Instruction Scheduling

Increasing Parallelism

Basic-Block Scheduling

Data-Dependency Graphs

The Model

- A very-long-instruction-word machine allows several operations to be performed at once.
  - Given: a list of “resources” (e.g., ALU) and delay required for each instruction.
- Schedule the intermediate code instructions of a basic block to minimize the number of machine instructions.

Register/Parallelism Tradeoff

- The more registers you use, the more parallelism you can get.
- For a basic block, SSA form = maximal parallelism.

Example

Assume 2 arithmetic operations per instruction

```
a = b+c
e = a+d
a = b-c
f = a+d
```

$\rightarrow$ Don’t reuse a $\rightarrow$

```
a1 = b+c
e = a1+d
a2 = b-c
f = a2+d
```

ALU1	ALU2
`a = b+c`	
`e = a+d`	`a = b-c`
`f = a+d`	

ALU1	ALU2
`a1 = b+c`	`a2 = b-c`
`e = a1+d`	`f = a2+d`

More Extreme Example

```
for (i=0; i<N; i++) {
     t = a[i]+1;
     b[i] = t*t;
} /* no parallelism */

for (i=0; i<N; i++) {
     t[i] = a[i]+1;
     b[i] = t[i]*t[i];
} /* All iterations can be
     executed in parallel */
```

Rules for Instruction Scheduling

1. Don't change the set of operations performed (on any computation path).
2. Make sure interfering operations are performed in the same order.
   - *Data dependence*.

Kinds of Data Dependence

1. Write-read (*true dependence*):
   - A read of x must continue to follow the previous write of x.
2. Read-write (*antidependence*):
   - A write of x must continue to follow previous reads of x.
3. Write-write (*output dependence*):
   - Writes of x must stay in order.

Eliminating Data Dependences

- Only true dependences cannot be eliminated.
- Eliminate output or anti- dependences by writing into different variables.

A Machine Model

- Arithmetic is register*register -> register.
  - Requires one unit of ALU.
- Loads (LD) and Stores (ST).
  - Requires one unit of MEM (memory buffer).

Timing in Our Machine Model

- Arithmetic requires one clock cycle ("*clock*").
- Store requires 1 clock.
- Load requires 2 clocks to complete .
  - But we can store into the same memory location at the next clock.
  - And one LD can be issued at each clock.

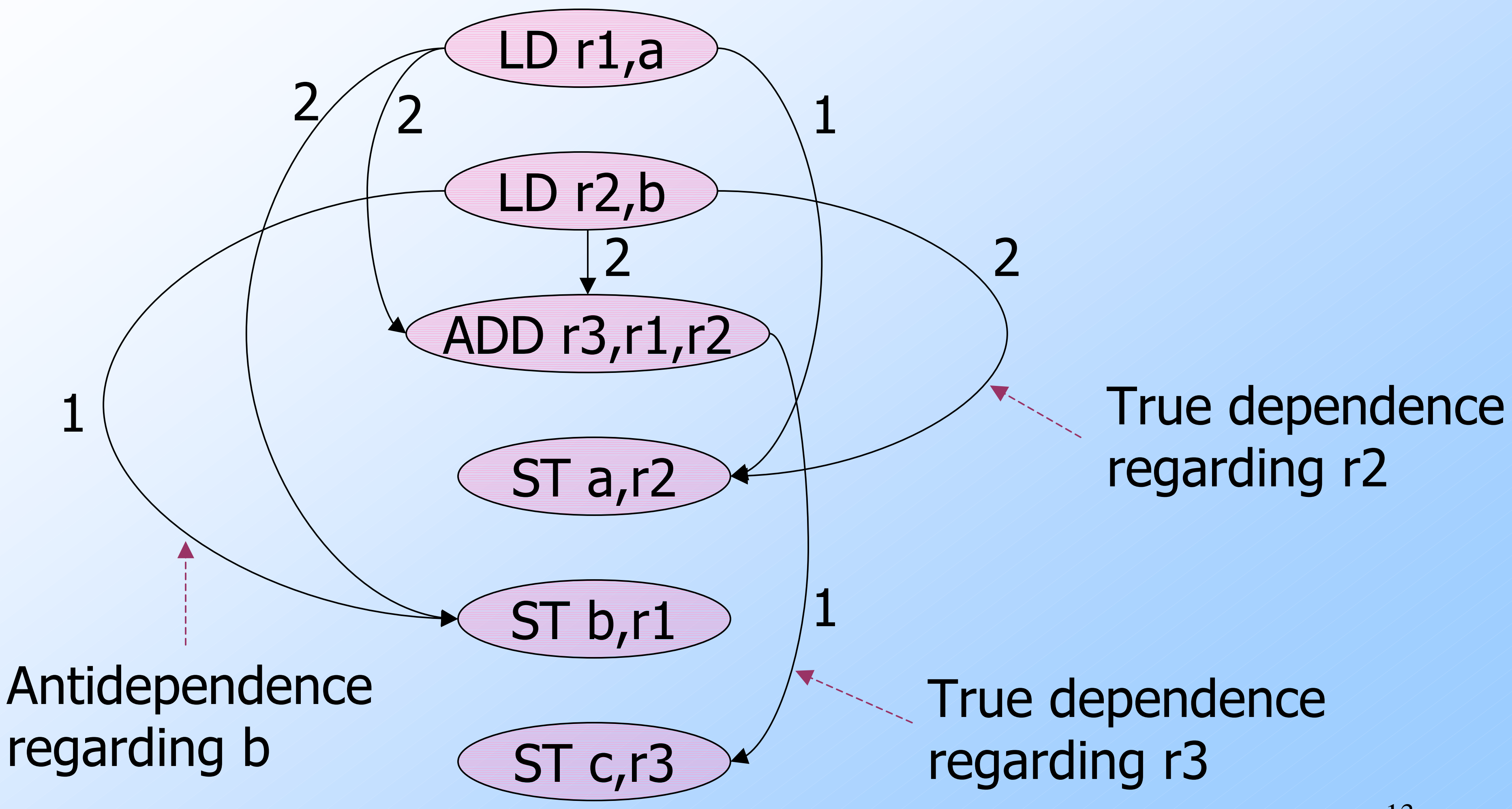
Data-Dependence Graphs

- Nodes = machine instructions.
- Edge i -> j if instruction (j) has a data dependence on instruction (i).
- Label an edge with the minimum delay interval between when (i) may initiate and when (j) may initiate.
  - Delay measured in clock cycles.

