

2. Machine Organization



Origins

Charles Babbage in the 19th Century

Turing Machine

Early 20th Century machines

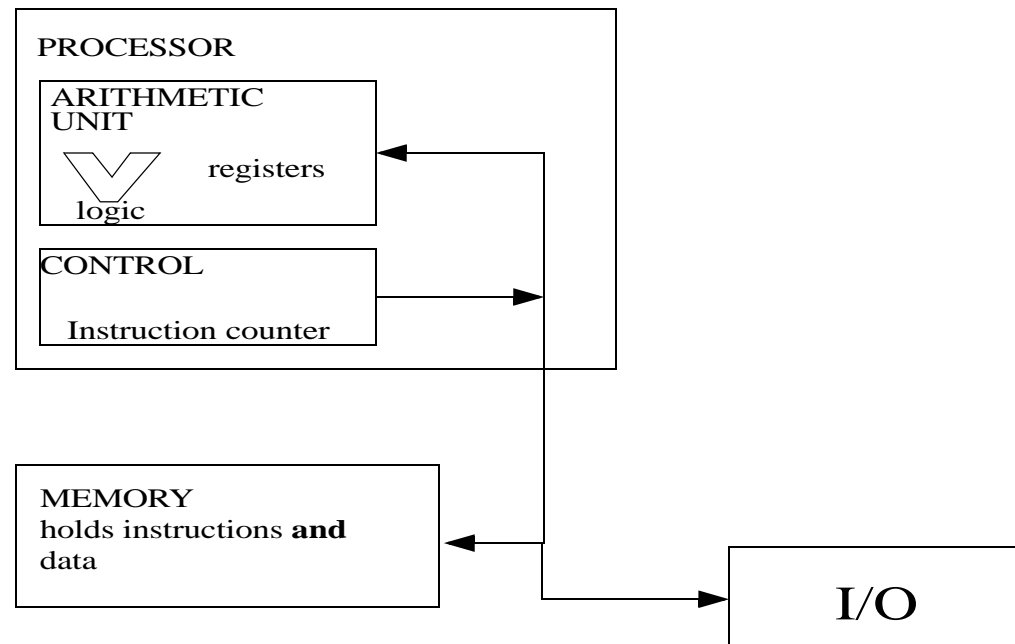
[see http://en.wikipedia.org/wiki/Harvard_Mark_I]



The Von Neumann Computational Model

[Almasi and Gottlieb: *Highly Parallel Computing*. Benjamin Cummings, 1988.]

- **IAS (Institute for Advanced Study (IAS), in Princeton)**
Designed by John Von Neumann in the late 1940s.
- **All widely used “conventional” machines follow this model. It is represented next:**



For an instruction to be executed, there are several steps that must be performed. For example:

- 1. Instruction Fetch and decode (IF). Bring the instruction from memory into the control unit and identify the type of instruction.**
- 2. Read data (RD). Read data from memory.**
- 3. Execution (EX). Execute operation.**
- 4. Write Back (WB). Write the results back.**

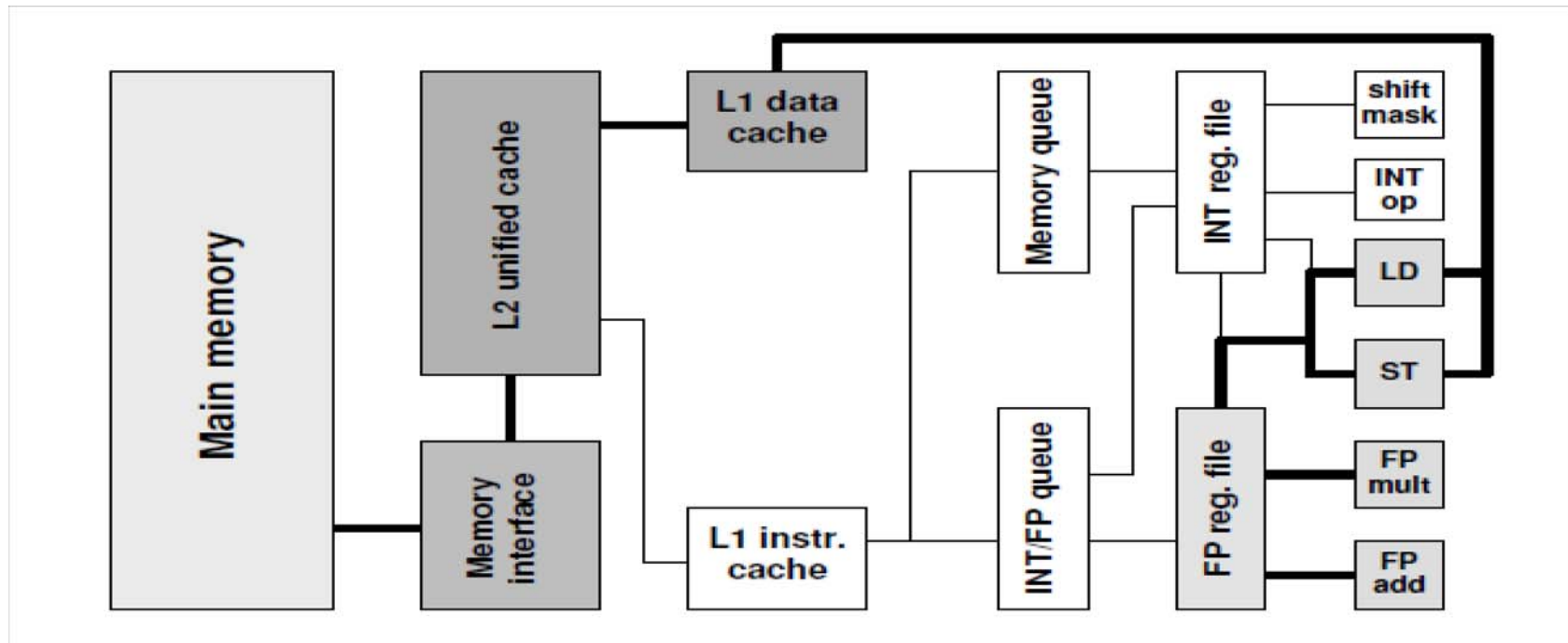


The machine's essential features are:

- 1. A processor that performs instructions such as “add the contents of these two registers and put the result in that register”**
- 2. A memory that stores both the instructions and data of a program in cells having unique addresses.**
- 3. A control scheme that fetches one instruction after another from the memory for execution by the processor, and shuttles data one word at a time between memory and processor.**



Von Neuman machines in the Textbook (1/2)

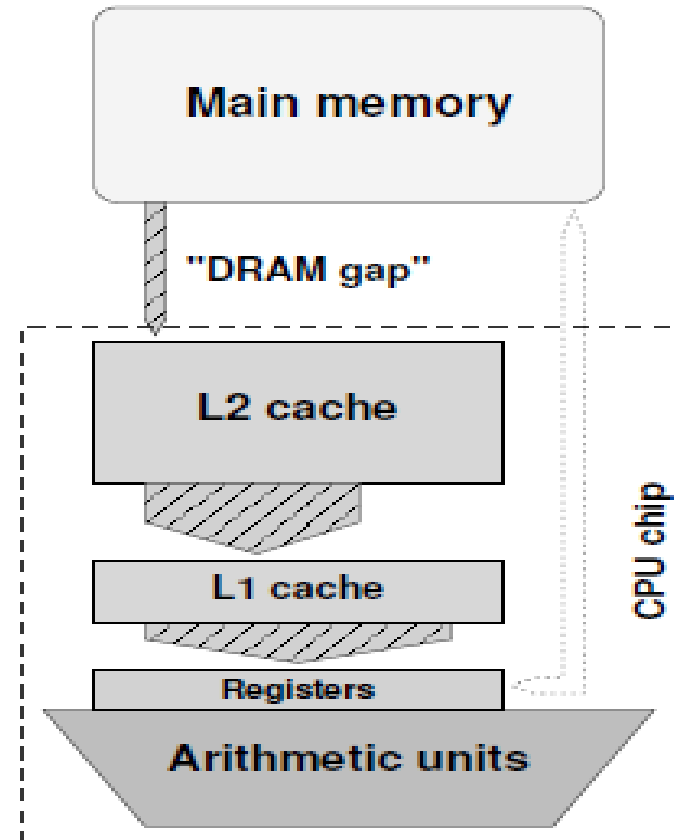


Von Neuman machines in the Textbook (2/2)

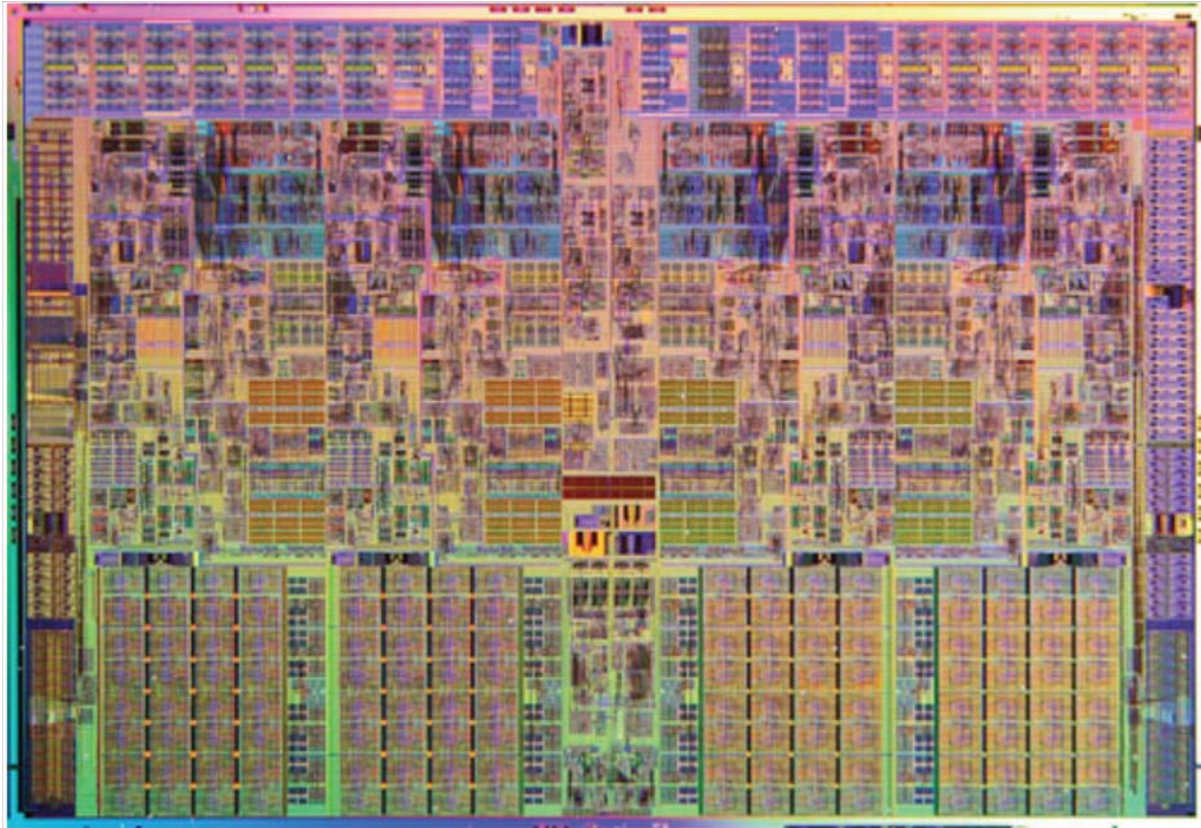
Larger but slower

- ❖ Tape
- ❖ Disk
- ❖ Main memory (DRAM)
- ❖ Level 2 cache
- ❖ Level 1 cache
- ❖ Registers

