Finding Additional Independent Instructions for Parallel Issue

We can extend the capabilities of processors:

- **Out of order execution allows a processor to "search ahead" for independent instructions to launch.**
- *But*, **since basic blocks are often quite small, the processor may need to accurately predict branches, issuing instructions before the execution path is fully resolved.**
- *But*, **since branch predictions may be wrong, it will be necessary to "undo" instructions executed speculatively.**

Reading Assignment

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

Compiler Support for Extended Scheduling

- **Trace Scheduling**

  Gather sequences of basic blocks together and schedule them as a unit.

- **Global Scheduling**

  Analyze the control flow graph and move instructions across basic block boundaries to improve scheduling.

- **Software Pipelining**

  Select instructions from several loop iterations and schedule them together.

Trace Scheduling

Reference:

J. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction,” IEEE Transactions on Computers, July 1981.

Idea:

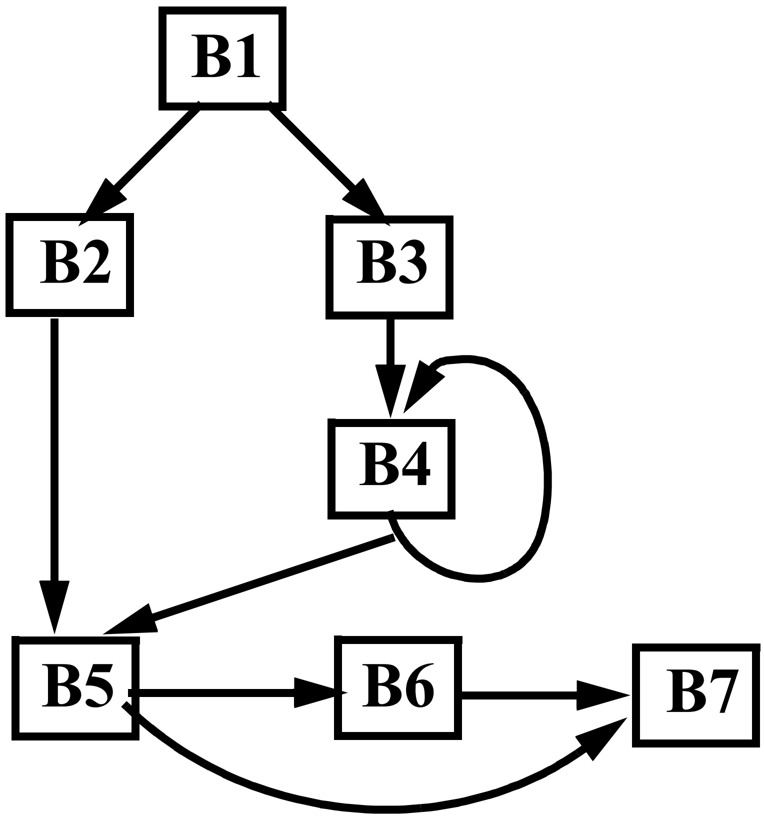
Since basic blocks are often too small to allow effective code scheduling, we will *profile* **a program’s execution and identify the most frequently executed paths in a program.**

Sequences of contiguous basic blocks on frequently executed paths will be gathered together into *traces*.

Trace

- **A sequence of basic blocks (excluding loops) executed together can form a trace.**
- **A trace will be scheduled as a unit, allowing a larger span of instructions for scheduling.**
- **A loop can be unrolled or scheduled individually.**
- *Compensation code* **may need to be added when a branch into, or out of, a trace occurs.**

Example



Assume profiling shows that
B1→B3→B4$^+$→B5→B7
is the most common execution path.

