The Static Single Assignment Form:

Construction and Application to Program Optimizations - Part 2

Y.N. Srikant

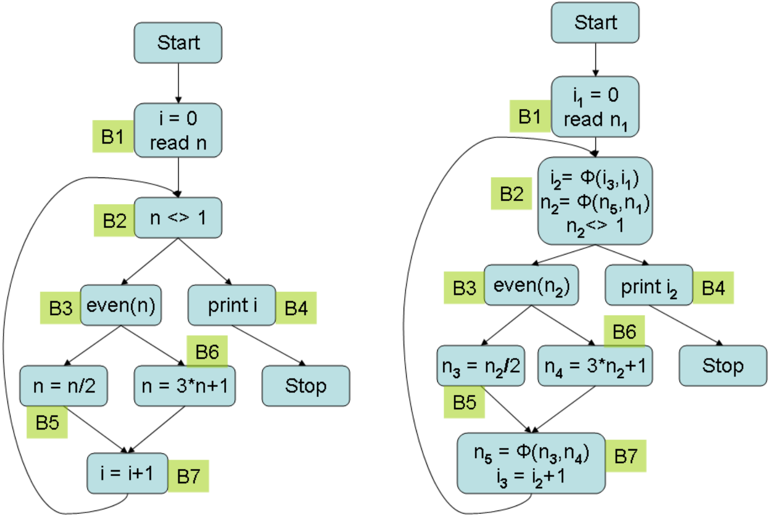
Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow 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, the 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 3 in non-SSA and SSA Form



Conditions on the SSA form

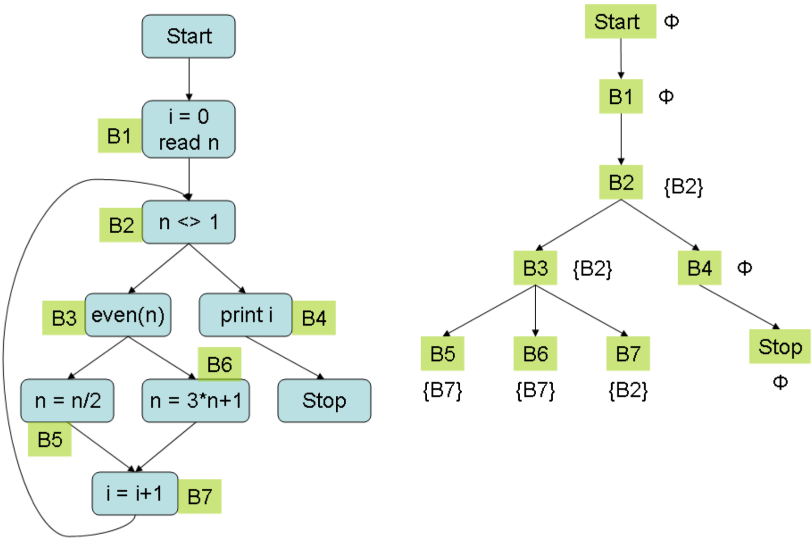
After translation, the SSA form should satisfy the following conditions for every variable v in the original program.

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 Each appearance of v in the original program or a ϕ-function in the new program has been replaced by a new variable v_i, leaving the new program in SSA form.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

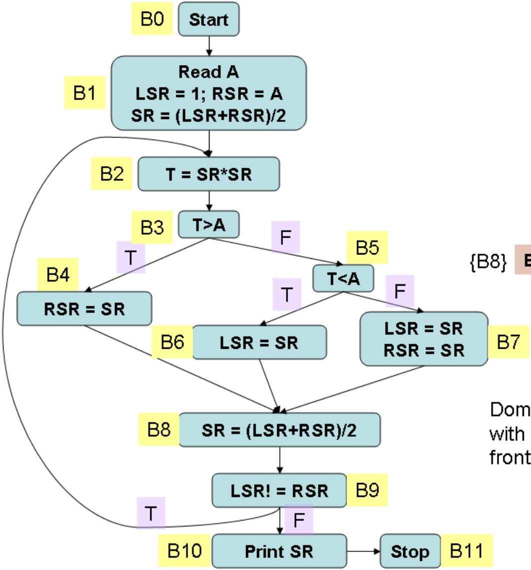
- Condition 1 in the previous slide is recursive.
  - It implies that ϕ-assignments introduced by the translation procedure will also qualify as assignments to *v*
  - This in turn may lead to introduction of more ϕ-assignments at other nodes
- It would be wasteful to place ϕ-functions in all join nodes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

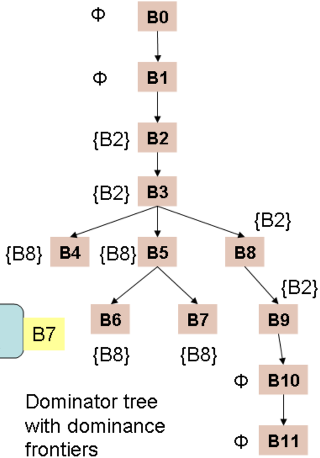
DF Example - 1



Start
B1
i = 0
read n
B2
n <> 1
B3
even(n)
print i
B4
B6
n = n/2
n = 3*n+1
Stop
B5
i = i+1
B7
Start Φ
B1 Φ
B2 {B2}
B3 {B2}
B4 Φ
B5 {B7}
B6 {B7}
B7 {B2}
Stop Φ

