The Static Single Assignment Form:

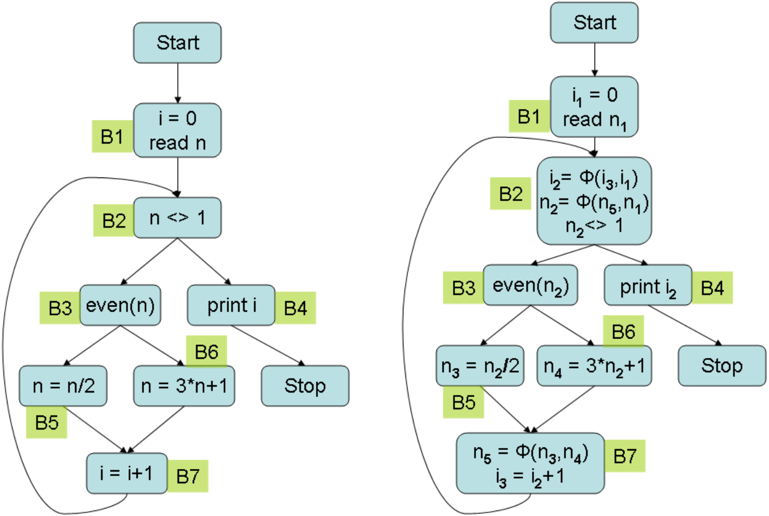
Construction and Application to Program Optimizations - Part 3

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Program 3 in non-SSA and SSA Form



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_1 and y_2
- Simple constant propagation
- Copy propagation
- Conditional constant propagation and constant folding
- Global value numbering

Simple Constant Propagation

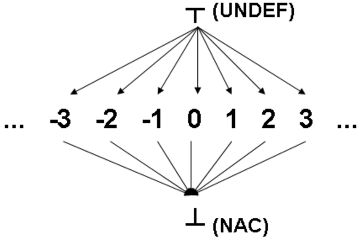
```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
        S = remove(Stmtpile);
        if S is of the form x = φ(c, c, ..., c) for some constant c
              replace S by x = c
        if S is of the form x = c for some constant c
            delete S from the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



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*.

