# Instruction Scheduling and Software Pipelining - 3

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

# Global Acyclic Scheduling

- Average size of a basic block is quite small (5 to 20 instructions)
  - Effectiveness of instruction scheduling is limited
  - This is a serious concern in architectures supporting greater ILP
    - VLIW architectures with several function units
    - superscalar architectures (multiple instruction issue)
- Global scheduling is for a set of basic blocks
  - Overlaps execution of successive basic blocks
  - Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.

# Trace Scheduling

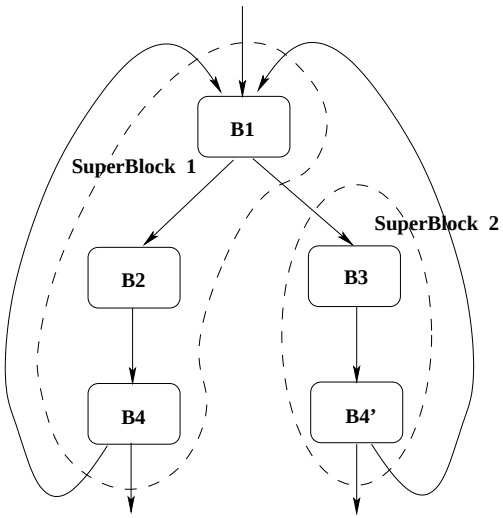
- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

# Superblock Scheduling

- A Superblock is a trace without side entrances
  - Control can enter only from the top
  - Many exits are possible
  - Eliminates several book-keeping overheads
- Superblock formation
  - Trace formation as before
  - Tail duplication to avoid side entrances into a superblock
  - Code size increases

# Superblock Example

- 5 cycles for the main trace and 6 cycles for the off-trace




(a) Control Flow Graph

| Time | Int. Unit 1 | | Int. Unit 2 | |
|---|---|---|---|---|
| 0 | i1: | r2 ← load a(r1) | i3: | r3 ← load b(r1) |
| 1 | | | | |
| 2 | i2: | if (r2!=0) goto i7 | i4: | r4 ← r3 + r7 |
| 3 | i5: | b(r1) ← r4 | i10: | r1 ← r1 + 4 |
| 4 | i9: | r5 ← r5 + r4 | i11: | if (r1<r6) goto i1 |

| | | | | |
|---|---|---|---|---|
| 3 | i7: | r4 ← r2 | i8: | b(r1) ← r2 |
| 4 | i9': | r5 ← r5 + r4 | i10': | r1 ← r1 + 4 |
| 5 | i11': | if (r1<r6) goto i1 | | |

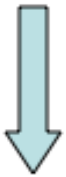
(b) Superblock Schedule

# Hyperblock Scheduling

- Superblock scheduling does not work well with control-intensive programs which have many control flow paths
- Hyperblock scheduling was proposed to handle such programs
- Here, the control flow graph is IF-converted to eliminate conditional branches
- IF-conversion replaces conditional branches with appropriate predicated instructions
- Now, control dependence is changed to a data dependence

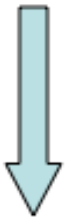
# IF-Conversion Example

```
for I = 1 to 100 do {
   if (A(I) <= 0) then contnue
   A(I) = B(I) + 3
}
```



```
for I = 1 to 100 do {
   p = (A(I) <= 0)
   (!p)  A(I) = B(I) + 3
}
```

```
      for I = 1 to N do {
S1:      A(I) = D(I) + 1
S2:      if (B(I) > 0) then
S3:         C(I) = C(I) + A(I)
S4:      else   D(I+1) = D(I+1) + 1
         end if
      }
```



```
      for I = 1 to N do {
S1:      A(I) = D(I) + 1
S2:      p = (B(I) > 0)
S3:      (p)  C(I) = C(I) + A(I)
S4:      (!p)  D(I+1) = D(I+1) + 1
      }
```

