Advanced Parallel Architecture
Lesson 2

Annalisa Massini - 2016/2017

Introduction

Motivations to Parallel Architectures

- The question is:
  - Which **forces** and **trends** are giving parallel architectures an increasingly important role throughout the computing field?

- We have:
  - application demands (for increased performance)
  - technological trends
  - architectural trends
  - economics

Architectural Trends

- Advances in **technology** determine what is possible
- **Architecture** translates the potential of the technology into performance and capability

- Two ways to improve performance:
  - ***Parallelism*** **- multiple operations** performed in parallel
    - reduction of number of cycles to execute the program
    - **but** need for resources supporting simultaneous activities
  - ***Locality*** **- data references** performed close to the processor
    - accessing deeper levels of the storage hierarchy avoided
    - **but** need for resources providing local storage

Architectural Trends

- These two ways compete for the same resources
- The best performance is obtained by an *intermediate strategy* which devotes resources to exploit a ***degree of parallelism*** and a ***degree of locality***
- Indeed, parallelism and locality interact in **systems of all scales**, from within a chip to across a large parallel machine

Generations of Computer

- The history of computer architecture is traditionally divided into four generations (basic logic technology):
  - Vacuum tube - 1946-1957
  - Transistor - 1958-1964
  - Integrated circuits
    - *Small scale integration* - 1965 on

      Up to 100 devices on a chip
    - *Medium scale integration* - to 1971

      100-3,000 devices on a chip
    - *Large scale integration* - 1971-1977

      3,000 - 100,000 devices on a chip

Generations of Computer

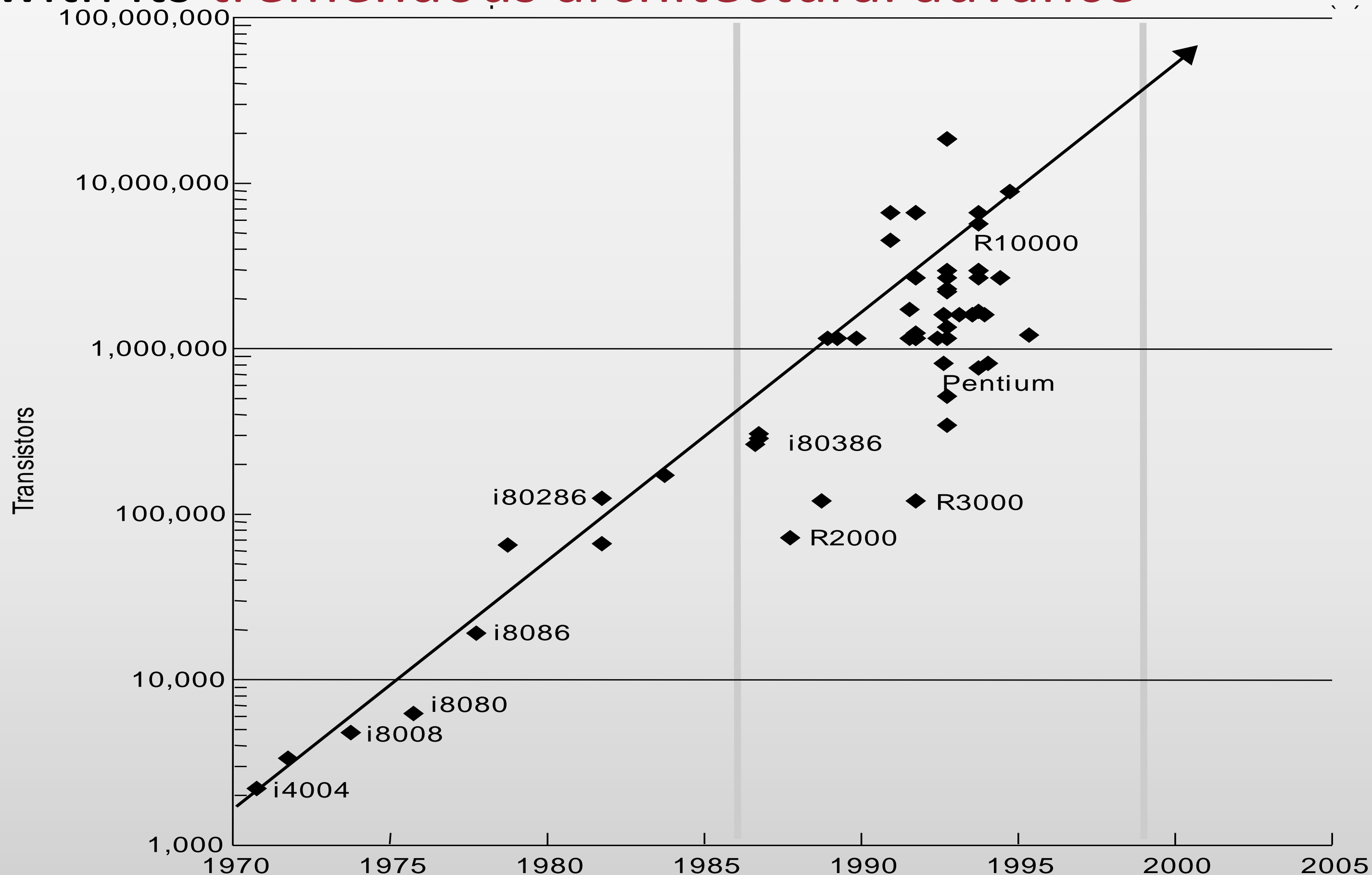
- The history of computer architecture is traditionally divided into four generations (basic logic technology):
  - VLSI
    - Very large scale integration - 1978-1991

      100,000 - 100,000,000 devices on a chip
    - Ultra large scale integration - 1991-