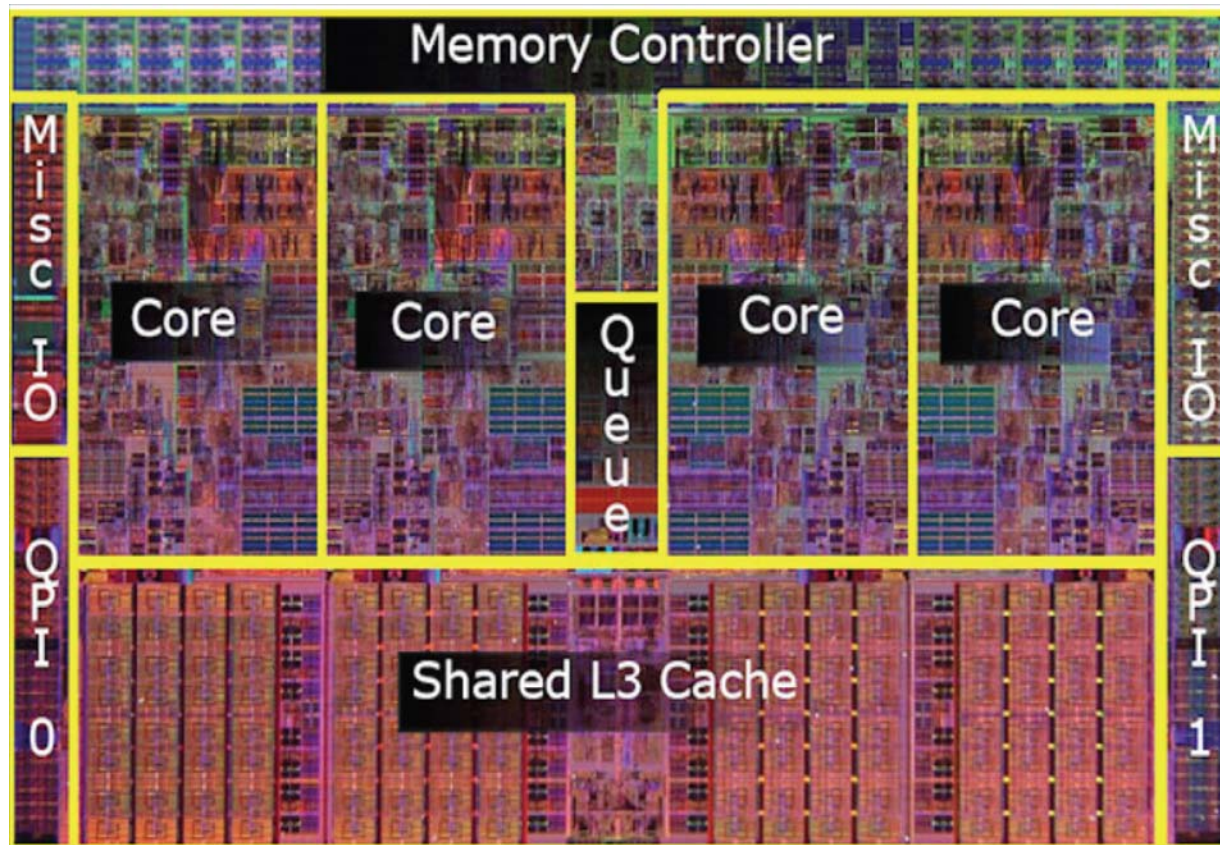
Smaller but faster



Example: Intel Core i7 (1/2)



Example: Intel Core i7 (2/2)



Performance

Floating Point operations per second (Flops/second)

Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^9
megaFLOPS	10^6
kiloFLOPS	10^3

Memory Bandwidth (Bytes/sec; gigaBytes/sec, megaBytes/sec)



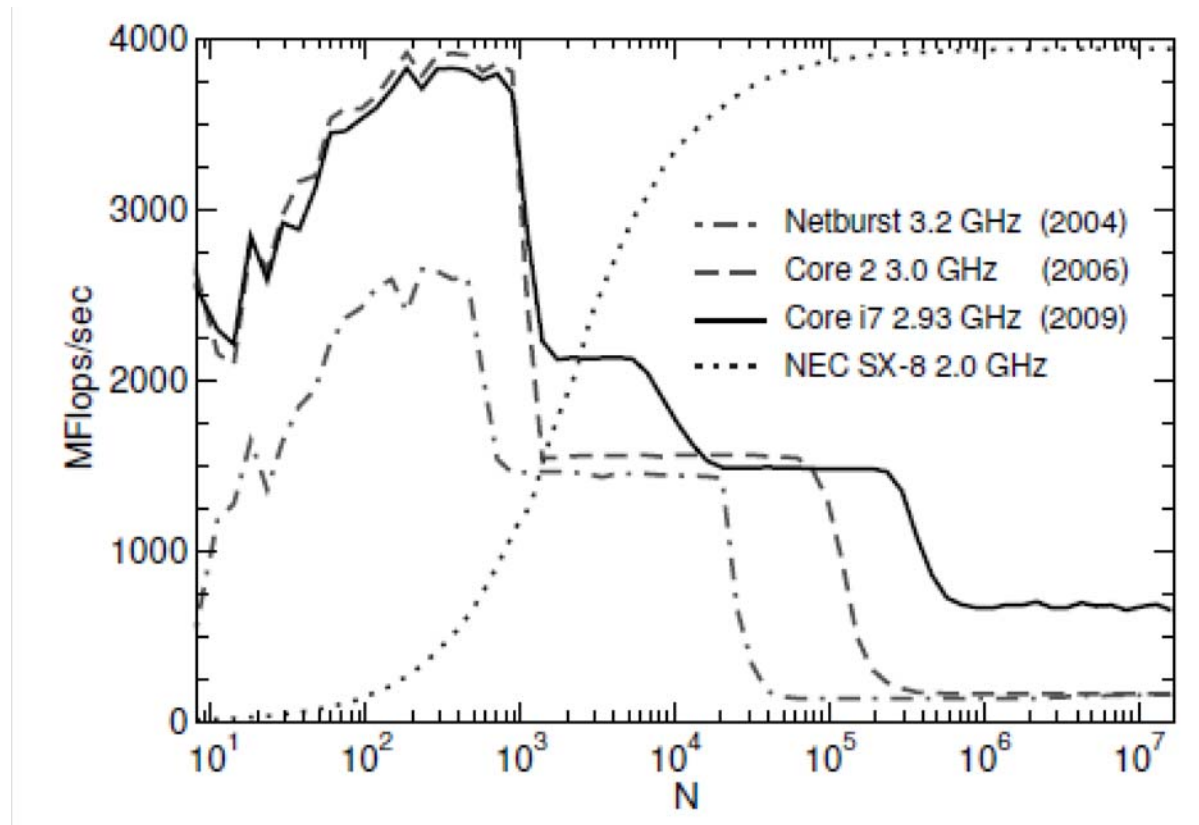
Measuring Performance (1/2)

```
1 double precision, dimension(N) :: A,B,C,D
2 double precision :: S,E,MFLOPS
3
4 do i=1,N                               !initialize arrays
5   A(i) = 0.d0; B(i) = 1.d0
6   C(i) = 2.d0; D(i) = 3.d0
7 enddo
8
9 call get_walltime(S)                   ! get time stamp
10 do j=1,R
11   do i=1,N
12     A(i) = B(i) + C(i) * D(i) ! 3 loads, 1 store
13   enddo
14   if(A(2).lt.0) call dummy(A,B,C,D) ! prevent loop interchange
15 enddo
16 call get_walltime(E)                   ! get time stamp
17 MFLOPS = R*N*2.d0/((E-S)*1.d6) ! compute MFlop/sec rate
```

```
1 #include <sys/time.h>
2
3 void get_walltime_(double* wcTime) {
4   struct timeval tp;
5   gettimeofday(&tp, NULL);
6   *wcTime = (double)(tp.tv_sec + tp.tv_usec/1000000.0);
7 }
8
9 void get_walltime(double* wcTime) {
10  get_walltime_(wcTime);
11 }
```



Measuring Performance (2/2)



Moore's Law (1/4)

From Wikipedia:

The law is named after Intel co-founder Gordon E. Moore, who described the trend in his 1965 paper. The paper noted that the number of components in integrated circuits had doubled every year from the invention of the integrated circuit in 1958 until 1965 and predicted that the trend would continue "for at least ten years".

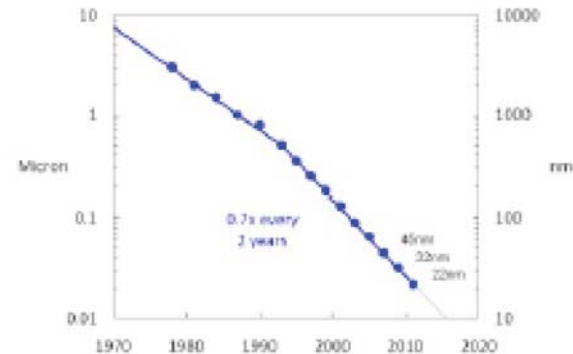
The law does not say that microprocessor performance or clock speed doubles every two years

Nevertheless, clock speed did in fact double every two years from roughly 1975 to 2005, but has now flattened at about 3 GHz due to limitations on power dissipation.



Moore's Law (3/4) - Implications

- ❖ Smaller circuits are more efficient, so one can either
 - maintain same clock speed but use less power
 - maintain same power but increase clock speed (historical trend)
 - maintain same power and clock speed but increase functionality (current trend)
- ❖ Limit on power (heat) dissipation has halted further increase in clock speeds



Moore's Law (4/4) - Performance Enhancements

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- ❖ Pipelined functional units
- ❖ Superscalar architecture (multiple instructions per cycle)
- ❖ Out-of-order execution of instructions
- ❖ SIMD instructions (multiple sets of operands per instruction)
- ❖ Memory hierarchy (larger caches and more levels of cache)
- ❖ Multicore and multithreaded processors

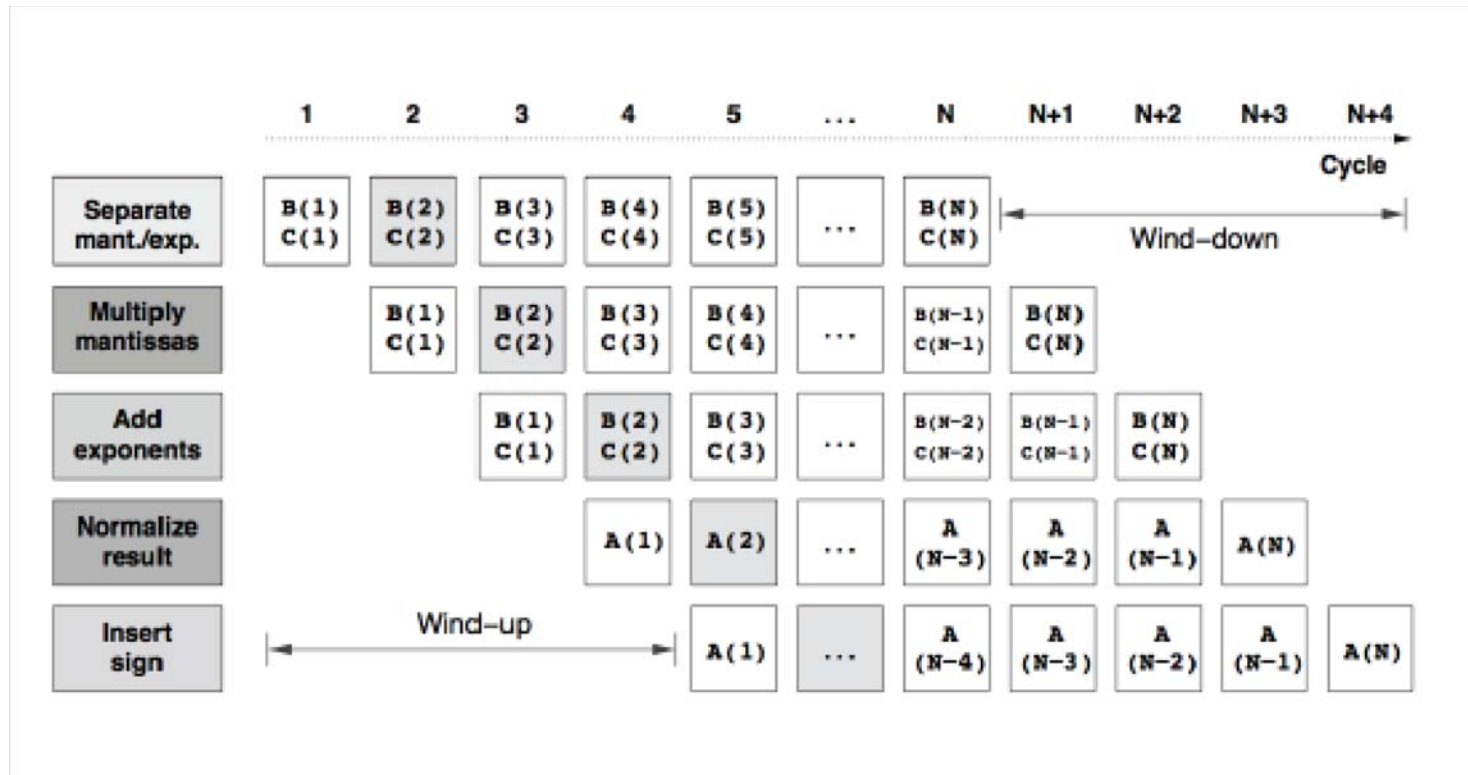


Pipelining (1/3)

- ✦ Analogous to manufacturing assembly line: each station performs same task on each object, with different objects at each station simultaneously
- ✦ Simple instruction pipeline: fetch → decode → execute
 - first instruction is fetched
 - second instruction is fetched while first instruction is decoded
 - third instruction is fetched while second is decoded and first is executed, etc.
- ✦ Complex instructions may have many more stages, with correspondingly deeper pipeline

Pipelining (2/3)

Example: $A(:,i) = B(:,i) * C(:,i)$

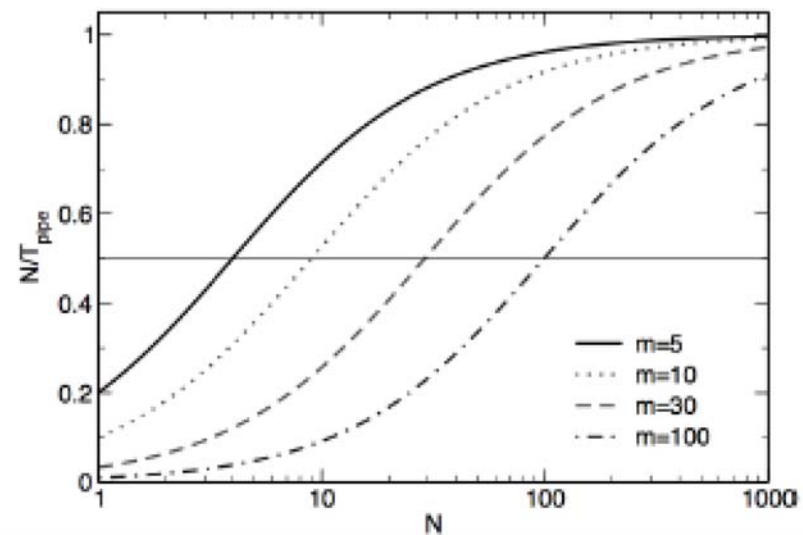


Pipelining (2/3)

Speedup from pipelined processing of N objects in m stages

$$\frac{T_{seq}}{T_{pipe}} = \frac{mN}{N + m - 1}$$

approaches m for large N



Software pilelining example

```
do i=1,N  
  A(i) = s * A(i) →  
enddo
```

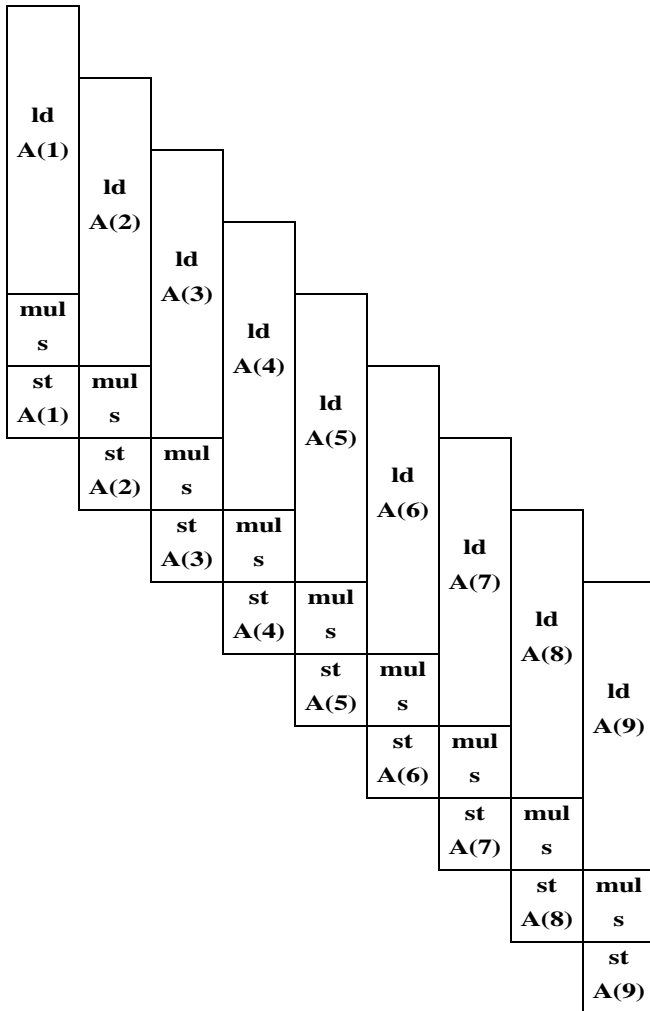
translation to lower level language

```
loop:  load A(i)  
       mult A(i) = A(i) * s  
       store A(i)  
       i = i + 1  
       branch -> loop
```

software pipelining

```
loop:  load A(i+6)  
       mult A(i+2) = A(i+2) * s  
       store A(i)  
       i = i + 1  
       branch -> loop
```

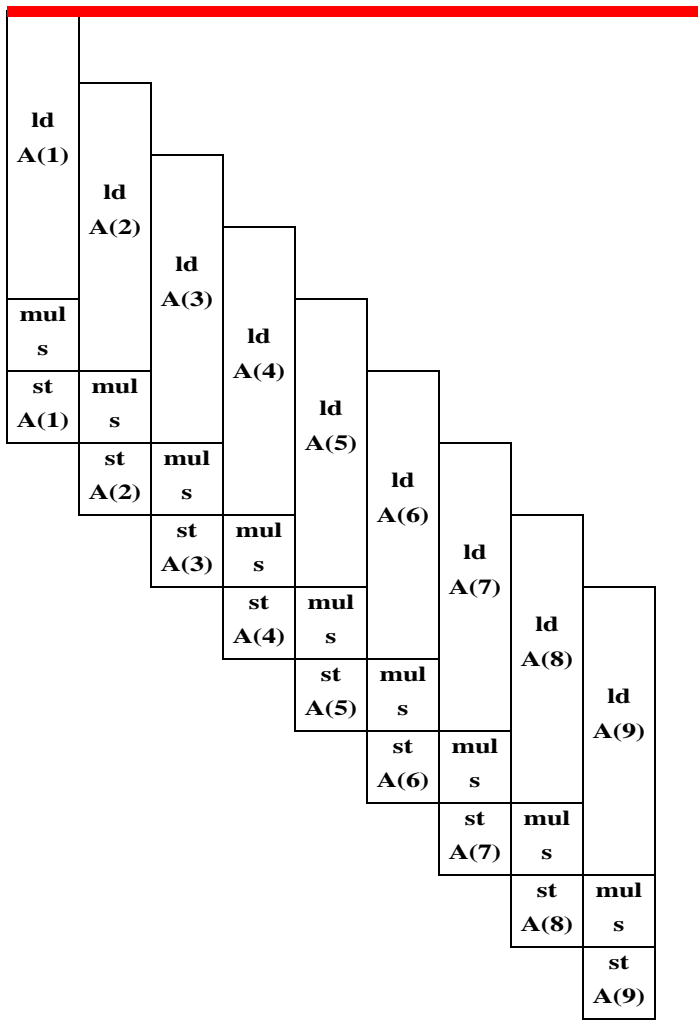




Assume:

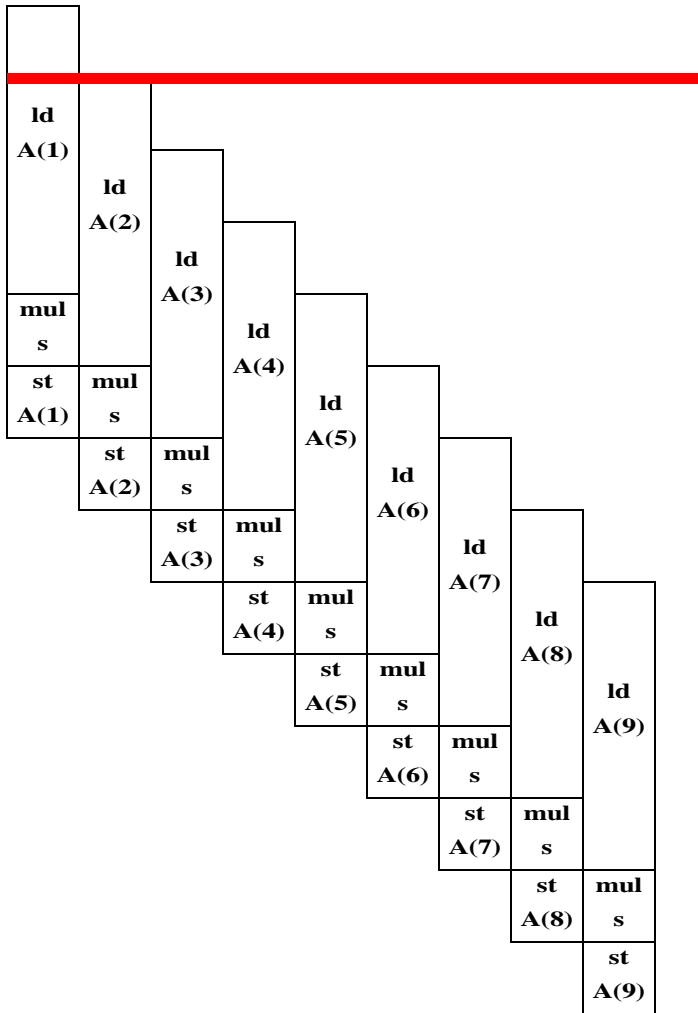
- **Load pipeline 4 stages**
- **Multiply and store a single step each**

Loads start at each step to keep pipeline busy.

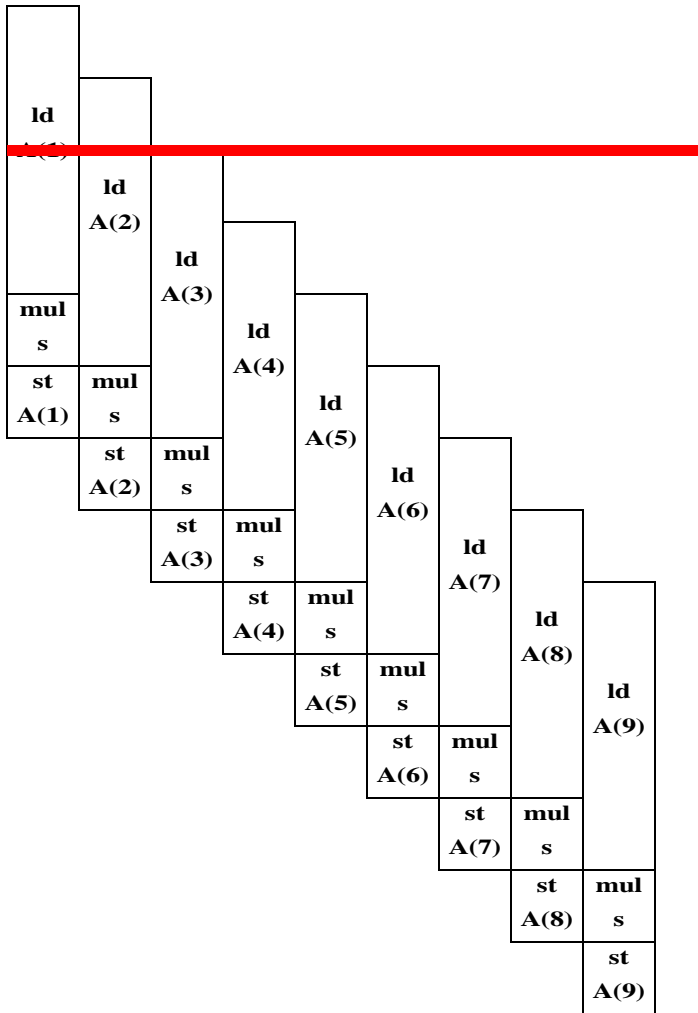


ld A(1)

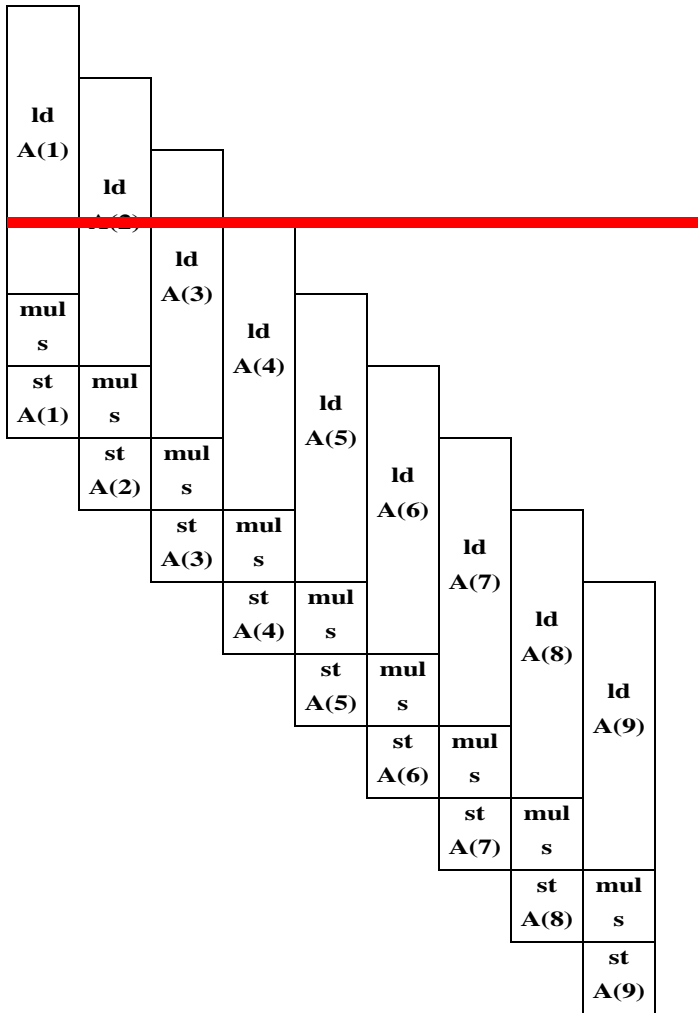




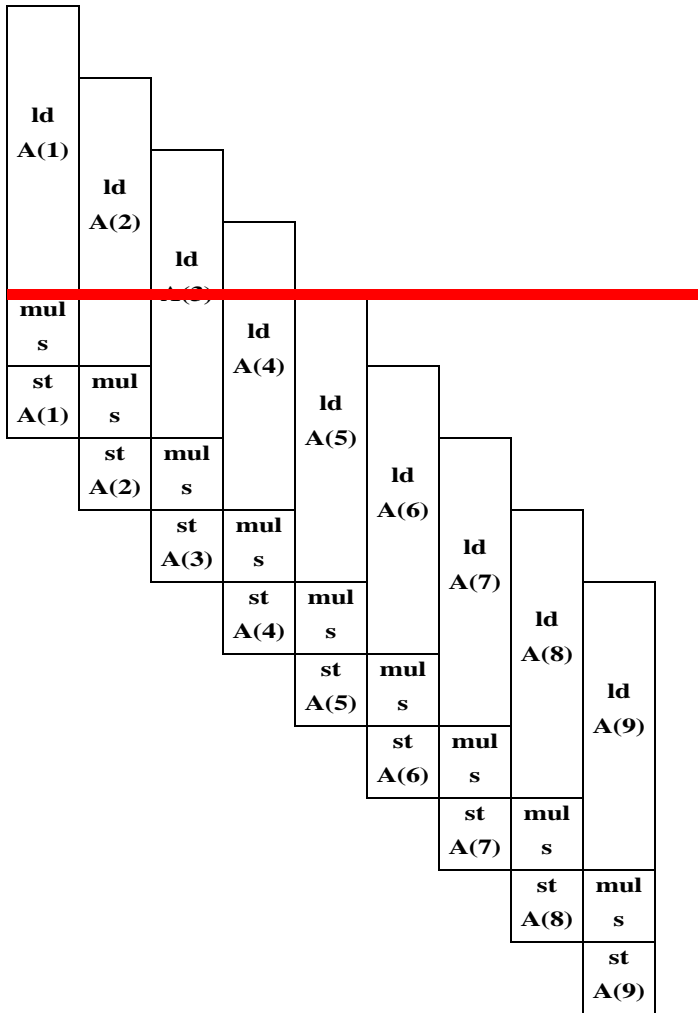
ld A(1)
ld A(2)



