CS 4240: Compilers

Lecture 14: Instruction Scheduling (contd)

Instructor: Vivek Sarkar
(vsarkar@gatech.edu)

February 27, 2019

ANNOUNCEMENTS & REMINDERS

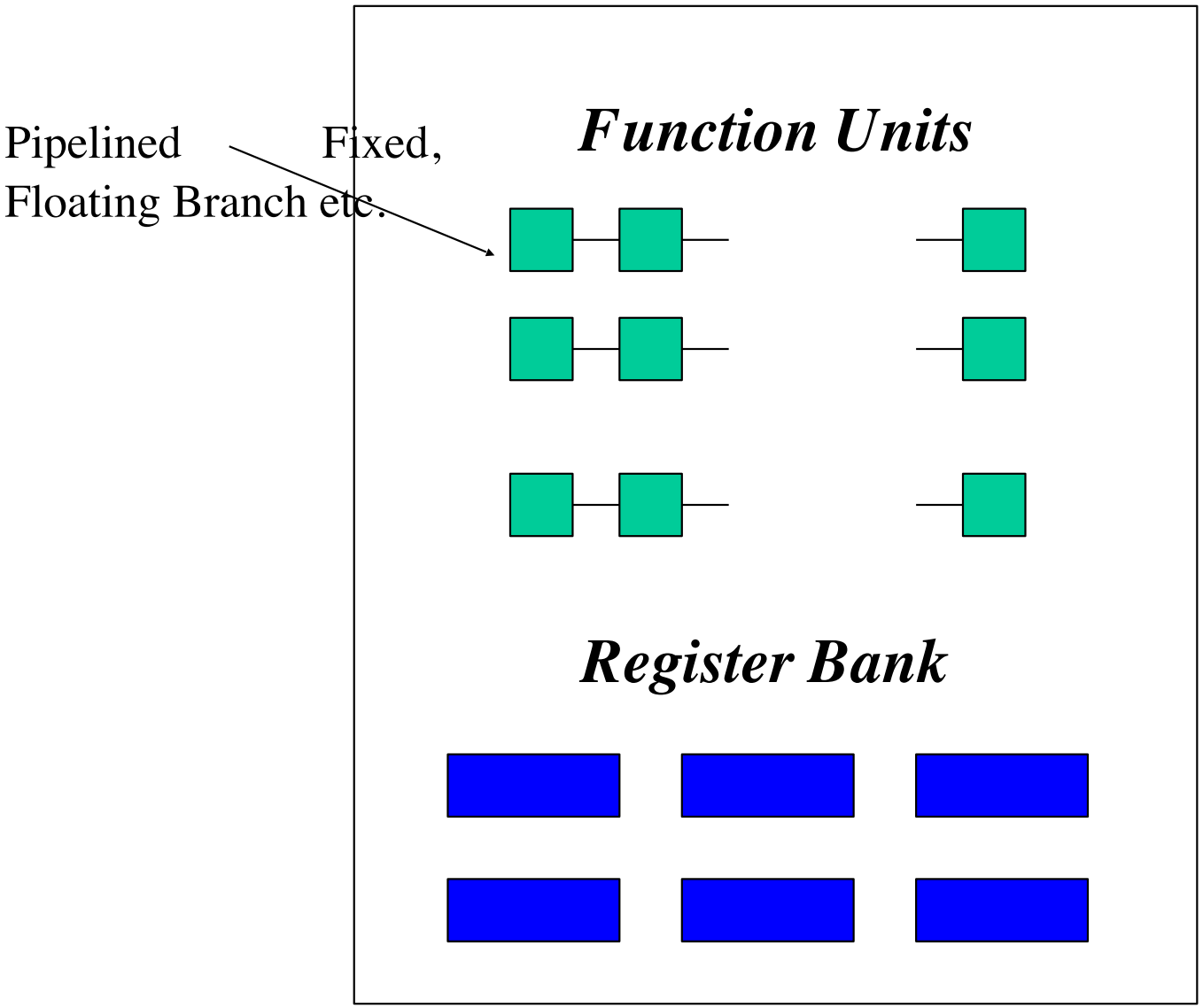
- Homework 2 to be released today
  - Due by 11:59pm on Monday, March 4th
  - 5% of course grade

- Project 2 released today
  - Due by 11:59pm on Wednesday, April 3rd
  - 15% of course grade

- MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm
  - 20% of course grade

- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm
  - 30% of course grade

Superscalar (RISC) Processors



What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is order-dependent (and has been since the 60's)

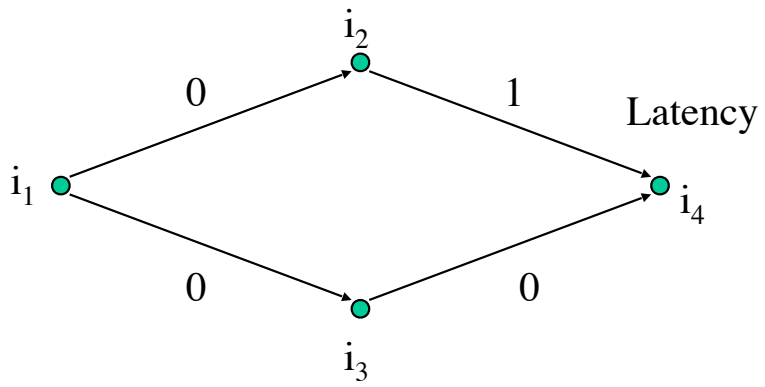
Assumed latencies (conservative)

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  - > Non-blocking ⇒ fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  - > Fill slots with unrelated operations
  - > Percolates branch upward
- Scheduler should hide the latencies

Example: Instruction Scheduling

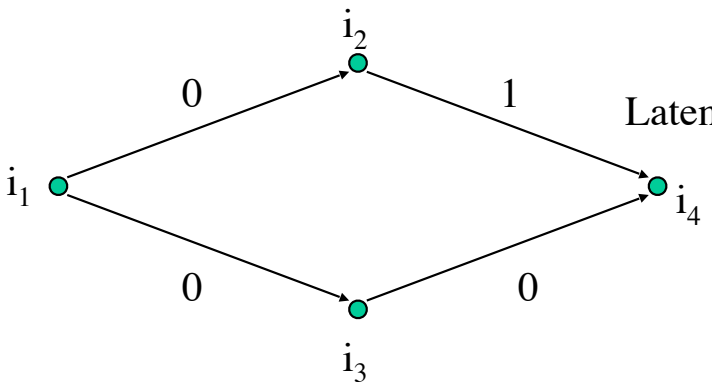
Input: *A basic block represented as a Directed Acyclic Graph (DAG)*

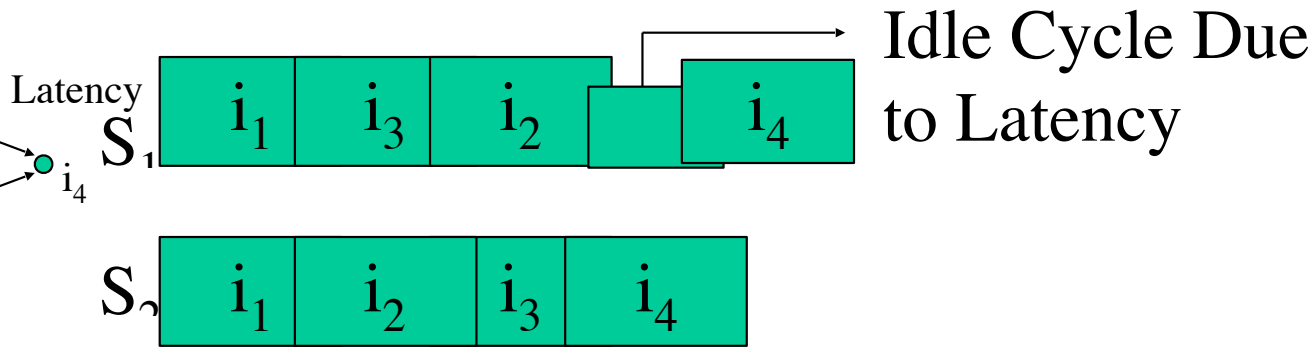


- $I_{\#}$ are instructions in the *basic block*; edges (*i, j*) represent dependence constraints
- i_2 is a load instruction.
- Latency of 1 on (i_2,i_4) means that i_4 cannot start for one cycle after i_2 completes.
- Assume 1 FU
- What are the possible schedules?

Example(cont): Possible Schedules

- Two possible schedules for the DAG
- The length of the schedule is the number of cycles required to execute the operations
  - Length(S_1) > Length(S_2)
- Which schedule is optimal?