      Over 100,000,000 devices on a chip

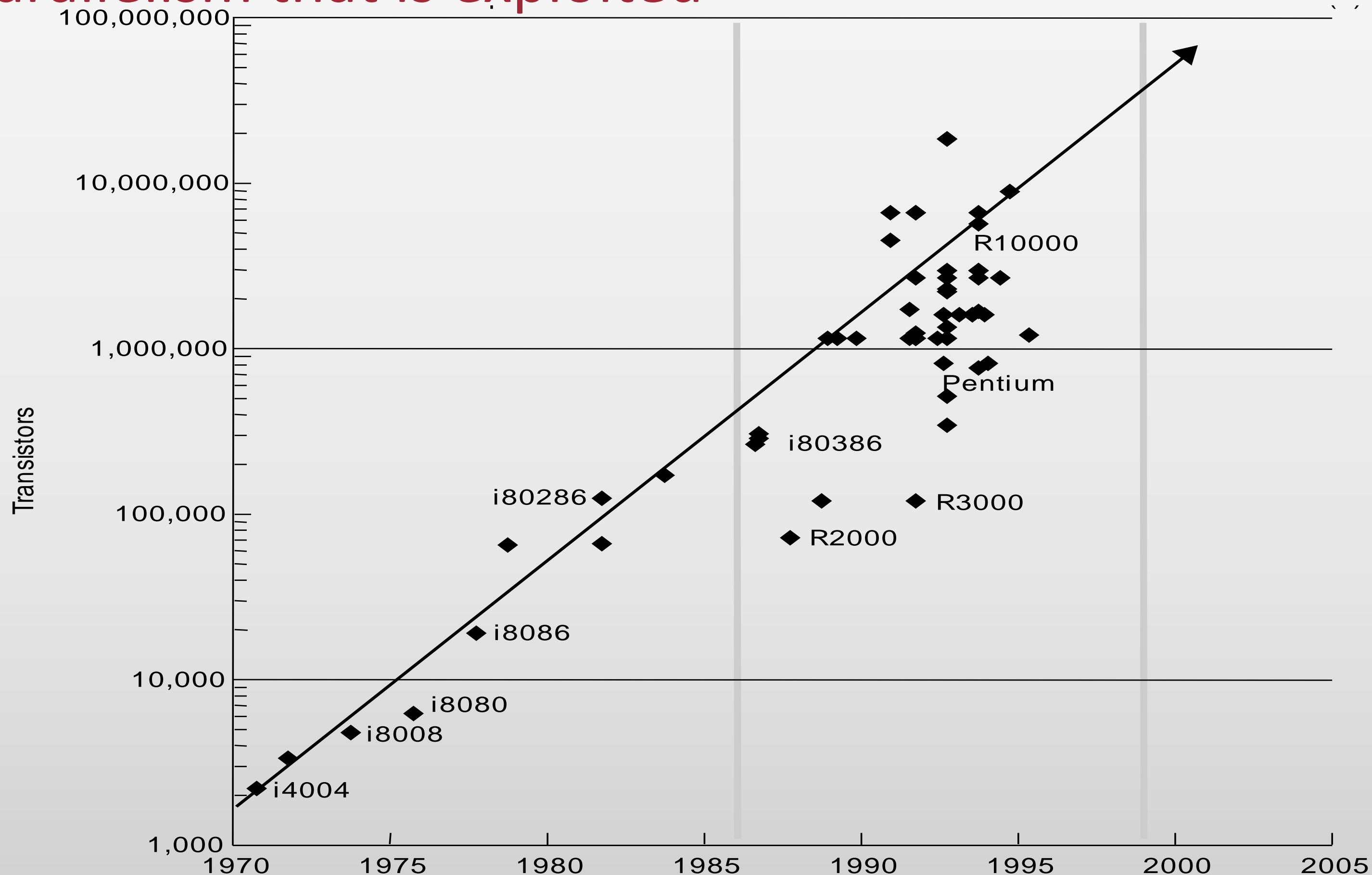
Architectural Trends

- The most interesting period is the fourth - VLSI generation - with its tremendous architectural advance

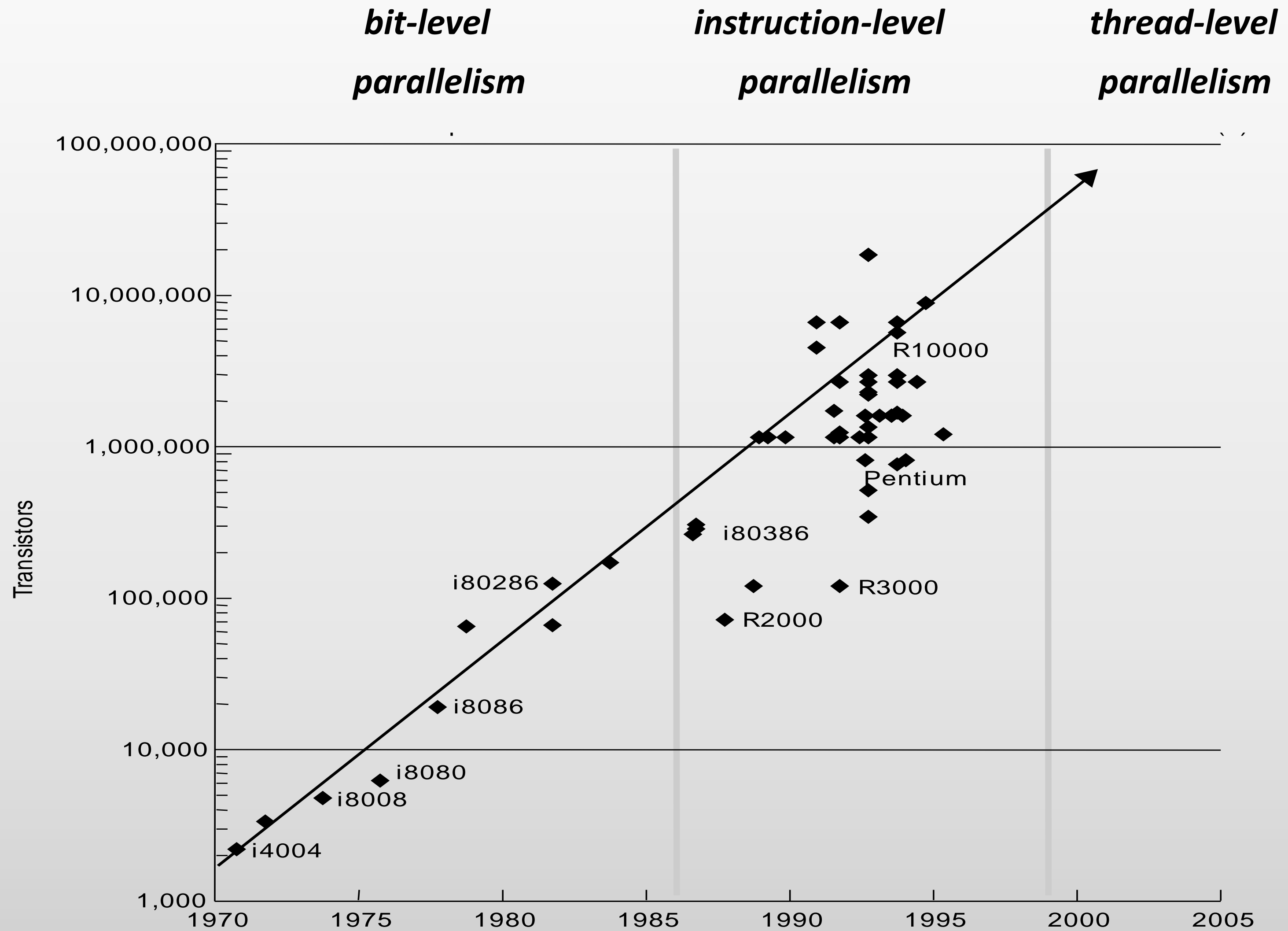


Architectural Trends

- The strongest delineation in VLSI generation is the kind of parallelism that is exploited



Architectural Trends



Architectural Trends

- The period up to about 1985 is dominated by advancements in ***bit-level parallelism***, with 4-bit microprocessors replaced by 8-bit, 16-bit, and so on

- Doubling the width of the datapath reduces the number of cycles required to perform a full 32-bit operation

- This trend slows once a 32-bit word size is reached in the mid-80s
- The adoption of 64-bit operation is reached decade later

Architectural Trends

- Further increases in word-width will be driven by demands for improved *floating-point representation* and a *larger address space*, rather than performance

- With address space requirements *growing by less than one bit per year*, the demand for 128-bit operation appears to be well in the future

- The early microprocessor period was able to reap the benefits of the easiest form of parallelism: bit-level parallelism in every operation

Architectural Trends

- ***Inflection point*** in the microprocessor growth curve corresponds to the arrival of full 32-bit word operation combined with **use of cache** (late ‘80s)
- The period from the mid-80s to mid-90s is dominated by advancements in ***instruction-level parallelism***
- Full word operation means that the basic steps in instruction processing (instruction decode, integer arithmetic, and address calculation) can be performed in ***a single cycle***

Architectural Trends

- By using caches, the instruction fetch and data access can also be performed in ***a single cycle***, most of the time

- Using the RISC approach - care in the instruction set design - it is straightforward to ***pipeline*** the stages of instruction execution
- Effect of the RISC approach → *an instruction is executed almost every cycle* (on average)

- In addition, advances in compiler technology made instruction pipelines more effective

Architectural Trends

- The mid-80s microprocessor-based computers consisted of a set of chips:
  - an integer processing unit
  - a floating-point unit
  - a cache controller
  - SRAMs for the cache data and tag storage

- As the chip capacity increased, these components were coalesced into ***a single chip*** - containing separate hardware for integer arithmetic, memory operations, branch operations, and floating-point operations - reducing the cost of communicating among them

Architectural Trends

- In addition to pipelining individual instructions, it became very attractive to fetch ***multiple instructions*** at a time and *issue them in parallel to distinct function units* whenever possible

- This form of instruction level parallelism - called ***superscalar*** execution - allow to exploit the increasing number of available chip resources

- *More function units* were added, *more instructions* were fetched at time, and more instructions could be issued in each clock cycle to the function units

Architectural Trends

- Instruction level parallelism approach is worthwhile if the processor can be supplied with instructions and data fast enough to keep it busy
- In order to satisfy this requirement, *larger and larger caches* were placed on-chip with the processor
- With the *processor and cache on the same chip*, the path between the two could be made very wide to satisfy the bandwidth requirement of multiple instruction and data accesses per cycle

Architectural Trends

- However, as more instructions are issued each cycle, the performance impact of each control transfer and each cache miss becomes more significant:
  - A **control transfer** may have to wait for the depth of the processor pipeline - *latency* - until a particular instruction reaches the end of the pipeline and determines which instruction to execute next
  - Similarly, instructions which use a value loaded from memory may cause the processor to wait for the latency of a **cache miss**

Architectural Trends

- Processor designs in the 90s deploy a variety of *complex instruction processing mechanisms* in an effort to reduce the performance degradation in superscalar processors

- Sophisticated *branch prediction techniques* are used *to avoid pipeline latency* by guessing the direction of control flow before branches are actually resolved

- Larger, *more sophisticated caches* are used *to avoid the latency of cache misses*

Architectural Trends

- Instructions are scheduled dynamically and allowed to complete ***out of order:***
  - If one instruction encounters a miss, other instructions can proceed ahead of it as long as they do not depend on the result of the instruction
  - A larger window of instructions that are waiting to issue is maintained within the processor
  - Whenever an instruction produces a new result, several waiting instructions may be issued to the function units

Architectural Trends

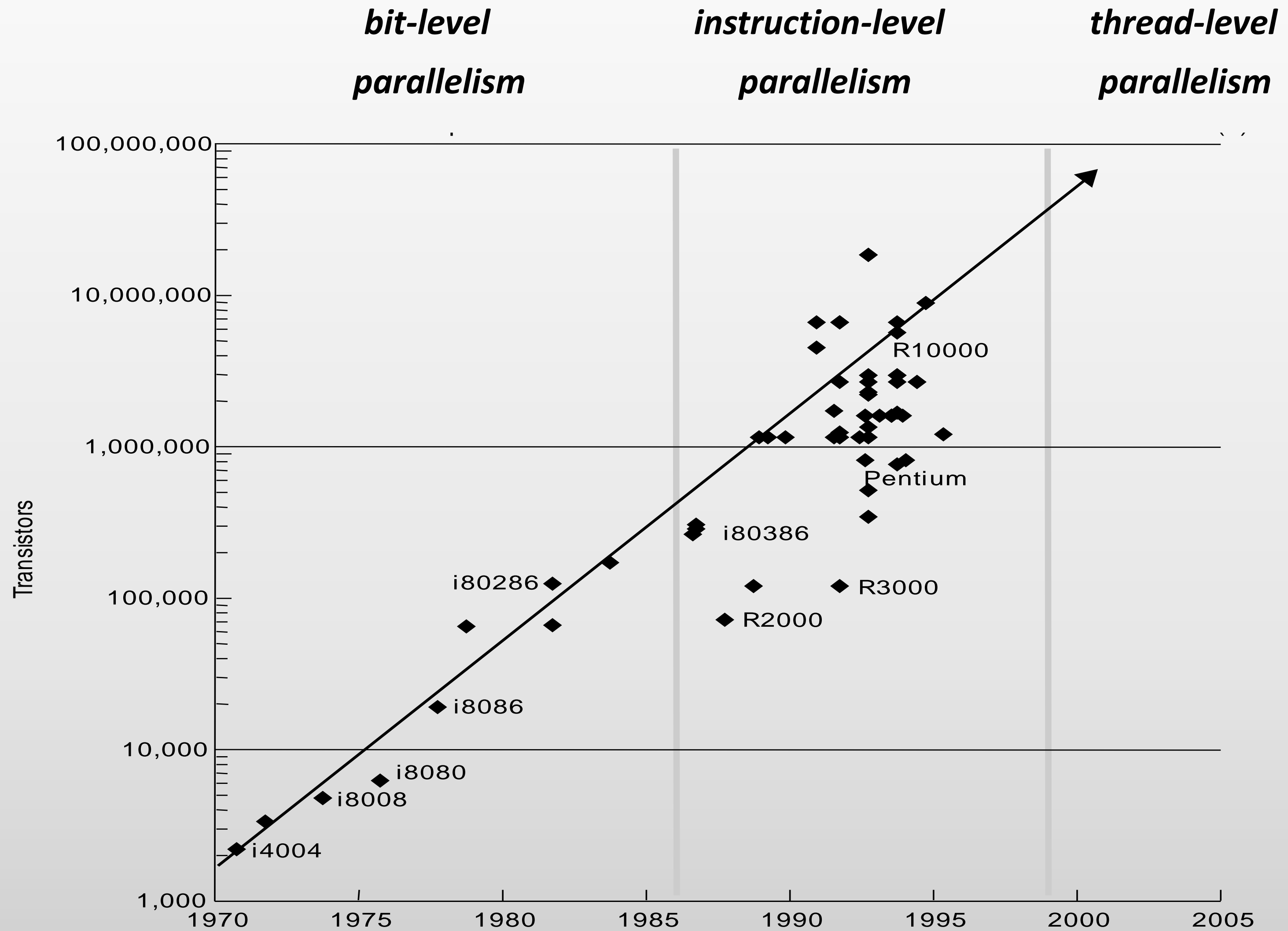
- These complex mechanisms allow the processor to *tolerate* the latency of a *cache-miss* or *pipeline dependence* when it does occur

- However, each of these mechanisms place a ***heavy demand*** on chip resources and a very heavy design cost

Architectural Trends

- In the late 90s, given the increases in chip density, the *instruction level parallelism within a single thread of control is overcome*
- The processors and their interconnect are all implemented on a single silicon chip and the new technology is multi-core processor
- The emphasis shifts to ***thread level parallelism***, parallelism available as multiple processes or multiple threads of control within a process

Architectural Trends



Architectural Trends

- By the early 2000s, CPU designers were thwarted from achieving higher performance from *instruction level parallelism techniques*
- The **growing disparity** between CPU operating frequencies and main memory operating frequencies as well as escalating CPU power dissipation implied new instruction level parallelism techniques
- CPU designers realize that to aggregate **performance of multiple programs** was more important than the *performance of a single thread or program*

Architectural Trends

- There was a proliferation of dual and multiple core CMP (chip-level multiprocessing) designs and the corresponding parallel execution at thread level

- Examples:
  - hyper-threading – 2 threads on the same pipeline executed in parallel (up to 30% speedup)
  - multi-core architectures – multiple CPUs on a single chip
  - multiprocessor systems (parallel systems)
  - manycore architectures (GPUs)

Supercomputers

- We have looked at the forces driving the development of parallel architecture and tecniques in the general market
- A second, confluent set of forces is driven by the request to achieve absolute maximum performance, or ***supercomputing***
- Although commercial and information processing applications are important drivers of the high end, historically, *scientific computing has been a kind of proving ground for innovative architecture*

Supercomputers

- Starting in the mid 70's, supercomputing was dominated by ***vector processors***
- Operations are executed on sequences of data elements - **vectors** - rather than individual scalar data
- Vector operations permit more parallelism to be obtained within a single thread of control

Supercomputers

- Within the ***vector processing approach***, the single processor performance improvement is dominated by **modest** improvements in cycle time and more substantial **increases** in the vector memory bandwidth

- In the ***microprocessor systems***, we see the combined effect of:
  - increasing clock rate
  - on-chip pipelined floating-point units
  - increasing on-chip cache size
  - increasing off-chip second-level cache size
  - increasing use of instruction level parallelism

Supercomputers

- ***Multiprocessor architectures*** are adopted by *both* the vector processor and microprocessor designs, but the scale is quite different

- The microprocessor based **supercomputers** provided initially about a *hundred processors*, increasing to roughly a *thousand from 1990 onward*

Supercomputers

- *Massively parallel processors* (MPPs) have tracked the microprocessor advance, with typically a lag of two years behind the leading microprocessor-based workstation or PC

- The performance advantage of the *MPP systems* over *traditional vector supercomputers* is less substantial on more complete applications owing to the relative immaturity of the programming languages, compilers, and algorithms

Summary: Why Parallel Architecture?