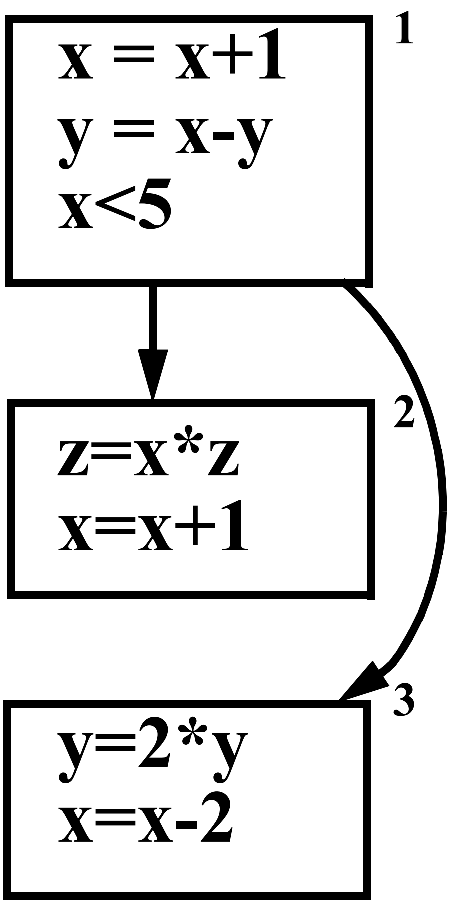
The traces extracted from this path are
B1→B3, B4, and B5→B7.

Compensation Code

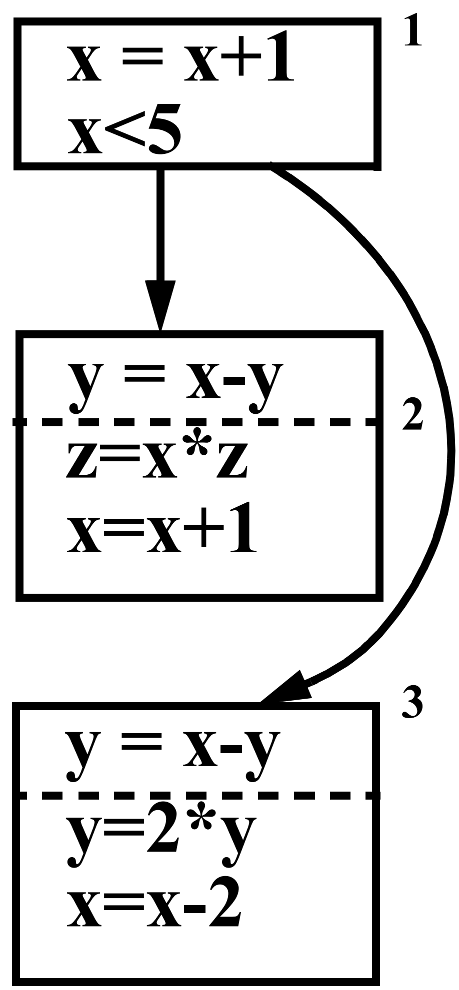
When we move instructions across basic block boundaries within a trace, we may need to add extra instructions that preserve program semantics on paths that enter or leave the trace.

Example

In the previous example, basic block B1 had B2 and B3 as successors, and B1→B3 formed a trace.



Before Scheduling



After Scheduling

Advantages & Disadvantages

- Trace scheduling allows scheduling to span multiple basic blocks. This can significantly increase the effectiveness of scheduling, especially in the context of superscalar processors (which need ILP to be effective).
- Trace Scheduling can also increase code size (because of compensation code).
  It is also sensitive to the accuracy of trace estimates.