Example

	Resource
`LD  r1, a`	MEM
`LD  r2, b`	MEM
`ADD r3, r1, r2`	ALU
`ST  a r2`	MEM
`ST  b r1`	MEM
`ST  c r3`	MEM

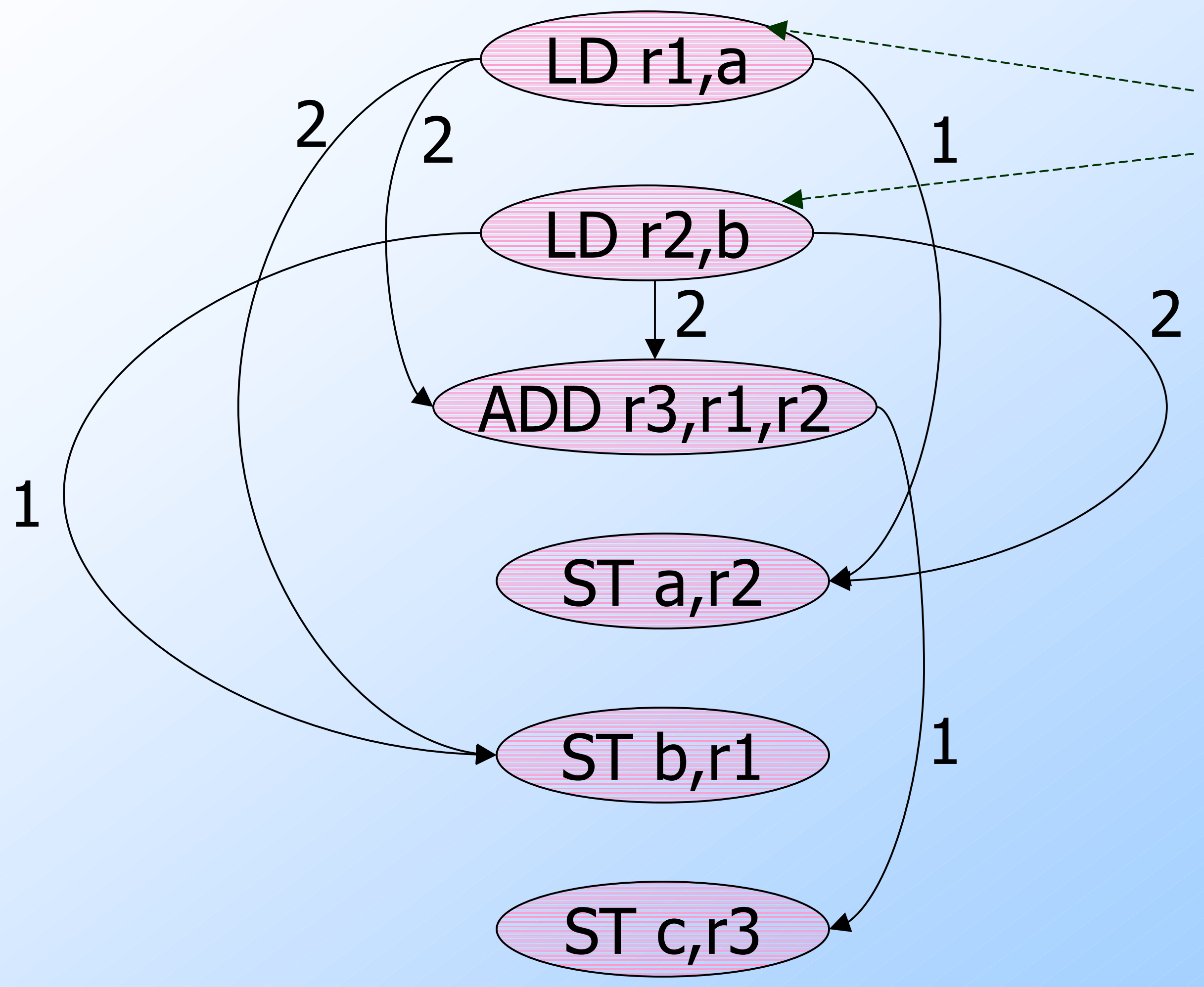
Example: Data-Dependence Graph


LD r1,a
LD r2,b
ADD r3,r1,r2
ST a,r2
ST b,r1
ST c,r3
2
2
1
2
2
1
1
True dependence regarding r2
Antidependence regarding b
True dependence regarding r3

Scheduling a Basic Block

- *List scheduling* is a simple heuristic.
- Choose a *prioritized topological order*.
  1. Respects the edges in the data-dependence graph ("topological").
  2. Heuristic choice among options, e.g., pick first the node with the longest path extending from that node ("prioritized").

