The Static Single Assignment Form:
Construction and Application to Program Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

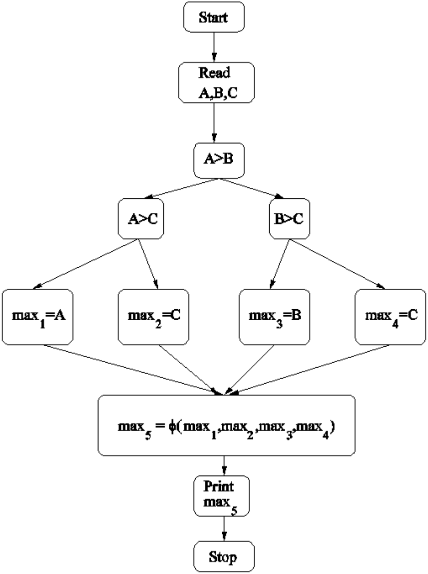
NPTEL Course on Compiler Design

The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
  - If a variable has N uses and M definitions, then *def-use chains* need space and time proportional to $N.M$
  - But, the corresponding instructions of uses and definitions are only $N + M$ in number
  - SSA form, for most realistic programs, is linear in the size of the original program

A Program in non-SSA Form and its SSA Form

read *A*,*B*,*C*
if (*A*>*B*)
    if (*A*>*C*) *max* = *A*
    else *max* = *C*
else if (*B*>*C*) *max* = *B*
        else *max* = *C*
printf (*max*)



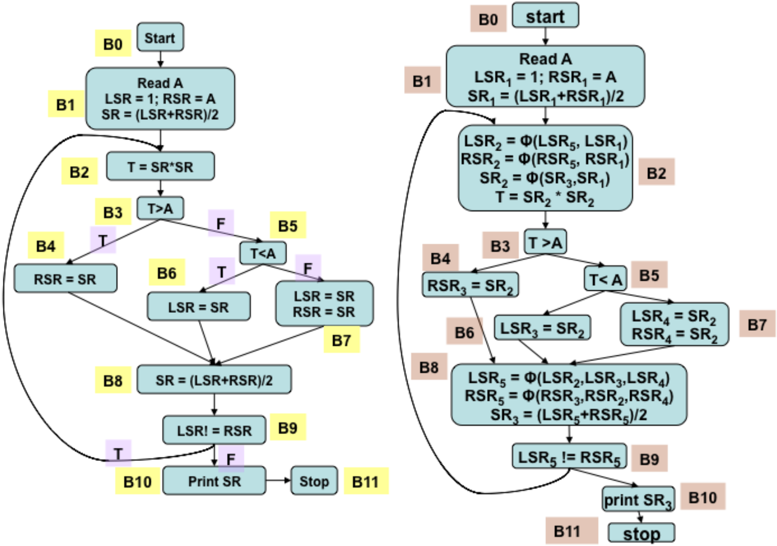
SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, ϕ, is used for selection of values in join nodes
- Not every join node needs a ϕ operator for every variable
- No need for a ϕ operator, if the same definition of the variable reaches the join node along all incoming edges
- Often, an SSA form is augmented with *u-d* and *d-u* chains to facilitate design of faster algorithms
- Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency

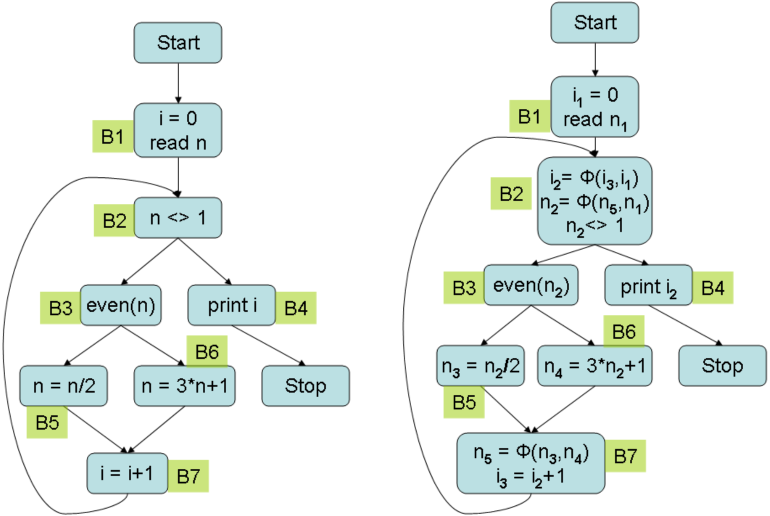
Program 2 in non-SSA Text Form

```
{ Read A; LSR = 1; RSR = A;
  SR = (LSR+RSR)/2;
  Repeat {
       T = SR*SR;
       if (T>A) RSR = SR;
       else if (T<A) LSR = SR;
              else { LSR = SR; RSR = SR}
       SR = (LSR+RSR)/2;
  Until (LSR ≠ RSR);
  Print SR;
}
```

Program 2 in non-SSA and SSA Form



Program 3 in non-SSA and SSA Form



Conditions on the SSA form

After translation, the SSA form should satisfy the following conditions for every variable v in the original program.

1. If two non-null paths from nodes X and Y each having a definition of v converge at a node p, then p contains a trivial ϕ-function of the form $v = \phi(v, v, ..., v)$, with the number of arguments equal to the in-degree of p.
2. Each appearance of v in the original program or a ϕ-function in the new program has been replaced by a new variable v_i, leaving the new program in SSA form.
3. Any use of a variable v along any control path in the original program and the corresponding use of v_i in the new program yield the same value for both v and v_i.

Conditions on SSA Forms

- Condition 1 in the previous slide is recursive.
  - It implies that ϕ-assignments introduced by the translation procedure will also qualify as assignments to *v*
  - This in turn may lead to introduction of more ϕ-assignments at other nodes
- It would be wasteful to place ϕ-functions in all join nodes
- It is possible to locate the nodes where ϕ-functions are *essential*
- This is captured by the *dominance frontier*

The Join Sets and ϕ Nodes

Given $\mathcal{S}$: set of flow graph nodes, the set $JOIN(\mathcal{S})$ is

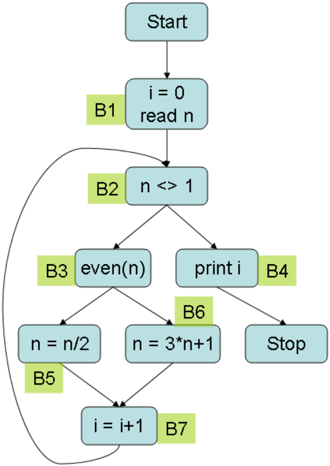
- the set of all nodes n, such that there are at least two non-null paths in the flow graph that start at two distinct nodes in $\mathcal{S}$ and converge at n
  - The paths considered should not have any other common nodes apart from n
- The *iterated join set*, $JOIN^{+}(\mathcal{S})$ is

$$\begin{aligned} JOIN^{(1)}(\mathcal{S}) &= JOIN(\mathcal{S}) \\ JOIN^{(i+1)}(\mathcal{S}) &= JOIN(\mathcal{S} \cup JOIN^{(i)}(\mathcal{S})) \end{aligned}$$

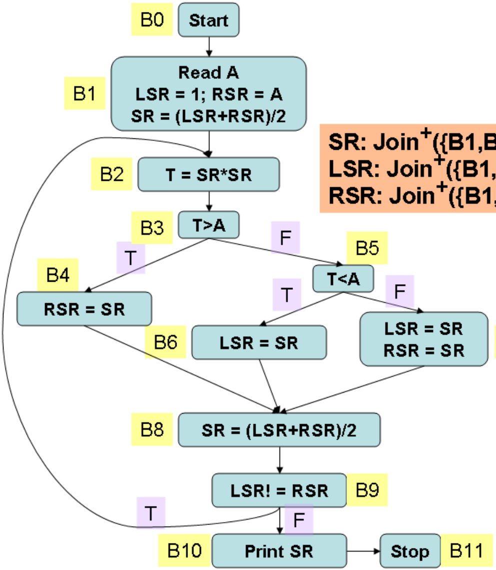
- If $\mathcal{S}$ is the set of assignment nodes for a variable v, then $JOIN^{+}(\mathcal{S})$ is precisely the set of flow graph nodes, where ϕ-functions are needed (for v)
- $JOIN^{+}(\mathcal{S})$ is termed the *dominance frontier*, $DF(\mathcal{S})$, and can be computed efficiently

JOIN Example -1

- variable i: $JOIN^{+}(\{B1, B7\}) = \{B2\}$
- variable n: $JOIN^{+}(\{B1, B5, B6\}) = \{B2, B7\}$



JOIN Example - 2



SR: $\text{Join}^{+}(\{B1,B8\}) = \{B2\}$
LSR: $\text{Join}^{+}(\{B1,B6,B7\}) = \{B2,B8\}$
RSR: $\text{Join}^{+}(\{B1,B4,B7\}) = \{B2,B8\}$

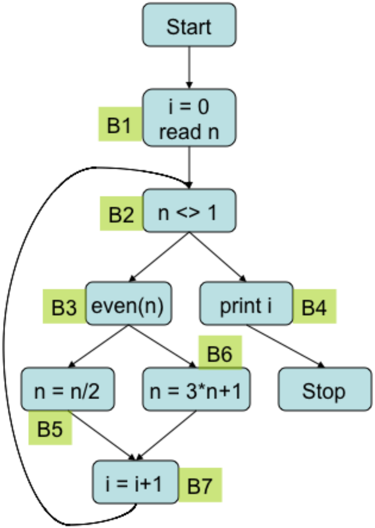
Dominators and Dominance Frontier

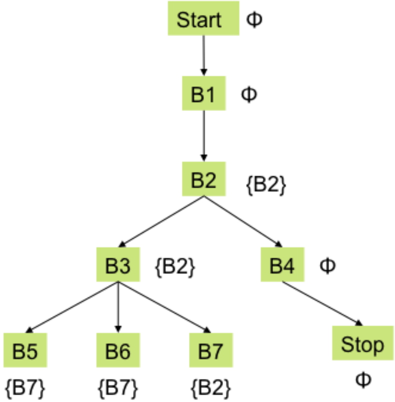
- Given two nodes x and y in a flow graph, x *dominates* y ($x \in dom(y)$) , if x appears in all paths from the *Start* node to y
- The node x *strictly dominates* y, if x dominates y and $x \neq y$
- x is the *immediate dominator* of y (denoted $idom(y)$), if x is the closest strict dominator of y
- A *dominator tree* shows all the immediate dominator relationships
- The *dominance frontier* of a node x, $DF(x)$, is the set of all nodes y such that
  - x dominates a predecessor of y ($p \in preds(y)$ *and* $x \in dom(p)$)
  - but x does not strictly dominate y ($x \notin dom(y) - \{y\}$)