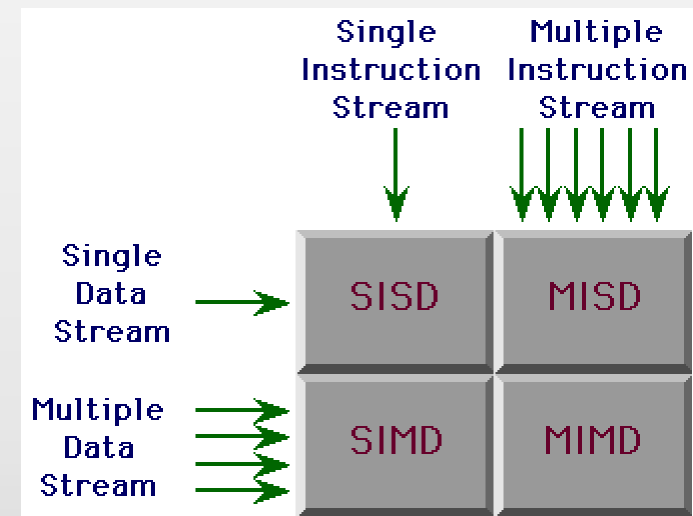
- Increasingly attractive
  - Economics, technology, architecture, application demand
- Increasingly central and mainstream
- Parallelism exploited at many levels
  - Instruction-level parallelism
  - Multiprocessor servers
  - Large-scale multiprocessors (“MPPs”)
- Same story from memory system perspective
  - Increase bandwidth, reduce average latency with local memories
- Spectrum of parallel architectures make sense
  - Different cost, performance and scalability

Taxonomy of Computer Architectures

- The idea of obtaining more performance by utilizing multiple resources is quite dated
- In 1966 Michael Flynn introduced a **taxonomy** of computer architectures that is still the *most common way of categorizing* systems with **parallel processing capability**

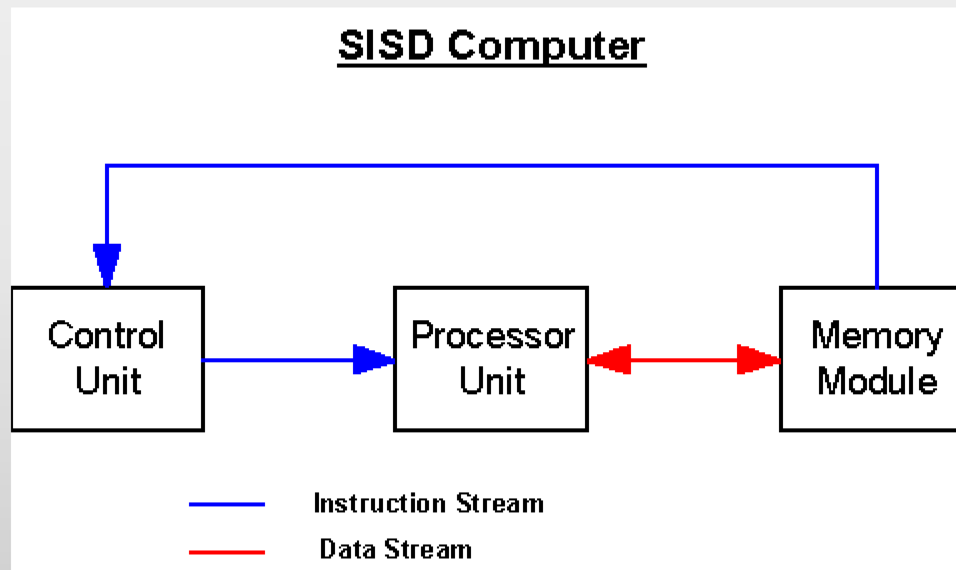
Taxonomy of Computer Architectures

- Machines are classified based on how many **data** items they can process concurrently and how many different **instructions** they can execute at the same time:
  - Single Instruction, Single Data - **SISD**
  - Single Instruction, Multiple Data - **SIMD**
  - Multiple Instruction, Single Data - **MISD**
  - Multiple Instruction, Multiple Data - **MIMD**



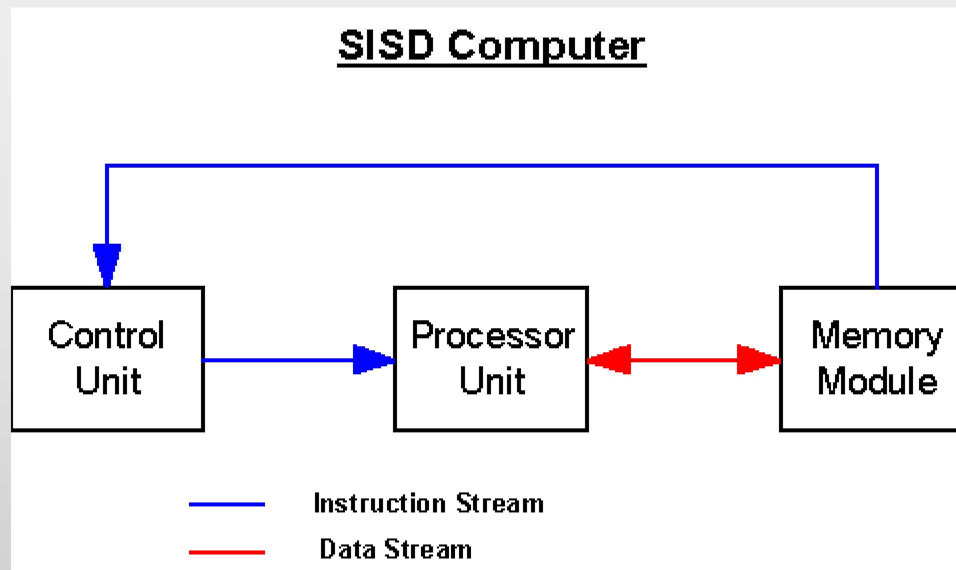
Single Instruction, Single Data Stream - SISD

