15-745

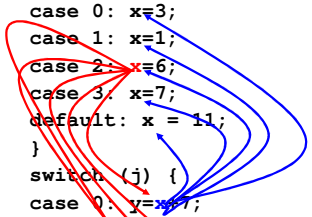
SSA

Def-Use chains are expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3;break;
  case 1: x=1; break;
  case 2: x=6; break;
  case 3: x=7; break;
  default: x = 11;
  }
  switch (j) {
  case 0: y=x+7; break;
  case 1: y=x+4; break;
  case 2: y=x-2; break;
  case 3: y=x+1; break;
  default: y=x+9;
  }
  …
```


Def-Use chains are expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3;
  case 1: x=1;
  case 2: x=6;
  case 3: x=7;
  default: x = 11;
  }
  switch (j) {
  case 0: y=x+7;
  case 1: y=x+4;
  case 2: y=x-2;
  case 3: y=x+1;
  default: y=x+9;
  }
  …
```



In general,
N defs
M uses
⇒ O(NM) space and time

A solution is to limit each var to ONE def site

Def-Use chains are expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3; break;
  case 1: x=1; break;
  case 2: x=6;
  case 3: x=7;
  default: x = 11;
  }
  x1 is one of the above x's
  switch (j) {
  case 0: y=x1+7;
  case 1: y=x1+4;
  case 2: y=x1-2;
  case 3: y=x1+1;
  default: y=x1+9;
  }
  …
```

A solution is to limit each var to ONE def site

Def-Use chains are expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3; break;
  case 1: x=1; break;
  case 2: x=6;
  case 3: x=7;
  default: x = 11;
  }
  x1 = x;
  switch (j) {
  case 0: y=x1+7;
  case 1: y=x1+4;
  case 2: y=x1-2;
  case 3: y=x1+1;
  default: y=x1+9;
  }
```

A solution is to limit each var to ONE def site

Advantages of SSA

- Makes du-chains explicit (and what of ud-chains?)
- Makes dataflow analysis easier
- Improves register allocation
  - Automatically builds Webs
  - Makes building interference graphs easier
- For most programs, reduces space/time requirements

Muchnick: "Thus, it is valuable to be able to translate a given representation of a procedure into SSA form, to operate on it and, when appropriate, to translate it back into the original form." Hmmm….?

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)

Straight-line SSA

```
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```

Straight-line SSA

```
a ← x + y          a1 ← x + y
b ← a + x          b1 ← a1 + x
a ← b + 2    ⇒     a2 ← b1 + 2
c ← y + 1          c1 ← y + 1
a ← c + a          a3 ← c1 + a2
```

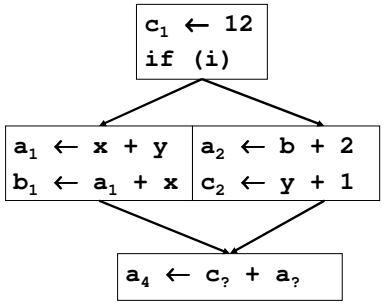

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?

Merging at Joins