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.**

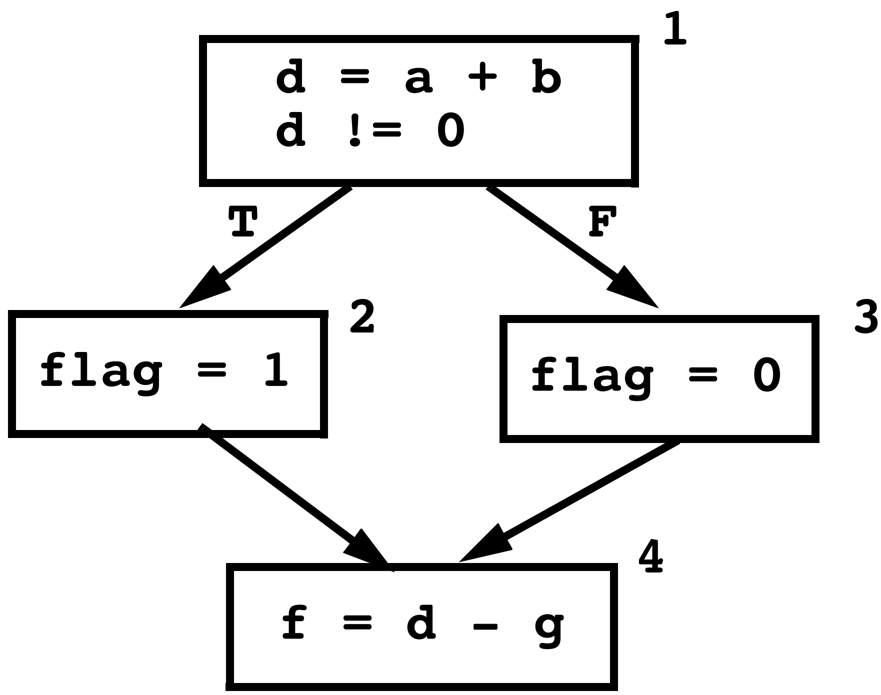
We prefer a move that does not require duplication. (Why?)

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 be 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.

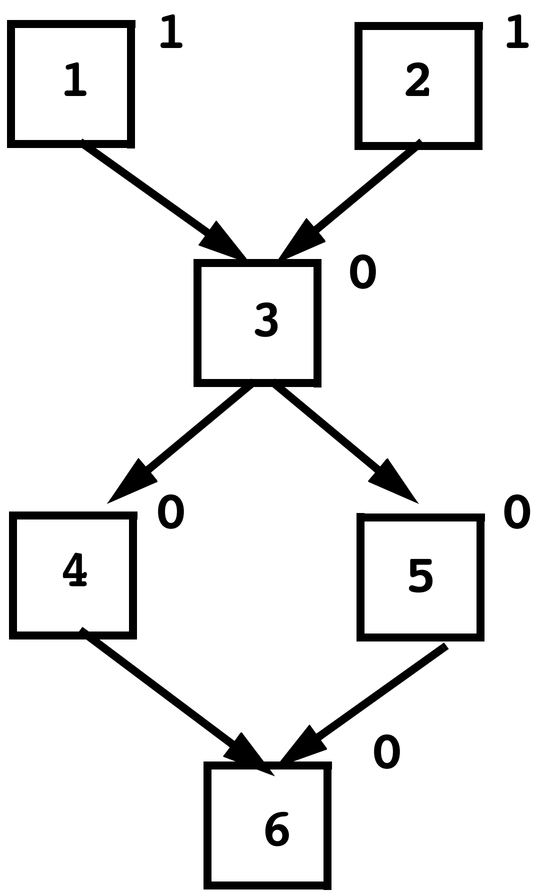
D(I) = 0 if I is a leaf.

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

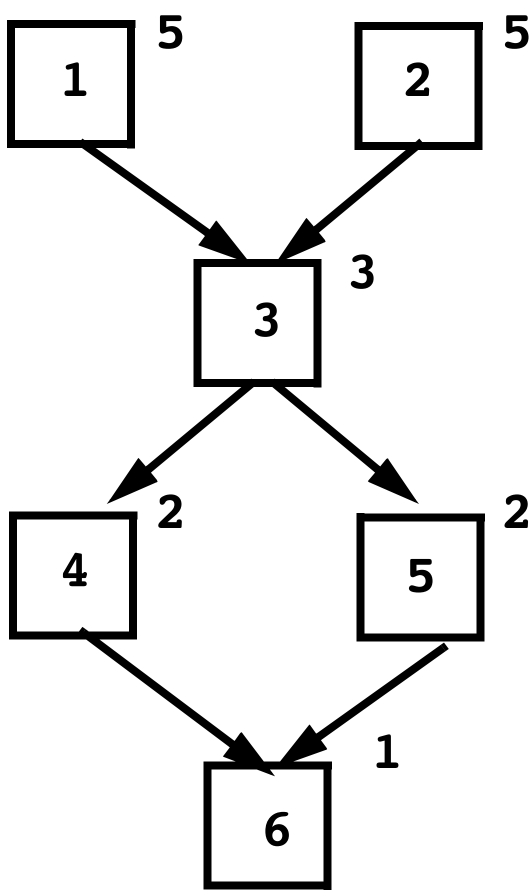
(normally 1 for pipelined machines)

