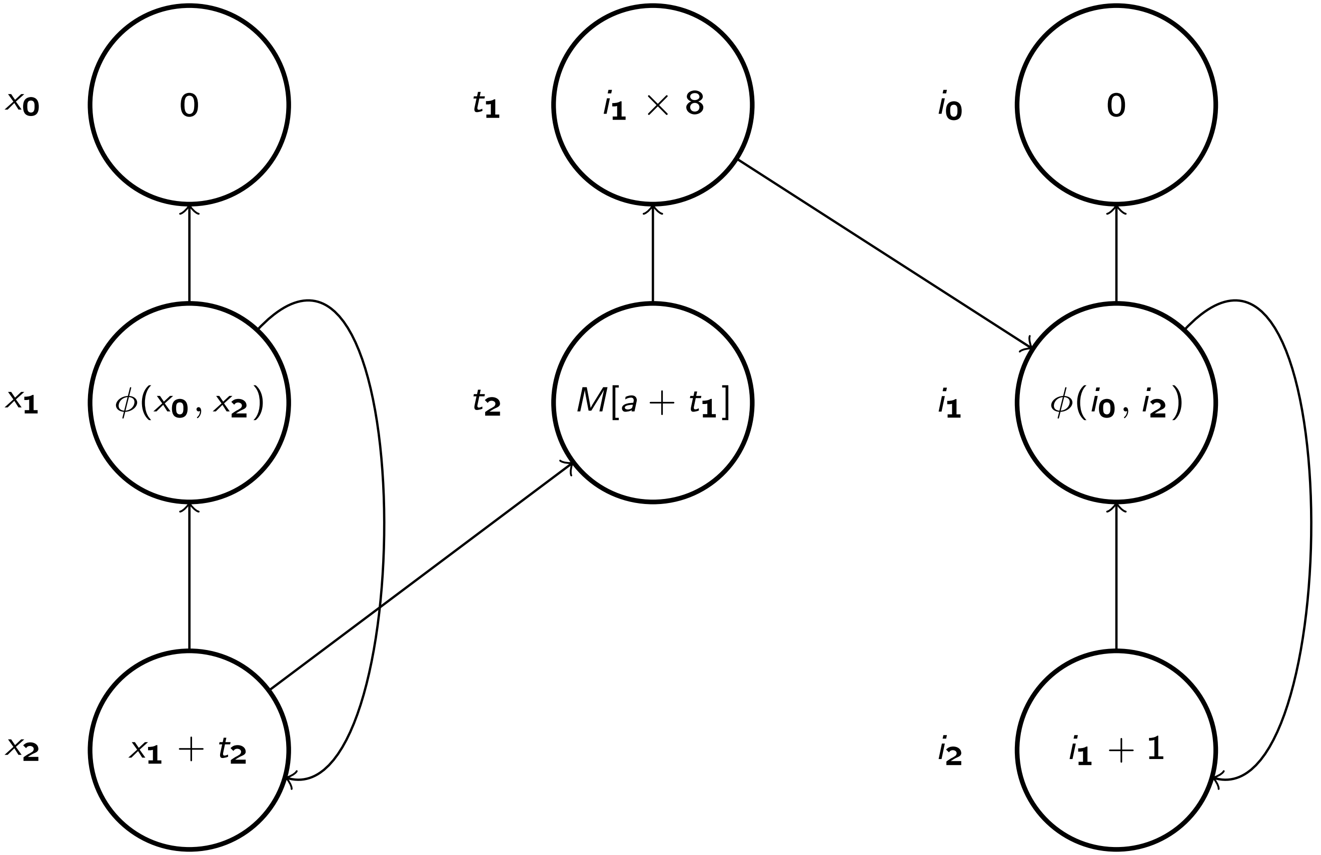
- $s = i \times b + c$ (we have $b = 4$ and $c = 0$)
- $i = \frac{s-c}{b}$
- $i < N \Rightarrow \frac{s-c}{b} < N \Rightarrow s < N \times b + c$, if $b > 0$

A Loop and its SSA Representation

```
double  a[N];

for (i = 0; i < N; ++i)
        x += a[i];
```

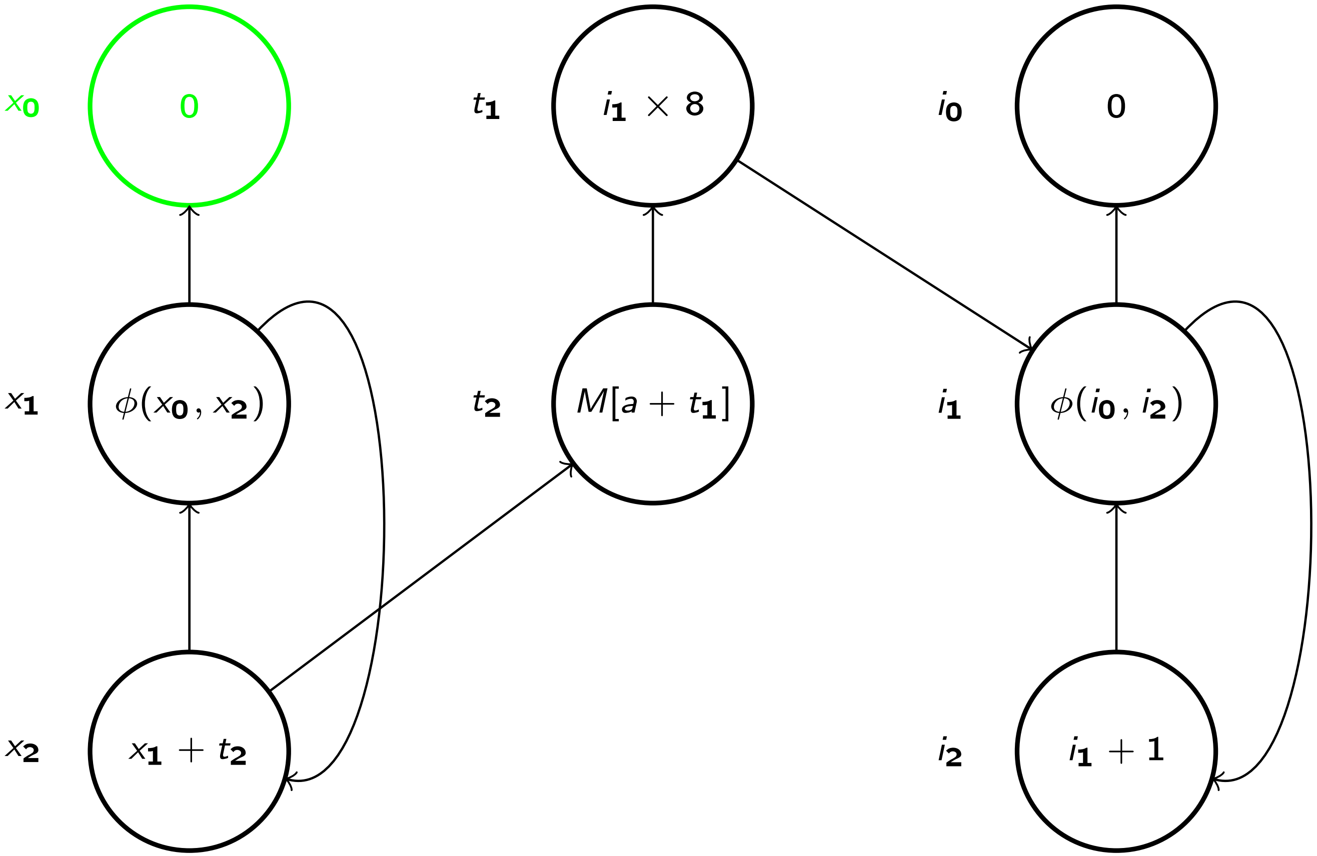

$x_0 \leftarrow 0$
$i_0 \leftarrow 0$
$x_1 \leftarrow \phi(x_0, x_2)$
$i_1 \leftarrow \phi(i_0, i_2)$
$i_1 \geq n?$
$t_1 \leftarrow i_1 \times 8$
$t_2 \leftarrow M[a + t_1]$
$x_2 \leftarrow x_1 + t_2$
$i_2 \leftarrow i_1 + 1$

The SSA Graph of the Loop



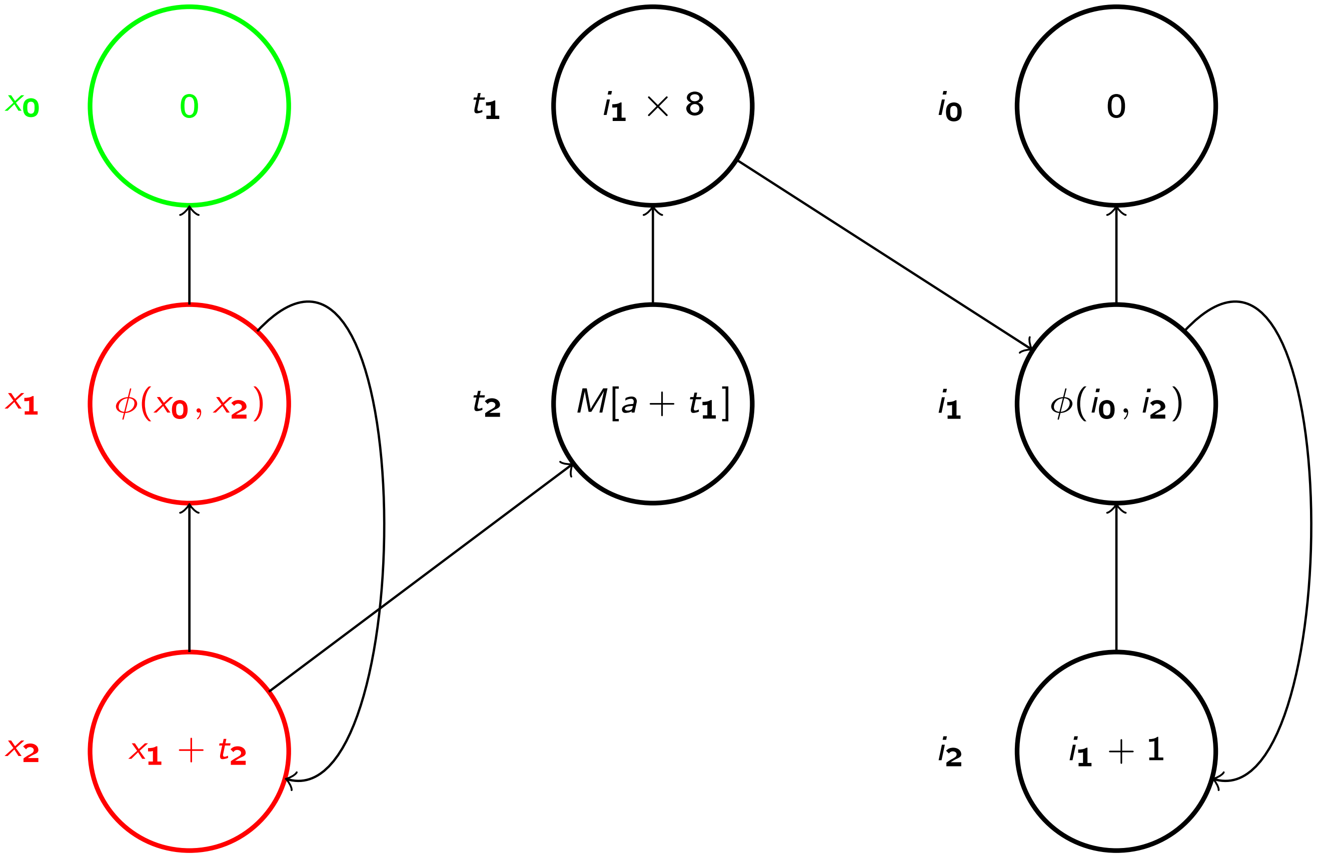
- We first find all strongly connected components of the SSA graph.
- We want to copy the SCC of i and modify the copy for t_1.
- Therefore we want to have processed i before processing t_1.
- Let us start with x.