- Single processor
- Single instruction stream
- Data stored in single memory



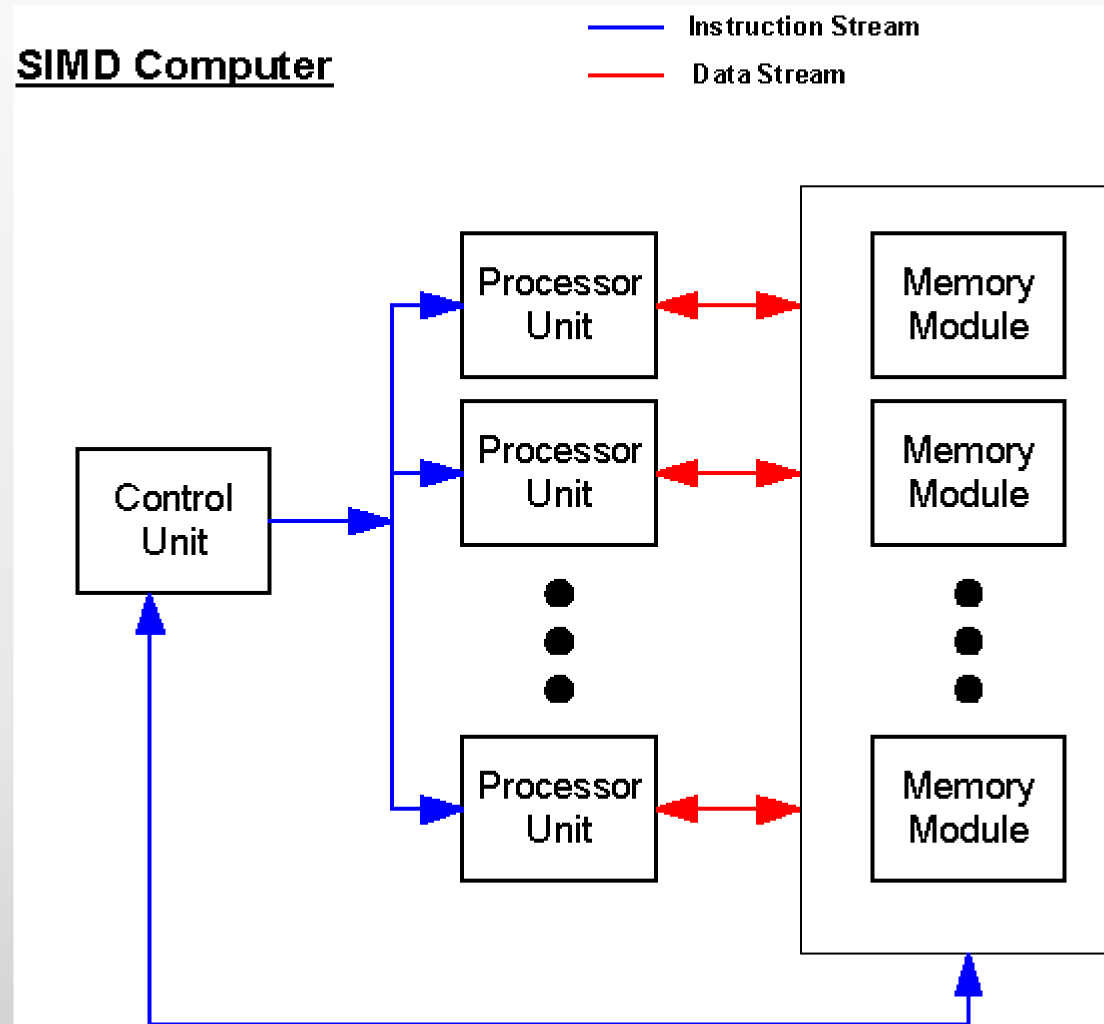
Single Instruction, Single Data Stream - SISD

- A single processor executes a single instruction at a time operating on data stored in a single memory
  - Uniprocessor fall into this category
  - The majority of contemporary CPUs is multicore
  - A single core can be considered a SISD machine



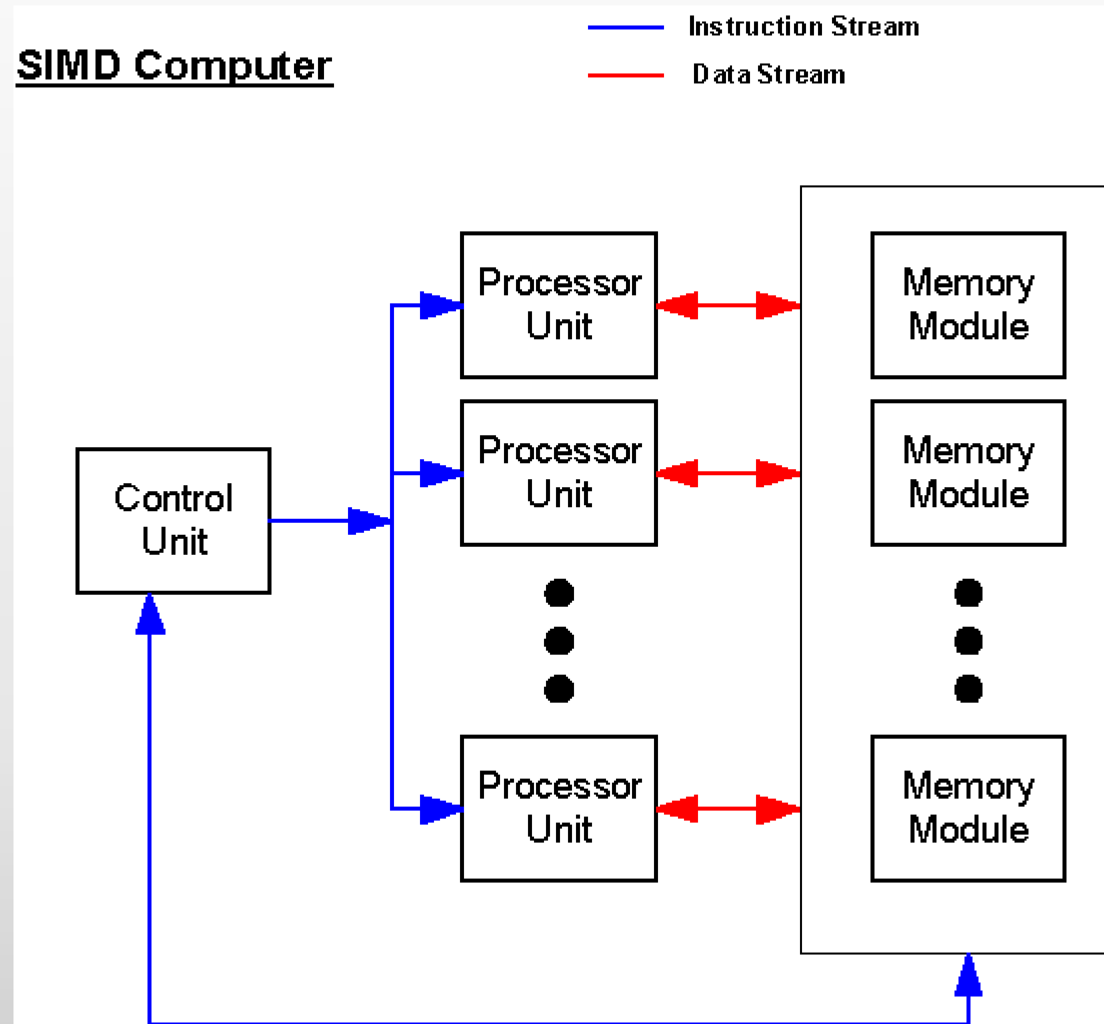
Single Instruction, Multiple Data Stream - SIMD

- A ***single machine instruction*** controls the simultaneous execution of a number of processing elements on a lockstep basis
- Each processing element has an associated data memory