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

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

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

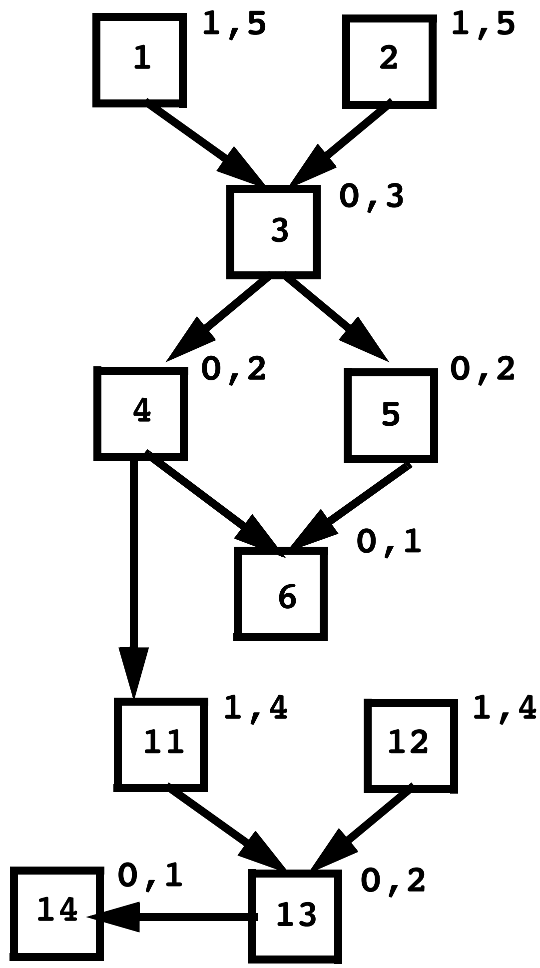
From the Ready Set (instructions with all dependencies satisfied, and which will not stall) use the following priority rules:

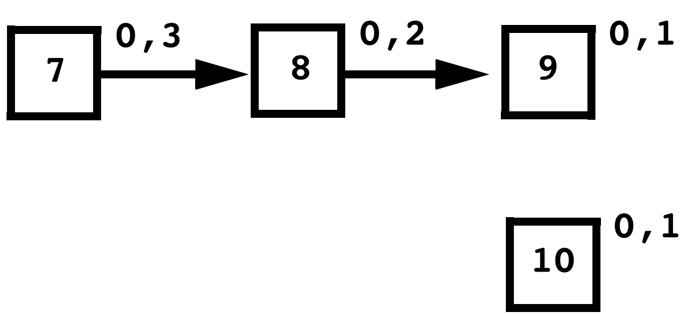
1. **Instructions in block A and blocks equivalent to A have priority over other (speculative) blocks.**
2. **Instructions with the highest D values have priority.**
3. **Instructions with the highest CP values have priority.**