Processing of x_0



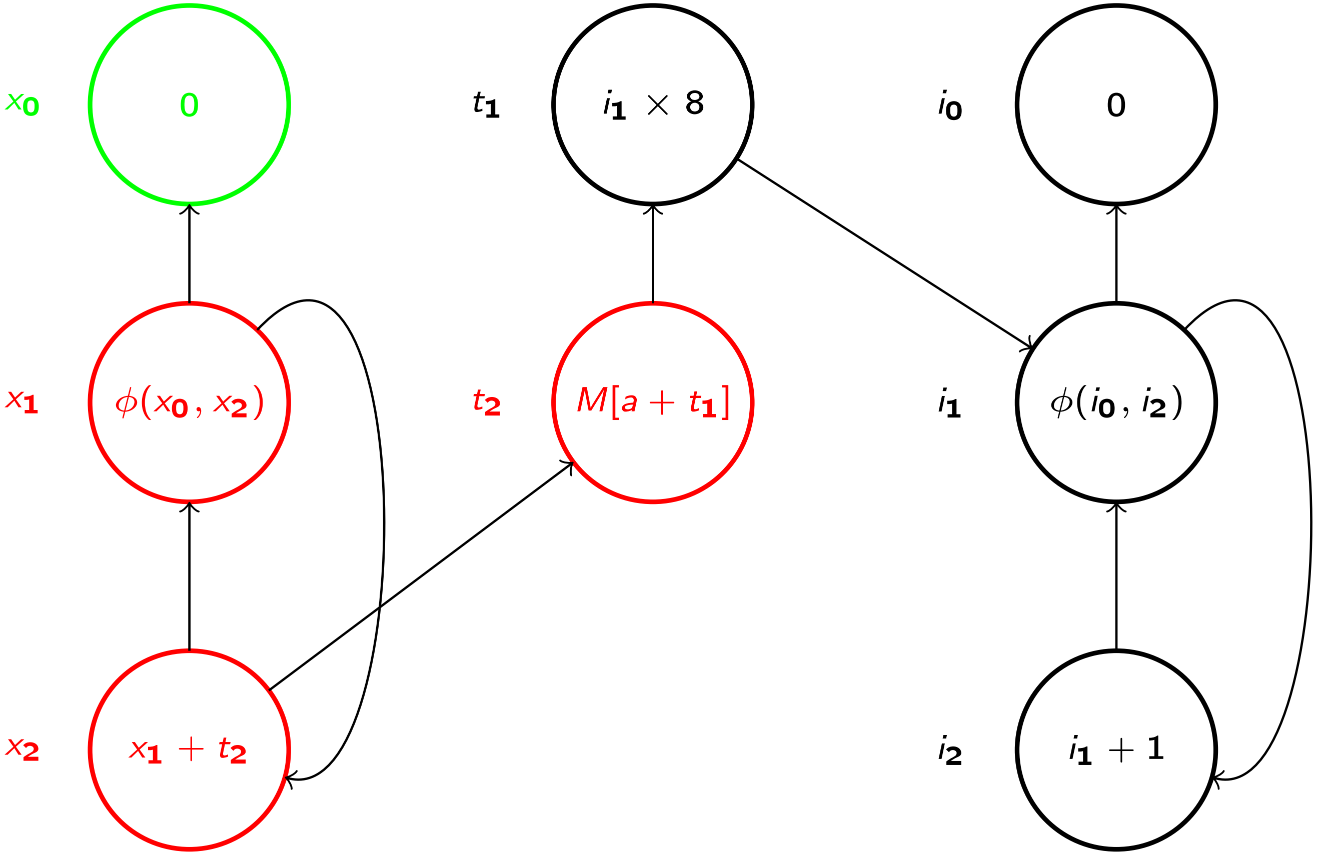
- $SCC_0 = \{x_0\}$. Empty stack.
- Nodes processed in a SCC are green.
- Next processing x_1.

Processing of x_1 and x_2



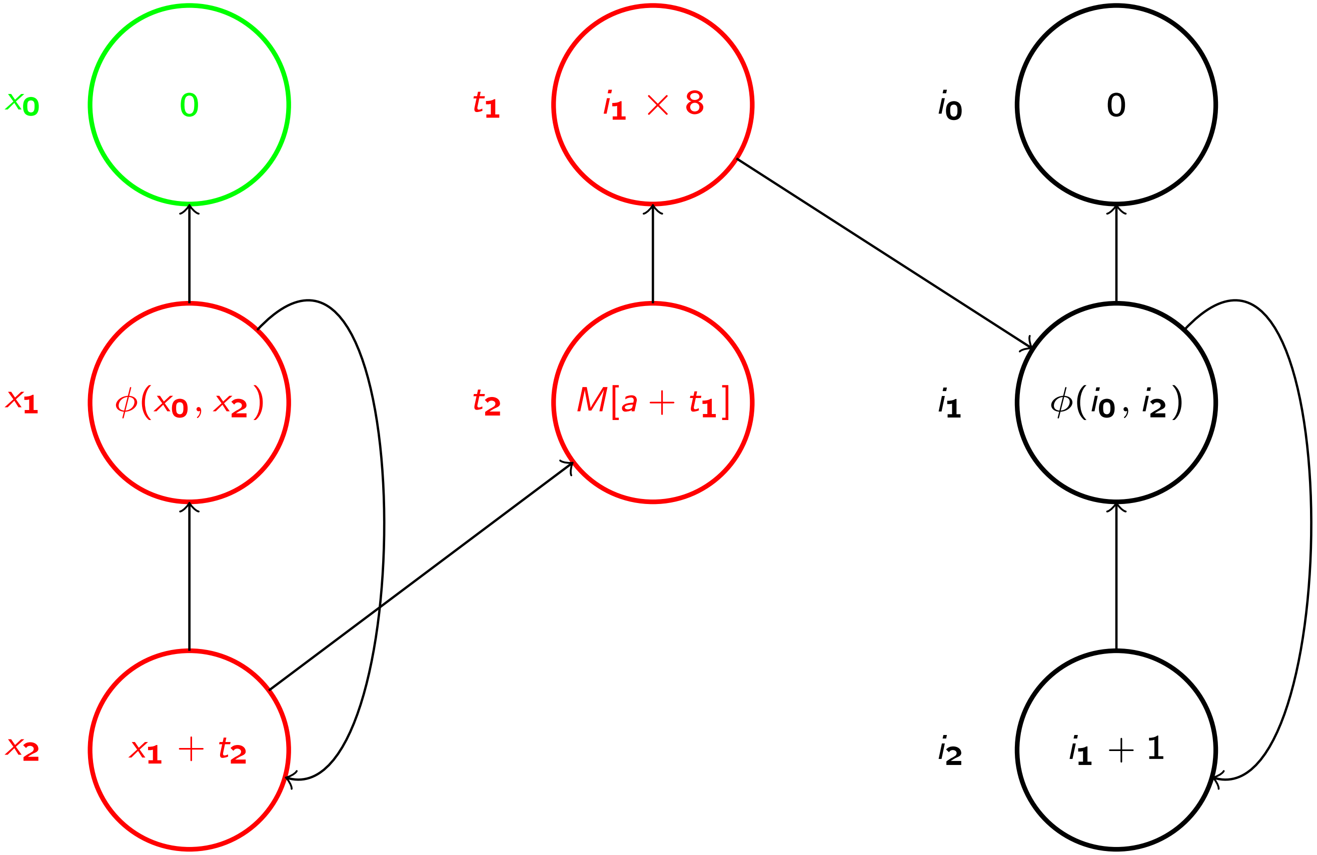
- x_1 and x_2 are pushed and then the search continues with t_2.
- Nodes on the stack are red.
- Next processing t_2.