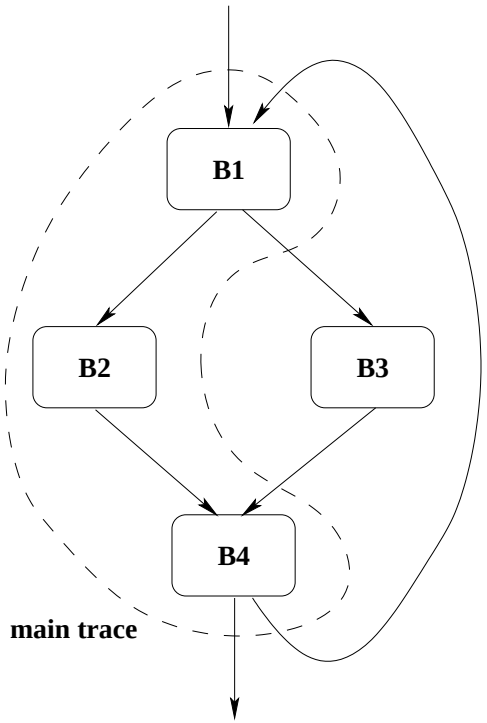
# Hyperblock Example Code

```
for (i=0; i < 100; i++)
{
    if (A[i] == 0)
      B[i] = B[i] + s;
    else
      B[i] = A[i];
    sum = sum + B[i];
}
```

(a) High-Level Code

| | | |
|---|---|---|
| | | %% r1 ← 0 |
| | | %% r5 ← 0 |
| | | %% r6 ← 400 |
| | | %% r7 ← s |
| B1: | i1: | r2 ← load a(r1) |
| | i2: | if (r2 != 0) goto i7 |
| B2: | i3: | r3 ← load b(r1) |
| | i4: | r4 ← r3 + r7 |
| | i5: | b(r1) ← r4 |
| | i6: | goto i9 |
| B3: | i7: | r4 ← r2 |
| | i8: | b(r1) ← r2 |
| B4: | i9: | r5 ← r5 + r4 |
| | i10: | r1 ← r1 + 4 |
| | i11: | if (r1 < r6) goto i1 |

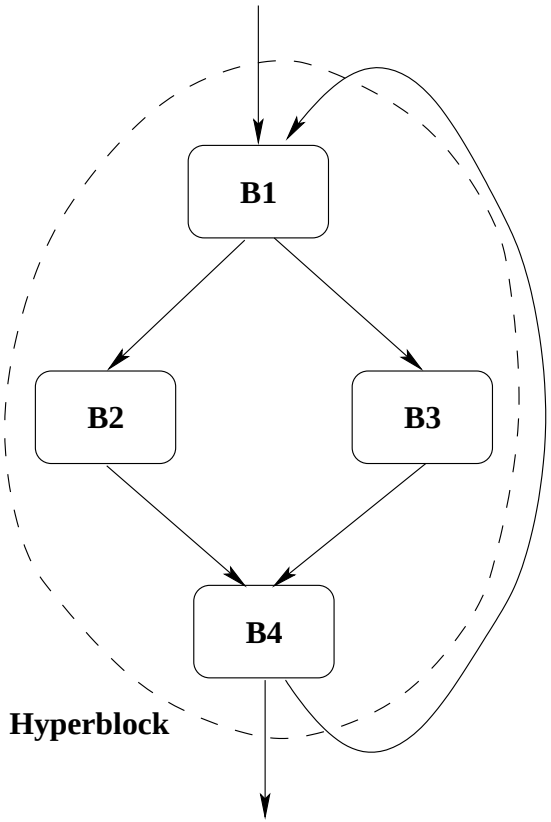
(b) Assembly Code




(c) Control Flow Graph

# Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2




(a) Control Flow Graph

| Time | Int. Unit 1 | | Int. Unit 2 | |
|---|---|---|---|---|
| 0 | i1: | r2 ← load a(r1) | i3: | r3 ← load b(r1) |
| 1 | | | | |
| 2 | i2': | p1 ← (r2 == 0) | i4: | r4 ← r3 + r7 |
| 3 | i5: | b(r1) ← r4, if p1 | i8: | b(r1) ← r2, if !p1 |
| 4 | i10: | r1 ← r1 + 4 | i7: | r4 ← r2, if !p1 |
| 5 | i9: | r5 ← r5 + r4 | i11: | if (r1<r6) goto i1 |