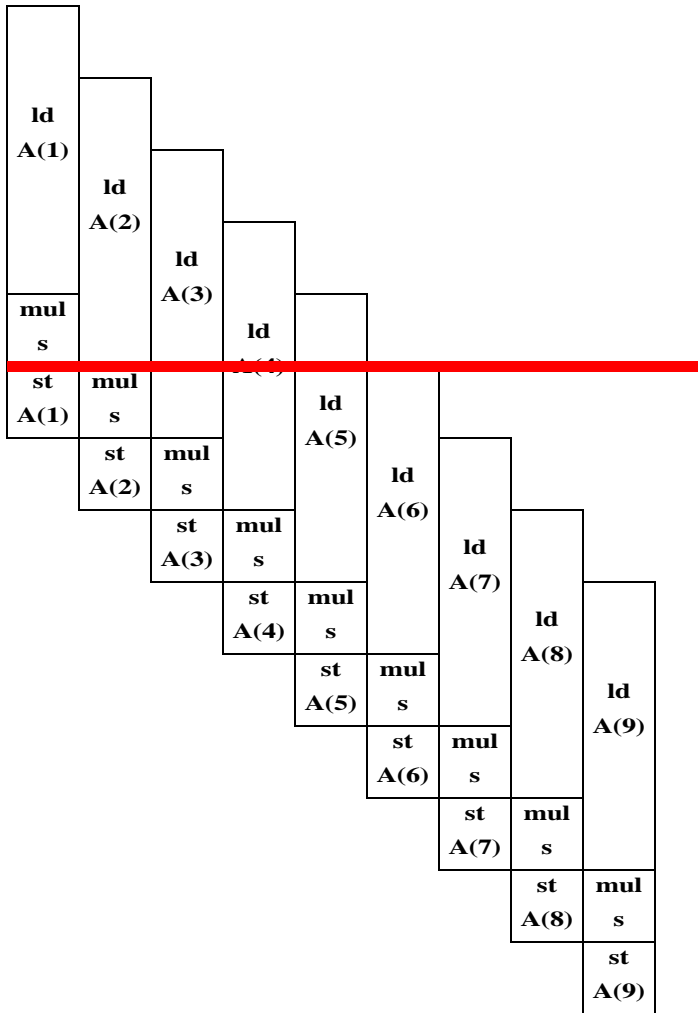
ld A(1)
ld A(2)
ld A(3)



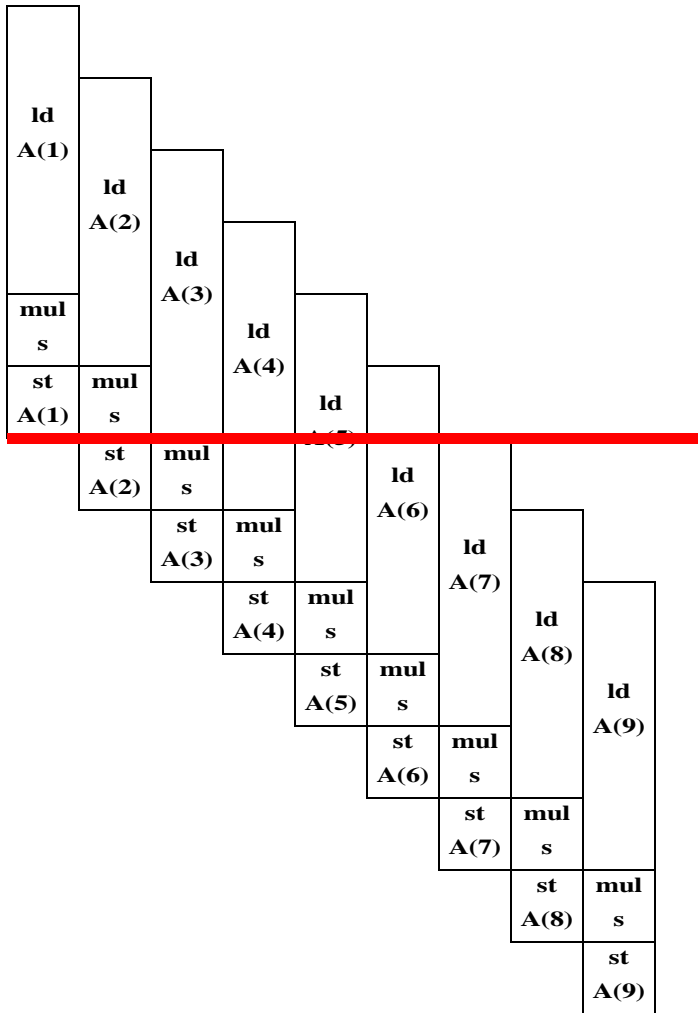
ld A(1)
 ld A(2)
 ld A(3)
 ld A(4)



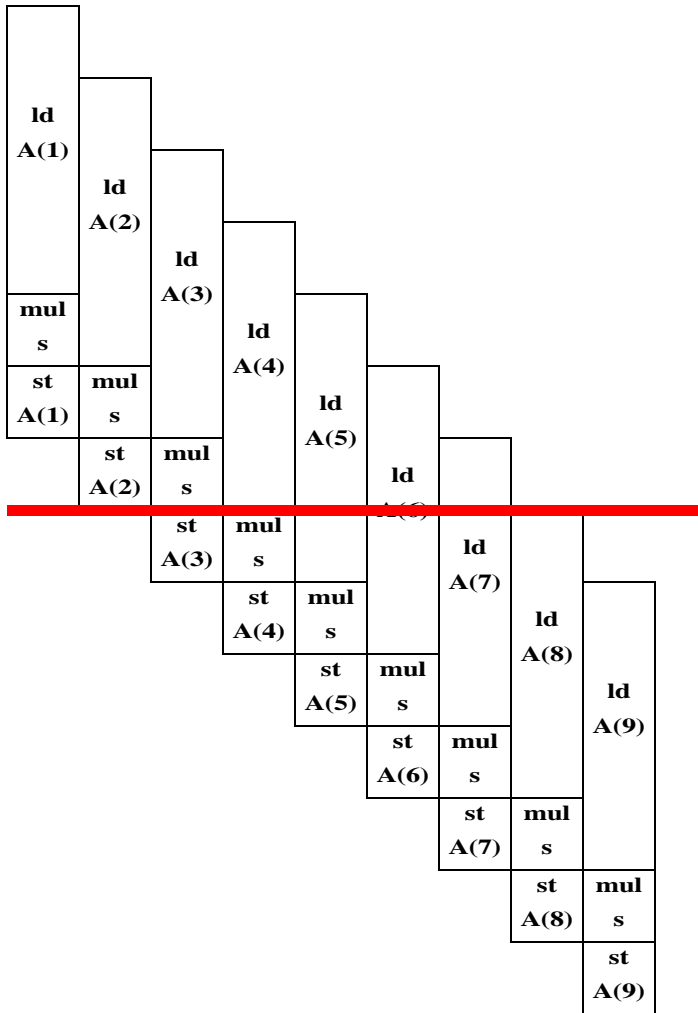
ld A(1)
 ld A(2)
 ld A(3)
 ld A(4)
 mul s
 ld A(5)



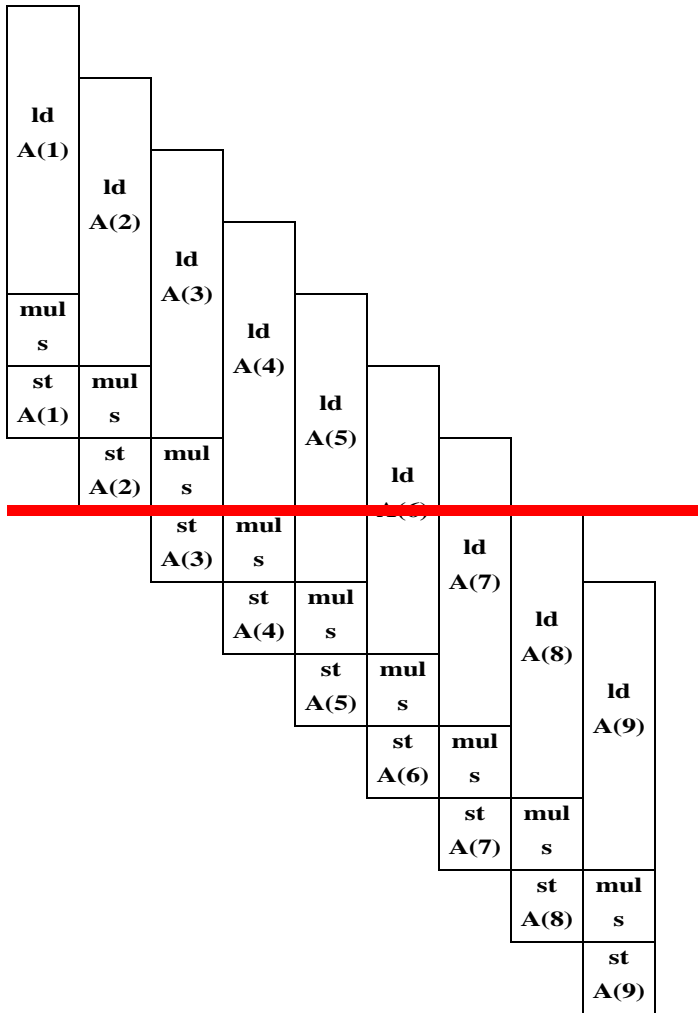
ld A(1)
 ld A(2)
 ld A(3)
 ld A(4)
 mul s
 ld A(5)
 st A(1)
 mul s
 ld A(6)



ld A(1)
 ld A(2)
 ld A(3)
 ld A(4)
 mul s
 ld A(5)
 st A(1)
 mul s
 ld A(6)
 st A(2)
 mul s
 ld A(7)



ld A(1)
 ld A(2)
 ld A(3)
 ld A(4)
 mul s
 ld A(5)
 st A(1)
 mul s
 ld A(6)
 st A(2)
 mul s
 ld A(7)
 st A(3)
 mul s
 ld A(8)



ld A(1)
 ld A(2)
 ld A(3)
 ld A(4)
 mul s
 ld A(5)
 st A(1)
 mul s
 ld A(6)

st A(2)
 mul s
 ld A(7)
 st A(3)
 mul s
 ld A(8)

pattern repeats:
 st A(i)
 mul s (byA(i+1))
 ld A(i+5)

Superscalarity

- **Multiple instructions can be fetched and decoded concurrently (3–6 nowadays).**
- **Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units (2–6). This is closely related to the previous point, because feeding those units requires code execution.**
- **Multiple floating-point pipelines can run in parallel. Often there are one or two combined multiply-add pipes that perform $a=b+c*d$ with a throughput of one each.**
- **Caches are fast enough to sustain more than one load or store operation per cycle, and the number of available execution units for loads and stores reflects that (2–4).**



