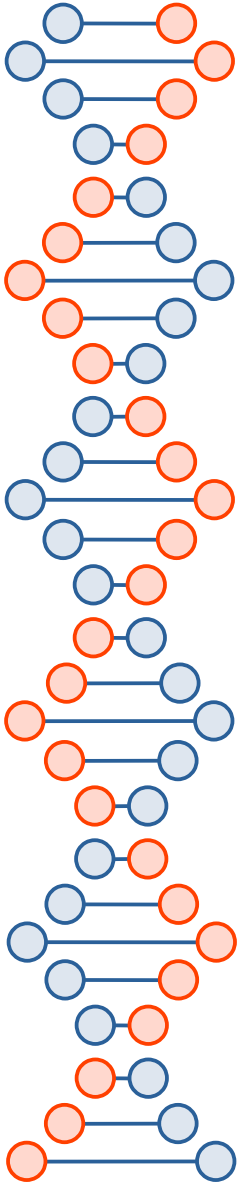
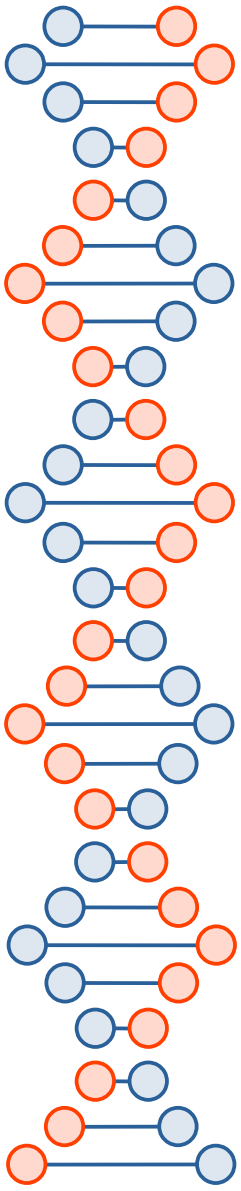


High Performance Computing

Ch. 1 Modern Processors

Computer Eng., KMITL
Assoc. Prof. Dr. Surin. K.



Modern Processors

- 1.1 Stored-program computer architecture . . . . . .
- 1.2 General-purpose cache-based microprocessor architecture
- 1.3 Memory hierarchies . . . . . .
- 1.4 Multicore processors . . . . . .
- 1.5 Multithreaded processors . . . . . .
- 1.6 Vector processors . . . . . .

1.1 Store-Program Computer Architecture

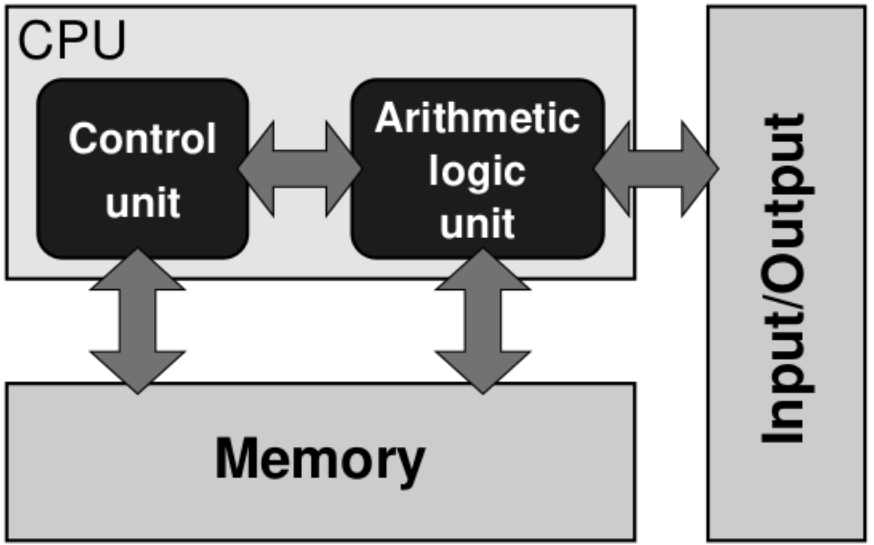
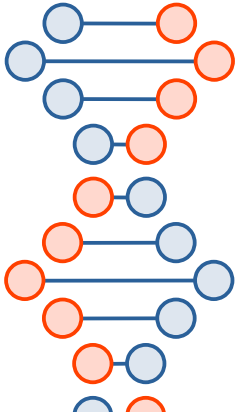
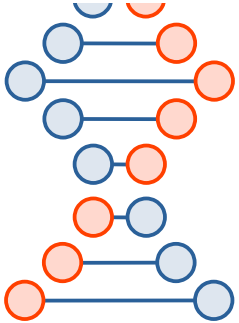


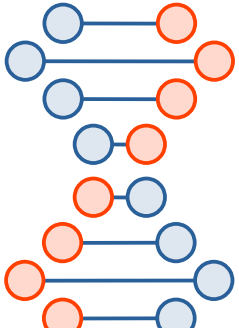
Figure 1.1: Stored-program computer architectural concept. The "program," which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.



1.1 Store-Program Computer Architecture

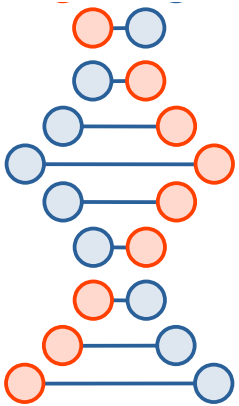
- Instructions and data must be continuously fed to the control and arithmetic units, so that the speed of the memory interface poses a limitation on compute performance. This is often called the *von Neumann bottleneck*. In the following sections and chapters we will show how architectural optimizations and programming techniques may mitigate the adverse effects of this constriction, but it should be clear that it remains a most severe limiting factor.



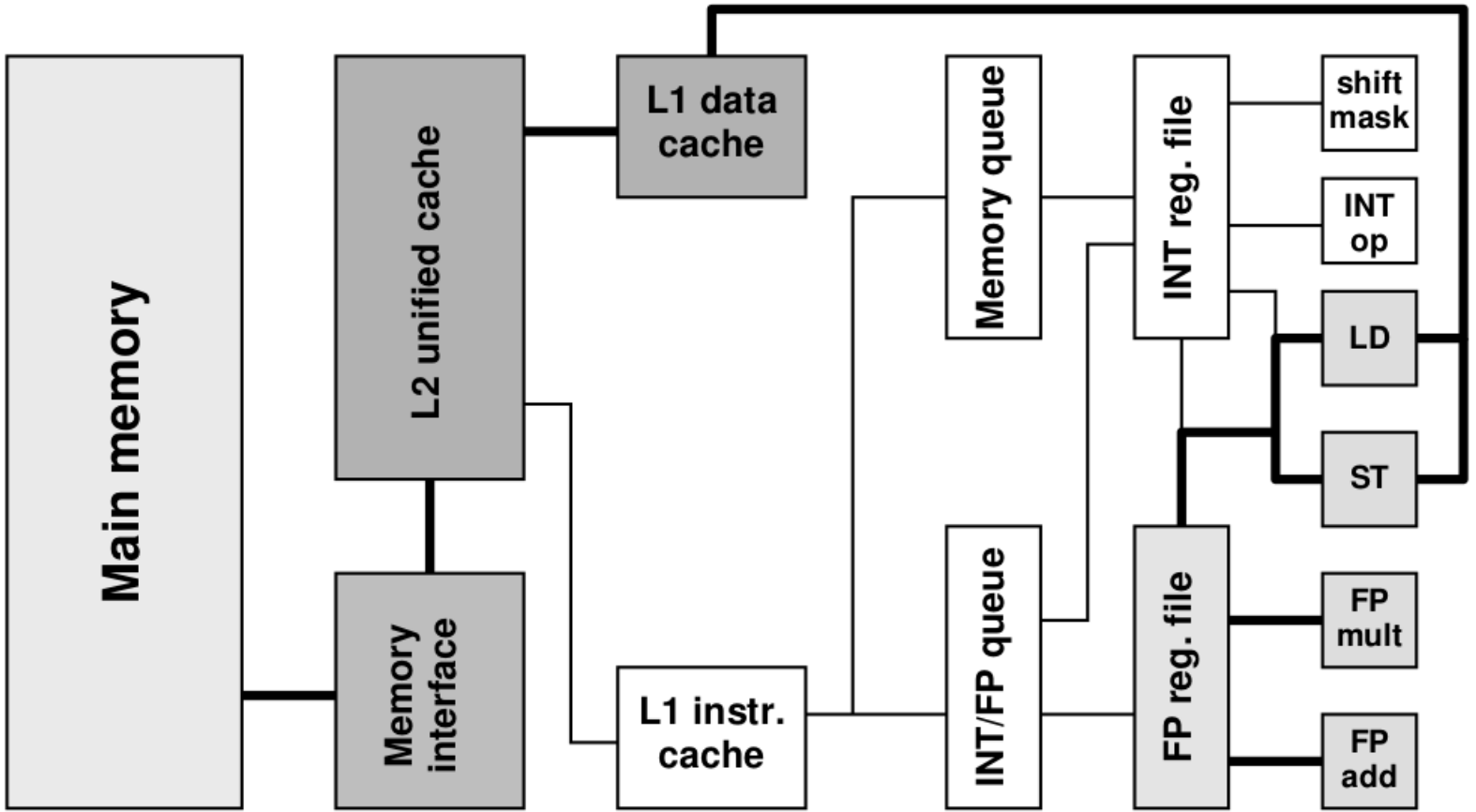


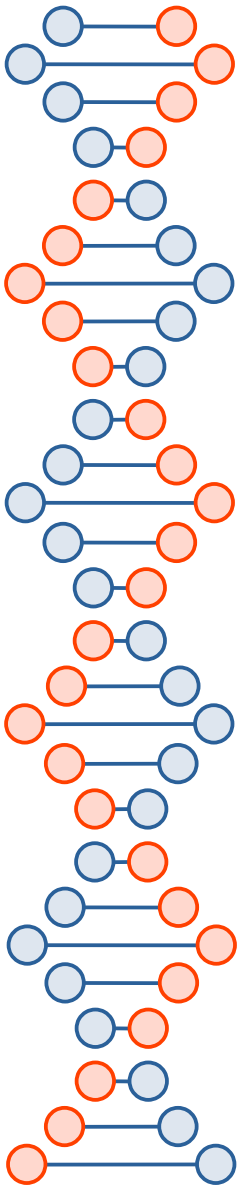
1.1 Store-Program Computer Architecture

- The architecture is inherently sequential, processing a single instruction with (possibly) a single operand or a group of operands from memory. The term *SISD* (Single Instruction Single Data) has been coined for this concept. How it can be modified and extended to support parallelism in many different flavors and how such a parallel machine can be efficiently used is also one of the main topics of this book.