```
c ← 12
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
a ← c + a
```

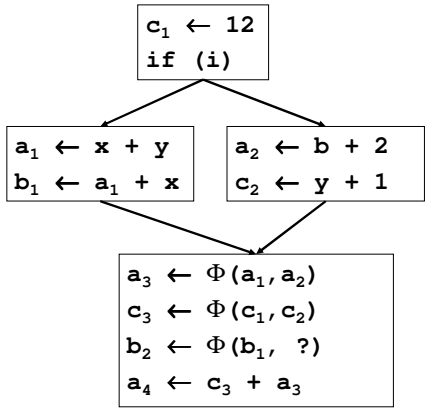
⇒



SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?
  - Use a notional fiction: A Φ function

Merging at Joins



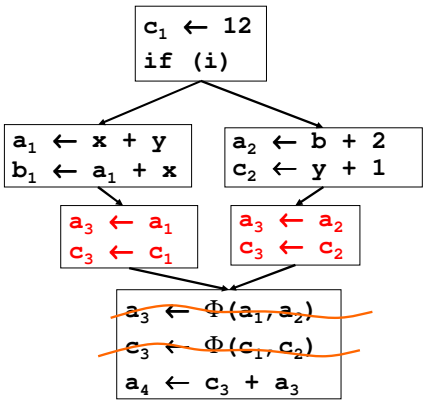
The Φ function

- Φ merges multiple definitions along multiple control paths into a single definition.
- At a BB with p predecessors, there are p arguments to the Φ function.

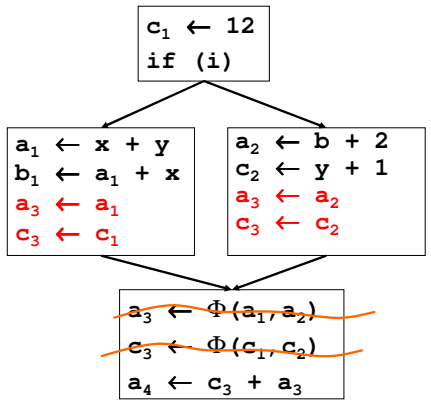
  $x_{new} \leftarrow \Phi(x_1, x_2, x_3, \ldots, x_p)$

- How do we choose which x_i to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

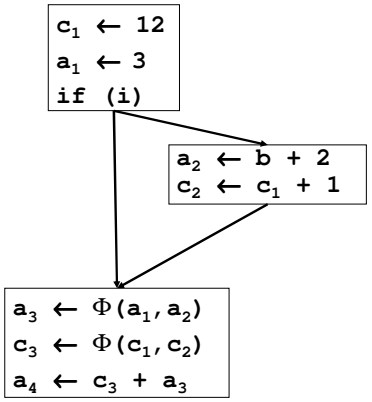
"Implementing" Φ



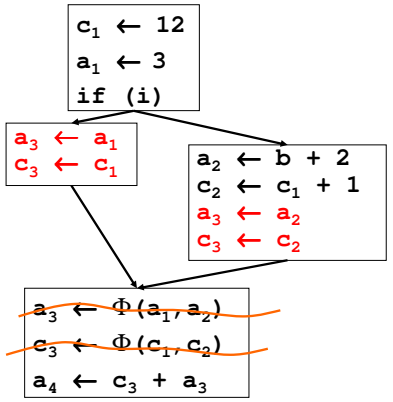
"Implementing" Φ



"Implementing" Φ: Example 2

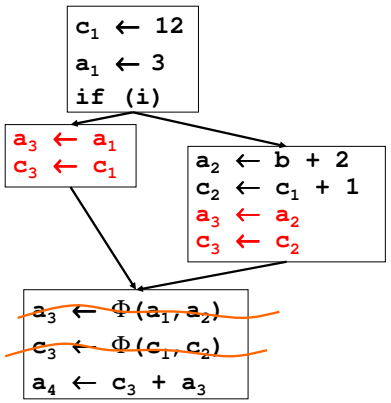


"Implementing" Φ: Example 2



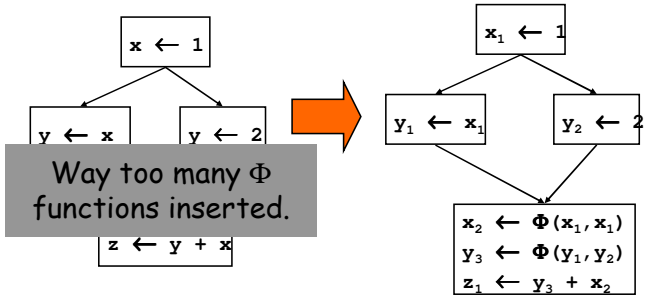
"Implementing" Φ: Example 2

- Sometimes keep moves on edges
- …we'll come back to this later



Trivial SSA

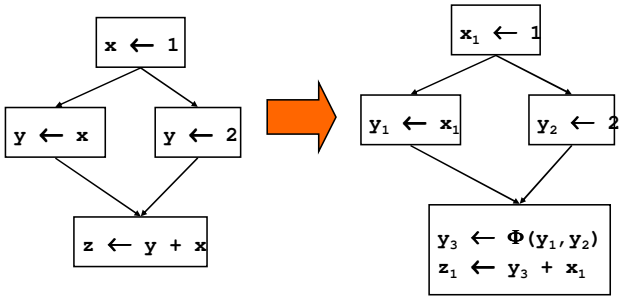
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables.



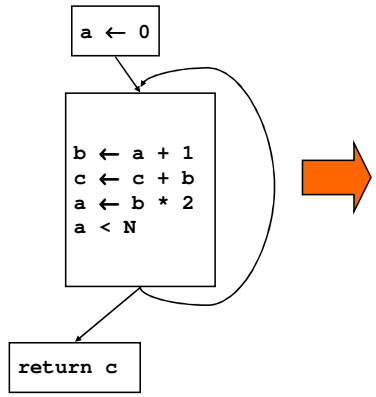
Way too many Φ functions inserted.

Minimal SSA

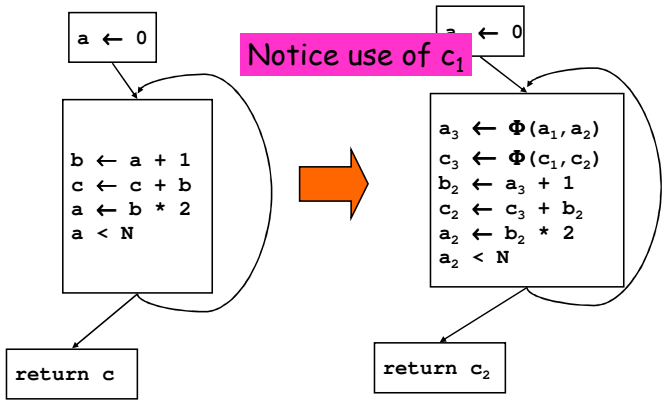
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



Another Example

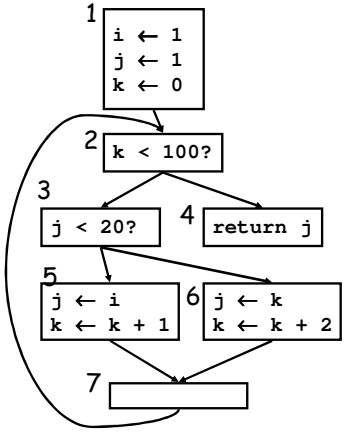


Another Example

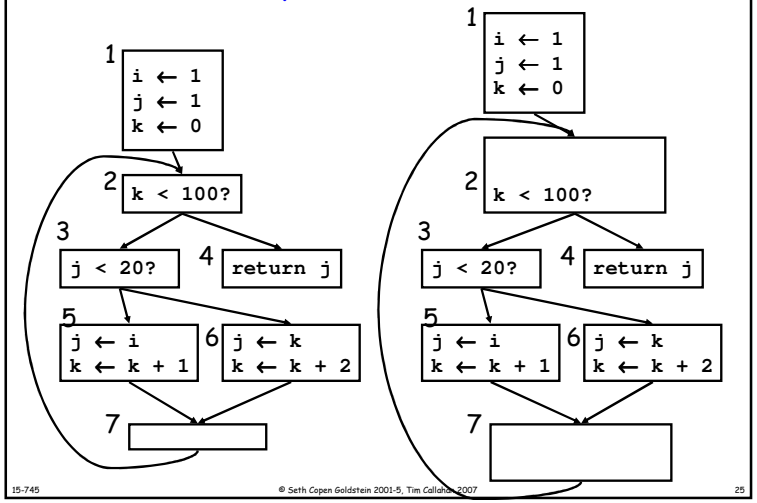


Let's optimize the following:

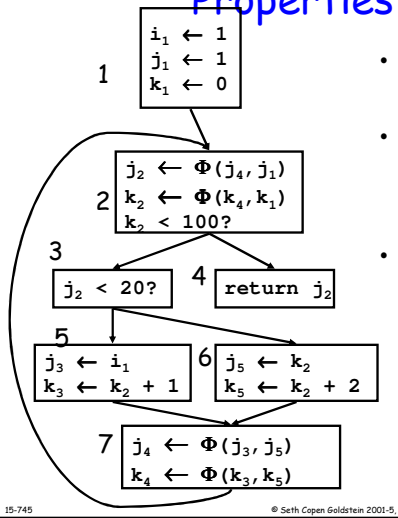
```
i=1;
j=1;
k=0;
while (k<100) {
  if (j<20) {
      j=i;
      k++;
  } else {
      j=k;
      k+=2;
  }
}
return j;
```



First, turn into SSA



Properties of SSA



- Only 1 assignment per variable
- definitions dominate uses
- Can we use this to help with constant propagation?

Constant Propagation

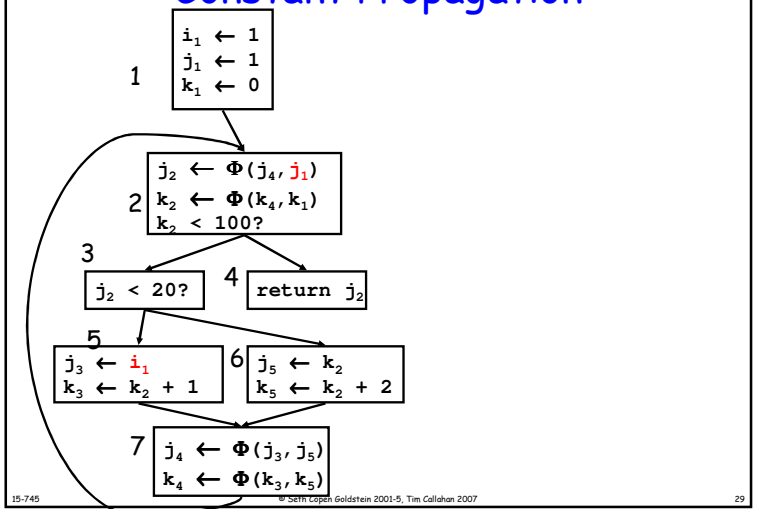
- If "v ← c", replace all uses of v with c
- If "v ← **Φ(c, c, c)**" replace all uses of v with c

