# Machine Independent Compiler Optimization -2

- Building and optimizing basic blocks
- Recovering code from expression DAGs
- Control flow graph
  - Reducibility properties
- Worklist iterative algorithm for dataflow analysis
  - Versions of the algorithm
  - Worklist REACH example
- Interval analysis



# Definitions

- *Basic block* code sequence that is entered at the beginning and only exited at the end
- *Control flow graph* G = ( N, E, ρ )

  N is { basic blocks}

  E is {(x,y)} where execution can flow from bblock x to bblock y

  ρ is unique entry node of CFG
- *Loop* strongly connected, single-entry region in CFG

## Machine Independent Optimizations

- Anomaly detection
  - Undefined variables, unreachable code, unused parameters
- Storage sharing
- *Common subexpression elimination*
- Copy propagation
- Dead code elimination
- Code motion
- Reduction in strength
- *Constant propagation*
- *Register allocation*

***optimizationss in red italics can be Performed locally, within a basic block, and globally on CFG***



## Basic Block Optimizations

- Local common subexpression elimination (while building the DAG)
- Dead code elimination
- Copy propagation
- Constant propagation
- Renaming of compiler-generated temporaries to share storage

# Building Expression DAGs

- Leaves are initial values
- Internal nodes labelled by operators, 2nd label is list of identifiers whose value is that node
- Mapping from each identifier to the DAG node containing its value at exit of the basic block
- During building uncover local CSEs and constants, unnecessary temporaries, last defs and first uses of variables

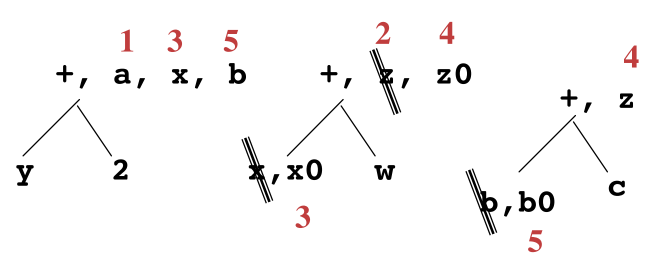


# Local Dead Code Elimination

```
1. a = y + 2
2. z = x + w
3. x = y + 2
4. z = b + c
5. b = y + 2
```

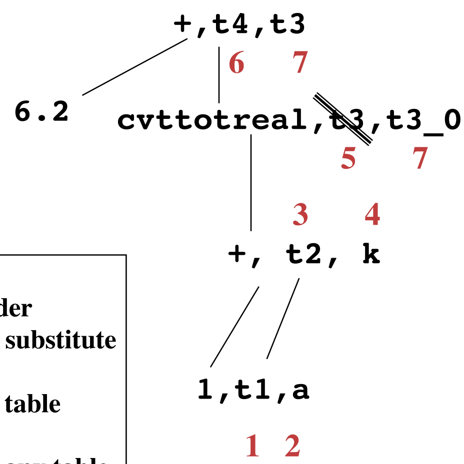
→

```
1'. a = y + 2
2'. x = a
3'. z = b + c
4'. b = a
```



# Local Constant Propagation

```
1. t1 = 1
2.  a = t1
3.  t2 = 1 + a
4.  k = t2
5.  t3 = cvttoreal(k)
6.  t4 = 6.2 + t3
7.  t3 = t4
```




**D. Gries' algorithm:**

- **Process 3 address statements in order**
- **Check if operand is constant; if so, substitute**
- **If all operands are constant,**
  - **Do operation and add value to table associated with L-value**
- **If all operands not constant, delete any table entry for this L-value**