Formalizing the Instruction Scheduling Problem (contd)

Feasible Schedule: A specification of a *start time* for each instruction such that the following constraints are obeyed:

1. Resource: Number of instructions of a given type of any time < corresponding number of FUs.
2. Dependence and Latency: For each predecessor *j* of an instruction *i* in the DAG, *i* is the started only δ cycles after *j* finishes where δ is the latency labeling the edge (*j*,*i*),

Output: A schedule with the minimum *overall completion time (makespan).*

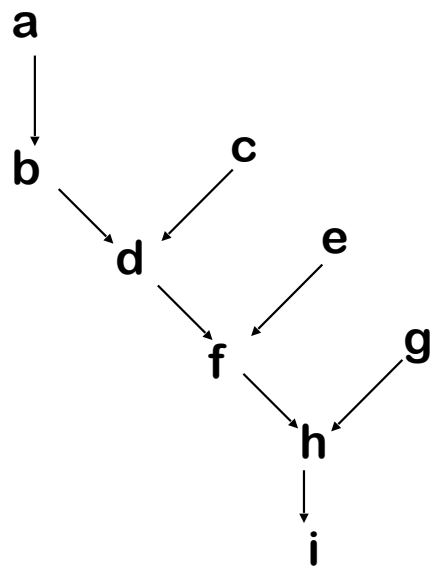
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a dependence graph G

- Nodes $n \in G$ are operations with type(n) and delay(n)
- An edge $e = (n_1,n_2) \in G$ if & only if n_2 uses the result of n_1

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r2
f:  mult     r1,r2  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

Example

w ← w * 2 * x * y * z

Simple schedule

```
 1  loadAI   r0,@w  ⇒ r1
 4  add      r1,r1  ⇒ r1
 5  loadAI   r0,@x  ⇒ r2
 8  mult     r1,r2  ⇒ r1
 9  loadAI   r0,@y  ⇒ r2
12  mult     r1,r2  ⇒ r1
13  loadAI   r0,@z  ⇒ r2
16  mult     r1,r2  ⇒ r1
18  storeAI  r1     ⇒ r0,@w
21  r1 is free
```

2 registers, 20 cycles

Schedule loads early

```
 1  loadAI   r0,@w  ⇒ r1
 2  loadAI   r0,@x  ⇒ r2
 3  loadAI   r0,@y  ⇒ r3
 4  add      r1,r1  ⇒ r1
 5  mult     r1,r2  ⇒ r1
 6  loadAI   r0,@z  ⇒ r2
 7  mult     r1,r3  ⇒ r1
 9  mult     r1,r2  ⇒ r1
11  storeAI  r1     ⇒ r0,@w
14  r1 is free
```

3 registers, 13 cycles

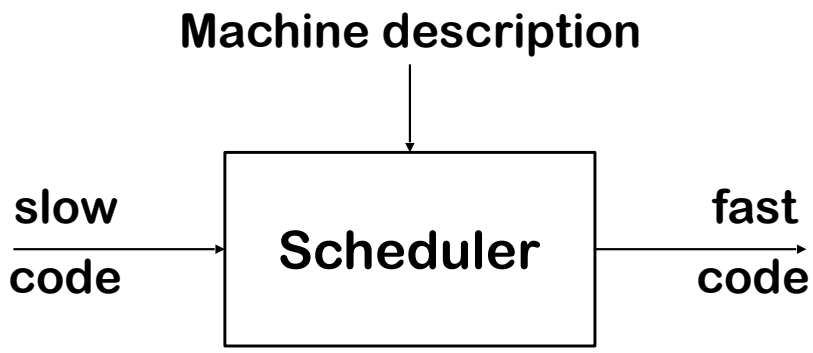
Reordering operations for speed is called instruction scheduling

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The Task

- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

Instruction Scheduling (Definitions)

A correct schedule S maps each $n \in N$ into a non-negative integer representing its cycle number, and

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1,n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
3. For each type t, there are no more operations of type t in any cycle than the target machine can issue

The length of a schedule S, denoted L(S), is

$L(S) = max_{n \in N} (S(n) + delay(n))$

The goal is to find the shortest possible correct schedule.

S is time-optimal if $L(S) \leq L(S_1)$, for all other schedules S_1

A schedule might also be optimal in terms of registers, power, or space….

