Building SSA Form

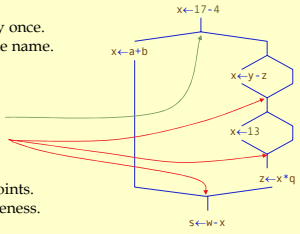
This lecture is primarily based on Konstantinos Sagonas set of slides (**Advanced Compiler Techniques**, (2AD518) at Uppsala University, January-February 2004). Used with kind permission.
(In turn based on Keith Cooper's slides)

What is SSA?

SSA-form:
- Each name is defined exactly once.
- Each use refers to exactly one name.

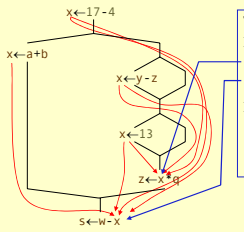
What's hard?
- Straight-line code is trivial.
- Splits in the CFG are trivial.
- Joins in the CFG are hard.

Building SSA Form:
- Insert Φ-functions at birth points.
- Rename all values for uniqueness.



Birth Points (*a notion due to Tarjan*)

Consider the flow of values in this example

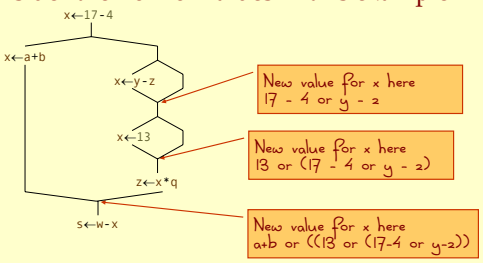


The value x appears everywhere.
It takes on several values.
- Here, x can be 13, y-z, or 17-4.
- Here, it can also be a+b.

If each value has its own name …
- Need a way to merge these distinct values.
- Values are "born" at merge points.

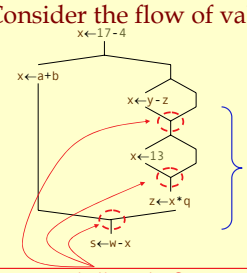
Birth Points (cont)

Consider the flow of values in this example



Birth Points (cont)

Consider the flow of values in this example



- All birth points are join points
- Not all join points are birth points
- Birth points are value-specific …

These are all birth points for values

Static Single Assignment Form

SSA-form:
- Each name is defined exactly once.
- Each use refers to exactly one name.

What's hard?
- Straight-line code is trivial.
- Splits in the CFG are trivial.
- Joins in the CFG are hard.

Building SSA Form:
- Insert Φ-functions at birth points.
- Rename all values for uniqueness.

A Φ-function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.

However, real machines do not implement a Φ-function in hardware.

SSA Construction Algorithm (High-level sketch)

1. Insert Φ-functions.
2. Rename values.

… *that's all …*

… of course, there is some bookkeeping to be done …

SSA Construction Algorithm (Less high-level)

1. Insert Φ-functions at every join for every name.
2. Solve *reaching definitions.*
3. Rename each use to the def that reaches it.
   (*will be unique*)

Reaching Definitions

The equations

REACHES($\mathcal{N}_0$) = Ø

REACHES($\mathcal{N}$) = $\cup_{\mathcal{P} \in preds(\mathcal{N})}$ DEFOUT($\mathcal{P}$) ∪ (REACHES($\mathcal{P}$) ∩ SURVIVED($\mathcal{P}$))

Domain is |DEFINITIONS|, same as number of operations

- REACHES($\mathcal{N}$) is the set of definitions that reach block $\mathcal{N}$
- DEFOUT($\mathcal{N}$) is the set of definitions in $\mathcal{N}$ that reach the end of $\mathcal{N}$
- SURVIVED($\mathcal{N}$) is the set of definitions not obscured by a new def in $\mathcal{N}$

Computing REACHES($\mathcal{N}$)

- Use any data-flow method (*i.e., the iterative method*)
- This particular problem has a very-fast solution (*Zadeck*)

F.K. Zadeck, "Incremental data-flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Conf. on Compiler Construction*, June, 1984, pages 132-143.