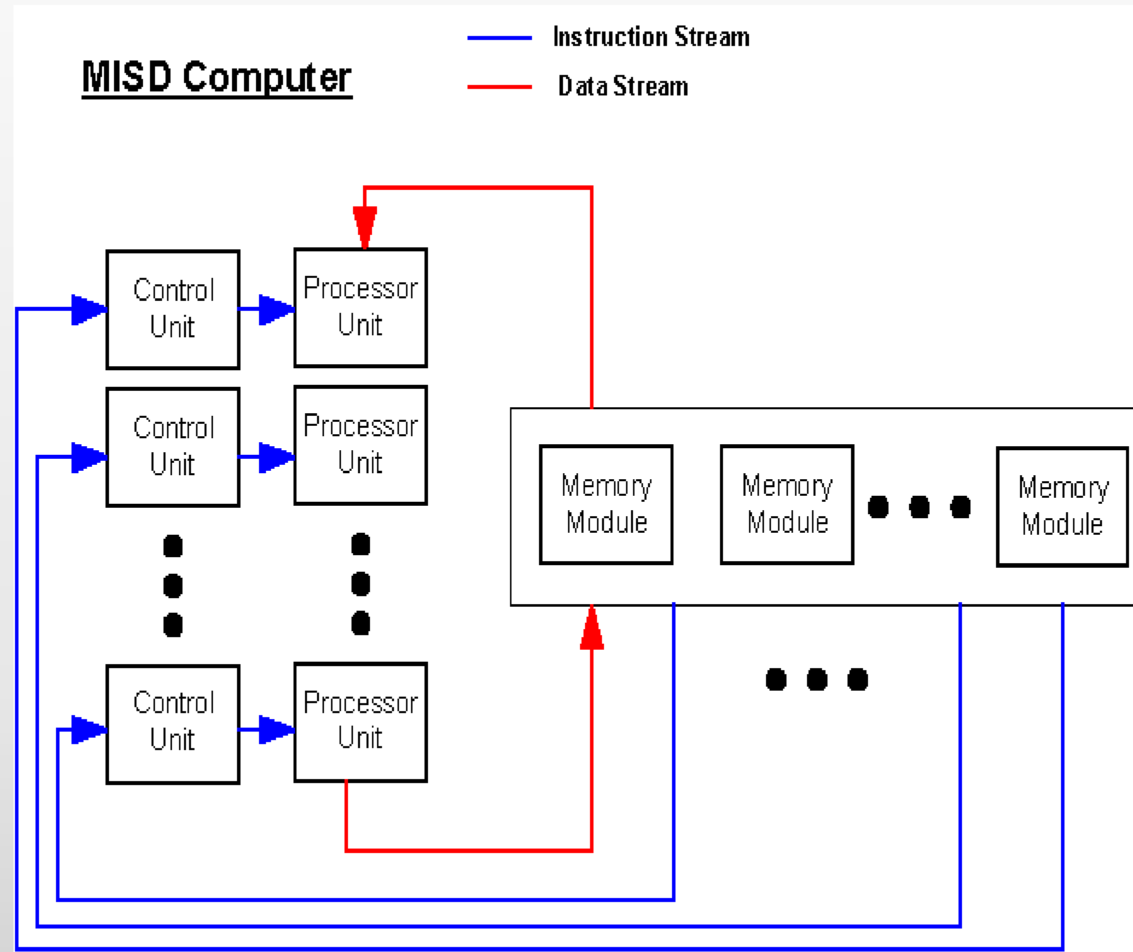
Single Instruction, Multiple Data Stream - SIMD

- ***Each instruction*** is executed on a ***different set of data*** by the ***different processors***
- Vector processors were the first SIMD machines
- GPUs follow this design at the level of Streaming multiprocessor



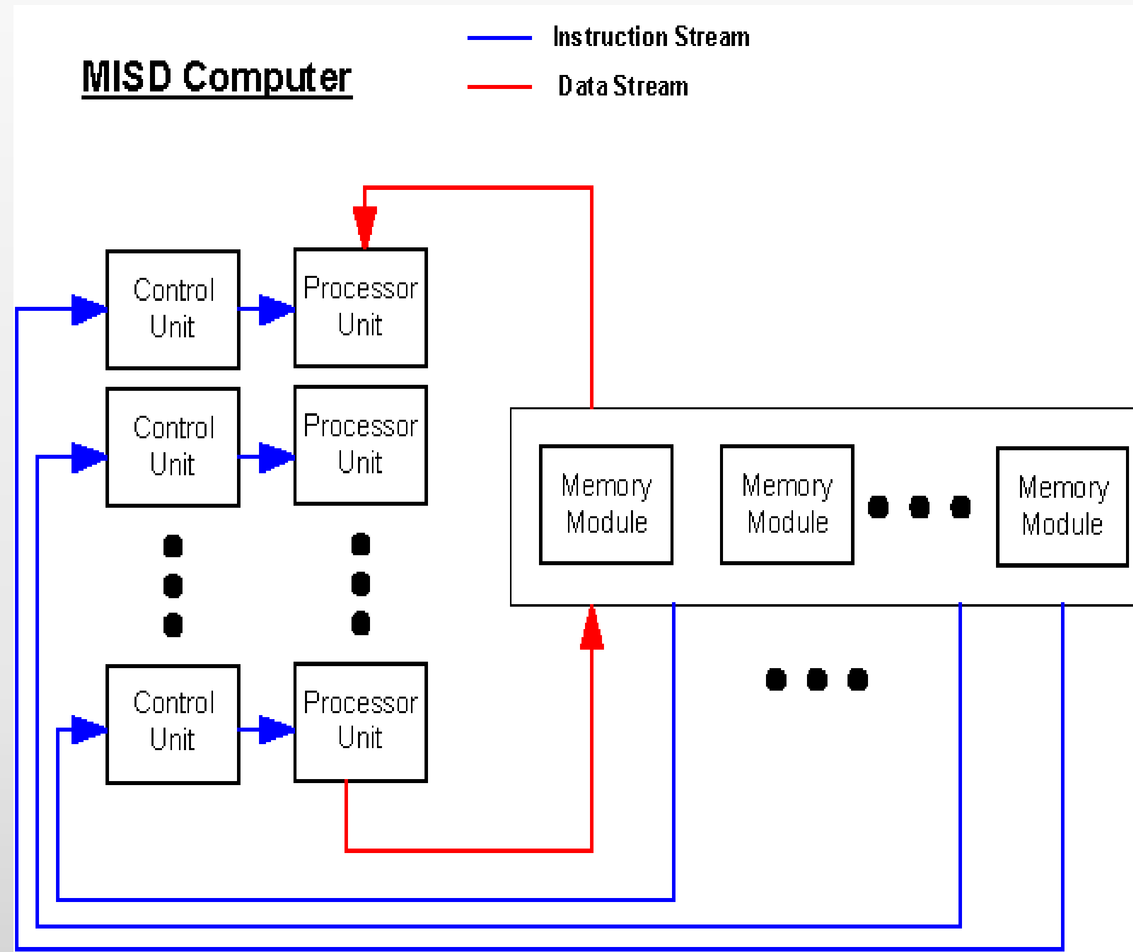
Multiple Instruction, Single Data Stream - MISD

- A ***sequence of data*** is transmitted to a *set of processors*, each of which *executes a **different** instruction* sequence
- This structure is not *commercially* implemented