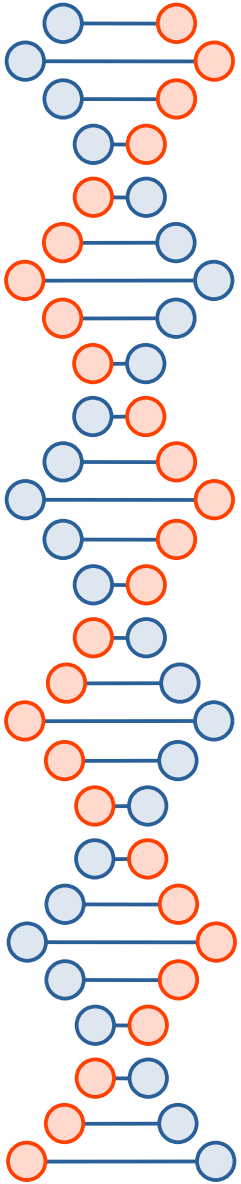
- 1.2 General-purpose cache-based microprocessor architecture





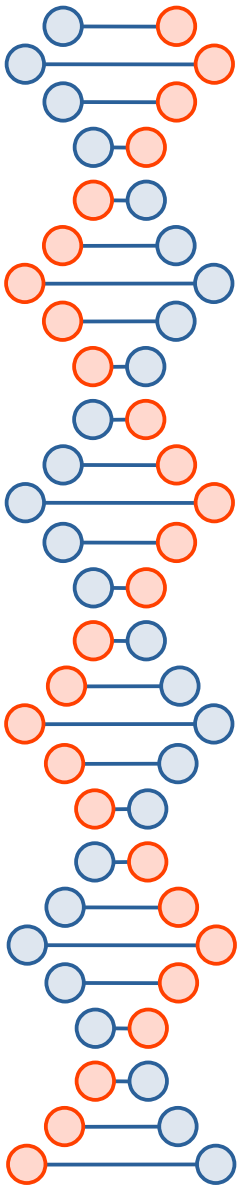
1.2 General-purpose cache-based microprocessor architecture

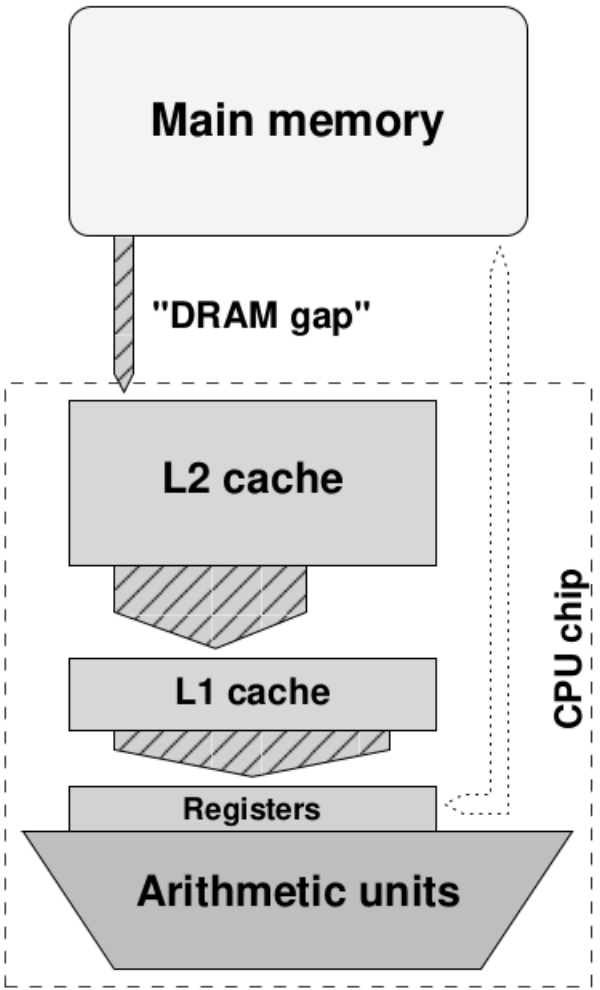
- Typical CPUs nowadays have between 16 and 128 user-visible registers of both kinds (Integer & Floating Point).
- Load (LD) and store (ST) units handle instructions that transfer data to and from registers.
- Instructions are sorted into several queues, waiting to be executed, probably not in the order they were issued (see below).
- Finally, caches hold data and instructions to be (re-)used soon.
- The major part of the chip area is usually occupied by caches.

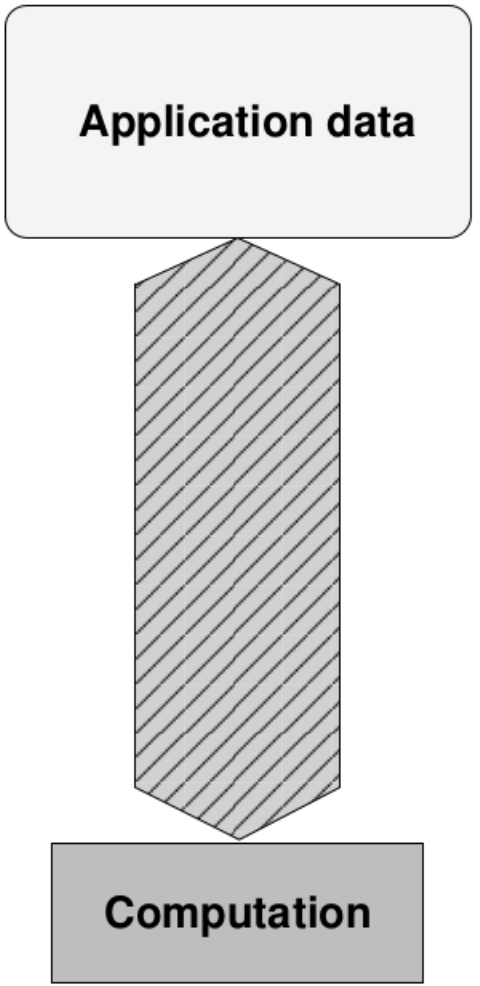


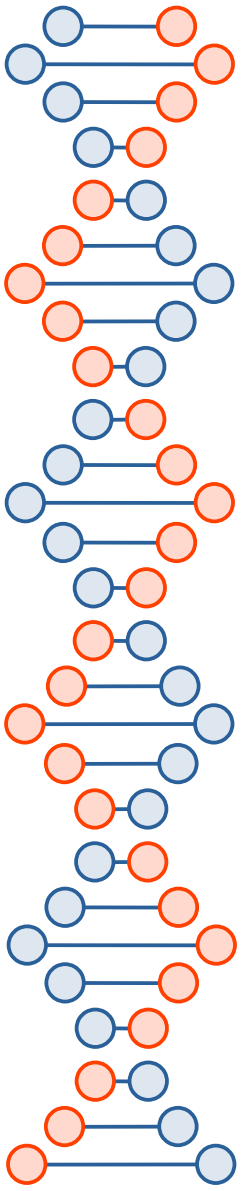
1.2 General-purpose cache-based microprocessor architecture

- Scientific computing tends to be quite centric to floating-point data, usually with “double precision.
- The performance at which the FP units generate results for multiply and add operations is measured in floating-point operations per second (Flops/sec).
- With typical clock frequencies between 2 and 3 GHz, this leads to a peak arithmetic performance between 4 and 12 GFlops/sec per core.




Main memory
"DRAM gap"
L2 cache
L1 cache
Registers
Arithmetic units
CPU chip


Application data
Computation



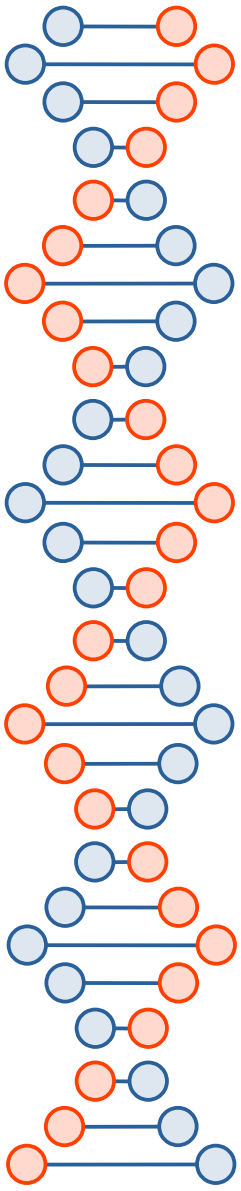
• 1.2 General-purpose cache-based microprocessor architecture

Listing 1.2: A C routine for measuring wallclock time, based on the `gettimeofday()` POSIX function. Under the Windows OS, the `GetSystemTimeAsFileTime()` routine can be used in a similar way.

```
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 }
```

Listing 1.1: Basic code fragment for the vector triad benchmark, including performance measurement.