A Canonical Greedy List Scheduling Algorithm

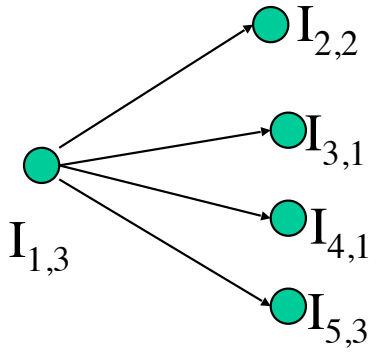
1. Assign a *Rank* (priority) to each instruction (or node).
2. Sort and build a priority *list* $\mathscr{L}$ of the instructions in non-decreasing order of Rank.
   - Nodes with smaller ranks occur earlier in this list
   - Smaller ranks imply higher priority
3. Greedily list-schedule $\mathscr{L}$.
   - An instruction is ready provided it has not been chosen earlier and all of its predecessors have been chosen and the appropriate latencies have elapsed.
   - Scan $\mathscr{L}$ iteratively and on each scan, choose the largest number of "ready" instructions from the front of the list subject to resource (FU) constraints.

Addressing Scheduling Questions

- Greediness helps in making sure that idle cycles don't remain if there are available instructions further "down stream."
  - If an instruction is available for a slot, then fill the slot
- Ranks help prioritize nodes such that choices made early on favor instructions with greater enabling power, so that there is no unforced idle cycle.
  - Ranks are an encoding for a scheduling heuristic
  - Ranks are based on characteristics of the operations, and allow the algorithm to compare operations

Applying the Canonical Greedy List Algorithm

Example: Consider the DAG shown below, where nodes are labeled (id, rank)



Sorting by ranks gives a list $\mathscr{L} = \langle i_{3,1}, i_{4,1}, i_{2,2}, i_{1,3}, i_{5,3} \rangle$

- The following slides apply the algorithm assuming 2 FUs.

more...

Applying the Canonical Greedy List Algorithm (cont.)

1. On the first scan
    1. $i_{1,3}$ is added to the schedule.
    2. No other ops can be scheduled, one empty slot
2. On the second and third scans
    1. $i_{3,1}$ and $i_{4,1}$ are added to the schedule
    2. All slots are filled, both FUs are busy
3. On the fourth and fifth scans
    1. $i_{2,2}$ and $i_{5,3}$ are added to the schedule
    2. All slots are filled, both FUs are busy
4. All ops have been scheduled

Instruction Scheduling: The Big Picture

1. Build a dependence graph, P
2. Compute a priority function over the nodes in P
3. Use list scheduling to construct a schedule, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose the highest priority ready operation and schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

Local List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
    if (Ready ≠ Ø) then
        remove an op from Ready
        S(op) ← Cycle
        Active ¬ Active  ∪ op

    Cycle ← Cycle + 1

    for each op ∈ Active
        if (S(op) + delay(op) ≤ Cycle) then
            remove op from Active
            for each successor s of op in P
                if (s is ready) then
                    Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

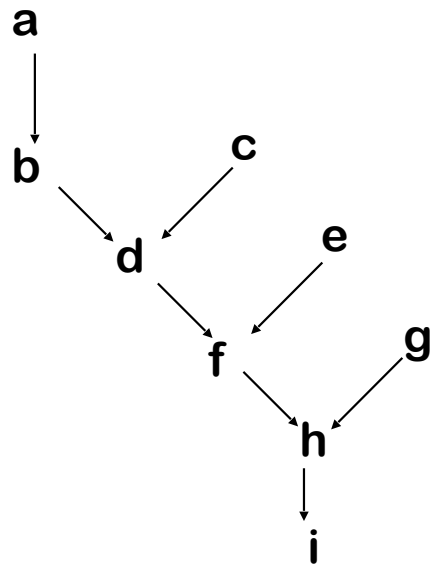
If successor's operands are ready, put it on Ready

Scheduling Example

1. Build the dependence graph

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r2
f:  mult     r1,r2  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