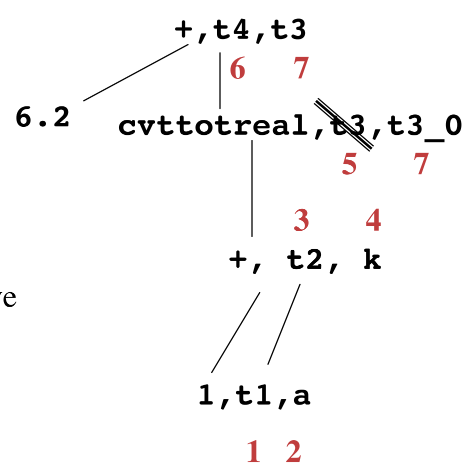


# Local Constant Propagation

Code constructed from final DAG will be

```
1'. a = 1
2'. k = 2
3'. t3 = 8.2
4'. t4 = t3
```

assuming **`a,k,t3,t4`** are all live on basic block exit



## Recovering Code from a DAG

- (Assuming that operands point to operators) topsort traversal of DAG recovers sequence of 3 address statements
  - Generate code for a node when its operands are known
- For nodes with lists of *live* labels y1, y2, …,yk on exit of the basic block, generate

  y1 = a op b, y2 = y1, …, yk = y1
- Nodes without labels correspond to labels that have been reassigned later in the block; create a new compiler temporary to hold result value, if necessary



## Possible Problems

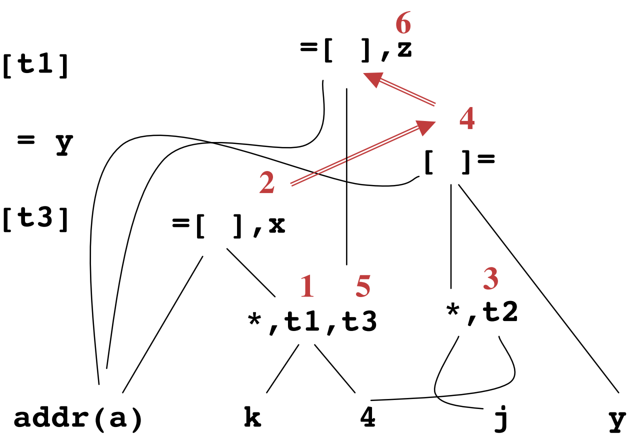
- Troubles with aliasing for accesses into arrays and through pointers

  x = a[k], a[j] = y, z = a[k]

  Does the store into a[j] affect the load from a[k]?
- DAG building algorithm imposes *ordering edges* on array accesses to preserve order of access in generated code
  - Can do similar 'fix' for variable accesses through pointers

# DAGs with Array Elements

```
1. t1 = k * 4
2. x = addr(a)[t1]
3. t2 = j * 4
4. addr(a)[t2] = y
5. t3 = k * 4
6. z = addr(a)[t3]
```




⟹ *preserve original order of array accesses to prevent problems with aliasing*



# DAGs with Array Elements

- **During DAG building, whenever store into an array ([ ] =), kill all existing array store nodes so that they will not be given additional labels (as CSE)**
  - **Can do with one bit (kill/live) and one list of array defn nodes**
- **Insert ordering edges between any array element defn and existing uses of that array, and between any array element use and any existing definitions of the same array**

## DAGs with Pointer Dereferences

- **What rules are sufficient to ensure safety of generated code?**
- **As we build the DAG, (in the absence of points-to information)**
  - **Any indirect store through a pointer must follow any previous stores or fetches through a pointer**
  - **Any indirect fetch through a pointer must follow any previous store through pointer**
- **Q: are these rules sufficient? Why or why not?**
- **Q: what about reference variables/fields in OOPLs?**



## Natural Loops

- **Single entry node, *header,* that *dominates* all other nodes in the loop**
- **<u>Invariant:</u> from every node there is one path back to the header.**
- **Loop construction:**
  - **Find back edge. Traverse edges in reverse execution direction until back edge target is reached. All nodes encountered. all nodes encountered in traversal are in corresponding natural loop (ASU algm 10.1, p 606)**

## Natural Loops

- **Loops with different headers are either**
  - – ***nested,*** **one contained entirely within the other - an *inner loop)* , or**
    - **If each pair of nodes, one in loop (n) and one in loop(k), are reachable from each other, and header(n) dominates header(k), then loop(k) is nested within loop(n)**
  - – ***disjoint***
- **Loops with the same header are assumed to be part of the same natural loop**



