CS395T: Introduction to Scientific and Technical Computing

topic: single-processor computer architecture

Instructors:

Dr. Karl W. Schulz, Research Associate, TACC

Dr. Victor Eijkhout, Research Scientist, TACC

Outline

- Basic Anatomy of a Server/Desktop/Laptop/Cluster-node
  - Memory Hierarchy
    - Structure, Size, Speed, Line Size, Associativity
    - Latencies and Bandwidths
  - Intel vs. AMD platforms
    - Memory Architecture

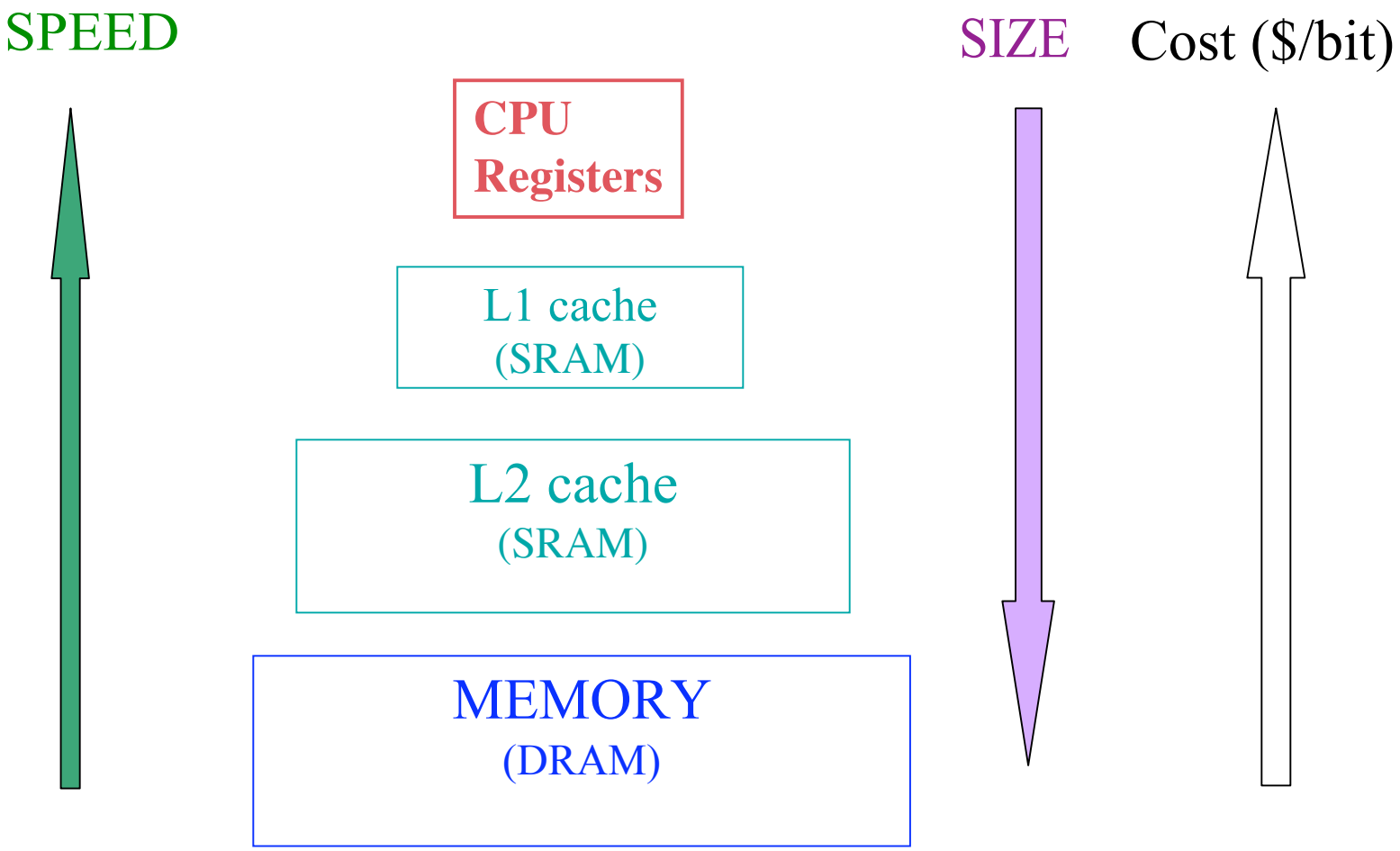
Microarchitecture

- Memory hierarchies
- Commodity CPUs
  - theoretical performance
  - pipelining
  - superscaling
  - Performance