DF Example - 2





Dominator tree with dominance frontiers

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);
       }
     }
  }
}
```

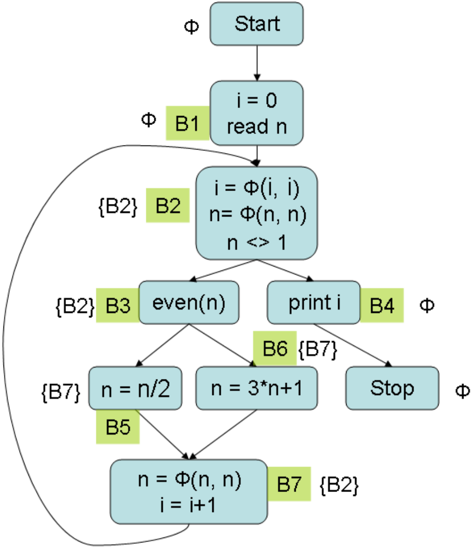
Minimal SSA Form Construction 1

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.

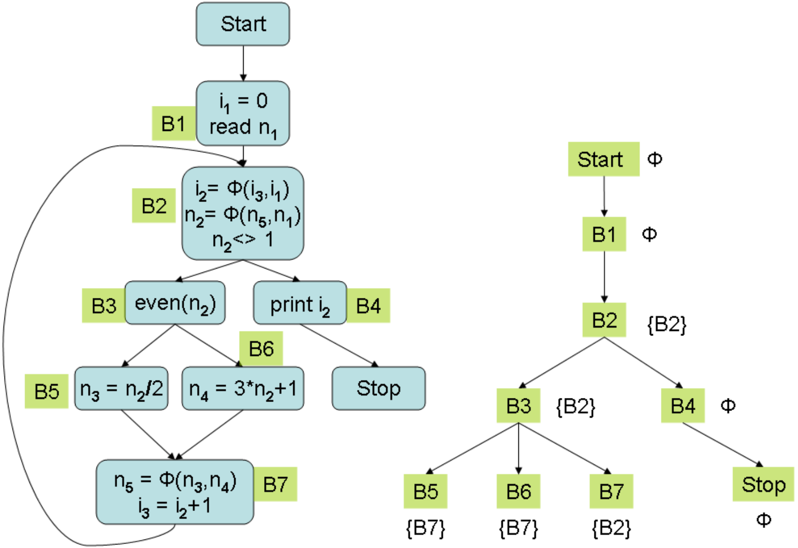
The function *place-phi-function(v)* - 1

```
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) is true if a φ-function has already
  // been placed in B
  // 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) = 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)) then
    begin
      place < v = φ(v, v, ..., v) > in y;
      has-phi(y) = true;
      if (not processed(y)) then
      begin processed(y) = true;
          Add(W, y);
      end
    end
  end for
end
end
```

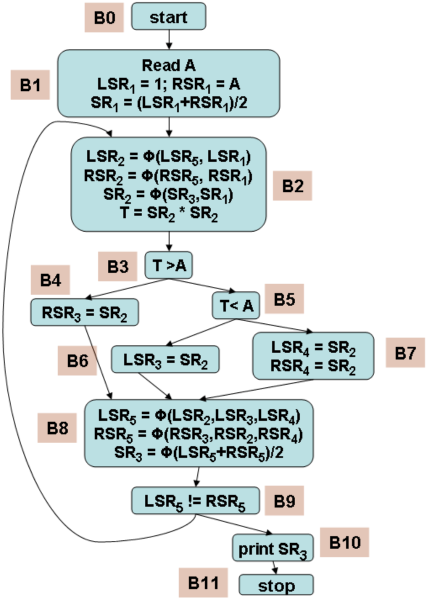
SSA Form Construction Example - 1

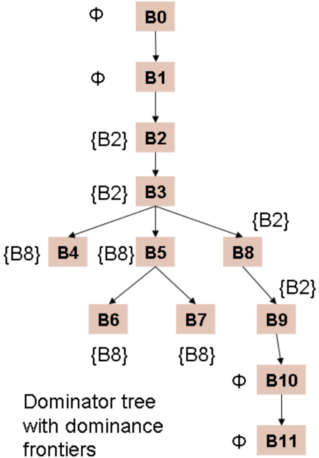


SSA form

Dominator tree with dominance frontier

SSA Form Construction Example - 2





Dominator tree with dominance frontiers

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*

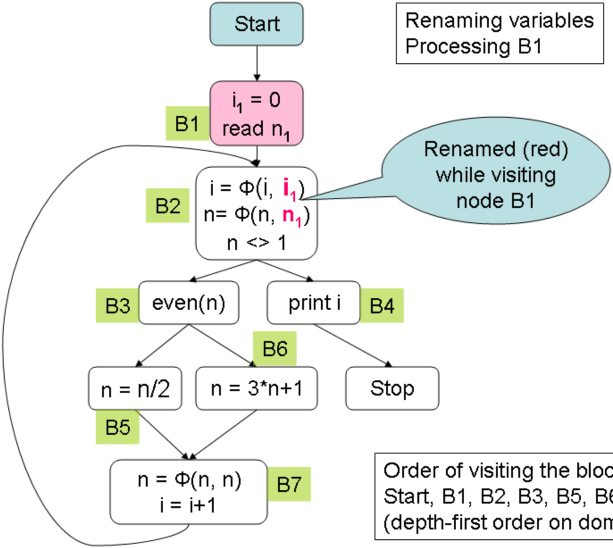
The function *Rename-variables(x,B)*