```
W <- list of all defs
while !W.isEmpty {
        Stmt S <- W.removeOne
        if S has form "v <- Φ(c,…,c)"
          replace S with V <- c
        if S has form "v <- c" then
          delete S
          foreach stmt U that uses v,
            replace v with c in U
            W.add(U)
}
```

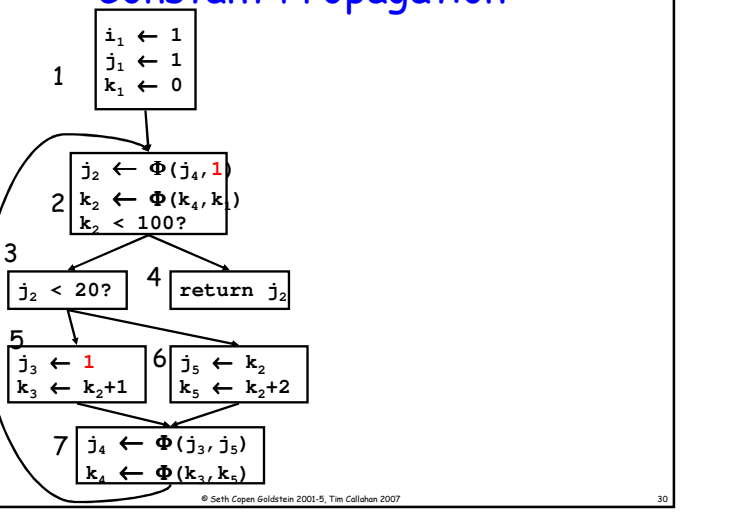

Other stuff we can do?

- Copy propagation
  delete "x ← Φ(y)" and replace all x with y
  delete "x ← y" and replace all x with y
- Constant Folding
  (Also, constant conditions too!)
- Unreachable Code
  Remember to delete all edges from unreachable block

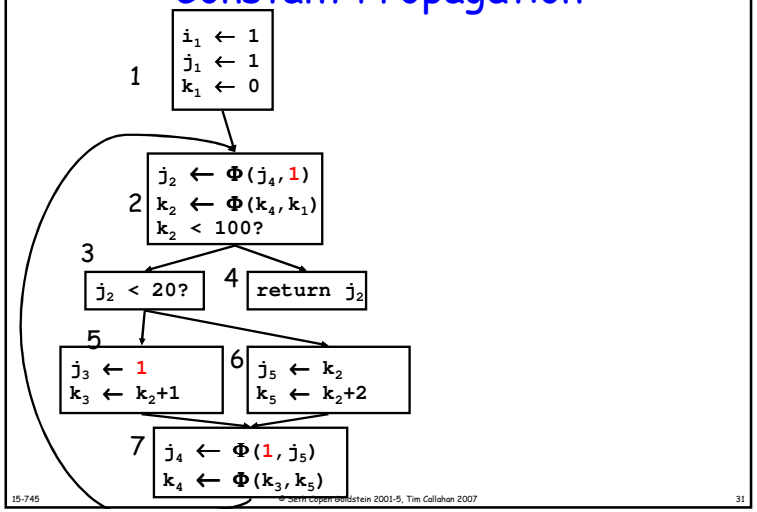
Constant Propagation



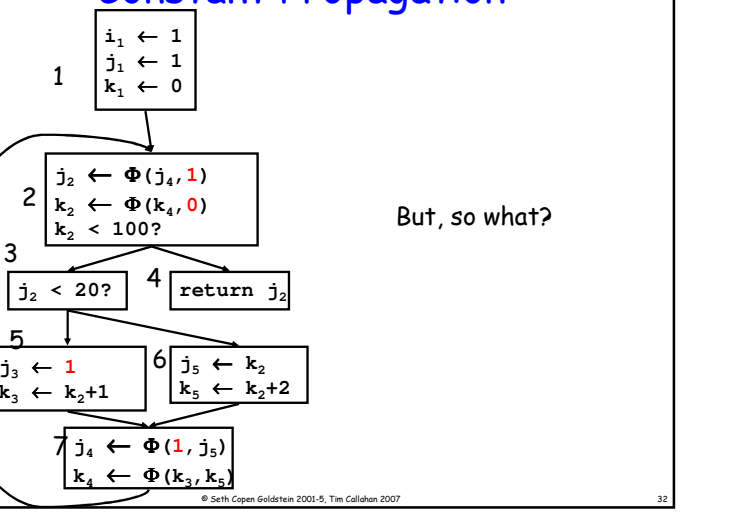
Constant Propagation



Constant Propagation

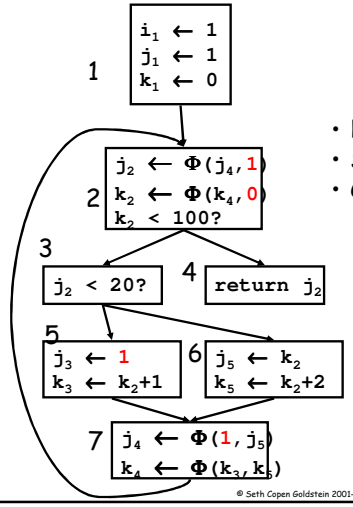


Constant Propagation



But, so what?

Conditional Constant Propagation



- Does block 6 ever execute?
- Simple CP can't tell
- CCP can tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes Values are constants until proven otherwise

Tracks:

- Blocks (assume unexecuted until proven otherwise)
- Variables (assume not executed, only with proof of assignments of a non-constant value do we assume not constant)

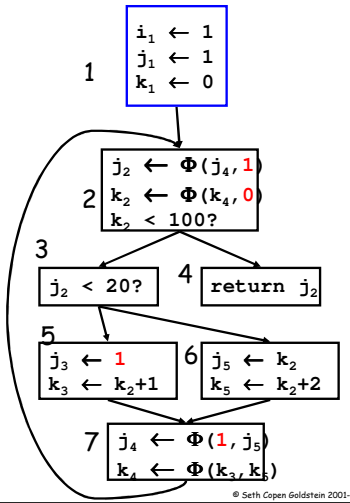
Use a lattice for variables:

BOT = we have evidence that variable can hold different values at different times

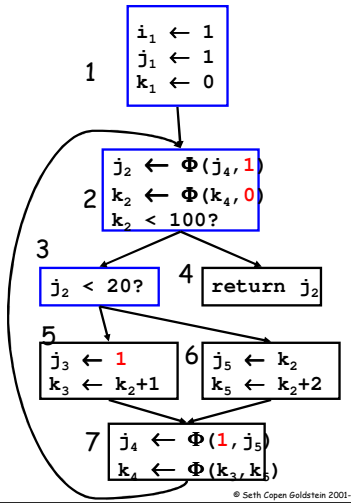
integers = we have seen evidence that the var has been assigned a constant with the value

