False Dependencies & Loop Unrolling

A limiting factor in how "tightly" we can software pipeline a loop is reuse of registers and the false dependencies reuse induces.

Consider the following simple function that copies array elements:

```
void f (int a[],int b[], int lim) {
  for (i=0;i<lim;i++)
     a[i]=b[i];
}
```

The loop that is generated takes 3 cycles:

```
cycle        instruction
1.     L:     ld    [%g3+%o1], %g2
1.            addcc %o2, -1, %o2
3.            st    %g2, [%g3+%o0]
3.            bne   L
3.            add   %g3, 4, %g3
```

We'd like to tighten the iteration interval to 2 or less. One cycle is unlikely, since doing a load and a store in the same cycle is problematic (due to a possible dependence through memory).

If we try to use modulo scheduling, we can't put a second copy of the load in cycle 2 because it would overwrite the contents of the first load. A load in cycle 3 will clash with the store.

The solution is to unroll the loop into two copies, using different registers to hold the contents of the load and the current offset into the arrays.

The use of a "count down" register to test for loop termination is helpful,

since it allows an easy exit from the middle of the loop.

With the renaming of the registers used in the two expanded iterations, scheduling to “tighten” the loop is effective.

After expansion we have:

```
cycle        instruction
1.      L:    ld    [%g3+%o1], %g2
1.            addcc %o2, -1, %o2
3.            st    %g2, [%g3+%o0]
3.            beq   L2
3.            add   %g3, 4, %g4
4.            ld    [%g4+%o1], %g5
4.            addcc %o2, -1, %o2
6.            st    %g5, [%g4+%o0]
6.            bne   L
6.            add   %g4, 4, %g3
        L2:
```

We still have 3 cycles per iteration, because we haven’t scheduled yet.

Now we can move the increment of **%g3** (into **%g4**) above other uses of **%g3**. Moreover, we can move the load into **%g5** *above* the store from **%g2** (*if the load and store are independent*):

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
1.           add   %g3, 4, %g4
2.           ld    [%g4+%o1], %g5
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           addcc %o2, -1, %o2
4.           st    %g5, [%g4+%o0]
4.           bne   L
4.           add   %g4, 4, %g3
      L2:
```

We can normally test whether **%g4+%o1** and **%g3+%o0** can be equal at compile-time, by looking at the actual array parameters. (Can **&a[0] == &b[1]**?)