```
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
  for all statements s ∈ 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
```

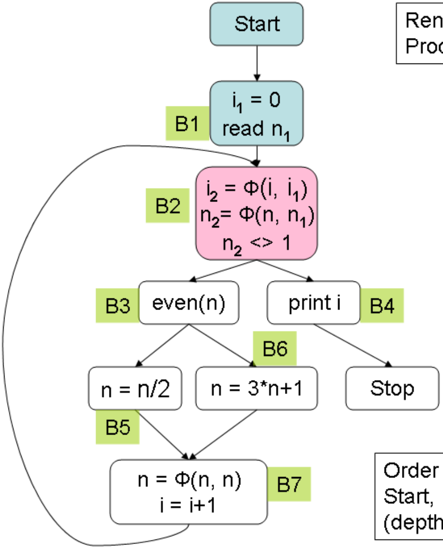
Renaming Variables Example 0.1



Renaming variables
Processing B1

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

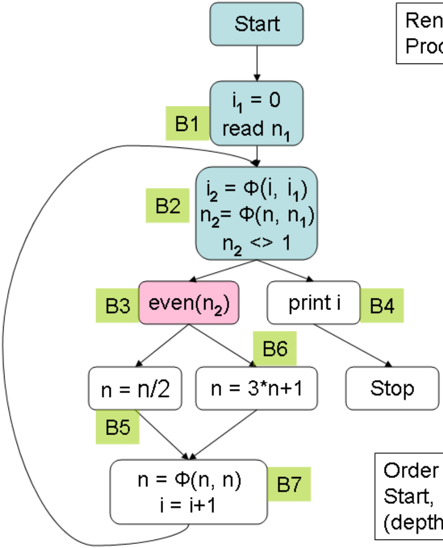
Renaming Variables Example 0.2



Renaming variables
Processing B2

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

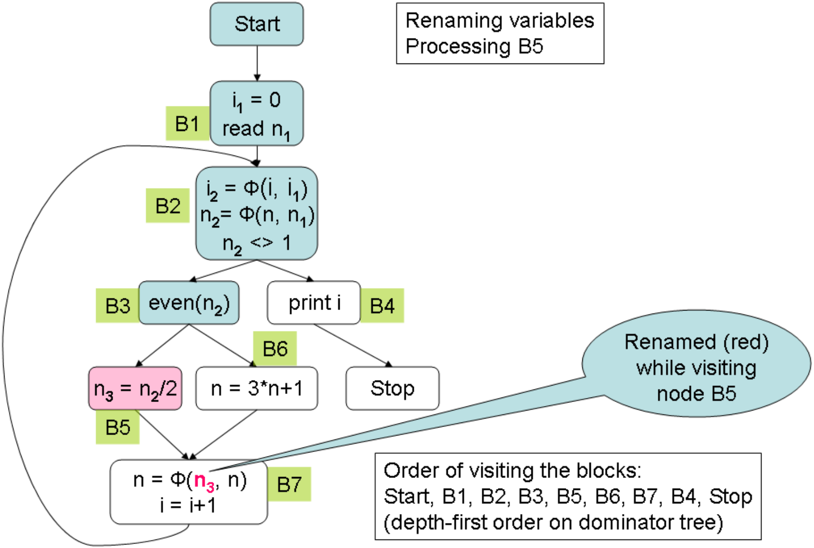
Renaming Variables Example 0.3



Renaming variables
Processing B3

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

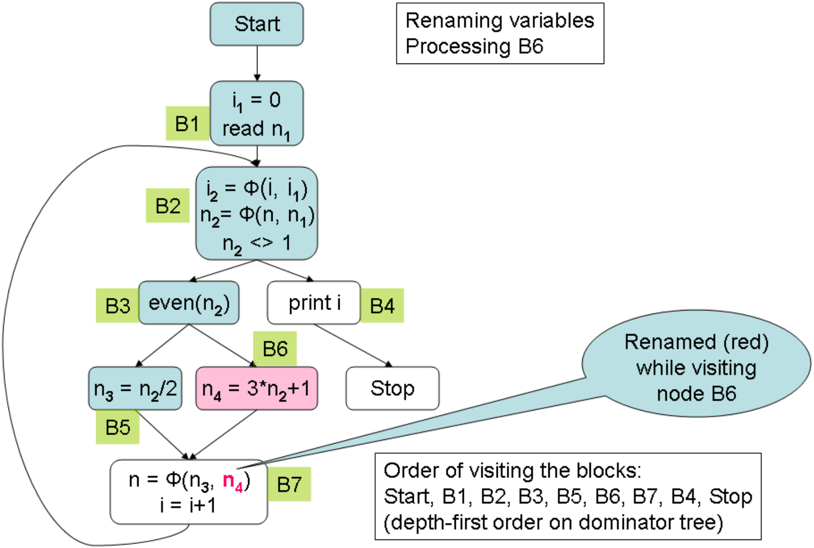
Renaming Variables Example 0.4



Renaming variables
Processing B5

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

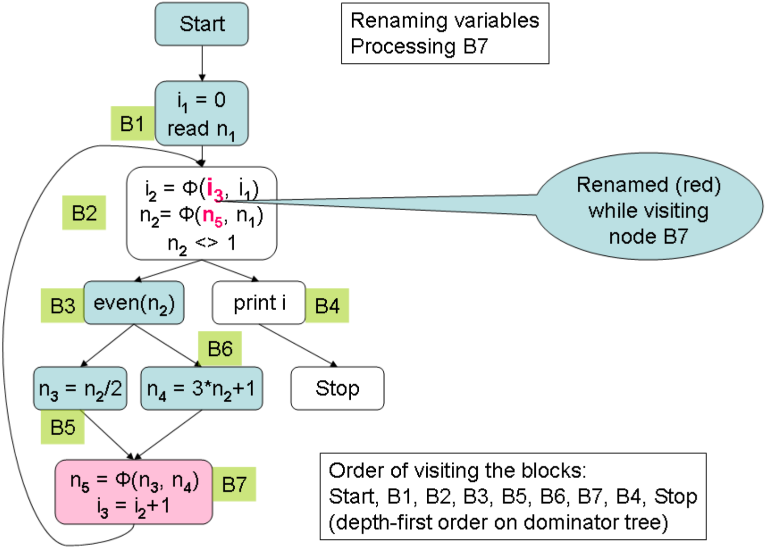
Renaming Variables Example 0.5



Renaming variables
Processing B6

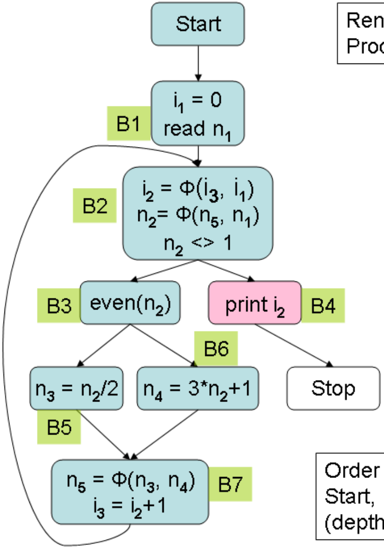
Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