CCP Algorithm - Contd.

// $\mathcal{G} = (\mathcal{N}, \mathcal{E}_f, \mathcal{E}_s)$ 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
```

CCP Algorithm - Contd.

```
        // 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*-$\phi(y)$ // $y \in \mathcal{N}$
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

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

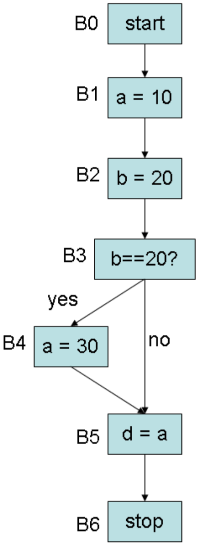
CCP Algorithm - Contd.

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

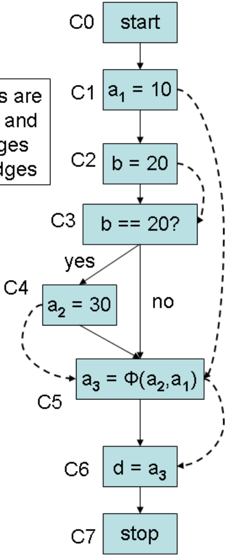
CCP Algorithm - Contd.

else if y is a branch node then
begin
if ($y.newval < y.oldval$) then
begin
$y.oldval = y.newval$;
switch($y.newval$)
case $\perp$: // Both *true* and *false* branches are equally likely
$Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f\}$;
