# Introduction to Machine-Independent Optimizations - 7

## Program Optimizations and the SSA Form

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

# Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis (in parts 4 and 5)
- Algorithms for machine-independent optimizations (in part 6)
- SSA form and optimizations

# SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, $\phi$, is used for selection of values in join nodes
- Conditional constant propagation is faster and more effective on SSA forms

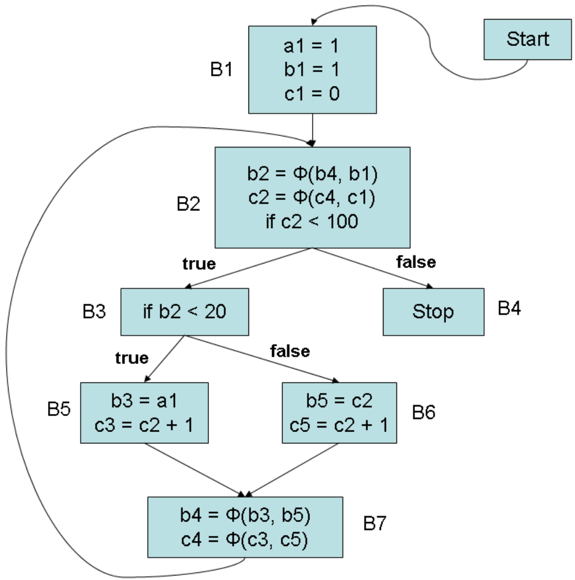
# Conditional Constant Propagation - 1

- 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

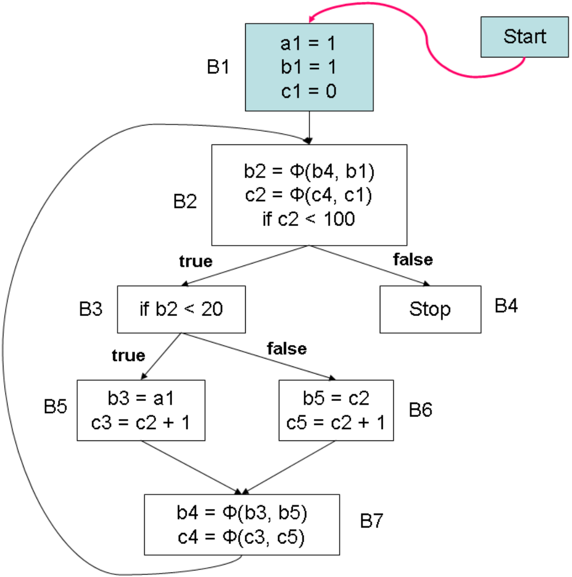
# Conditional Constant Propagation - 2

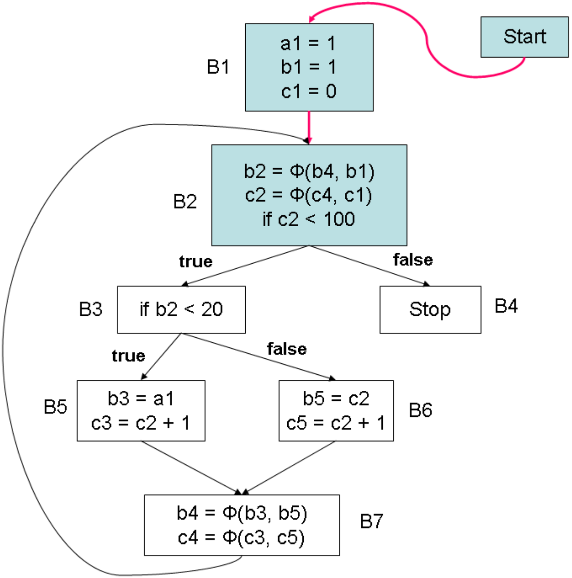
- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs much lesser storage compared to its non-SSA counterpart
- 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 2

