CIS501: Superscalar
(Multiple-Issue)

Shreyas S. Shivakumar

December 2, 2019

1 Lecture Notes

1. **Parallelism**
   - **Pipeline Level Parallelism (PLP):** Your program will work on the *Execute Stage* of one instruction in parallel to working on the *Decode Stage* of the next instruction.
   - **Instruction Level Parallelism (ILP):** Your program will have two *independent* instructions working in parallel in the *Execute Stage*. Here parallelism is exhibited *within* a pipeline stage.
   - **Data Level Parallelism (DLP):** A single instruction will process multiple data in parallel, for *example:* one instruction results in 4 64-*bit* ADD operations. This is seen in **GPUs**.
   - **Thread Level Parallelism (TLP):** Multiple software threads running at the same time on multiple different CPU cores.

2. **What is the *Flynn Bottleneck*?**

   Even under ideal circumstances, there is a performance limit, i.e CPI = IPC = 1. And *practically* speaking, this limit is never achieved because of all the different hazards and overheads that exist in hardware.

   We can try to *super-pipeline* our design by adding more pipelined layers, but as we've seen before there is diminishing returns associated with this.

3. What is the motivation behind *Superscalar* design?

If we had two *independent* instructions in sequence that in no way conflicted with each other, why not execute them *both at the same time*?

We need to ensure that the two instructions are *independent*. This involves *dependency checking* of destination and source registers of the instructions.

4. How many instructions can you stack up?

Usually designs are approximately 4 instructions *wide*. The benefits start to diminish quickly once you go beyond this range.

However, certain designs will allow different stages of the pipeline to have different widths. We will be instead looking at *uniform width superscalar designs*.

5. Overview

(a) Fetch an entire block into cache since more than one instruction needs to be fetched at a time. This is usually 16B or 32B cache blocks resulting in 4 or 8 instructions being fetched at a time. Many of these instructions could be *branch instructions*, but for simplicity we will assume **a single branch instruction per cycle**.

(b) Decoding of both instructions cannot happen completely in parallel since we need to *check for conflicting instructions*, i.e the output register of instruction I_1 is an input register to I_2? Additionally, checks for *load-to-use*, *structual hazards* and *bypassing* would be needed too.

(c) The register file would need more ports - this means many more MUXes and more *read* and *write* ports.

(d) More **ALUs** are required in the *Execution Stage* - simple adders are easy but bypass paths are expensive.

(e) The *Memory Unit* is not doubled - this means that only a single *load* instruction can be executed per cycle but this is ok.

6. Superscalar Implementation Challenges

- We need to determine when instructions can proceed in parallel. This also requires **more complex stall logic** - order N^2 for an $N - wide$ machine.

- For superscalar *Register Read*, we would need 2 read ports per $N-width$, i.e a 4-wide machine would require 8 read ports. **Quadratic scaling** in the number of ports.

$$\text{Latency } \textit{and} \text{ Area } \propto \text{ Number of ports}^2$$

- We require more complex bypassing since there are more possible sources for data values. It is **Order**$(N^2 \times P)$ for $N-wide$ machines with pipeline depth P. This is clearly *expensive* to scale.

7. Superscalar Bypass and Register Files

For an $N-wide$ machine, you would ideally need an N^2 bypass network with $(N+1)$ input muxes at each ALU input and N^2 point-to-point connections. And this is just one **MX** bypass - we need to also do this for **WX** bypassing.

Q: Which is worse, N^2 bypassing or N^2 stalling?

N^2 bypassing is far worse since the values are much larger (64-bit quantities instead of 5-bit quantities). There are multiple levels of bypassing (MX, WX, etc.).

Q: How can you mitigate the N^2 bypassing?

Clustering - group ALUs into **K** clusters. Within each cluster provide full bypassing and limited bypassing between clusters. This constraining of the connections gives up arbitrary bypassing by typical cases will be much faster. We can also take advantage of the full bypassing between a cluster by **Steering** dependent instructions to the same cluster.

From $(N+1)$ inputs it is now $(\frac{N}{K}+1)$. Instead of N^2 bypass paths, it is now $(\frac{N}{K})^2$ bypass paths.

Q: What about Register Files?

We can replicate a register file per cluster. This causes some additional complexity since there are now two copies in each register. But we can design the system such that *register writes* update all the replicas. Ideally, we design systems with *more read ports* than *write ports* and this allows us to cut down on *read ports* by reducing the number of read ports from $2N$ to $\frac{2\times N}{K}$.

8. **How do we improve Superscalar Fetching?**
    - **Over-Fetch and Buffer** - add a queue between *Fetch Stage* and the *Decode Stage*. This compensates for cycles that fetch less than maximum instructions. *Fetch* and *Decode* are at different rates allowing for smoother execution.
    - **Loop Stream Detector** - Special pattern detector to identify and process loops that is very fast.
9. **Superscalar Implementations**
    - Statically Scheduled (in-order) Superscalar - *unmodified sequential programs*
    - Very Long Instruction Word (VLIW) - *compiler identifies independent instructions*. This is good because you don't need to do any dependency checking, no bypassing, move all burden to the compiler. If the compiler cannot identify such instructions, a *no-op* is inserted.
    - Dynamically Scheduled Superscalar - *hardware extracts more ILP by reordering* - Out of order execution. It is more complicated but is very popular now.
10. **Performance**

$$CPI_{base-superscalar} = \frac{1}{Superscalar\ Width}$$