ECE 511: Computer Architecture **Nov. 1, 2004**

Lecture 18: Limits of Superscalar Processors

Lecturer: Matt Frank *Scribe: Mark Dykstra*

There are two problems with superscalar processors: the bypasses they require add complexity and they don't capitalize on all types of instruction level parallelism (ILP).

1) Bypassing:

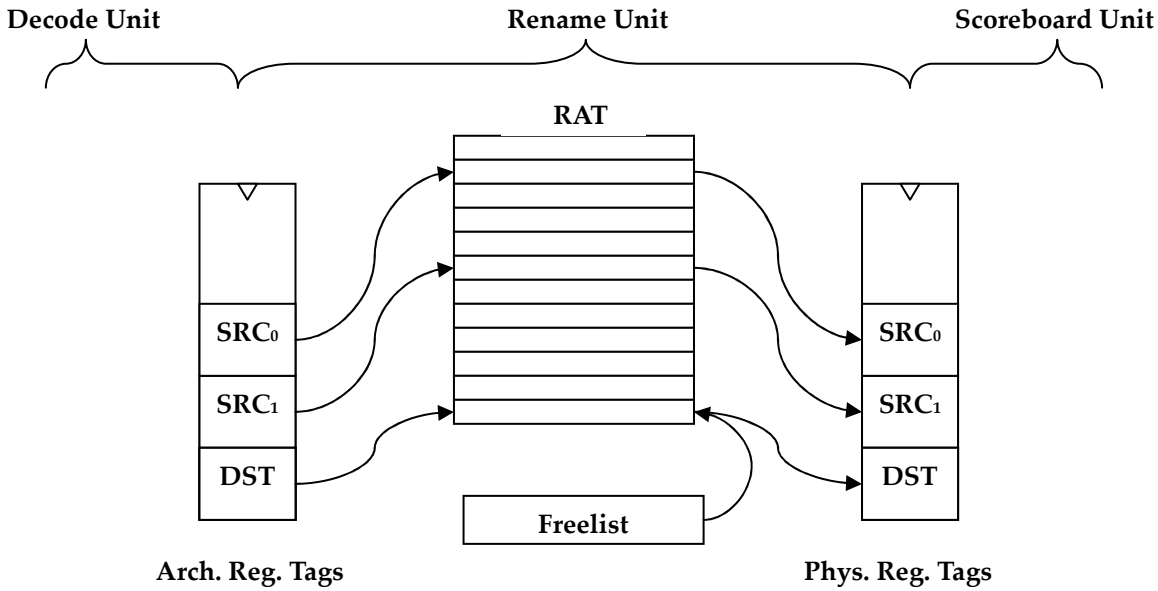
Our goal so far has been to increase the throughput of our machine as much as possible (the number of instructions retiring per second). If the instructions were all independent (the output of one instruction was never the source of another), then it would be easy to simply increase the width of our machine. That's far from reality though. Remember our equation that says throughput, X, is dependent on both the number, N, of things you can do at once and the amount of time, R, spent on each.

$$X = \frac{N}{R}$$

If we want to increase X, we can increase the number of things the machine can do at once (N). But, as you increase N, R tends to grow too; being able to do more things requires that you use more chip area which in turn means that signals take longer to move around the chip, thereby increasing R.

Rename Unit Example:

Inside the rename unit, we have something that looks like the diagram at the top of the next page. As an instruction arrives from the decoder, its two source registers are mapped to physical registers by the RAT. Meanwhile, a new physical register is pulled from the top of the freelist and becomes the new mapping for the destination architectural register. This physical register updates the RAT and also is written to the output latch.



So how can we expand this to handle renaming two instructions at once? If everything was completely independent, you could just add a second set of latches and double the number of ports into the RAT. Instead, we have to handle the case where the first instruction is a source for the second.