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

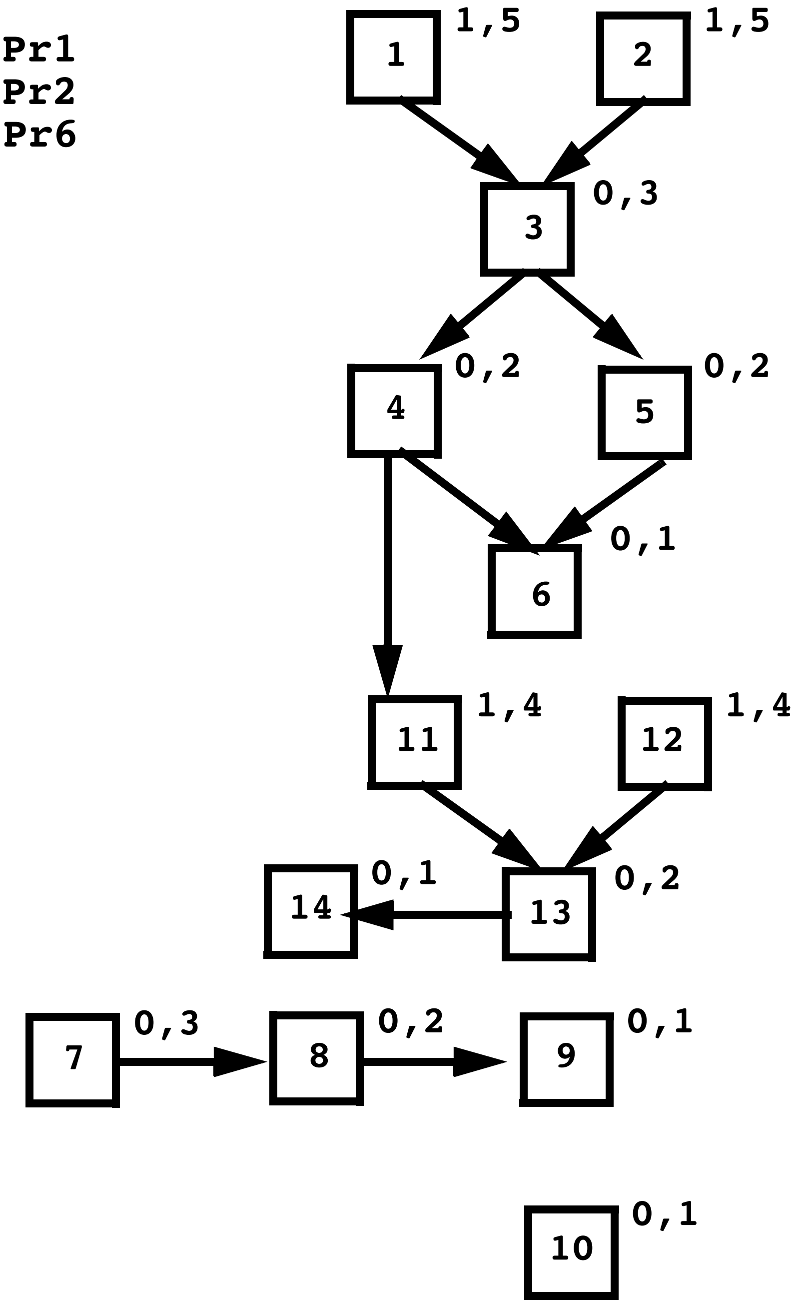
```
block1:
1.   ld   [a],Pr1
2.   ld   [b],Pr2
3.   add  Pr1,Pr2,Pr3
4.   st   Pr3,[d]
5.   cmp  Pr3,0
6.   be   block3
block2:
7.   mov  1,Pr4
8.   st   Pr4,[flag]
9.   b    block4
block3:
10.  st   0,[flag]
block4:
11.  ld   [d],Pr5
12.  ld   [g],Pr6
13.  sub  Pr5,Pr6,Pr7
14.  st   Pr7,[f]
```