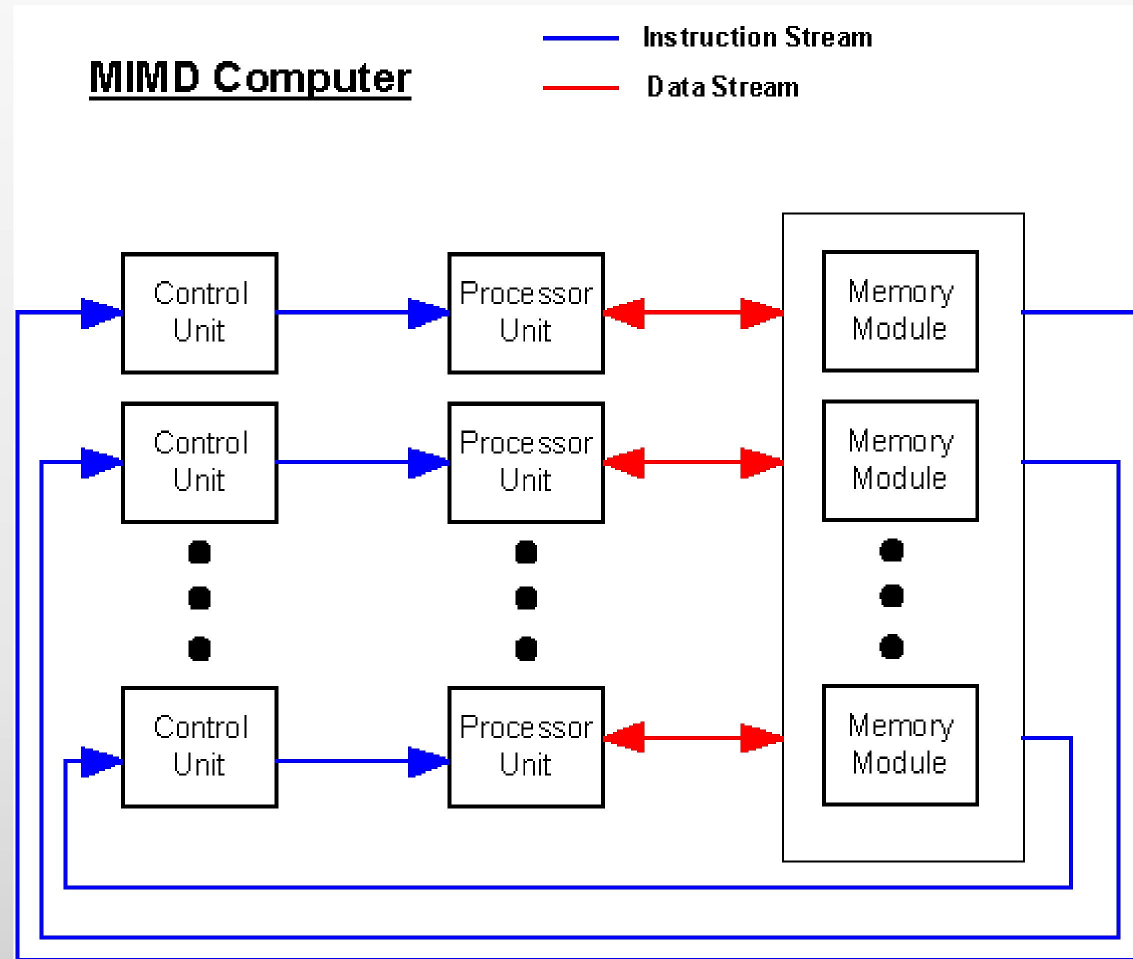
Multiple Instruction, Single Data Stream - MISD

- MISD computers can be useful in applications of a specialized nature:
  - robot vision
  - when *fault tolerance* is required in a system (military or aerospace application) data can be processed by multiple machines and decisions can be made on a majority principle



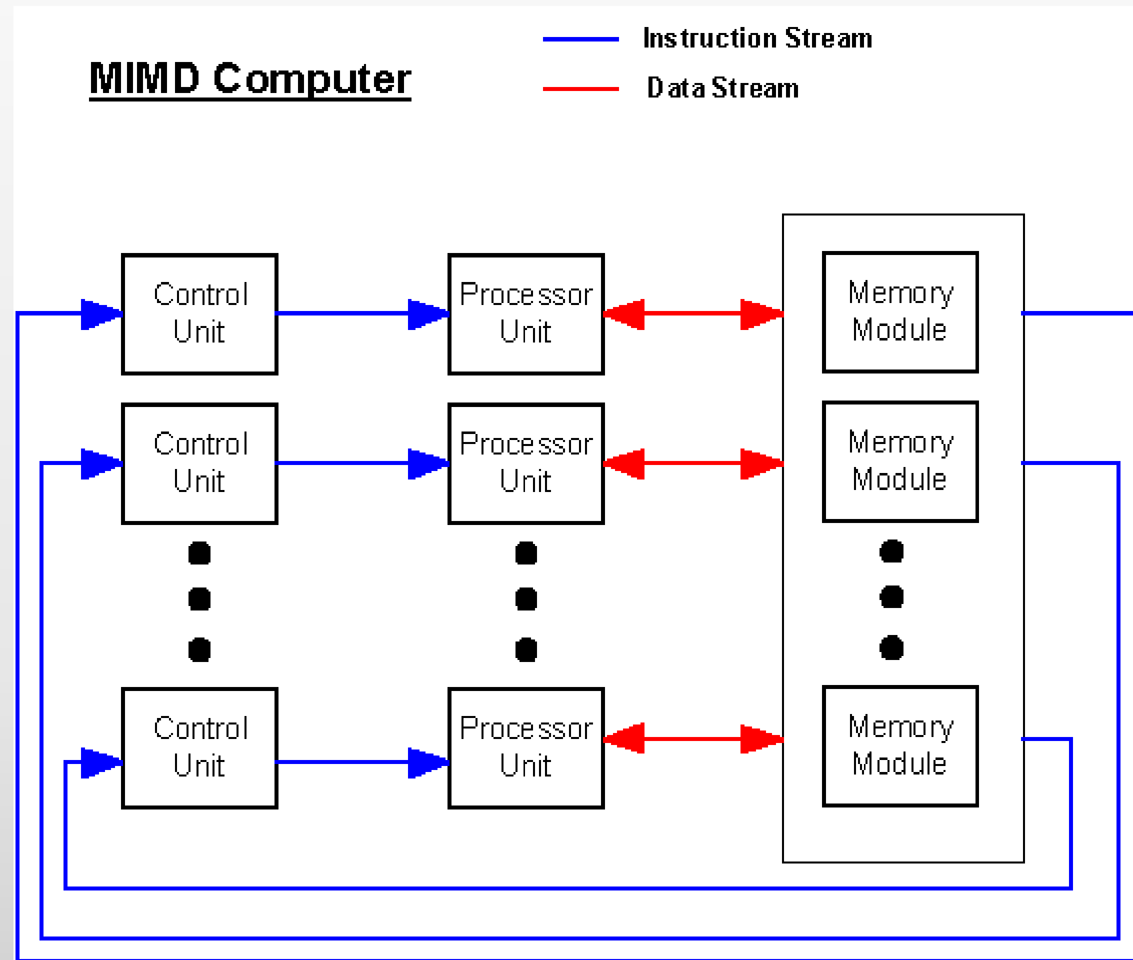
Multiple Instruction, Multiple Data Stream- MIMD

- A **set of processors** simultaneously execute **different instruction sequences** on **different data sets**
- This architecture is the most common and widely used form of parallel architectures