## Flowgraphs

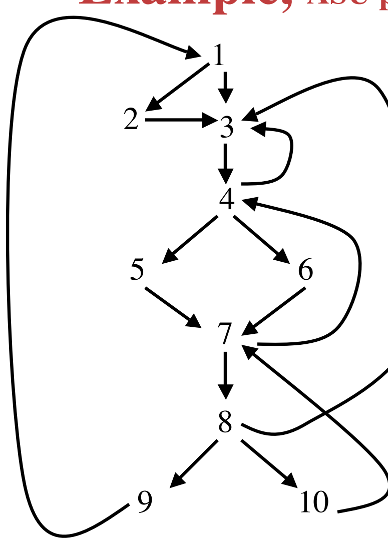
- **Spanning tree of the control flowgraph**
  - – **Built using depth-first search**
  - – **Nodes numbered in preorder**
  - – ***Back edge*** **goes from a node to one of its ancestors in the tree rooted at cfg entry node**
  - – **Divides edges into four groups: *tree, forward, cross* and *back* edges**
  - – **Note: different spanning trees on an arbitrary digraph may result in different edges as *back edges***

# Properties of Reducible Flowgraphs

- Can partition their edges into 2 disjoint sets
  - Forward edges form an acyclic graph in which each node is reachable from flowgraph entry, ρ
  - Back edges consist only of edges whose targets *dominate* their sources.
  - *The set of back edges of reducible flowgraph is unique.*
    - Means set of spanning-tree-induced *back edges* and back edges are same.
- All loops are single-entry which facilitates code motion to a preheader node of the loop
- Allows dataflow analysis methods based on graph decomposition (*elimination* methods)

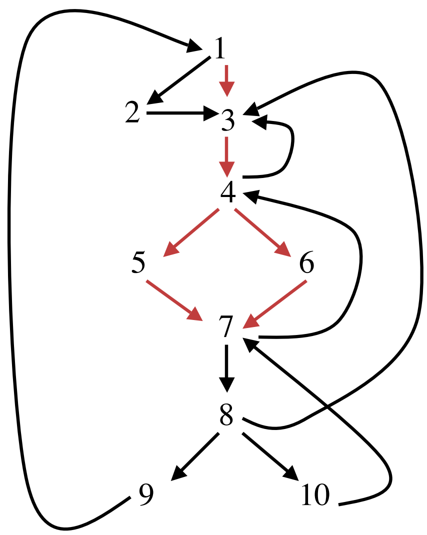


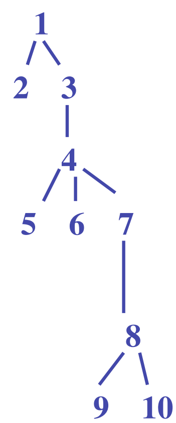
# Example, ASU p 603



node 1 dominates node 7

# Example

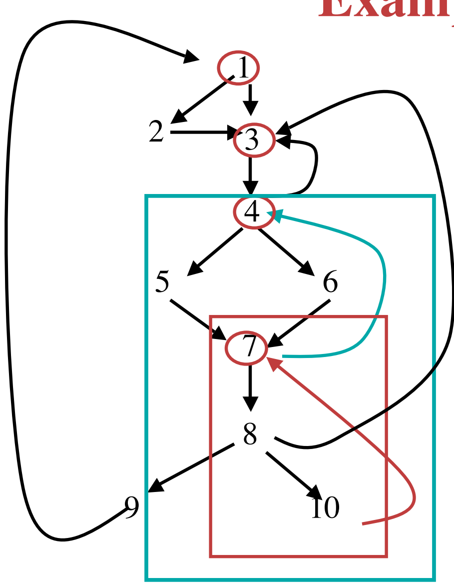


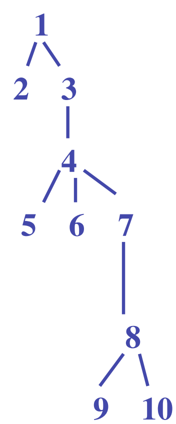




**Dominator Tree**



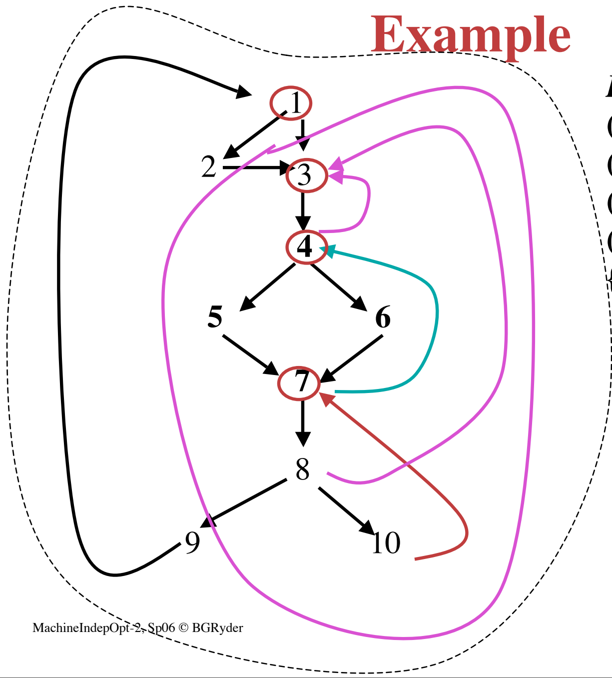
# Example




**Dominator Tree**

# Example




*Loops*
(10,7): {7,8,10}
(7,4): {4,5,6,7,8,10}
(4,3)(8,3): {3,4,5,6,7,8,10}
(9,1):
{1,2,3,4,5,6,7,8,9,10}



# Dominators, ASU p 671

- **How to find dominators of flowgraph,**

**G=(N, E, ρ)? use fixed point iteration (justification later)**

$D(\rho) = \{\rho\}$
for $n \in N - \{\rho\}$ do
{ $D(n) = N$;}
while changes to any D(n) occur do
{ for $n \in N - \{\rho\}$ do

$$D(n) = \{n\} \cup \bigcap_{p \in pred(n)} D(p)$$

}

## Dominators

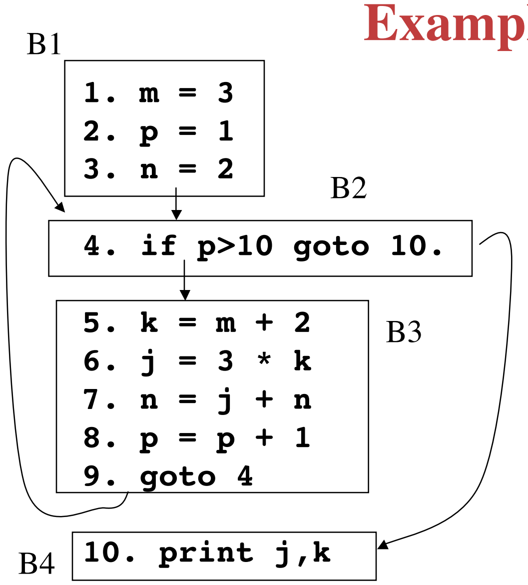
- **Algorithm terminates since at every step some set D(k) becomes smaller; this cannot occur indefinitely, so loop terminates**
- **Can code iteration using bitvector representation for node set and logical and and or**
- **<u>Invariant:</u> Node *k* is parent of node *n* in the dominator tree, if node *k* is the *immediate dominator* of *n***



## Invariant Code Motion

- **Computation is *loop invariant* if its value does not change while control stays within the loop**
- **How find loop invariant computations?**
  - **Mark invariant all 3 address statements whose operands are constant or have all reaching definitions from outside the loop**
  - **Mark invariant all 3 address statements not previously marked such that all operands are constant or all operand reaching definitions are outside the loop or 1 reaching definition in loop is marked invariant already. Repeat until no new statements are marked invariant.**
- **Create loop pre-header node (immediate predecessor of header) as destination for moved code**

# Example

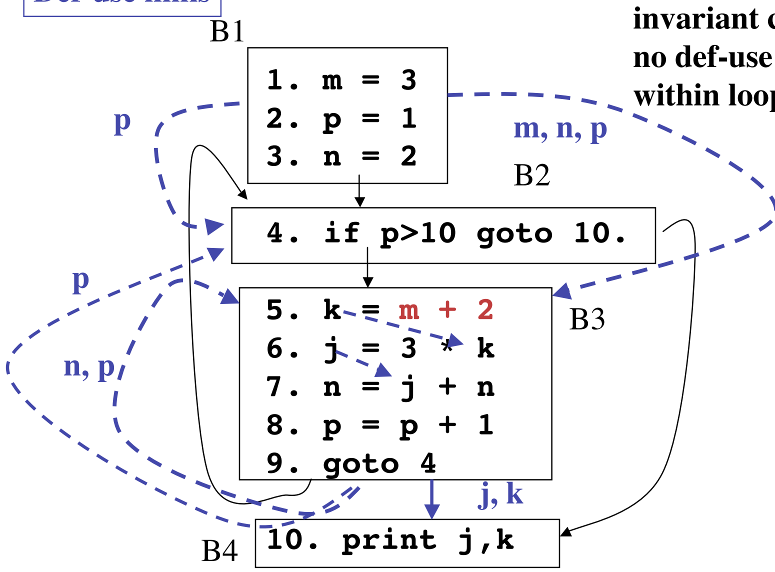



**First, need to find def-use links.**
**all-defs= {<m,B1> <p,B1> <n,B1> <k,B3> <j,B3> <n,B3> <p,B3>}**

**Reaching defs solution:**
**Reach(B1)=ϕ**
**Reach(B2)=Reach(B3)= Reach(B4)= all-defs**

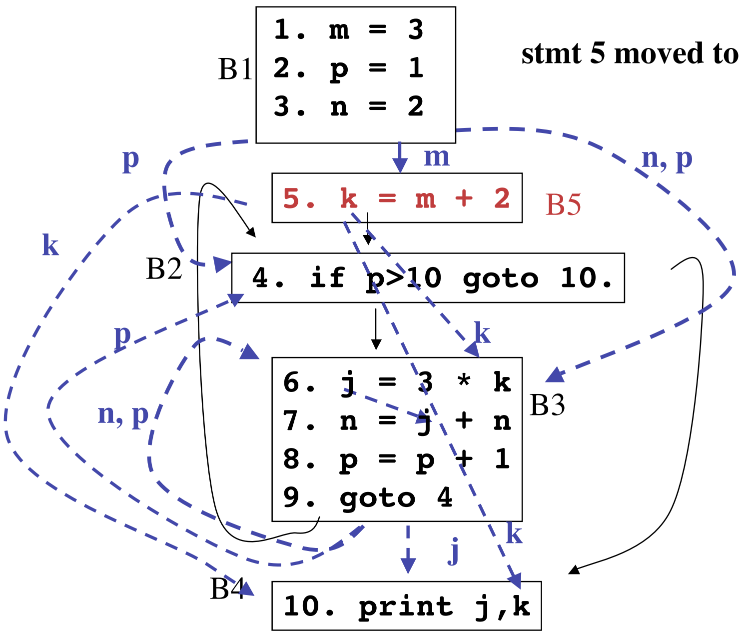


## Def-use links



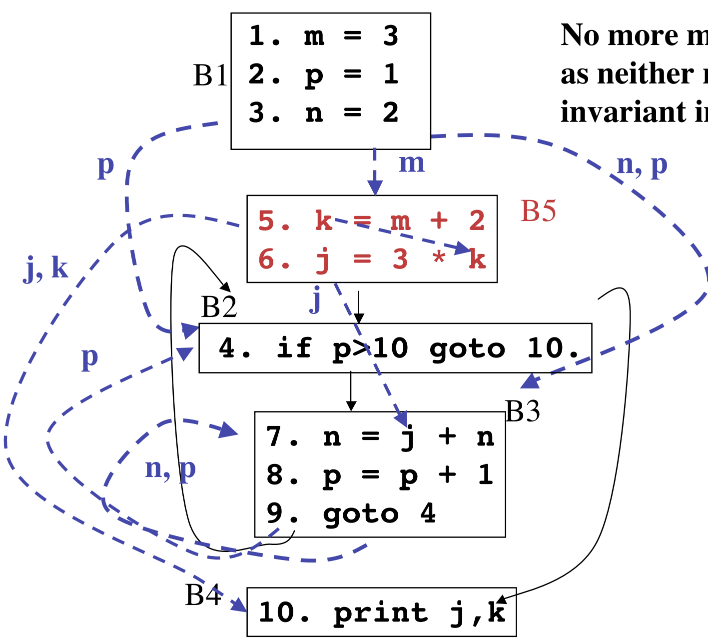

**Second, look for invariant code with no def-use links from within loop (B2,B3)**

stmt 5 moved to preheader B5






No more movt possible as neither n nor p are invariant in loop.



## Maintaining Dataflow Information

- **If we perform code motion, what dataflow information is invalidated?**
  - **Dominators: can easily insert preheader into this relation**
  - **Def-use, use-def links: need to update for further use**
- **Idea- can use indirect notation for chain**
  - **To update after a move, look at def-use chain of moved stmt to find which use-def chains must be updated**



## Updating use-def chains

- **Store def-use chain as a linked list of pointers to defs (e.g., <m,B1>)**
- **Follow use-def links of moved defn.**
  - **When move x=a*b, lookup def-use(x) and for each use identified, change old defn block number to new defn block number**

## Worklist Algm for Bitvector DFA

change = true;
initialize Reach(m) = $\emptyset$;
*while* (change) *do*
{ change = false;
*while* ( $\exists$ j $\ni$ Reach(j) $\neq$ $\bigcup_{m \in pred(j)}$ (Reach(m) $\cap$ pres(m) $\cup$ dgen(m)) *do*
{ Reach(j) = $\bigcup_{m \in pred(j)}$ (Reach(m) $\cap$ pres(m) $\cup$ dgen(m))
change = true; }
}

Will justify later why fixed point iteration is appropriate for this problem; Algorithm needs to be optimized and should be deterministic.