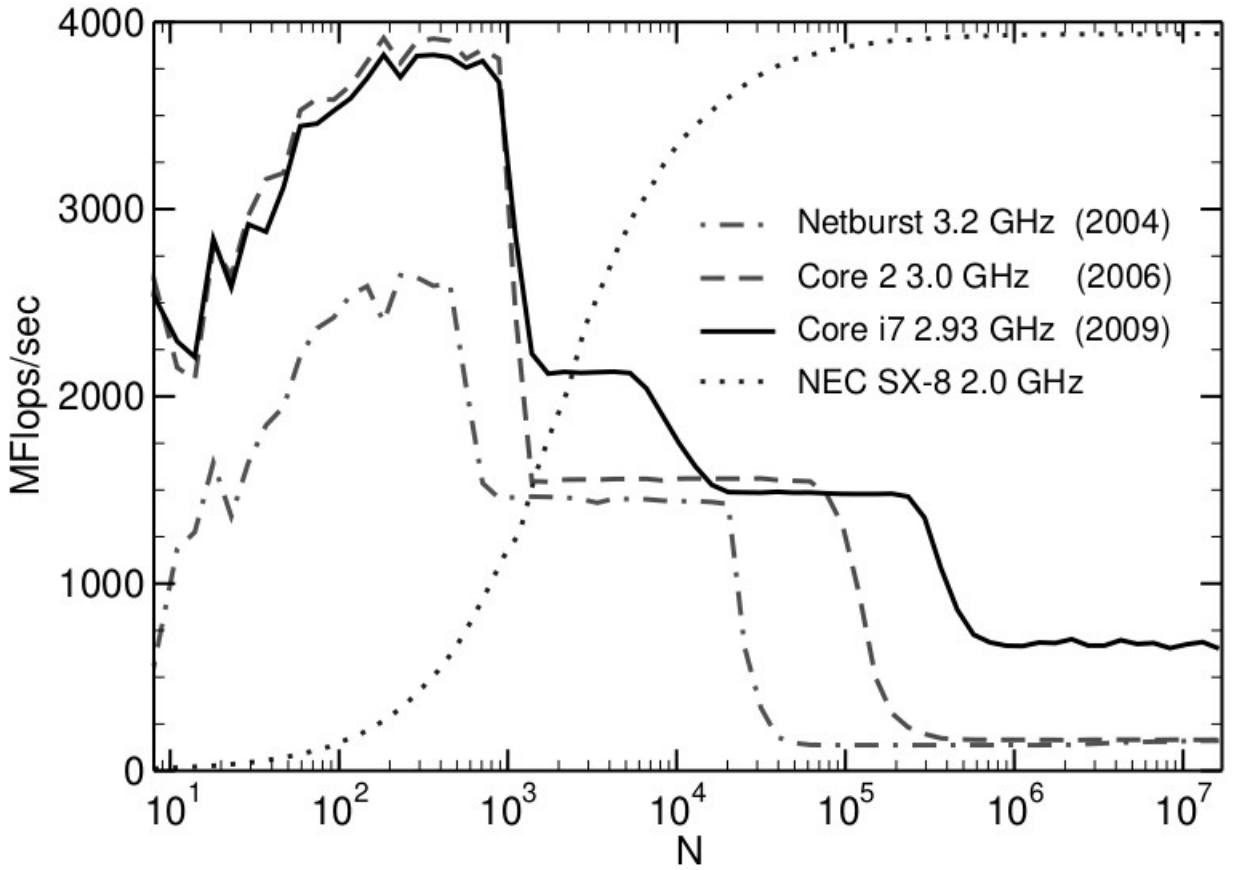
```
1  double precision, dimension(N) :: A,B,C,D
2  double precision :: S,E,MFLOPS
3
4  do i=1,N                              !initialize arrays
5    A(i) = 0.d0; B(i) = 1.d0
6    C(i) = 2.d0; D(i) = 3.d0
7  enddo
8
9  call get_walltime(S)                  ! get time stamp
10 do j=1,R
11   do i=1,N
12     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
```

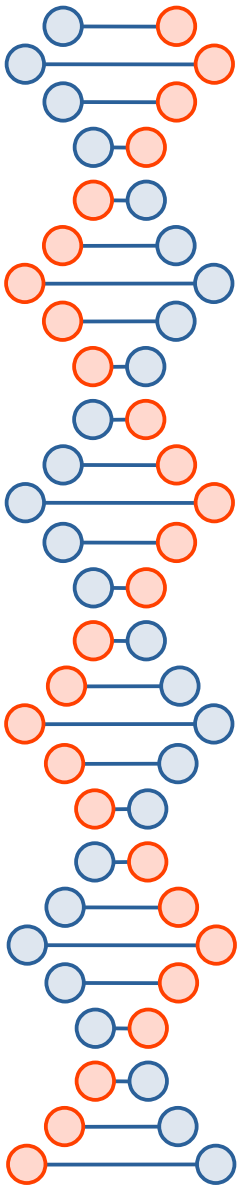


1.2 General-purpose cache-based microprocessor architecture

- On the inner level, three load streams for arrays B, C and D and one store stream for A are active.
- Depending on N, this loop might execute in a very small time, which would be hard to measure.
- The outer loop thus repeats the triad R times so that execution time becomes large enough to be accurately measurable.

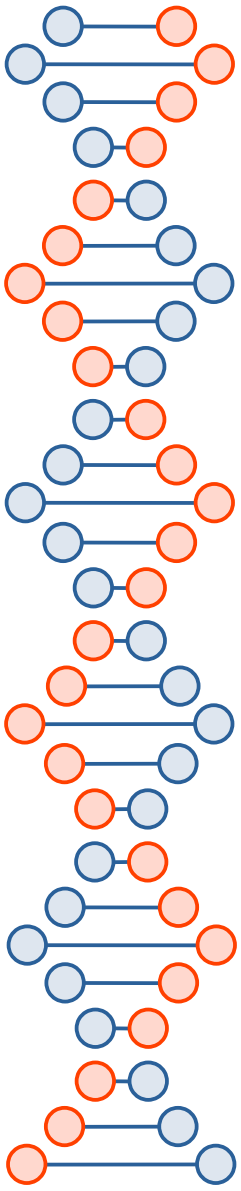
Figure 1.4 shows performance graphs for the vector triad





1.2 General-purpose cache-based microprocessor architecture

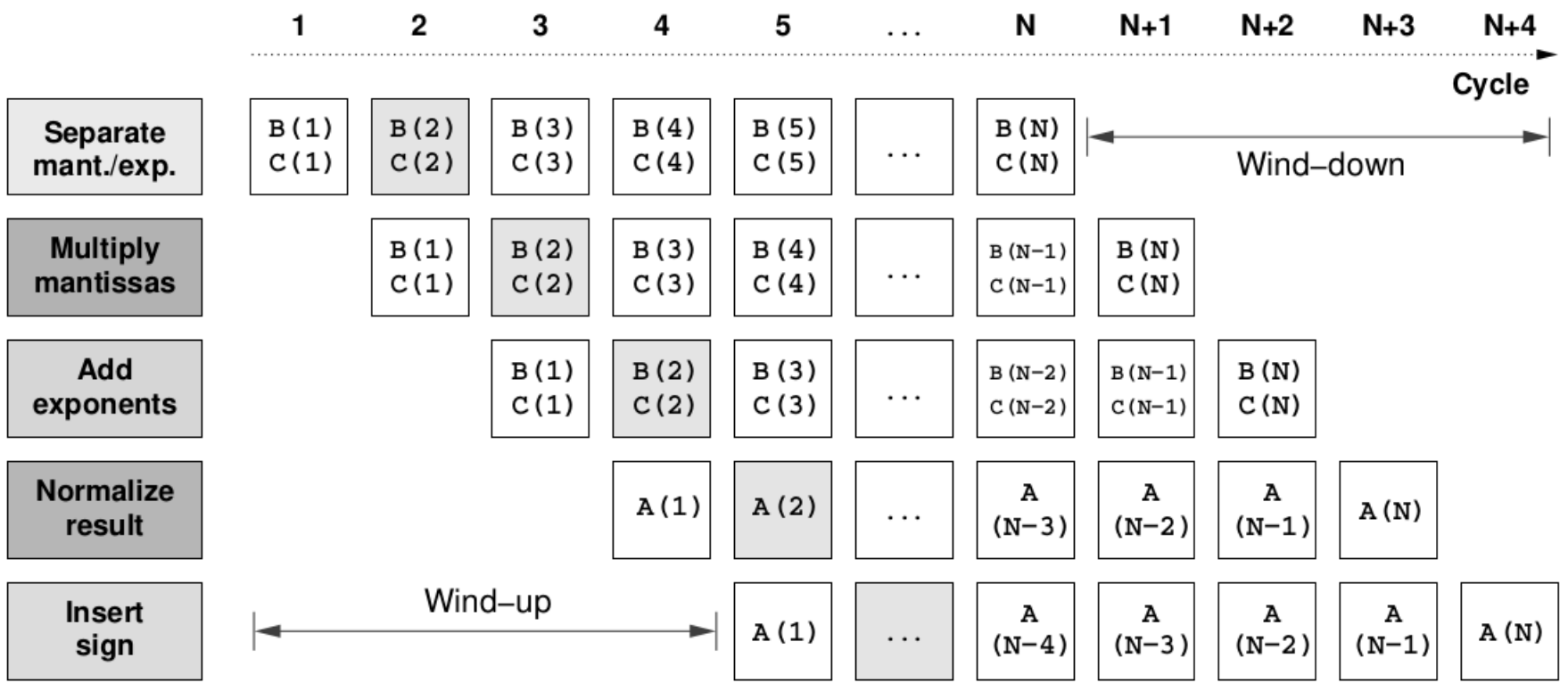
- Please note that the most sensible time measure in benchmarking is wallclock time, also called elapsed time.
- Any other “time” that the system may provide, first and foremost the much stressed CPU time, is prone to misinterpretation because there might be contributions from I/O, context switches, other processes, etc., which CPU time cannot encompass.
- On standard microprocessors, performance grows with N until some maximum is reached, followed by several sudden breakdowns.
- Finally, performance stays constant for very large loops. Those characteristics will be analyzed in detail in Section 1.3

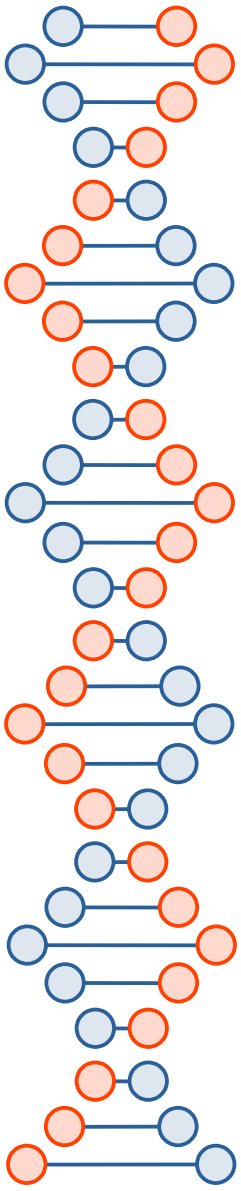


1.2 General-purpose cache-based microprocessor architecture

- Low-level benchmarks are powerful tools to get information about the basic capabilities of a processor. However, they often cannot accurately predict the behavior of “real” application code.
- This means that an application code is used with input parameters that reflect as closely as possible the real requirements of production runs.
- The decision for or against a certain architecture should always be heavily based on application benchmarking.
- Standard benchmark collections like the SPEC suite [W118] can only be rough guidelines.