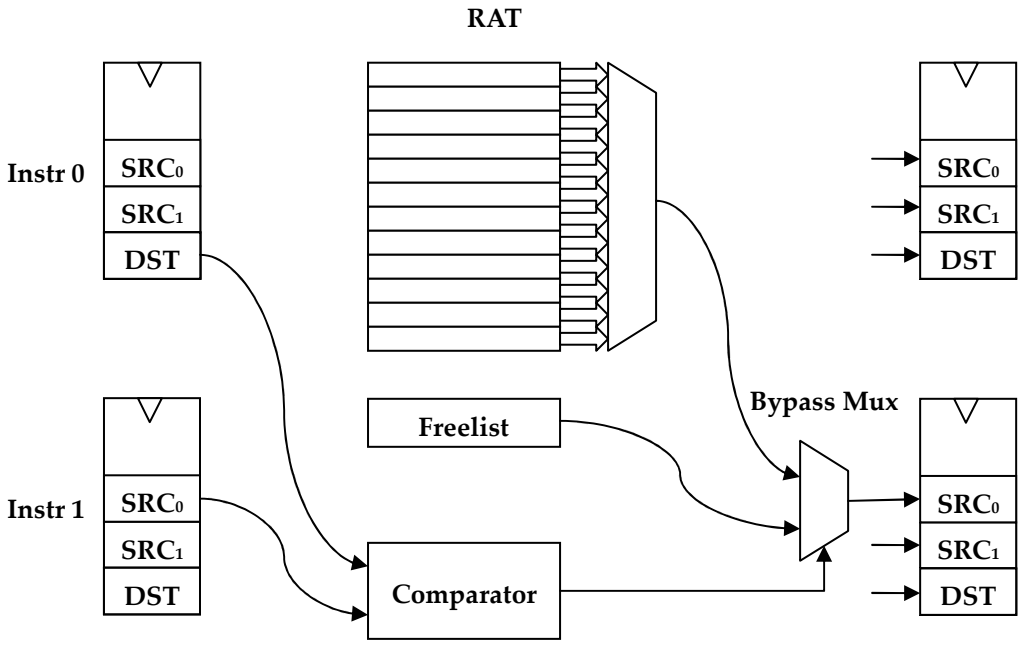
Anti-dependencies: Don't matter because we read the correct value out of the RAT before writing a new one.

Output-dependencies: Do matter but we're not worrying about them. Can be handled by making sure you ultimately write the latter destination register and by only pulling one register off the freelist when the destinations are the same.

True-dependencies: Do matter and are the common problem we have to handle. You can use a bypassing register file to take care of this problem. Ultimately, we just need the machine state to change in the same way it would have if we had renamed these two instructions in two separate cycles.

What does it actually cost to implement this change? Initially the logic involved would effectively be a 32:1 mux to select between the 32 RAT entries depending on the 5 bits that identify each source register (a literal mux or a 5:32 decoder + tri-states). This will require O(log(N)) = O(5) levels of logic to implement.

To rename two instructions per cycle, we need another mux to select between the RAT output and a register from the freelist. To determine which line to select from the mux we will need additional logic, a comparator to check if there is a dependency (i.e., the destination of instruction 0 is a source of instruction 1). If the comparator operates quickly compared to accessing the RAT, it may not add any latency to the system. The second mux will however.



So, is it worth this cost? Often it is for the renamer since similar things are going on elsewhere in the pipeline in a much more complex fashion; the renamer is not really on the critical path. For example, the RAT has only 32 entries and outputs 6 or 7 bit data, whereas further into the pipeline you have to deal with the physical register file with many more entries and 32-bit outputs. Furthermore, the renamer is an in-order part of the pipeline so the dependence analyses are greatly simplified compared to the out-of-order portion.

What about renaming four instructions then? It quickly gets ugly. A 4-wide renamer requires a 4:1 mux (making the latency O(7)) and much more complex circuitry to analyze all the possible dependencies between the 4 instructions.

