Instruction Level Parallelism

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called ***instruction-level parallelism (ILP)*** since the instructions can be evaluated in parallel.

The amount of parallelism available within a basic block (a straight-line code sequence with no branches in and out except for entry and exit) is quite small. The average dynamic branch frequency in integer programs was measured to be about 15%, meaning that about 7 instructions execute between a pair of branches.

Since the instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than 7.

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called ***loop-level parallelism***.

Example 1

```
for (i=1; i<=1000; i= i+1)
  x[i] = x[i] + y[i];
```

This is a parallel loop. Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little opportunity for overlap.

Example 2

```
for (i=1; i<=100; i= i+1){
  a[i] = a[i] + b[i];        //s1
  b[i+1] = c[i] + d[i];      //s2
}
```

Is this loop parallel? If not how to make it parallel?

Statement **s1** uses the value assigned in the previous iteration by statement **s2**, so there is a loop-carried dependency between **s1** and **s2**. Despite this dependency, this loop can be made parallel because the dependency is not circular:
- neither statement depends on itself;
- while **s1** depends on **s2**, **s2** does not depend on **s1**.

A loop is parallel unless there is a cycle in the dependecies, since the absence of a cycle means that the dependencies give a partial ordering on the statements.

To expose the parallelism the loop must be transformed to conform to the partial order. Two observations are critical to this transformation:

- There is no dependency from **s1** to **s2**. Then, interchanging the two statements will not affect the execution of **s2**.
- On the first iteration of the loop, statement **s1** depends on the value of **b[1]** computed prior to initiating the loop.

This allows us to replace the loop above with the following code sequence, which makes possible overlapping of the iterations of the loop:

```
a[1] = a[1] + b[1];
for (i=1; i<=99; i= i+1){
  b[i+1] = c[i] + d[i];
  a[i+1] = a[i+1] + b[i+1];
}
b[101] = c[100] + d[100];
```

Example 3

```
for (i=1; i<=100; i= i+1){
  a[i+1] = a[i] + c[i];       //S1
  b[i+1] = b[i] + a[i+1];     //S2
}
```

This loop is not parallel because it has cycles in the dependencies, namely the statements **S1** and **S2** depend on themselves!

There are a number of techniques for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop.