Reading Assignment

- Read pp 367–386 of Allan et. al.'s paper, "Software Pipelining." (Linked from the class Web page.)

Global Code Scheduling

- Bernstein and Rodeh approach.
- A *prepass scheduler* (does scheduling before register allocation).
- Can move instructions across basic block boundaries.
- Prefers to move instructions that *must* eventually be executed.
- Can move Instructions *speculatively*, possibly executing instructions unnecessarily.

Data & Control Dependencies

When moving instructions across basic block boundaries, we must respect both data dependencies and control dependencies.

Data dependencies specify necessary orderings among instructions that produce a value and instructions that use that value.

Control dependencies determine when (and if) various instructions are executed. Thus an instruction is control dependent on expressions that affect flow of control to that instruction.

Definitions used in Global Scheduling

- Basic Block A *dominates* Basic Block B if and only if A appears on *all* paths to B.
- Basic Block B *postdominates* Basic Block A if and only if B appears on *all* paths from A to an exit point.
- Basic Blocks A and B are *equivalent* if and only if A dominates B and B postdominates A.
- Moving an Instruction from Basic Block B to Basic Block A is *useful* if and only if A and B are equivalent.
- Moving an Instruction from Basic Block B to Basic Block A is *speculative* if B does not postdominate A.

- Moving an Instruction from Basic Block B to Basic Block A requires *duplication* if A does not dominate B.

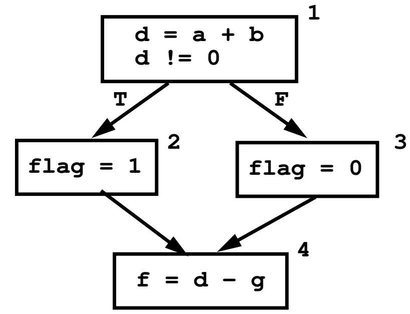
We prefer a move that does not require duplication. (Why?)

The degree of speculation in moving an instruction from one basic block to another can be quantified:

- Moving an Instruction from Basic Block B to Basic Block A is *n-branch* speculative if n conditional branches occur on a path from A to B.

Example

```
d = a + b;
if ( d != 0)
     flag = 1;
else flag = 0;
f = d - g;
```



Blocks 1 and 4 are equivalent.

Moving an Instruction from B2 to B1 (or B3 to B1) is 1-branch speculative.

Moving an Instruction from B4 to B2 (or B4 to B3) requires duplication.

Limits on Code Motion

Assume that pseudo registers are used in generated code (prior to register allocation).

To respect data dependencies:

- A use of a Pseudo Register can't be moved above its definition.
- Memory loads can't be moved ahead of Stores to the same location.
- Stores can't be moved ahead of either loads or stores to the same location.
- A load of a memory location *can* be moved ahead of another load of the same location (such a load may often optimized away by equivalencing the two pseudo registers).

Example (Revisited)

```
block1:
  ld   [a],Pr1
  ld   [b],Pr2
  add  Pr1,Pr2,Pr3  ← Stall
  st   Pr3,[d]
  cmp  Pr3,0
  be   block3
block2:
  mov  1,Pr4
  st   Pr4,[flag]
  b    block4
block3:
  st   0,[flag]
block4:
  ld   [d],Pr5
  ld   [g],Pr6
  sub  Pr5,Pr6,Pr7  ← Stall
  st   Pr7,[f]
```

In B1 and B4, the number of available registers is *irrelevant* in avoiding stalls. There are too few independent instructions in each block.

Global Scheduling Restrictions (in Bernstein/ Rodeh Heuristic)

1. Subprograms are divided into *Regions*. A region is a loop body or the subprogram body without enclosed loops.
2. Regions are scheduled inside-out.
3. Instructions never cross region boundaries.
4. All instructions move "upward" (to earlier positions in the instruction order).
5. The original order of branches is preserved.

Lesser (temporary) restrictions Include:

6. No code duplication.
7. Only 1-branch speculation.
8. No new basic blocks are created or added.

Scheduling Basic Blocks in a CFG

Basic blocks are visited and scheduled in *Topological Order*. Thus all of a block's predecessors are scheduled before it is.

Two levels of scheduling are possible (depending on whether speculative execution is allowed or not):

1. When Basic Block A is scheduled, only Instructions in A and blocks equivalent to A that A dominates are considered.
   (Only "useful" instructions are considered.)

2. Blocks that are immediate successors of those considered in (1) are also considered.
   (This allows 1-branch speculation.)

Candidate Instructions

We first compute the set of basic blocks that may contribute instructions when block A is scheduled. (Either blocks equivalent to A or blocks at most 1-branch speculative.)