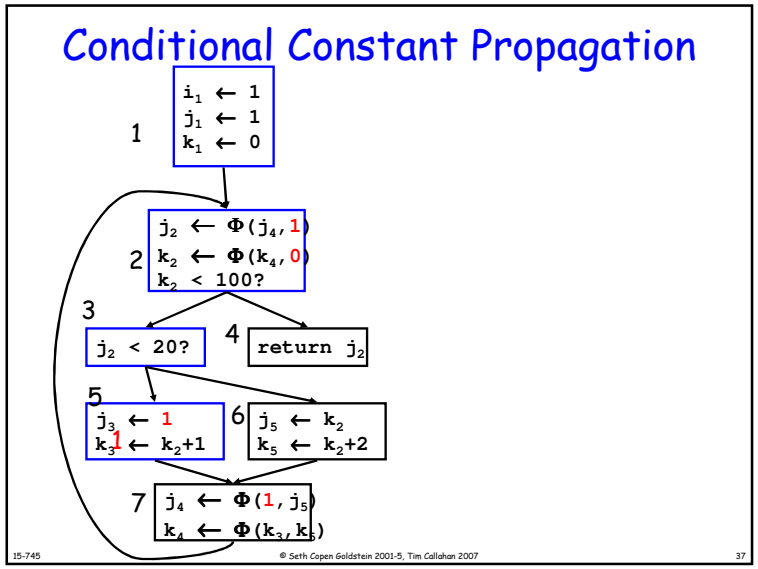
TOP = not known to be executed

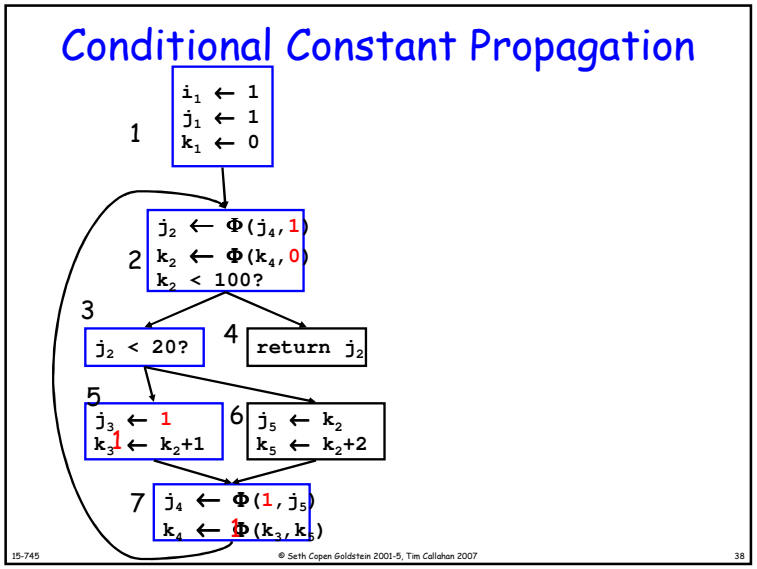
Conditional Constant Propagation

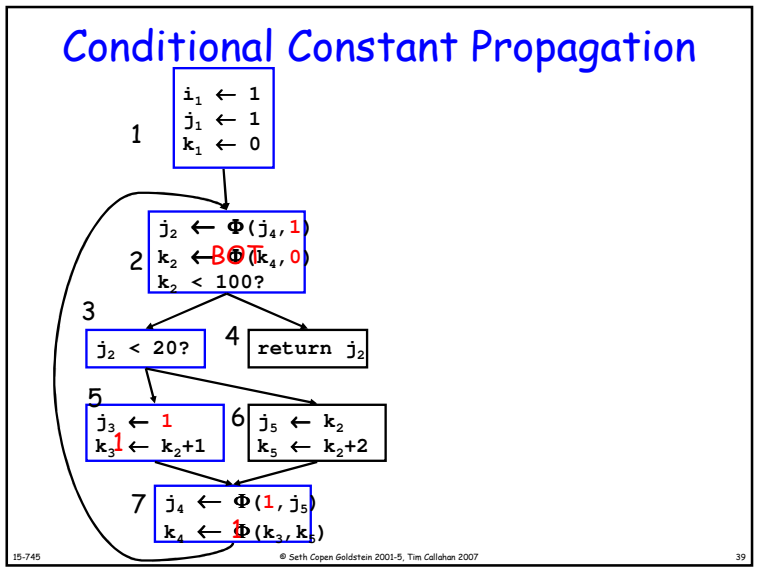


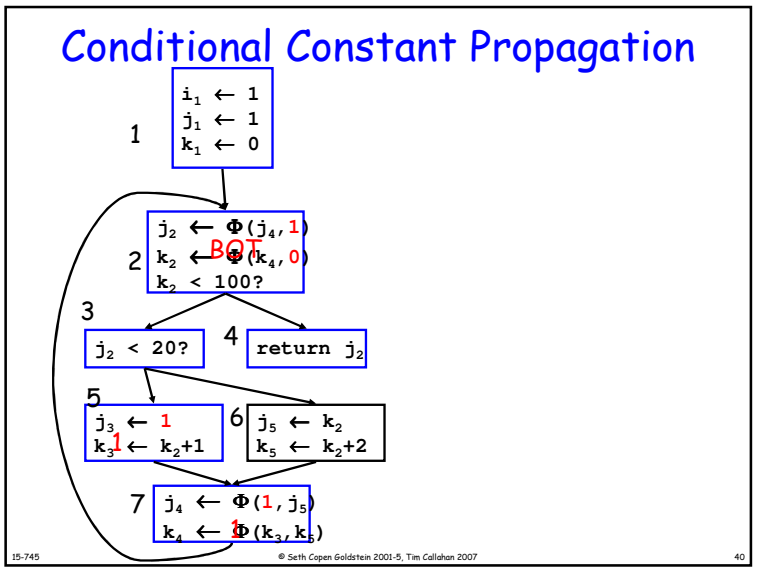
Conditional Constant Propagation



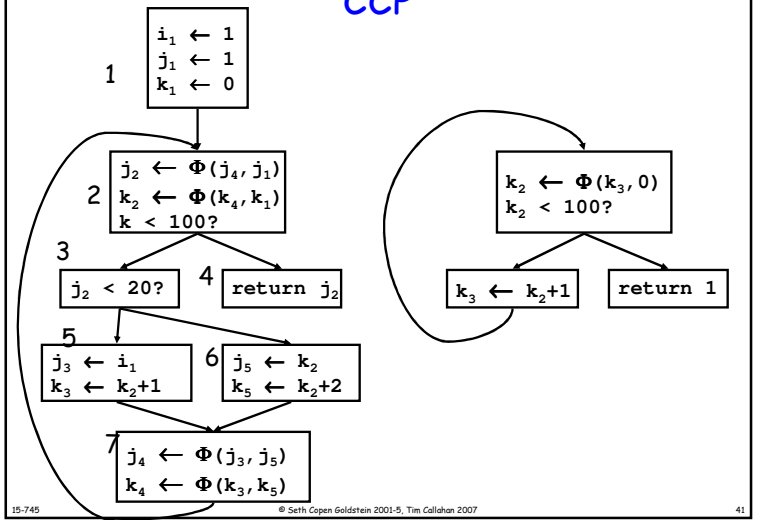

Conditional Constant Propagation
1
i1 ← 1
j1 ← 1
k1 ← 0
2
j2 ← Φ(j4,1)
k2 ← Φ(k4,0)
k2 < 100?
3
j2 < 20?
4
return j2
5
j3 ← 1
k3 ← k2+1
6
j5 ← k2
k5 ← k2+2
7
j4 ← Φ(1,j5)
k4 ← Φ(k3,k5)
15-745
© Seth Copen Goldstein 2001-5, Tim Callahan 2007
37


Conditional Constant Propagation
1
i1 ← 1
j1 ← 1
k1 ← 0
2
j2 ← Φ(j4,1)
k2 ← Φ(k4,0)
k2 < 100?
3
j2 < 20?
4
return j2
5
j3 ← 1
k3 ← k2+1
6
j5 ← k2
k5 ← k2+2
7
j4 ← Φ(1,j5)
k4 ← Φ(k3,k5)
15-745
© Seth Copen Goldstein 2001-5, Tim Callahan 2007
38


Conditional Constant Propagation
1
i1 ← 1
j1 ← 1
k1 ← 0
2
j2 ← Φ(j4,1)
k2 ← Φ(k4,0)
k2 < 100?
3
j2 < 20?
4
return j2
5
j3 ← 1
k3 ← k2+1
6
j5 ← k2
k5 ← k2+2
7
j4 ← Φ(1,j5)
k4 ← Φ(k3,k5)
15-745
© Seth Copen Goldstein 2001-5, Tim Callahan 2007
39


Conditional Constant Propagation
1
i1 ← 1
j1 ← 1
k1 ← 0
2
j2 ← Φ(j4,1)
k2 ← Φ(k4,0)
k2 < 100?
3
j2 < 20?
4
return j2
5
j3 ← 1
k3 ← k2+1
6
j5 ← k2
k5 ← k2+2
7
j4 ← Φ(1,j5)
k4 ← Φ(k3,k5)
15-745
© Seth Copen Goldstein 2001-5, Tim Callahan 2007
40

CCP

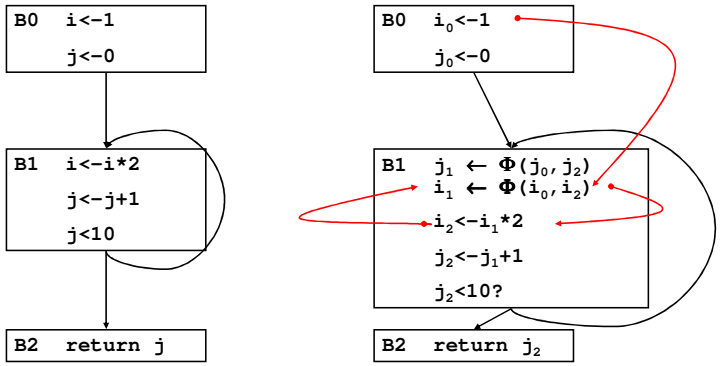


Dead Code Elimination

Since we are using SSA, this is just a list of all variable assignments.