SSA Construction Algorithm (Less high-level)

1. Insert Φ-functions at **every join** for **every name**.
2. Solve *reaching definitions.*
3. Rename each use to the def that reaches it. (*will be unique*)

Builds maximal SSA

What's wrong with this approach?

- Too many Φ-functions. (*precision*)
- Too many Φ-functions. (*space*)
- Too many Φ-functions. (*time*)
- Need to relate edges to Φ-functions parameters. (*bookkeeping*)

To do better, we need a more complex approach.

SSA Construction Algorithm (Less high-level)

1. Insert Φ-functions
   a.) calculate dominance frontiers — Moderately complex
   b.) find global names
   for each name, build a list of blocks that define it
   c.) insert Φ-functions — Compute list of blocks where each name is assigned & use as a worklist
   ∀ global name n
   ∀ block $\mathcal{B}$ in which n is defined
   ∀ block $\mathcal{D}$ in $\mathcal{B}$'s dominance frontier
   insert a Φ-function for n in $\mathcal{D}$
   add $\mathcal{D}$ to n's list of defining blocks — This adds to the worklist !

Creates the iterated dominance frontier

Use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same Φ-function twice.

SSA Construction Algorithm (Less high-level)

2. Rename variables in a pre-order walk over dominator tree
   (use an array of stacks, one stack per global name)
   Staring with the root block, $\mathcal{B}$
   a.) generate unique names for each Φ-function — 1 counter per name for subscripts
   and push them on the appropriate stacks
   b.) rewrite each operation in the block
   i. Rewrite uses of global names with the current version
   (from the stack)
   ii. Rewrite definition by inventing & pushing new name — Need the end-of-block name for this path
   c.) fill in Φ-function parameters of successor blocks
   d.) recurse on $\mathcal{B}$'s children in the dominator tree
   e.) <on exit from block $\mathcal{B}$> pop names generated in $\mathcal{B}$ from stacks

Reset the state

Aside on Terminology: Dominators

Definitions

$\mathcal{X}$ dominates $\mathcal{Y}$ if and only if every path from the entry of the control-flow graph to the node for $\mathcal{Y}$ includes $\mathcal{X}$

- By definition, $\mathcal{X}$ dominates $\mathcal{X}$
- We associate a set of dominators (Dom) with each node
- $|\text{Dom}(x)| \geq 1$

Immediate dominators

- For any node $\mathcal{X}$, there must be a $\mathcal{Y}$ in Dom($\mathcal{X}$) closest to $\mathcal{X}$
- We call this $\mathcal{Y}$ the immediate dominator of $\mathcal{X}$
- As a matter of notation, we write this as IDom($\mathcal{X}$)

Dominators (cont)

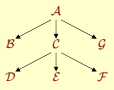
Dominators have many uses in program analysis & transformation:

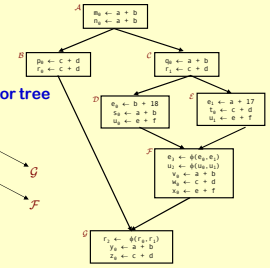
- Finding loops.
- Building SSA form.
- Making code motion decisions.

Dominator sets

Block	Dom	IDom
$\mathcal{A}$	$\mathcal{A}$	–
$\mathcal{B}$	$\mathcal{A},\mathcal{B}$	$\mathcal{A}$
$\mathcal{C}$	$\mathcal{A},\mathcal{C}$	$\mathcal{A}$
$\mathcal{D}$	$\mathcal{A},\mathcal{C},\mathcal{D}$	$\mathcal{C}$
$\mathcal{E}$	$\mathcal{A},\mathcal{C},\mathcal{E}$	$\mathcal{C}$
$\mathcal{F}$	$\mathcal{A},\mathcal{C},\mathcal{F}$	$\mathcal{C}$
$\mathcal{G}$	$\mathcal{A},\mathcal{G}$	$\mathcal{A}$

Dominator tree





Let's look at how to compute dominators...