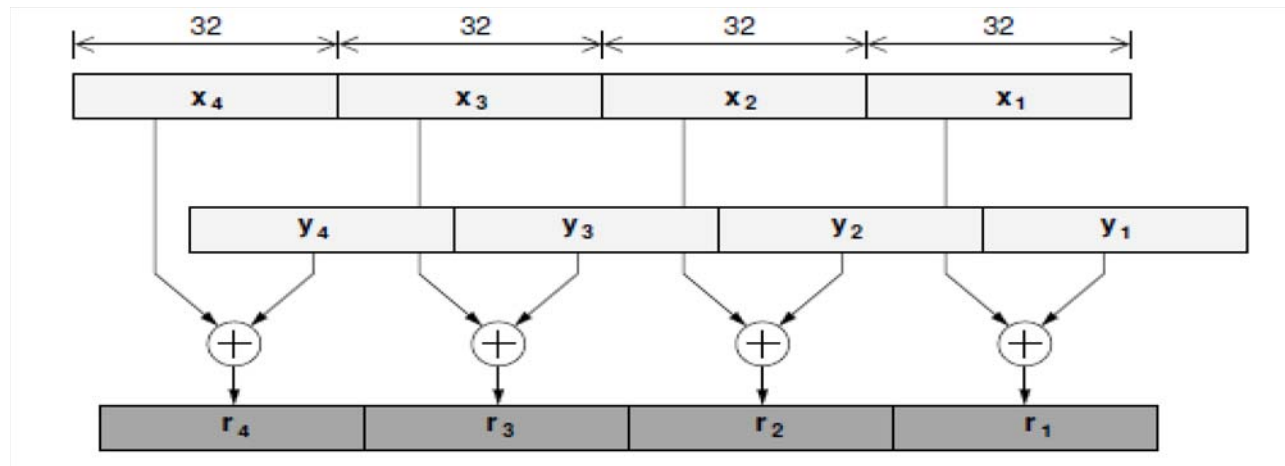
SIMD

It means: SINGLE Instruction/Multiple Data (see below)

The main idea is that there is a single control unit with the ability to control multiple identical operations with each instruction.

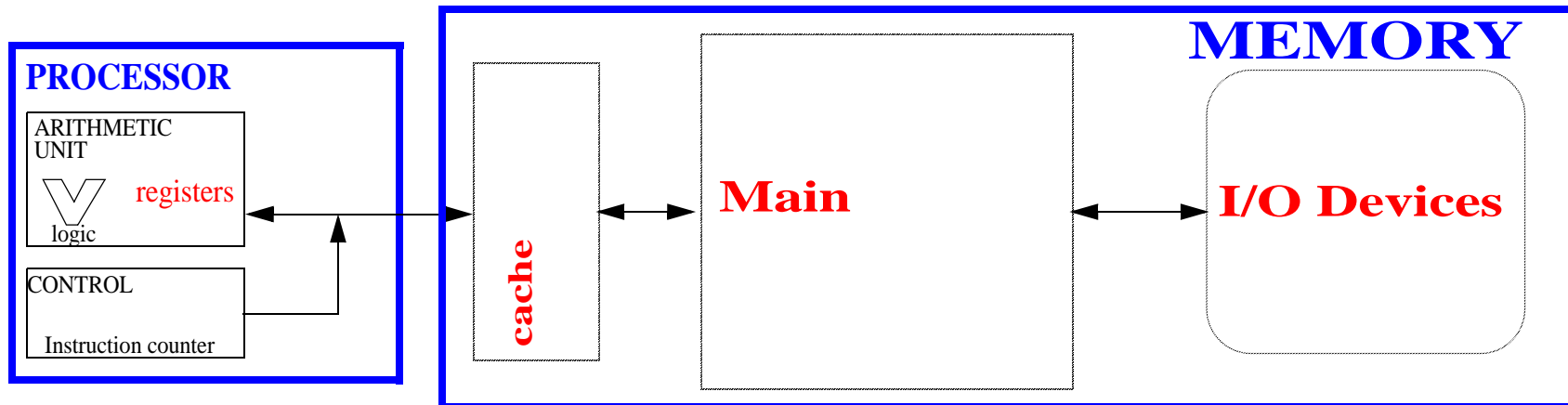
The book does not mention it, but the most famous and influential SIMD machine is Illiac IV (U of I ca. 1960s)

Today, it can be found in GPGPUs and microprocessor vector extensions (SSE[Intel], Altivec[IBM],...)



Memory Hierarchy and Cache Memories

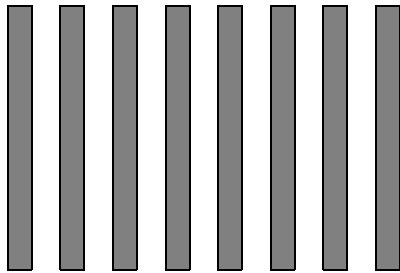
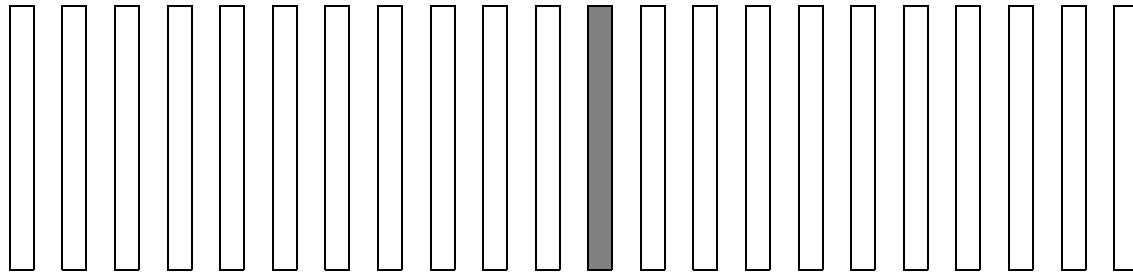
- **Programs tend to exhibit temporal and spatial locality:**
 - Temporal locality: Once programs access a data items or instruction, they tend to access them again in the near term.**
 - Spatial locality: Once programs access a data items or instruction, they tend to access nearby data items or instruction in the near term.**
- **Memory is organized in a hierarchy.**
 - As we move farther away from the CPU, memory components become (1) slower, (2) larger, and (3) less expensive per bit .**



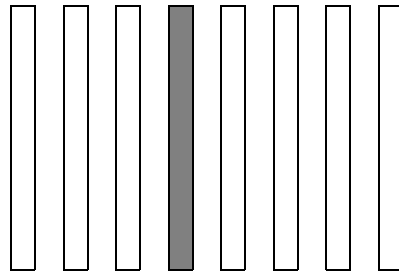
- **Cache is the first level of the memory hierarchy outside the CPU.**
- **Hit and Miss ratios: When the data is found in the cache, we have a cache hit, otherwise it is a miss. Hence Miss Ratio (M_R) and Hit Ratio (H_R)(= $1-M_R$) are the fraction of references that hit or miss respectively.**
- **Average access time: If all the data fits in main memory:
average memory access time =
 M_R * main memory access time + H_R * cache access time**
- **In the textbook**
 1. **hit ratio (H_R) is called cache reuse ratio (β)**
 2. **main memory access time is T_m**
 3. **cache access time is $T_c = T_m / \tau$**

- **Cache line: When there is a cache miss, a fixed size block of consecutive data elements, or line, is copied from main memory to the cache. Typical cache line size is 4-128 bytes.**
- **Main memory can be seen as a sequence of lines, some of which can have a copy in the cache.**
- **Placement of lines in the cache: Caches are divided into sets of lines. Cache lines from memory can be mapped to one and only one set. The set is usually chosen by bit selection: line address MOD number of sets in the cache.**
 - > **If there is a single set, the cache is said to be fully associative.**
 - > **If there are as many sets as lines in the cache, the cache is said to be direct mapped.**
 - > **Otherwise, if there are two or more sets, each containing exactly n elements, the cache is said to be n -way**

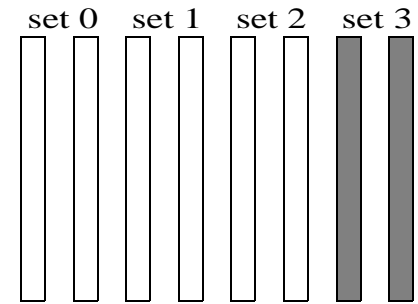




Fully



Direct mapped



Set associative

- **Replacement strategy: When a miss occurs and the cache is fully associative or n-way set associative, one of the cache lines must be replaced. Candidates for replacement are the lines in the set where the incoming line can be placed. In the case of direct mapped caches, there is only one candidate for replacement.**
 - > **Random: candidate line to be replaced is randomly selected.**
 - > **Least recently used (LRU): The candidate line that has been unused for the longest time is replaced.**
 - > **First in, First out(FIFO): The candidate line that has been in the cache for the longest time is replaced.**
- **Writes are processed in one of two ways:**
 - > **Write through: Data is written both to main memory and cache. Memory and cache are kept consistent.**
 - > **Write back: memory is updated only when the line is replaced.**

Programming for Locality

- **Program locality** was defined above as a tendency towards repetitive accesses. We say that program locality increases when this tendency becomes stronger by having repetitions happening on the average within shorter time spans.
- **Program locality can be controlled by the programmer to a certain extent by reorganizing computations.**
- **Consider the following loop where $f(i, j)$ is any function of i and j :**

```
for i=1:n
    for j=1:n
        A(j) = A(j) + f(i, j)
    end
end
```

The elements of A are re-referenced every n iteration of the inner loop. So, once an item has been referenced, the average “time” between repetitions is n iterations.



- **If the loop headers are interchanged:**


```

for j=1:n
  for i=1:n
    A(j)=A(j)+f(i,j)
  end
end

```

we obtain a semantically equivalent loop, but now the “time” between repetitions is reduced to one iteration.

- **we can say that the new loop has better locality.**
- **A similar transformation can be applied to straight line code by “clustering” uses and definitions of variables:**

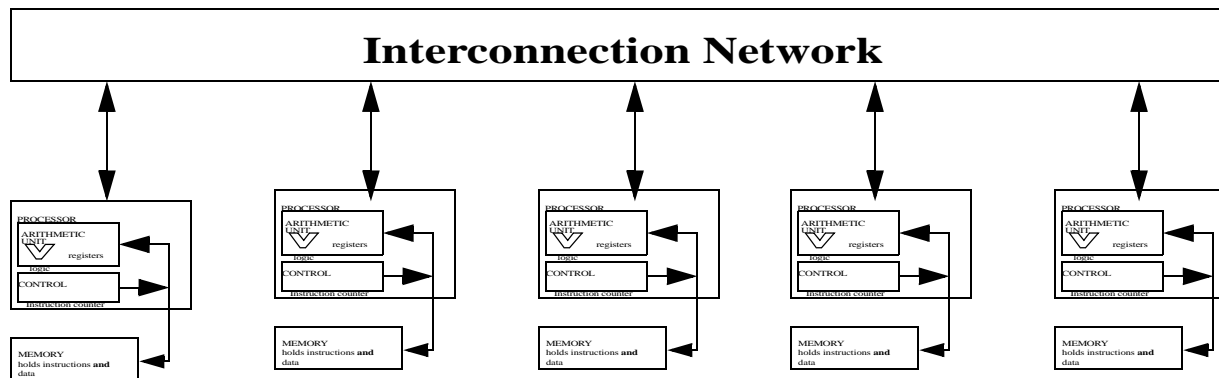
a=...		a=...
b=...		...=a
c=...		b=...
...		...=b
...=a		c=...
...=b		...