(b) Hyperblock Schedule

# Introduction to Software Pipelining

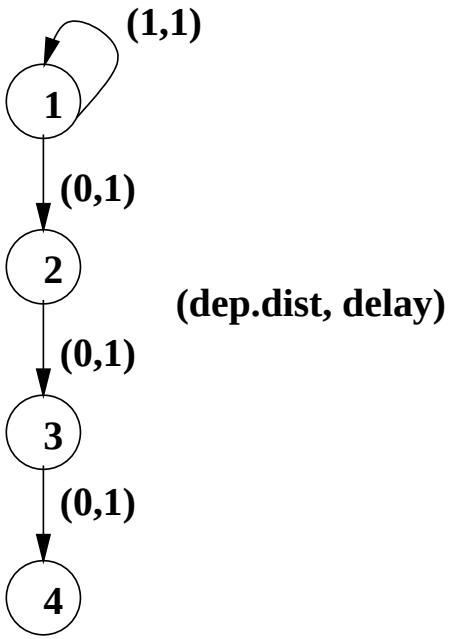
- Overlaps execution of instructions from multiple iterations of a loop
- Executes instructions from different iterations in the same pipeline, so that pipelines are kept busy without stalls
- Objective is to sustain a high initiation rate
  - Initiation of a subsequent iteration may start even before the previous iteration is complete
- Unrolling loops several times and performing global scheduling on the unrolled loop
  - Exploits greater ILP within unrolled iterations
  - Very little or no overlap across iterations of the loop

# Introduction to Software Pipelining - contd.

- More complex than instruction scheduling
- NP-Complete
- Involves finding initiation interval for successive iterations
  - Trial and error procedure
  - Start with minimum II, schedule the body of the loop using one of the approaches below and check if schedule length is within bounds
    - Stop, if yes
    - Try next value of II, if no
- Requires a modulo reservation table (GRT with II columns and R rows)
- Schedule lengths are dependent on II, dependence distance between instructions and resource contentions

# Software Pipelining Example-1

```
for (i=1; i<=n; i++) {
  a[i+1] = a[i] + 1;
  b[i] = a[i+1]/2;
  c[i] = b[i] + 3;
  d[i] = c[i]
}
```




**Iterations**

| TIME | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | S1 | | | | | | | |
| 2 | S2 | S1 | | | | | | |
| 3 | S3 | S2 | S1 | | | | | |
| 4 | S4 | S3 | S2 | S1 | | | | |
| 5 | | S4 | S3 | S2 | S1 | | | |
| 6 | | | S4 | S3 | S2 | S1 | | |
| 7 | | | | S4 | S3 | S2 | S1 | |
| 8 | | | | | S4 | S3 | S2 | |
| 9 | | | | | | S4 | S3 | |
| 10 | | | | | | | S4 | |