SSA Construction Algorithm (Low-level detail)

Computing Dominance

- First step in Φ-function insertion computes dominance.
- A node $\mathcal{N}$ dominates $\mathcal{M}$ iff $\mathcal{N}$ is on every path from $\mathcal{N}_0$ to $\mathcal{M}$
  - Every node dominates itself
  - $\mathcal{N}$'s immediate dominator is its closest dominator, IDom($\mathcal{M}$)†

$$\text{Dom}(\mathcal{N}_0) = \{\mathcal{N}_0\}$$

Initially, Dom(n) = $\mathbb{N}$, $\forall\, n \neq n_0$

$$\text{Dom}(\mathcal{M}) = \{\mathcal{M}\} \cup (\cap_{\mathcal{P} \in preds(\mathcal{M})}\ \text{Dom}(\mathcal{P}))$$

Computing DOM

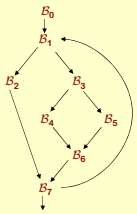
- These equations form a rapid data-flow framework
- Iterative algorithm will solve them in d($\mathbb{G}$) + 3 passes
  - Each pass does $|\mathbb{N}|$ unions & $|\mathbb{E}|$ intersections,
  - $\mathbb{E}$ is $O(\mathbb{N}^2) \Rightarrow O(\mathbb{N}^2)$ work

d(G) is the loop-connectedness of the graph w.r.t a DFST
- Maximal number of back edges in an acyclic path.
- Several studies suggest that, in practice, **d(G)** is small. (<3)
- For most CFGs, **d(G)** is independent of the specific DFST.

†IDom($\mathcal{M}$) ≠ $\mathcal{N}$, unless $\mathcal{N}$ is $\mathcal{N}_0$, by convention.

Example

Control Flow Graph



Progress of iterative solution for DOM

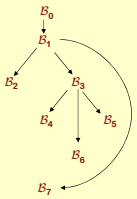
Iter-ation	DOM(n)							
0	0	N	N	N	N	N	N	N
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7

Results of iterative solution for DOM & IDOM

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDOM	0	0	1	1	3	3	3	1

Example

Dominance Tree



Progress of iterative solution for DOM

Iter-ation	DOM(n)							
0	0	N	N	N	N	N	N	N
1	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
2	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7

Results of iterative solution for DOM & IDOM

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDOM	0	0	1	1	3	3	3	1

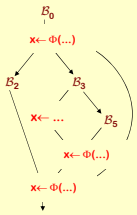
There are asymptotically faster algorithms.

With the right data structures, the iterative algorithm can be made faster.

See Cooper, Harvey, and Kennedy.

Example

Dominance Frontiers



Dominance Frontiers & Φ-Function Insertion

- A definition at $\mathcal{N}$ forces a Φ-function at $\mathcal{M}$ iff $\mathcal{N} \notin$ Dom($\mathcal{M}$) but $\mathcal{N} \in$ Dom($\mathcal{P}$) for some $\mathcal{P} \in preds(\mathcal{M})$
- DF($\mathcal{N}$) is the fringe just beyond the region that $\mathcal{N}$ dominates.

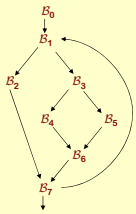
	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	–	7	7	6	6	7	1

- DF($\mathcal{B}_4$) is {$\mathcal{B}_6$}, so ← in $\mathcal{B}_4$ forces a Φ-function in $\mathcal{B}_6$
- ← in $\mathcal{B}_6$ forces a Φ-function in DF($\mathcal{B}_6$) = {$\mathcal{B}_7$}
- ← in $\mathcal{B}_7$ forces a Φ-function in DF($\mathcal{B}_7$) = {$\mathcal{B}_1$}
- ← in $\mathcal{B}_1$ forces a Φ-function in DF($\mathcal{B}_1$) = ∅ (*halt*)

For each assignment, we insert the Φ-functions

Example

Dominance Frontiers



Computing Dominance Frontiers

- Only join points are in DF($\mathcal{N}$) for some $\mathcal{N}$
- Leads to a simple, intuitive algorithm for computing dominance frontiers
  - For each join point $\mathcal{M}$ (i.e., $|preds(\mathcal{M})| > 1$)
  - For each CFG predecessor of $\mathcal{M}$
    - Run up to IDOM($\mathcal{M}$) in the dominator tree, adding $\mathcal{M}$ to DF($\mathcal{N}$) for each $\mathcal{N}$ between $\mathcal{M}$ and IDOM($\mathcal{M}$)