Renaming variables
Processing B7



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

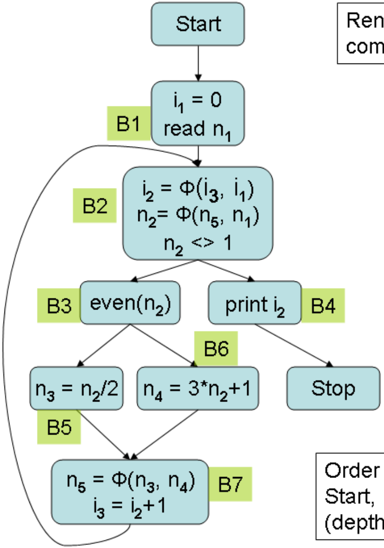
Renaming Variables Example 0.7



Renaming variables
Processing B4

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

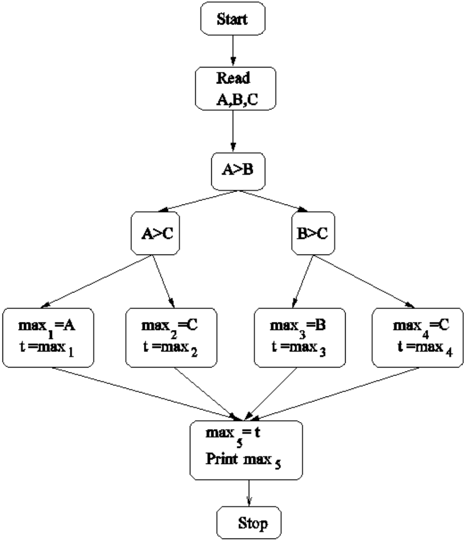
Renaming Variables Example 0.8



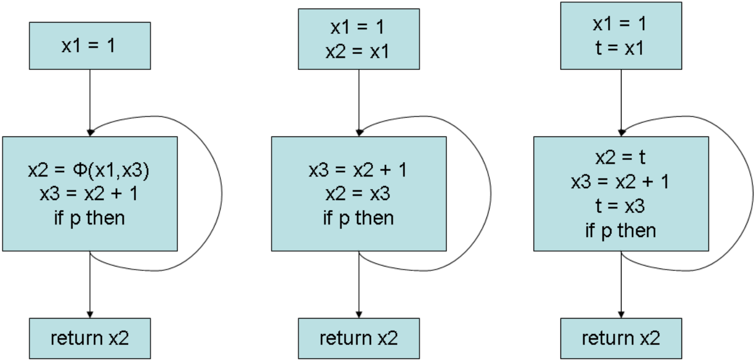
Renaming variables completed

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



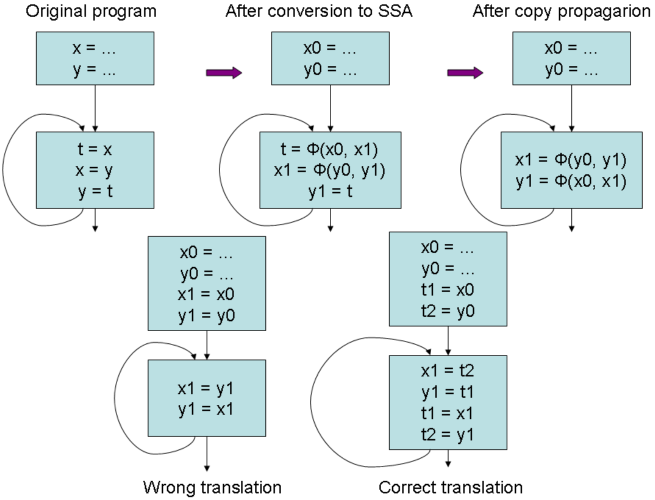
Original program

Wrong tranlsation

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

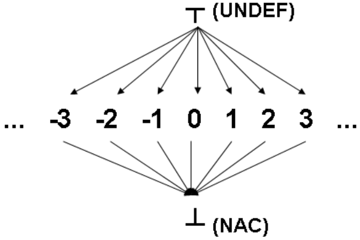
```
{  Stmtpile = {S|S is a statement in the program}
   while Stmtpile is not empty {
      S = remove(Stmtpile);
      if S is of the form x = φ(c, c, ..., c) for some constant c
           replace S by x = c
      if S is of the form x = c for some constant c
          delete S from the program
 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