## Implementation

Use bitstring representation for sets: 1 bit position per variable definition

For each control flowgraph node j,

pres(j)

has 0 in bit positions corresponding to definitions of variables which are *defined* at node j

otherwise 1's.

dgen(j)

has a 1 in each bit position corresponding to a definition of a variable at node j, *downwards-exposed defns*

otherwise 0's.

# Algorithm

```
/* initially all reaching sets are empty */
for m := 1 to n do          Reach(m) := 0B;
W := {1,2,...,n} /*put every cfg node on worklist*/
while W ≠ ∅ do
{   Remove k from W;
    new = ∪ { Reach(m) ∩  pres(m) ∪ dgen(m) } ;
       m ∈ pred(k)
    if new ≠ Reach(k) then
           { Reach(k) := new;
            for j ∈  succ(k) do
               add j  to W, if is not already there;
           }
}
```



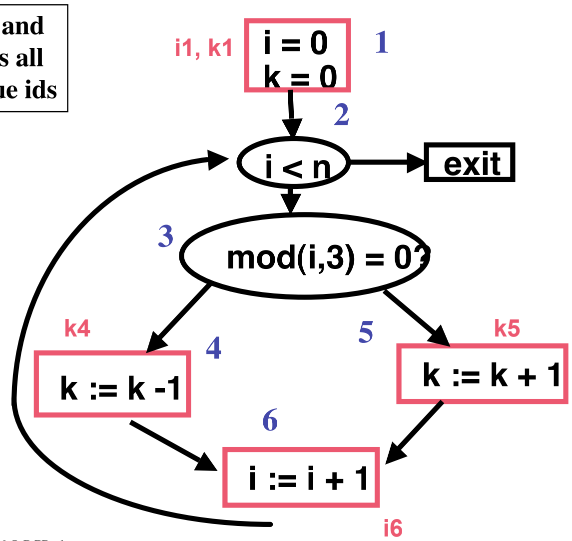
# Detailed Algorithm

```
W = empty /* initialize worklist */
for (i  = 1; i < n+1; i++) /* i varies over nodes */
  for (j = 1; j <m+1; j++) /* j varies over defs */
{    if (k ∈ pred(i) with j ∈ dgen(k))
             then   {set j bit to 1 in REACH(i);
                     add (j, i) to W}
             else   {set j bit to 0 in REACH(i)}
     endif;
}
while (W not empty) do
{    Remove (j,i) from W
     if j  ∈ pres(i) then
             {for (k  ∈ succ(i))
                       if (j bit in REACH(k) == 0)
                                then {set j bit to 1 in REACH(k);
                                       add (j,k) to W}
                       endif;
             }
     endif;
}
```

This version of algorithm accomplishes 1 iteration in the initialization loop, by passing dgen sets to successors. Second while loop performs worklist propagation. Can order entries on worklist for better performance.