case *true*: $Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f$ and
(y, z) is the *true* branch edge at $y\}$;
case *false*: $Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f$ and
(y, z) is the *false* branch edge at $y\}$;
end switch
end
end
end

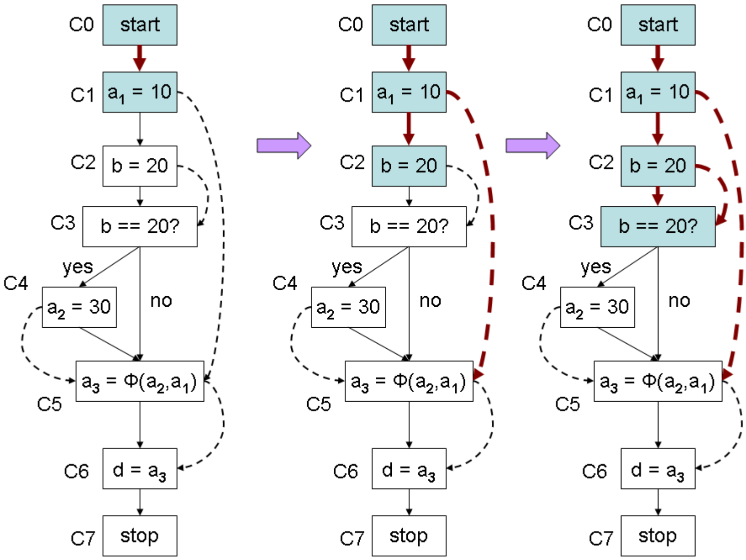
CCP Algorithm - Example - 1



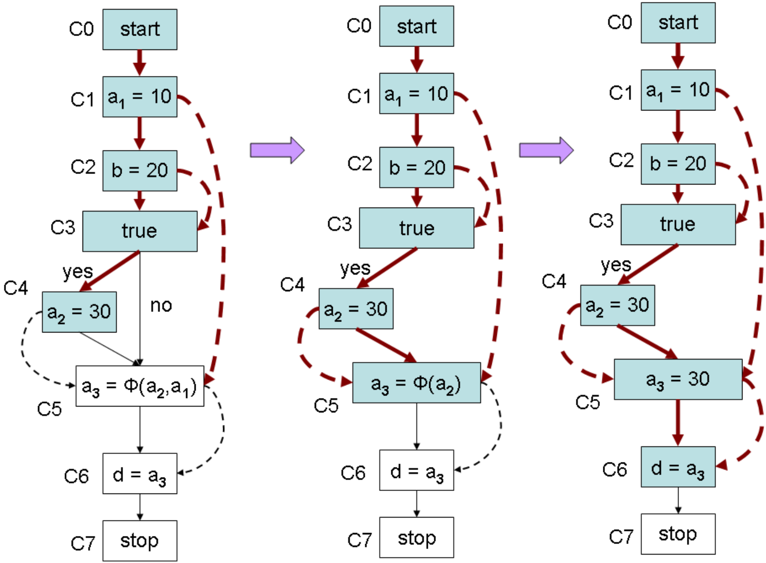
Solid edges are flow edges and dashed edges are SSA edges



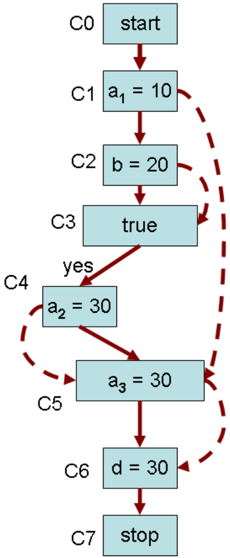
CCP Algorithm - Example 1 - Trace 1



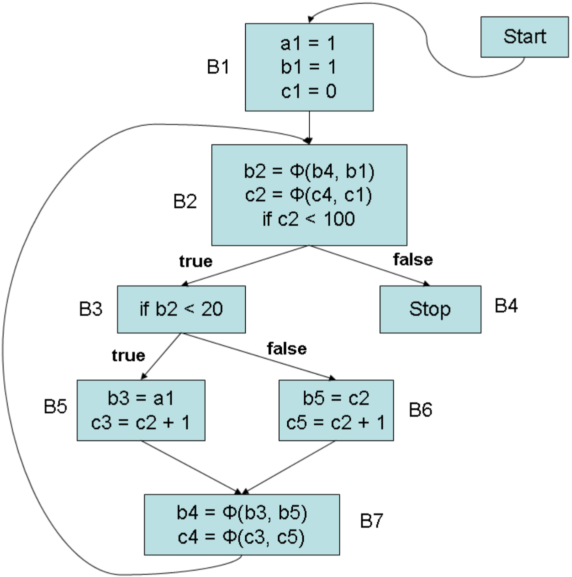
CCP Algorithm - Example 1 - Trace 2



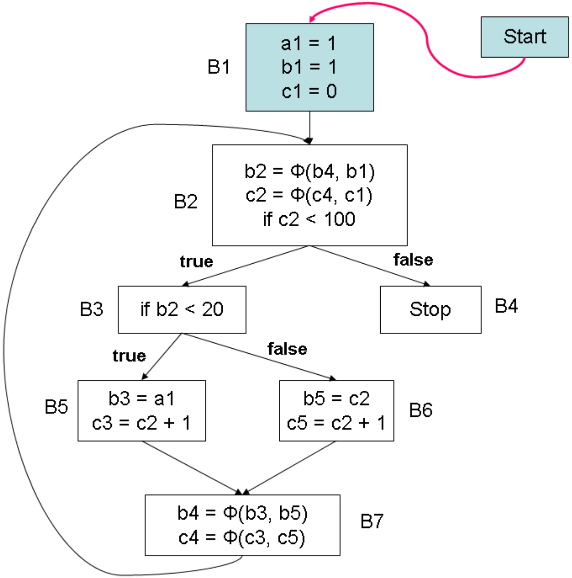
CCP Algorithm - Example 1 - Trace 3



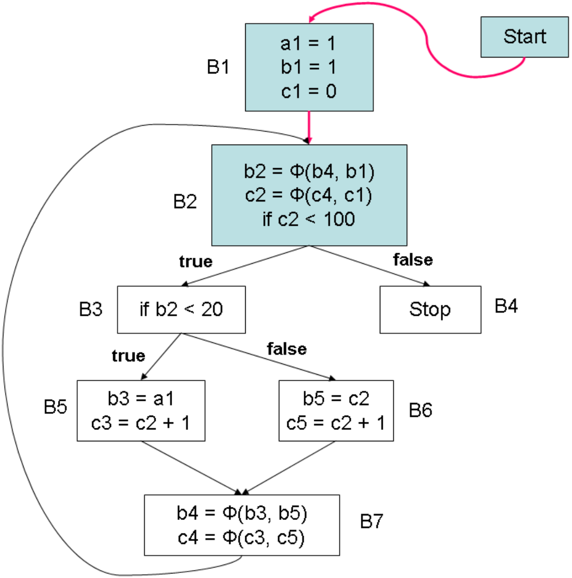
CCP Algorithm - Example 2



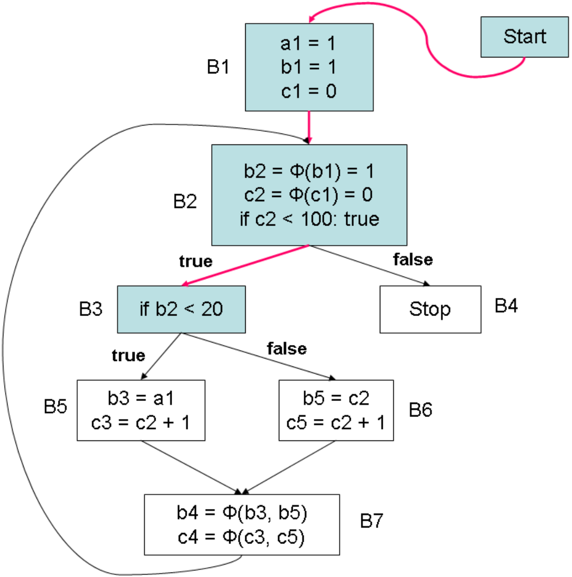
CCP Algorithm - Example 2 - Trace 1



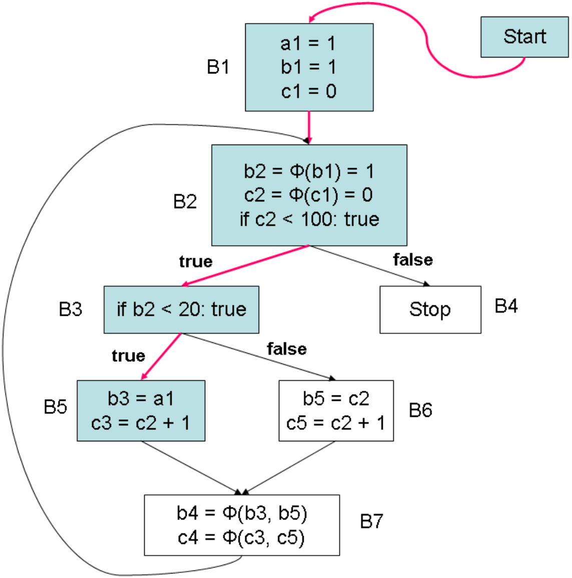
CCP Algorithm - Example 2 - Trace 2