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

- For some applications, we need post-dominance, the post-dominator tree, and reverse dominance frontiers, RDF($\mathcal{N}$)
  - > Just dominance on the reverse CFG
  - > Reverse the edges & add unique exit node
- We will use these in dead code elimination

SSA Construction Algorithm (Reminder)

1. Insert Φ-functions at every join for every name
   a.) calculate dominance frontiers
   b.) find global names — Needs a little more detail
   for each name, build a list of blocks that define it
   c.) insert Φ-functions

∀ global name n
  ∀ block $\mathcal{B}$ in which n is defined
    ∀ block $\mathcal{D}$ in $\mathcal{B}$'s dominance frontier
      insert a Φ-function for n in $\mathcal{D}$
      add $\mathcal{D}$ to n's list of defining blocks

SSA Construction Algorithm

Finding global names

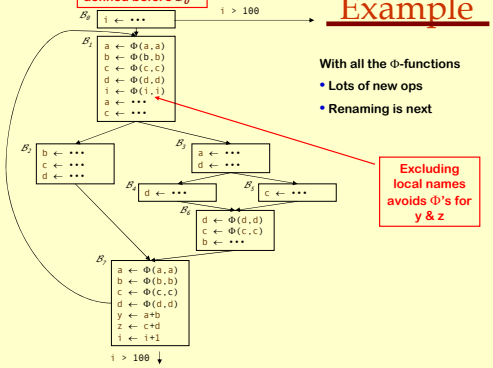
- Different between two forms of SSA
- Minimal uses all names
- Semi-pruned SSA uses names that are *live* on entry to some block
  - Shrinks name space & number of Φ-functions
  - Pays for itself in compile-time speed
- For each "global name", need a list of blocks where it is defined
  - Drives Φ-function insertion
  - $\mathcal{B}$ defines x implies a Φ-function for x in every $\mathcal{C} \in$ DF($\mathcal{B}$)

Otherwise, we do not need a Φ-function

Pruned SSA adds a test to see if x is live at insertion point

Assume a, b, c, & d defined before $\mathcal{B}_0$

Example



With all the Φ-functions
- Lots of new ops
- Renaming is next

Excluding local names avoids Φ's for y & z

SSA Construction Algorithm (Less high-level)

2. Rename variables in a pre-order walk over dominator tree
   (use an array of stacks, one stack per global name)
   Staring with the root block, $\mathcal{B}$
   a.) generate unique names for each Φ-function
   and push them on the appropriate stacks
   b.) rewrite each operation in the block
   i. Rewrite uses of global names with the current version (from the stack)
   ii. Rewrite definition by inventing & pushing new name
   c.) fill in Φ-function parameters of successor blocks
   d.) recurse on $\mathcal{B}$'s children in the dominator tree
   e.) <on exit from block $\mathcal{B}$> pop names generated in $\mathcal{B}$ from stacks

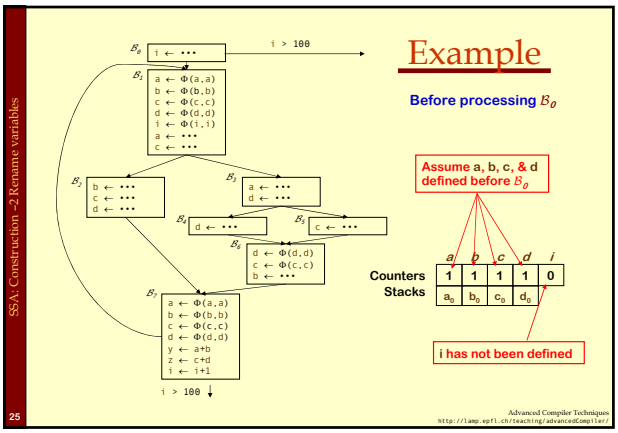
SSA Construction Algorithm (Less high-level)

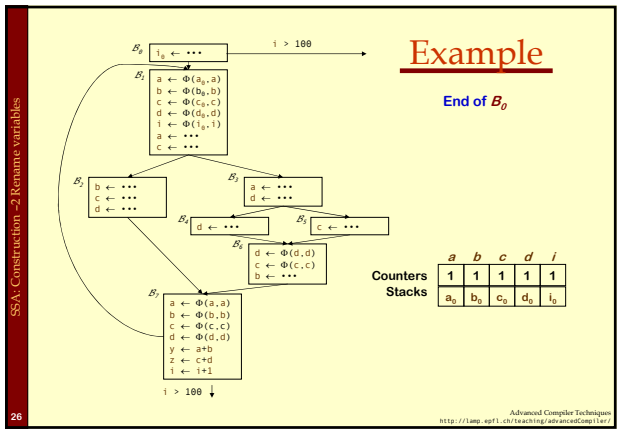
Adding all the details ...