Memory Hierarchies

- Due primarily to cost, memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory
- Memory is accessed through the hierarchy
  - registers where possible
  - ... then the caches
  - ... then main memory

Memory Relativity

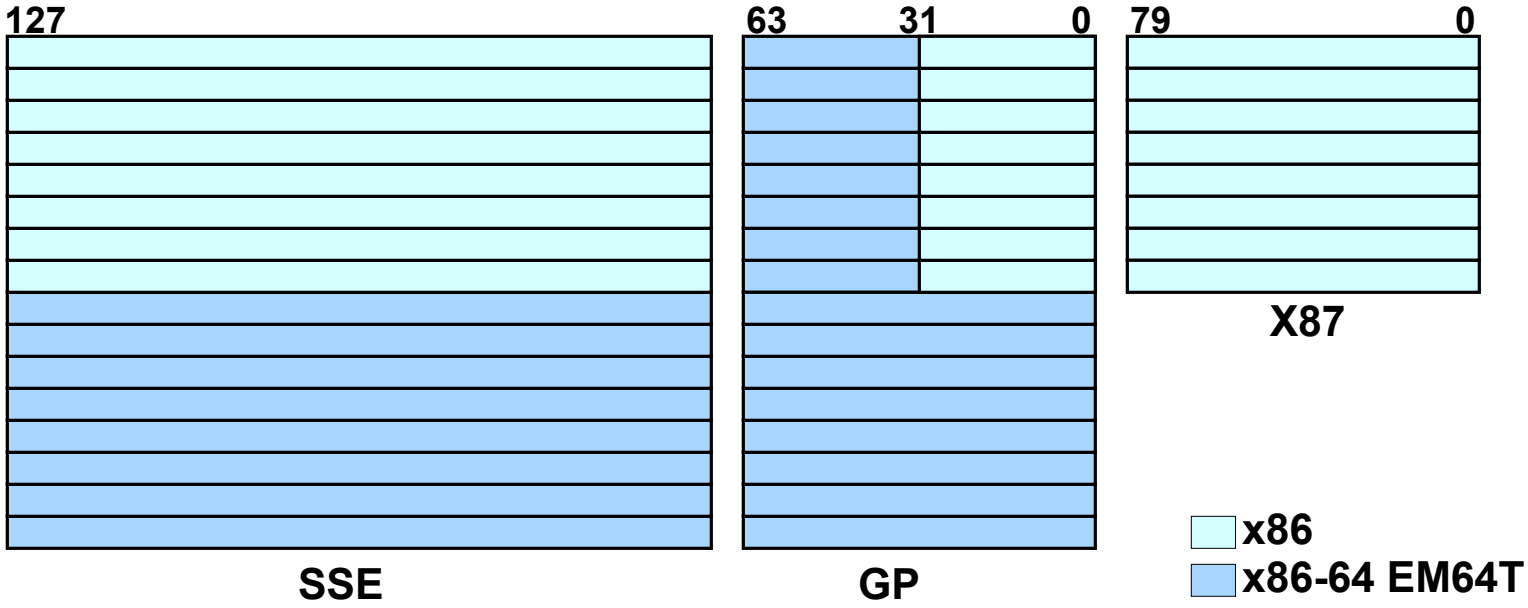


Latency and Bandwidth

- The two most important terms related to performance for memory subsystems and for networks are:
  - Latency
    - How long does it take to retrieve a word of memory?
    - Units are generally nanoseconds or clock periods (CP).
  - Bandwith
    - What data rate can be sustained once the message is started?
    - Units are B/sec (MB/sec, GB/sec, etc.)