Dominance frontiers - An Intuitive Explanation

- A definition in node *n* forces a ϕ-function in join nodes that lie just outside the region of the flow graph that *n* dominates; hence the name *dominance frontier*
- Informally, $DF(x)$ contains the *first* nodes reachable from *x* that *x* does not dominate, on *each* path leaving *x*
  - In example 1 (next slide), $DF(B1) = \emptyset$, since B1 dominates all nodes in the flow graph except *Start* and B1, and there is no path from B1 to *Start* or B1
  - In the same example, $DF(B2) = \{B2\}$, since B2 dominates all nodes except *Start*, B1, and B2, and there is a path from B2 to B2 (via the back edge)
  - Continuing in the same example, B5, B6, and B7 do not dominate any node and the first reachable nodes are B7, B7, and B2 (respectively). Therefore, $DF(B5) = DF(B6) = \{B7\}$ and $DF(B7) = \{B2\}$
  - In example 2 (second next slide), B5 dominates B6 and B7, but not B8; B8 is the first reachable node from B5 that B5 does not dominate; therefore, $DF(B5) = \{B8\}$

DF Example - 1

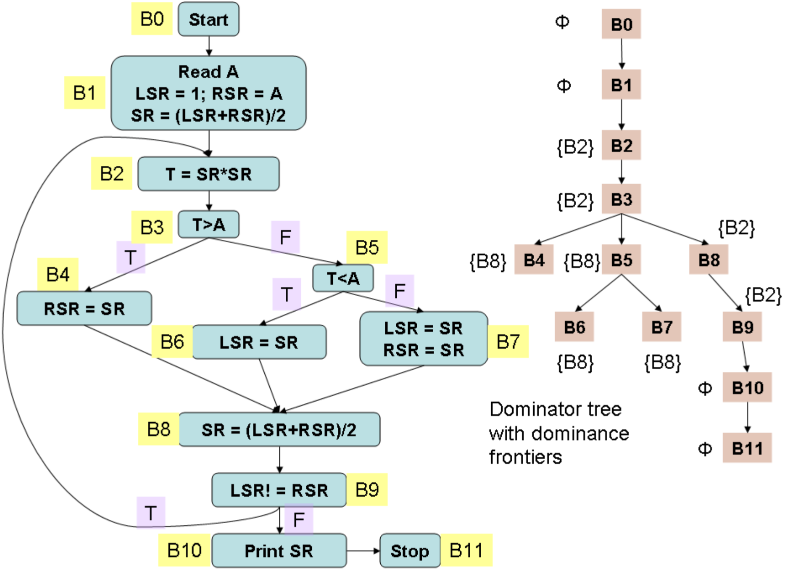




DF(x) is the set of all nodes y such that x dominates a predecessor of y, but x does not strictly dominate y

DF(x) contains the first nodes reachable from x, that x does not dominate

DF Example - 2


B0
Start
B1
Read A
LSR = 1; RSR = A
SR = (LSR+RSR)/2
B2
T = SR*SR
B3
T>A
T
F
B4
RSR = SR
B5
T<A
T
F
B6
LSR = SR
B7
LSR = SR
RSR = SR
B8
SR = (LSR+RSR)/2
B9
LSR! = RSR
T
F
B10
Print SR
Stop
B11
Φ B0
Φ B1
{B2} B2
{B2} B3
{B8} B4
{B8} B5
{B2} B8
B6 {B8}
B7 {B8}
{B2} B9
Φ B10
Φ B11
Dominator tree
with dominance
frontiers

Computation of Dominance Frontiers - 2

1. Identify each join node x in the flow graph
2. For each predecessor, p of x in the flow graph, traverse the dominator tree upwards from p, till $idom(x)$
3. During this traversal, add x to the DF-set of each node met

- In example 1 (second previous slide), consider the join node B2; its predecessors are B1 and B7
  - B1 is also $idom(B2)$ and hence is not considered
  - Starting from B7 in the dominator tree, in the upward traversal till B1 (i.e., $idom(B2)$) B2 is added to the DF sets of B7, B3, and B2
- In example 2 (previous slide), consider the join node B8; its predecessors are B4, B6, and B7
  - Consider B4: B8 is added to $DF(B4)$
  - Consider B6: B8 is added to $DF(B6)$ and $DF(B5)$
  - Consider B7: B8 is added to $DF(B7)$; B8 has already been added to $DF(B5)$
  - All the above traversals stop at B3, which is $idom(B8)$

DF Algorithm

```
{
  for all nodes n in the flow graph do
  DF(n) = ∅;
  for all nodes n in the flow graph do {
  /* It is enough to consider only join nodes */
  /* Other nodes automatically get their DF sets */
  /* computed during this process /*
    for each predecessor p of n in the flow graph do {
        t = p;
        while (t ≠ idom(n)) do {
            DF(t) = DF(t) ∪ {n};
            t = idom(t);
        }
     }
  }
}
```

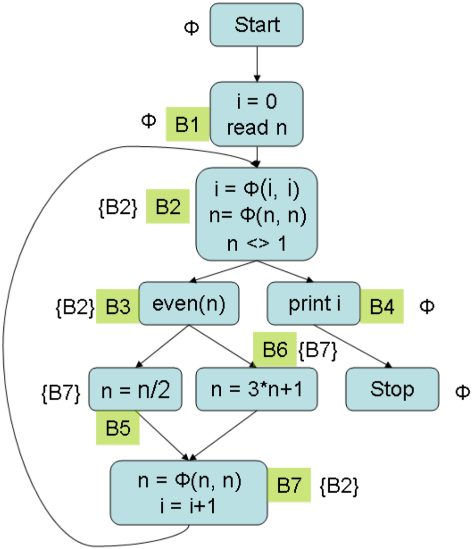
Minimal SSA Form Construction 1

1. Compute *DF* sets for each node of the flow graph
2. For each variable v, place trivial ϕ-functions in the nodes of the flow graph using the algorithm *place-phi-function(v)*
3. Rename variables using the algorithm *Rename-variables(x,B)*

ϕ-Placement Algorithm

- The ϕ-placement algorithm picks the nodes n_i with assignments to a variable
- It places trivial ϕ-functions in all the nodes which are in $DF(n_i)$, for each i
- It uses a work list (i.e., queue) for this purpose

ϕ-function placement Example



Dominance frontier is written beside BB no.

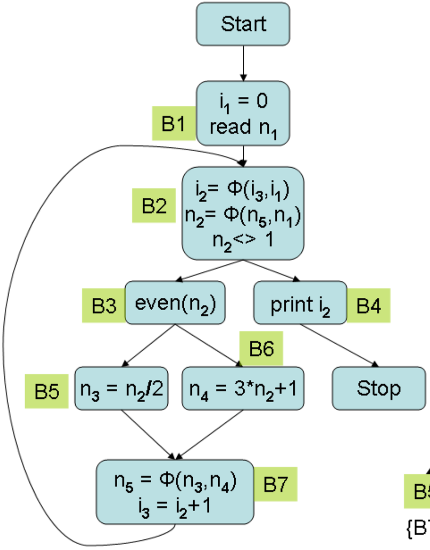
The function *place-phi-function(v)* - 1

```
function Place-phi-function(v) // v is a variable
// This function is executed once for each variable in the flow graph
begin
  // has-phi(B, v) is true if a φ-function has already
  // been placed in B, for the variable v
  // processed(B) is true if B has already been processed once
  // for variable v
  for all nodes B in the flow graph do
     has-phi(B, v) = false; processed(B) = false;
  end for
  W = ∅; // W is the work list
  // Assignment-nodes(v) is the set of nodes containing
  // statements assigning to v
  for all nodes B ∈ Assignment-nodes(v) do
     processed(B) = true; Add(W, B);
  end for
```

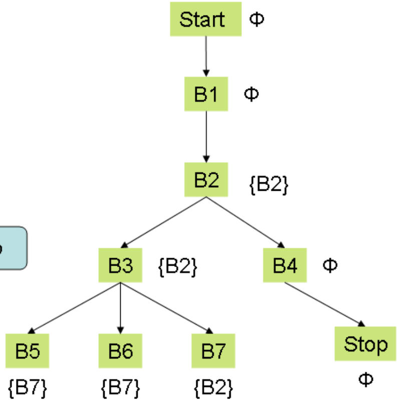
The function *place-phi-function(v)* - 2

```
while W ≠ ∅ do
begin
  B = Remove(W);
  for all nodes y ∈ DF(B) do
    if (not has-phi(y, v)) then
    begin
      place < v = φ(v, v, ..., v) > in y;
      has-phi(y, v) = true;
      if (not processed(y)) then
      begin processed(y) = true;
            Add(W, y);
      end
    end
  end for
end
end
```

SSA Form Construction Example - 1

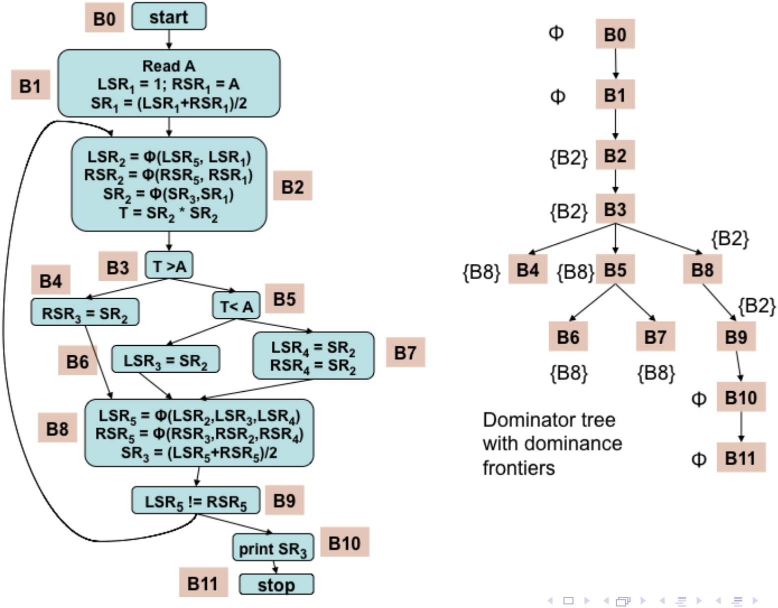


SSA form



Dominator tree with dominance frontier

SSA Form Construction Example - 2



Minimal SSA Form Construction 2

Renaming Algorithm

- The renaming algorithm performs a top-down traversal of the dominator tree
- A separate pair of version stack and version counter are used for each variable
  - The top element of the version stack V is always the version to be used for a variable usage encountered (in the appropriate range, of course)
  - The counter v is used to generate a new version number
- The alogorithm shown later is for a single variable only; a similar algorithm is executed for all variables with an array of version stacks and counters

The Renaming Algorithm

- An SSA form should satisfy the *dominance property*:
  - the definition of a variable dominates each use or
  - when the use is in a ϕ-function, the predecessor of the use
- Therefore, it is apt that the renaming algorithm performs a top-down traversal of the dominator tree
  - Renaming for non-ϕ-statements is carried out while visiting a node *n*
  - Renaming parameters of a ϕ-statement in a node *n* is carried out while visiting the appropriate predecessors of *n*