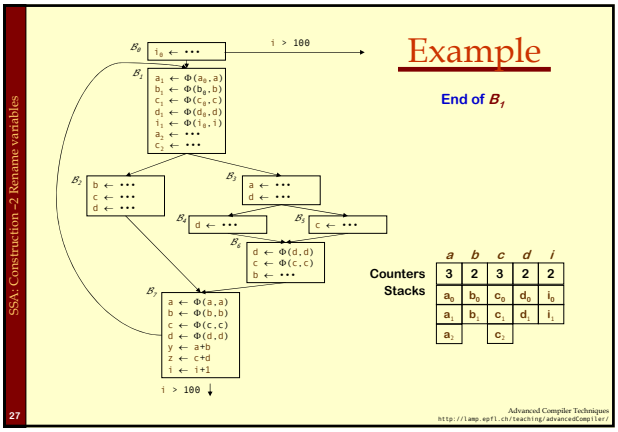
```
for each global name i
    counter[i] ← 0
    stack[i] ← Ø
call Rename(B0)
```

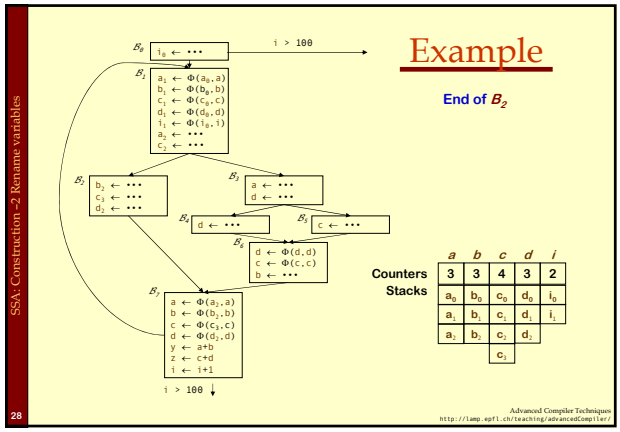
```
NewName(v)
    i ← counter[v]
    counter[v] ← counter[v] + 1
    push vi onto stack[v]
    return vi
```

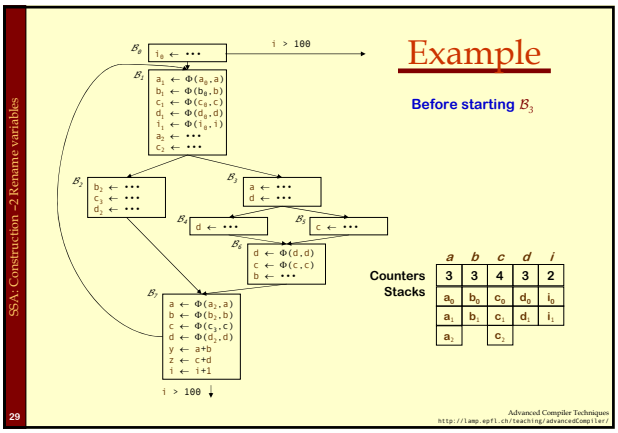
```
Rename(B)
    for each Φ-function in B, x ← Φ(…)
        rename x as NewName(x)
    for each operation "x←y op z" in B
        rewrite y as top(stack[y])
        rewrite z as top(stack[z])
        rewrite x as NewName(x)
    for each successor of B in the CFG
        rewrite appropriate Φ parameters
    for each successor S of B in dom. tree
        Rename(S)
    for each operation "x←y op z" in B
        pop(stack[x])
```

