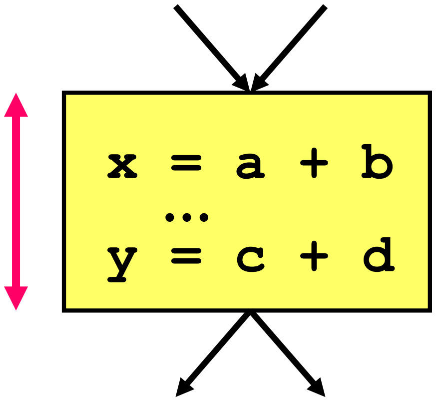
Lecture 20

Global Scheduling

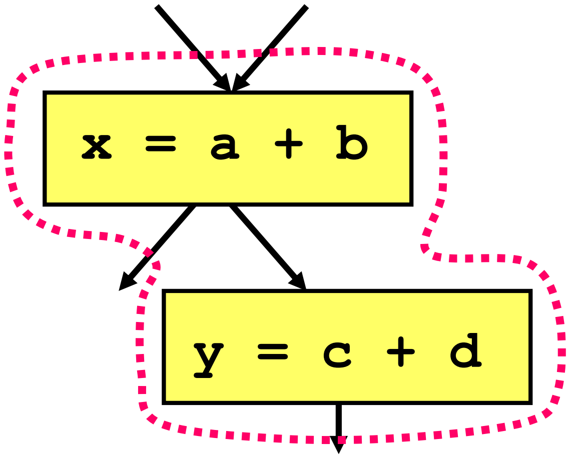
I. Legal code motions
II. Basic Algorithm

[ALSU 10.4]

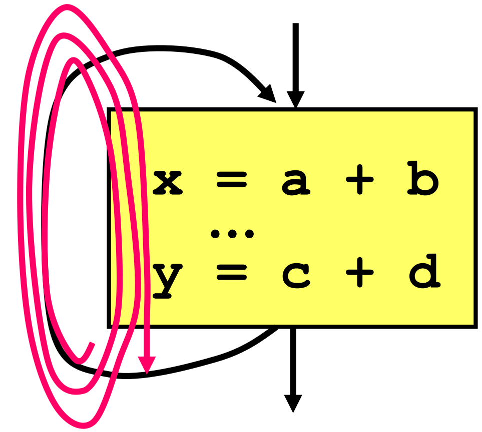
Scheduling Roadmap



List Scheduling:
- *within* a basic block
  (prior lecture)



Global Scheduling:
- *across* basic blocks



Software Pipelining:
- *across* loop iterations

Review: List Scheduling

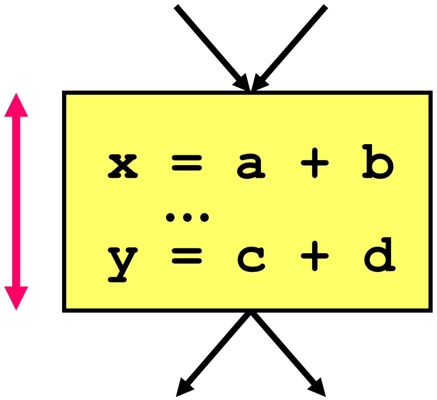
- The most common technique for scheduling instructions within a basic block

We don’t need to worry about:

- control flow

We do need to worry about:

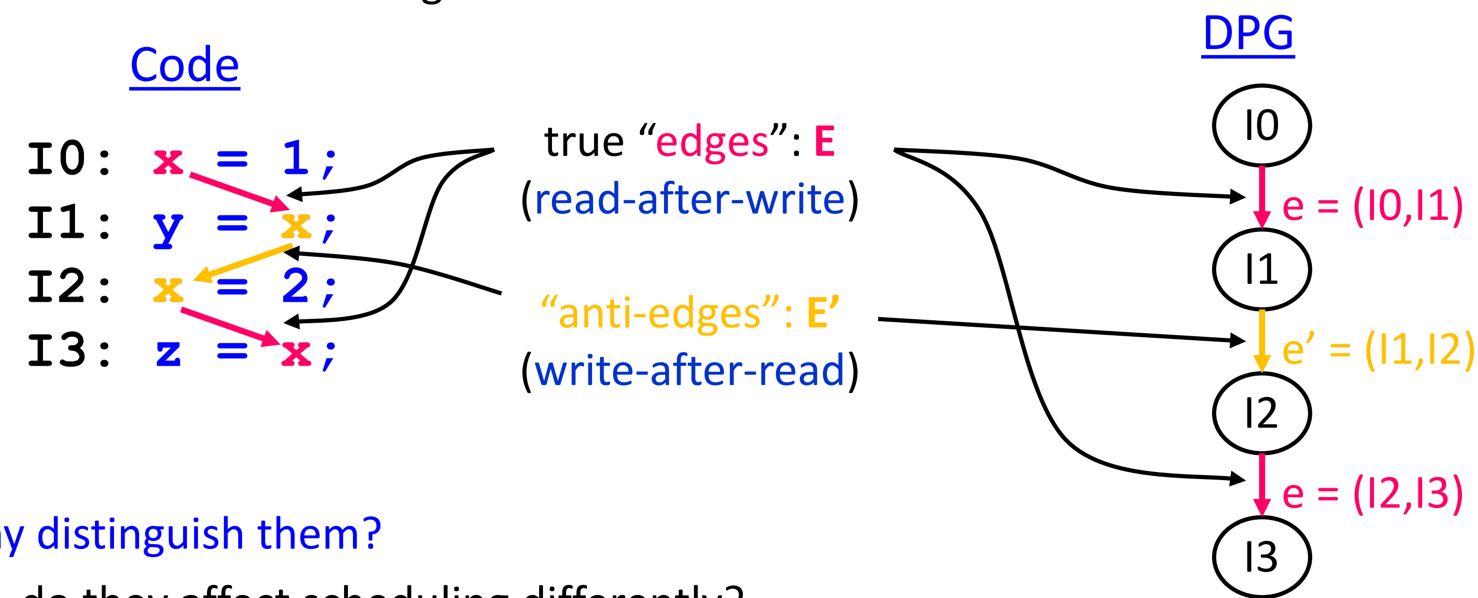
- data dependences
- hardware resources



- Even without control flow, the problem is still NP-hard

Review: Representing Data Dependences: The Data Precedence Graph (DPG)

- Two different kinds of edges:



- Why distinguish them?
  - do they affect scheduling differently?

    RAW: read waits for value to be computed

    WAR: write only needs ensure it's not started ahead of the read

- What about output dependences?

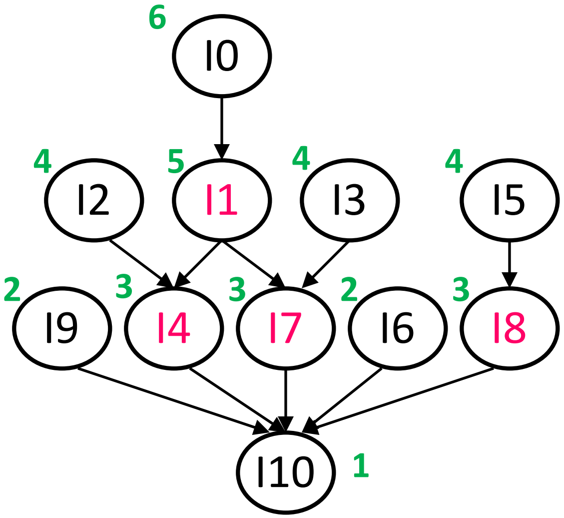
  WAW: earlier write is removed by Dead Code Elimination

  (recall we are scheduling a single basic block, so WAW is unconditional)

Review: List Scheduling

$$priority(x) = \begin{cases} latency(x) & \text{if } x \text{ is a leaf} \\ max(latency(x) + max_{(x,y)\in E}(priority(y)), & \\ \quad max_{(x,y)\in E'}(priority(y))) & otherwise. \end{cases}$$

```
I0:  a = 1
I1:  f = a + x
I2:  b = 7
I3:  c = 9
I4:  g = f + b
I5:  d = 13
I6:  e = 19
I7:  h = f + c
I8:  j = d + y
I9:  z = -1
I10: JMP L1
```



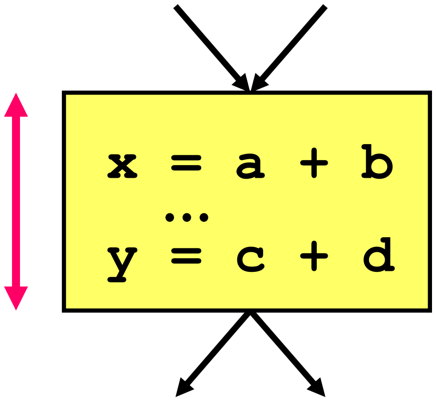
Cycle

		Cycle
I0	I2	0
I1	I3	1
I5	I6	2
I4	I7	3
I8	I9	4
NOP	NOP	5
I10	NOP	6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

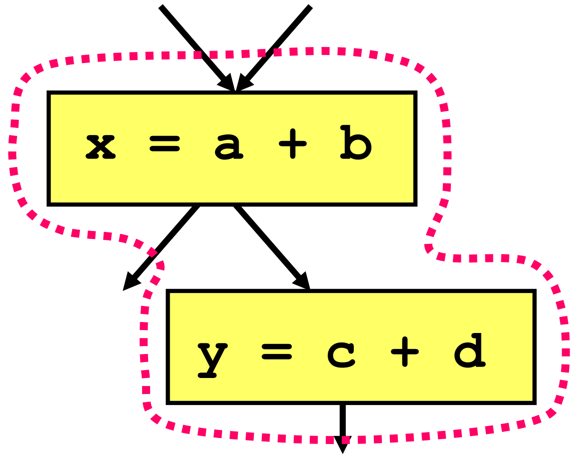
Break ties by lower instruction number

Scheduling Roadmap



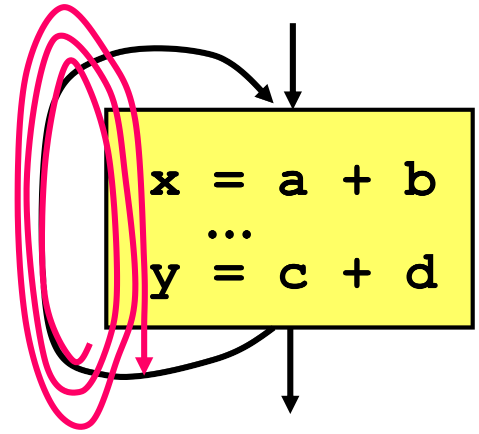
List Scheduling:

- *within* a basic block



Global Scheduling:

- *across* basic blocks



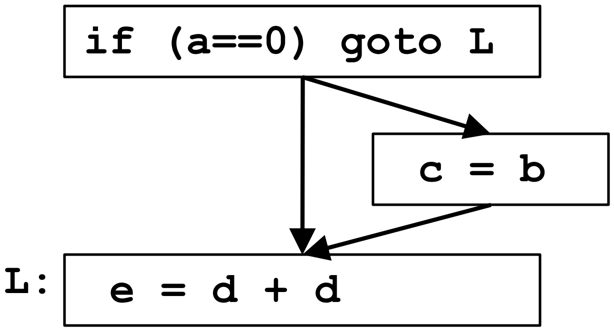
Software Pipelining:

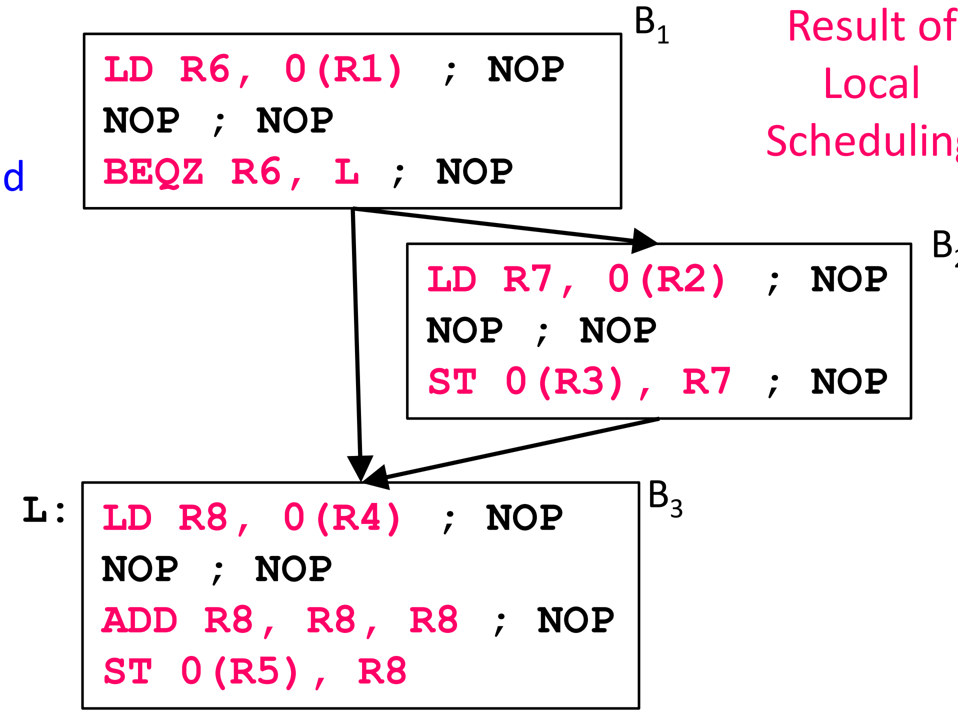
- *across* loop iterations

Introduction to Global Scheduling

Assume each clock can execute
2 operations of any kind

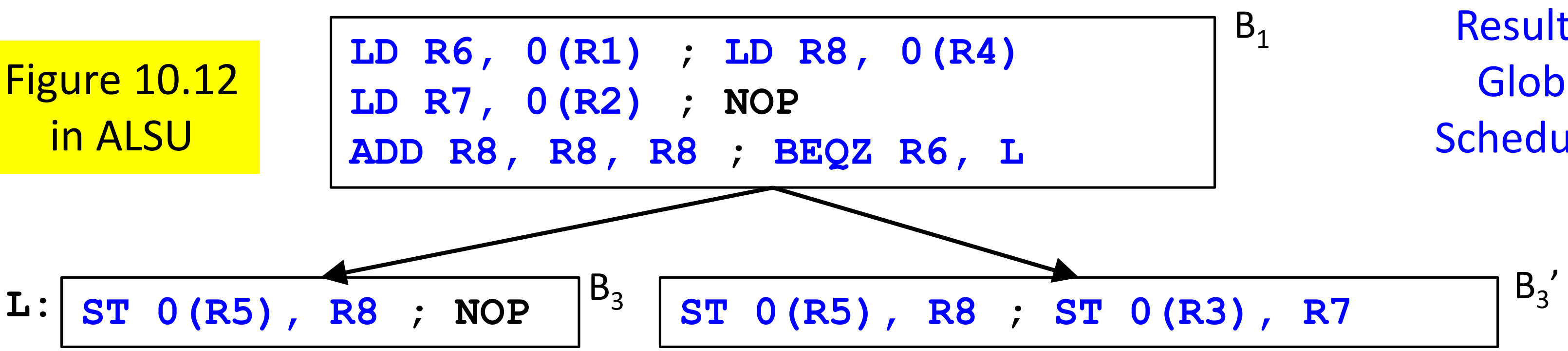
Assume LD takes 2 cycles, fully pipelined