Processing of t_2



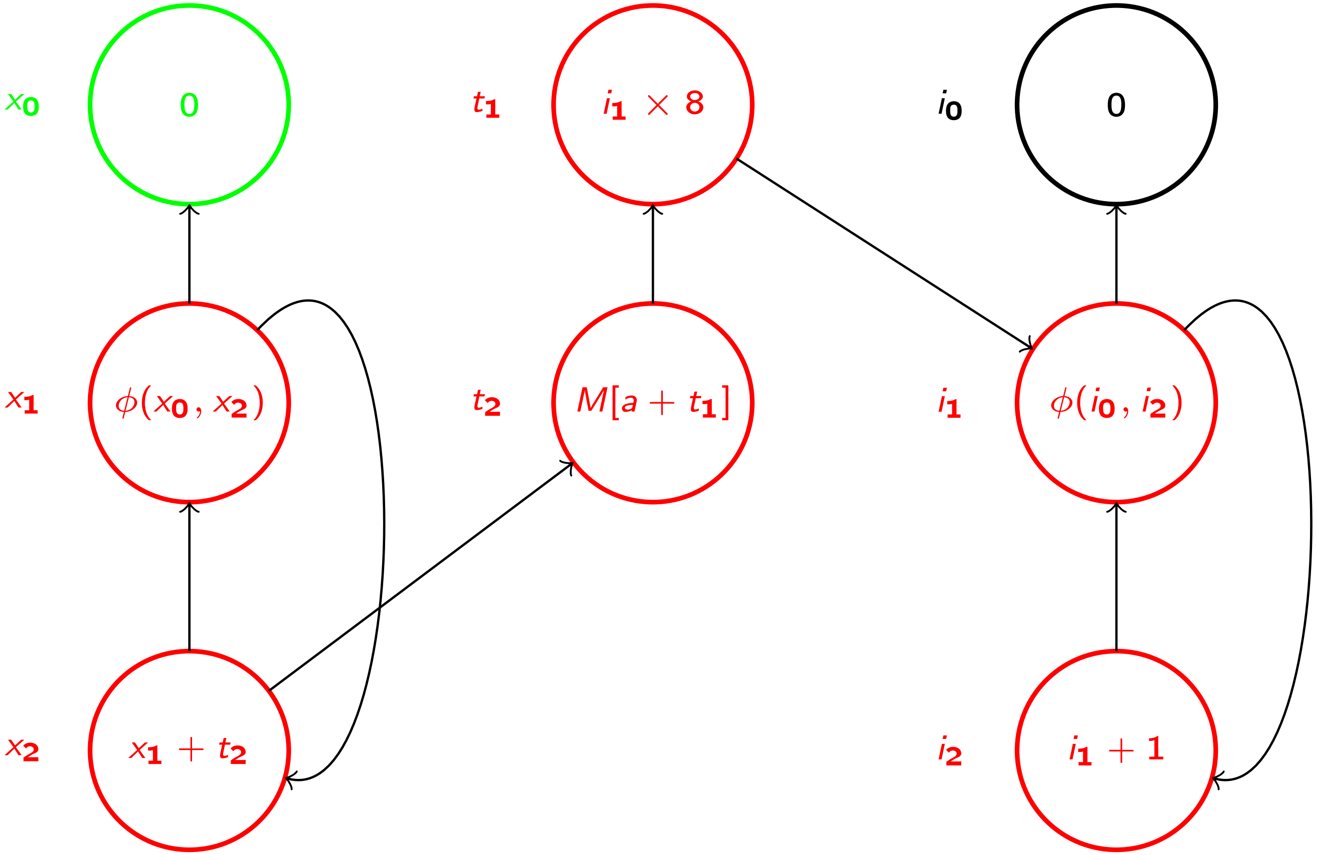
- Next processing t_1.

Processing of t_1



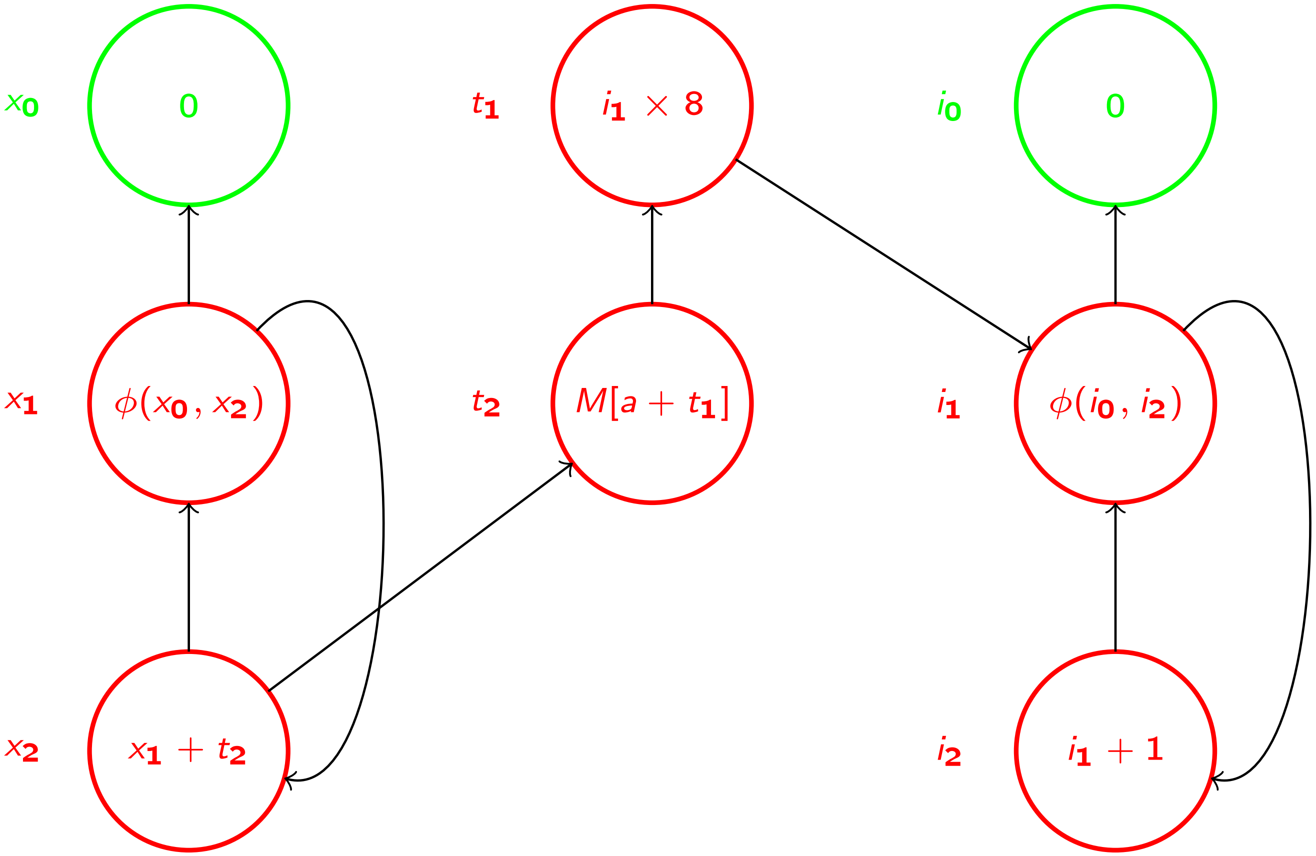
- Next processing i_2.

Processing of i_2 and i_1



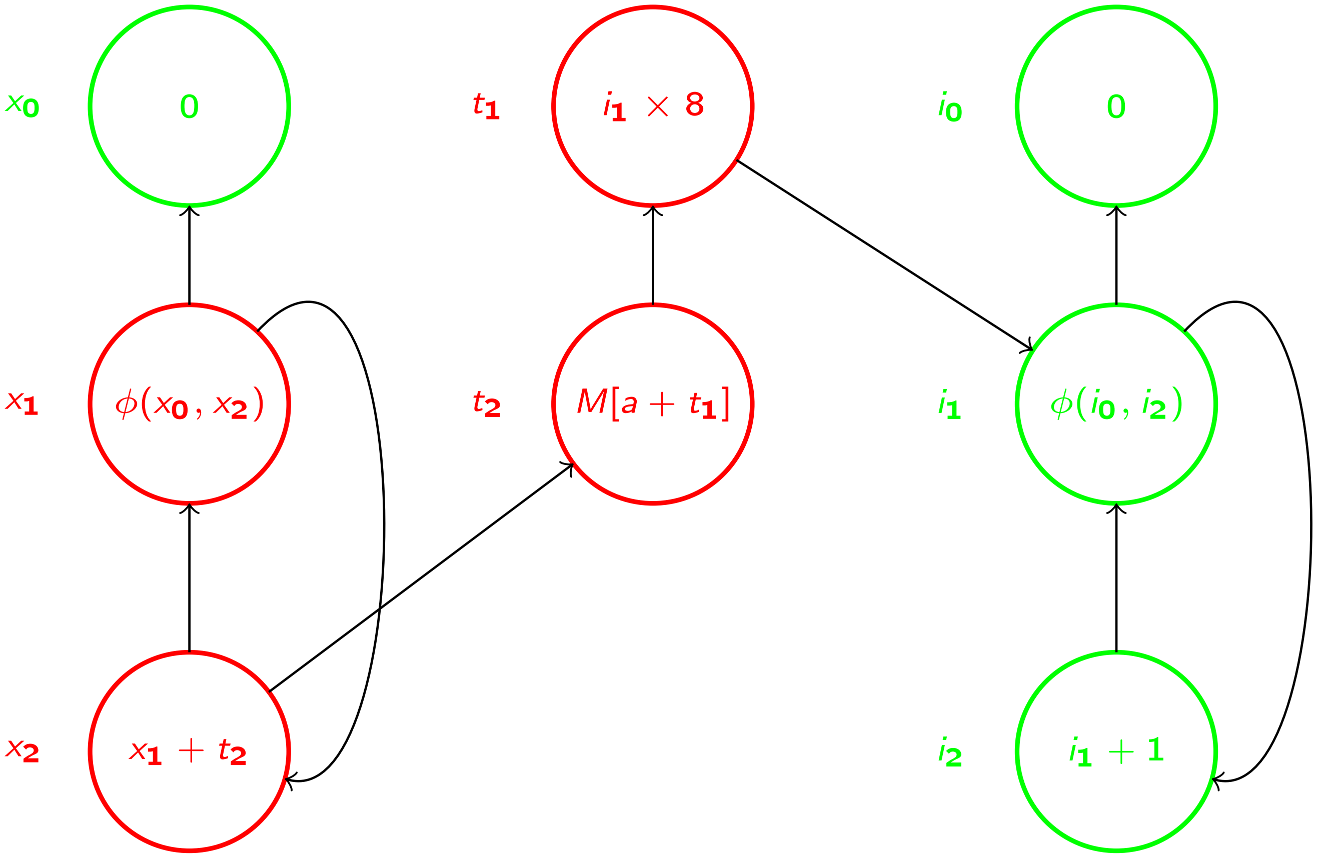
- Next processing i_0.