Registers

- Highest bandwidth, lowest latency memory that a modern processor can acces
  - built into the CPU
  - often a scarce resource
  - not RAM
- AMD x86-64 and Intel EM64T Registers



Registers

- Processors instructions operate on registers directly
  - have names like: eax, ebx, ecx, etc.
  - sample instruction:

```
addl   %eax, %edx
```

- Separate instructions and registers for floating-point operations

Cache

- Between the CPU Registers and main memory
- L1 Cache : Data cache closest to registers (on die)
- L2 Cache: Secondary data cache, stores both data and instructions (on die)
  - Data from L2 has to go through L1 to registers
  - L2 is 10 to 100 times larger than L1
  - Some systems have a off-die L3 cache, ~10x larger than L2
- Cache line
  - The smallest unit of data transferred between main memory and the caches (or between levels of cache)
  - *N* sequentially-stored, multi-byte words (usually N=8 or 16).

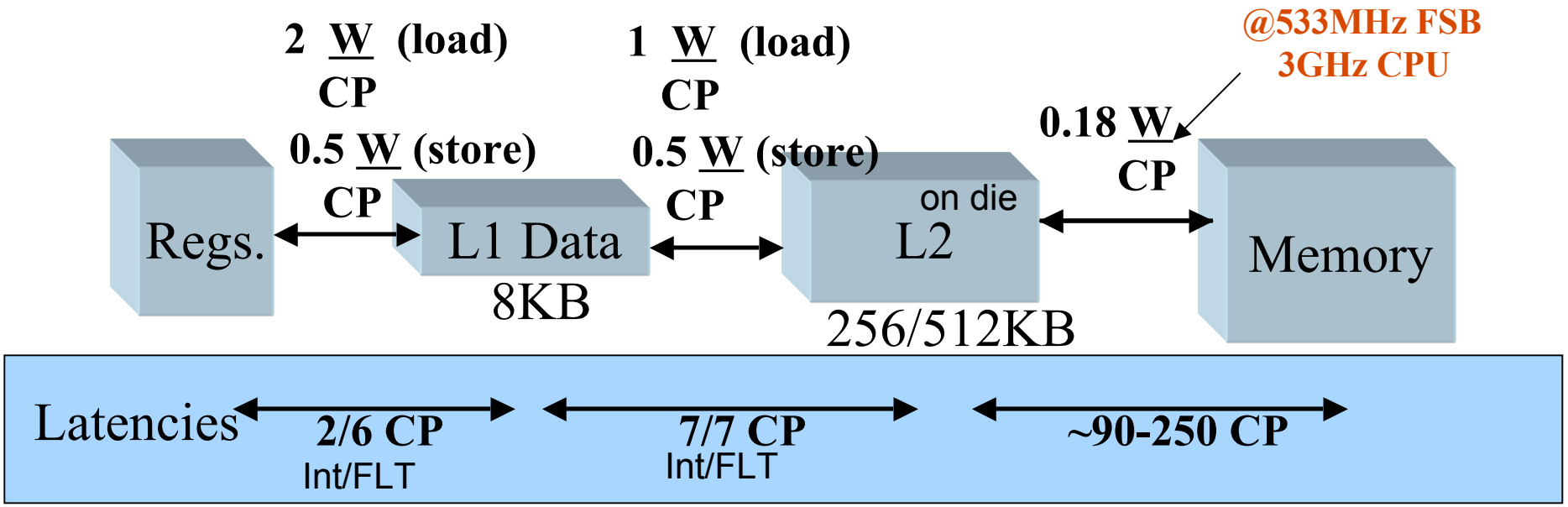
Main Memory

- Cheapest form of RAM
- Also the slowest
  - lowest bandwidth
  - highest latency
- Unfortunately most of our data lives out here