```
W <- list of all defs
while !W.isEmpty {
        Stmt S <- W.removeOne
        if |S.users| != 0 then continue
        if S.hasSideEffects() then continue
        foreach def in S.definers {
          def.users <- def.users - {S}
          if |def.uses| == 0 then
            W <- W UNION {def}
        }
        delete S
}
```


Example DCE



Standard DCE leaves Zombies!

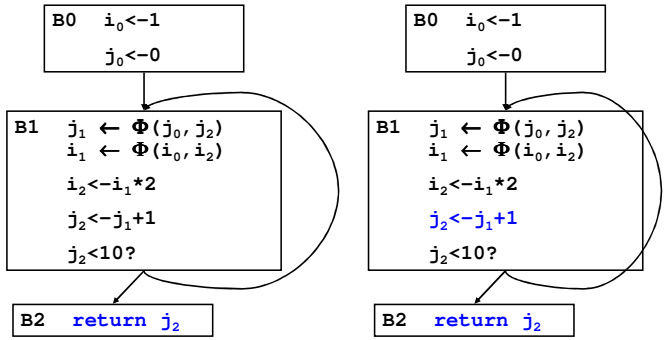
Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

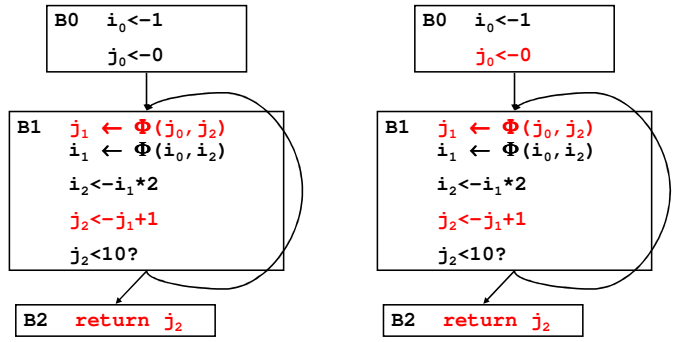
```
init:
   mark as live all stmts that have side-effects:
       - I/O
       - stores into memory
       - returns
       - calls a function that MIGHT have side-effects
   As we mark S live, insert S.defs into W