# Software Pipelining Example-2.1

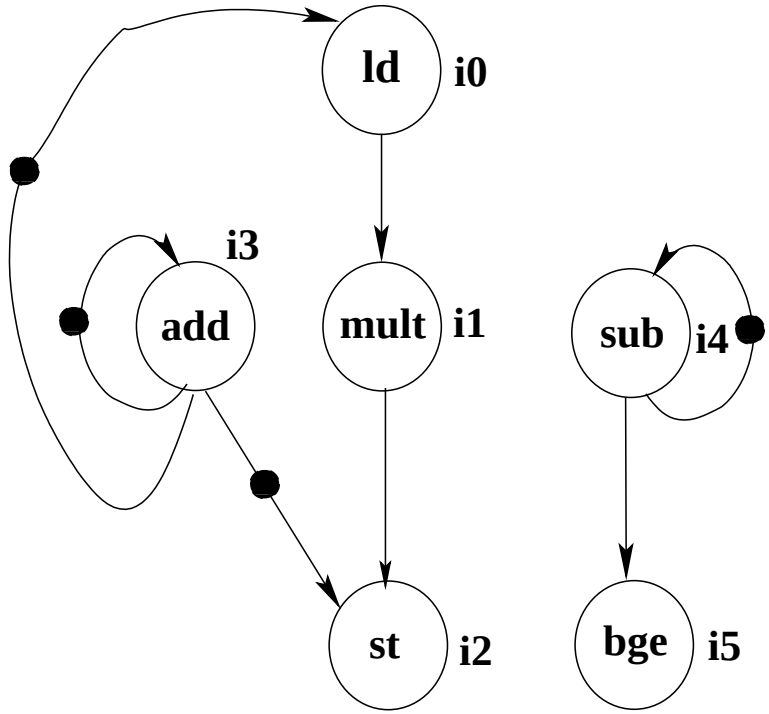
No. of tokens present on an arc indicates the dependence distance

```
for (i = 0; i < n; i++) {
    a[i] = s * a[i];
}
```

(a) High-Level Code

```
      % t0 ← 0 %
      % t1 ← (n-1) %
      % t2 ← s %
i0:   t3     ←  load a(t0)
i1:   t4     ←  t2 * t3
i2:   a(t0)  ←  t4
i3:   t0     ←  t0 + 4
i4:   t1     ←  t1 - 1
i5:   if (t1 ≥ 0) goto i0
```

(b) Instruction Sequence




(c) Dependence graph

Software Pipelining Example

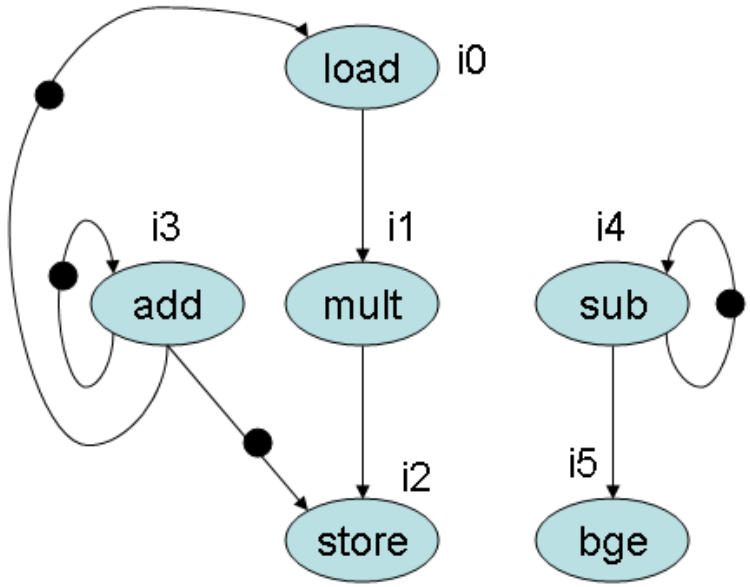
# Software Pipelining Example-2.2

- Number of tokens present on an arc indicates the dependence distance
- Assume that the possible dependence from i2 to i0 can be disambiguated
- Assume 2 INT units (latency 1 cycle), 2 FP units (latency 2 cycles), and 1 LD/STR unit (latency 2 cycles/1 cycle)
- Branch can be executed by INT units
- Acyclic schedule takes 5 cycles (see figure)
- Corresponds to an initiation rate of 1/5 iteration per cycle
- Cyclic schedule takes 2 cycles (see figure)