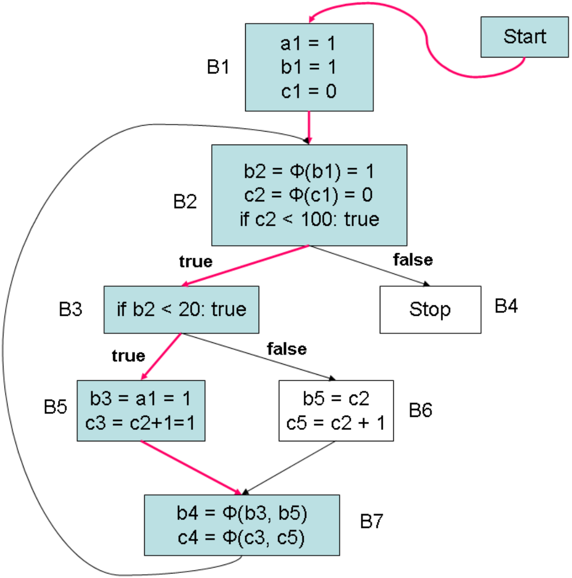
CCP Algorithm - Example 2 - Trace 3



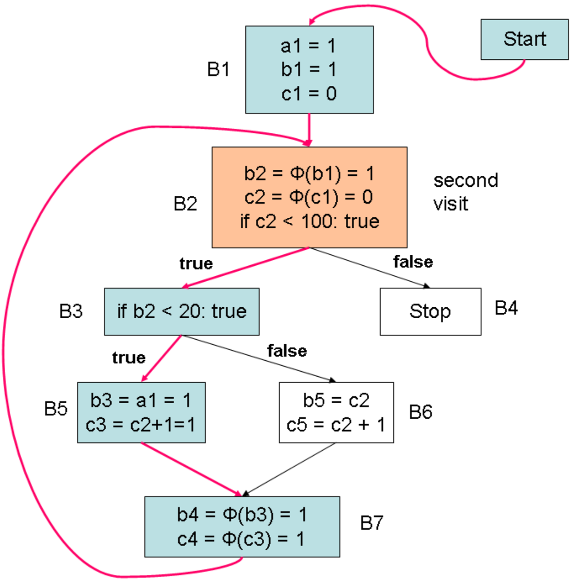
CCP Algorithm - Example 2 - Trace 4



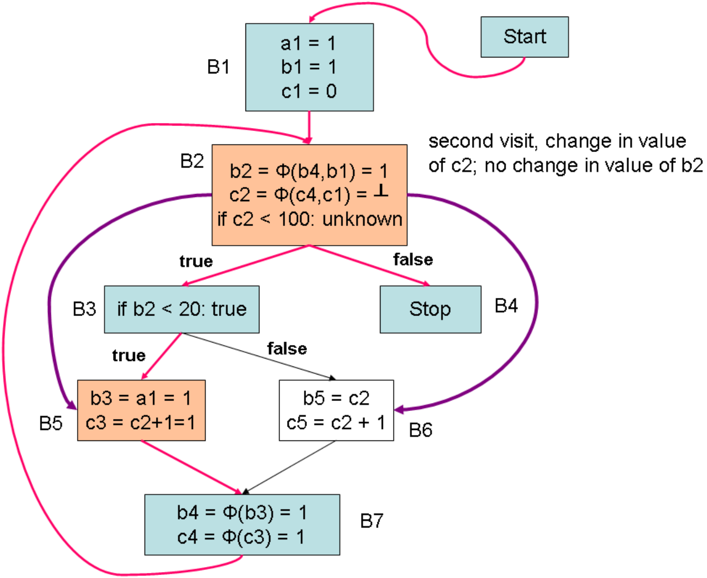
CCP Algorithm - Example 2 - Trace 5



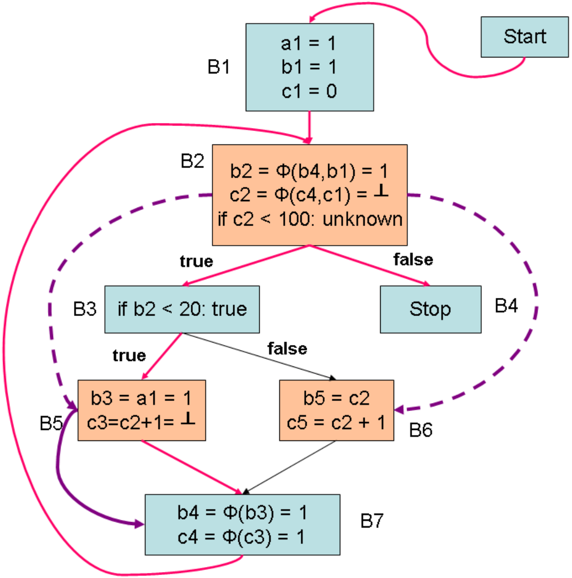
CCP Algorithm - Example 2 - Trace 6



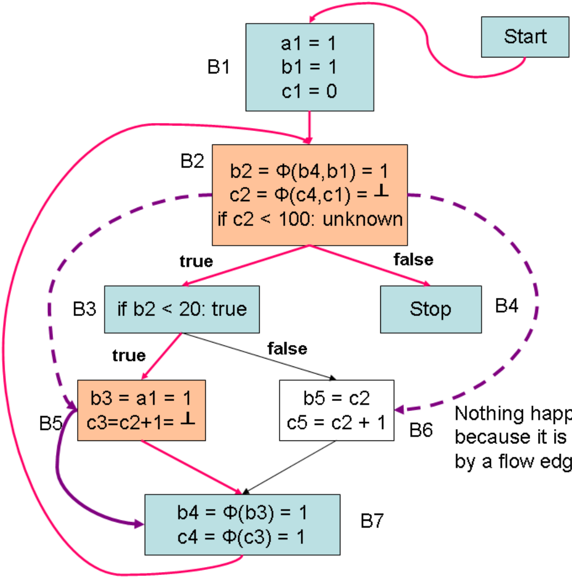
CCP Algorithm - Example 2 - Trace 7



CCP Algorithm - Example 2 - Trace 8

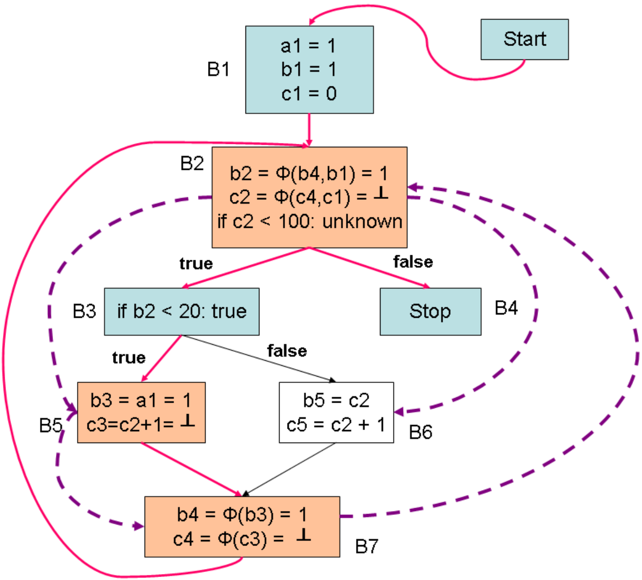


CCP Algorithm - Example 2 - Trace 9

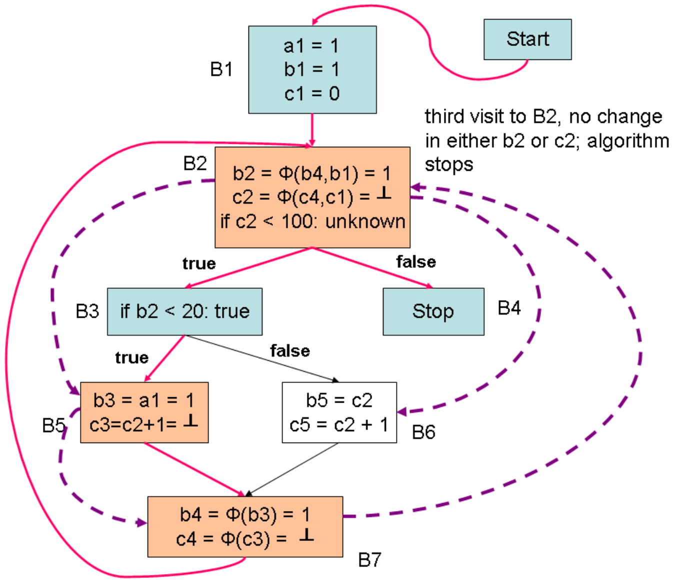


Nothing happens in B6 because it is not reachable by a flow edge

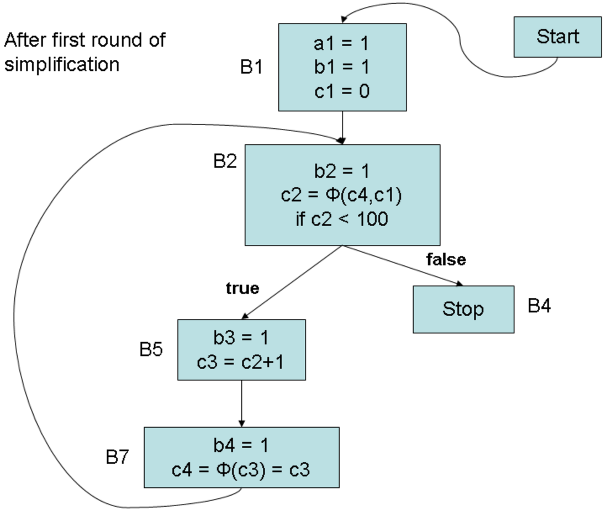
CCP Algorithm - Example 2 - Trace 10



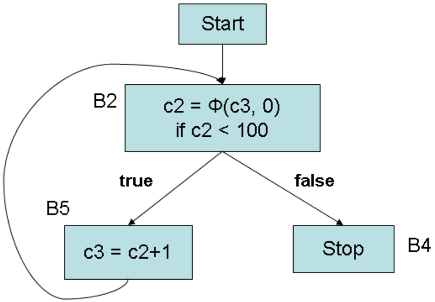
CCP Algorithm - Example 2 - Trace 11



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.