A(:)=B(:)*C(:) in Floating-Point ALU





- 1.2 General-purpose cache-based microprocessor architecture

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

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

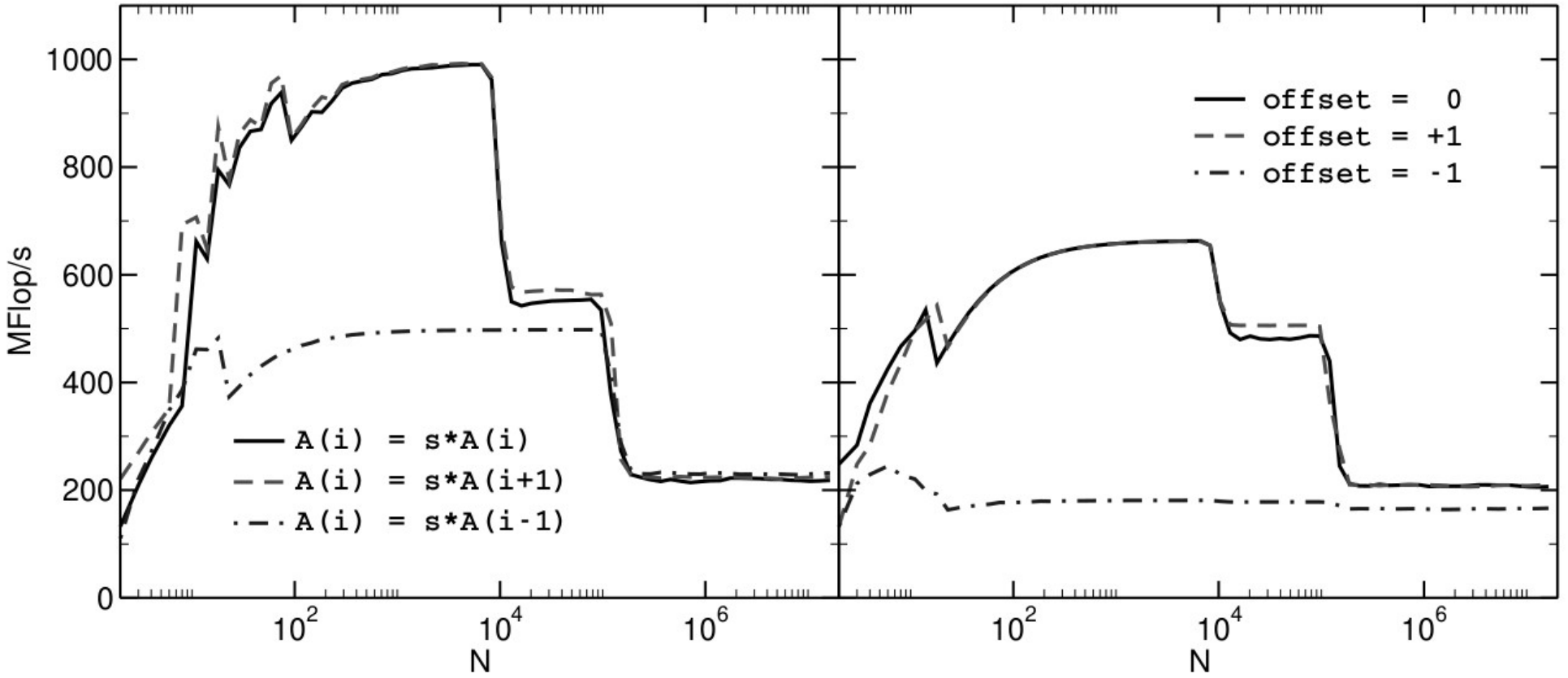
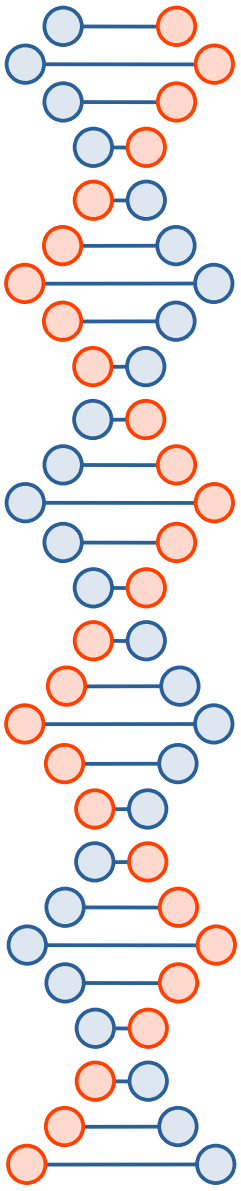
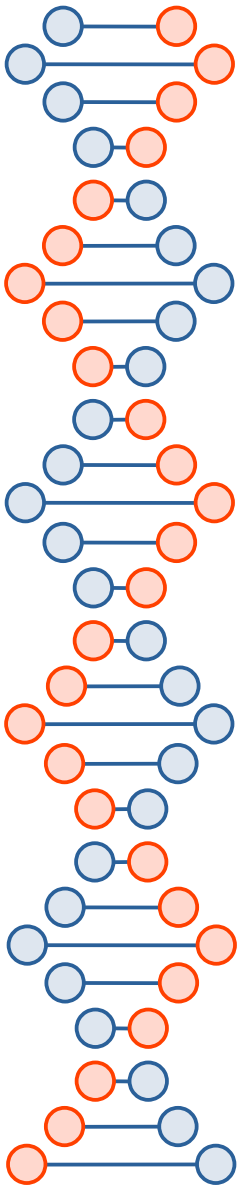


Figure 1.7: Influence of constant (left) and variable (right) offsets on the performance of a scaling loop. (AMD Opteron 2.0 GHz).



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

Although the multiply operation can be pipelined, the pipeline will *stall* if the load operation on `A(i)` does not provide the data on time. Similarly, the store operation can only commence if the latency for `mult` has passed and a valid result is available. Assuming a latency of four cycles for `load`, two cycles for `mult` and two cycles for `store`, it is clear that above pseudocode formulation is extremely inefficient. It is indeed required to *interleave* different loop iterations to bridge the latencies and avoid stalls:



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

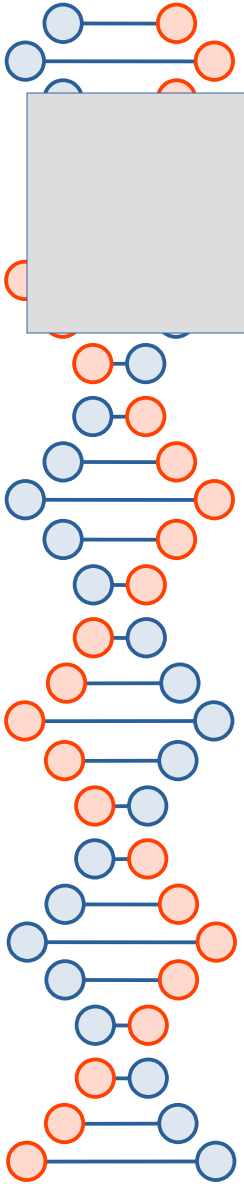
```
1 loop:  load A(i)
2        mult A(i) = A(i) * s
3        store A(i)
4        i = i + 1
5        branch -> loop
```

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

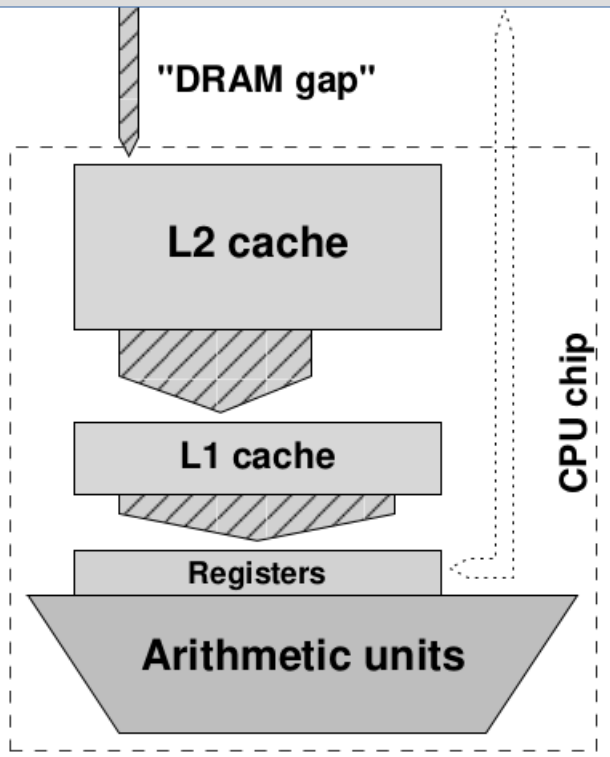
- 1.2 General-purpose cache-based microprocessor architecture

real dependency	pseudodependency	general version

`do i=2,N`
`  A(i)=s*A(`**`i-1`**`)`
`enddo`
	`do i=1,N-1`
	`  A(i)=s*A(`**`i+1`**`)`
	`enddo`
		`start=max(1,1-offset)`
		`end=min(N,N-offset)`
		`do i=start,end`
		`  A(i)=s*A(`**`i+offset`**`)`
		`enddo`