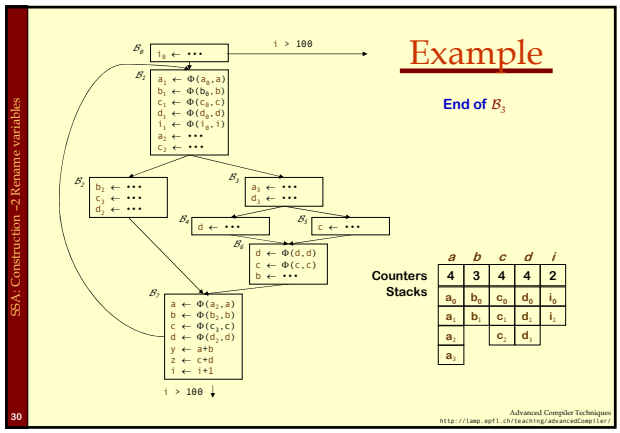

SSA: Construction –2 Rename variables
Example
Before processing B_0
Assume a, b, c, & d defined before B_0
Counters
Stacks
i has not been defined
25
Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/

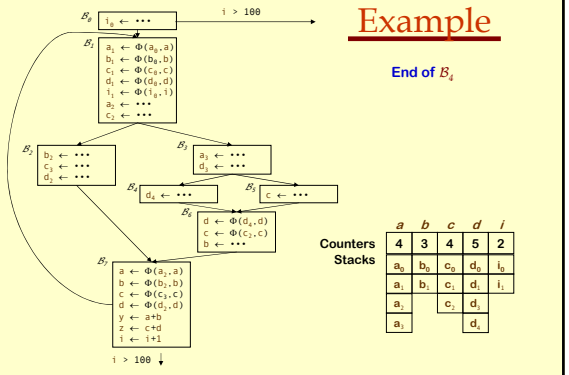

SSA: Construction –2 Rename variables
Example
End of B_0
Counters
Stacks
26
Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/


SSA: Construction –2 Rename variables
Example
End of B_1
Counters
Stacks
27
Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/


SSA: Construction –2 Rename variables
Example
End of B_2
Counters
Stacks
28
Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/


SSA: Construction –2 Rename variables
Example
Before starting B_3
Counters
Stacks
29
Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/

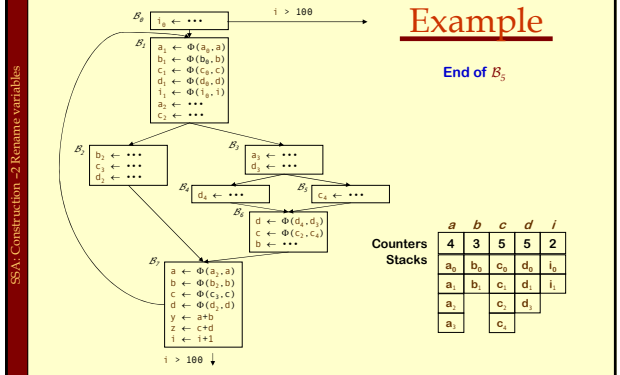

SSA: Construction –2 Rename variables
Example
End of B_3
Counters
Stacks
30
Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



Example

End of B_4

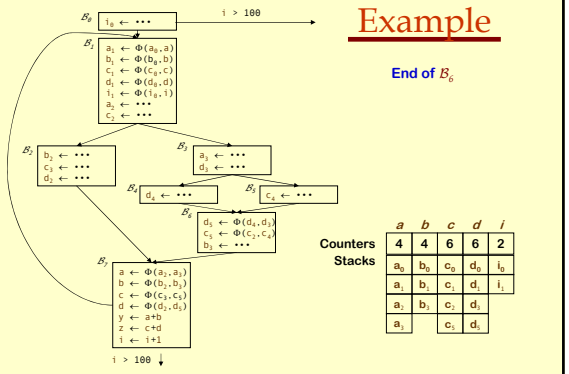
Counters	a	b	c	d	i
	4	3	4	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3			d_4	