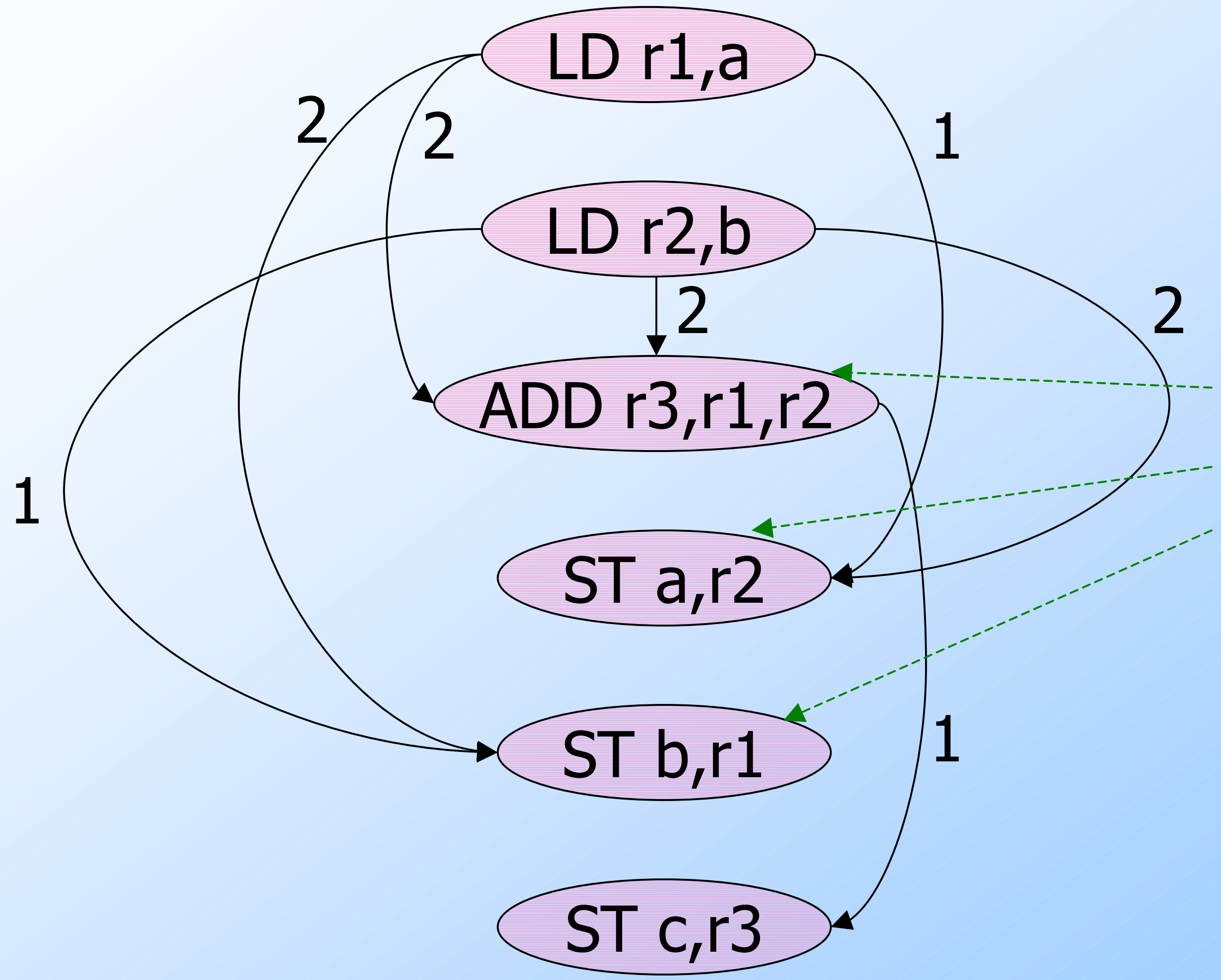
Example: Data-Dependence Graph



Either of these could be first --- no predecessors, paths of length 3.

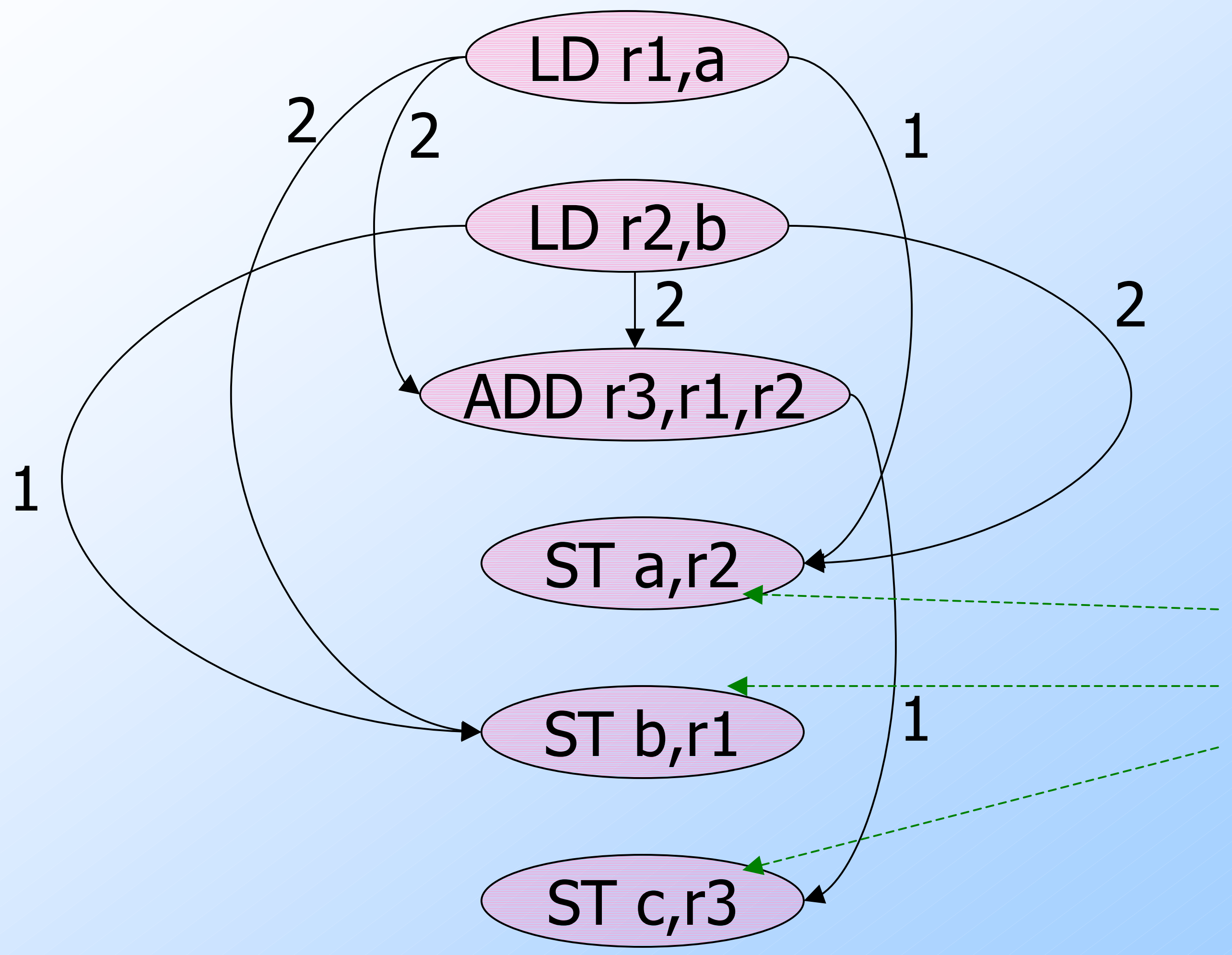
Pick LD r1,a first. No other node is enabled; so pick LD r2,b second.