Phy Memory (DRAM)



1.3 Memory hierarchies

1.3 Memory hierarchies

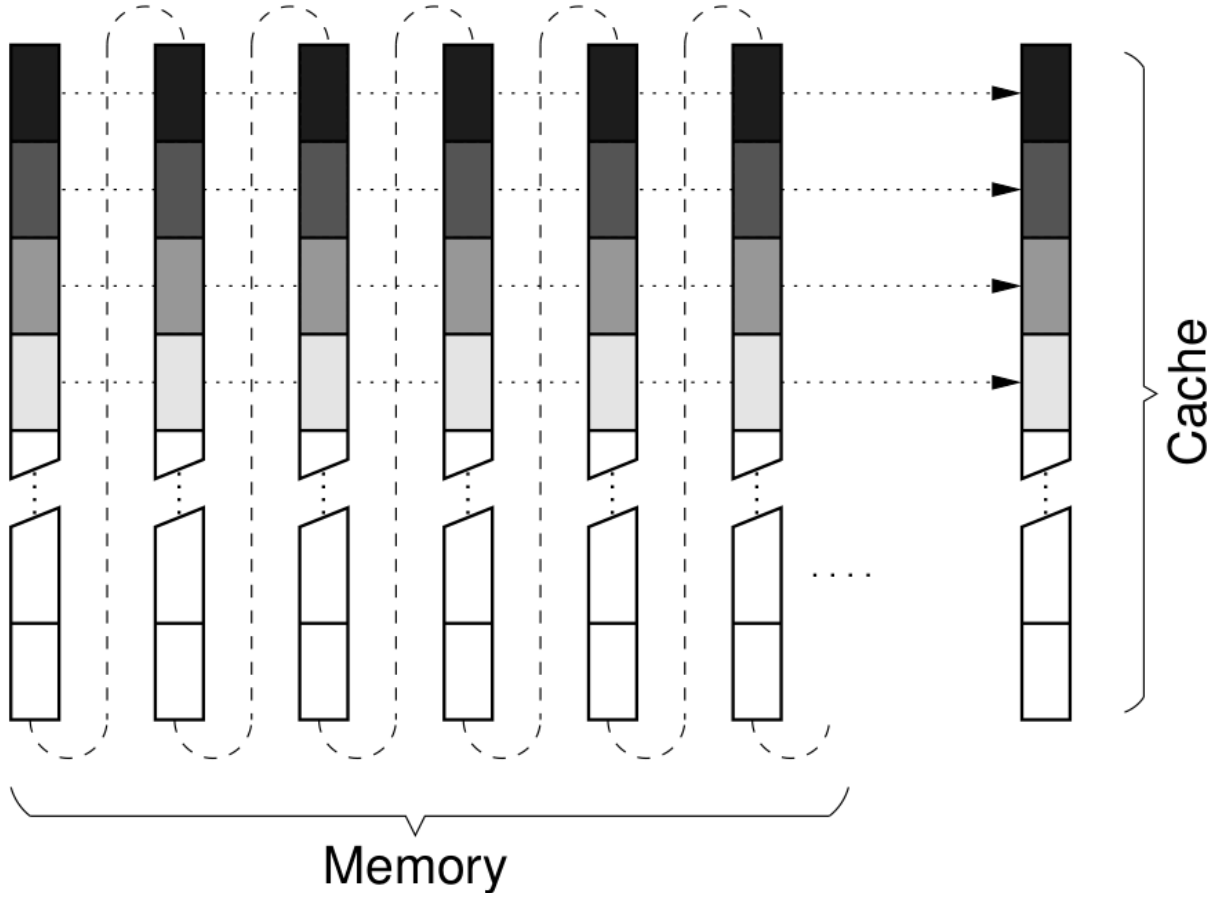


Figure 1.10: In a direct-mapped cache, memory locations which lie a multiple of the cache size apart are mapped to the same cache line (shaded boxes).

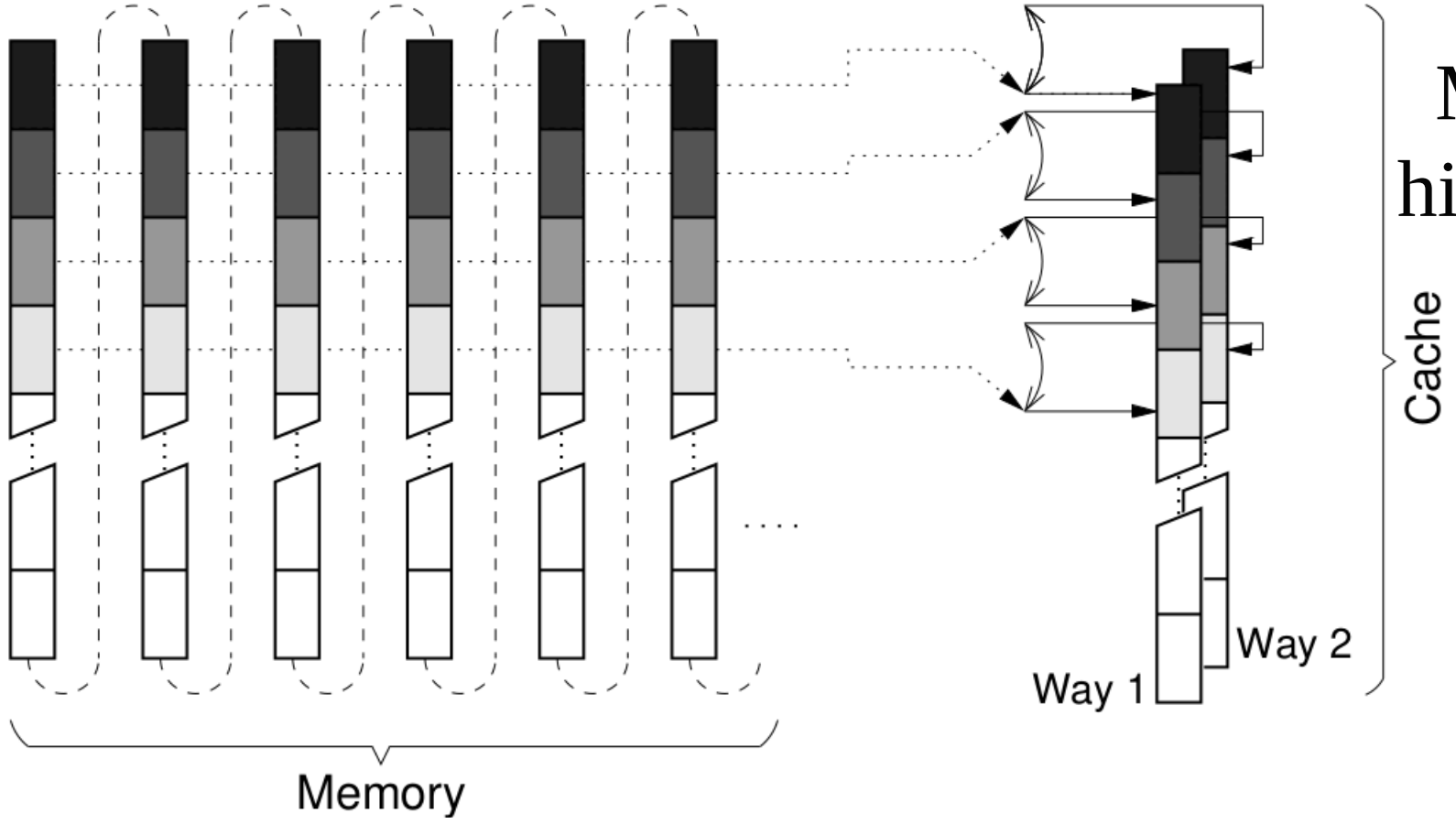


Figure 1.11: In an m-way set-associative cache, memory locations which are located a multiple of $\frac{1}{m}$th of the cache size apart can be mapped to either of m cache lines (here shown for $m = 2$).