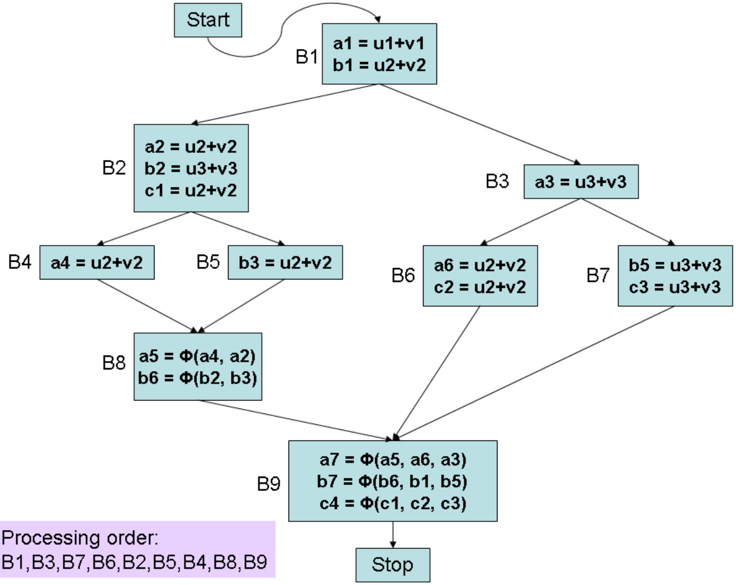
Value Numbering with SSA Forms

- 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 (e.g., expressions in block *B*8, such as $d_1 = u_1 + v_1$, which are equivalent to a_1 in block *B*1)
- 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)
- Scoped *HashTable* (scope over the dominator tree)
  - For example, an assignment $a_{10} = u_1 + v_1$ in block *B*9 (if present) can use the value of the expression $u_1 + v_1$ of block *B*1, since *B*1 is a dominator of *B*9

Value Numbering with SSA Forms

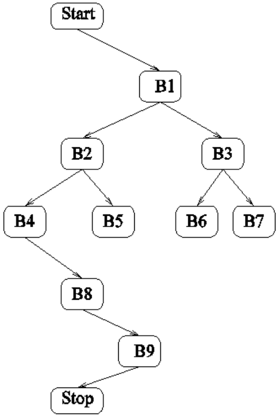
- Variable names are not reused in SSA forms
  - Hence, no need to restore old entries in the scoped *HashTable* when the processing of a block is completed
  - Just deleting new entries will be sufficient
- Any copies generated because of common subexpressions can be deleted immediately
- Copy propagation 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 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)

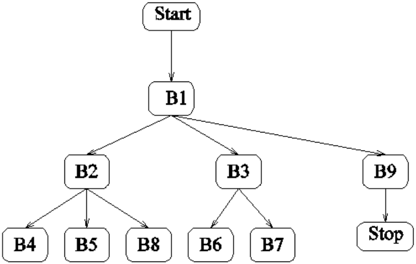