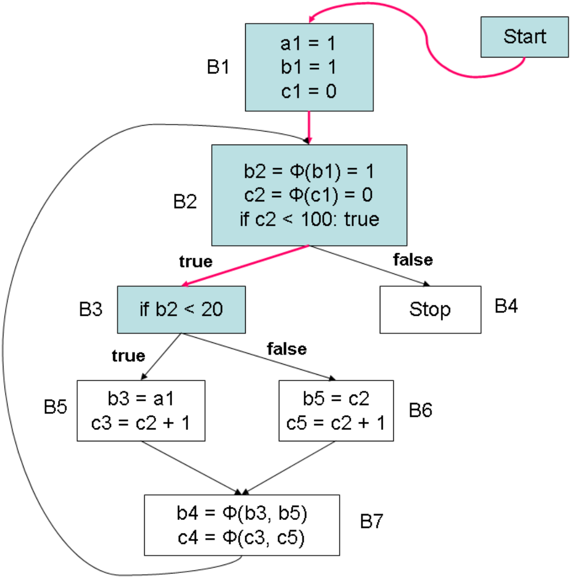


# CCP Algorithm - Example 2 - Trace 1

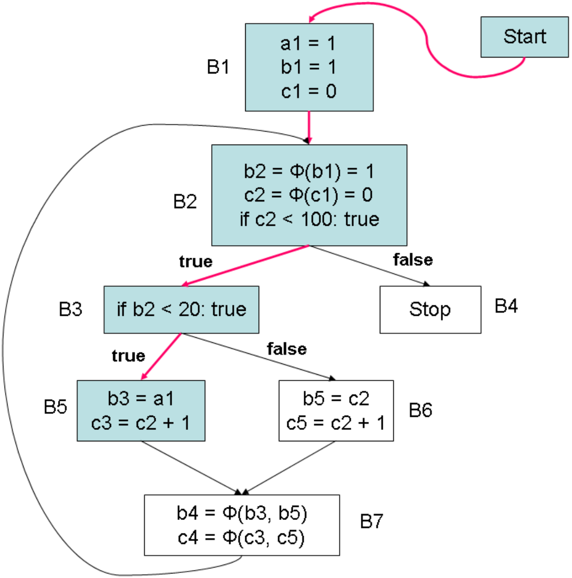



Start
B1
a1 = 1
b1 = 1
c1 = 0
B2
b2 = Φ(b4, b1)
c2 = Φ(c4, c1)
if c2 < 100
true
false
B3
if b2 < 20
Stop
B4
true
false
B5
b3 = a1
c3 = c2 + 1
b5 = c2
c5 = c2 + 1
B6
b4 = Φ(b3, b5)
c4 = Φ(c3, c5)
B7