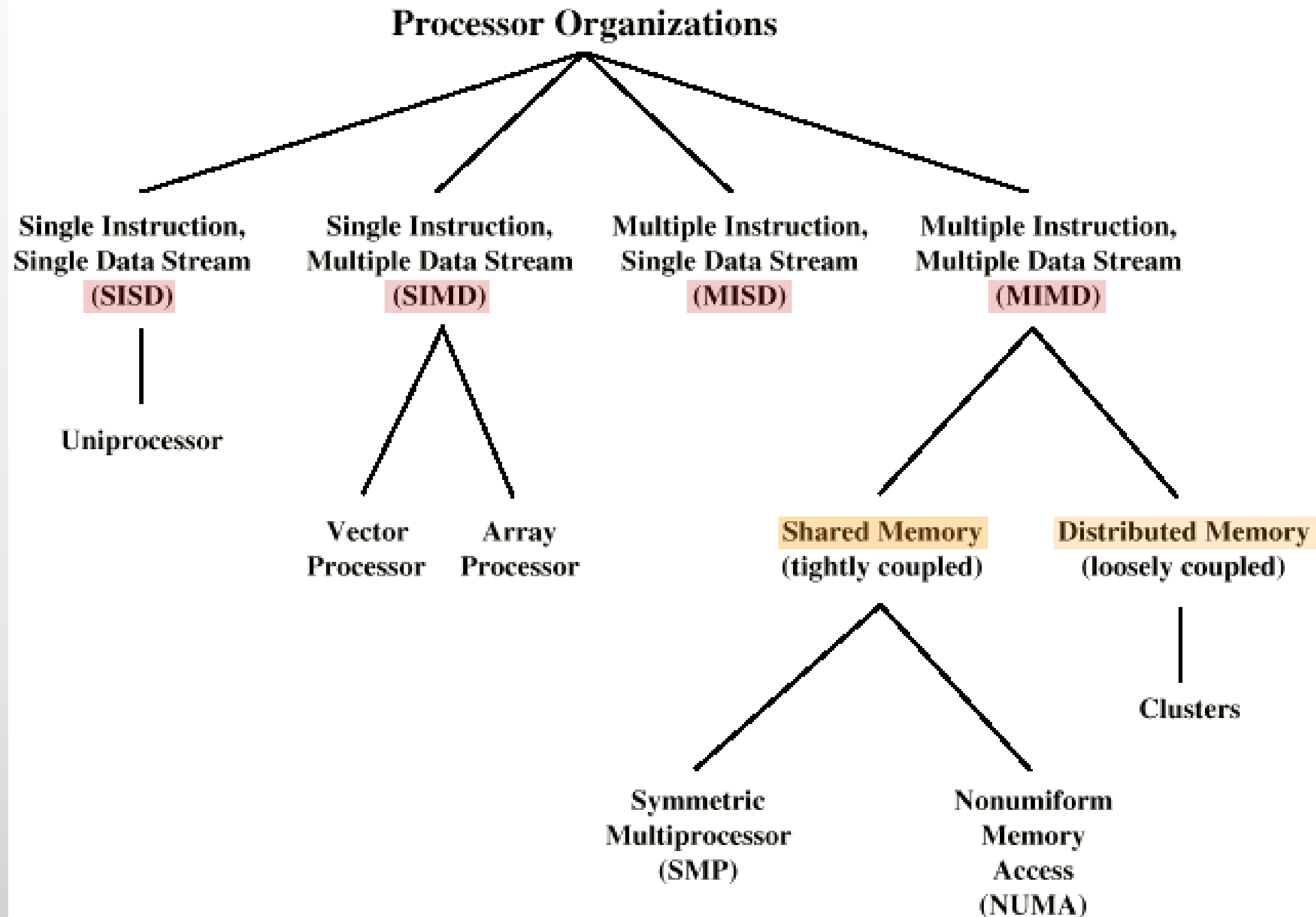


Multiple Instruction, Multiple Data Stream- MIMD

- General purpose processors
- Each processor can process all instructions necessary
- **Further classified** by method of processor communication



Taxonomy of Parallel Processor Architectures



Performance: speedup

- A key reference point for both the architect and the application developer is how the use of parallelism improves the **performance** of the application

- We may define the ***speedup*** on processors as

$$\text{Speedup (p processors)} = \frac{\textit{performance (p processors)}}{\textit{performance (1 processor)}}$$

- For a fixed problem size $\quad$ *performance = 1/time*

- $$\text{Speedup fixed problem (p processors)} = \frac{\textit{time (1 processor)}}{\textit{time (p processors)}}$$

Communication Performance

- To complete the introduction to the fundamental issues of parallel computer architecture, we need to understand performance at many levels of design
- Fundamentally, there are three performance metrics:
  - ***Latency***: time taken for an operation
  - ***Bandwidth***: rate of performing operations
  - ***Cost***: impact on execution time of program

Communication Performance

- If processor does one thing at a time:
  - bandwidth (operation per second) is about 1/latency
  - cost is simply latency x number of operations
- But actually it is more complex in modern systems
- Modern computer systems do many different operations at once and the relationship between these performance metrics is much more complex

Communication Performance

- Characteristics apply to overall operations, as well as individual components of a system

- Since the unique property of parallel computer architecture is **communication**, the operations that we are concerned with most often are **data transfers**

Linear Model of Data Transfer Latency

- The time for a **data transfer operation** is generally described by a linear model:
  - *Transfer time (n) = T_0 + n/B*
    - ***n*** is the amount of data (e.g. number of bytes),
    - ***B*** is the transfer rate of the component moving the data (e.g. bytes per second),
    - the constant term T_0 is the start-up cost

- This is a very convenient model, and it is used to describe a diverse collection of operations: messages, memory accesses, bus transactions, and vector operations

Linear Model of Data Transfer Latency

- It applies in many aspects of traditional computer architecture, as well, and we can observe:
  - For **memory operations**, it is essentially the **access time**
  - For **bus transactions**, it reflects the bus arbitration and command phases
  - For any sort of **pipelined operation**, including pipelined instruction processing or vector operations, it is the **time to fill pipeline**

Linear Model of Data Transfer Latency

- But a linear model is not enough:
  - It does not give any indication when the ***next such operation can be initiated***
  - It does not indicate whether other ***useful work can be performed during the transfer***

- These other factors depend on how the transfer is performed:
  - ***need to know how transfer is performed***

Communication Cost Model

- The data transfer in which we are most interested is the one that occurs **across the network** in parallel machines
- It is initiated by the processor through the ***communication assist***
- The essential components of this operation can be described by the following simple model:

Communication Time (n)= Overhead + Network Delay + Occupancy

Communication Cost Model

Communication Time (n)= Overhead + Network Delay + Occupancy

- The **Overhead** is the time the processor spends initiating the transfer
  - It may be a fixed cost, if the processor simply has to tell the communication assist to start
  - It may be linear in *n*, if the processor has to copy the data into the assist
- This is time the processor:
  - is busy with the communication event
  - cannot do other useful work or initiate other communication

Communication Cost Model

Communication Time (n)= Overhead + Network Delay + Occupancy

- The remaining portions of the communication time is considered the **network latency**; it is the part that can be hidden by other processor operations

Communication Cost Model

Communication Time (n)= Overhead + Network Delay + Occupancy

- The **Occupancy** is the time it takes for the data to pass through the slowest component on the communication path:
  - The data will occupy other resources, including buffers, switches, and the communication assist
  - Often the communication assist is the bottleneck that determines the occupancy
  - The occupancy limits how frequently communication operations can be initiated
  - The next data transfer will have to wait until the critical resource is no longer occupied before it can use that same resource

Communication Cost Model

Communication Time (n)= Overhead + Network Delay + Occupancy

- The remaining communication time is the ***Network Delay***, which includes:
  - the time for a bit to be routed across the actual network
  - other factors, such as the time to get through the communication assist
- From the processors viewpoint, the specific hardware components contributing to network delay are indistinguishable

Communication Cost Model

- A useful model connecting the program characteristics to the hardware performance is given by

  Communication Cost = frequency * (Comm time - overlap)

- The *frequency of communication*:
  - is defined as the number of communication operations per unit of work in the program
  - it depends on many programming factors and many hardware design factors

Communication Cost Model

Communication Cost = frequency * (Comm time - overlap)

Note that:

- Hardware may:
  - limit the transfer size and determine the min number of messages
  - replicate data or migrate it to where it is used
- A certain amount of communication is inherent to parallel execution, since data must be shared and processors must coordinate their work
- A machine can support programs with a high communication frequency if the other parts of the communication cost are small: low overhead, low network delay, and small occupancy