Approximate Latencies and Bandwidths in a Memory Hierarchy

	Latency	Bandwidth
Registers		
L1 Cache	**~5 CP**	**~2 W/CP**
L2 Cache	**~15 CP**	**~1 W/CP**
Memory	**~300 CP**	**~0.25 W/CP**
Dist. Mem.	**~10000 CP**	**~0.01 W/CP**

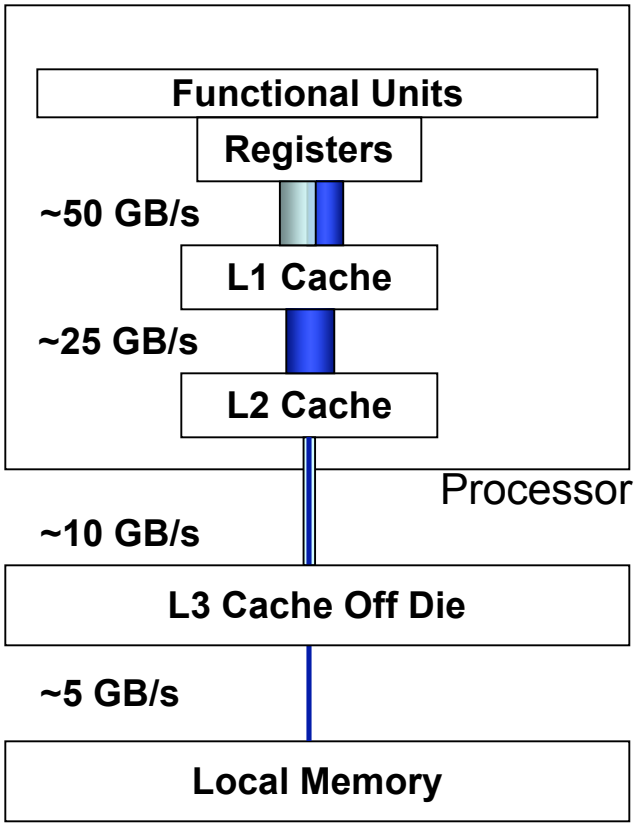
Example: Pentium 4



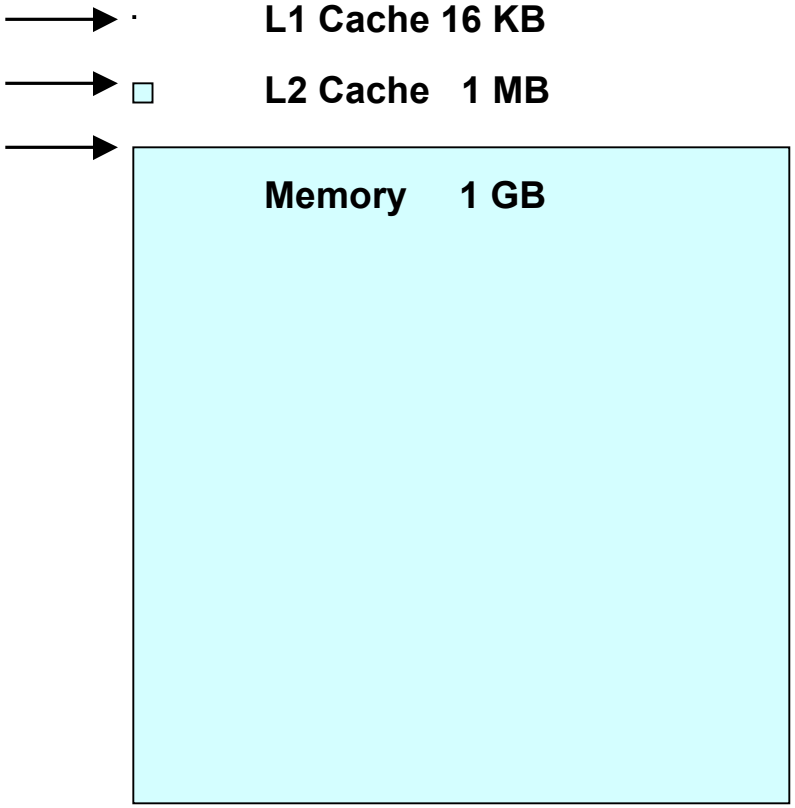
Line size L1/L2 =8W/16W

Memory Bandwidth and Size Diagram

Relative Memory Bandwidths

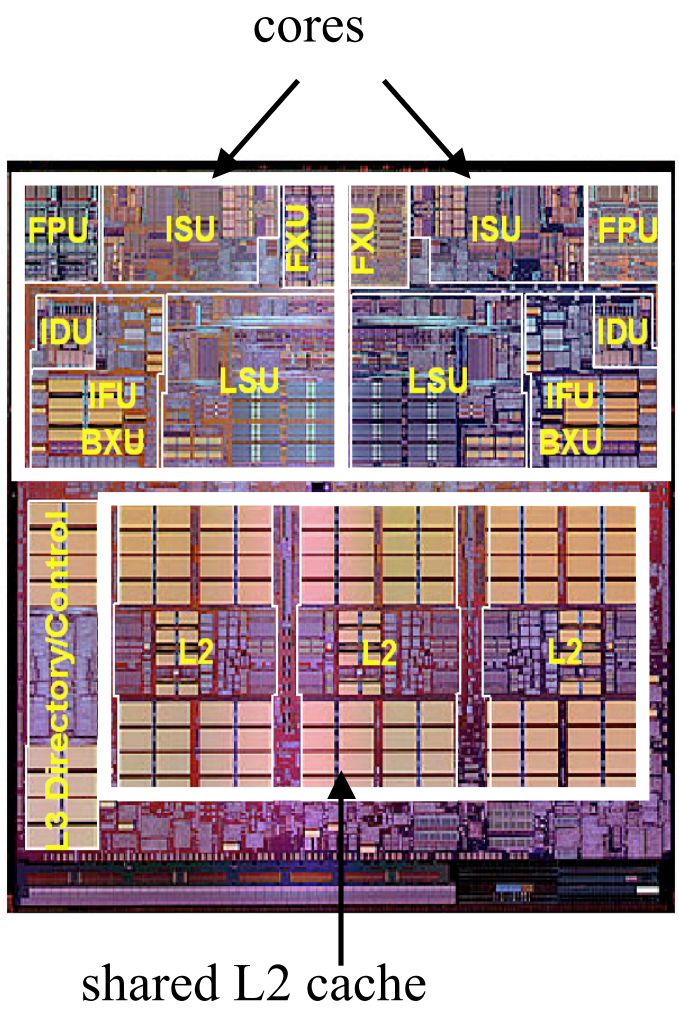


Relative Memory Sizes

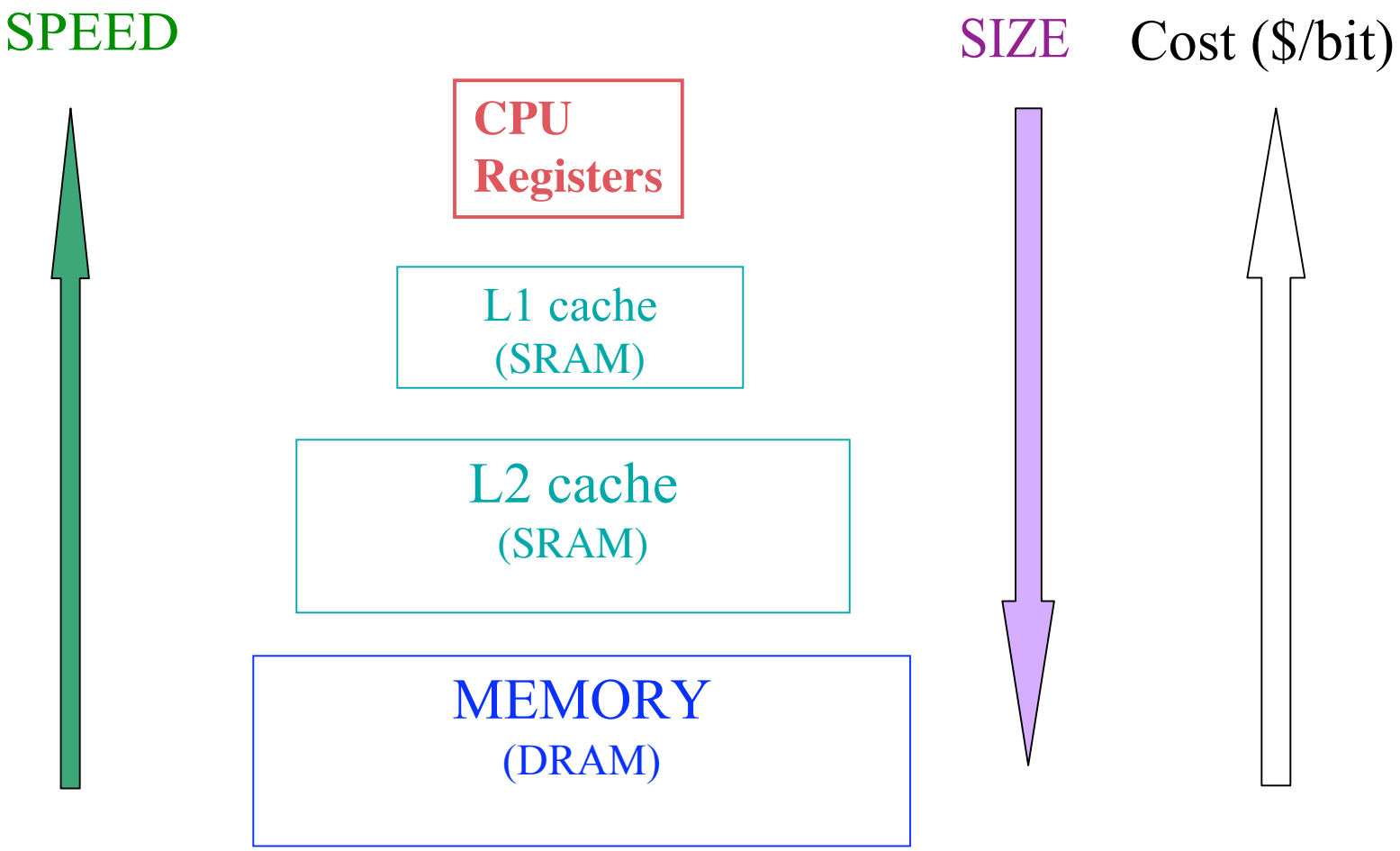


TACC

IBM Power4 Chip Layout



Memory/Cache Related Terms



Why Caches?

- Since registers are expensive
- ... and main memory slow
- Caches provide a buffer between the two
- Access is transparent
  - either it's in a register or
  - it's in a memory location
  - processor/cache controller/MMU hides cache access from the programmer

Memory Access Example

```
#include <stdlib.h>
#include <stdio.h>
#define N 1234
int main()
{
  int i;
  int *buf=malloc(N*sizeof(int));
  buf[0]=1;
  for (i=1; i < N; ++i)
    buf[i]=i;
  printf("%d\n",buf[N-1]);
}
```

```
        movl    $1, (%eax)
        movl    $1, %edx
.L2:
        movl    %edx, (%eax,%edx,4)
        addl    $1, %edx
        cmpl    $1234, %edx
        jne     .L2
```

Hits, Misses, Thrashing

Cache hit

- location referenced is found in the cache

- Cache miss
  - location referenced is not found in cache
  - triggers access to the next higher cache or memory
- Cache thrashing
  - a thrashed cache line (TCL) must be repeatedly recalled in the process of accessing its elements
  - caused when other cache lines, assigned to the same location, are simultaneous accessing data/instructions that replace the TCL with their content.