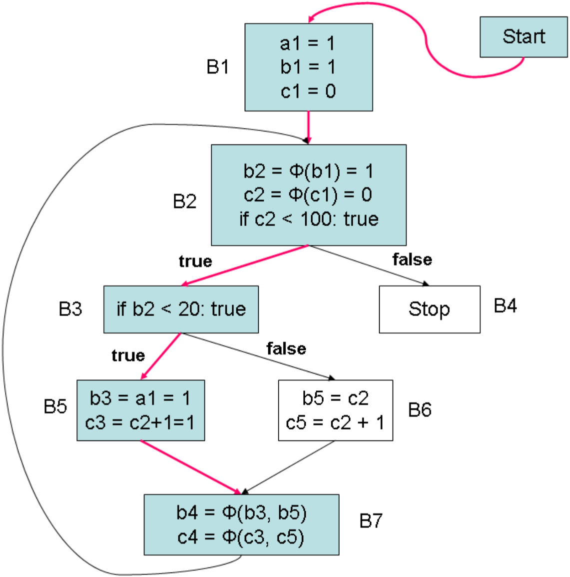
# CCP Algorithm - Example 2 - Trace 3



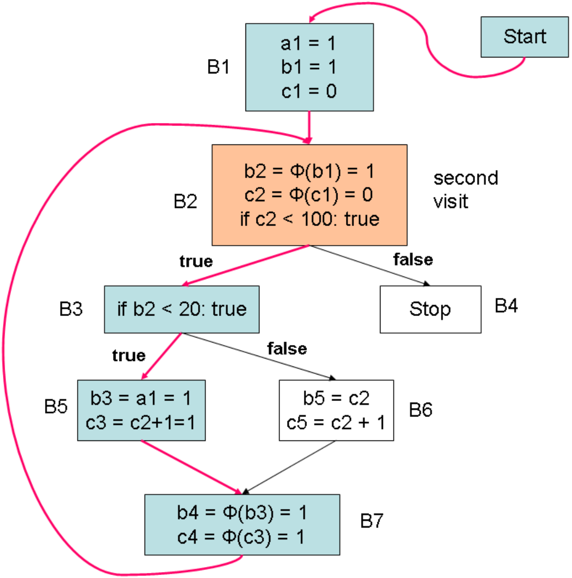
# CCP Algorithm - Example 2 - Trace 4



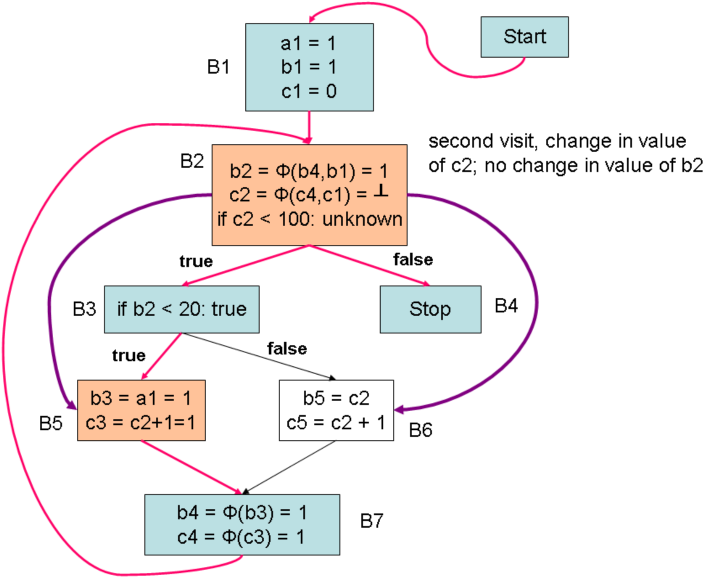
# CCP Algorithm - Example 2 - Trace 5



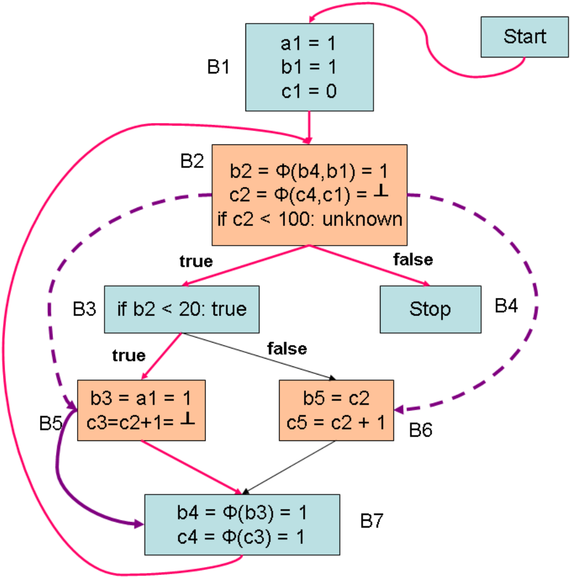
# CCP Algorithm - Example 2 - Trace 6

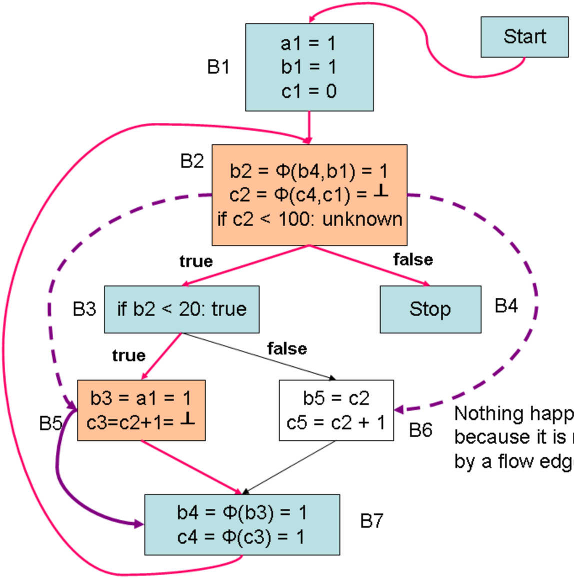


# CCP Algorithm - Example 2 - Trace 7



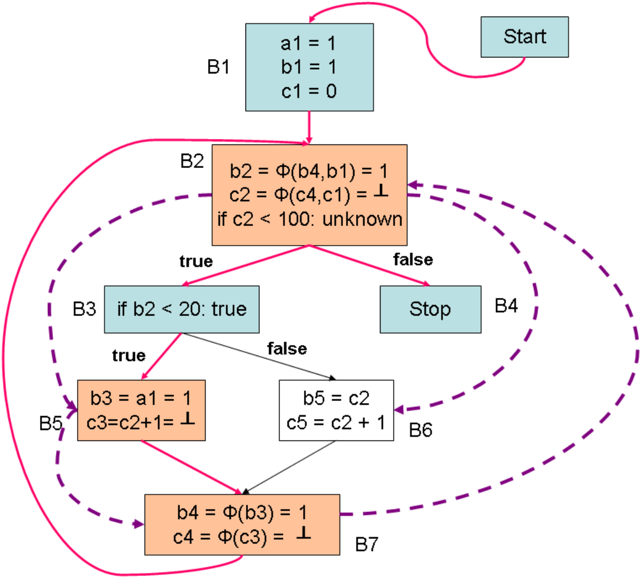
# CCP Algorithm - Example 2 - Trace 8



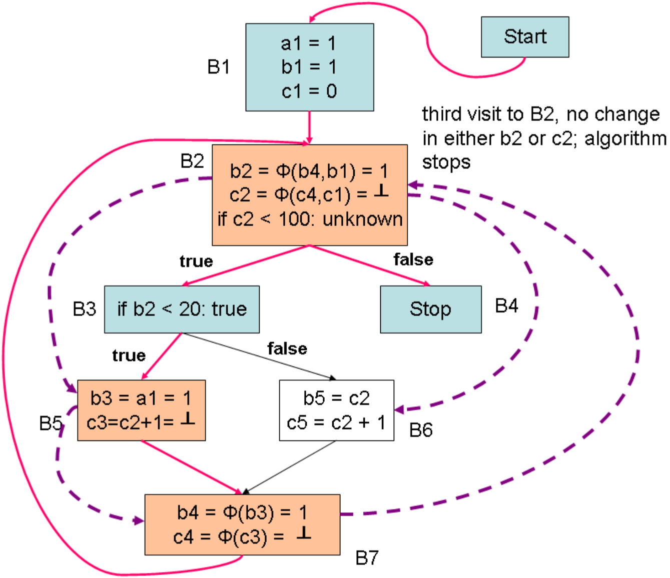