while (|W| > 0) {
   S <- W.removeOne()
   if (S is live) continue;
   mark S live, insert S.defs into W
}
```


Example DCE



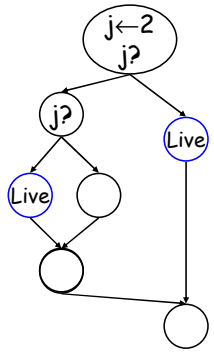
Example DCE



Problem!

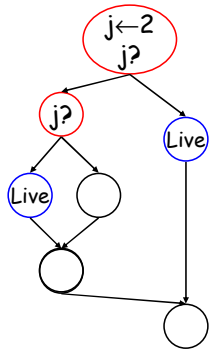
Fixing DCE

If S is live, then
If T determines if S can execute, T should be live



Fixing DCE

If S is live, then
If T determines if S can execute, T should be live

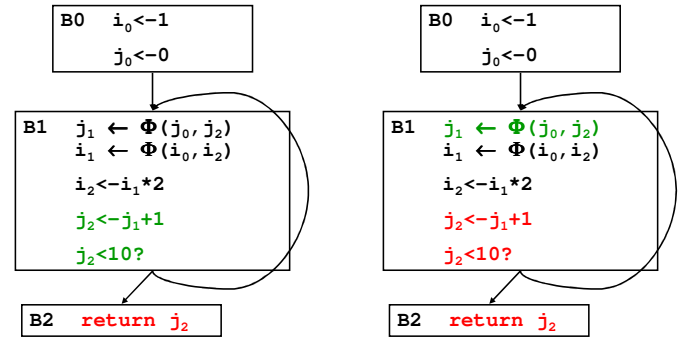


Aggressive Dead Code Elimination

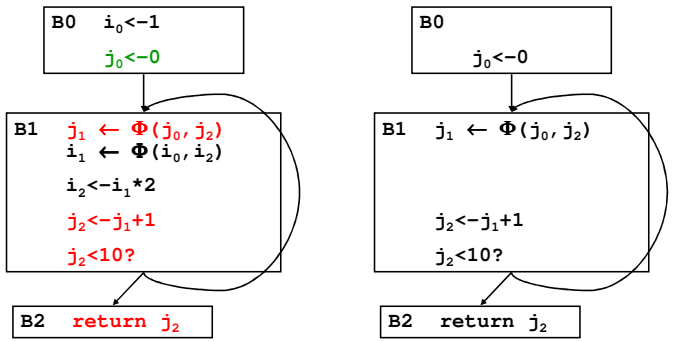
Assume a stmt is dead until proven otherwise.