Processing of i_0



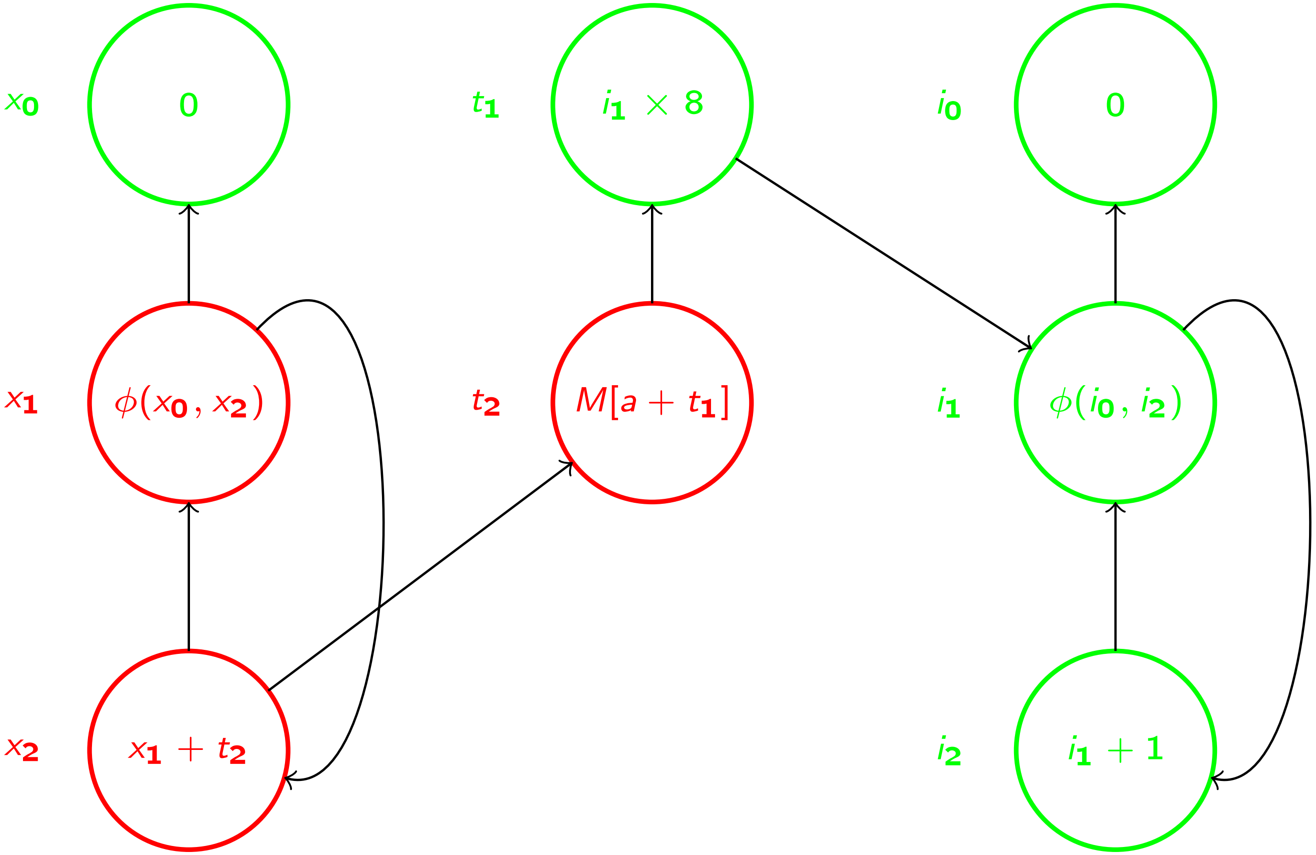
- $SCC_1 = \{i_0\}$
- Next more processing in i_2.

Classifying $SCC_2 = \{i_1, i_2\}$



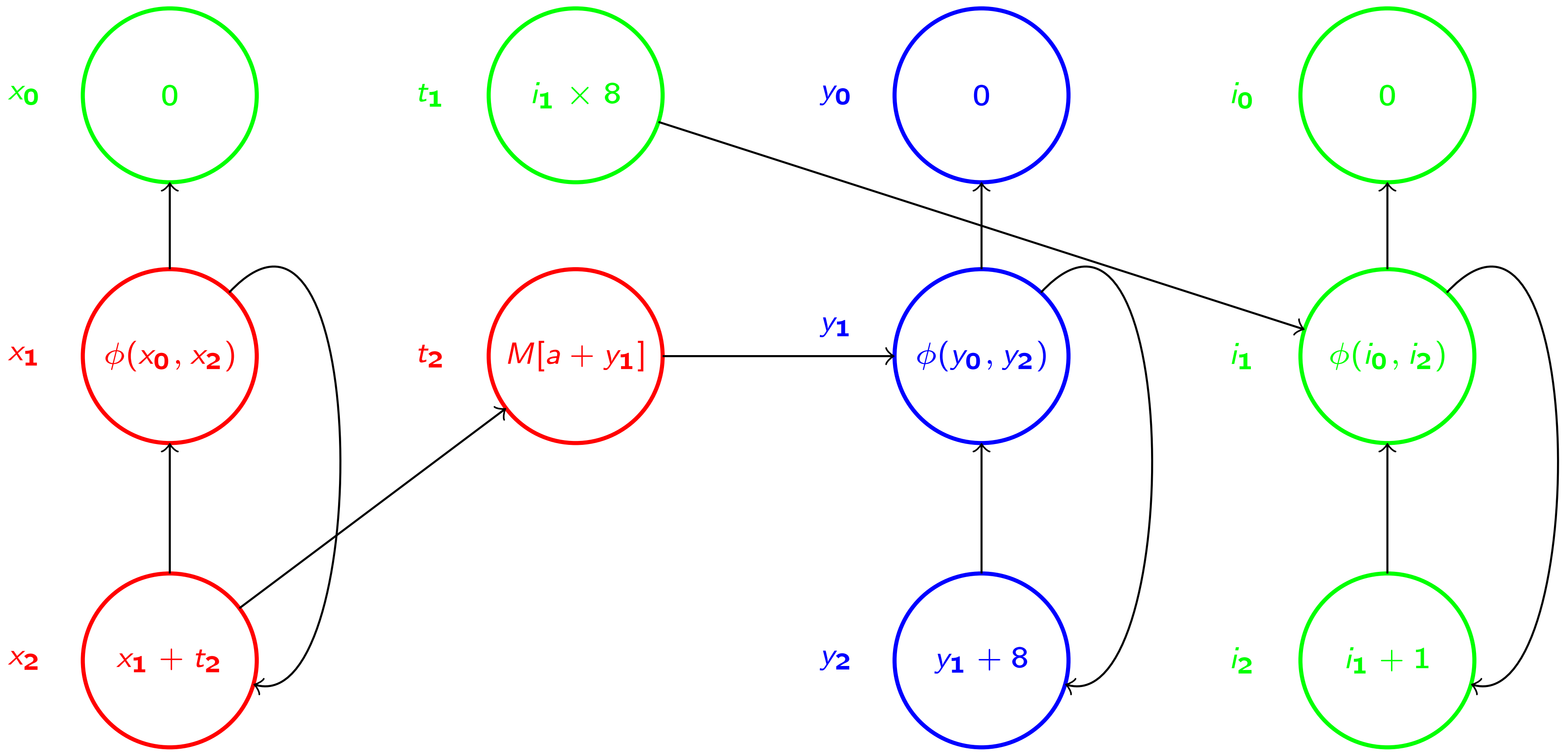
- $SCC_2 = \{i_1, i_2\}$
- SCC_2 is an **induction variable** due it consists of a ϕ-function and an add with a **region constant**.
- A region constant is not modified in a loop, i.e. it's a number or its definition strictly dominates the loop header.

Replacing $i_1 \times 8$



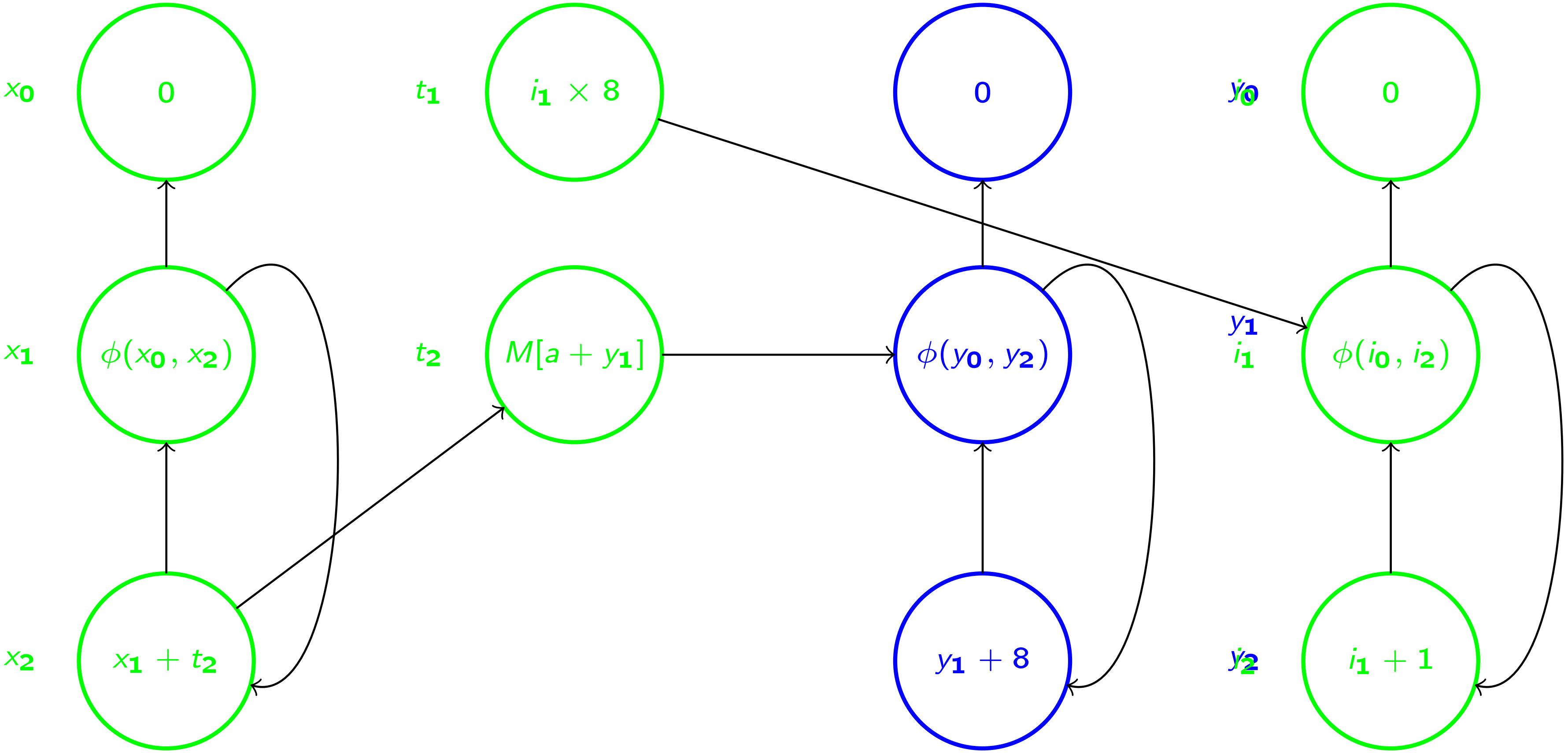
- $SCC_3 = \{t_1\}$
- SCC_3 is a multiplication of an induction variable and a region constant.
- Therefore SCC_3 is replaced by a modified copy of SCC_2 with $\phi(i)$.

Modifying a Copy of SCC_2 to Compute t_1



- $SCC_4 = \{y_1, y_2\}$
- Due to the replacement, the assignment to t_1 becomes dead code.
- There is a very beautiful algorithm to remove t_1 and other dead code that we will look at soon.