Example

End of B_5

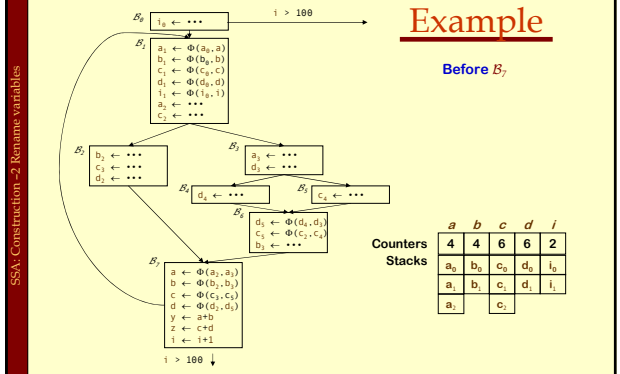
Counters	a	b	c	d	i
	4	3	5	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3		c_4		



Example

End of B_6

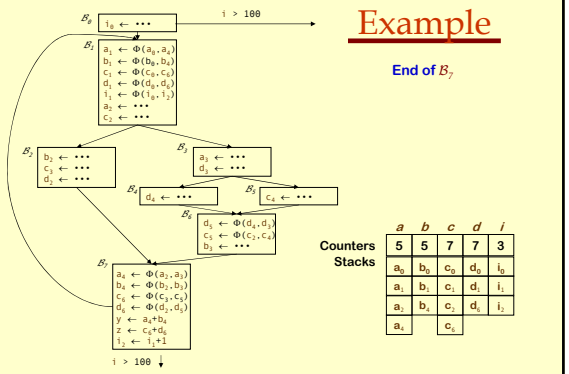
Counters	a	b	c	d	i
	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_3	
	a_3		c_5	d_5	



Example

Before B_7

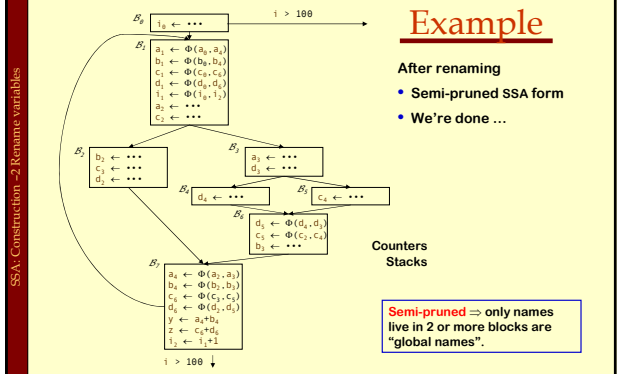
Counters	a	b	c	d	i
	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		



Example

End of B_7

Counters	a	b	c	d	i
	5	5	7	7	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_6	i_2
	a_4		c_6		



Example

After renaming

- Semi-pruned SSA form
- We're done …

Counters
Stacks

Semi-pruned ⇒ only names live in 2 or more blocks are "global names".

SSA Construction Algorithm (Pruned SSA)

What's this "pruned SSA" stuff?
- Minimal SSA still contains extraneous Φ-functions.
- Inserts some Φ-functions where they are dead.
- Would like to avoid inserting them.

Two ideas
- *Semi-pruned SSA:* discard names used in only one block.
  - Significant reduction in total number of Φ-functions.
  - Needs only local liveness information. (*cheap to compute*)
- *Pruned SSA:* only insert Φ-functions where their value is live.
  - Inserts even fewer Φ-functions, but costs more to do.
  - Requires global live variable analysis. (*more expensive*)

In practice, both are simple modifications to step 1.

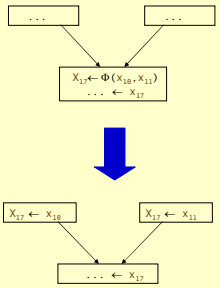
SSA Construction Algorithm

We can improve the stack management.
- Push at most one name per stack per block. (save push & pop)
- Thread names together by block.
- To pop names for block $\mathcal{B}$, use $\mathcal{B}$'s thread.

This is a good use for a scoped hash table.
- Significant reductions in pops and pushes.
- Makes a minor difference in SSA construction time.
- Scoped table is a clean, clear way to handle the problem.

SSA Deconstruction

At some point, we need executable code.
- Few machines implement Φ operations.
- Need to fix up the flow of values.



Basic idea.
- Insert copies Φ-function pred's.
- Simple algorithm.
  - Works in most cases.
- Adds lots of copies.
  - Most of them coalesce away.