# Example, Bitvector Calculation

Definitions and basic blocks all given unique ids

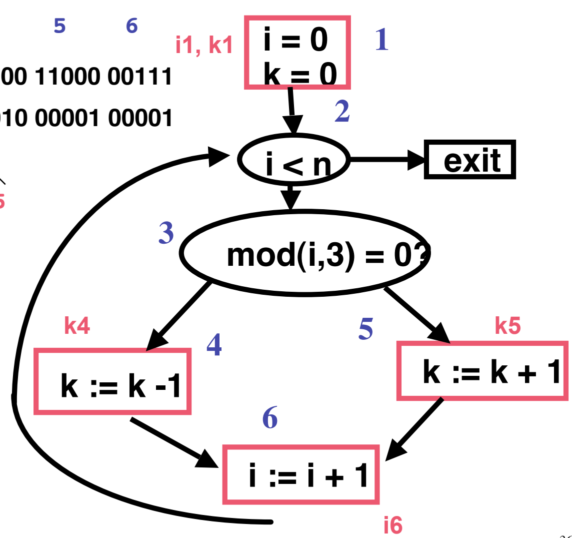




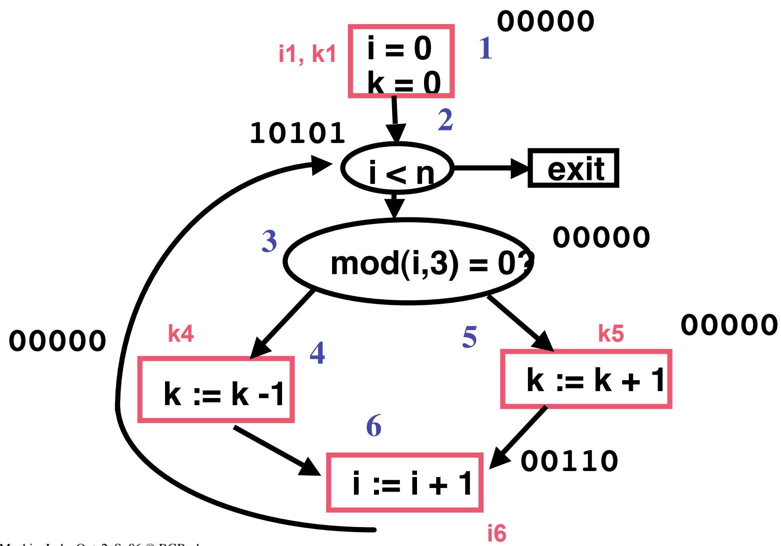

# Initialization

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| pres | 00000 | 11111 | 11111 | 11000 | 11000 | 00111 |
| dgen | 10100 | 00000 | 00000 | 00010 | 00001 | 00001 |

Bits: i1 i6 k1 k4 k5



# After initialization loop






# Trace of propagation loop

Worklist W = *{(i1,2),(k1,2), (i6,2), (k4,6), (k5,6)}*

Choose *(i1,2)* from W; pres(2)==11111, so REACH(3) = 10000 and we add *(i1,3)* to W.