Also $a + t_1$ can be Replaced



- Due to Tarjan's algorithm we can start in any node and be sure we have already processed the operand nodes, when a variable's definition is going to be replaced.
- Not only multiplications but also some additions can be replaced, but we don't show this in the example.

Two Simple Forms of Dead Code Elimination

```
#include <stdio.h>

int main(void)
{
        int     a;

        a = 1;
        a = a + 2;

        goto L;

        printf("a = %d\n", a);

L:
        return 0;
}
```

- DFS
- Liveness Analysis

Depth First Search and Dominance Analysis

- DFS from the start vertex visits all basic blocks reachable from the start vertex, obviously.
- All other vertices are removed before performing dominance analysis.
- For some minor modifications of the control flow graph an existing dominator tree can be updated.
- In general, it's easier and probably faster to recompute the dominator tree from scratch, according to some researchers who tried to update the DT.

Limitations of DCE Based on Liveness Analysis

```
for (i = 0; i < n; ++i)
        a = a + i * i;
return;
```

- The variable a is live in the loop but will not affect program output.
- The loop should be deleted but it cannot be using DCE based on liveness.

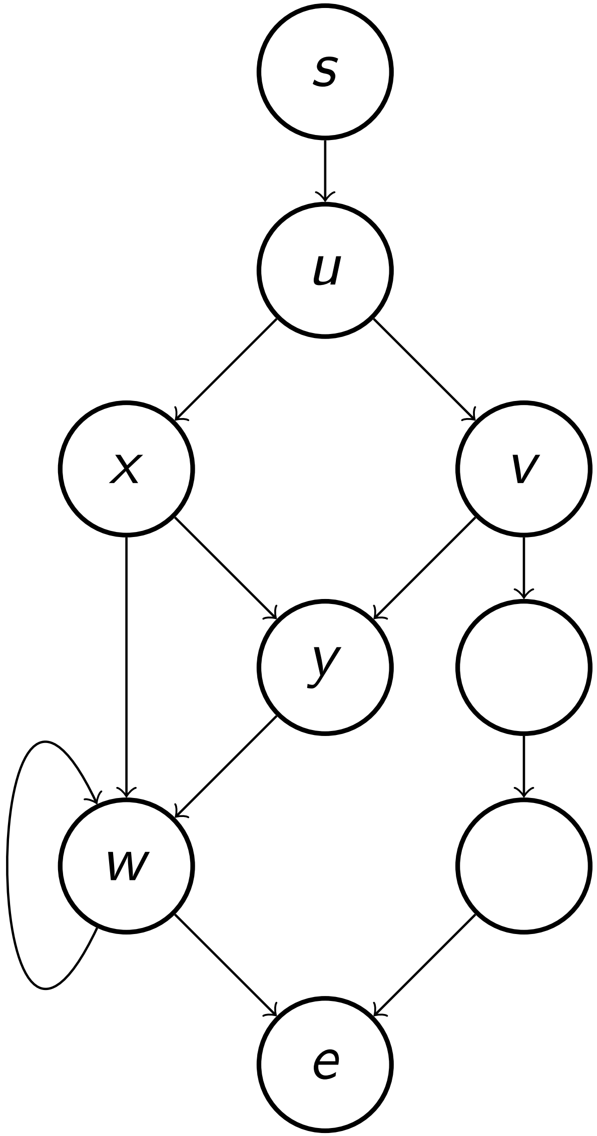
DCE Based on Observable Output

- The correct approach to DCE is to delete all code which cannot affect the observable output.
- In each function, some instructions are marked as **live**, e.g. calls to `printf`, and are put in a worklist.
- Then, recursively, all instructions which provide input to a live instruction is marked as live and put on the worklist.
- Eventually no new instructions are marked as live and all other instructions can be deleted (but read more about branches first!).
- Instructions initially marked live include: function calls, memory writes, and return instructions.
- Why did it take more than 30 years to invent this obvious approach to DCE?

SSA and DCE

- The main reason why it was not invented earlier is that the other approaches usually were sufficient.
- With SSA Form, however, it's more likely there will be lots of instructions, in particular ϕ-functions, which remain after other optimizations.
- For example, operator strength reduction explicitly copies and modifies the strongly connected components in the SSA Graph of induction variables, which can leave a lot of work to DCE.
- The article in ACM Transactions on Programming Languages and Systems (TOPLAS) which presented SSA Form also presented the DCE algorithm we will study.

Control Dependence



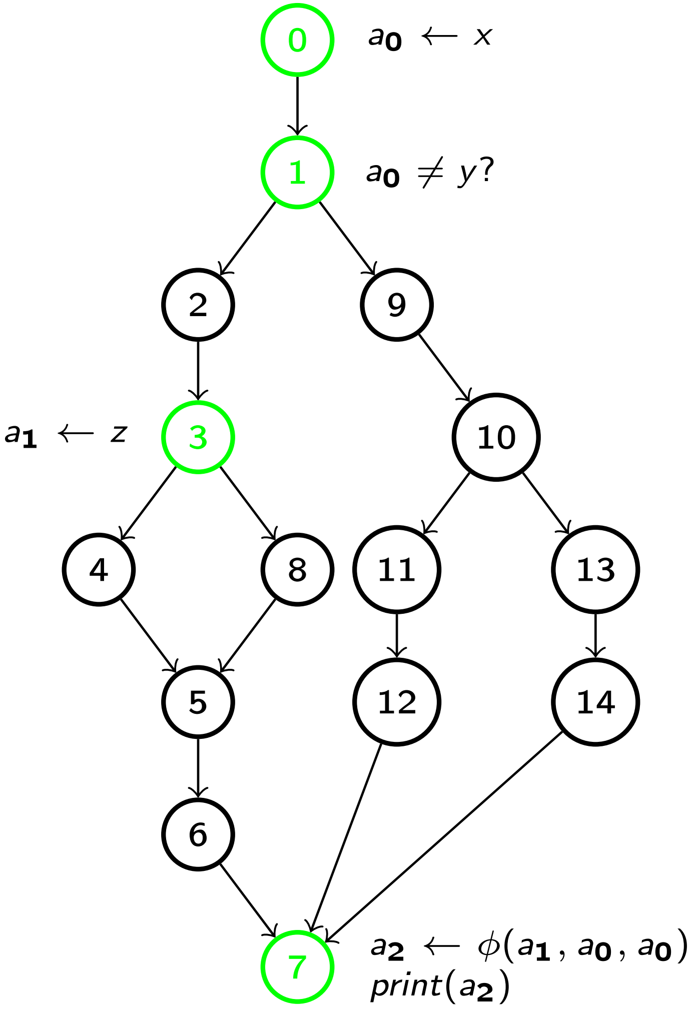
- Assume there is a live instruction in vertex x.
- The DCE algorithm must assure execution actually reaches x exactly as the original program would.
- Therefore some conditional branch instructions (and the instructions providing their input etc) which branch to x must also be marked live.
- In this example the branch in u controls whether x certainly will be executed.
- The vertices that control whether w will be executed are u, v, and w itself.

The DCE Algorithm

```
procedure eliminate_dead_code(G)
    for each statement s do
        if (s is prelive) {
            live(s) ← true
            add s to worklist
        } else
            live(s) ← false
    worklist ← prelive
    while (worklist $\neq \emptyset$) do {
        take s from worklist
        v ← vertex(s)
        live(v) ← true
        for each source operand $\omega$ of s do {
            t ← def($\omega$)
            if (not live(t)) {
                live(t) ← true
                add t to worklist
            }
        }
        for each vertex v $\in CD^{-1}$(vertex(s)) do {
            t ← multiway branch of v
            if (not live(t)) {
                live(t) ← true
                add t to worklist
            }
        }
    }
    for each statement s do
        if (not live(s)  and s $\notin$ {label, branch})
            delete s from vertex(S)
    simplify(G)
end
```

Simplifying the CFG after DCE

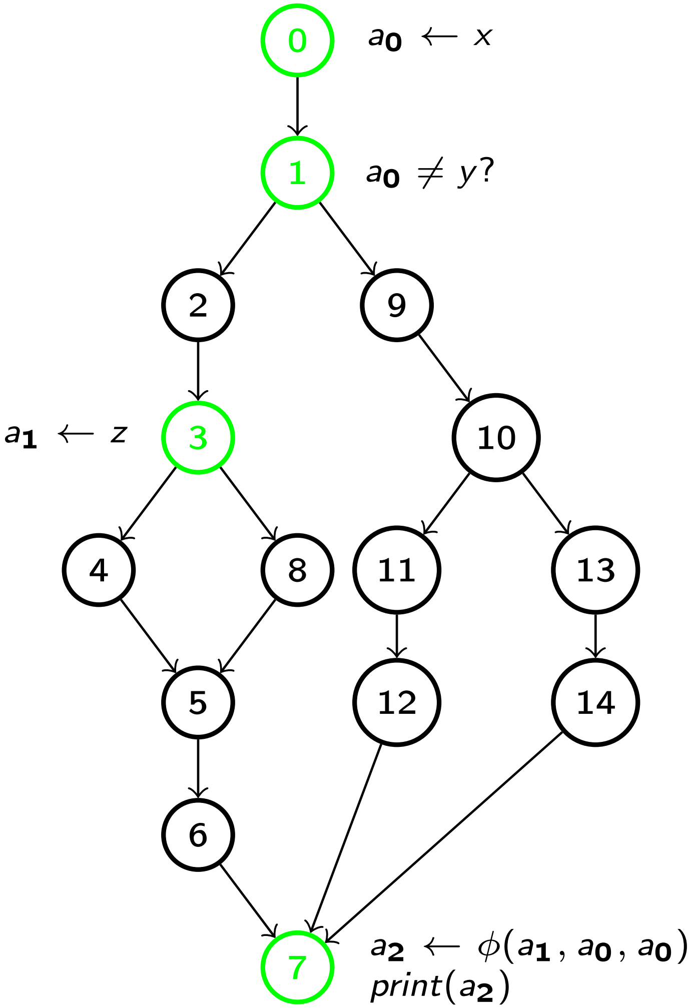
```
procedure simplify(G)
    live(e) ← true
    modified ← false
    for each vertex $u \in G$ do {
        if (not live(u))
            continue
        for each $v \in succ(u)$ do {
            if (live(v))
                continue
            w ← ipdom(v) /* idom in RCFG */
            while (not live(w))
                w ← ipdom(w)
            replace (u, v) with (u, w)
            update the branch in u to its new target w
            update $\phi$-functions in w if necessary
            modified ← true
        }
    }
    if (modified) {
        delete vertices from G which now have become unreachable
        update dominator tree DT
    }
end
```