Example: An SSA Form



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

Dominator Tree and Reverse Post order





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

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

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
- They may be defined in any block that has a control path to 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 block, $B6$, and
  - a_3: defined in the dominator of the predecessor of $B9$, i.e., $B3$
- However, each incoming arc corresponds to exactly one parameter of the ϕ-instruction
- Hence we need an *unscoped ValnumTable*

HashTable and ValnumTable

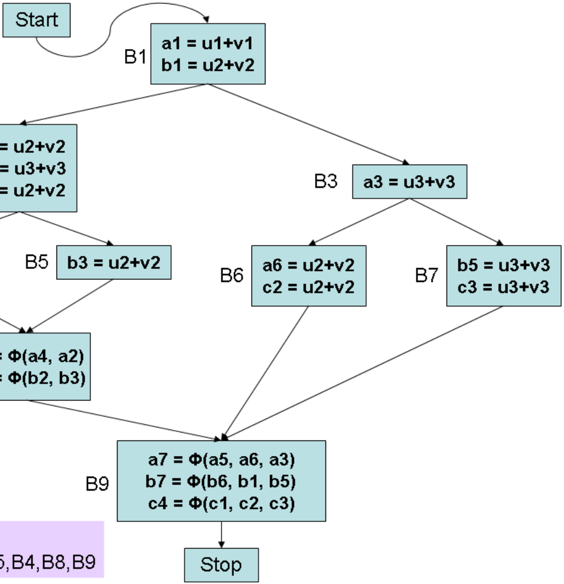
HashTable entry
(indexed by expression hash value)

Expression	Value number	Parameters for ϕ-function	Defining variable

ValnumTable
(indexed by name hash value)

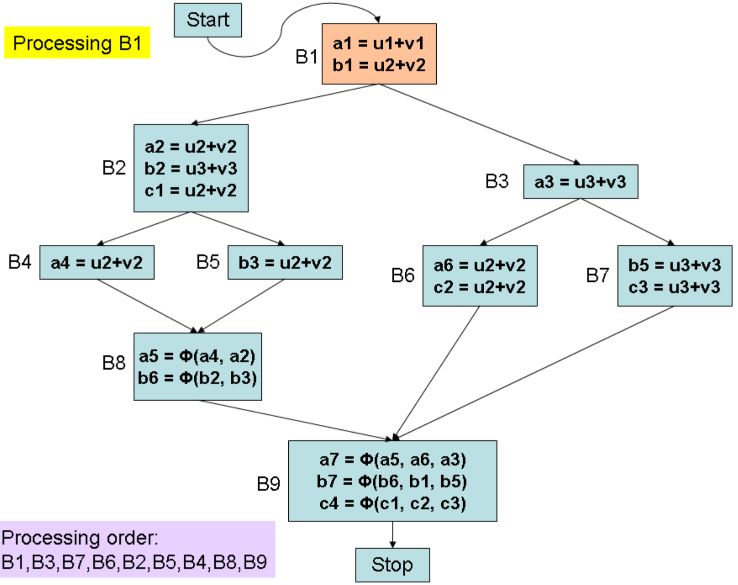
Variable name	Value number	Constant value	Replacing variable

SSA Value-numbering Example - 1.0



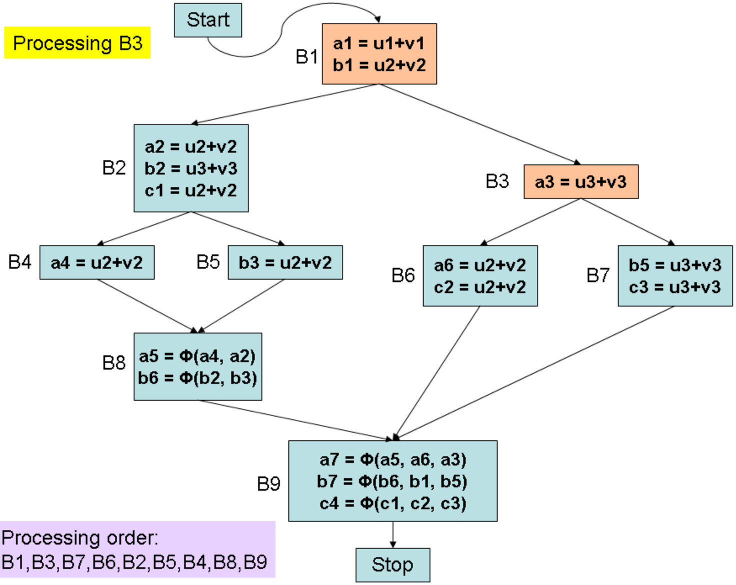
Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.1