The function *Rename-variables(x,B)*

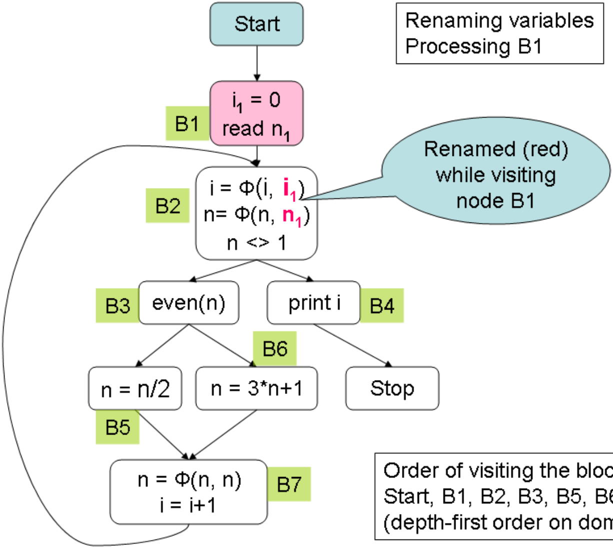
function *Rename-variables*(x, B) // x is a variable and B is a block
begin
  v_e = $Top(V)$; // V is the version stack of x
  // variables are defined before use; hence no renaming can
  // happen on empty stack
  for all statements $s \in B$ do
    if s is a non-ϕ statement then
      replace all uses of x in the $RHS(s)$ with $Top(V)$;
    if s defines x then
    begin
      replace x with x_v in its definition; push x_v onto V;
      // x_v is the renamed version of x in this definition
      $v = v + 1$; // v is the version number counter
    end
  end for

The function *Rename-variables(x,B)*

```
    for all successors s of B in the flow graph do
      j = predecessor index of B with respect to s
      for all φ-functions f in s which define x do
        replace the j^th operand of f with Top(V);
      end for
    end for
    for all children c of B in the dominator tree do
       Rename-variables(x, c);
    end for
    repeat Pop(V); until (Top(V) == v_e);
end
begin // calling program
    for all variables x in the flow graph do
       V = ∅; v = 1; push 0 onto V; // end-of-stack marker
       Rename-variables(x, Start);
    end for
end
```

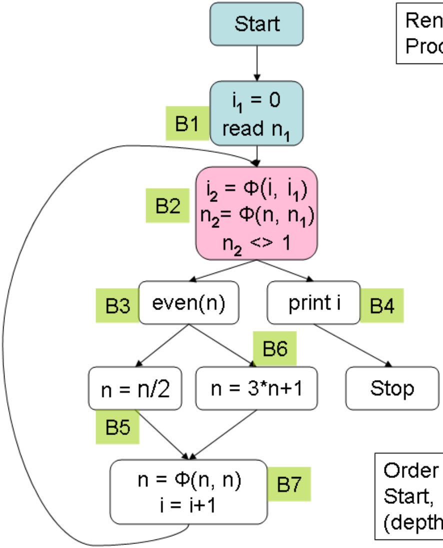
Renaming Variables Example 0.1

Renaming variables
Processing B1



Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

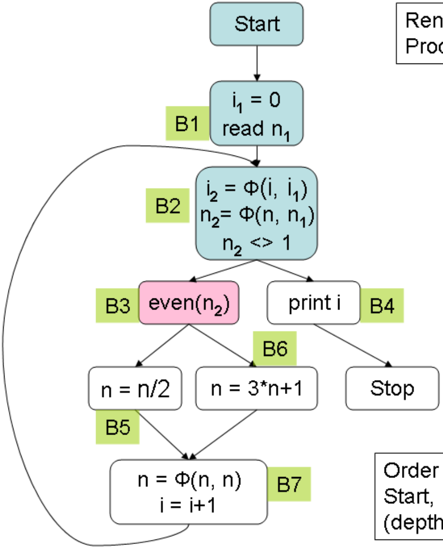
Renaming Variables Example 0.2



Renaming variables
Processing B2

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

Renaming Variables Example 0.3

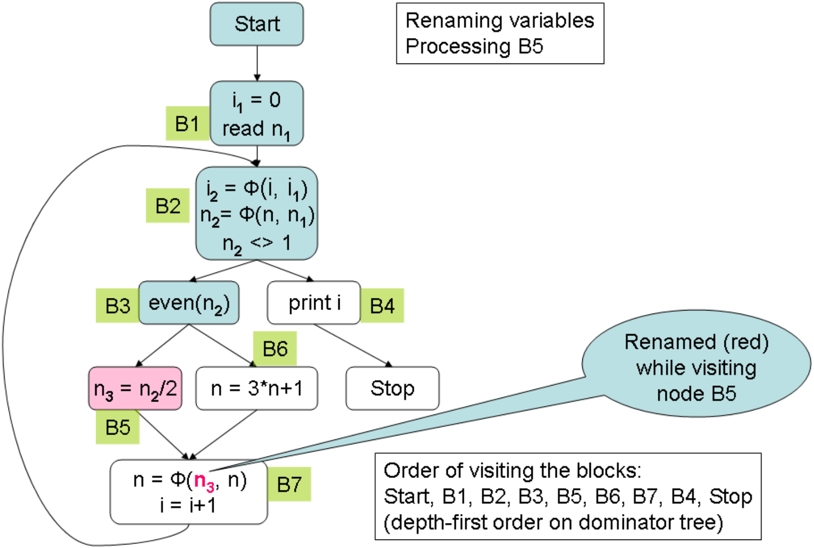


Renaming variables
Processing B3

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

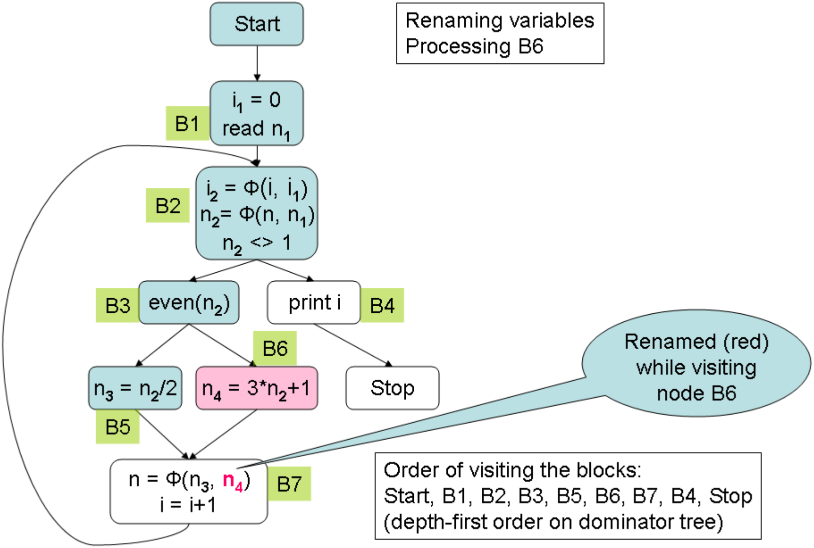
Renaming Variables Example 0.4

Renaming variables
Processing B5



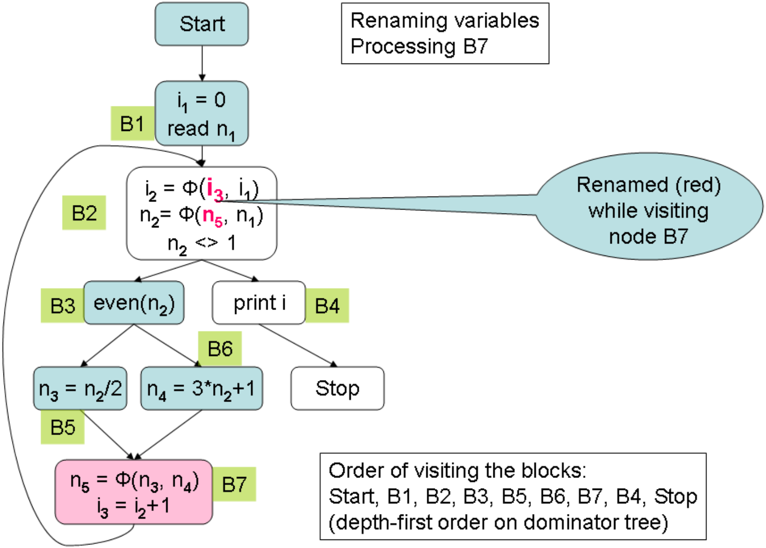
Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