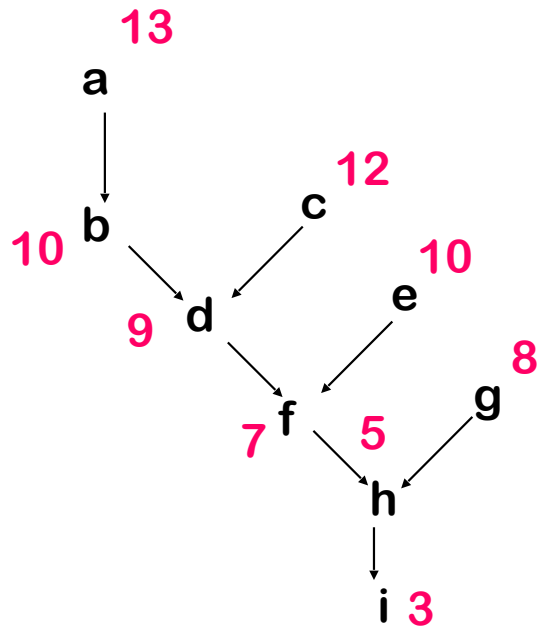
The Dependence Graph

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r2
f:  mult     r1,r2  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

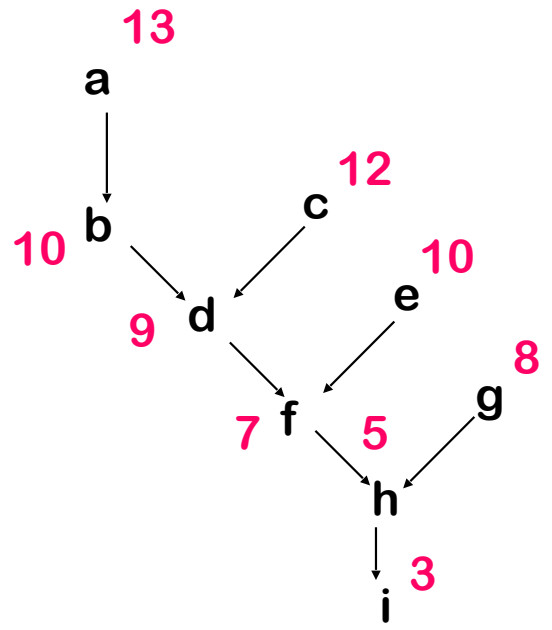
Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

New register name used

```
1)  a:   loadAI    r0,@w   ⇒ r1
2)  c:   loadAI    r0,@x   ⇒ r2
3)  e:   loadAI    r0,@y   ⇒ r3
4)  b:   add       r1,r1   ⇒ r1
5)  d:   mult      r1,r2   ⇒ r1
6)  g:   loadAI    r0,@z   ⇒ r2
7)  f:   mult      r1,r3   ⇒ r1
9)  h:   mult      r1,r2   ⇒ r1
11) i:   storeAI   r1      ⇒ r0,@w
```

The Code



The Dependence Graph

Detailed Scheduling Algorithm I

Idea: Keep a collection of worklists W[c], one per cycle

- We need MaxC = max delay + 1 such worklists
- Dependence graph is (N,E)

Code:

```
for each n ∈ N do begin count[n] := 0; earliest[n] = 0 end
for each (n1,n2) ∈ E do begin
      count[n2] := count[n2] + 1;
      successors[n1] := successors[n1] ∪ {n2};
end
for i := 0 to MaxC – 1 do W[i] := ∅;
Wcount := 0;
for each n ∈ N do
      if count[n] = 0 then begin
            W[0] := W[0] ∪ {n}; Wcount := Wcount + 1;
      end
c := 0;  // c is the cycle number
cW := 0;// cW is the number of the worklist for cycle c
instr[c] := ∅;
```

Detailed Scheduling Algorithm II

```
while Wcount > 0 do begin
    while W[cW] = ∅ do begin
        c := c + 1; instr[c] := ∅; cW := mod(cW+1,MaxC);
    end
    nextc := mod(c+1,MaxC);
    while W[cW] ≠ ∅ do begin
Priority → select and remove an arbitrary instruction x from W[cW];
        if ∃ free issue units of type(x) on cycle c then begin
            instr[c] := instr[c] ∪ {x}; Wcount := Wcount - 1;
            for each y ∈ successors[x] do begin
                count[y] := count[y] – 1;
                earliest[y] := max(earliest[y], c+delay(x));
                if count[y] = 0 then begin
                    loc := mod(earliest[y],MaxC);
                    W[loc] := W[loc] ∪ {y}; Wcount := Wcount + 1;
                end
            end
        else W[nextc] := W[nextc] ∪ {x};
    end
end
```

More List Scheduling

List scheduling breaks down into two distinct classes

Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready ⇒ all operands available	• Ready ⇒ latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in dependence graph (minimize registers)
- Breadth first in dependence graph (minimize interlocks)
- Prefer operation with most successors