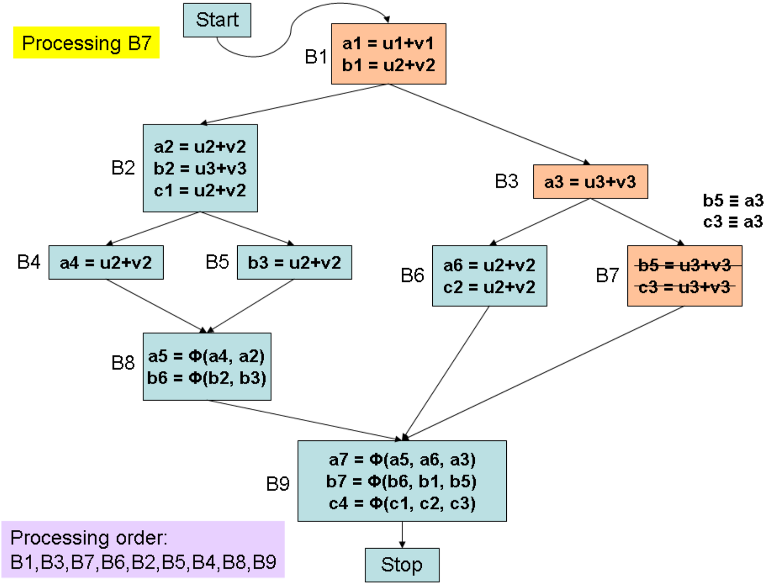
Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.2

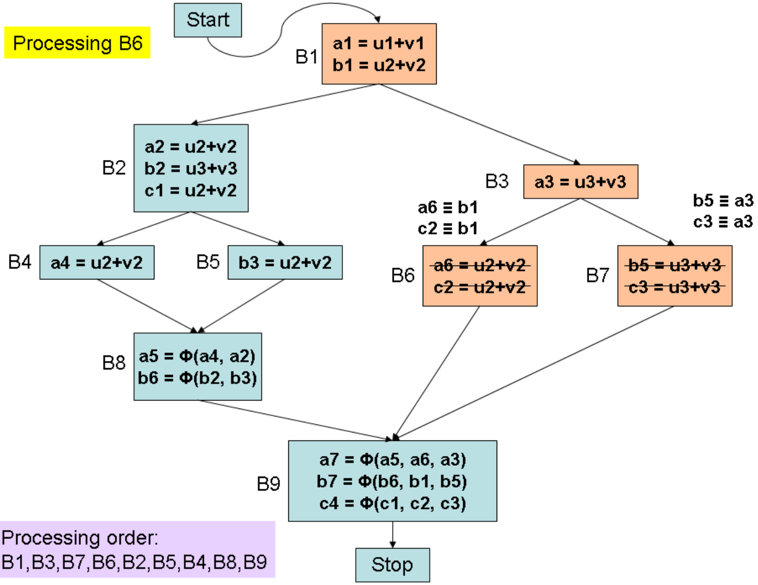


Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.3

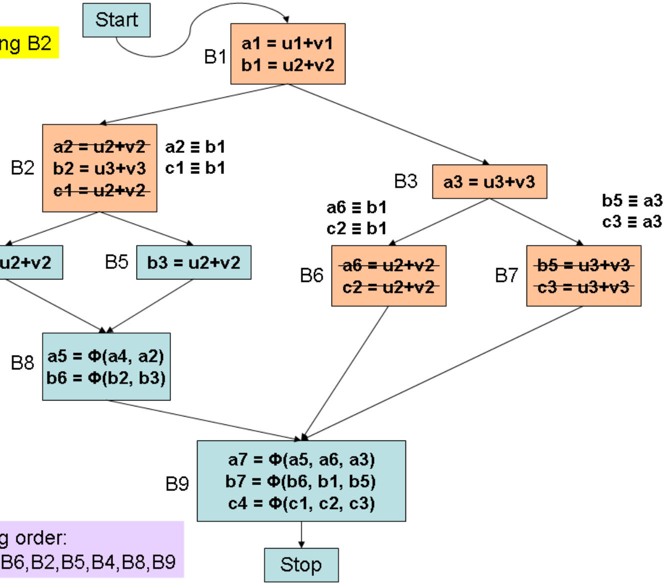


SSA Value-numbering Example - 1.4



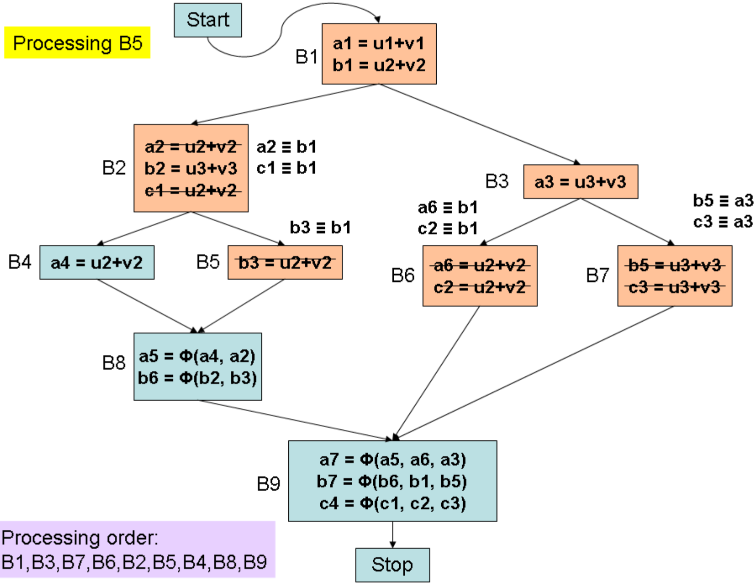
SSA Value-numbering Example - 1.5

Processing B2



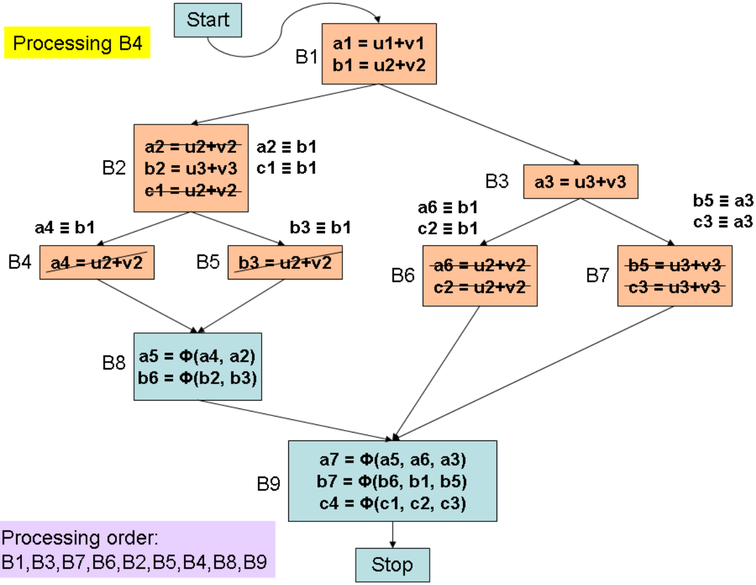
Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

SSA Value-numbering Example - 1.6



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

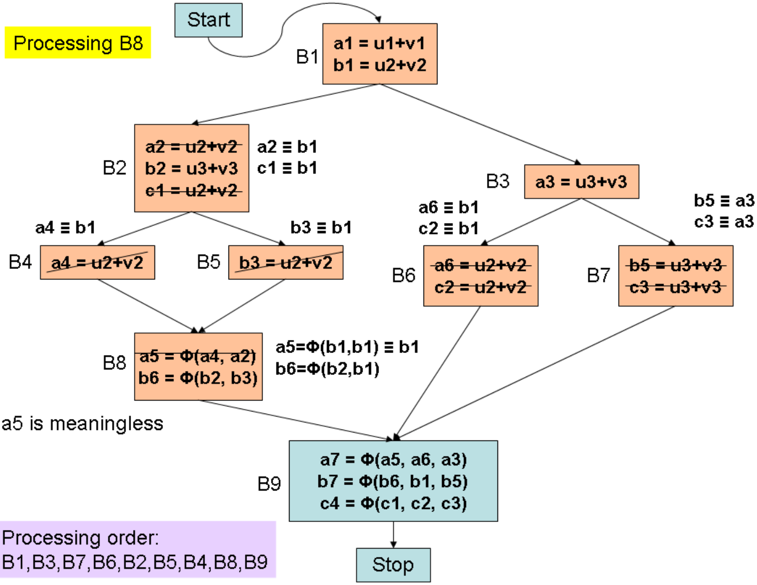
SSA Value-numbering Example - 1.7



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

SSA Value-numbering Example - 1.8

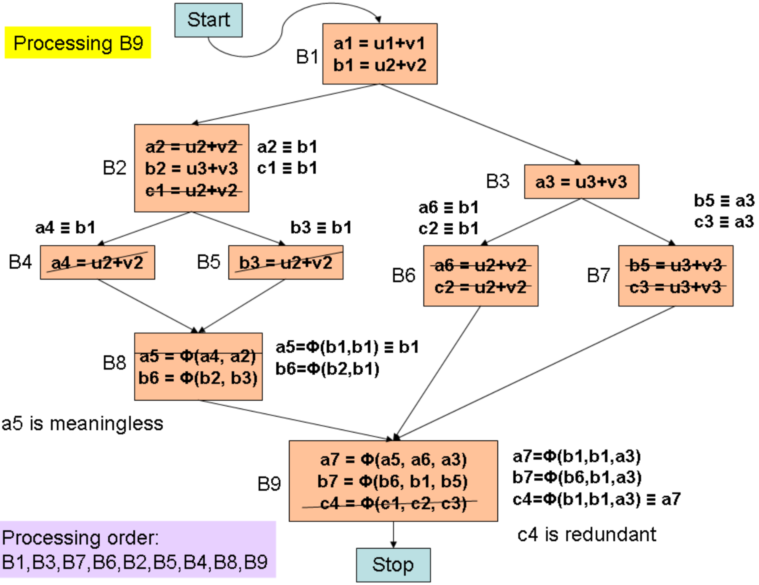
Processing B8



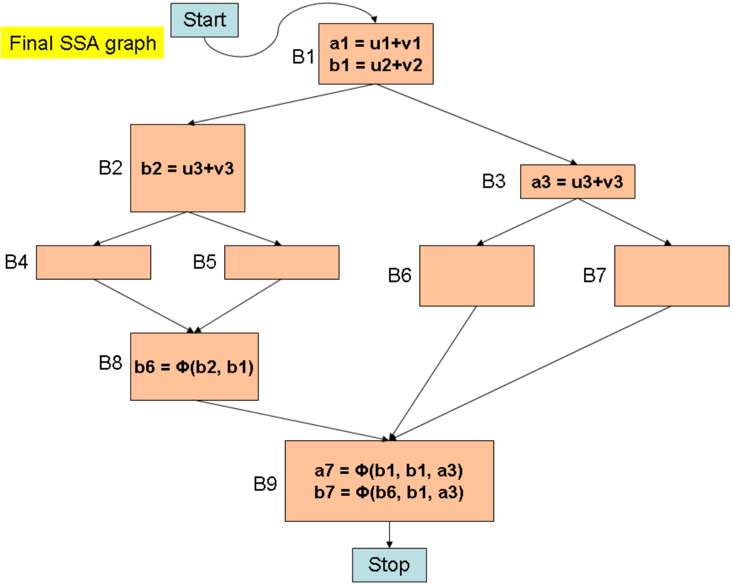
a5 is meaningless

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(y_1, ..., y_n) in B do {
    search for f in HashTable;
    //This involves getting the value numbers of the parameters also