Renaming Variables Example 0.5



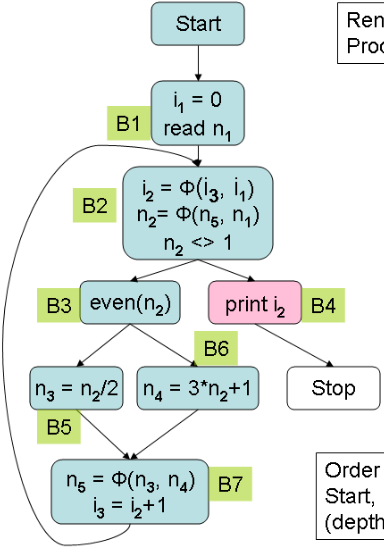
Renaming Variables Example 0.6

Renaming variables
Processing B7



Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

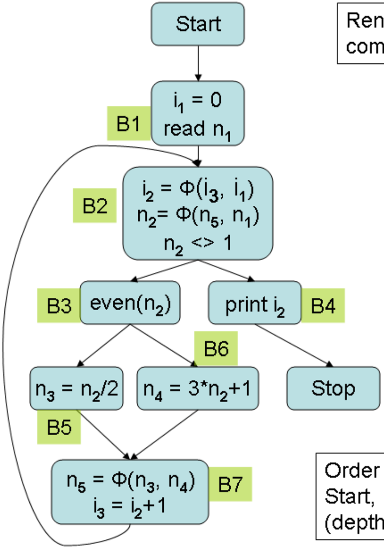
Renaming Variables Example 0.7



Renaming variables
Processing B4

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

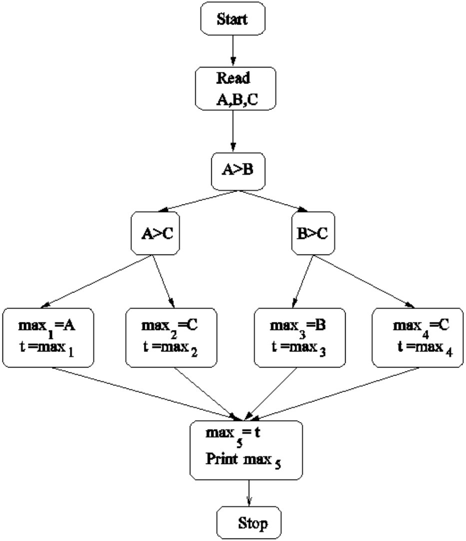
Renaming Variables Example 0.8



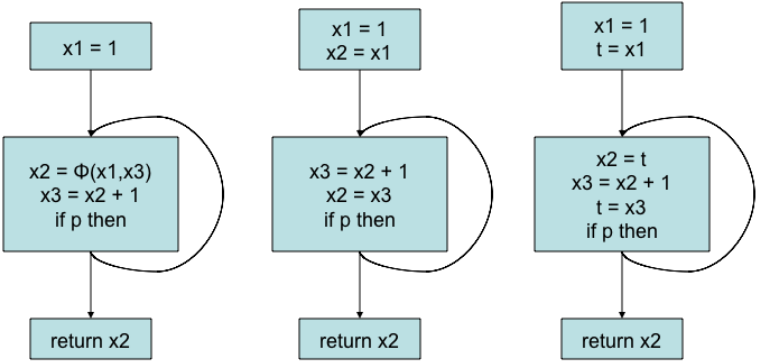
Renaming variables completed

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

Translation to Machine Code - 1



Translation to Machine Code - 2



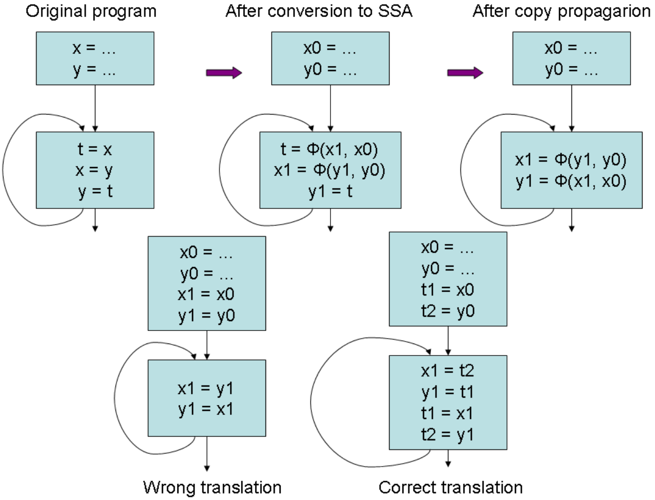
Original program

Wrong translation returned value is incorrect

Correct translation

Translation to Machine Code - 3

The parameters of all ϕ-functions in a basic block are supposed to be read concurrently before any other evaluation begins



Optimization Algorithms with SSA Forms

- Dead-code elimination
  - Very simple, since there is exactly one definition reaching each use
  - Examine the *du-chain* of each variable to see if its use list is empty
  - Remove such variables and their definition statements
  - If a statement such as $x = y + z$ (or $x = \phi(y_1, y_2)$) is deleted, care must be taken to remove the deleted statement from the *du-chains* of y and z (or y_1 and y_2)
- Simple constant propagation
- Copy propagation
- Conditional constant propagation and constant folding
- Global value numbering

Simple Constant Propagation

```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
       S = remove(Stmtpile);
       if S is of the form x = φ(c, c, ..., c) for some constant c
              replace S by x = c
       if S is of the form x = c for some constant c
             delete S from the program
             for all statements T in the du-chain of x do
                   substitute c for x in T; simplify T
                   Stmtpile = Stmtpile ∪ {T}
}
```

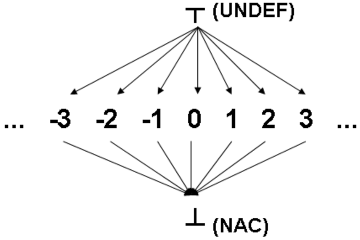
Copy propagation is similar to constant propagation

- A single-argument ϕ-function, $x = \phi(y)$, or a copy statement, $x = y$ can be deleted and y substituted for every use of x

The Constant Propagation Framework - An Overview

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

$any \sqcap UNDEF = any$
$any \sqcap NAC = NAC$
$c_1 \sqcap c_2 = NAC, \text{ if } c_1 \neq c_2$
$c_1 \sqcap c_2 = c_1, \text{ if } c_1 = c_2$



Conditional Constant Propagation - 1

- SSA forms along with extra edges corresponding to *d-u* information are used here
  - Edge from every definition to each of its uses in the SSA form (called henceforth as *SSA edges*)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile* , resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

Conditional Constant Propagation - 2

- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs only one lattice cell per *variable* (globally, not on a per node basis) and two lattice cells per node to store expression values
- Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to *true* or *false*) or both edges (corresponding to $\perp$) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.

// $\mathcal{G} = (\mathcal{N},\mathcal{E}_f,\mathcal{E}_s)$ is the SSA graph,
// with flow edges and SSA edges, and
// $\mathcal{V}$ is the set of variables used in the SSA graph
begin
  $Flowpile = \{(Start \rightarrow n) \mid (Start \rightarrow n) \in \mathcal{E}_f\}$;
  $SSApile = \emptyset$;
  for all $e \in \mathcal{E}_f$ do $e.executable = false$; end for
  //$v.cell$ is the lattice cell associated with the variable v
  for all $v \in \mathcal{V}$ do $v.cell = \top$; end for
  // $y.oldval$ and $y.newval$ store the lattice values
  // of expressions at node y
  for all $y \in \mathcal{N}$ do
    $y.oldval = \top$; $y.newval = \top$;
  end for

CCP Algorithm - Contd.

```
while (Flowpile ≠ ∅) or (SSApile ≠ ∅) do
begin
  if (Flowpile ≠ ∅) then
  begin
    (x, y) = remove(Flowpile);
    if (not (x, y).executable) then
    begin
      (x, y).executable = true;
      if (φ-present(y)) then visit-φ(y)
        else if (first-time-visit(y)) then visit-expr(y);
      // visit-expr is called on y only on the first visit
      // to y through a flow edge; subsequently, it is called
      // on y on visits through SSA edges only
      if (flow-outdegree(y) == 1) then
        // Only one successor flow edge for y
        Flowpile = Flowpile ∪ {(y, z) | (y, z) ∈ ℰ_f};
    end
```

```
        // if the edge is already marked, then do nothing
    end
    if (SSApile ≠ ∅) then
      begin
        (x, y) = remove(SSApile);
        if (φ-present(y)) then visit-φ(y)
          else if (already-visited(y)) then visit-expr(y);
          // A false returned by already-visited implies
          // that y is not yet reachable through flow edges
      end
    end // Both piles are empty
end
function φ-present(y) // y ∈ N
begin
  if y is a φ-node then return true
    else return false
end
```

CCP Algorithm - Contd.

function *visit*-ϕ(y) // $y \in \mathcal{N}$
begin
  $y.newval$ = $\top$; //$\|y.instruction.inputs\|$ is the number of
  // parameters of the ϕ-instruction at node y
  for i = 1 to $\|y.instruction.inputs\|$ do
    Let p_i be the i^{th} predecessor of y ;
    if $((p_i, y).executable)$ then
    begin
      Let a_i = $y.instruction.inputs[i]$;
      // a_i is the i^{th} input and $a_i.cell$ is the lattice cell
      // associated with that variable
      $y.newval = y.newval \sqcap a_i.cell$;
    end
  end for

CCP Algorithm - Contd.

```
    if (y.newval < y.instruction.output.cell) then
    begin
      y.instruction.output.cell = y.newval;
      SSApile = SSApile ∪ {(y, z) | (y, z) ∈ E_s };
    end
end

function already-visited(y) // y ∈ N
// This function is called when processing an SSA edge
begin // Check in-coming flow graph edges of y
  for all e ∈ {(x, y) | (x, y)∈ E_f}
    if e.executable is true for at least one edge e
      then return true else return false
  end for
end
```

CCP Algorithm - Contd.

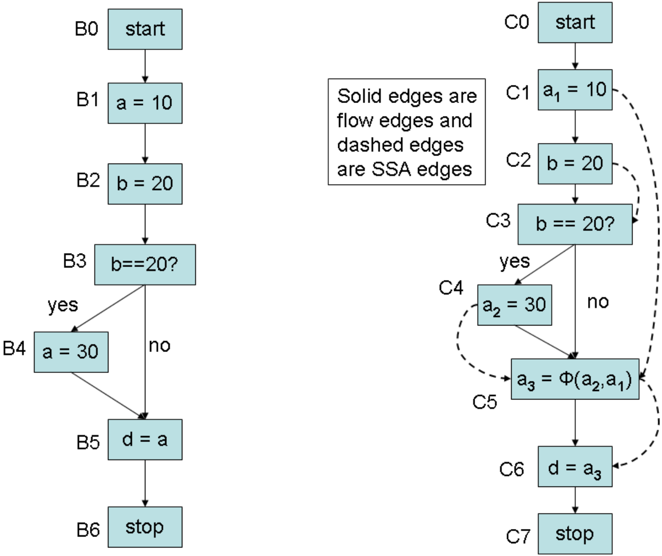
function *first-time-visit*(y) // $y \in \mathcal{N}$
// This function is called when processing a flow graph edge
begin // Check in-coming flow graph edges of y
  for all $e \in \{(x, y) \mid (x, y) \in \mathcal{E}_f\}$
    if $e.executable$ is true for more than one edge e
      then return *false* else return *true*
  end for
  // At least one in-coming edge will have *executable* true
  // because the edge through which node y is entered is
  // marked as *executable* before calling this function
end

CCP Algorithm - Contd.

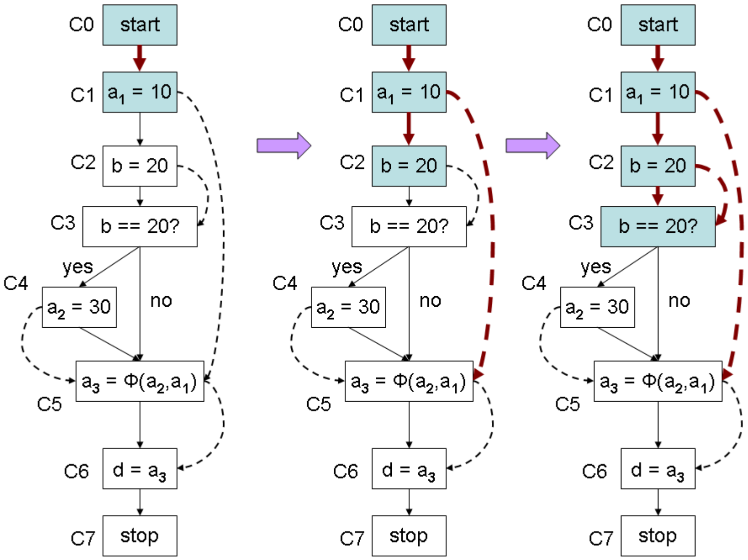
function *visit-expr*(y) // $y \in \mathcal{N}$
begin
  Let $input_1 = y.instruction.inputs[1]$;
  Let $input_2 = y.instruction.inputs[2]$;
  if ($input_1.cell == \perp$ or $input_2.cell == \perp$) then
    $y.newval = \perp$
  else if ($input_1.cell == \top$ or $input_2.cell == \top$) then
      $y.newval = \top$
    else // evaluate expression at y as per lattice evaluation rules
      $y.newval = evaluate(y)$;
      // It is easy to handle instructions with one operand
  if y is an assignment node then
    if ($y.newval < y.instruction.output.cell$) then
    begin
      $y.instruction.output.cell = y.newval$;
      $SSApile = SSApile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_s\}$;
    end

```
else if y is a branch node then
  begin
    if (y.newval < y.oldval) then
    begin
      y.oldval = y.newval;
      switch(y.newval)
        case ⊥: // Both true and false branches are equally likely
          Flowpile = Flowpile ∪ {(y, z) | (y, z) ∈ ℰ_f };
        case true: Flowpile = Flowpile ∪ {(y, z) | (y, z) ∈ ℰ_f and
                                   (y, z) is the true branch edge at y };
        case false: Flowpile = Flowpile ∪ {(y, z) | (y, z) ∈ ℰ_f and
                                   (y, z) is the false branch edge at y };
      end switch
    end
  end
end
```