Nothing happens in B6 because it is not reachable by a flow edge

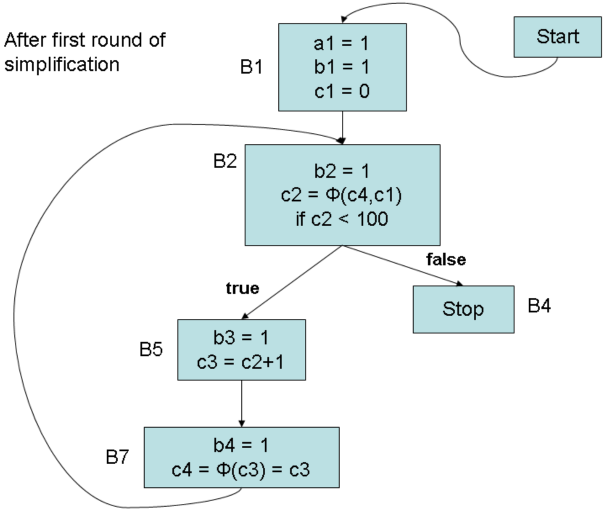
# CCP Algorithm - Example 2 - Trace 10



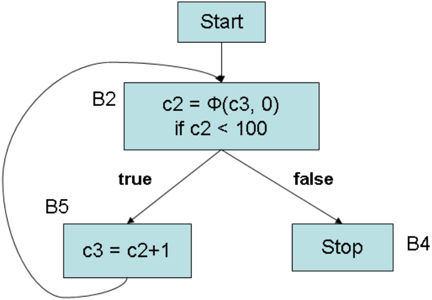
# CCP Algorithm - Example 2 - Trace 11



# 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.

# Instruction Scheduling and Software Pipelining - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

# Outline

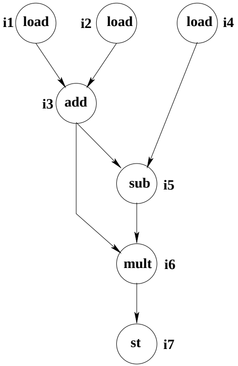
- Instruction Scheduling
  - Simple Basic Block Scheduling
  - Trace, Superblock and Hyperblock scheduling
- Software pipelining