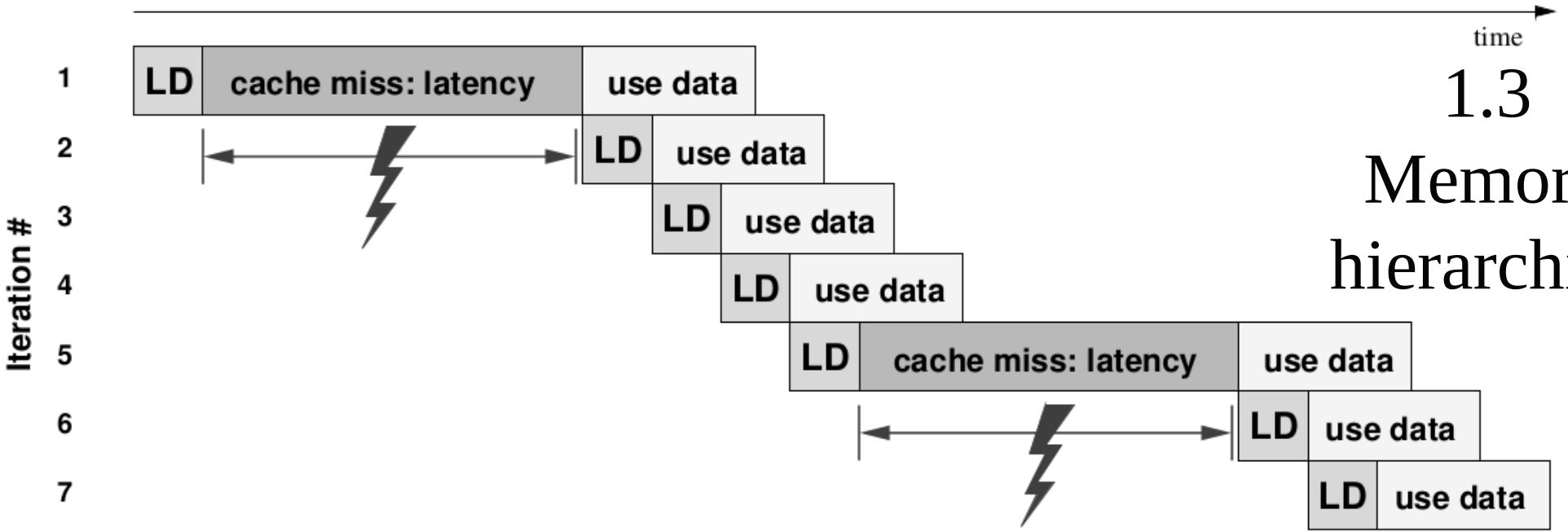
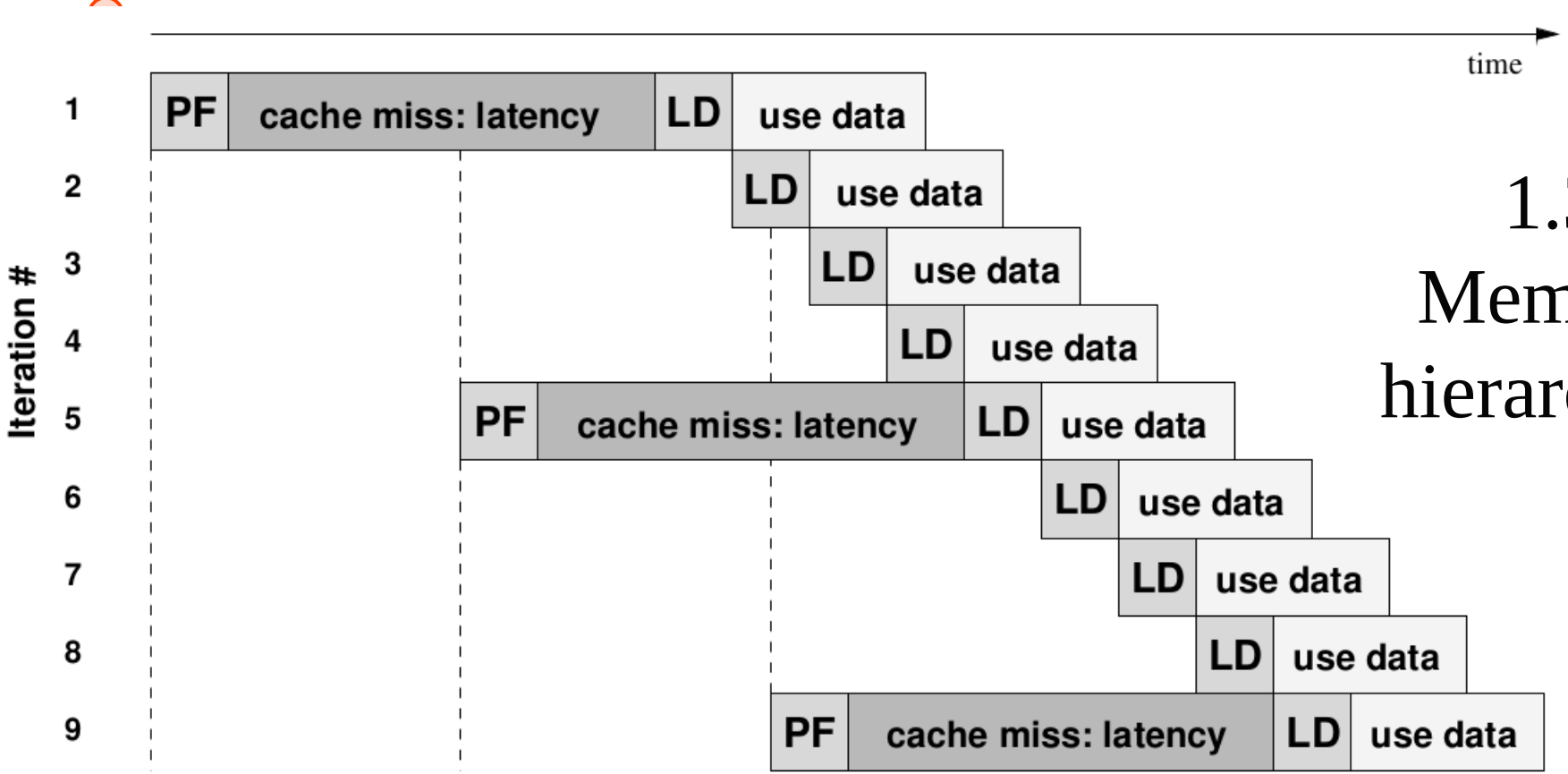


Figure 1.12: Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.



1.3 Memory hierarchies

Figure 1.13: Computation and data transfer can be overlapped much better with prefetching. In this example, two outstanding prefetches are required to hide latency completely.

• 1.4 Multicore processors

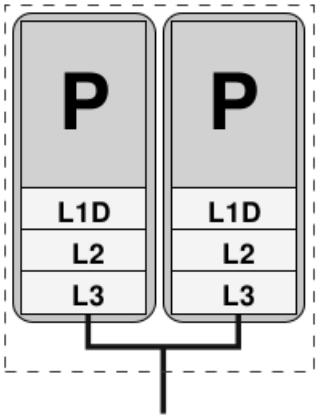


Figure 1.15: Dual-core processor chip with separate L1, L2, and L3 caches (Intel "Montecito"). Each core constitutes its own cache group on all levels.

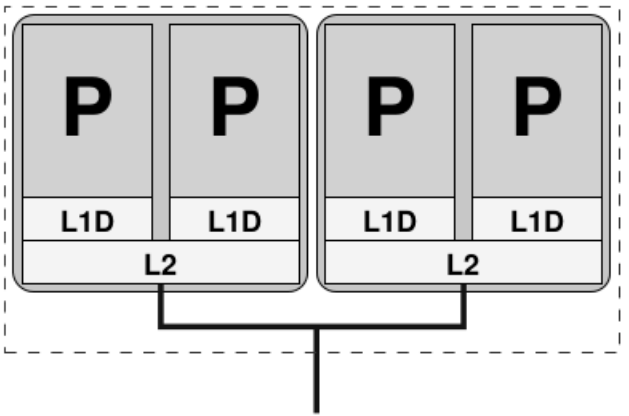


Figure 1.16: Quad-core processor chip, consisting of two dual-cores. Each dual-core has shared L2 and separate L1 caches (Intel "Harpertown"). There are two dual-core L2 groups.

1.4 Multicore processors

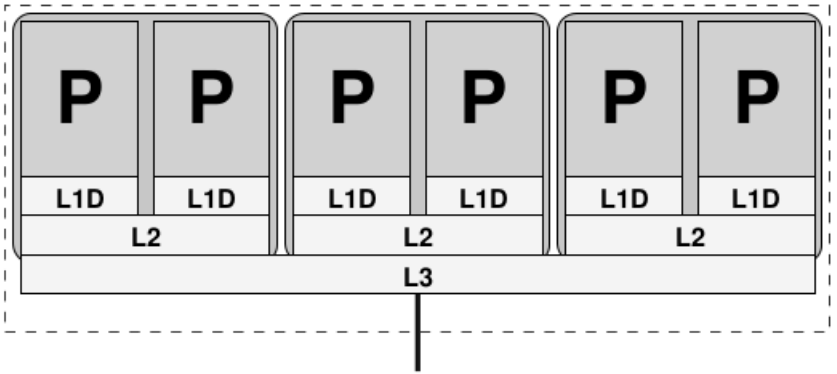


Figure 1.17: Hexa-core processor chip with separate L1 caches, shared L2 caches for pairs of cores and a shared L3 cache for all cores (Intel "Dunnington"). L2 groups are dual-cores, and the L3 group is the whole chip.

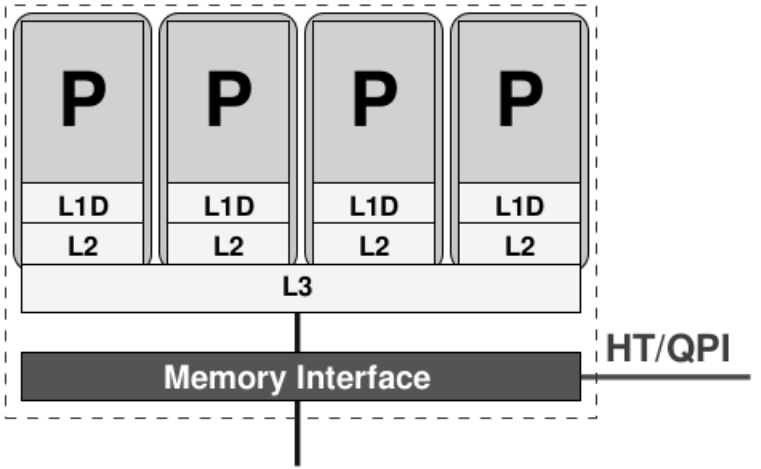


Figure 1.18: Quad-core processor chip with separate L1 and L2 and a shared L3 cache (AMD "Shanghai" and Intel "Nehalem"). There are four single-core L2 groups, and the L3 group is the whole chip. A built-in memory interface allows to attach memory and other sockets directly without a chipset.