Example: Data-Dependence Graph



Now, these three are enabled. Pick the ADD, since it has the longest path extending.

Example: Data-Dependence Graph

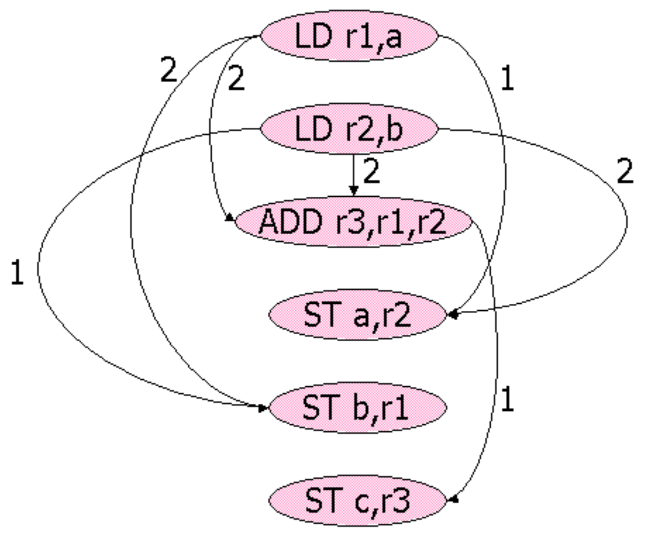


These three can now occur in any order. Pick the order shown.

Using the List to Schedule

- For each instruction in list order, find the earliest clock cycle at which it can be scheduled.
- Consider first when predecessors in the dependence graph were scheduled; that is a lower bound.
- Then, if necessary, delay further until the necessary resources are available.

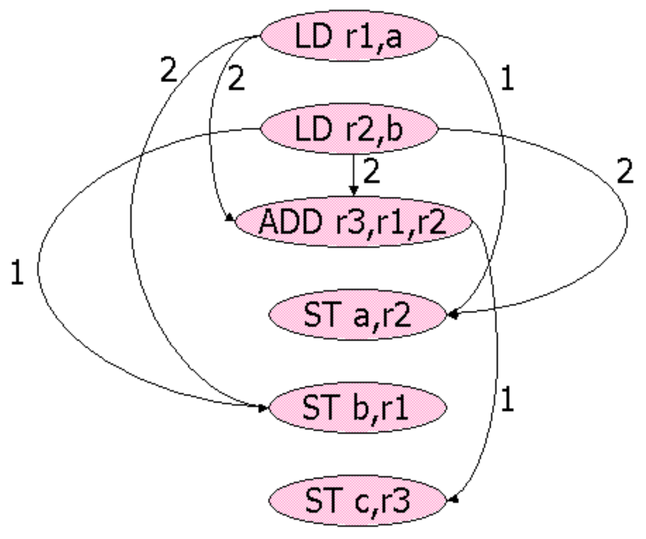
Example: Making the Schedule



```
LD r1,a
```

LD r1,a:
clock 1 earliest.
MEM available.

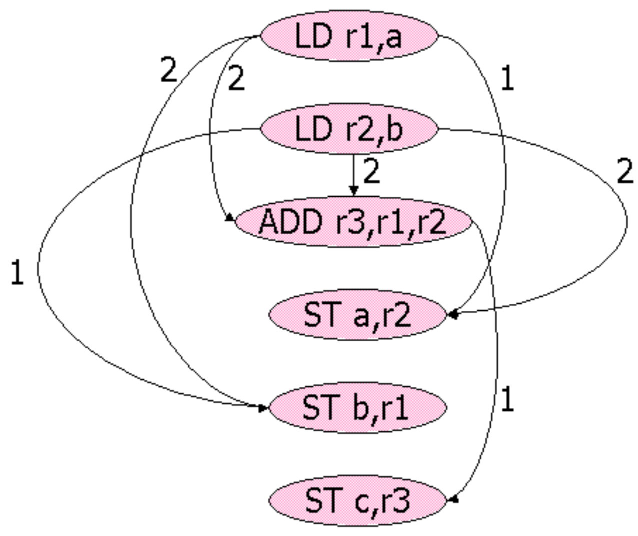
Example: Making the Schedule



```
LD r1,a
LD r2,b
```

LD r2,b:
clock 1 earliest.
MEM not available.
Delay to clock 2.