- Reordering of instructions so as to keep the pipelines of functional units full with no stalls
- NP-Complete and needs heuristcs
- Applied on basic blocks (local)
- Global scheduling requires elongation of basic blocks (super-blocks)

- time: load - 2 cycles, op - 1 cycle
- This code has 2 stalls, at i3 and at i5, due to the loads

| | | | |
|---|---|---|---|
| i1: | r1 | ← | load a |
| i2: | r2 | ← | load b |
| i3: | r3 | ← | r1 + r2 |
| i4: | r4 | ← | load c |
| i5: | r5 | ← | r3 - r4 |
| i6: | r6 | ← | r3 * r5 |
| i7: | d | ← | st r6 |

(a) Sample Code Sequence




(b) DAG

# Scheduled Code - no stalls

- There are no stalls, but dependences are indeed satisfied

| | | | |
|---|---|---|---|
| i1: | r1 | $\leftarrow$ | load a |
| i2: | r2 | $\leftarrow$ | load b |
| i4: | r4 | $\leftarrow$ | load c |
| i3: | r3 | $\leftarrow$ | r1 + r2 |
| i5: | r5 | $\leftarrow$ | r3 - r4 |
| i6: | r6 | $\leftarrow$ | r3 * r5 |
| i7: | d | $\leftarrow$ | st r6 |

# Definitions - Dependences

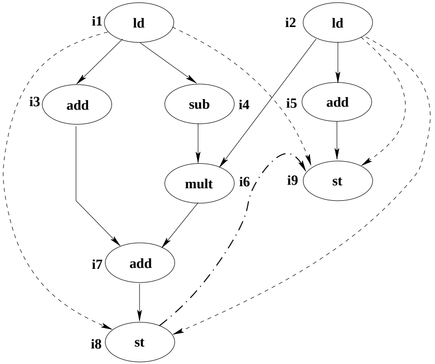
- Consider the following code:
  $i_1: \; r1 \leftarrow load(r2)$
  $i_2: \; r3 \leftarrow r1 + 4$
  $i_3: \; r1 \leftarrow r4 + r5$
- The dependences are
  $i_1 \; \delta \; i_2$ (flow dependence) $i_2 \; \overline{\delta} \; i_3$ (anti-dependence)
  $i_1 \; \delta^o \; i_3$ (output dependence)
- anti- and ouput dependences can be eliminated by register renaming

# Dependence DAG

- full line: *flow* dependence, dash line: *anti*-dependence
  dash-dot line: *output* dependence
- some anti- and output dependences are because memory disambiguation could not be done

| | |
|---|---|
| i1: | t1 ← load a |
| i2: | t2 ← load b |
| i3: | t3 ← t1 + 4 |
| i4: | t4 ← t1 - 2 |
| i5: | t5 ← t2 + 3 |
| i6: | t6 ← t4 * t2 |
| i7: | t7 ← t3 + t6 |
| i8: | c ← st t7 |
| i9: | b ← st t5 |

(a) Instruction Sequence




(b) DAG

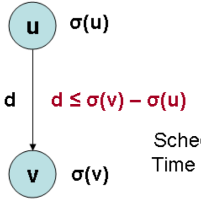
# Basic Block Scheduling

- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
  - PC's relate to data dependences and execution delays
  - RC's relate to limited availability of shared resources

# The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, $G = (V, E)$
  - $R$: number of resources
  - Nodes ($V$): MOS; Edges ($E$): Precedence
  - Label on node $v$
    - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with $v$
    - length $l(v)$ of node $v$
  - Label on edge $e$: Execution delay of the MOS, $d(e)$
- Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that $\forall e = (u, v) \in E,\ \sigma(v) - \sigma(u) \geq d(e)$ and $\forall i, \sum^{v \in V} \rho_v(i - \sigma(v)) \leq R$, where length of the schedule is $\max_{v \in V}\{\sigma(v) + l(v)\}$

# Instruction Scheduling - Precedence and Resource Constraints




Precedence constraint

MOS substeps (time)

Schedule Time $\sigma(u)$ ↓

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $\sigma(v_1)=0$ | 1 | 1 | 2 | 2 | |
| $\sigma(v_2)=1$ | 2 | 3 | 1 | 1 | 2 |
| $\sigma(v_3)=2$ | 3 | 1 | 2 | | |
| $\sigma(v_4)=3$ | 1 | 2 | 3 | 2 | |

Resource constraint

Consider R = 5. Each MOS substep takes 1 time unit.