- Green denotes live vertices

Processing 0

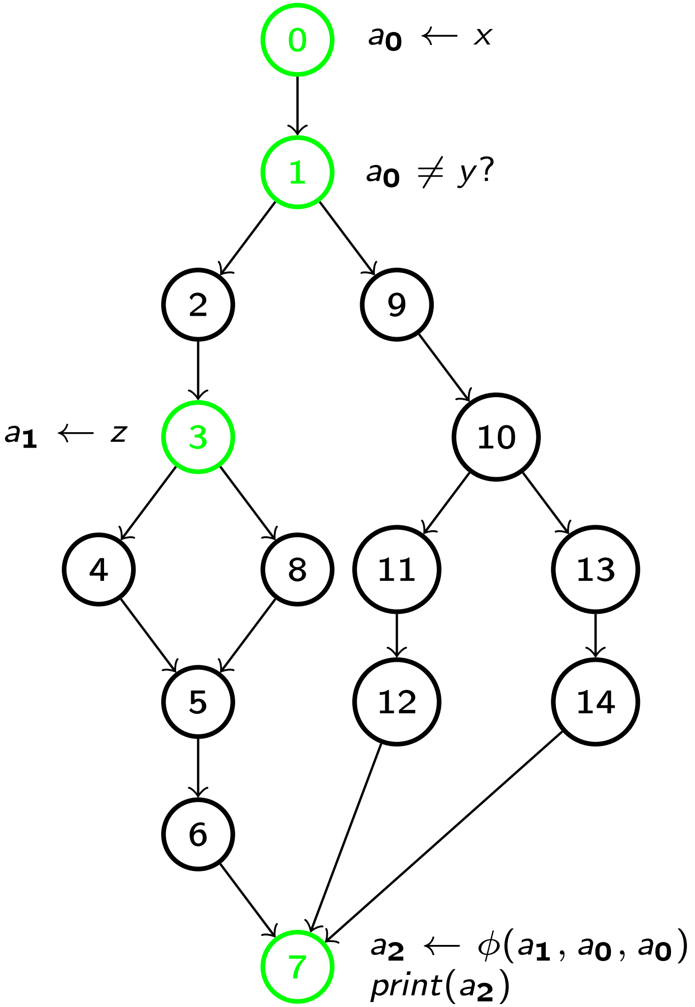
```
procedure simplify(G)
    live(e) ← true
    modified ← false
    for each vertex $u \in G$ do {
        if (not live(u))
            continue
        for each $v \in succ(u)$ do {
            if (live(v))
                continue
            $w \leftarrow ipdom(v)$ /* idom in RCFG */
            while (not live(w))
                $w \leftarrow ipdom(w)$
            replace $(u, v)$ with $(u, w)$
            update the branch in u to its new target w
            update $\phi$-functions in w if necessary
            modified ← true
        }
    }
    if (modified) {
        delete vertices from G which now have become unreachable
        update dominator tree DT
    }
end
```



- Only successor is live.

Processing 1: Edge $(1, 2)$

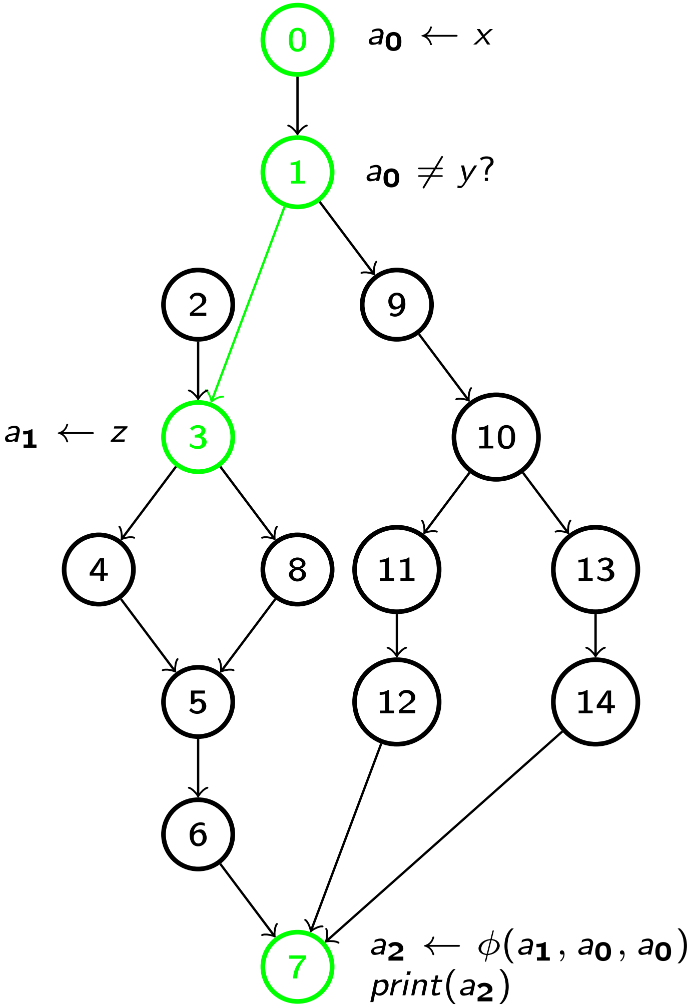
```
procedure simplify(G)
    live(e) ← true
    modified ← false
    for each vertex u ∈ G do {
        if (not live(u))
            continue
        for each v ∈ succ(u) do {
            if (live(v))
                continue
            w ← ipdom(v) /* idom in RCFG */
            while (not live(w))
                w ← ipdom(w)
            replace (u, v) with (u, w)
            update the branch in u to its new target w
            update ϕ-functions in w if necessary
            modified ← true
        }
    }
    if (modified) {
        delete vertices from G which now have become unreachable
        update dominator tree DT
    }
end
```



- 2 is dead. Nearest live is 3.

Processing 1: Edge (1, 2)

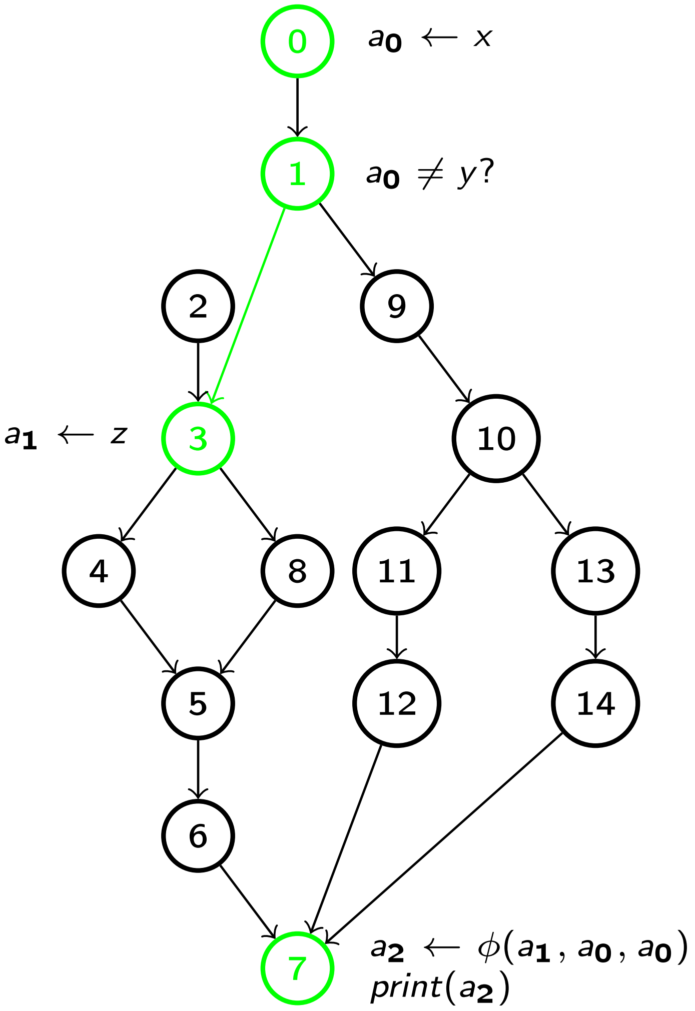
```
procedure simplify(G)
    live(e) ← true
    modified ← false
    for each vertex $u \in G$ do {
        if (not live(u))
            continue
        for each $v \in succ(u)$ do {
            if (live(v))
                continue
            w ← ipdom(v) /* idom in RCFG */
            while (not live(w))
                w ← ipdom(w)
            replace (u, v) with (u, w)
            update the branch in u to its new target w
            update $\phi$-functions in w if necessary
            modified ← true
        }
    }
    if (modified) {
        delete vertices from G which now have become unreachable
        update dominator tree DT
    }
end
```



- 2 is dead. Nearest live is 3.

Processing 1: Edge $(1, 9)$

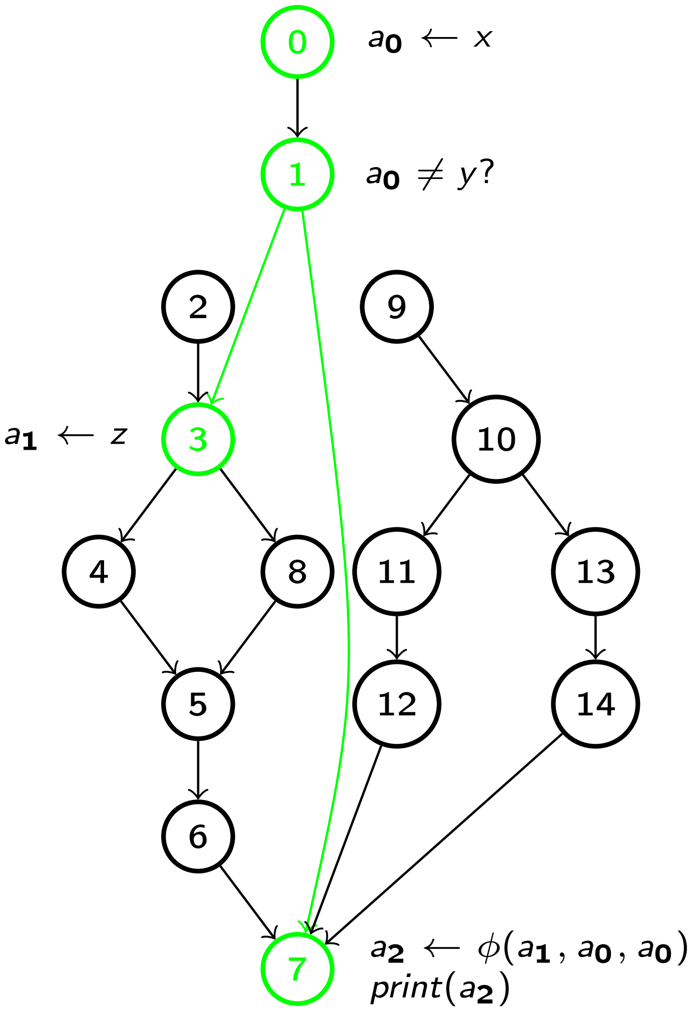
```
procedure simplify (G)
        live (e) ← true
        modified ← false
        for each vertex u ∈ G do {
                if (not live (u))
                        continue
                for each v ∈ succ (u) do {
                        if (live (v))
                                continue
                        w ← ipdom (v) /* idom in RCFG */
                        while (not live (w))
                                w ← ipdom (w)
                        replace (u, v) with (u, w)
                        update the branch in u to its new target w
                        update φ-functions in w if necessary
                        modified ← true
                }
        }
        if (modified) {
                delete vertices from G which now have become unreachable
                update dominator tree DT
        }
end
```



- 9 is dead. Nearest live is 7.