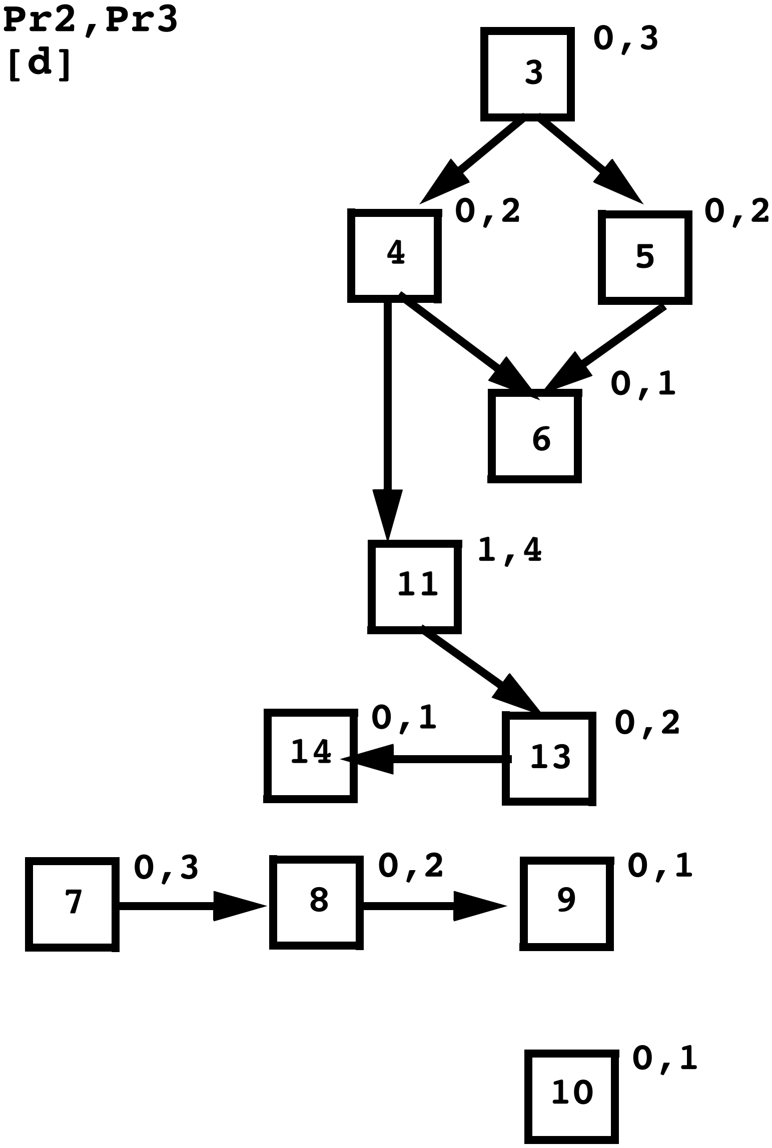
We'll schedule without speculation; highest D values first, then highest CP values.

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



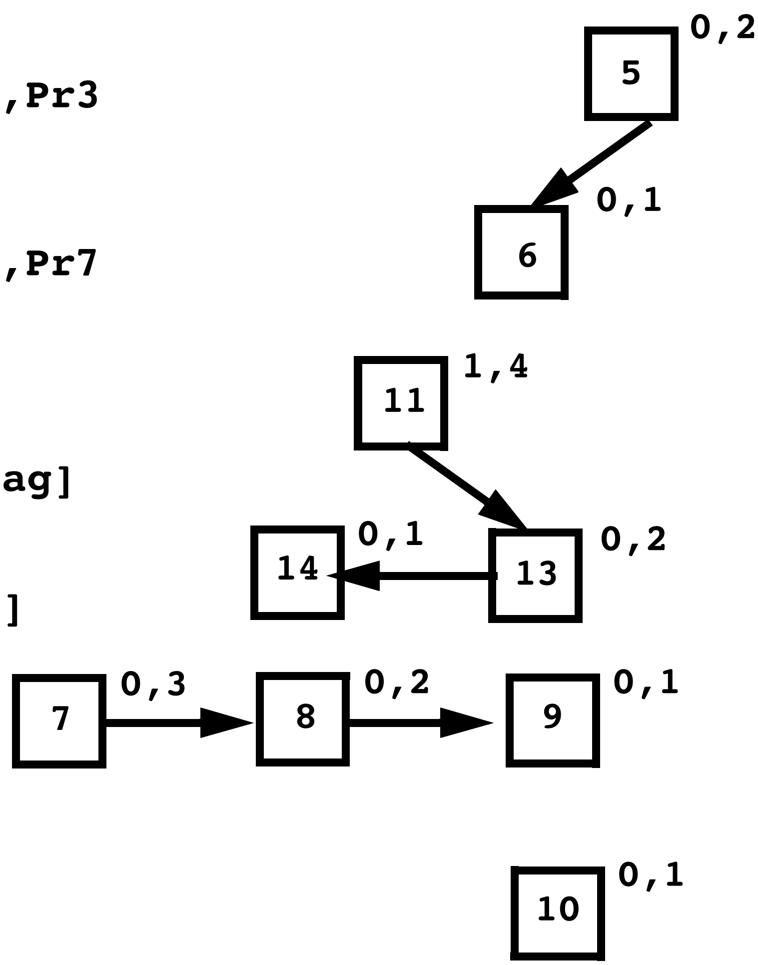
Next, come Instructions 3 and 4.

```
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 B2 and B3 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.

Hardware Support for Global Code Motion

We want to be aggressive in scheduling loads, which incur high latencies when a cache miss occurs.

In many cases, control and data dependencies may force us to restrict how far we may move a critical load.

Consider