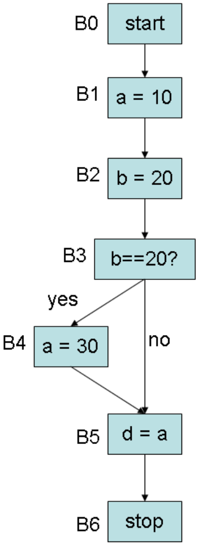
Conditional Constant Propagation - 1

- 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

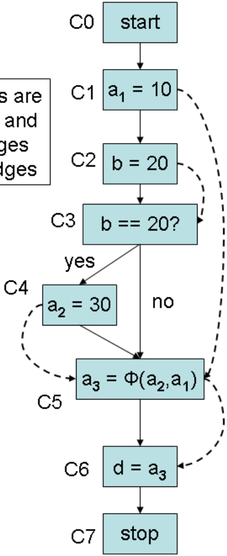
Conditional Constant Propagation - 2

- 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*.

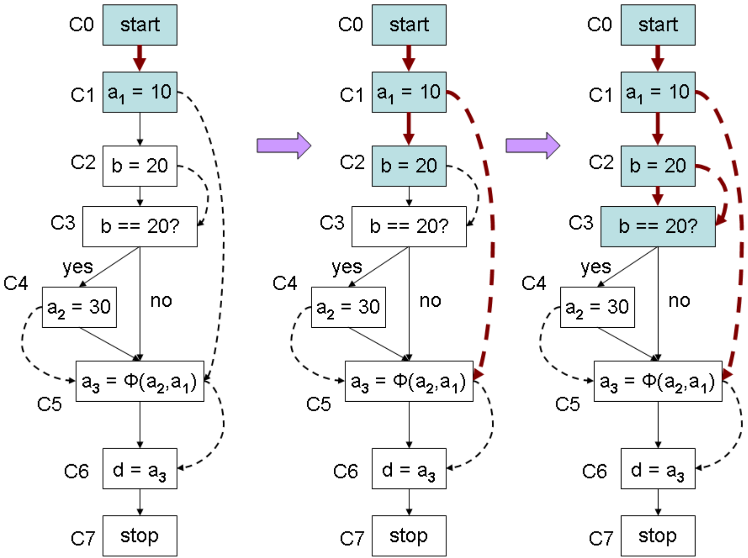
CCP Algorithm - Example - 1



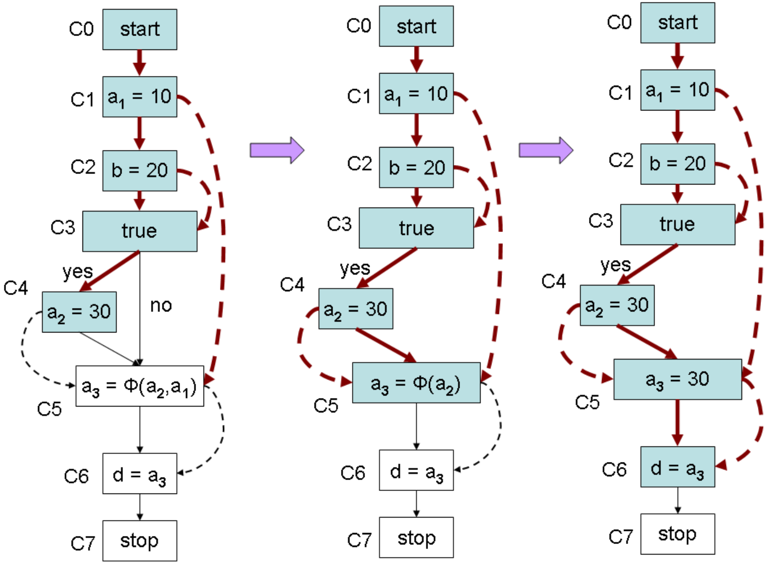
Solid edges are flow edges and dashed edges are SSA edges