At i=4, $\varsigma_{v4}(1)+\varsigma_{v3}(2)+\varsigma_{v2}(3)+\varsigma_{v1}(4) = 2+2+1+0 = 5 \leq R$, satisfied

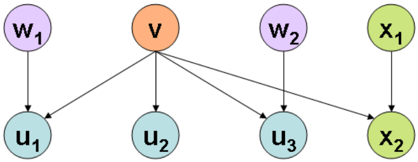
At i=2, $\varsigma_{v3}(0)+\varsigma_{v2}(1)+\varsigma_{v1}(2) = 3+3+2 = 8 > R$, NOT satisfied

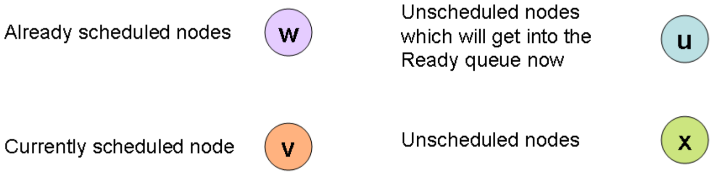
# A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

FUNCTION ListSchedule (V,E)
BEGIN
  *Ready* = root nodes of V; *Schedule* = $\phi$;
  WHILE *Ready* $\neq \phi$ DO
  BEGIN
    v = highest priority node in *Ready*;
    *Lb* = SatisfyPrecedenceConstraints $(v,\ Schedule,\ \sigma)$;
    $\sigma$(v) = SatisfyResourceConstraints $(v,\ Schedule,\ \sigma,\ Lb)$;
    $Schedule\ =\ Schedule\ +\ \{v\}$;
    $Ready = Ready\ -\ \{v\}\ +\ \{u \mid NOT\ (u\ \in Schedule)$
          $AND\ \forall\ (w, u)\ \in\ E,\ w\ \in\ Schedule\}$;
  END
  RETURN $\sigma$;
END

# List Scheduling - Ready Queue Update



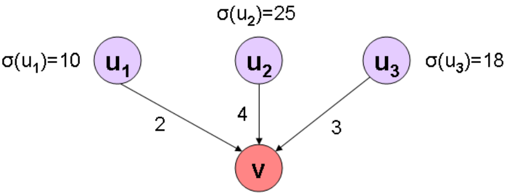



# Constraint Satisfaction Functions

FUNCTION SatisfyPrecedenceConstraint(v, Sched, $\sigma$)
BEGIN
RETURN ( $\max\limits_{u \in Sched} \sigma(u) + d(u, v)$)
END

FUNCTION SatisfyResourceConstraint(v, Sched, $\sigma$, Lb)
BEGIN
FOR i := Lb TO $\infty$ DO
IF $\forall 0 \leq j < l(v),\ \rho_v(j) + \sum^{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R$ THEN
RETURN (i);
END

# Precedence Constraint Satisfaction




Lower bound for $\sigma(v)$ = 29

Already scheduled nodes

Node to be scheduled


Precedence constraint satisfaction:

v can be scheduled only after all of $u_1$, $u_2$, and, $u_3$, finish

Lower bound for $\sigma(v)$
$= \max(10+2, 25+4, 18+3)$
$= \max(12, 29, 21) = 29$

# Resource Constraint Satisfaction

Resource constraint satisfaction
Consider R = 5. Each MOS substep takes 1 time unit.

MOS substeps (time)

Schedule Time $\sigma(u)$

| Schedule Time $\sigma(u)$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $\sigma(v_1)=0$ | 1 | 1 | 2 | 2 | |
| $\sigma(v_2)=1$ | 2 | 3 | 1 | 1 | 2 |
| 2 | | | | | |
| 3 | | | | | |
| $\sigma(v_3)=4$ | 3 | 1 | 2 | | |
| $\sigma(v_4)=5$ | 1 | 2 | 3 | 2 | |

Time slots 2 and 3 are vacant because scheduling node $v_3$ in either of them violates resource constraints