Result of Local Scheduling

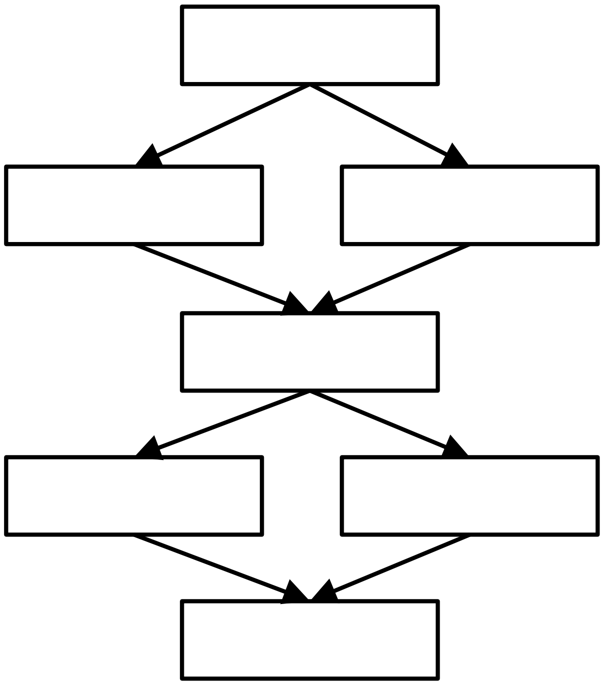
Figure 10.12 in ALSU



Result of Global Scheduling

Carnegie Mellon

Terminology



Control equivalence:

- Two operations o_1 and o_2 are *control equivalent* if o_1 is executed if and only if o_2 is executed.

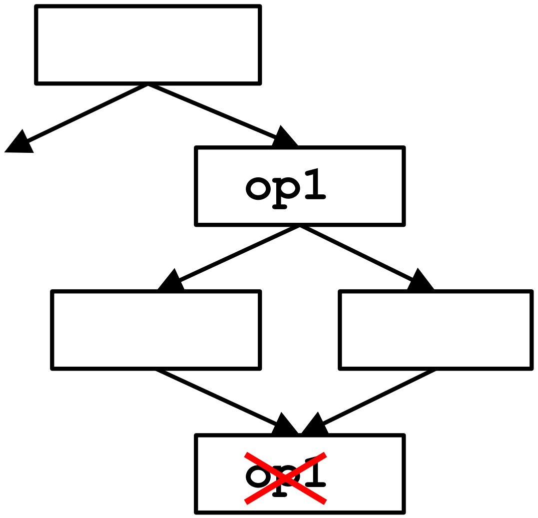
Control dependence:

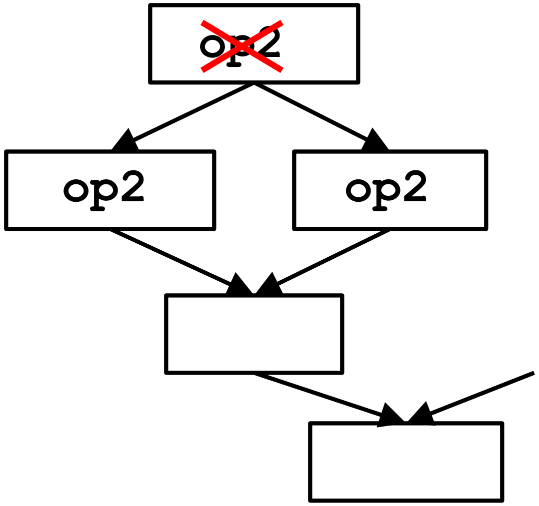
- An op o_2 is *control dependent* on op o_1 if the execution of o_2 depends on the outcome of o_1.

Speculation:

- An operation o is *speculatively* executed if it is executed before all the operations it depends on (control-wise) have been executed.
- Requirements to execute operation speculatively?
  - No side-effects, does not raise an exception
  - Does not violate data dependences

Code Motion





Goal: Shorten execution time probabilistically
(based on estimated frequency of control path)