```
while (|W| > 0) {
   S <- W.removeOne()
   if (S is live) continue;
   mark S live, insert
       - forall operands, S.operand.definers into W
       - S.CD^-1 into W
}
```

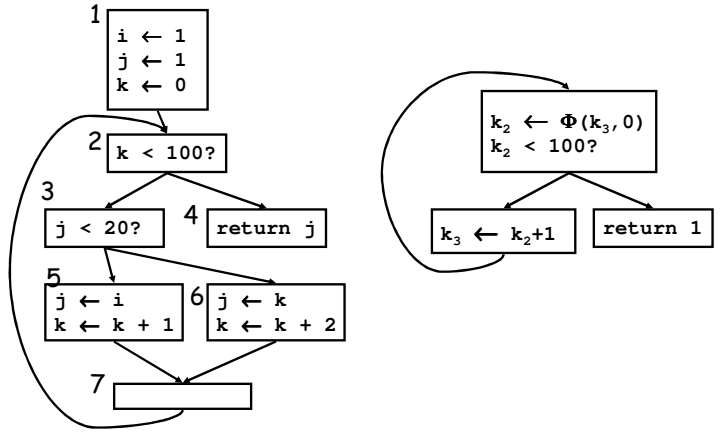

Example DCE



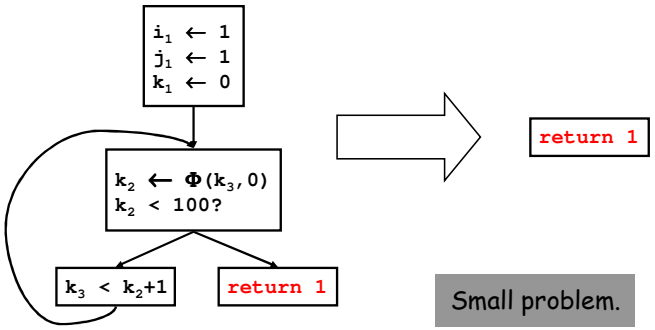
Example DCE



CCP Example



CCP -> DCE



Small problem.

Extending SSA -> Pegasus

- SUIF is the front-end
- standard transformation
- some optimizations

- Then, to graph-based SSA

- Once in SSA form, why leave?

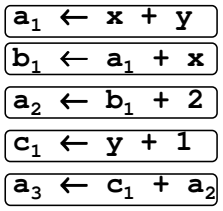
Straight-line SSA

$$
\begin{array}{l}
a \leftarrow x + y \\
b \leftarrow a + x \\
a \leftarrow b + 2 \\
c \leftarrow y + 1 \\
a \leftarrow c + a
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
a_1 \leftarrow x + y \\
b_1 \leftarrow a_1 + x \\
a_2 \leftarrow b_1 + 2 \\
c_1 \leftarrow y + 1 \\
a_3 \leftarrow c_1 + a_2
\end{array}
$$

Straight-line SSA to Pegasus

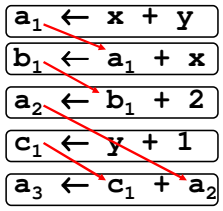
$$
\begin{array}{l}
a_1 \leftarrow x + y \\
b_1 \leftarrow a_1 + x \\
a_2 \leftarrow b_1 + 2 \\
c_1 \leftarrow y + 1 \\
a_3 \leftarrow c_1 + a_2
\end{array}
$$

Straight-line SSA to Pegasus



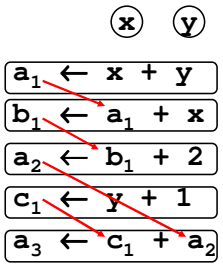
Put a circly-box around each statement

Straight-line SSA to Pegasus



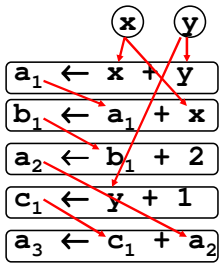
Draw the explicit def-use edges

Straight-line SSA to Pegasus



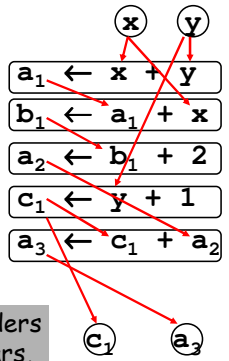
Make placeholders for input vars

Straight-line SSA to Pegasus



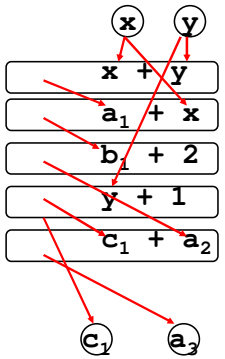
Make placeholders for input vars… and connect them

Straight-line SSA to Pegasus



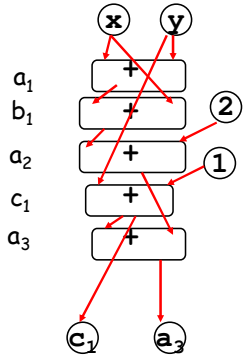
Make placeholders for live-out vars, and connect

Straight-line SSA to Pegasus



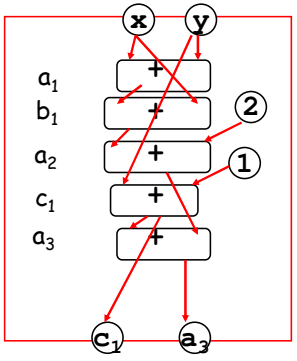