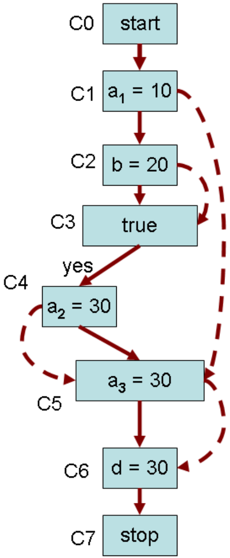
CCP Algorithm - Example 1 - Trace 1



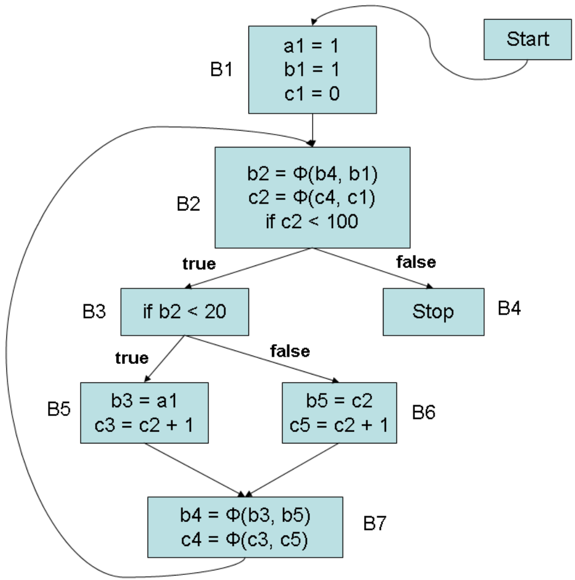
CCP Algorithm - Example 1 - Trace 2



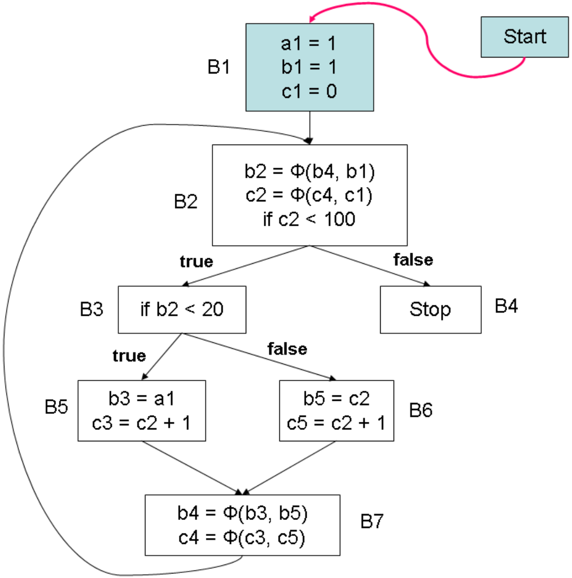
CCP Algorithm - Example 1 - Trace 3



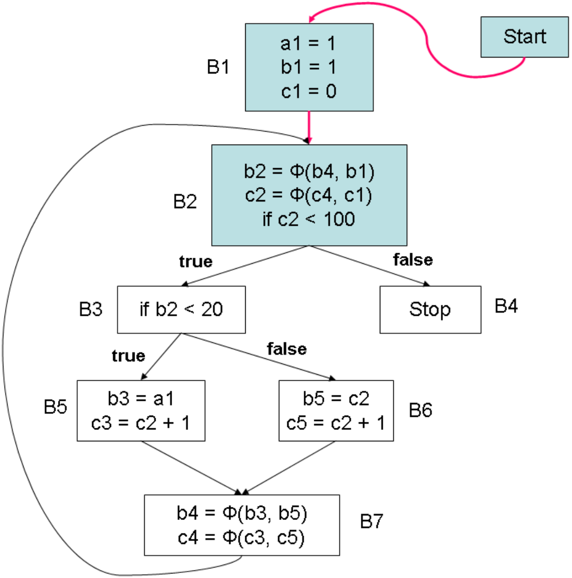
CCP Algorithm - Example 2



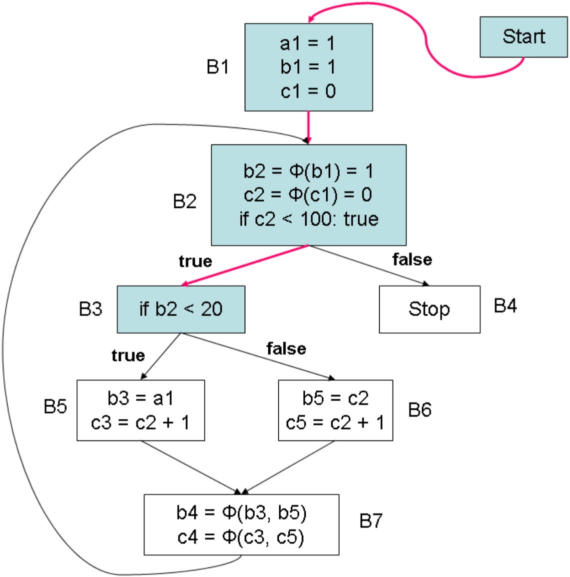
CCP Algorithm - Example 2 - Trace 1



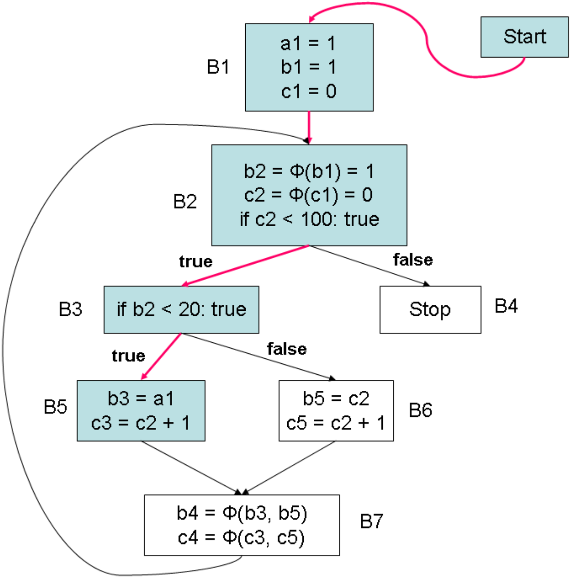
CCP Algorithm - Example 2 - Trace 2



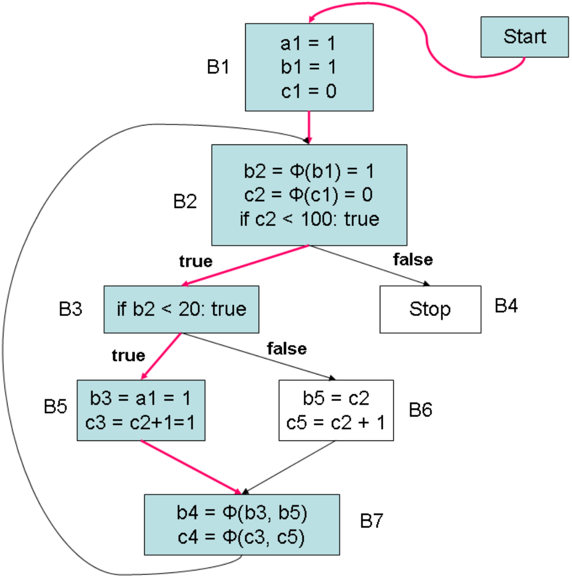