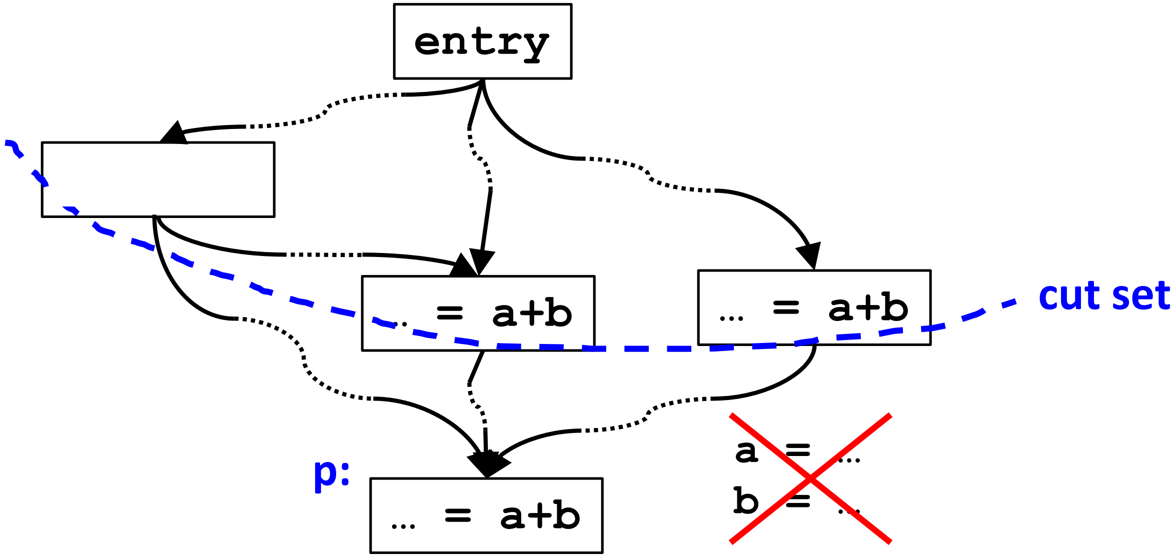
Moving instructions up:

- Move instruction to a cut set (from entry)
- Speculation: even when not anticipated

Moving instructions down:

- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code

Review: Code Motion for Partial Redundancy Elimination



- **Partial redundancy at p: redundant on some but not all paths**
  - Add operations to create a cut set containing a+b
  - Note: Moving operations up can eliminate redundancy
- **Constraint on placement: no wasted operation**
  - a+b is "anticipated" at B if its value computed at B will be used along ALL subsequent paths
  - a, b not redefined, no branches that lead to exit without use
- **Range where a+b is anticipated → Choice**

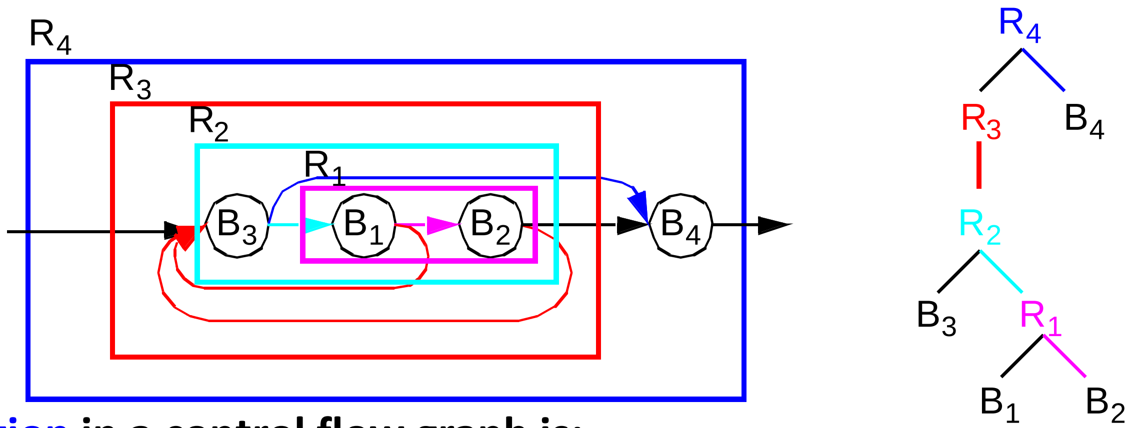
General-Purpose Applications

- **Lots of data dependences**
- **Key performance factor: memory latencies**
- **Move memory fetches up**
  - Speculative memory fetches can be expensive
- **Control-intensive: get execution profile**
  - Static estimation
    - Innermost loops are frequently executed
      - back edges are likely to be taken
    - Edges that branch to exit and exception routines are not likely to be taken
  - Dynamic profiling
    - Instrument code and measure using representative data

A Basic Global Scheduling Algorithm

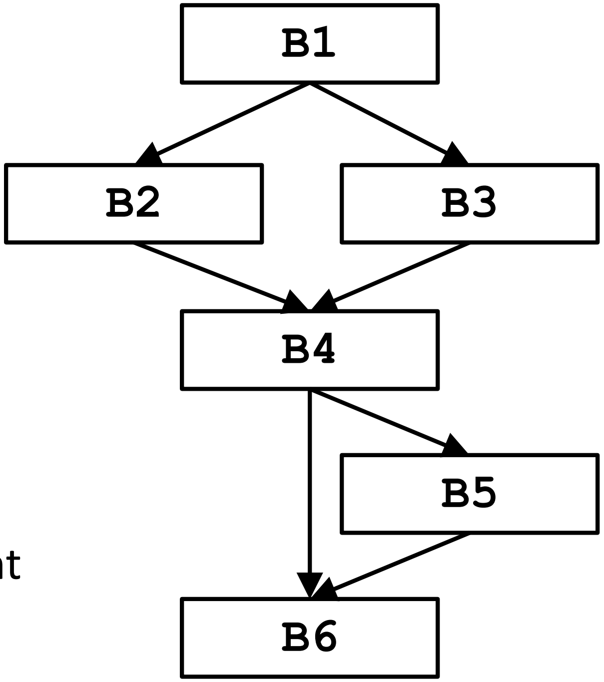
- **Schedule innermost loops first**
- **Only upward code motion, to either:**
  - **a “control-equivalent” block (non-speculative), or**
  - **a control-equivalent block of a dominating predecessor (speculative, 1 branch)**
- **No creation of copies**