Processing 1: Edge $(1, 9)$

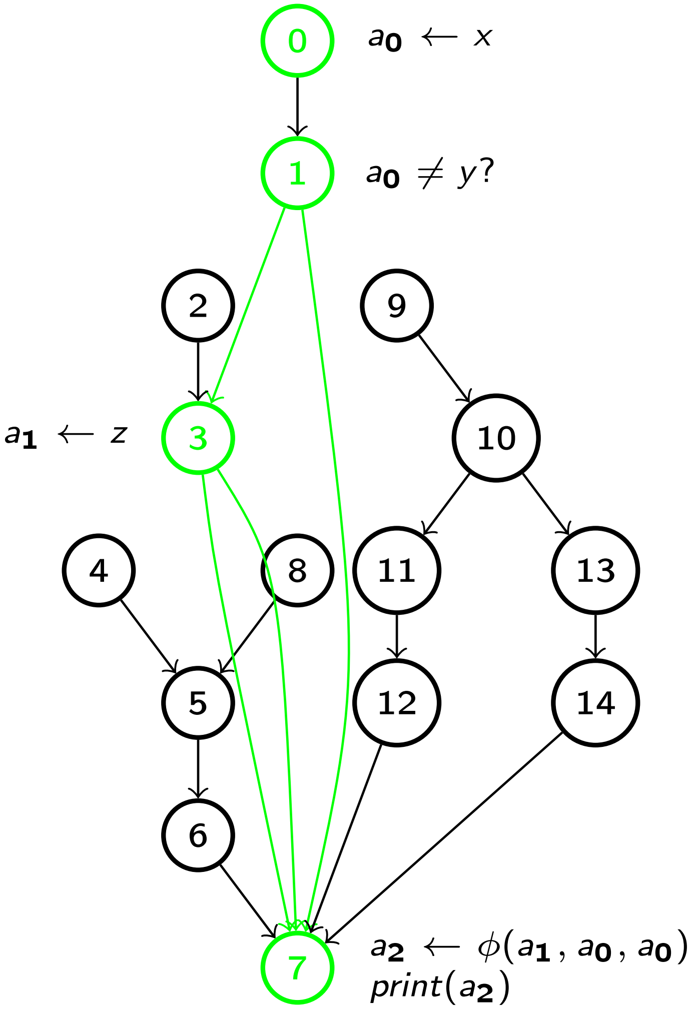
```
procedure simplify(G)
    live(e) ← true
    modified ← false
    for each vertex $u \in G$ do {
        if (not live(u))
            continue
        for each $v \in succ(u)$ do {
            if (live(v))
                continue
            $w \leftarrow ipdom(v)$ /* idom in RCFG */
            while (not live(w))
                $w \leftarrow ipdom(w)$
            replace $(u, v)$ with $(u, w)$
            update the branch in u to its new target w
            update $\phi$-functions in w if necessary
            modified ← true
        }
    }
    if (modified) {
        delete vertices from G which now have become unreachable
        update dominator tree DT
    }
end
```



- Must fix $\phi(a)$ in 7.

Result of Processing 3

```
procedure simplify (G)
    live (e) ← true
    modified ← false
    for each vertex u ∈ G do {
        if (not live (u))
            continue
        for each v ∈ succ (u) do {
            if (live (v))
                continue
            w ← ipdom (v) /* idom in RCFG */
            while (not live (w))
                w ← ipdom (w)
            replace (u, v) with (u, w)
            update the branch in u to its new target w
            update φ-functions in w if necessary
            modified ← true
        }
    }
    if (modified) {
        delete vertices from G which now have become unreachable
        update dominator tree DT
    }
end
```



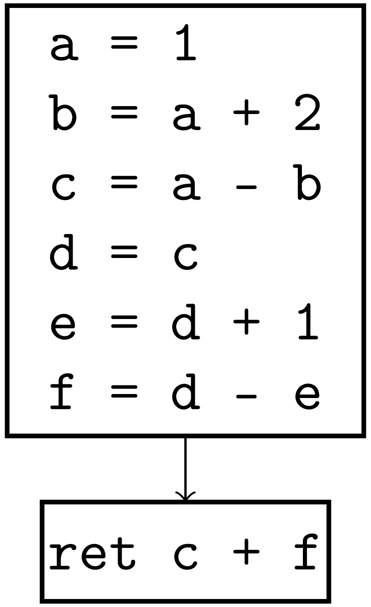
- Later remove one $(3, 7)$!
- Keep only live vertices.

Live Variables Analysis

```
int h(int a, int b)
{
        int     c;

S1:     c = a + b;

S2:     if (c < 0)
                return c * 44;

S3:     a = b - 14;

        return -a;
}
```

- A variable x is **live** at a point p (instruction) if it may be used in the future without being assigned to.
- a is live from the function start and up to and including the add, and then after S_3 and up to and including the negation.
- b is live from the start and up to and including the subtraction.
- c is live from S_1 and up to and including the multiplication.

An Example of Graph Coloring

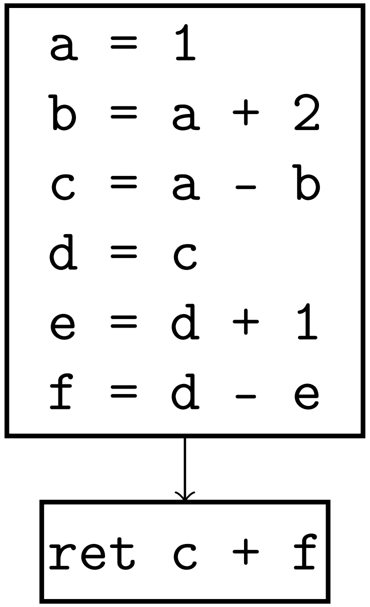


```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

```
ret c + f
```

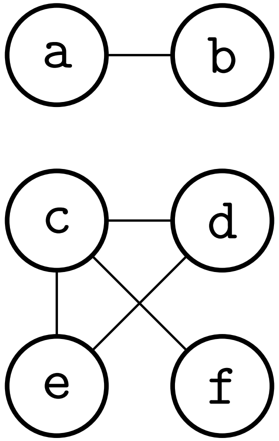
- Which variables cannot use the same register?
- How many registers are needed?

The Interference Graph



```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

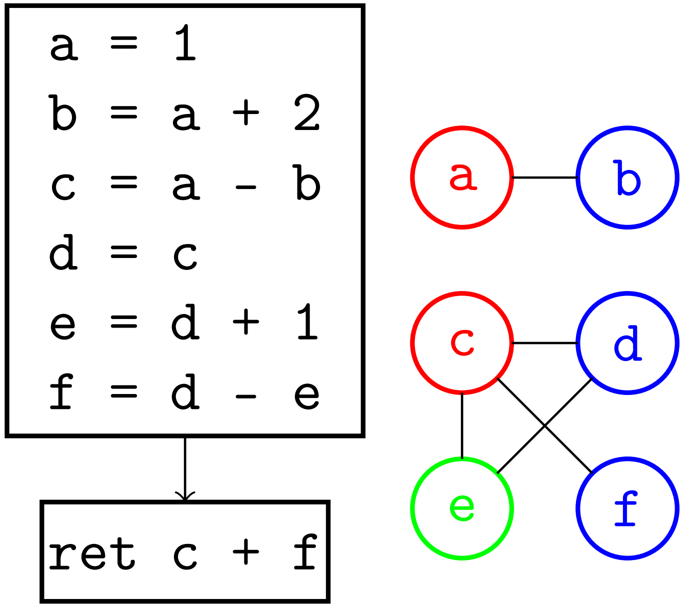
```
ret c + f
```



$$live = use(i) \cup (live - \{def(i)\})$$

1. Initially $live = out = \{c, f\}$.
2. $def(f)$: add edge (c, f).
   $live = \{c, d, e\}$.
3. $def(e)$: add edges $(e, c), (e, d)$.
   $live = \{c, d\}$.
4. $def(d)$: add edge (d, c).
   $live = \{c\}$.
5. $def(c)$: no new edge.
   $live = \{a, b\}$.
6. $def(b)$: add edge (a, b).
   $live = \{a\}$.
7. $def(a)$: no new edge. $live = \emptyset$.