    if f is meaningless //all y_i 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(u_1, ..., u_n)
          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.

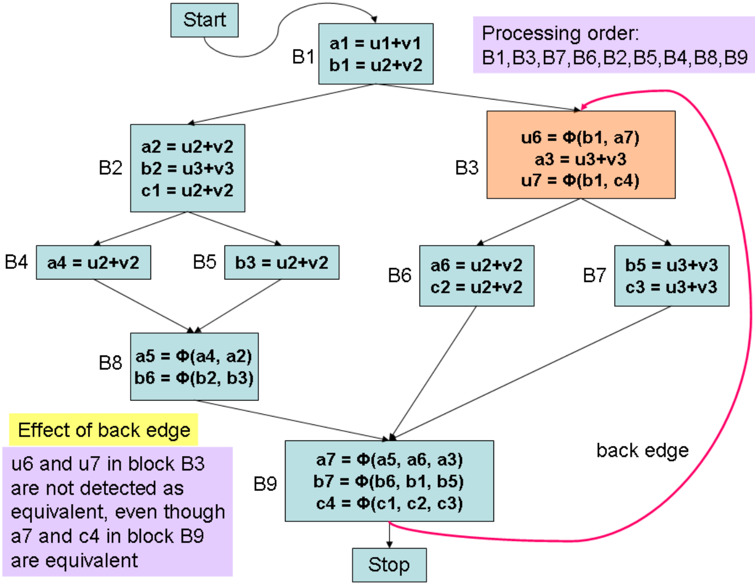
```
    For each assignment a of the form x = y + z in B do {
        search for y + z in HashTable;
        //This involved getting value numbers of y and z also
        If present with value number n
            replace value number of x by n in ValnumTable;
            delete a;
        else add simplified 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 yet be defined
  - They may reach through the back edge of a loop
  - Such entries will not be found in the *ValnumTable*
  - For example, see *a*7 and *c*4 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*
  - Replace the inputs by the respective entries in the *ValnumTable*
  - Now, check whether the ϕ-instruction is either *meaningless* or *redundant*
  - If neither, enter the simplified expression into the tables as before

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