• 1.5 Multithreaded processors

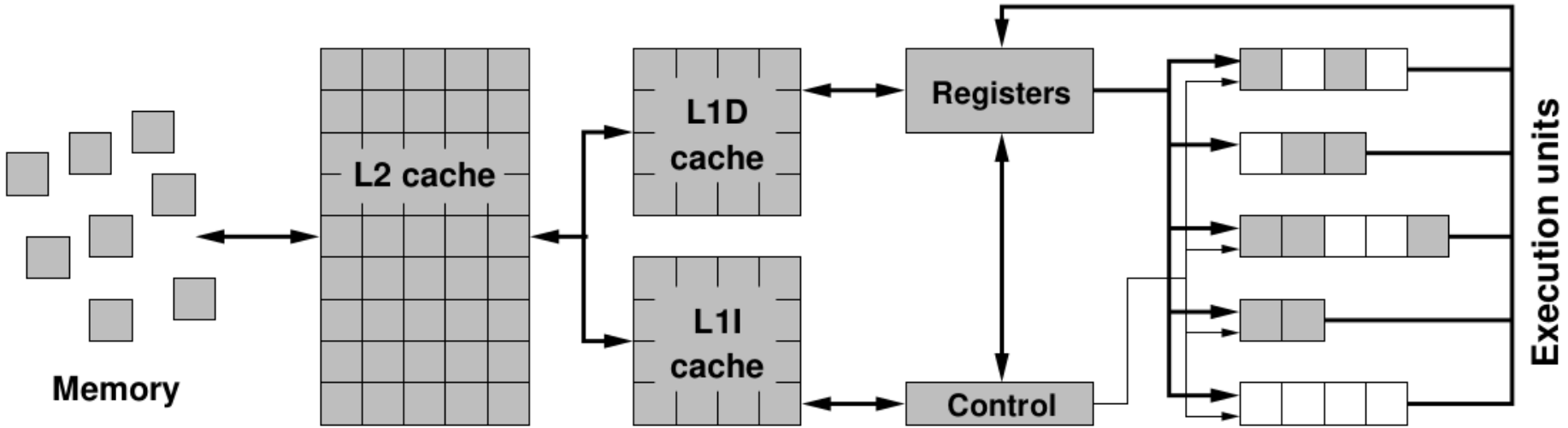


Figure 1.19: Simplified diagram of control/data flow in a (multi-)pipelined microprocessor without SMT. White boxes in the execution units denote pipeline bubbles (stall cycles). Graphics by courtesy of Intel.

• 1.5 Multithreaded processors

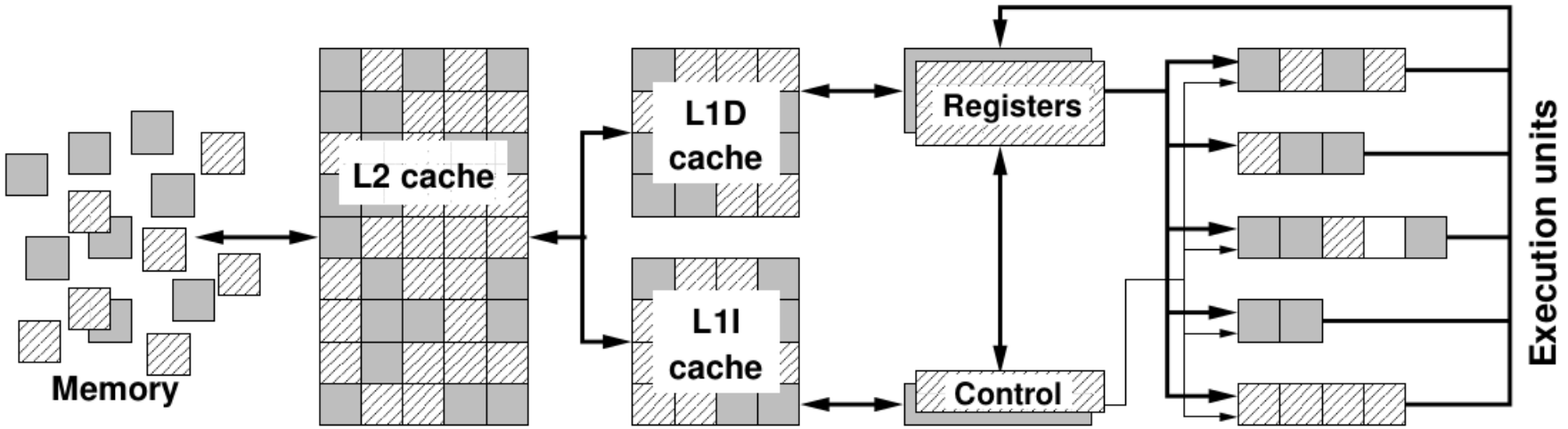


Figure 1.20: Simplified diagram of control/data flow in a (multi-)pipelined microprocessor with fine-grained two-way SMT. Two instruction streams (threads) share resources like caches and pipelines but retain their respective architectural state (registers, control units). Graphics by courtesy of Intel.

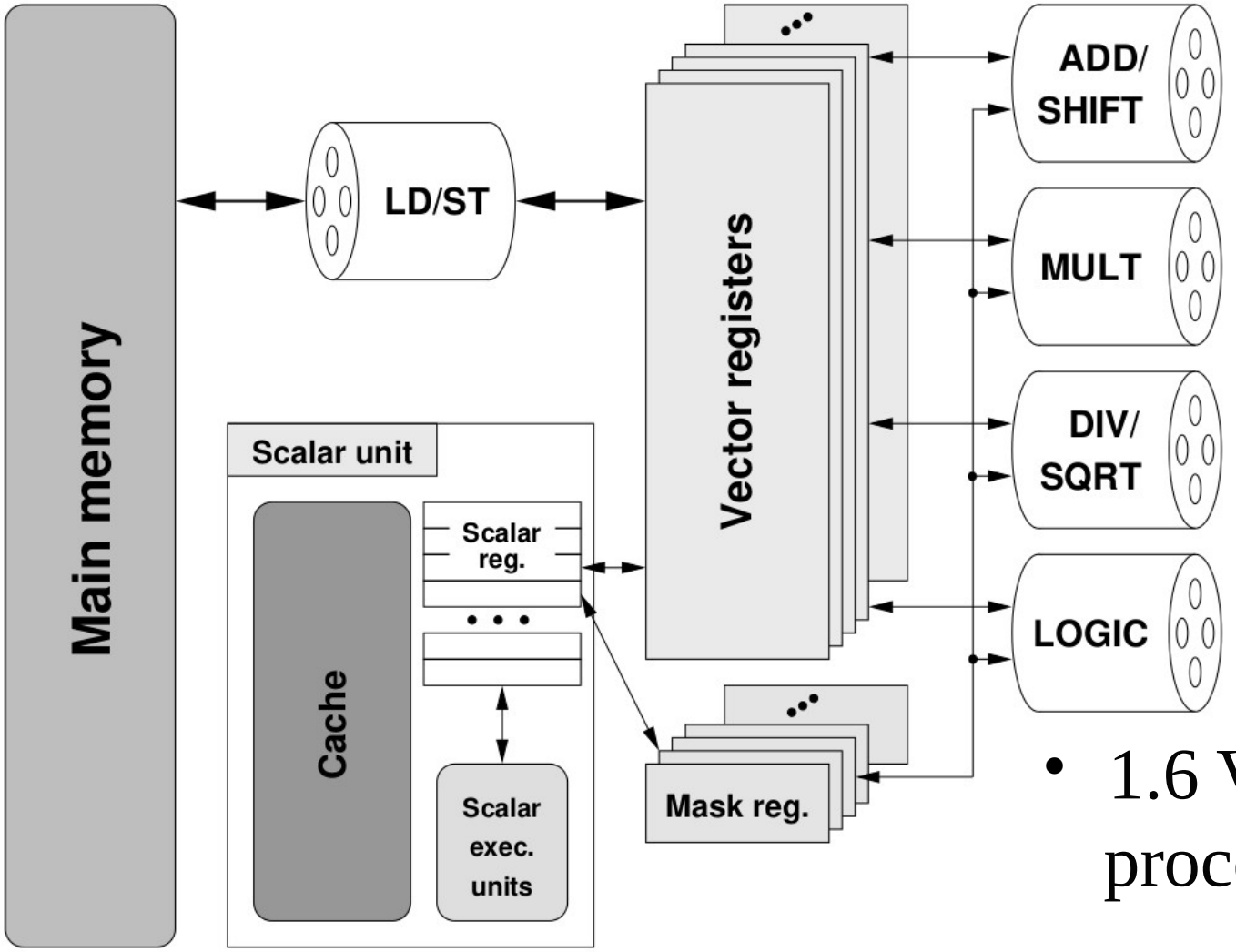


Figure 1.21: Block diagram of a prototypical vector processor with 4-track pipelines.