# Acyclic and Cyclic Schedules

## Acyclic Schedule

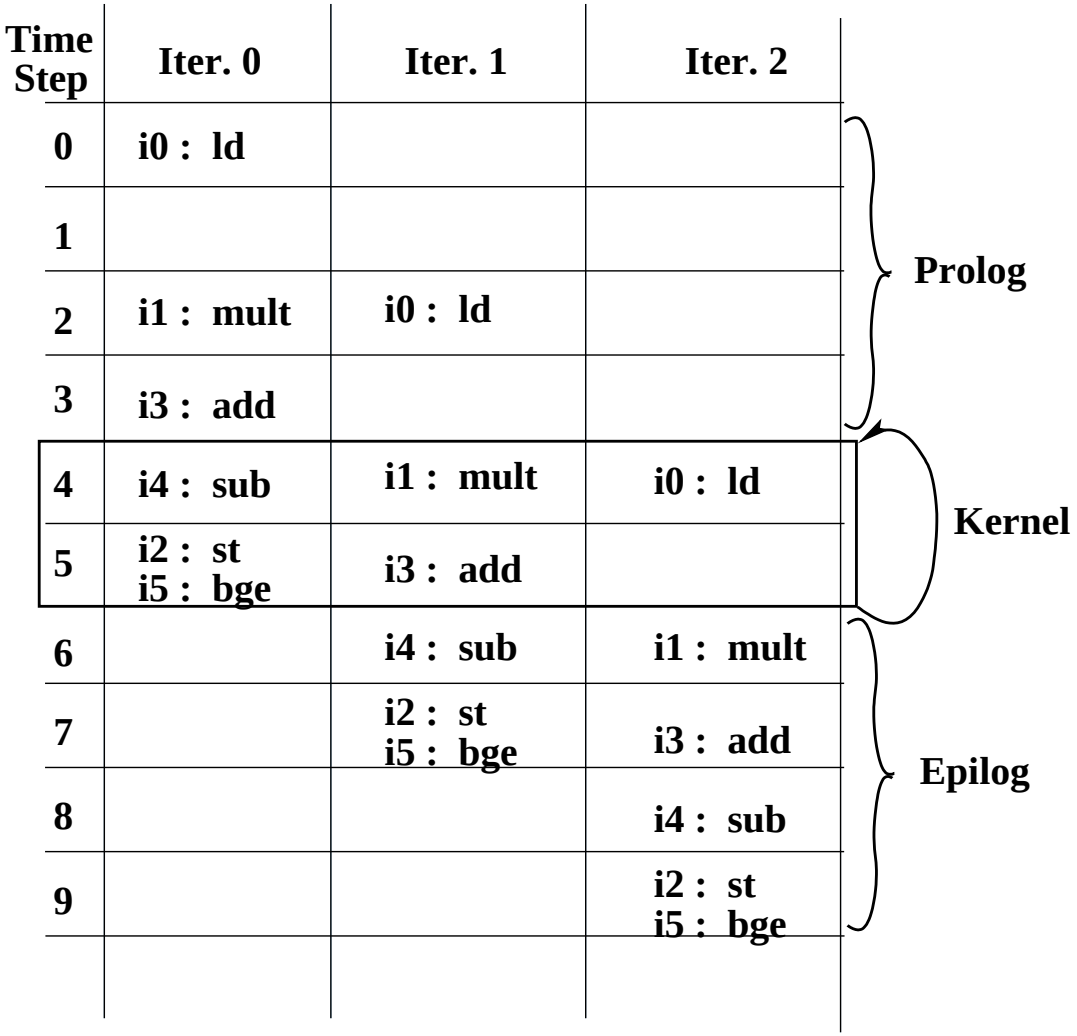
| | |
|---|---|
| 0 | i0: load |
| 1 | |
| 2 | i1: mult, i3: add, i4: sub |
| 3 | |
| 4 | i2: store, i5: bge |




## Cyclic Schedule

| | | | |
|---|---|---|---|
| 4 | i4: sub | i1: mult | i0: load |
| 5 | i2: store<br>i5: bge | i3: add | |