Program Representation



- **Recall: A region in a control flow graph is:**
  - a set of basic blocks and all the edges connecting these blocks (expect possibly back edges into the header)
  - such that control from outside the region must enter through the header
- **A procedure is represented as a hierarchy of loop regions**
  - The entire control flow graph is a region
  - Each natural loop (single entry with back edge to it) in the flow graph is a region
  - Natural loops are hierarchically nested
- **Schedule regions from inner to outer**
  - treat inner loop as a black box unit: can schedule around it but not into it
  - ignore all the loop back edges → get an acyclic graph

Useful Definitions



- Blocks B and B' are control equivalent if
  - B is executed if and only if B' is executed
  - E.g., which sets of blocks are control equivalent?

    {B1,B4,B6}, {B1}, {B2},…,{B6}

  Note: Two ops (instructions) are control equivalent iff their basic blocks are control equivalent (could be from same basic block)

- NonSpeculative(B) = all blocks that are control equivalent to B and dominated by B

- Speculative(B) = all blocks B' not control equivalent to B such that
  - B' is a successor of at least one block B'' that is control equivalent to B, and
  - B' is dominated by B''

NonSpeculative(B1)? {B1,B4,B6}

NonSpeculative(B2)? {B2}

Speculative(B1) ? {B2,B3,B5}

Speculative(B2) ? {}

Move up to a control-equivalent block or
a control-equivalent block of a dominating predecessor

Basic Algorithm

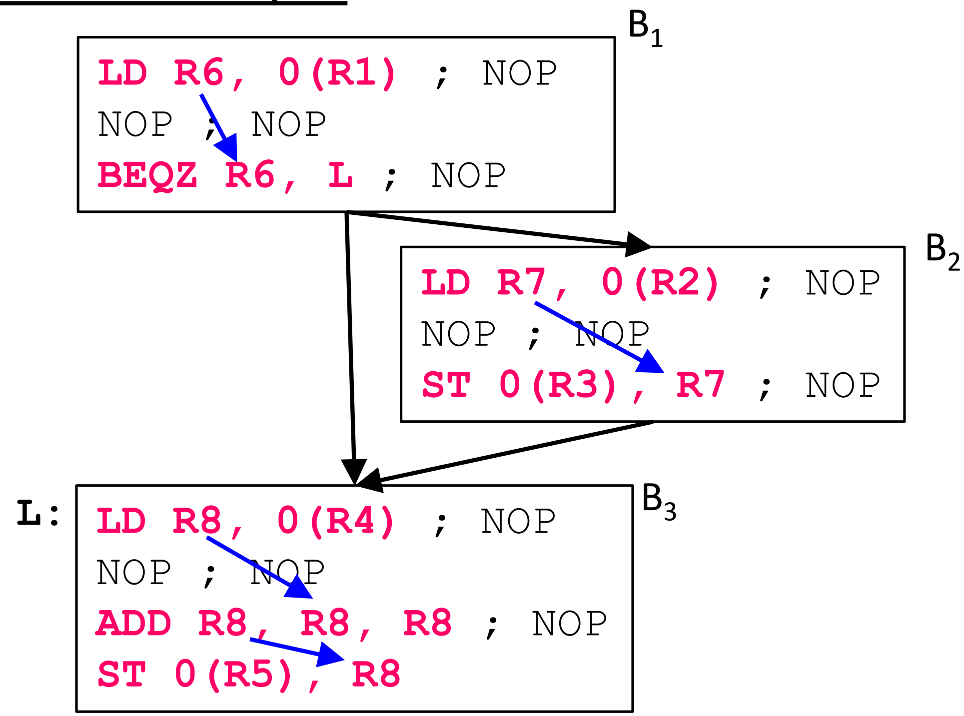
```
Compute data dependences;
For each region R in the hierarchy of loop regions from inner to outer {
  For each basic block B of R in prioritized topological order {
     CandInsts = ready instructions in NonSpeculative(B) ∪ Speculative(B);
     For (t = 1, 2, ... until all instructions from B are scheduled)  {     // schedule time slots in order
       For (n in CandInst in priority order) {                              // may or may not be from B
         if (ok to move n to B && n has no resource conflicts at time t) {
           S(n) = < B, t > ;                    // instruction n is mapped to basic block B and time slot t
           Update resource commitments;
           Update data dependences;
         }
       }
       Update CandInsts;                        // scheduled insts will often make new insts ready
     }
  }
}
```