- 1.6 Vector processors

```
1 vload V1(1:N) = B(1:N)
2 vload V2(1:N) = C(1:N)
3 vadd V3(1:N) = V1(1:N) + V2(1:N)
4 vstore A(1:N) = V3(1:N)
```

Here, V1, V2, and V3 denote vector registers. The distribution of vector indices across the pipeline tracks is automatic. If the array length is larger than the vector length, the loop must be *stripmined*, i.e., the original arrays are traversed in chunks of the vector length:

```
1 do S = 1,N,Lv
2   E = min(N,S+Lv-1)
3   L = E-S+1
4   vload V1(1:L) = B(S:E)
5   vload V2(1:L) = C(S:E)
6   vadd V3(1:L) = V1(1:L) + V2(1:L)
7   vstore A(S:E) = V3(1:L)
8 enddo
```

- 1.6 Vector processors

• 1.6 Vector processors

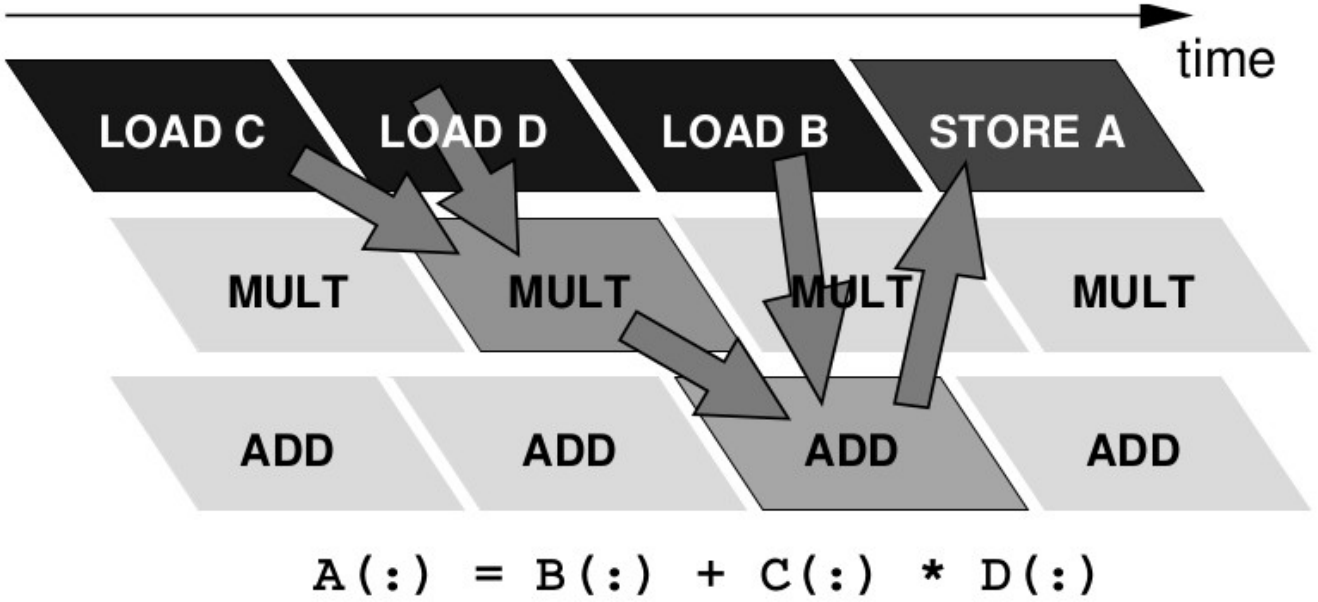


Figure 1.22: Pipeline utilization timeline for execution of the vector triad (see Listing 1.1) on the vector processor shown in Figure 1.21. Light gray boxes denote unused arithmetic units.

• 1.6 Vector processors

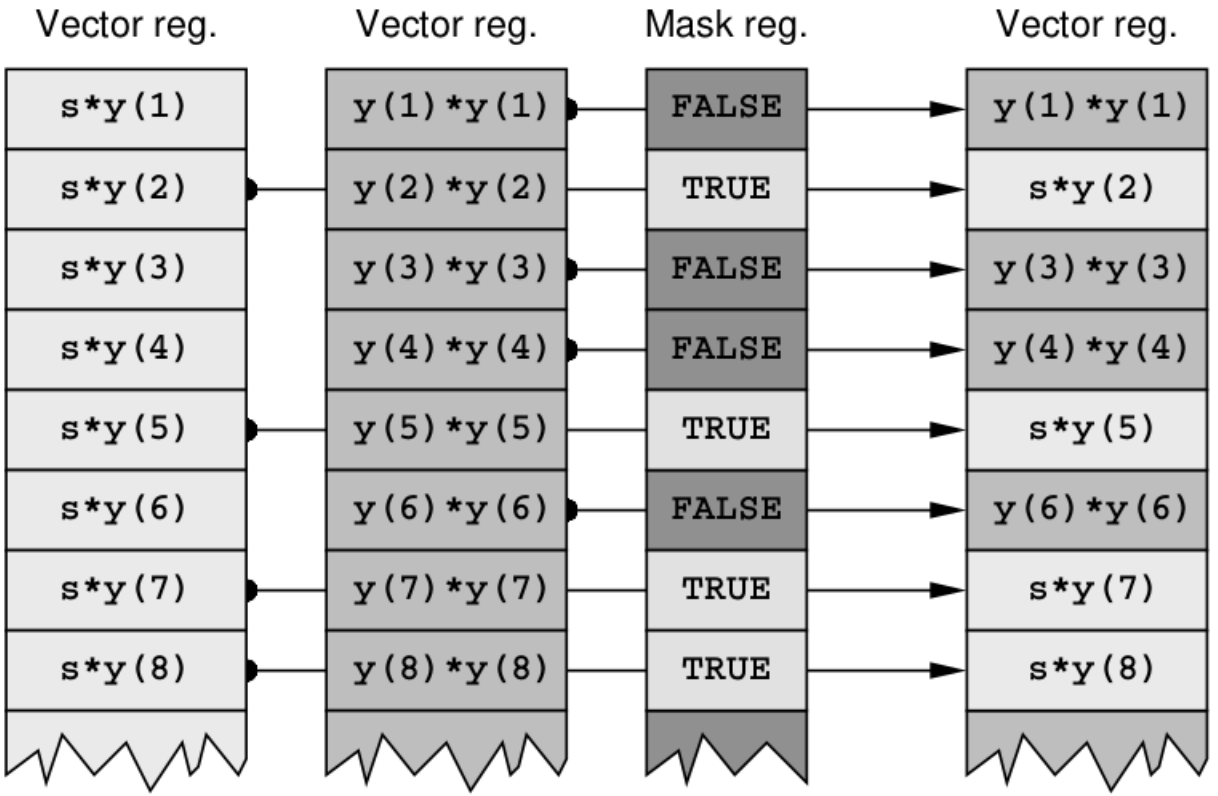


Figure 1.23: On a vector processor, a loop with an if/else branch can be vectorized using a mask register.

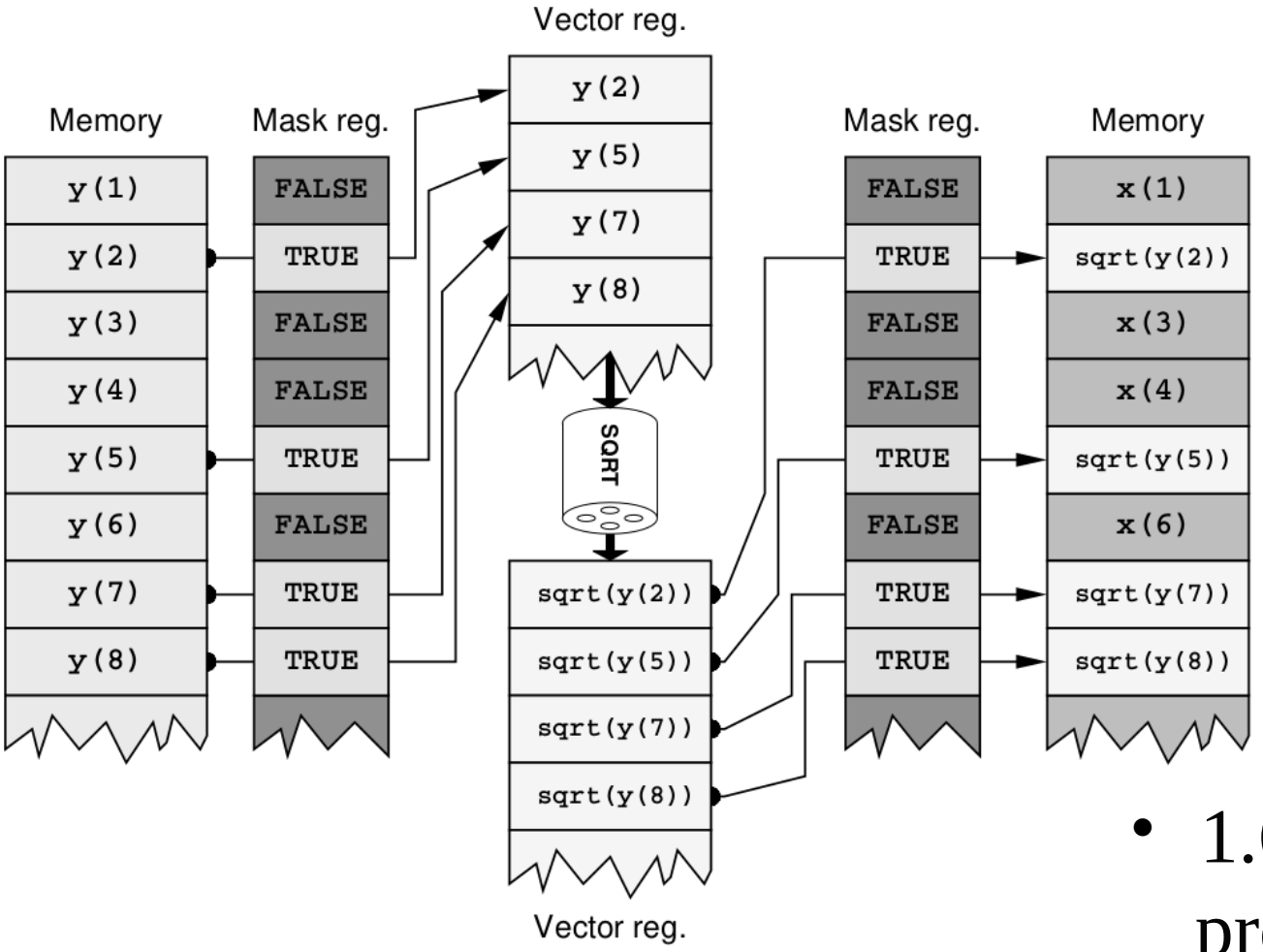


Figure 1.24: Vectorization by the gather/scatter method. Data transfer from/to main memory occurs only for those elements whose corresponding mask entry is true. The same mask is used for loading and storing data.

- 1.6 Vector processors