Don't need destinations any more…

Straight-line SSA to Pegasus



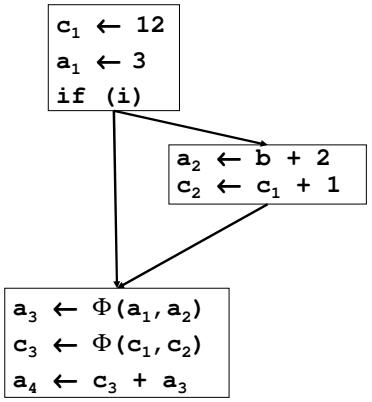
Actually, only need the operation

Straight-line SSA to Pegasus

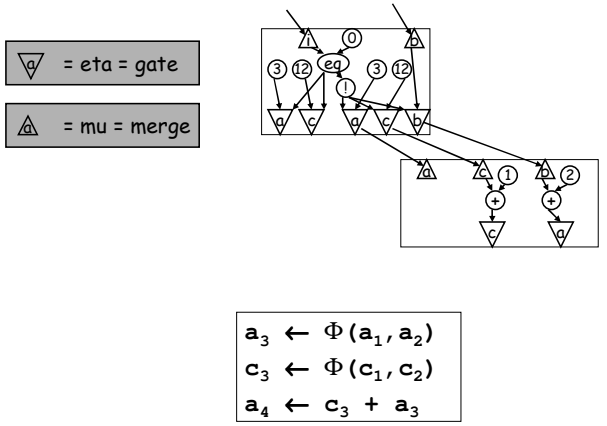


So that's our basic block… but how do they connect?

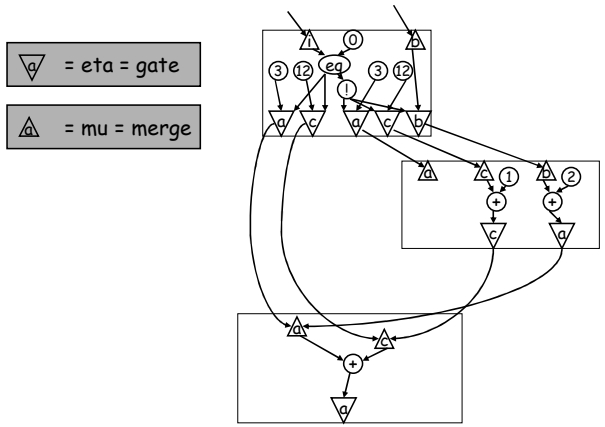
"Implementing" Φ: Example 2



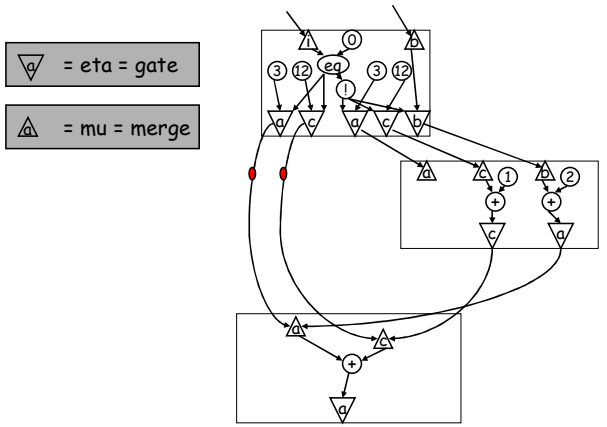
"Implementing" Φ: Example 2



"Implementing" Φ: Example 2



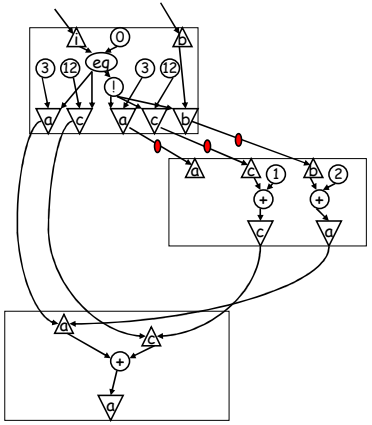
"Implementing" Φ: Example 2



"Implementing" Φ: Example 2

= eta = gate

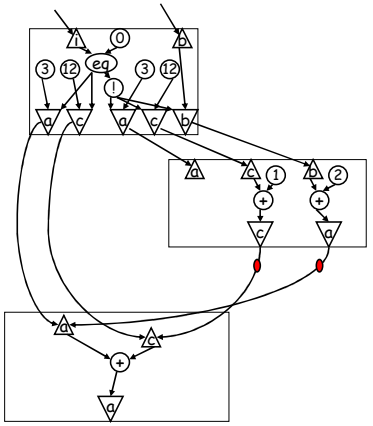
= mu = merge



"Implementing" Φ: Example 2

= eta = gate

= mu = merge

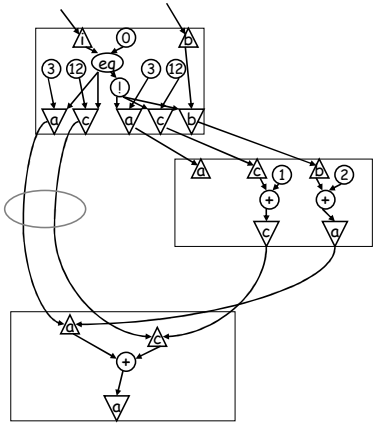


"Implementing" Φ: Example 2

= eta = gate

= mu = merge

Note 1: the set of etas targeting the same successor correspond to one edge in the control flow graph.

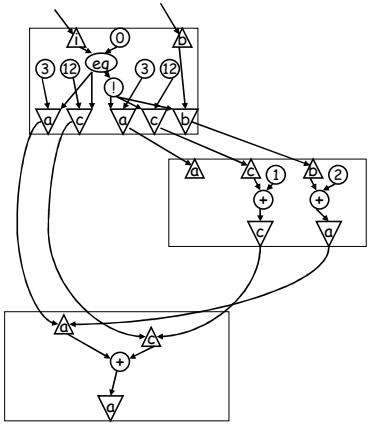


"Implementing" Φ: Example 2

= eta = gate

= mu = merge

Note 2: for a dataflow analysis problem, where do the facts go? how are the facts different?

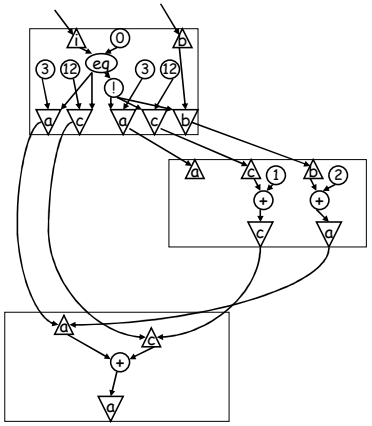


"Implementing" Φ: Example 2

= eta = gate

= mu = merge

Note 3: the etas are like the "implemented Φ" moves on the edges..



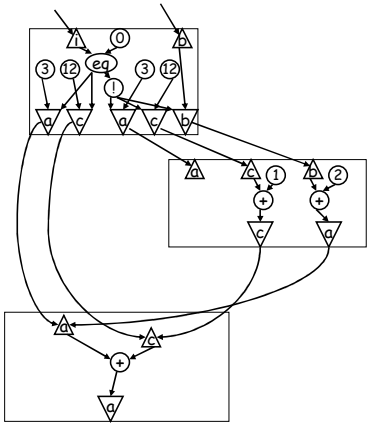
"Implementing" Φ: Example 2

= eta = gate

= mu = merge

Note 4: WHY?

- can be mapped to spatial dataflow hardware directly
- allows detailed, per-data-edge dataflow analysis



Currently

- Pegasus used to convert C into hardware
- Several orders of magnitude improvement in energy-delay!

- 15-745: Use Pegasus to target advanced architecture, e.g., Itanium
  - VLIW (requires compiler to schedule)
  - Predicated (supports predicated ops)
  - Speculation (explicit speculation in ISA)