- **The great benefit of increasing locality is that it tends to reduce the time to access memory.**
- **In the case of registers, allocation is required.**
Typically done by compiler or, less often, manually by the programmer.
Programs with better locality tend to require fewer register spills and loads.
- **In the case of caches and paged main memory, the program will tend to have fewer misses and therefore fewer accesses to lower levels of the hierarchy.**



Multicomputers

- **A natural way to get parallelism given a collection of conventional computers is to connect them:**



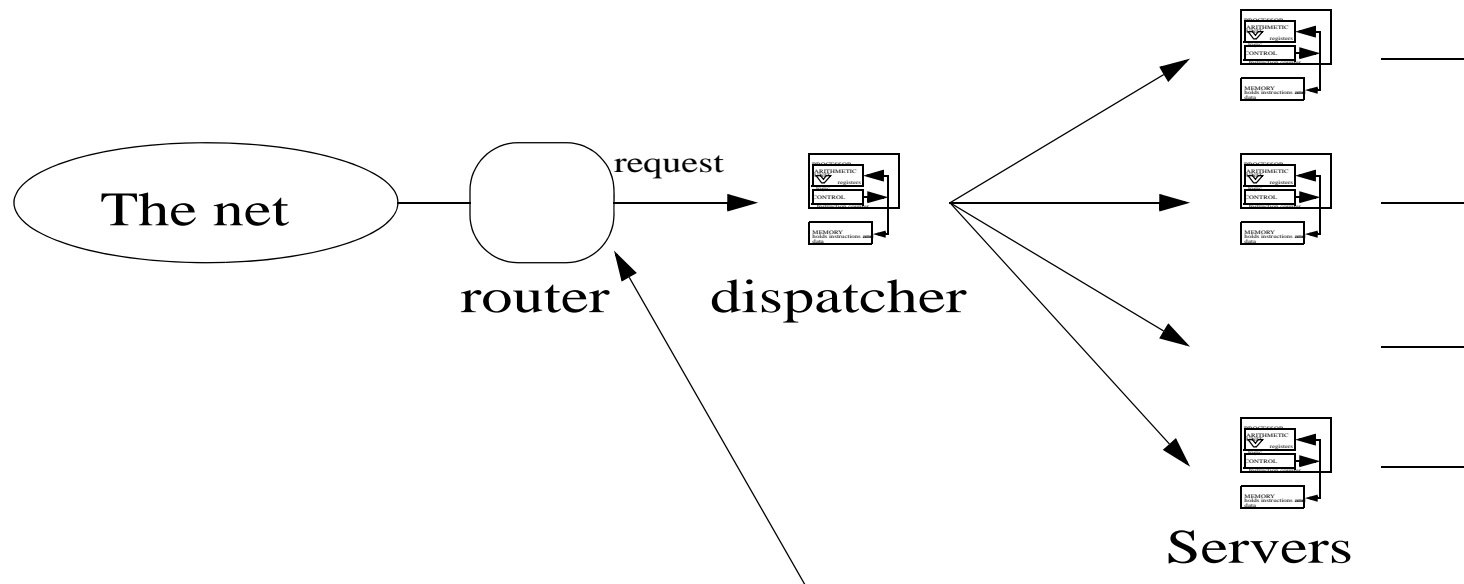
- **Each machine can proceed independently and communicate with the others via the interconnection network.**
- **There are two main classes of multicomputers: clusters and distributed-memory multiprocessors. They are quite similar, but the latter is designed as a single computer and its**

components are typically not sold separately while clusters are made out of off-the-shelf components.

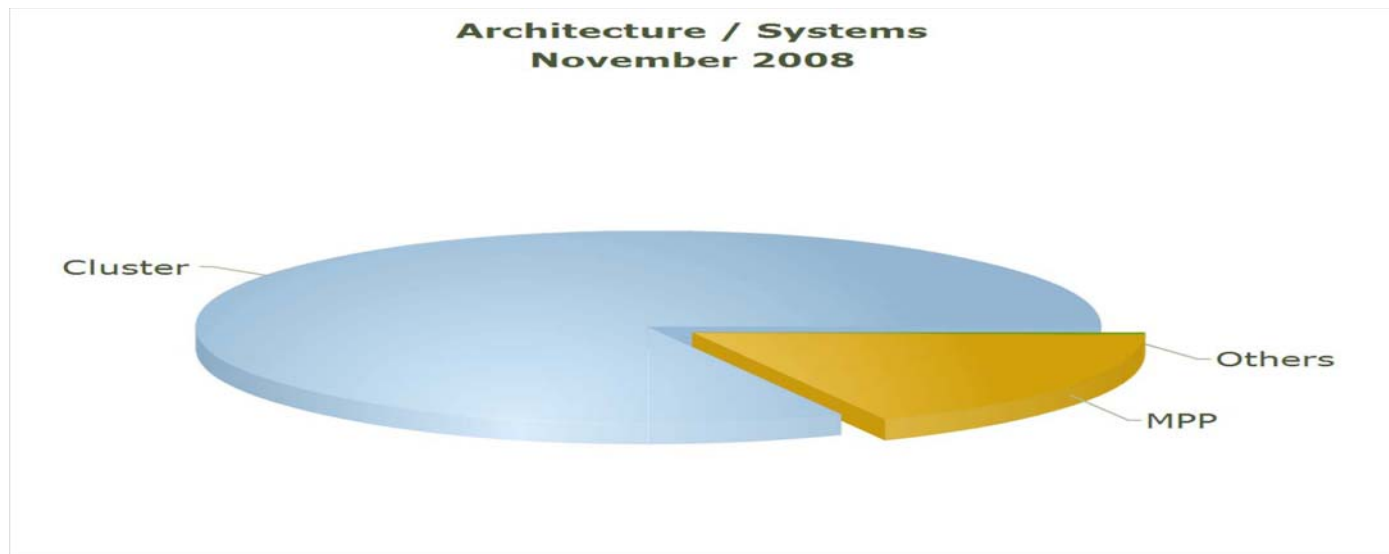
Furthermore, a cluster consists of a collection of interconnected whole computers (including I/O) used as a single, unified computing resource.

Not all nodes of a distributed memory multiprocessor need have complete I/O resources.

- An example of cluster is a web server**



- **Another example was a workstation cluster at Fermilab, which consisted of about 400 Silicon Graphics and IBM workstations. The system is used to analyze accelerator events. Analyzing any one of those events has nothing to do with analyzing any of the others. Each machine runs a sequential program that analyzes one event at a time. By using several machines it is possible to analyze many events simultaneously.**
- **Most important for parallel programming are high-end clusters that pervade the Top500**



Performance Issues in Multicomputers

- **Here we have our first parallel computing platform.**
- **Performance will depend on communication time.**
- **Typically, multicomputers communicate via messages. The time for a message to go from source to destination, the latency, is modeled by the formula where σ is the startup cost,**

$$\sigma + n\tau$$

τ is the transfer time per data item, and n is the number of data items to be sent.

-



- **The time for a message or communication latency can also be represented by the formula:**
sender overhead+time of flight+transmission time+receiver overhead
 - > **sender overhead is the time consumed by the processor to inject the message into the interconnection network.**
 - > **time of flight is the time for the first bit of the message to arrive at the receiver**
 - > **transmission time is the time for the rest of the message to arrive at the receiver after the first bit has arrived**
 - > **receiver overhead is the time for the processor to pull the message from the interconnection network.**
- **The τ of the above formula could contain components from all terms of this formula except for time of flight.**



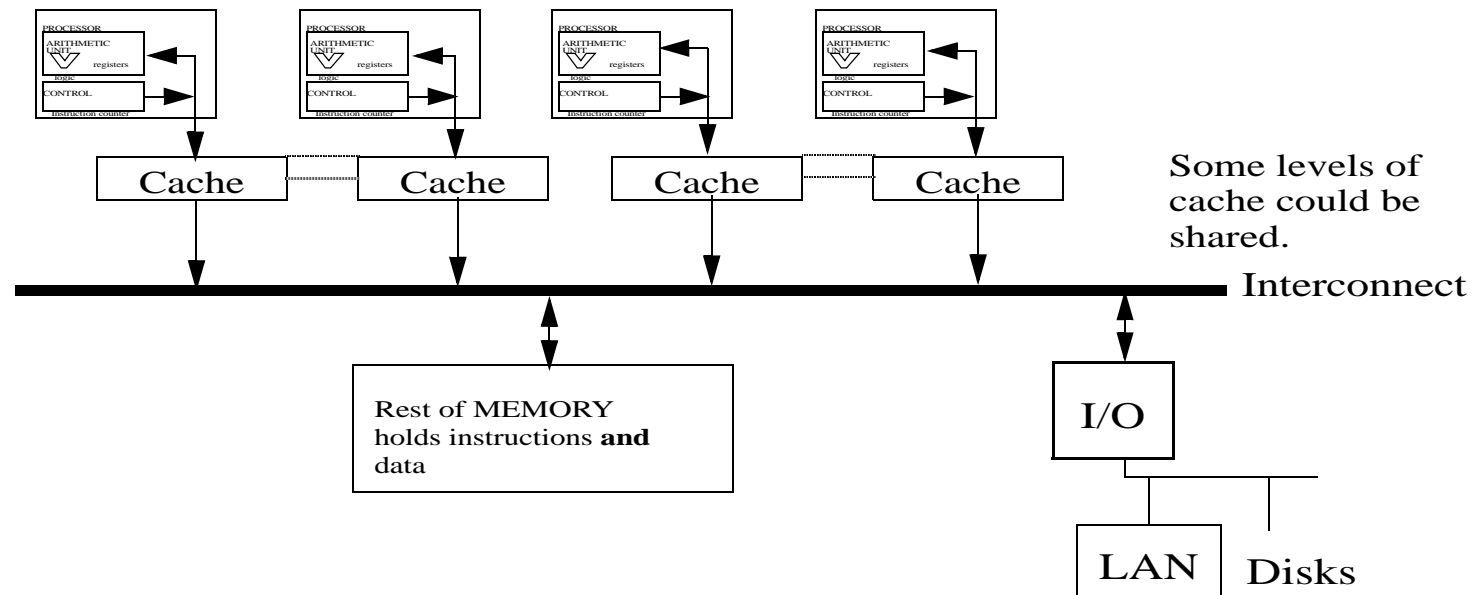
- **Communication costs per item transferred are reduced if the length of the message increases. As will be discussed later, the problem is not simply to increase the size of messages since the program may attain such increase at the expense of parallelism.**
- **Communication costs can also be reduced if communication is overlapped with computation or other communication. This is called latency hiding. To benefit from latency hiding, the program must be organized so that data is send as early as possible and check for its availability as late as possible.**