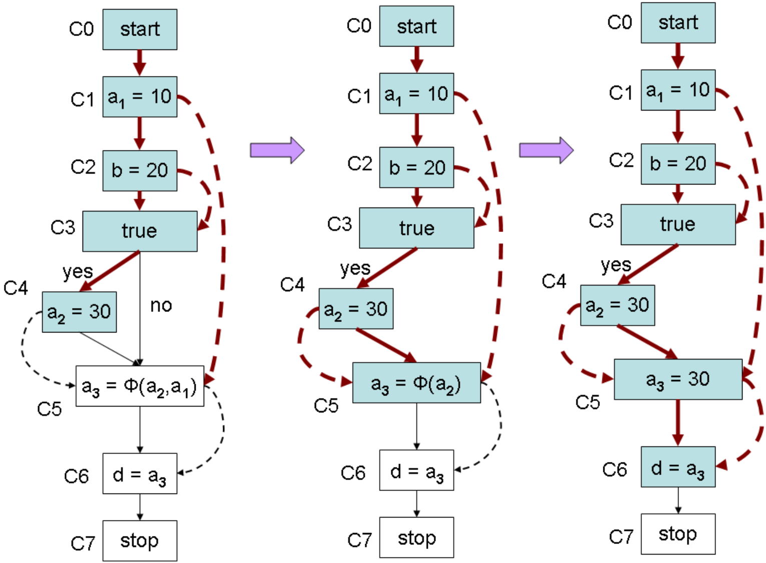
CCP Algorithm - Example - 1



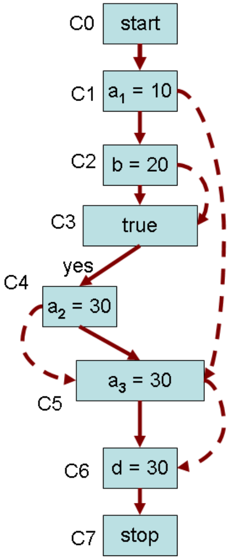
CCP Algorithm - Example 1 - Trace 1



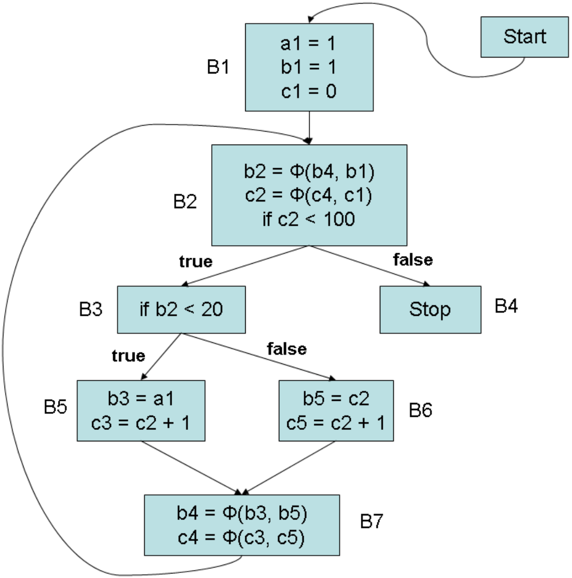
CCP Algorithm - Example 1 - Trace 2



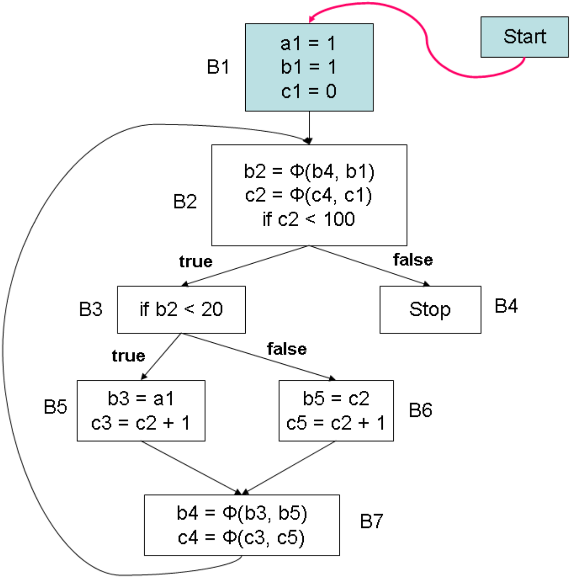
CCP Algorithm - Example 1 - Trace 3



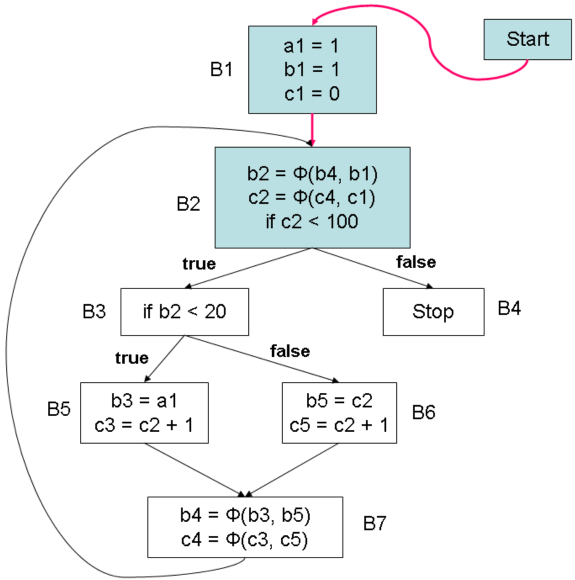
CCP Algorithm - Example 2



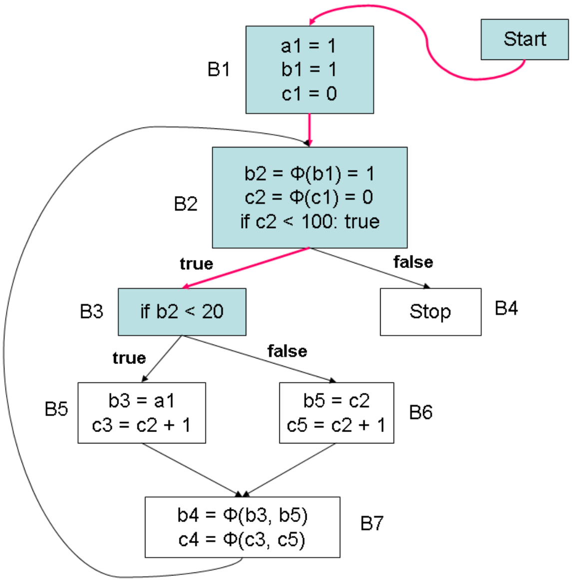
CCP Algorithm - Example 2 - Trace 1



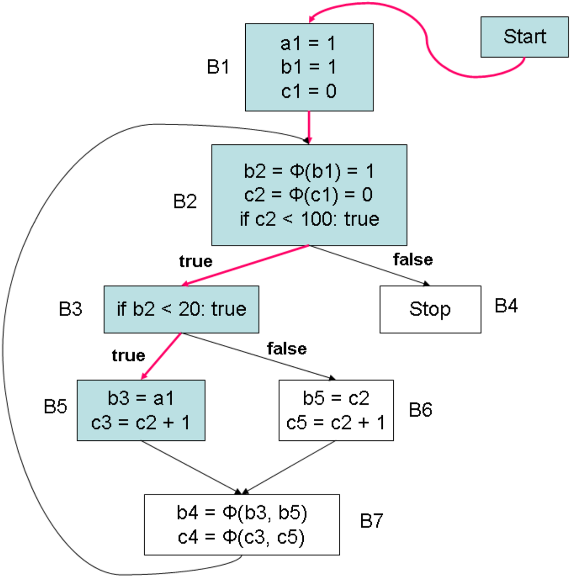
CCP Algorithm - Example 2 - Trace 2



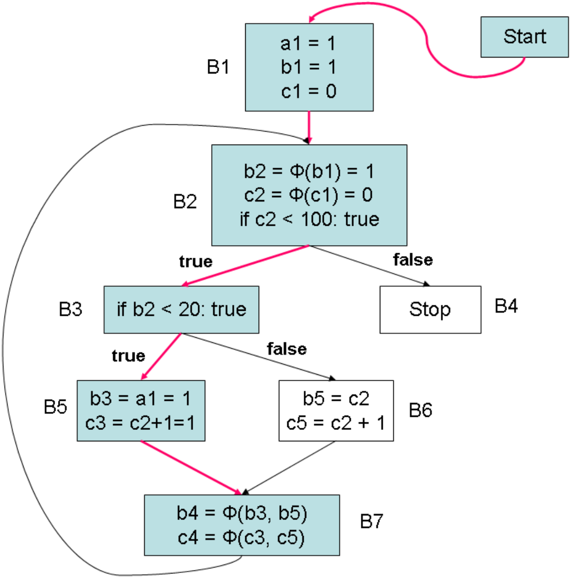
CCP Algorithm - Example 2 - Trace 3



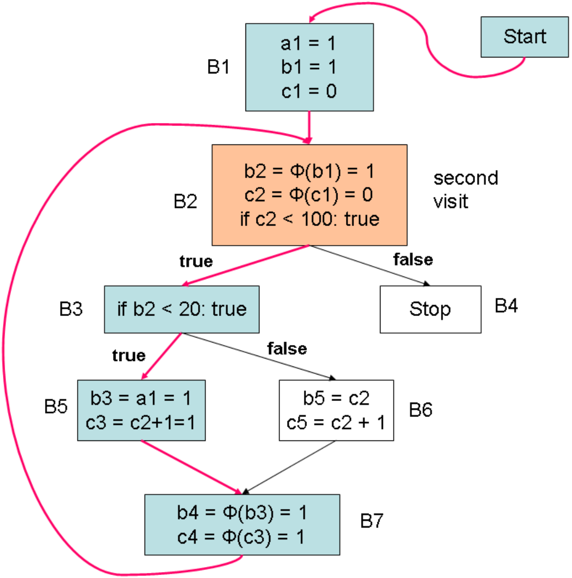
CCP Algorithm - Example 2 - Trace 4



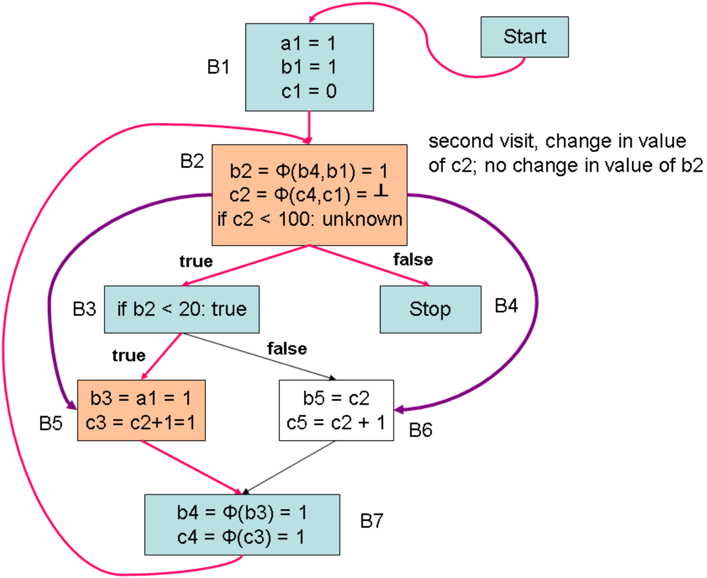
CCP Algorithm - Example 2 - Trace 5



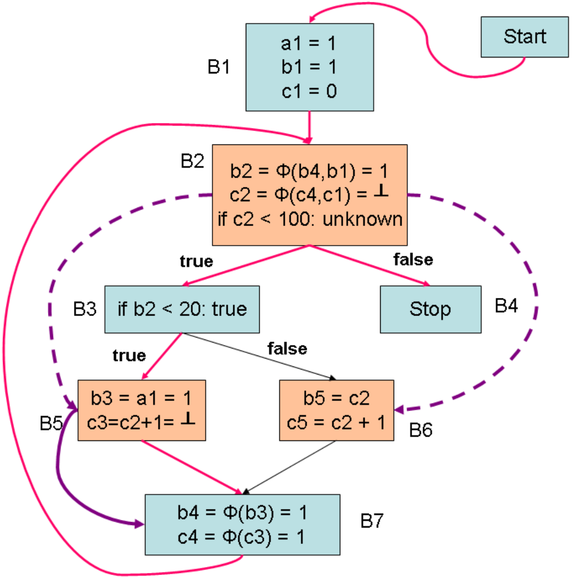
CCP Algorithm - Example 2 - Trace 6

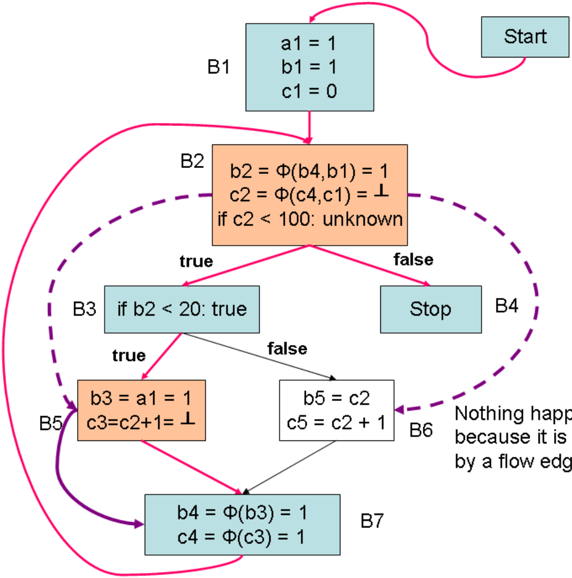


CCP Algorithm - Example 2 - Trace 7



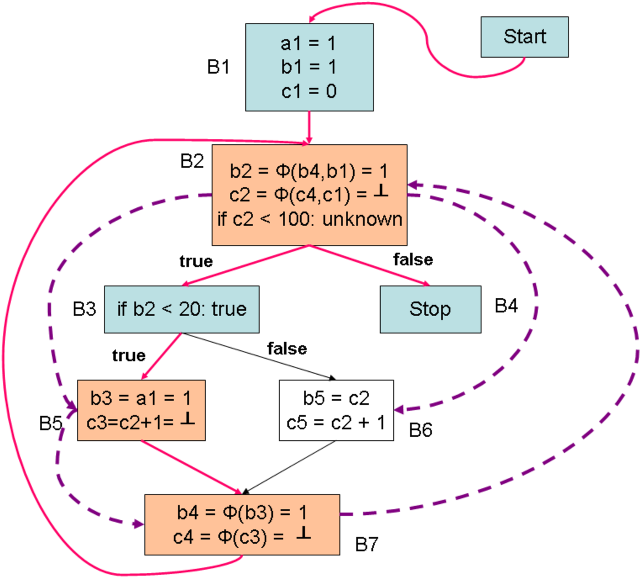
CCP Algorithm - Example 2 - Trace 8



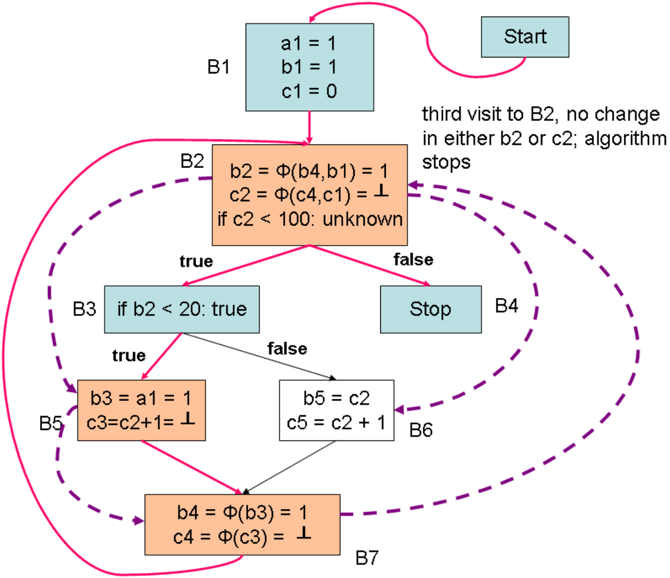


Nothing happens in B6 because it is not reachable by a flow edge

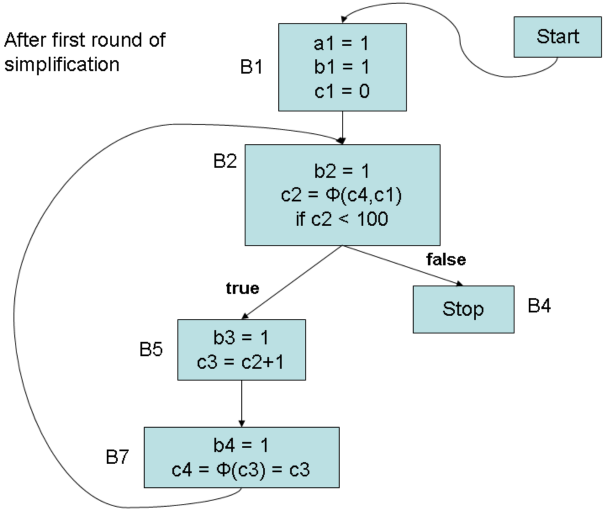
CCP Algorithm - Example 2 - Trace 10



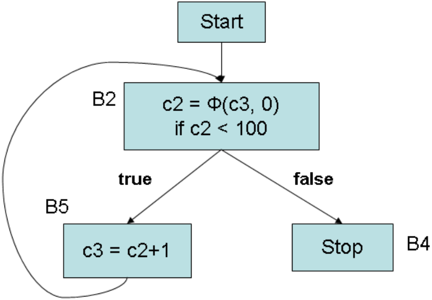
CCP Algorithm - Example 2 - Trace 11



CCP Algorithm - Example 2 - Trace 12



CCP Algorithm - Example 2 - Trace 13



After second round of simplification – elimination of dead code, elimination of trivial Φ-functions, copy propagation etc.

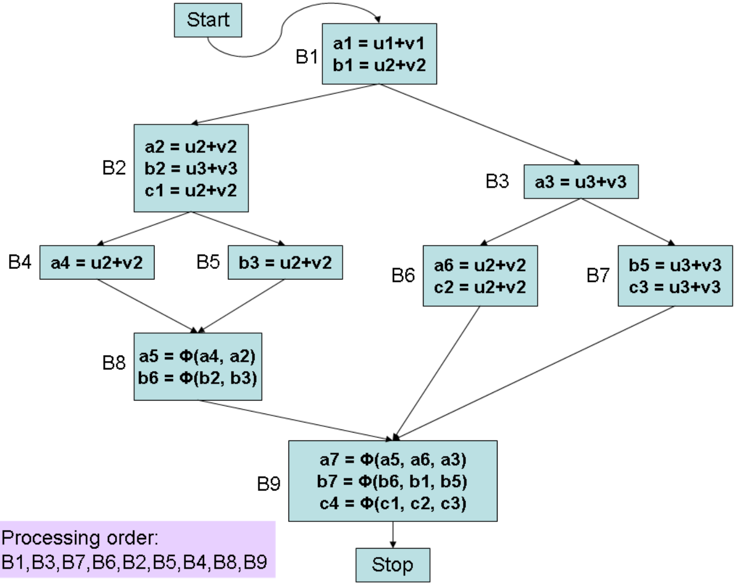
Value Numbering with SSA Forms

- Global value numbering scheme
  - Similar to the scheme with extended basic blocks
  - Scope of the tables is over the dominator tree