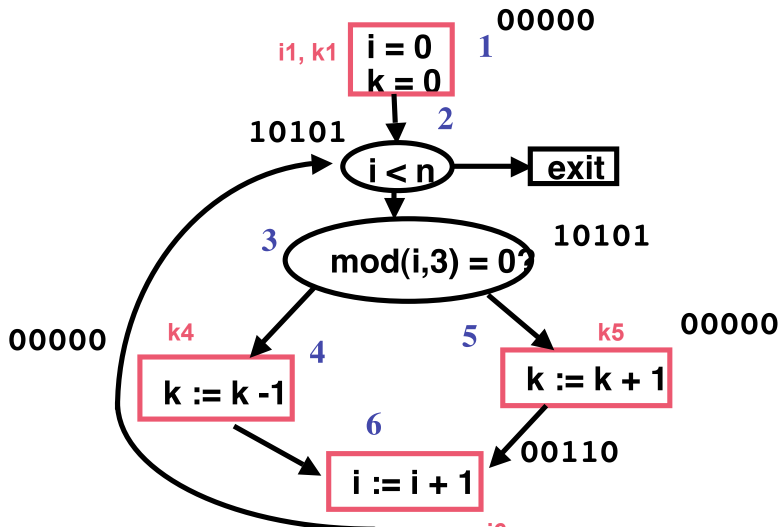
Then choose *(k1,2)* off W and set REACH(3) = 10100 and we add *(k1,3)* to W.

Then choose *(i6,2)* off W and set REACH(3) = 10101 and add *(i6,3)* to W. Now

W= *{(k4,6), (k5,6), (i1,3),(k1,3), (i6,3)}*

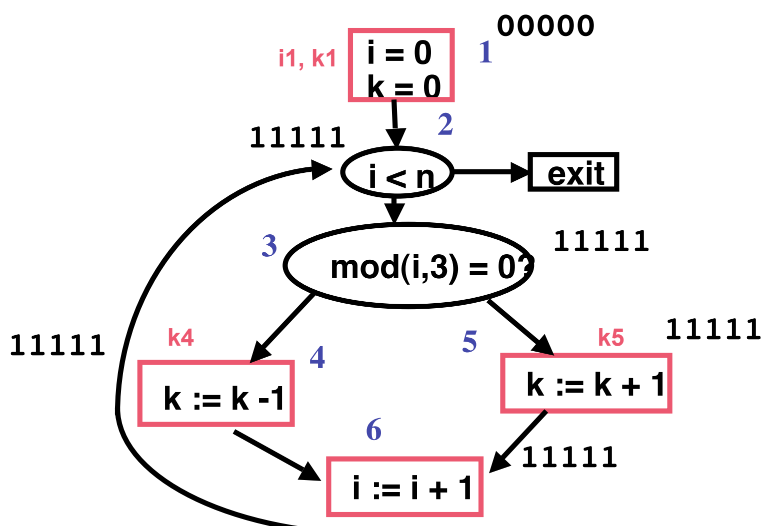
Iteration continues until worklist is empty.

# After steps in previous slide

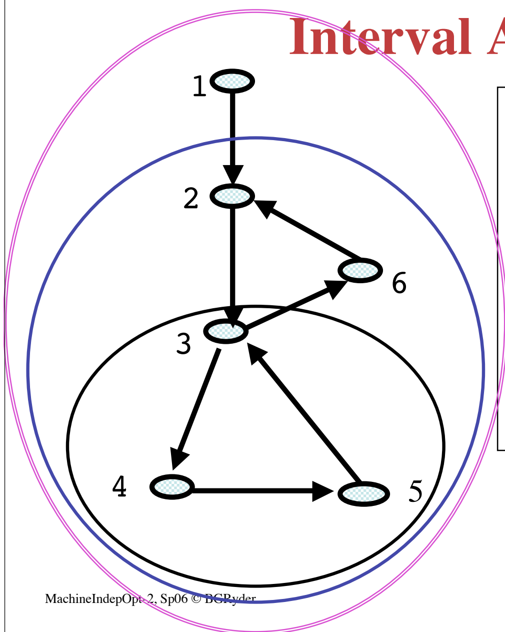





# Solution



# Elimination Methods - Interval Analysis




**Gaussian-elimination-like solution procedure for classical df problems.**
**1. Solve dataflow problems within regions of cfg, obtaining df eqns for each node in terms of the loop entry node. Effectively collapse loops into their header node as solve.**
**2. Back substitute value for loop entry node into all equations to obtain solution at node**
**Effectively, expand loops as substitute.**