Shared-Memory Multiprocessors

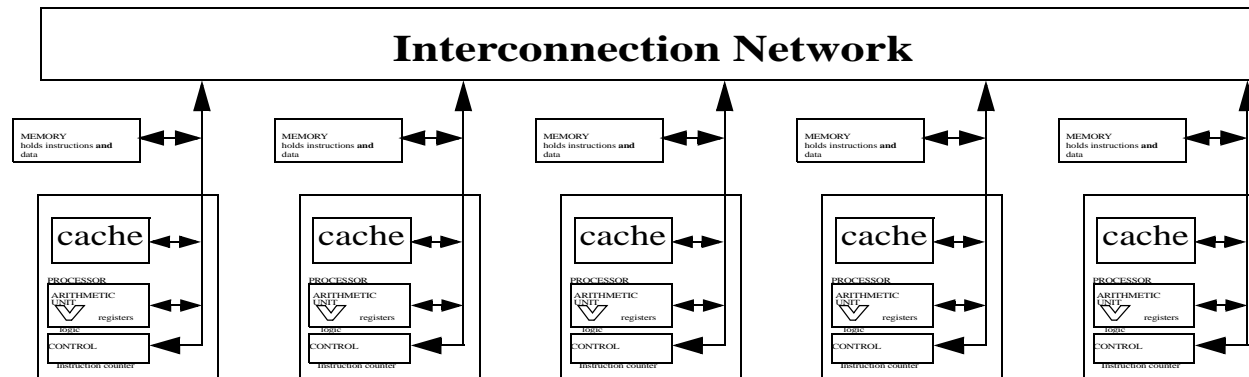
- **In shared-memory multiprocessors, there is an address space that can be accessed by all processors in the system. An address within that space represents the same location in all processors.**
- **The shared address space does not require a single, centralized memory module.**



- **The simplest form of a shared-memory multiprocessor is the symmetric multiprocessor (SMP). By symmetric we mean that each of the processors has exactly the same abilities. Therefore any processor can do anything: they all have equal access to every location in memory; they all can control every I/O device equally well, etc. In effect, from the point of view of each processor the rest of the machine looks the same, hence the term symmetric**



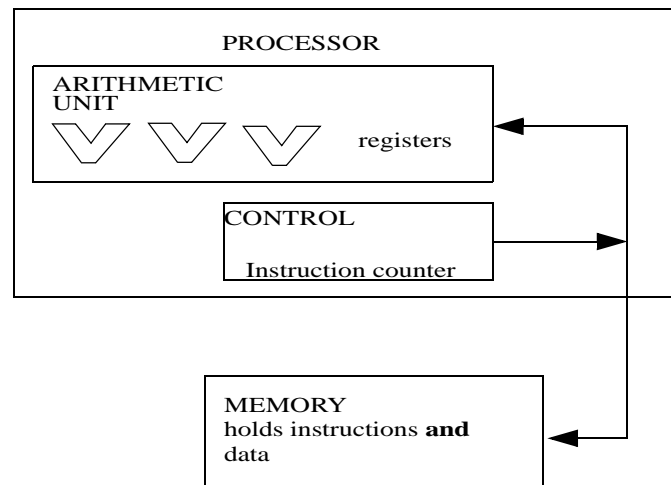
- **An alternative design are distributed shared-memory. These are also called NUMA (nonuniform memory accesses) machines. These designs reduce the cost of access to local memory and are a cost-effective way of scaling the memory bandwidth if most of the accesses are to local memory.**



- **In shared-memory multiprocessors communication and synchronization is typically implemented exclusively by write and read operations on the shared address space.**

Other Forms of Parallelism

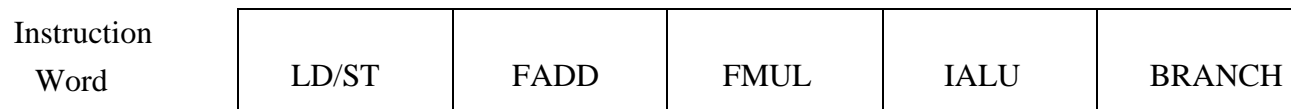
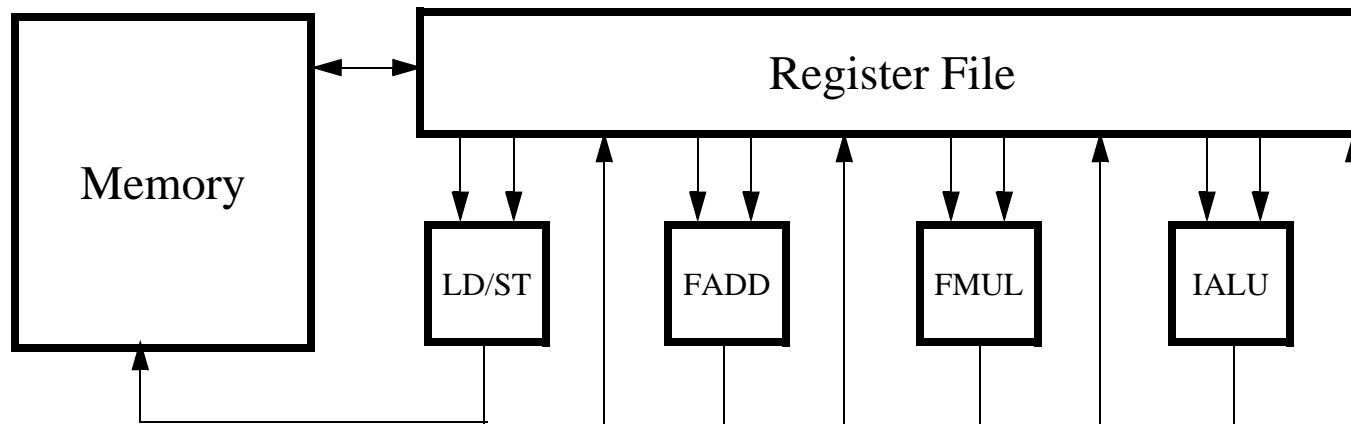
- **As discussed above, there are other forms of parallelism that are widely used today. These usually coexist with the coarse grain parallelism of multicomputers and multiprocessors.**
- **Pipelining of the control unit and/or arithmetic unit.**
- **Multiple functional units**



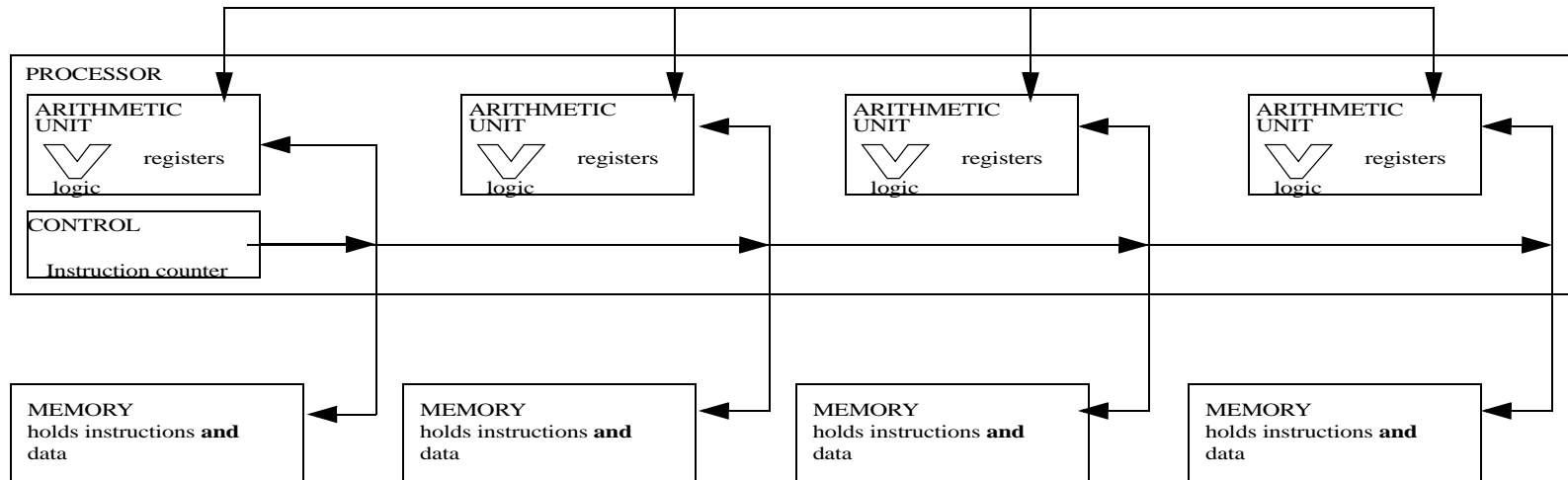
- **Most microprocessors today take advantage of this type of parallelism.**

- VLIW (Very Long Instruction Word) processors are an important class of multifunctional processors. The idea is that each instruction may involve several operations that are performed simultaneously. This parallelism is usually exploited by the compiler and not accessible to the high-level language programmer. However, the programmer can control this type of parallelism in assembly language.**

Multifunction Processor (VLIW)



- **Array processors. Multiple arithmetic units**



- **Illiacc IV is the earliest example of this type of machine. Each processing element (containing an arithmetic unit) of the Illiac IV was connected to four others to form a two-dimensional array (torus).**
- **A modern example is the NVIDIA GPU.**

Flynn's Taxonomy

- **Michael Flynn published a paper in 1972 in which he picked two characteristics of computers and tried all four possible combinations. Two stuck in everybody's mind, and the others didn't:**
- **SISD: Single Instruction, Single Data. Conventional Von Neumann computers.**
- **MIMD: Multiple Instruction, Multiple Data. Multicomputers and multiprocessors.**
- **SIMD: Single Instruction, Multiple Data. Array processors.**
- **MISD: Multiple Instruction, Single Data. Not used and perhaps not meaningful.**