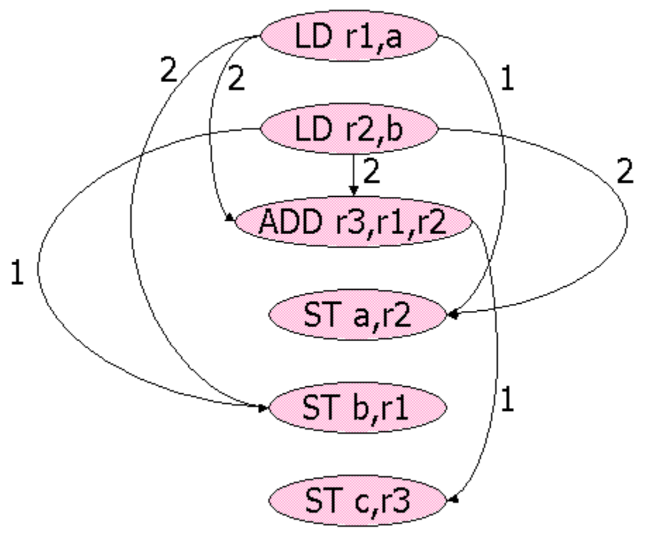
Example: Making the Schedule



```
LD r1,a
LD r2,b


ADD r3,r1,r2
```

ADD r3,r1,r2:
clock 4 earliest.
ALU available.

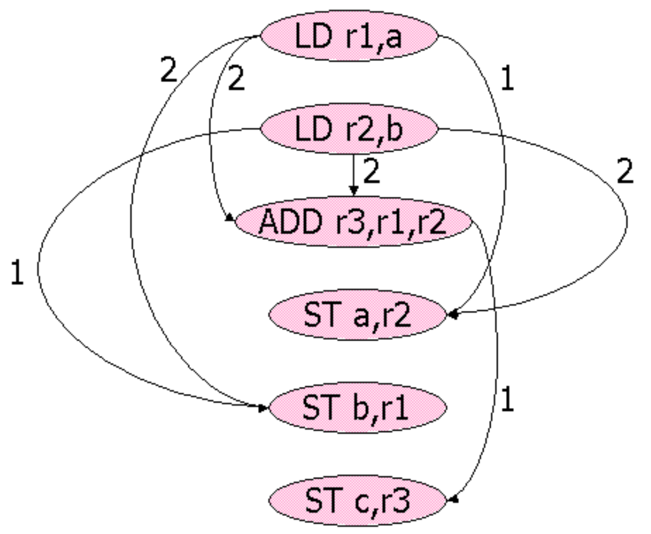
Example: Making the Schedule



```
LD r1,a
LD r2,b

ADD r3,r1,r2    ST a,r2
```

ST a,r2:
clock 4 earliest.
MEM available.

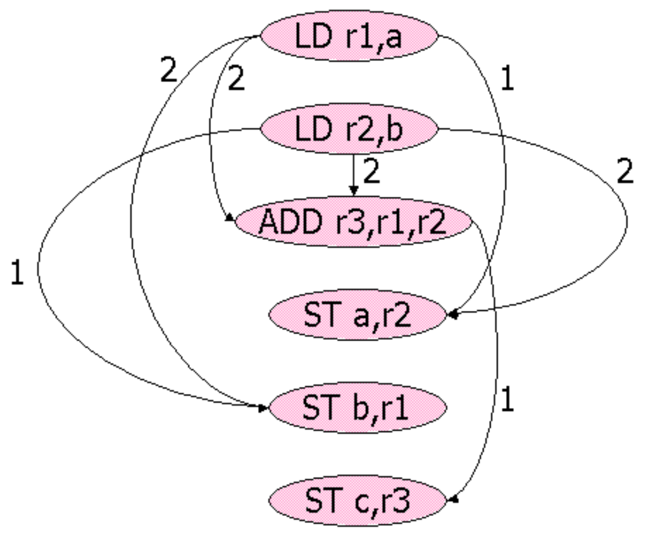
Example: Making the Schedule



```
LD r1,a
LD r2,b
ST b,r1
ADD r3,r1,r2   ST a,r2
```

ST b,r1:
clock 3 earliest.
MEM available.

Example: Making the Schedule



```
LD r1,a
LD r2,b
ST b,r1
ADD r3,r1,r2    ST a,r2
ST c,r3
```

ST c,r3:
clock 5 earliest.
MEM available.

New Topic: Global Code Motion

- We can move code from one basic block to another, to increase parallelism.
  - Must obey all dependencies.
- *Speculative execution* (execute code needed in only one branch) OK if operation has no side effects.
  - Example: LD into an unused register.

Upwards Code Motion

- Can move code to a dominator if:
  1. Dependencies satisfied.
  2. No side effects unless source and destination nodes are *control equivalent* :
     - Destination dominates source.
     - Source postdominates destination.
- Can move to a nondominator if *compensation code* is inserted.