  - Therefore more redundancies can be caught
    - For example, an assignment $a_{10} = u_1 + v_1$ in block $B9$ (if present) can use the value of the expression $u_1 + v_1$ of block $B1$, since $B1$ is a dominator of $B9$
- No *d-u* or *u-d* edges needed
- Uses *reverse post order* on the DFS tree of the SSA graph to process the dominator tree
  - This ensures that definitions are processed before use
- Back edges make the algorithm find *fewer* equivalences (more on this later)

Value Numbering with SSA Forms

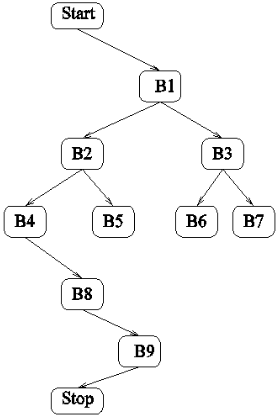
- Variable names are not reused in SSA forms
  - Hence, no need to restore old entries in the scoped *HashTable* when the processing of a block is completed
  - Just deleting new entries will be sufficient
- Any copies generated because of common subexpressions can be deleted immediately
- Copy propagation is carried out during value-numbering
- Ex: Copy statements generated due to value numbering in blocks B2, B4, B5, B6, B7, and B8 can be deleted
- The *ValnumTable* stores the SSA name and its value number and is global; it is not scoped over the dominator tree (reasons in the next slide)
- Value numbering transformation retains the *dominance property* of the SSA form
  - Every definition dominates all its uses or predecessors of uses (in case of *phi*-functions)

Example: An SSA Form

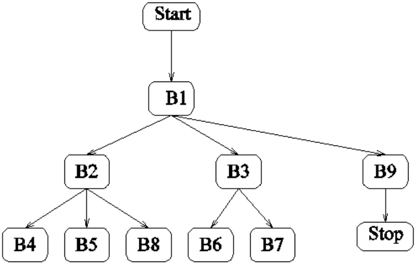


Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

Dominator Tree and Reverse Post order



Postorder on the DFS tree:
Stop, B9, B8, B4, B5, B2, B6, B7, B3, B1, Start



Reverse postorder on the SSA graph that is used with the dominator tree above:

Start,B1,B3,B7,B6,B2,B5,B4,B8,B9,Stop

Global Unscoped *ValnumTable*

- Needed for ϕ-instructions
- A ϕ-instruction receives inputs from several variables along different predecessors of a block
- These inputs are defined in the immediate predecessors or dominators of the predecessors of the current block
- For example, while processing block $B9$, we need definitions of a_5, a_6, and a_3
  - a_5, a_6: defined in the predecessor blocks, $B8$, and $B6$ (resp.)
  - a_3: defined in $B3$, the dominator of the predecessor of $B9$
  - If the *ValnumTable* were to be scoped, only names in $B1$ would be available while processing $B9$
- SSA names being unique, unscoped *ValnumTable* does not cause problems
- Making *HashTable* also unscoped is not possible since expressions are not unique

HashTable and ValnumTable

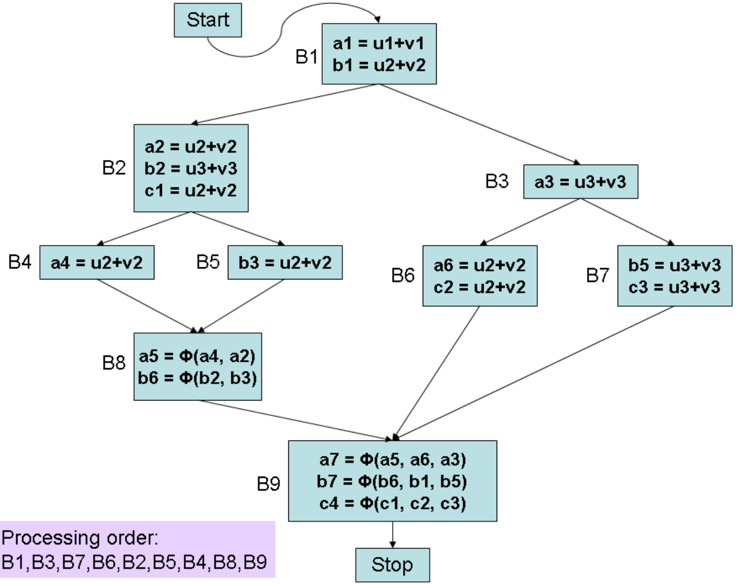
HashTable entry
(indexed by expression hash value)

Expression	Value number	Parameters for ϕ-function	Defining variable

ValnumTable
(indexed by name hash value)