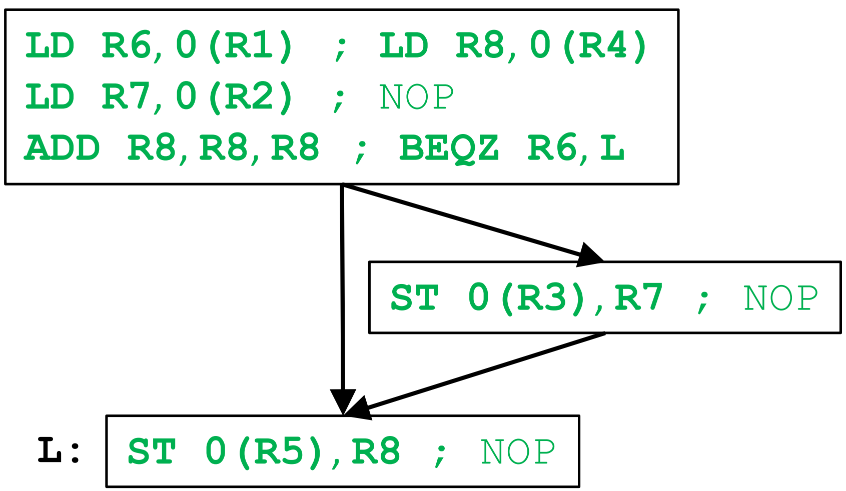
Priority functions: Non-speculative before speculative, and otherwise use same priority as in list scheduling

Ok to move: Don't speculatively move a store instruction, don't move a procedure call, etc

Basic Algorithm Example

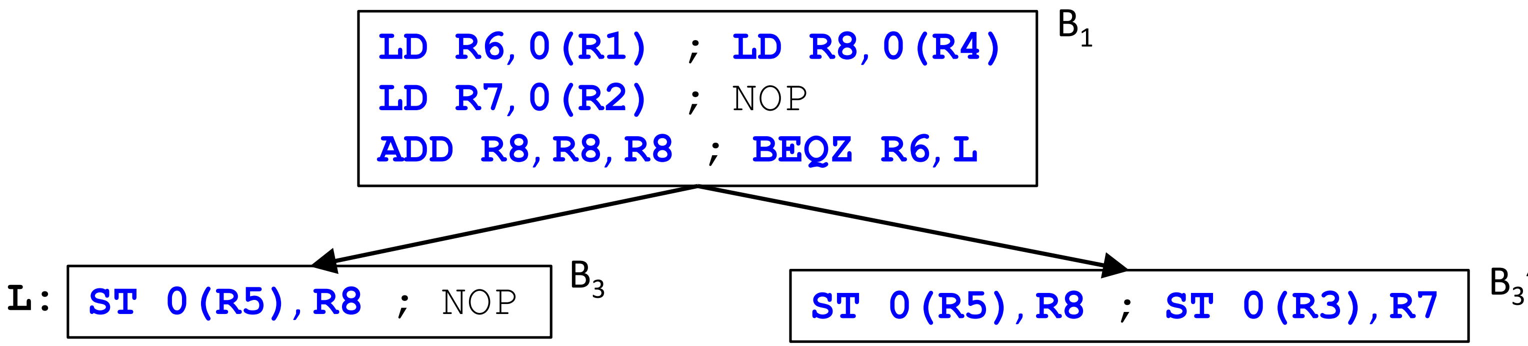
- Each clock: 2 operations of any kind
- LD takes 2 cycles, fully pipelined
- Priority order of blocks: B_1, B_2, B_3
- Data dependences? blue arcs at right
- Control equivalent Blocks? {B_1, B_3}, {B_2}
- NonSpeculative(B_1)? {B_1, B_3}
- Speculative(B_1)? {B_2}
- CandInsts? {`LD R6`; `LD R8`; `LD R7`}

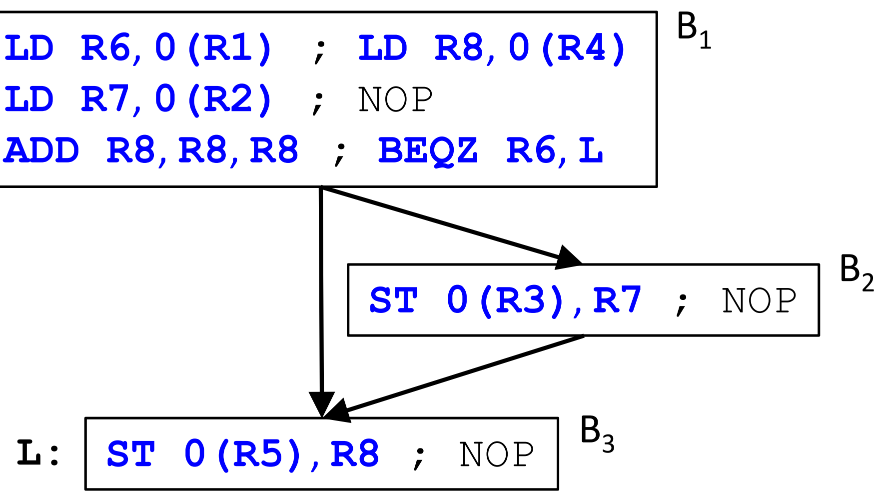




- t=2? {`ADD R8`; `BEQZ R6`; `LD R7`}
- t=3? {`ADD R8`; `BEQZ R6`; `ST R7`}
- NonSpeculative(B_2)? {B_2}
- Speculative(B_2)? {}
- CandInsts for B_2? {`ST R7`}
- CandInsts for B_3? {`ST R8`}