Downwards Code Motion

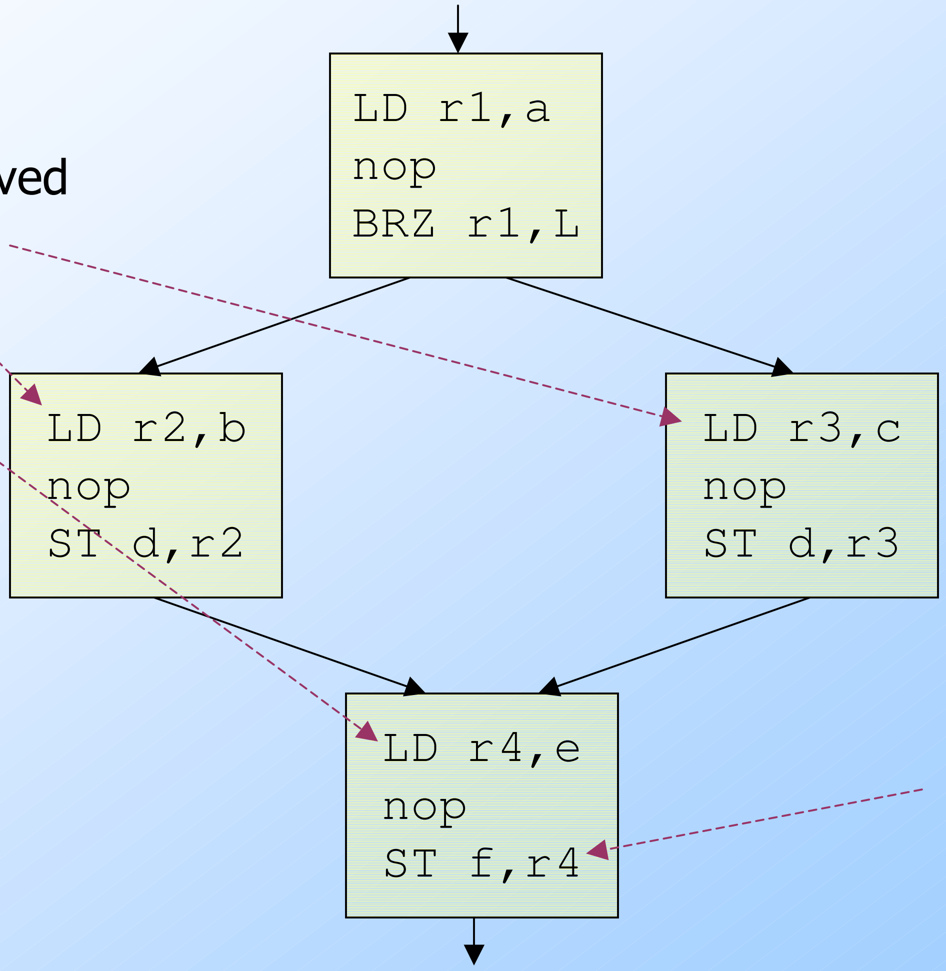
- Can move to a postdominator if:
  1. Dependencies satisfied.
  2. No side effects unless control equivalent.
- Can move to a non-postdominator if compensation code added.

Machine Model for Example

- Same timing as before.
  - LD = 2 clocks, others = 1 clock.
- Machine can execute any two instructions in parallel.