Variable name	Value number	Constant value	Replacing variable

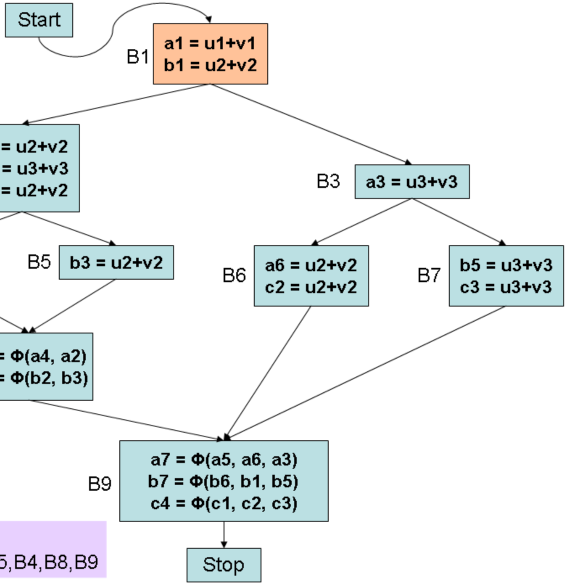
SSA Value-numbering Example - 1.0



Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.1

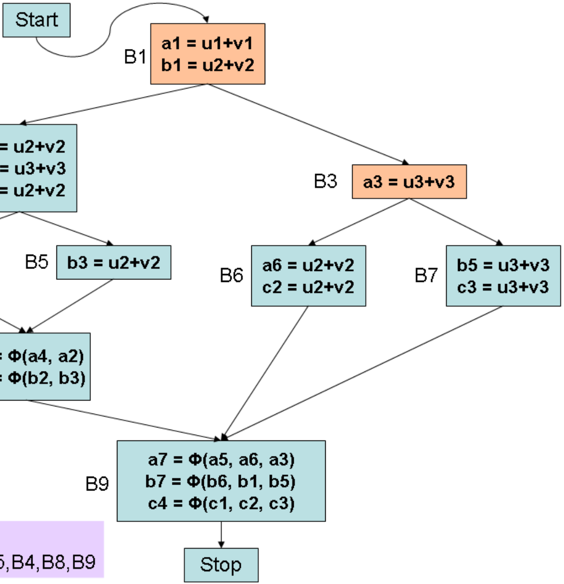
Processing B1



Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.2

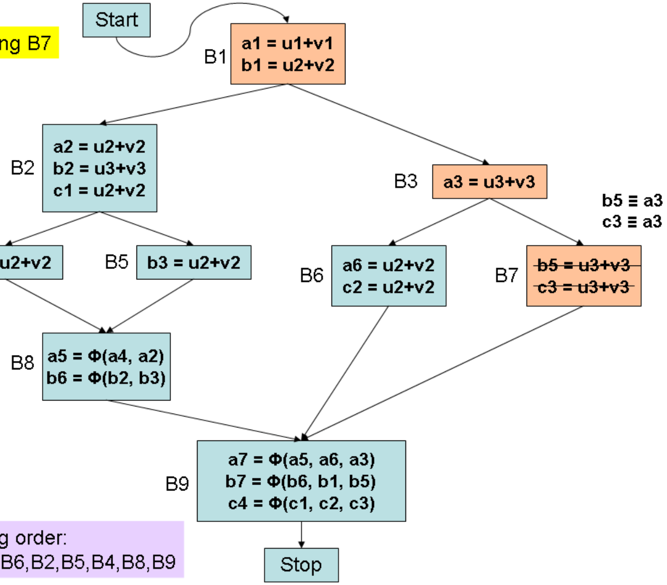
Processing B3



Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.3

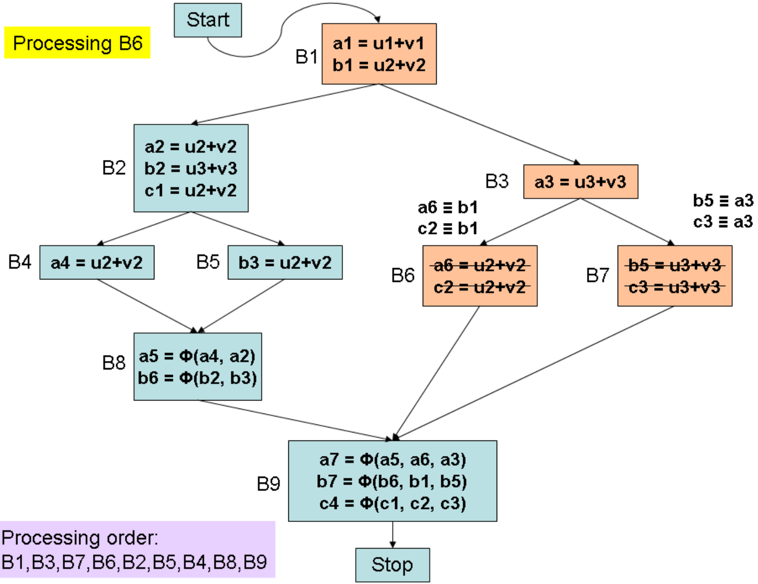
Processing B7



Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

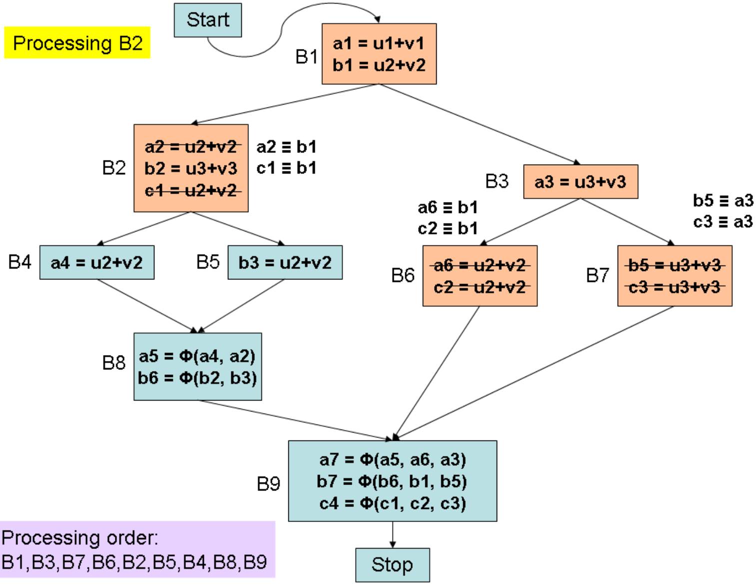
SSA Value-numbering Example - 1.4

Processing B6



Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

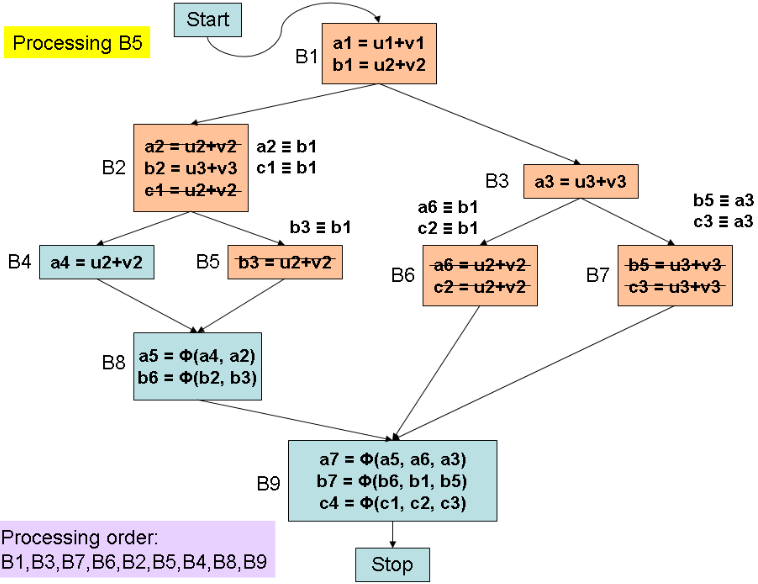
SSA Value-numbering Example - 1.5



Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

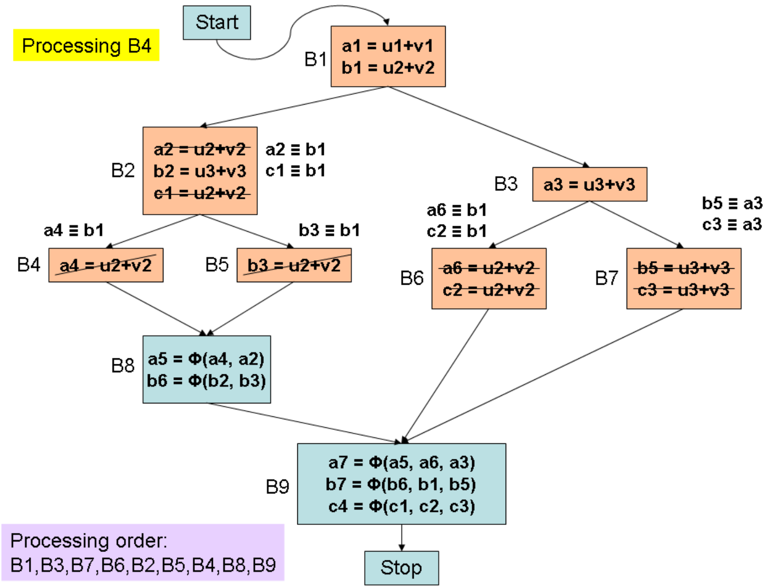
SSA Value-numbering Example - 1.6

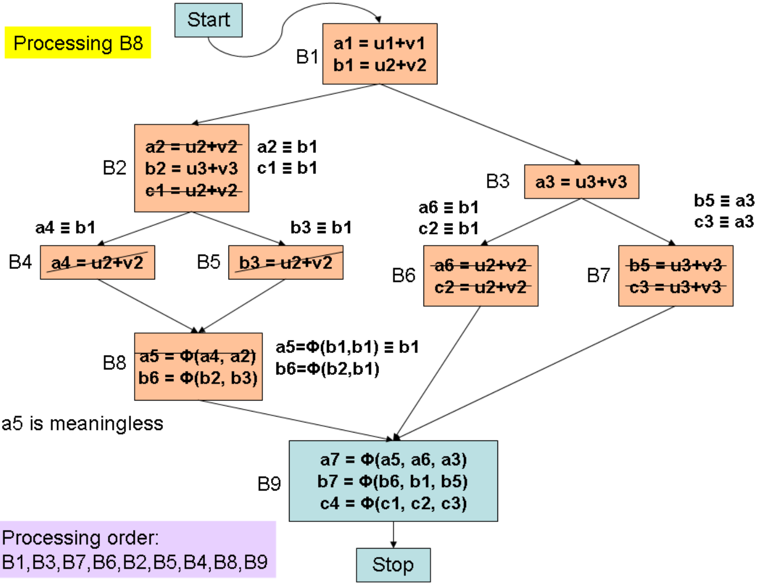
Processing B5



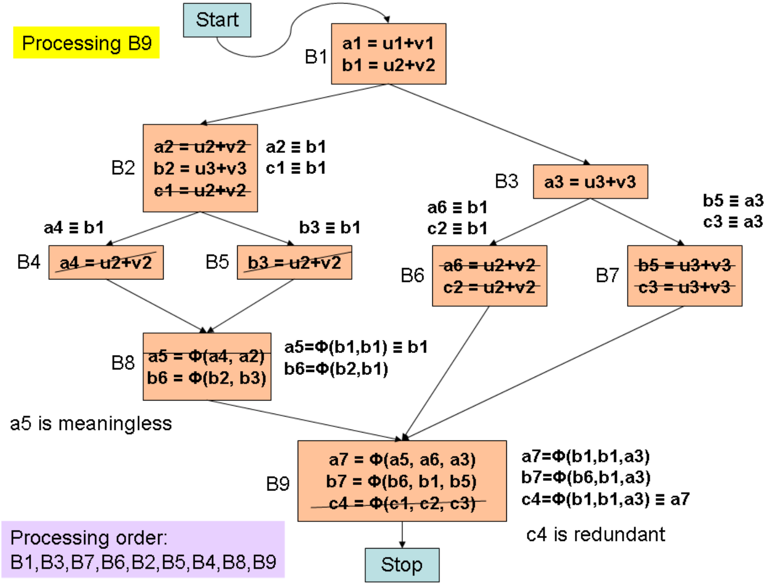
Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.7

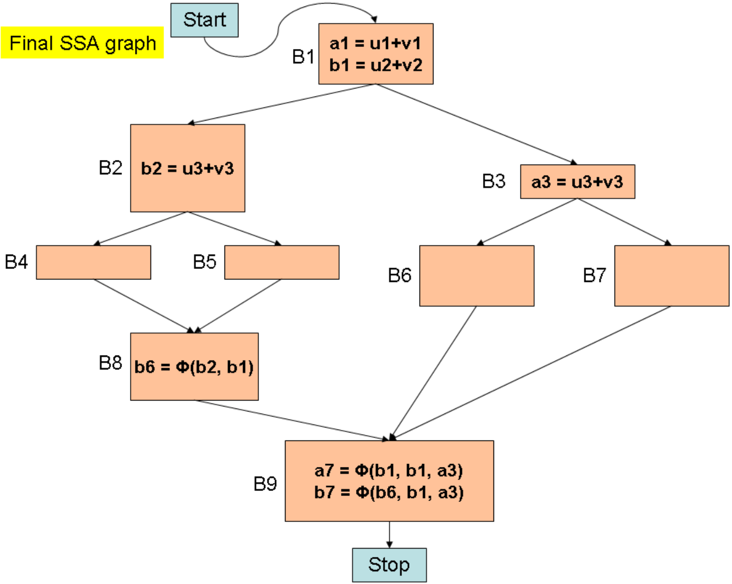



Processing B8
Start
B1
a1 = u1+v1
b1 = u2+v2
B2
~~a2 = u2+v2~~
b2 = u3+v3
~~c1 = u2+v2~~
a2 ≡ b1
c1 ≡ b1
B3
a3 = u3+v3
a4 ≡ b1
b3 ≡ b1
a6 ≡ b1
c2 ≡ b1
b5 ≡ a3
c3 ≡ a3
B4
~~a4 = u2+v2~~
B5
~~b3 = u2+v2~~
B6
~~a6 = u2+v2~~
~~c2 = u2+v2~~
B7
~~b5 = u3+v3~~
~~c3 = u3+v3~~
B8
~~a5 = Φ(a4, a2)~~
b6 = Φ(b2, b3)
a5=Φ(b1,b1) ≡ b1
b6=Φ(b2,b1)
a5 is meaningless
B9
a7 = Φ(a5, a6, a3)
b7 = Φ(b6, b1, b5)
c4 = Φ(c1, c2, c3)
Stop
Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.9



SSA Value-numbering Example - 1.10



SSA Value-Numbering Algorithm

```
function SSA-Value-Numbering (Block B) {
  Mark the beginning of a new scope;
  For each φ-function f of the form x = φ(y1, ..., yn) in B do {
     search for f in HashTable;
     //This involves getting the value numbers of the parameters also
     //Dominance property ensures that parameters are assigned
     //either in predecessor or dominator of predecessor of B
     if f is meaningless //all yi are equivalent to some w
        replace value number of x by that of w in ValnumTable;
        delete f;
     else if f is redundant and is equivalent to z = φ(u1, ..., un)
             replace value number of x by that of z in ValnumTable;
             delete f;
     else insert simplified f into HashTable and ValnumTable;
  }
```

SSA Value-Numbering Algorithm - Contd.

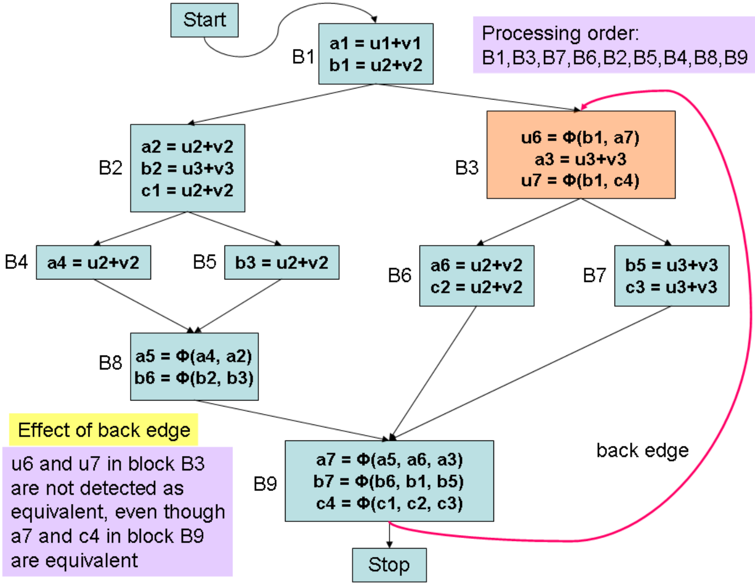
```
For each assignment a of the form x = y + z in B do {
    search for y + z in HashTable;
    //This involves getting value numbers of y and z also
    If present with value number n
        replace value number of x by n in ValnumTable;
        delete a;
    else add simplified y + z to HashTable and x to ValnumTable;
}
For each child c of B in the dominator tree do
//in reverse postorder of DFS over the SSA graph
      SSA-Value-Numbering(c);
clean up HashTable after leaving this scope;
}

//Calling program
SSA-Value-Numbering(Start);
```

Processing ϕ-instructions

- Some times, one or more of the inputs of a ϕ-instruction may not be defined yet
  - They may reach through the back edge of a loop
  - Such entries will not be found in the *ValnumTable*
  - For example, see $a7$ and $c4$ in the ϕ-functions in block B3 (next slide); their equivalence would not have been decided by the time B3 is processed
  - Simply assign a new value number to the ϕ-instruction and record it in the *ValnumTable* and the *HashTable* along with the new value number and the defining variable
- If all the inputs are found in the *ValnumTable* (subject to dominance property being satisfied)
  - Replace the inputs by the respective entries in the *ValnumTable*
  - Now, check whether the ϕ-instruction is either *meaningless* or *redundant*
  - If neither, simplify expression and enter into the tables

Example: Effect of Back Edge on Value Numbering

Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9



Effect of back edge

u6 and u7 in block B3 are not detected as equivalent, even though a7 and c4 in block B9 are equivalent

Processing ϕ-instructions

Meaningless ϕ-instruction

- All inputs are identical. For example, see block B8
- It can be deleted and all occurences of the defining variable can be replaced by the input parameter. *ValnumTable* is updated

Redundant ϕ-instruction

- There is another ϕ-instruction in the *same basic block* with exactly the same parameters
- Block B9 has a redundant ϕ-instruction
- Another ϕ-instruction from a dominating block cannot be used because the control conditions may be different for the two blocks and hence the two ϕ-instructions may yield different values at runtime
- *HashTable* can be used to check redundancy
- A redundant ϕ-instruction can be deleted and all occurences of the defining variable in the redundant instruction can be replaced by the earlier non-redundant one. Tables are updated

Liveness Analysis with SSA Forms

- For each variable v, walk backwards from each use of v, stopping when the walk reaches the definition of v
- Collect the block numbers on the way, and the variable v is *live* at the entry/exit (one or both, as the case may be) of each of these blocks
- In the example (next slide), consider uses of the variable i_2 in B7 and B4. Traversing upwards till B2, we get: B7, B5, B6, B3, B4(IN and OUT points), and OUT[B2], as blocks where i_2 is live
- This procedure works because the SSA forms and the transformations we have discussed satisfy (preserve) the *dominance property*
  - the definition of a variable dominates each use or the predecessor of the use (when the use is in a ϕ-function)
  - Otherwise, the whole SSA graph may have to be searched for the corresponding definition

Liveness Analysis with SSA Forms - Example