```
p = Lookup(Id);
   ...
if (p != null)
   print(p.a);
```

It may well be that the object returned by `Lookup` is not in the L1 cache. Thus we'd like to schedule the load generated by `p.a` as soon as possible; ideally right after the lookup.

But moving the load above the `p != null` check is clearly unsafe.

A number of modern machine architectures, including Intel's Itanium, have proposed a *speculative load* to allow freer code motion when scheduling.

A speculative load,

```
ld.s  [adr],%reg
```

acts like an ordinary load as long as the load does not force an interrupt. If it does, the interrupt is suppressed and a special `NaT` (not a thing) bit is set in the register (a hidden 65th bit). A `NaT` bit can be propagated through instructions before being tested.

In some cases (like our table lookup example), a register containing a `NaT` bit may simply not be used because control doesn't reach its intended uses.

However a `NaT` bit need not indicate an outright error. A load may force a TLB (translation lookaside buffer) fault or a

page fault. These interrupts are probably too costly to do speculatively, but if we decide the loaded value is really needed, we will want to allow them.

A special check instruction, of the form,

```
chk.s  %reg,adr
```

checks whether `%reg` has its `NaT` bit set. If it does, control passes to `adr`, where user-supplied "fixup" code is placed. This code can redo the load non-speculatively, allowing necessary interrupts to occur.

Hardware Support for Data Speculation

In addition to supporting control speculation (moving instructions above conditional branches), it is useful to have hardware support for data speculation.

In data speculation, we may move a load above a store if we believe the chance of the load and store conflicting is slim.

Consider a variant of our earlier lookup example,

```
p = Lookup(Id);
  ...
q.a = init();
print(p.a);
```

We'd like to move the load implied by `p.a` above the assignment to `q.a`. This allows `p` to miss in the L1 cache, using the execution of `init()` to cover the miss latency.

***But*, we need to be sure that `q` and `p` don't reference the same object and that `init()` doesn't indirectly change `p.a`. Both possibilities may be remote, but proving non-interference may be difficult.**

The Intel Itanium provides a special "advanced load" that supports this sort of load motion.

The instruction

```
ld.a  [adr],%reg
```

loads the contents of memory location `adr` into `%reg`. It also stores `adr` into

special *ALAT* (Advanced Load Address Table) hardware.

When a store to address `x` occurs, an ALAT entry corresponding to address `x` is removed (if one exists).

When we wish to use the contents of `%reg`, we execute a

```
ld.c  [adr],%reg
```

instruction (a *checked* load).

If an ALAT entry for `adr` is present, this instruction does nothing; `%reg` contains the correct value. If there is no corresponding ALAT entry, the `ld.c` simply acts like an ordinary load.

(Two versions of `ld.c` exist; one preserves an ALAT entry while the other purges it).

And yes, a speculative load (`ld.s`) and an advanced load (`ld.a`) may be combined to form a speculative advanced load (`ld.sa`).

Speculative Multi-threaded Processors

The problem of moving a load above a store that may conflict with it also appears in multi-threaded processors.

How do we know that two threads don't interfere with one another by writing into locations both use?

Proofs of non-interference can be difficult or impossible. Rather than severely restrict what independent threads can do, researchers have proposed *speculative* multi-threaded processors.

In such processors, one thread is primary, while all other threads are secondary and speculative. Using hardware tables to remember locations read and written, a secondary thread

can commit (make its updates permanent) only if it hasn't read locations the primary thread later wrote and hasn't written locations the primary thread read or wrote. Access conflicts are automatically detected, and secondary threads are automatically restarted as necessary to preserve the illusion of serial memory accesses.