Coloring the Interference Graph

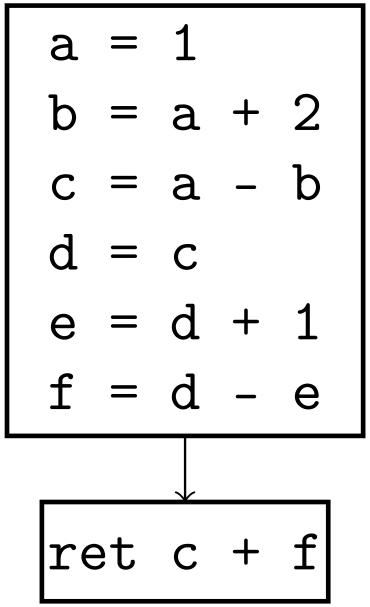


```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

```
ret c + f
```

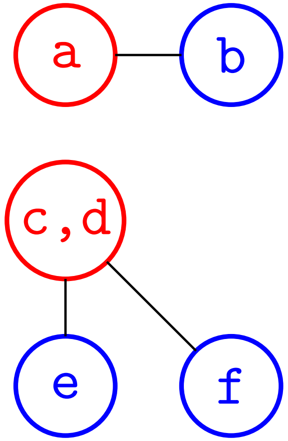
- This interference graph needs three colors.
- Can we use fewer colors?

Register Coalescing



```
a = 1
b = a + 2
c = a - b
d = c
e = d + 1
f = d - e
```

```
ret c + f
```



- *c* and *d* have the same value so they can use the same register!
- It is done using a technique called **register coalescing**.
- Register coalescing is an example of **node merging**.
- Register coalescing needs a minor modification to the construction of the interference graph.

Simplifying the Interference Graph

- Consider an interference graph IG and a number of available colors K.
- Assume the IG can be colored with K colors and there is a node $v \in IG$ with fewer than K neighbors.
- Since v has fewer than K neighbors there must be at least one unused color left for v.
- Therefore we can remove v from the IG without affecting the colorability of IG.
- We remove v from IG and push v on a stack.
- Then we proceed looking for a new node with fewer than K neighbors.
- Assume the original IG was colorable and all its nodes have been pushed on the stack.
- Then each node is popped and re-inserted into IG and given a color which no neighbor has.

Spilling

- The number of neighbors of a node v is denoted its **degree**, or $deg(v)$.
- When there is no node with $deg(v) < K$ a variable is selected for spilling.
- Spilling means that a variable will reside in memory instead of being allocated a register.
- Through spilling the IG eventually will become empty, obviously.
- Heuristics are used to decide which variable (i.e. node) to spill.
- The expected number of memory accesses removed by allocating a variable is calculated, and this count is typically divided by a ”size” of the node.
- By size is meant the number of vertices or instructions that the register would be reserved in for that variable, and hence cannot be used for any other variable.

Rewriting the Program after Spills

```
a = b+c;

...

d = a + c;

-----------

t1 = b + c;
a = t1;

...

t2 = a;
d = t2 + c;
```

- On a RISC machine where operands cannot be in memory a new tiny live range is created at each original memory access of the spilled variable.
- These tiny live ranges should never be spilled.
- The rewriting is done after all nodes have been removed from the interference graph.
- If there was spilling the algorithm is re-executed.
- Eventually it will terminate and three iteration almost always suffice.

Instruction Scheduling Example

- The purpose of **instruction scheduling** is to improve performance by reducing the number of pipeline stalls suffered during execution.
- The following example illustrates the concept, where the right column is the scheduled code.
- Due to instructions only are scheduled within one basic block, only a limited improvement is achieved — the `fsub` and `stf` are not helped at all.

```
ldf  t2,a,t1       ldf  t2,a,t1
ldf  t3,b,t1       ldf  t3,b,t1
fadd t4,t2,t3      ldf  t5,c,t1
ldf  t5,c,t1       ldf  t6,d,t1
ldf  t6,d,t1       fadd t4,t2,t3
fmul t7,t5,t6      fmul t7,t5,t6
fsub t8,t3,t7      fsub t8,t3,t7
stf  t8,e,t1       stf  t8,e,t1
```

Instruction Scheduling vs. Register Allocation

- The goal of instruction scheduling is to reduce pipeline stall and this is achieved by separating the producer and consumer.
- This separation makes it more difficult to perform register allocation.
- **Question:** Which of instruction scheduling and register allocation should be performed first?
  Answer: Instruction scheduling because register allocation would create unnecessary constraints for the scheduler, and advanced instruction scheduling would be seriously limited with already assigned registers.
- If register allocation results in spill code, the instruction scheduler is usually run a second time in order to separate the load instructions from the uses of the loaded register.

Register Pressure of Different Schedules

- The left schedule needs three floating point registers and the right schedule one more.

```
ldf  f2,ra,ri          ldf  f2,ra,ri
ldf  f3,rb,ri          ldf  f3,rb,ri
fadd f2,f2,f3          ldf  f4,rc,ri
ldf  f3,rc,ri          ldf  f5,rd,ri
ldf  f4,rd,ri          fadd f2,f2,f3
fmul f3,f3,f4          fmul f4,f4,f5
fsub f2,f2,f3          fsub f2,f2,f4
stf  f2,re,ri          stf  f2,re,ri
```

Modulo Scheduling

- Consider the following loop and assume there are true dependencies from *A* to *B* and from *B* to *C*.

```
void h()
{
        int     i;

        for (i = 0; i < 100; ++i) {
                A;
                B;
                C;
        }
}
```

- Due to list scheduling only works with one basic block, it cannot improve this loop.
- Such loops are of course extremely common.

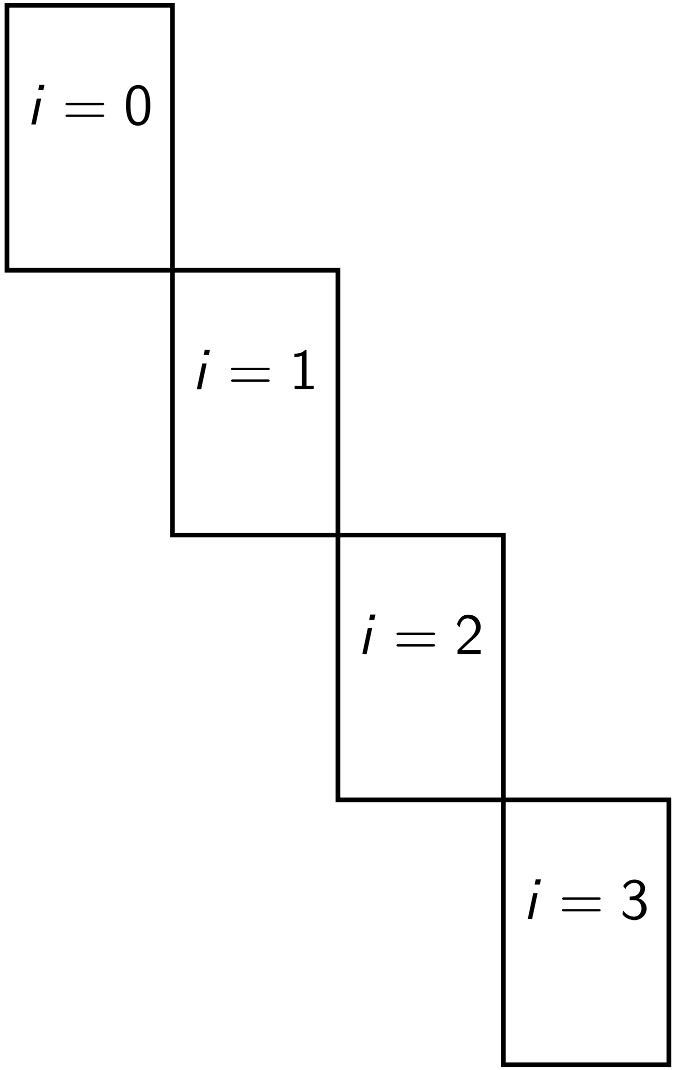
Modulo Scheduling the Loop

- Let us take instructions from three iterations and interleave them.
- First we need to execute instructions from the first two iterations in a prologue.

cycle	i	ii	iii
0	A_0		
1	B_0	A_1	
2	C_0	B_1	A_2
3	A_3	C_1	B_2
4	B_3	A_4	C_2
5	C_3	B_4	A_5
6	A_6	C_4	B_5
7	B_6	A_7	C_5
8	C_6	B_7	
9		C_7	

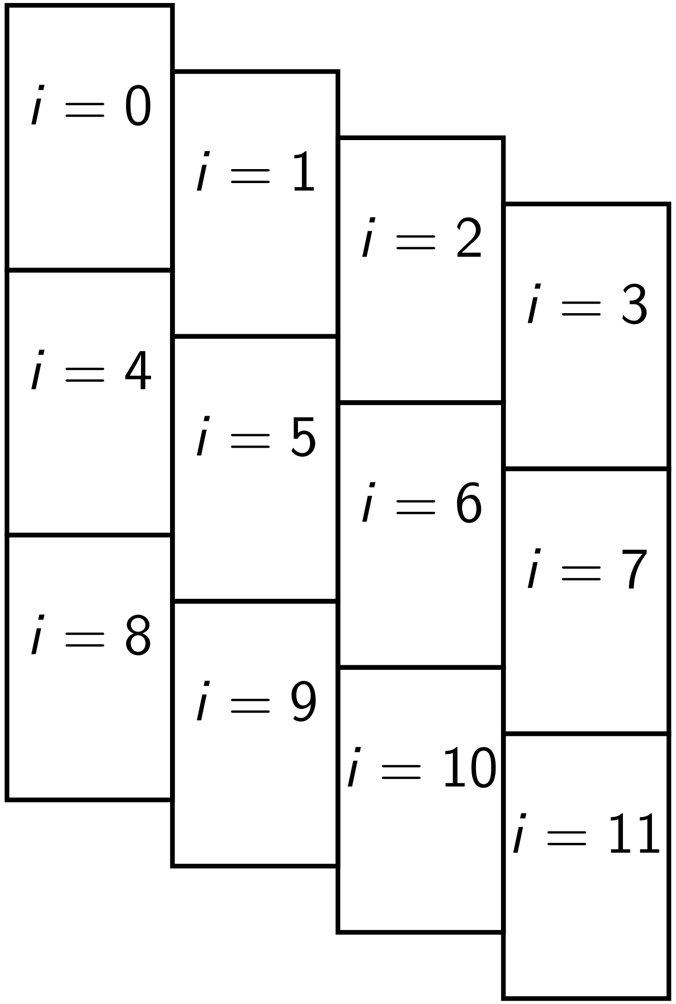
- Assume for illustration only 8 iterations are executed.
- For example A_3 denotes instruction A in iteration 3.
- After a steady-state with 2×3 iterations there is an epilogue.
- Consider instruction B_3. While it waits for A_3, the CPU can also execute C_1 and B_2, assuming a pipelined superscalar CPU.

List Scheduled Execution



- Each iteration is completed before the next starts.
- The height of an iteration is the number of clock cycles it takes.

Parallelism with Modulo Scheduling



- A new iteration is started before the current has completed.
- We wish to start the next iteration as early as possible.
- If we start the next iteration the **same** clock cycle, we need a multicore with one core per loop iteration.

Optimizing Object Oriented Programs

- All normal optimizations are applicable to OOP as well.
- Virtual function calls, i.e. calls through a pointer to an unknown method limits optimization opportunities.
- Therefore, it is important to find calls which must refer to a specific method.
- Sometimes that can be done by only analyzing the type hierarchy, but at other times the assignments must be tracked.
- It is of course not always possible to find which method is called statically.
- There are function pointers in C as well, and they can sometimes be analyzed using symbol table information (number and types of parameters) plus tracking assignments.

EDAN75 Optimizing Compilers in LP3

- If you are interested in optimizing compilers, there is the course EDAN75 in January.
- It is focused on SSA Form and you will start with a subset C compiler which first compiles and then simulates the input C program.
- There you will implement:
  - Lengauer-Tarjan dominance analysis
  - Translation to/from SSA Form
  - Constant propagation on SSA Form
  - Dead code elimination on SSA Form
  - A simple SSA-based optimization in LLVM/Clang 21 (or newer if available)