Macroarchitecture vs. microarchitecture

Microarchitecture is concerned with how processors and other components are put together. Macroarchitecture is concerned with how processors and other components can be connected to do useful work.

This is a course in macroarchitecture.

Why parallel architecture?

In the early days of computing, the best way to increase the speed of a computer was to use faster logic devices.

However, the time is long past when we could rely on this approach to making computers faster.

As device-switching times grow shorter, propagation delay becomes significant.

Logic signals travel at the speed of light, approximately 30 cm./nsec. in a vacuum. If two devices are one meter apart, the propagation delay is approximately

In 1960, switching speed was 10-100 nsec.

Nowadays, switching speed is typically measured in picoseconds.

Then how can we build faster computers?

The performance of highly integrated, single-chip CMOS microprocessors is steadily increasing.
In fact, these fast processors are now the best building blocks for multiprocessors.

So, to get performance better than that provided by the fastest single processor, we should figure out how to hook those processors together rather than rely on exotic circuit technologies and unconventional machine organizations.

**Application trends**

Given a serial program, it is usually not easy to transform it into an effective parallel program.

The measure of whether a parallel program is effective is how much better it performs than the serial version. This is usually measured by speedup.

Given a fixed problem, the speedup is measured by—

\[
\text{Speedup}(p \text{ processors}) \equiv \frac{\text{Time}(1 \text{ processor})}{\text{Time}(p \text{ processors})}.
\]

What kinds of programs require the performance only multiprocessors can deliver?

A lot of these programs are simulations:

- Weather forecasting over several days
- Ocean circulation
- Evolution of galaxies
- Human genome analysis
- Superconductor modeling

Parallel architectures are now the mainstay of scientific computing—chemistry, biology, physics, materials science, etc.

Visualization is an important aspect of scientific computing, as well as entertainment.
In the commercial realm, parallelism is needed for on-line transaction processing and “enterprise” Web servers.

A good example of parallelization is given on pp. 8–9 of Culler, Singh, and Gupta.

AMBER (Assisted Model Building through Energy Refinement) was used to simulate the motion of large biological models, such as proteins and DNA.

The code was developed on Cray vector supercomputers, and ported to the microprocessor-based Intel Paragon.

- The initial program (8/94) achieved good speedup on small configurations only.
- Load-balancing between processors improved the performance considerably (9/94).
- Optimizing the communication turned it into a truly scalable application (12/94).

This example illustrates the interaction between application and architecture. The application writer and the architect must understand each other’s work.

**Technology trends**

The most important performance gains derive from a steady reduction in VLSI feature size.

In addition, the die size is also growing.

This is more important to performance than increases in the clock rate. Why?

- 
- 
- 
- 

Clock rates have been increasing by about 30%/yr., while the number of transistors has been increasing by about 40%.

However, memory speed has lagged far behind. From 1980 to 1995,

- the capacity of a DRAM chip increased 1000 times,
- but the memory cycle time fell by only a factor of two.

This has led designers to use multilevel caches.

**Microprocessor design trends**

The history of computer architecture is usually divided into four generations:

- Vacuum tubes
- Transistors
- Integrated circuits
- VLSI

Within the fourth generation, there have been several subgenerations, based on the kind of parallelism that is exploited.

- The period up to ≈ 1986 is dominated by advancements in *bit-level* parallelism.

  However, this trend has slowed considerably.

  How did this trend help performance?

  Why did this trend slow?

- The period from the mid-1980s to mid-1990s is dominated by advancements in *instruction-level* parallelism.

  Pipelines (which we will describe in a few minutes) made it possible to start an instruction in nearly every cycle, even
though some instructions took much longer than this to finish.

- Today, efforts are focused on “tolerating latency.” Some operations, e.g., memory operations, take a long time to complete. What can the processor do to keep busy in the meantime?

**The Flynn taxonomy of parallel machines**

Traditionally, parallel computers have been classified according to how many instruction and data *streams* they can handle simultaneously.

- *Single or multiple instruction* streams.
- *Single or multiple data* streams.

**SISD machine**

An ordinary serial computer.

At any given time, at most one instruction is being executed, and the instruction affects at most one set of operands (data).

**SIMD machine**

At the right is a diagram of an an *array processor*.

Several identical ALUs, may process, for example, a whole array at once. However, the same instructions must be performed on *all* data items.
It is also possible for a single processor to perform the same instruction on a large set of data items. In this case, parallelism is achieved by pipelining—

- one set of operands starts through the pipeline, and
- before the computation is finished on this set of operands, another set of operands starts flowing through the pipeline.

We will describe the organization of a pipeline in a few minutes.

**MISD machine**

Several instructions operate simultaneously on each operand.

Generally unrealistic for parallel computers!

**MIMD machine**

Several complete processors connected together to form a multiprocessor.

- The processors are connected together via an *interconnection network* to provide a means of cooperating during the computation.
- The processors need not be identical.
- Can handle a greater variety of tasks than an array processor.
The MOMS taxonomy of parallel machines

The SIMD/MIMD taxonomy leaves something to be desired, since there are many subclasses of MIMD that do not appear in the model, and one class (MISD) that appears in the model but not in real life.

Gustafson (1990) proposes the following taxonomy.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td>MO</td>
</tr>
<tr>
<td>Distributed</td>
<td>DO</td>
</tr>
</tbody>
</table>

This yields the following classifications:

- MOMS (Monolithic operations, monolithic storage)
  
  “Apple-pie computing.”

- MODS (Monolithic operations, distributed data).

Examples: Connection Machine (CM-1, CM-2), MassPar.
• DOMS (Distributed operations, monolithic storage)

Examples: Sequent Balance and Symmetry, BBN Butterfly, Cray Y-MP.

• DODS (Distributed operations, distributed storage)

Examples: N-CUBE, Intel iPSC, Meiko Computing Surface.

In addition to these methods of obtaining parallelism among processors, there is this important approach to achieving parallelism within a processor.

**Pipelining**

Parallelism is achieved by starting to execute one instruction before the previous one is finished.

- The simplest kind overlaps the execution of one instruction with the fetch of the next instruction, as on a RISC.

Because two instructions can be processed simultaneously, we say that the pipeline has two *stages.*
Load and store reference memory, so they take two cycles.

- A pipeline may have more than two stages. Suppose, for example, that an instruction consists of four phases:
  1. Instruction fetch
  2. Instruction decode
  3. Operand fetch
  4. Execute

In a non-pipelined processor, these must be executed sequentially, so that a result is only available each four pipeline cycles (subcycles):

In a pipelined processor, after a delay to load the pipeline, a result is available each pipeline cycle.

- The type of pipelining described above achieves instruction-level parallelism—execution of multiple instructions in parallel.
• It is also possible to use pipelining to achieve data parallelism.

A vector processor usually has a long pipeline, and allows a large number of the same operations to take place concurrently. (Same operations, different data =

• A single “processor” may possess multiple pipelines, allowing different operations to use different pipelines (e.g., there might be a specialized addition pipeline, and another load pipeline).

For example, the CDC 6600 had ten separate functional units, with a scoreboard to keep track of which was in use at any time.

• Branches are a problem for pipelined computers.

Execution of some instructions may take longer than others. If there are two (or more) units capable of performing a given function (e.g., multiplication), then two operations of that type may be performed at once, providing that—