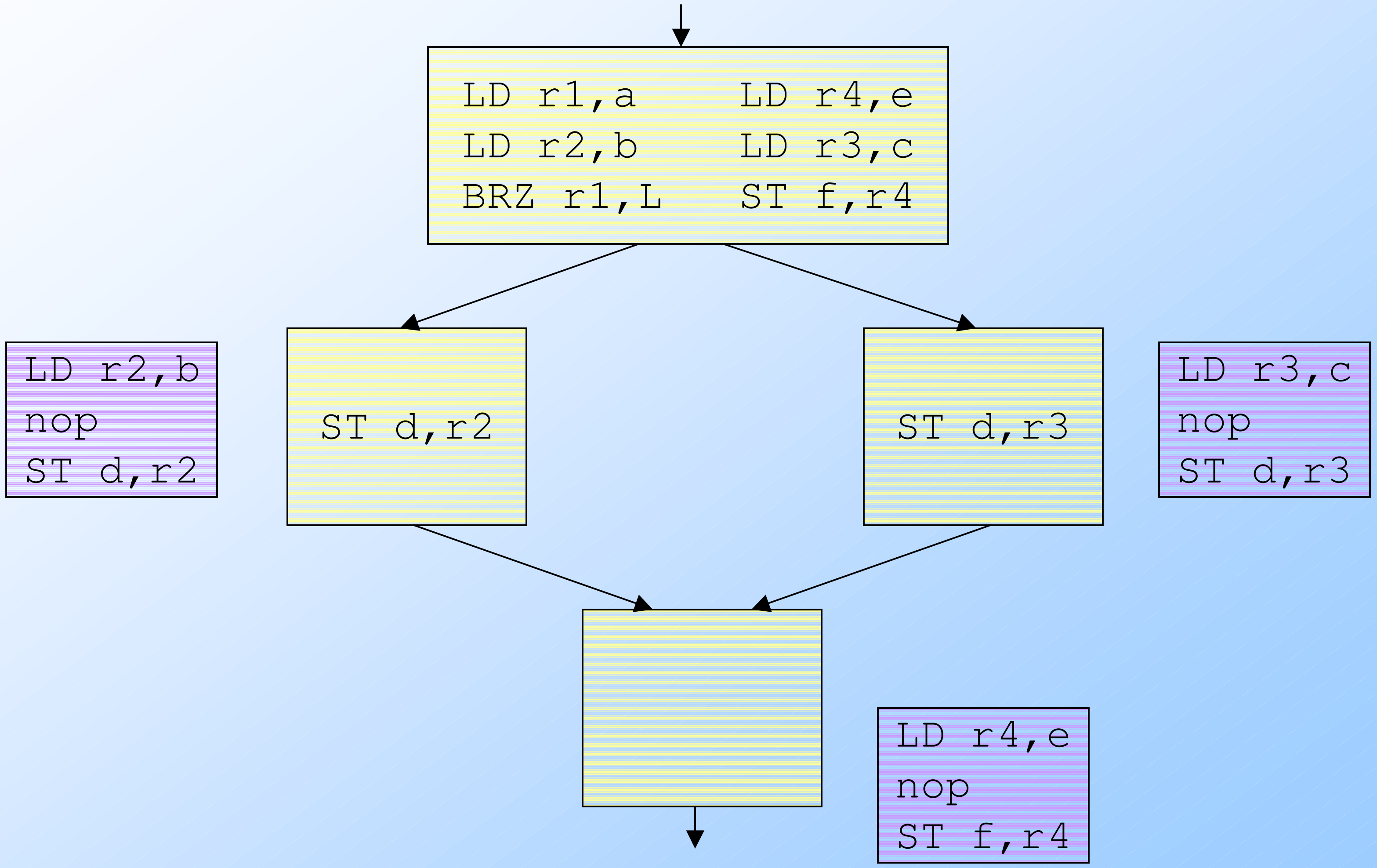
Example: Code Motion

These LD's are side-effect free and can be moved to entry node.



We can move this ST to the entry if we move LD r4 as well, because this node is control-equivalent to the entry.

Example: Code Motion --- (2)



Software Pipelining

- Obtain parallelism by executing iterations of a loop in an overlapping way.
- We'll focus on simplest case: the *do-all* loop, where iterations are independent.
- Goal: Initiate iterations as frequently as possible.
- Limitation: Use same schedule and delay for each iteration.

Machine Model

- Same timing as before (LD = 2, others = 1 clock).
- Machine can execute one LD or ST and one arithmetic operation (including branch) at any one clock.
  - I.e., we're back to one ALU resource and one MEM resource.

Example

```
for (i=0; i<N; i++)
    B[i] = A[i];
```

- r9 holds 4N; r8 holds 4*i.

```
L:  LD r1, a(r8)
    nop
    ST b(r8), r1
    ADD r8, r8, #4
    BLT r8, r9, L
```

Notice: data dependences force this schedule. No parallelism is possible.

Let’s Run 2 Iterations in Parallel

- Focus on operations; worry about registers later.

```
LD
nop    LD
ST     nop
ADD    ST
BLT    ADD
       BLT
```

Oops --- violates ALU resource constraint.

Introduce a NOP

```
LD
nop     LD
ST      nop     LD
ADD     ST      nop
nop     ADD     ST
BLT     nop     ADD
        BLT     nop
                BLT
```

Add a third iteration. Several resource conflicts arise.

Is It Possible to Have an Iteration Start at Every Clock?

- Hint: No.
- Why?
- An iteration injects 2 MEM and 2 ALU resource requirements.
  - If injected every clock, the machine cannot possibly satisfy all requests.
- Minimum delay = 2.

A Schedule With Delay 2

```
LD
nop
nop    LD
ST     nop
----------------------------
ADD    nop    LD
BLT    ST     nop
----------------------------
       ADD    nop    LD
       BLT    ST     nop
----------------------------
              ADD    nop
              BLT    ST
                     ADD
                     BLT
```

Initialization

Identical iterations of the loop

Coda

Assigning Registers

- We don’t need an infinite number of registers.
- We can reuse registers for iterations that do not overlap in time.
- But we can’t just use the same old registers for every iteration.

Assigning Registers --- (2)

- The inner loop may have to involve more than one copy of the smallest repeating pattern.
  - Enough so that registers may be reused at each iteration of the expanded inner loop.
- Our example: 3 iterations coexist, so we need 3 sets of registers and 3 copies of the pattern.

Example: Assigning Registers

- Our original loop used registers:
  - r9 to hold a constant 4N.
  - r8 to count iterations and index the arrays.
  - r1 to copy a[i] into b[i].
- The expanded loop needs:
  - r9 holds 4N.
  - r6, r7, r8 to count iterations and index.
  - r1, r2, r3 to copy certain array elements.

The Loop Body

To break the loop early

Each register handles every third element of the arrays.

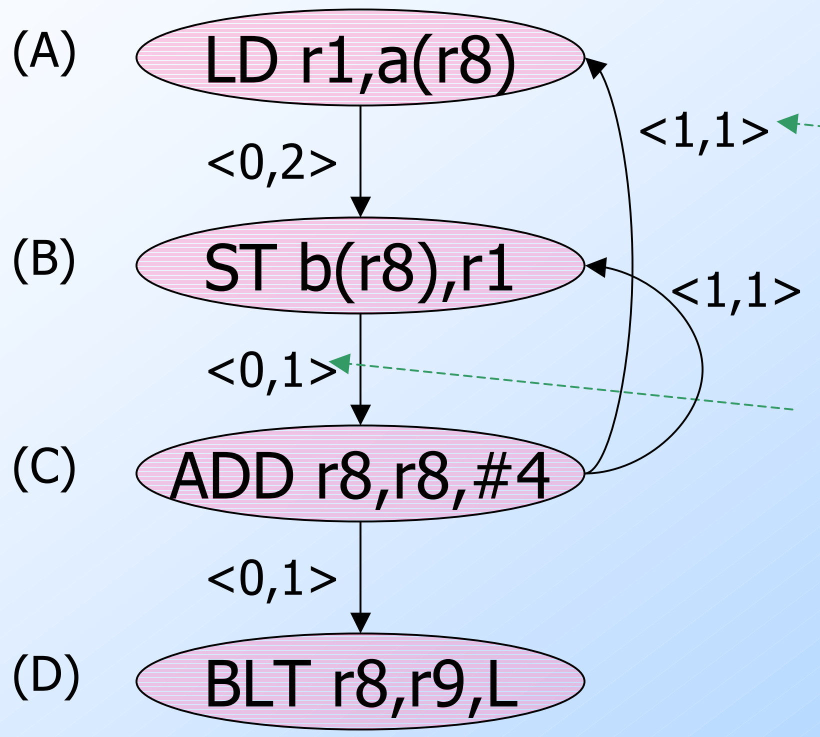
```
         Iteration i              Iteration i + 1          Iteration i + 2

L:   |  ADD r8,r8,#12       |  nop                  |  LD   r3,a(r6)      |
     |  BGE r8,r9,L'        |  ST   b(r7),r2        |  nop                |
     |----------------------|  ADD r7,r7,#12        |  nop                |
     |  LD   r1,a(r8)       |  BGE r7,r9,L''        |  ST   b(r6),r3      |
     |  nop                 |-----------------------|  ADD r6,r6,#12      |
     |  nop                 |  LD   r2,a(r7)        |  BLT r6,r9,L        |
     |  ST   b(r8),r1       |  nop                  |                     |

         Iteration i + 3          Iteration i + 4
```