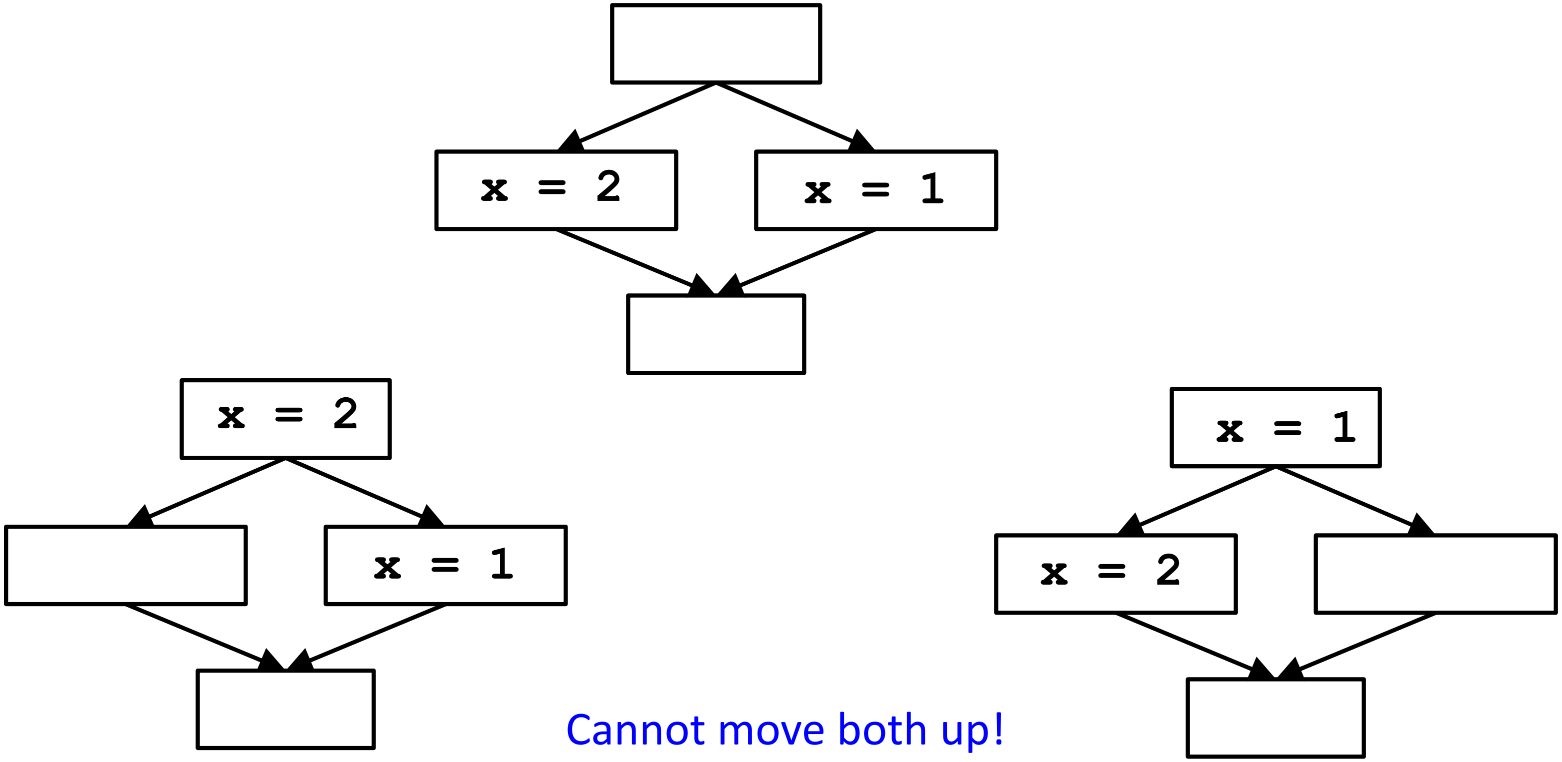
Comparison to Earlier Global Schedule





Basic Algorithm's schedule requires one more cycle when branch not taken

Updating Data Dependence after Code Motion



Cannot move both up!

If a variable is live at a program point, then we cannot move
a speculative definition to the variable above that program point

Extension

- **In region-based scheduling, loop iteration boundary limits code motion: operations from one iteration cannot overlap with those from another**
- **Prepass before scheduling: loop unrolling**
- **Especially important to move operation up loop back edges**

```
for (i = 0; i < N; i++) {
    S(i);
}
```

Original Loop

```
for (i = 0; i+4 < N; i+=4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for ( ; i < N; i++) {
    S(i);
}
```

Unrolled Loop

Today's Class: Global Scheduling

I. Legal code motions
II. Basic Algorithm

Monday's Class

- Software Pipelining & Prefetching
  - ALSU 10.5, ALSU 11.11.4