# Software Pipelining Example-2.3



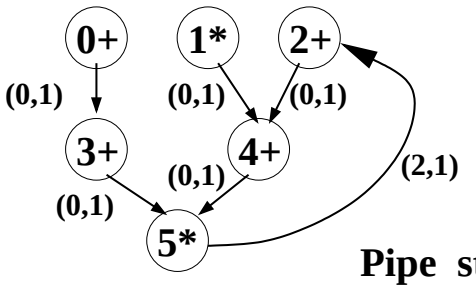

A Software Pipelined Schedule with II = 2

# Software Pipelining Example-3

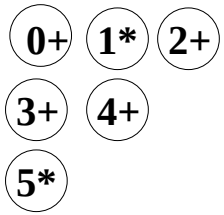
```
for i = 1 to n {
   0: t0[i] = a[i] + b[i];
   1: t1[i] = c[i] * const1;
   2: t2[i] = d[i] + e[i−2];
   3: t3[i] = t0[i] + c[i];
   4: t4[i] = t1[i] + t2[i];
   5:   e[i] = t3[i] * t4[i];
}
```

Program

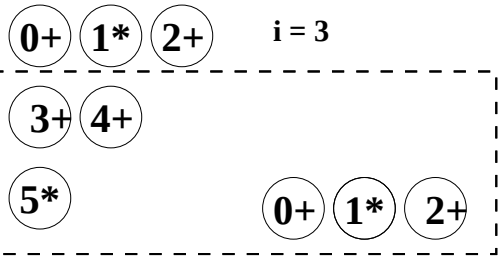
Dependence Graph



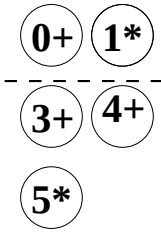

i = 1




i = 2




i = 3




Loop unrolled to reveal the software pipeline

Pipe stages

| time | PS0 | PS1 |
|---|---|---|
| 0 | 3+ 4+ | |
| 1 | 5* | 0+ 1* 2+ |

2 multipliers, 2 adders, 1 cluster, single cycle operations

# Automatic Parallelization - 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

# Automatic Parallelization

- Automatic conversion of sequential programs to parallel programs by a compiler
- Target may be a vector processor (vectorization), a multi-core processor (concurrentization), or a cluster of loosely coupled distributed memory processors (parallelization)
- Parallelism extraction process is normally a source-to-source transformation
- Requires dependence analysis to determine the dependence between statements
- Implementation of available parallelism is also a challenge
  - For example, can all the iterations of a 2-nested loop be run in parallel?

# Example 1

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}
```

can be converted to

```
X(1:100) = X(1:100) + Y(1:100)
```

The above code can be run on a vector processor in O(1) time. The vectors X and Y are fetched first and then the vector X is written into

# Example 2

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

forall I = 1 to 100 do {
   X(I) = X(I) + Y(I)
```

The above code can be run on a multi-core processor with all the 100 iterations running as separate threads. Each thread "owns" a different I value