CCP Algorithm - Example 2 - Trace 3



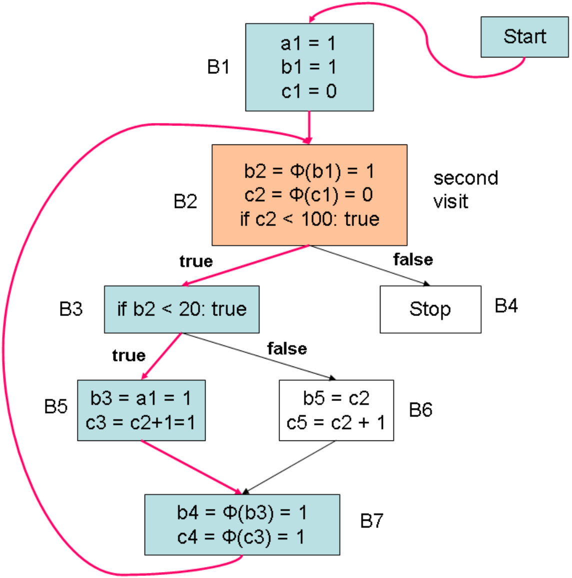
CCP Algorithm - Example 2 - Trace 4



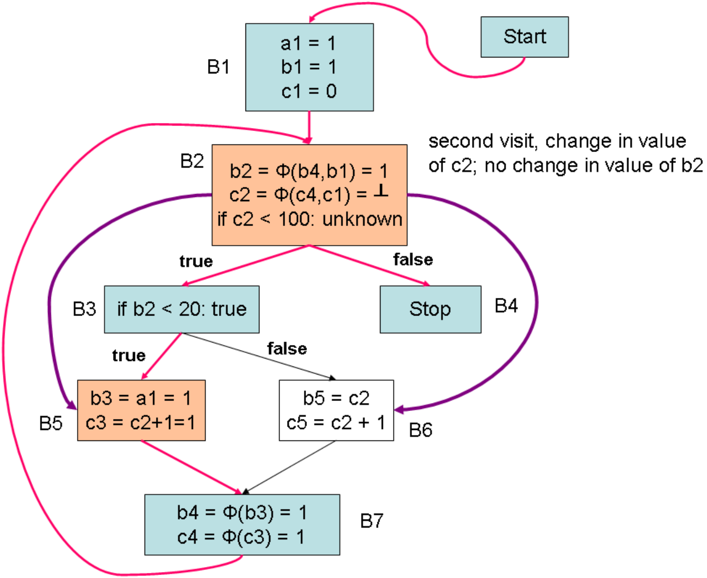
CCP Algorithm - Example 2 - Trace 5



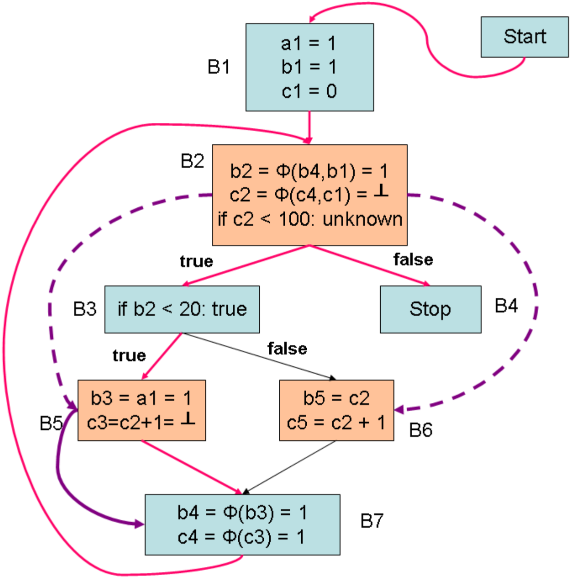
CCP Algorithm - Example 2 - Trace 6



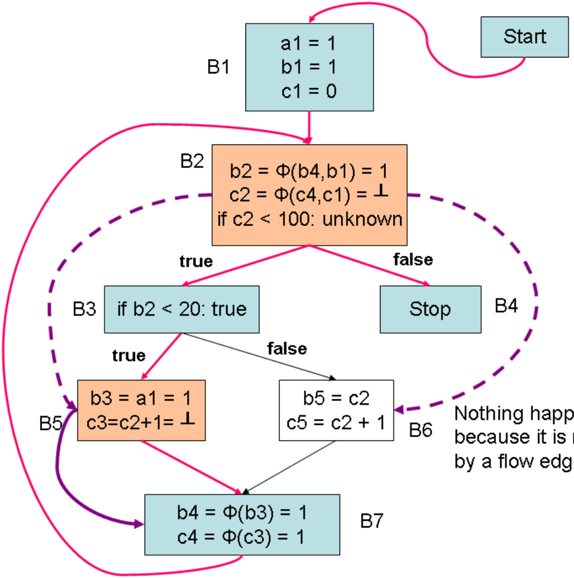
CCP Algorithm - Example 2 - Trace 7



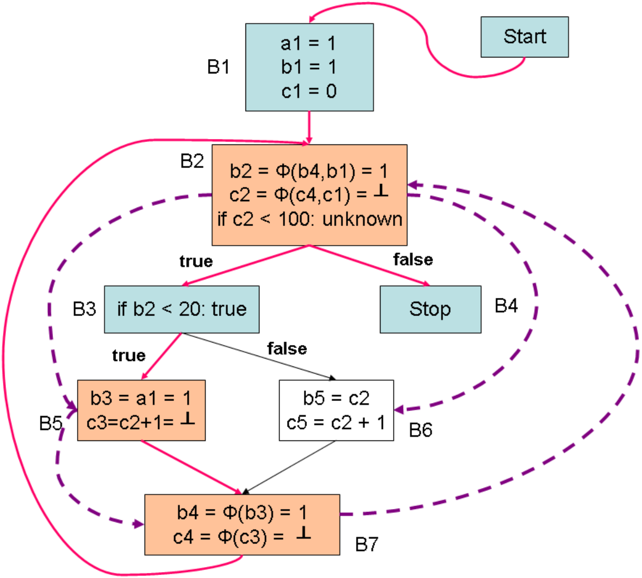
CCP Algorithm - Example 2 - Trace 8



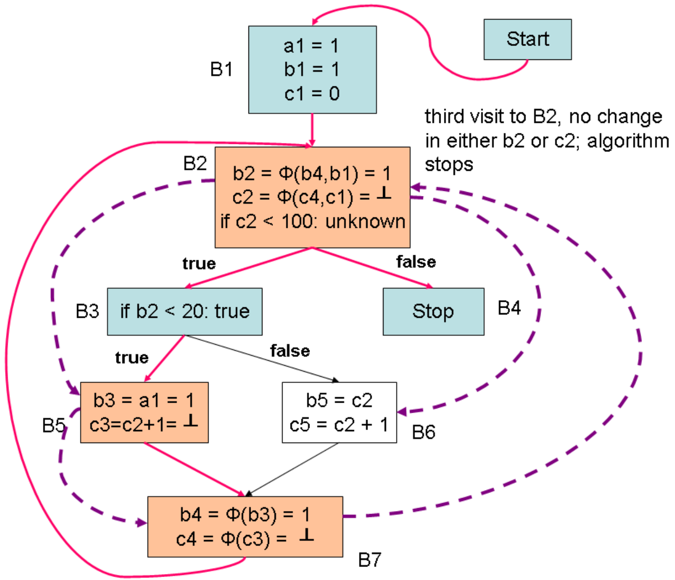
CCP Algorithm - Example 2 - Trace 9



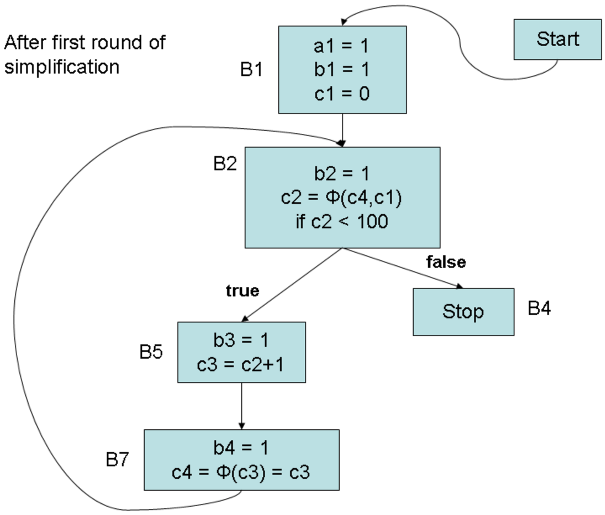
Nothing happens in B6 because it is not reachable by a flow edge


Start
B1
a1 = 1
b1 = 1
c1 = 0
B2
b2 = Φ(b4,b1) = 1
c2 = Φ(c4,c1) = ⊥
if c2 < 100: unknown
true
false
B3
if b2 < 20: true
Stop
B4
true
false
B5
b3 = a1 = 1
c3=c2+1= ⊥
b5 = c2
c5 = c2 + 1
B6
b4 = Φ(b3) = 1
c4 = Φ(c3) = ⊥
B7

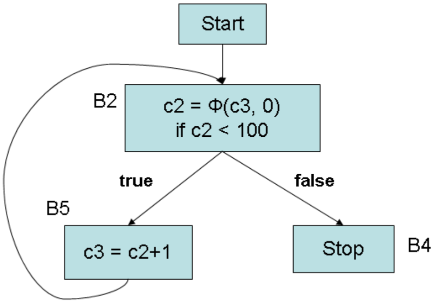
CCP Algorithm - Example 2 - Trace 11



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.