

How are high speeds being realized?

- Faster and faster processors (implicit parallelism and/or fine grain parallelism)
- More and more parallelism (explicit parallelism and/or medium/coarse grain parallelism)

Implicit Parallelism

- Serial Parallelism, Peephole Optimizations, Pipelining
- Mostly in the **order of 2-6**
- Inherently part of processor/cache/memory design
- Requires no active involvement of the programmer (it's for free)
- Enabled through the explosion of transistor on chip (billions on a processor IC, tens of billions on a memory IC)

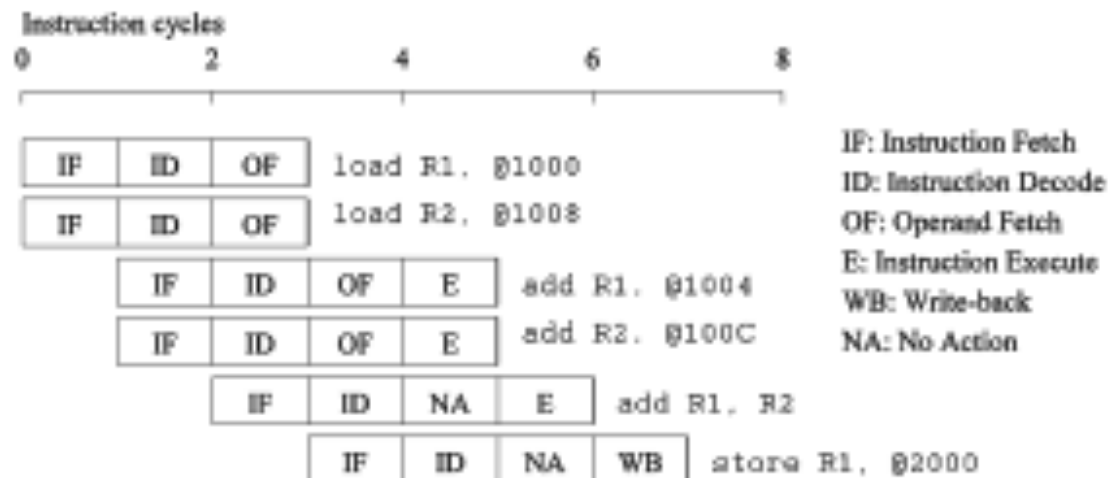
Trends in Processor Architectures:

➔ Substantial Increase in clock speeds and transistor counts

- How best to utilize these resources.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.

Pipelining and Superscalar Execution

- Pipelining overlaps various stages of instruction execution to achieve better performance.
- An instruction can be executed while the next one is being decoded and the next one is being fetched.



Pipelining and Superscalar Execution

- Pipelining, however, has several limitations.
- The speed of a pipeline is eventually limited by the slowest stage.
- For this reason, conventional processors rely on very deep pipelines (up to 20 stage pipelines in state-of-the-art Intel Core processors).
- However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.
- The penalty of a miss-prediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

→→→ Multiple Pipelines (Superscalar)

Illustration

Speedup of a pipeline of depth k and average number of pipelined instructions n :

$$S(n, k) = k \cdot n / (n + k - 1)$$

So, for $k = 20$: $S(n, 20) = 20n / (n + 19)$.

$$n=10: \quad 6.9$$

$$n=100: \quad 16.8$$

$$n=1000: \quad 19.6 \quad \rightarrow \quad 20 (k)$$

However, with $p\%$ miss prediction and branch every 5 instructions: average length $(100/p) \cdot 5 = 500/p$. So

$p = 10\%$: average length = 50. So for all n : maximal speedup = 14.5

$p = 20\%$: average length = 25. So for all n : maximal speedup = 11.4

$p = 30\%$: average length = 16. So for all n : maximal speedup = 9.4

Considering the fact that implementing pipelines generates delays (latches), in general there is a performance loss of 50%, making the maximal speedups:

7.2 ($p=10\%$)

5.7 ($p=20\%$)

4.7 ($p=30\%$)

Superscalar Execution

Scheduling of instructions is determined by a number of factors:

- True Data Dependency: The result of one operation is an input to the next.
- Resource Dependency: Two operations require the same resource.
- Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.
- The scheduler looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
- The complexity of this scheduler is an important constraint on superscalar processors.

Instruction Issue Mechanisms

- instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called **in-order** issue.
- In a more aggressive model, instructions can be issued **out-of-order**. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.
- Performance of in-order issue is generally limited.

Superscalar Execution: Efficiency Considerations

- Not all functional units can be kept busy at all times.
- If during a cycle, no functional units are utilized, this is referred to as vertical waste.
- If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.
- Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.
- Conventional microprocessors typically support four-way superscalar execution.

Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
- These instructions are packed and dispatched together, and thus the name very long instruction word.
- This concept was used with some commercial success in the Multiflow Trace machine (circa 1984).
- Variants of this concept are employed in the Intel IA64 processors.

Very Long Instruction Word (VLIW)

Processors: Considerations

- Issue hardware is simpler.
- Compiler has a bigger context from which to select co-scheduled instructions.
- Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.
- Branch and memory prediction is more difficult.
- VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
- Typical VLIW processors are limited to 4-way to 8-way parallelism.

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters: **latency** and **bandwidth**.
- Latency can be improved by providing caches between processor and memory
- Bandwidth can be improved by increasing the amount of memory interleaving (banks) and thereby increasing memory block size.

Impact of Memory Bandwidth: an Example

Consider the following code fragment, which sums columns of the matrix `b` into a vector `column_sum`:

```
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    column_sum[i] += b[j][i];
```

- ➔ Normally the vector `column_sum` is small and easily fits into the cache.
- ➔ The matrix `b` is accessed in a column order, resulting in very bad striding behavior, reducing memory bandwidth significantly

Impact of Memory Bandwidth: an Example

We can fix the code as follows:

```
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    for (i = 0; i < 1000; i++)
        column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

Other ways of reducing (memory) latencies

- Multithreading allows delays to be hidden by delaying execution of one thread in favor of a thread which is not delayed.
- Prefetching allows data to be put in cache before the processor actually needs the data