# Example 3

```
for I = 1 to 100 do {
   X(I+1) = X(I) + Y(I)
}
```

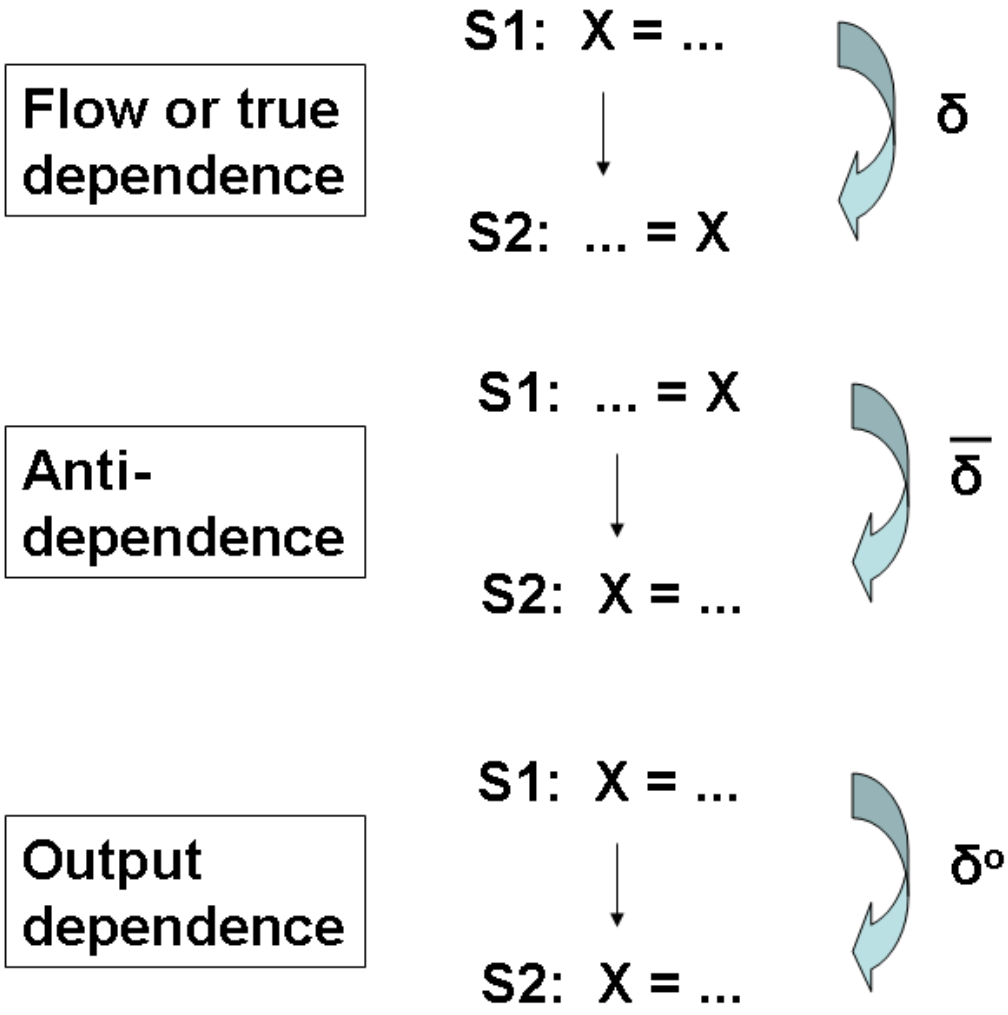
cannot be converted to

```
X(2:101) = X(1:100) + Y(1:100)
```

because of dependence as shown below

```
X(2) = X(1) + Y(1)
X(3) = X(2) + Y(2)
X(4) = X(3) + Y(3)
...
```

# Data Dependence Relations



# Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence (direction vector)
- There is one direction vector component for each loop in a nest of loops
- The *data dependence direction vector* (or direction vector) is $\Psi = (\Psi_1, \Psi_2, ..., \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- Forward or "<" direction means dependence from iteration $i$ to $i + k$ (*i.e.,* computed in iteration $i$ and used in iteration $i + k$)
- Backward or ">" direction means dependence from iteration $i$ to $i - k$ (*i.e.,* computed in iteration $i$ and used in iteration $i - k$). This is not possible in single loops and possible in two or higher levels of nesting
- Equal or "=" direction means that dependence is in the same iteration (*i.e.,* computed in iteration $i$ and used in iteration $i$)

# Direction Vector Example 1

```
for J = 1 to 100 do {
S:    X(J) = X(J) +c
}
```

$S\ \bar{\delta}_{=}\ S$

X(1) = X(1) +c
X(2) = X(2)+c

```
for J = 1 to 99 do {
S:    X(J+1) = X(J) +c
}
```

$S\ \delta_{<}\ S$

X(2) = X(1) +c
X(3) = X(2)+c

```
for J = 1 to 99 do {
S:    X(J) = X(J+1) +c
}
```

$S\ \bar{\delta}_{<}\ S$

X(1) = X(2) +c
X(2) = X(3)+c

```
for J = 99 downto 1 do {
S:    X(J) = X(J+1) +c
}
```

$S\ \delta_{<}\ S$

X(99) = X(100) +c
X(98) = X(99)+c
note '-ve' increment

```
for J = 2 to 101 do {
S:    X(J) = X(J-1) +c
}
```

$S\ \delta_{<}\ S$

X(2) = X(1) +c
X(3) = X(2)+c