An individual Instruction, Inst, in this set of basic blocks may be scheduled in A if:

1. It is located in A.
2. It is in a block equivalent to A and may be moved across block boundaries.
   (Some instructions, like calls, can't be moved.)
3. It is not in a block equivalent to A, but may be scheduled speculatively.
   (Some instructions, like stores, can't be executed speculatively.)

Selecting Instructions to Issue

- A list of "ready to issue" instructions in block A and in bocks equivalent to A (or 1-branch distant from A) is maintained.
- All data dependencies must be satisfied and stalls avoided (if possible).
- N independent instructions are selected, where N is the processor's issue-width.
- But what if more than N instructions are ready to issue?
- Selection is by *Priority*, using two *Scheduling Heuristics*.

Delay Heuristic

This value is computed on a per-basic block basis.

It estimates the worst-case delay (stalls) from an Instruction to the end of the basic block.

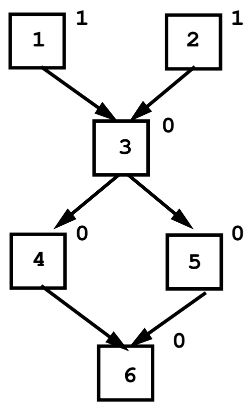
D(I) = 0 if I is a leaf.

Let d(I,J) be the delay if instruction J follows instruction I in the code schedule.

$$D(I) = \max_{J_i \in Succ(I)} (D(J_i)+d(I,J_i))$$

Example of Delay Values

```
block1:
1.  ld   [a],Pr1
2.  ld   [b],Pr2
3.  add  Pr1,Pr2,Pr3
4.  st   Pr3,[d]
5.  cmp  Pr3,0
6.  be   block3
```



(Assume only loads can stall.)

Critical Path Heuristic

This value is also computed on a per-basic block basis.

It estimates how long it will take to execute Instruction I, and all I's successors, assuming unlimited parallelism.

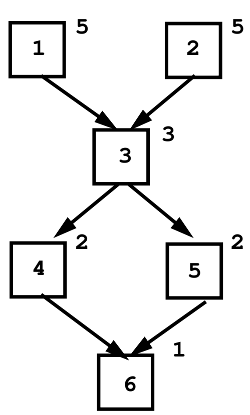
E(I) = Execution time for instruction I (normally 1 for pipelined machines)

CP(I) = E(I) if I is a leaf.

$$CP(I) = E(I) + \max_{J_i \in Succ(I)} (CP(J_i)+d(I,J_i))$$

Example of Critical Path Values

```
block1:
1.  ld   [a],Pr1
2.  ld   [b],Pr2
3.  add  Pr1,Pr2,Pr3
4.  st   Pr3,[d]
5.  cmp  Pr3,0
6.  be   block3
```



Selecting Instructions to Issue

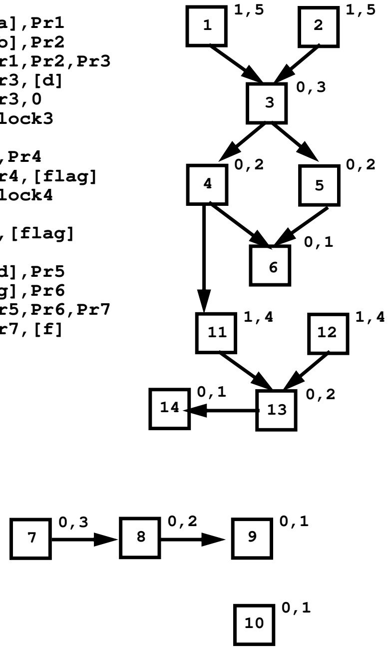
From the Ready Set (instructions with all dependencies satisfied, and which will not stall) use the following priority rules:

1. Instructions in block A and blocks equivalent to A have priority over other (speculative) blocks.
2. Instructions with the highest D values have priority.
3. Instructions with the highest CP values have priority.

These rules imply that we schedule useful instructions before speculative ones, instructions on paths with potentially many stalls over those with fewer stalls, and instructions on critical paths over those on non-critical paths.