We attempted to double our throughput (X) by doubling the amount of renaming (N), but in the process our latency (R) went up so we didn't achieve a full 2x benefit. Asymptotically it tends to be the case that when you increase N by a factor of 4, R goes up by a factor of 2. That is,

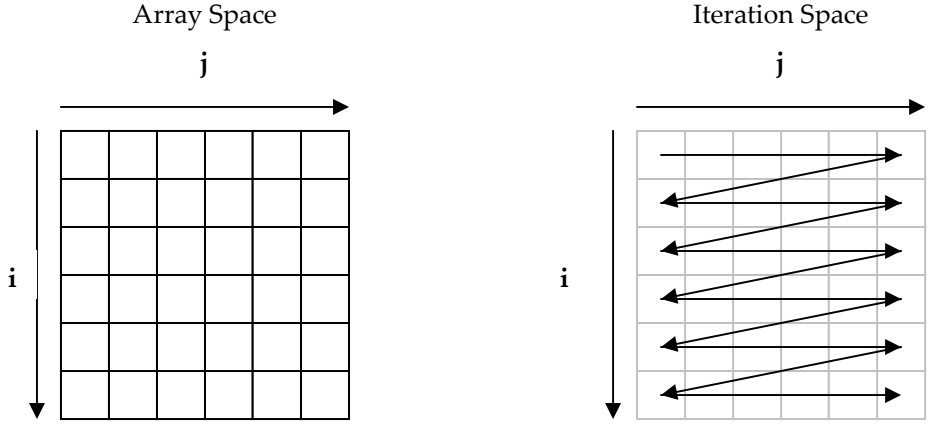
$$X \approx \frac{N}{\sqrt{N}} = \sqrt{N}$$

2) ILP = Inner Loop Parallelism:

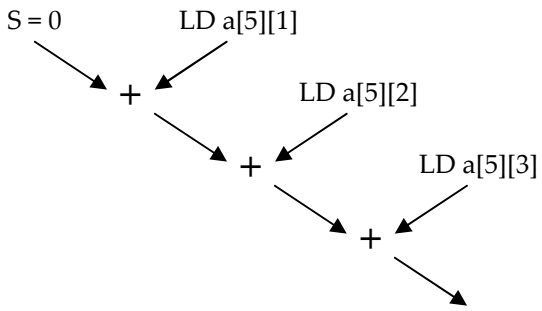
Example 1) :

Here's some pseudo-Fortran to sum the rows of a 2D array:

```
do i = 1 to n
    s = 0
    do j = 1 to n
        s = s + a[i][j]
    t[i] = s
```



How does the array get passed through the pipeline? Each element is dependent on the previous one because the sum of all previous entries in the row is one of the source operands it needs. The dependence chains look something like this:



So at best we can achieve an ILP of 2 (useful instructions). Compilers often will not improve this code either. GCC doesn't and possibly with good reason. If you don't know the number of registers available during optimization, it's possible to make this code even slower by attempting to optimize.

Since it's up to us to improve this, what if we use multiple 's' variables? You can improve the ILP but there are still dependencies in the outer loop that don't get fixed. But here's how that might look:

```
do j = 1 to n
    do i = 1 to n
        t[i] = t[i] + a[i][j]
```

First we changed the orders of the loops. Since "ILP = Inner Loop Parallelism," let's make the i loop, which has more parallelism, the inner loop. Also we effectively create n separate 's' variables and perform renaming in software. The result of this code will be the same as the initial version but now the operations in the inner loop are completely independent. Our dependence graphs are much shorter now, as seen below.

So this is better right? Well, perhaps not since we've added slow memory accesses into the code. What about a more moderate approach then, say, using two 's' variables. Two variables could presumably be kept in registers and should double the amount of parallelism in the code.

```
do i = 1 to n by 2
    s = 0
    do j = 1 to n
        s = s + a[i][j]
    t[i] = s
    s = 0
    do j = 1 to n
        s = s + a[i+1][j]
    t[i+1] = s
```

Now we have two 's' variables, but we don't get any benefit from them; we've only made the code longer. So let's rename the second s since there are no true dependencies between the two halves of the loop, only anti-dependencies that renaming will resolve.

```
do i = 1 to n by 2
    s0 = 0
    s1 = 0
    do j = 1 to n
        s0 = s0 + a[i][j]
        s1 = s1 + a[i+1][j]
    t[i] = s0
    t[i+1] = s1
```

Now we have fused the two loops into one that has two independent operations inside it. And sure enough, this version has double the performance of the previous versions. But not only do superscalar machines not try to take advantage of this, even compilers generally don't. There is still a lot of concurrency out there waiting to be exploited.