L’ and L” are places for appropriate codas.

Cyclic Data-Dependence Graphs

- We assumed that data at an iteration depends only on data computed at the same iteration.
  - Not even true for our example.
    - r8 computed from its previous iteration.
    - But it doesn't matter in this example.
- Fixup: edge labels have two components: (iteration change, delay).

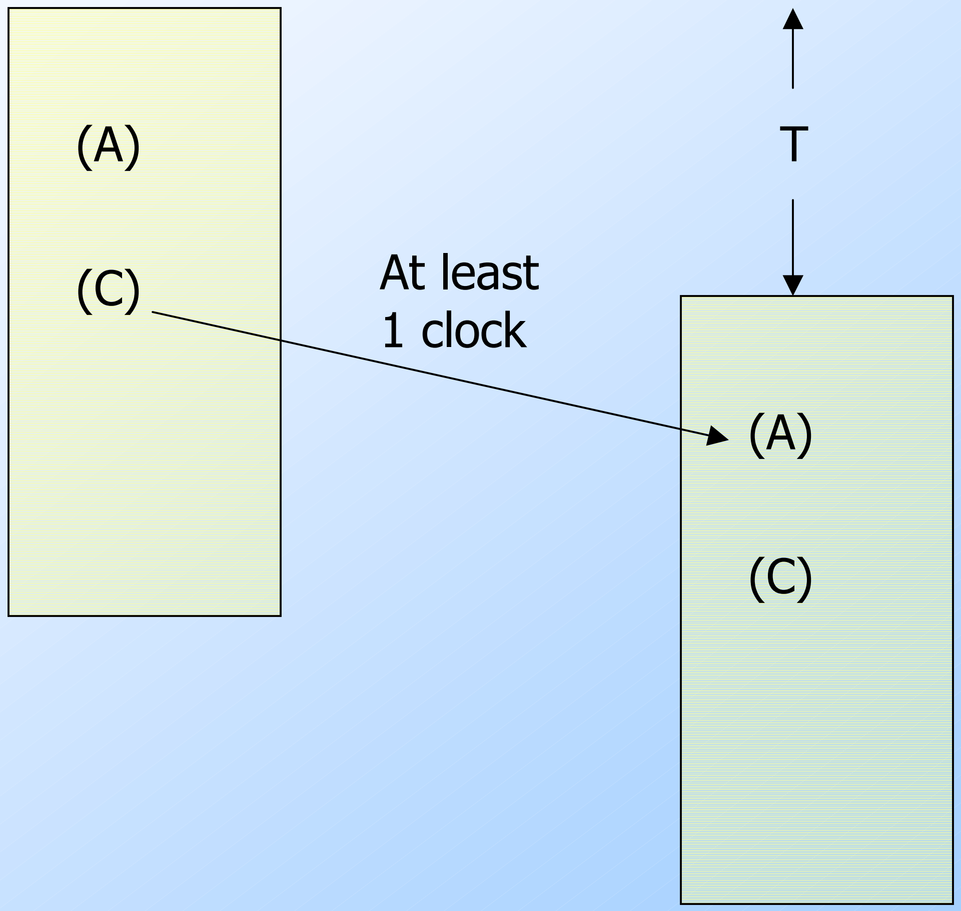
Example: Cyclic D-D Graph



(A) must wait at least one clock after the (C) from the previous iteration.

(C) must wait at least one clock after the (B) from the same iteration.

Inter-Iteration Constraint



Matrix of Delays

- Let T be the delay between the start times of one iteration and the next.
- Replace edge label <i,j> by delay j-iT.
- Compute, for each pair of nodes n and m the total delay along the longest acyclic path from n to m.
- Gives upper and lower bounds relating the times to schedule n and m.

The Schedule

- Iterations commence at times 0, T, 2T,…
- A statement corresponding to node n is scheduled S(n) clocks after the commencement of its iteration.

Example: Delay Matrix

	A	B	C	D
A		2		
B			1	
C	1-T	1-T		1
D				

Edges

	A	B	C	D
A		2	3	4
B	2-T		1	2
C	1-T	3-T		1
D				

Acyclic Transitive Closure

Note: Implies $T \geq 4$. If $T=4$, then A (LD) must be 2 clocks before B (ST). If $T=5$, A can be 2-3 clocks before B.

$S(B) \geq S(A)+2$

$S(A) \geq S(B)+2-T$

$S(B)-2 \geq S(A) \geq S(B)+2-T$

A Query

- When we considered software pipelining, we found that it was possible to initiate an iteration every 2 clocks.
- Now, we’ve concluded that $T \geq 4$.
- What’s going on?