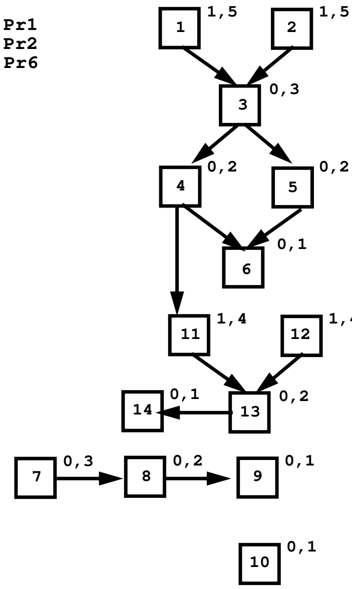
Example

```
block1:
1.    ld    [a],Pr1
2.    ld    [b],Pr2
3.    add   Pr1,Pr2,Pr3
4.    st    Pr3,[d]
5.    cmp   Pr3,0
6.    be    block3
block2:
7.    mov   1,Pr4
8.    st    Pr4,[flag]
9.    b     block4
block3:
10.   st    0,[flag]
block4:
11.   ld    [d],Pr5
12.   ld    [g],Pr6
13.   sub   Pr5,Pr6,Pr7
14.   st    Pr7,[f]
```



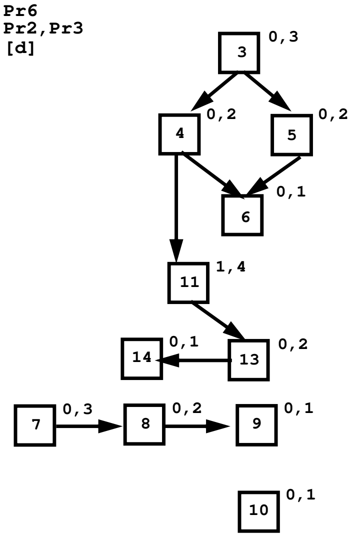
We'll schedule without speculation; highest D values first, then highest CP values.

```
block1:
1.    ld    [a],Pr1
2.    ld    [b],Pr2
12.   ld    [g],Pr6
```



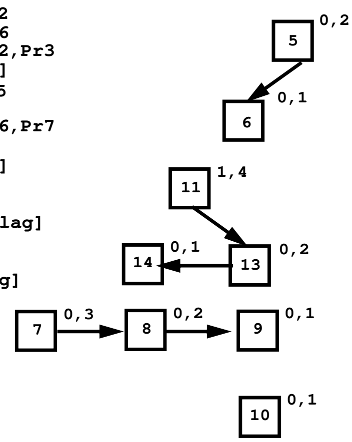
Next, come Instructions 3 and 4.

```
block1:
1.    ld    [a],Pr1
2.    ld    [b],Pr2
12.   ld    [g],Pr6
3.    add   Pr1,Pr2,Pr3
4.    st    Pr3,[d]
```



Now 11 can issue (D=1), followed by 5, 13, 6 and 14. Block B4 is now empty, so B3 and B4 are scheduled.

```
block1:
1.    ld    [a],Pr1
2.    ld    [b],Pr2
12.   ld    [g],Pr6
3.    add   Pr1,Pr2,Pr3
4.    st    Pr3,[d]
11.   ld    [d],Pr5
5.    cmp   Pr3,0
13.   sub   Pr5,Pr6,Pr7
6.    be    block3
14.   st    Pr7,[f]
block2:
7.    mov   1,Pr4
8.    st    Pr4,[flag]
9.    b     block4
block3:
10.   st    0,[flag]
block4:
```



There are no stalls. In fact, if we equivalence `Pr3` and `Pr5`, Instruction 11 can be removed.

Software Pipelining

Often loop bodies are too small to allow effective code scheduling. But loop bodies, being "hot spots," are exactly where scheduling is most important.

Consider

```
void f (int a[],int last) {
  for (p=&a[0];p!=&a[last];p++)
     (*p)++;
}
```

The body of the loop might be:

```
L: ld   [%g3],%g2
   nop
   add  %g2,1,%g2
   st   %g2,[%g3]
   add  %g3,4,%g3
   cmp  %g3,%g4
   bne  L
   nop
```


Scheduling this loop body in isolation is ineffective—each instruction depends upon its immediate predecessor.

So we have a loop body that takes 8 cycles to execute 6 "core" instructions.

We could unroll the loop body, but for how many iterations? What if the loop ends in the "middle" of an expanded loop body? Will extra registers be a problem?

In this case *software pipelining* offers a nice solution. We expand the loop body *symbolically*, intermixing instructions from several iterations. Instructions can overlap, increasing parallelism and forming a "tighter" loop body:

```
   ld   [%g3],%g2
   nop
   add  %g2,1,%g2
L: st   %g2,[%g3]
   add  %g3,4,%g3
   ld   [%g3],%g2
   cmp  %g3,%g4
   bne  L
   add  %g2,1,%g2
```

Now the loop body is ideal—exactly 6 instructions. Also, no extra registers are needed!

But, we do "overshoot" the end of the loop a bit, loading one element past the exit point. (How serious is this?)