Design Considerations

- Data cache designed with two key concepts in mind
- Spatial Locality
  - when an element is referenced, its neighbors will be referenced, too
  - all items in the cache line are fetched together
  - work on consecutive data elements in the same cache line gives performance boost
- Temporal Locality
  - when an element is referenced, it will be referenced again soon
  - arrange code so that date in cache is reused as often as possible

Cache Line Size vs Access Mode

Access Line-size	Sequential data	Random data
Short Line	more fetches (overhead)	Best—low “latency”
Long Line	Best--Fewer fetches, but higher probability for cache trashing.	Longer “latency”, effectively smaller cache

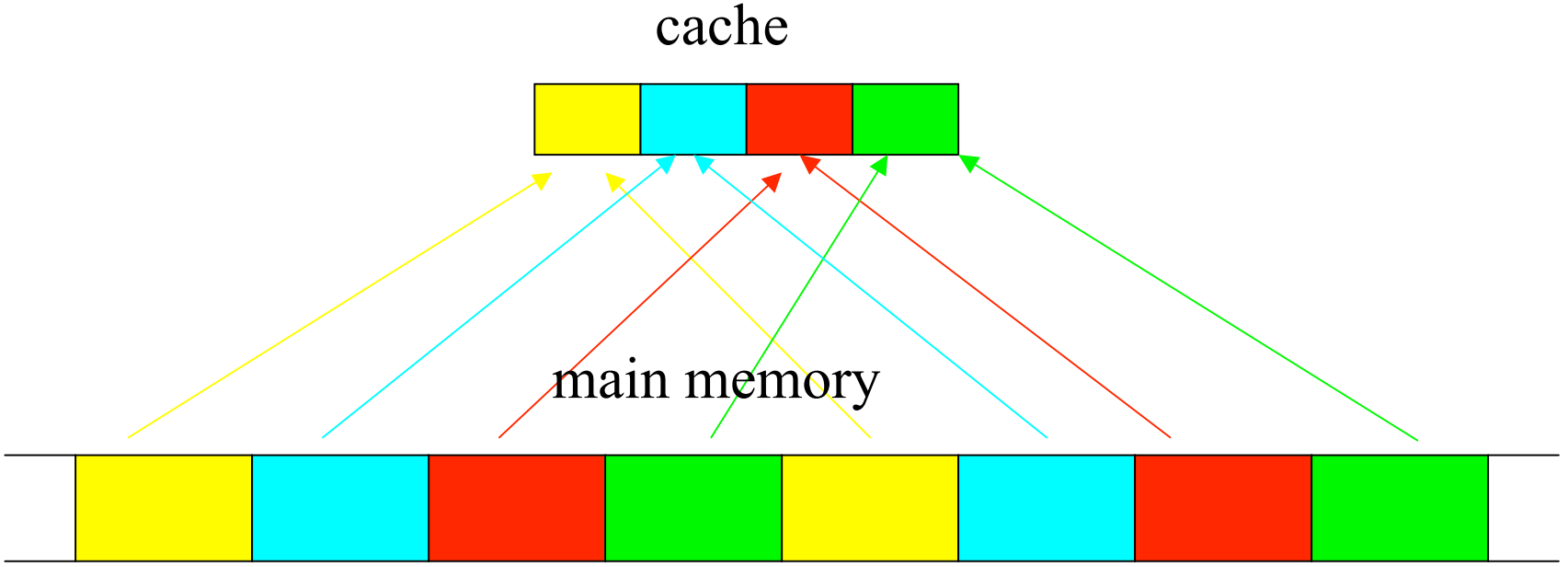
Cache Mapping

- Because each memory subsystem is smaller than the next-closer level, data must be mapped

- Types of mapping
  - Direct
  - Set associative
  - Fully associative

Direct Mapped Caches

Direct mapped cache: A block from main memory can go in exactly one place in the cache. This is called direct mapped because there is direct mapping from any block address in memory to a single location in the cache.

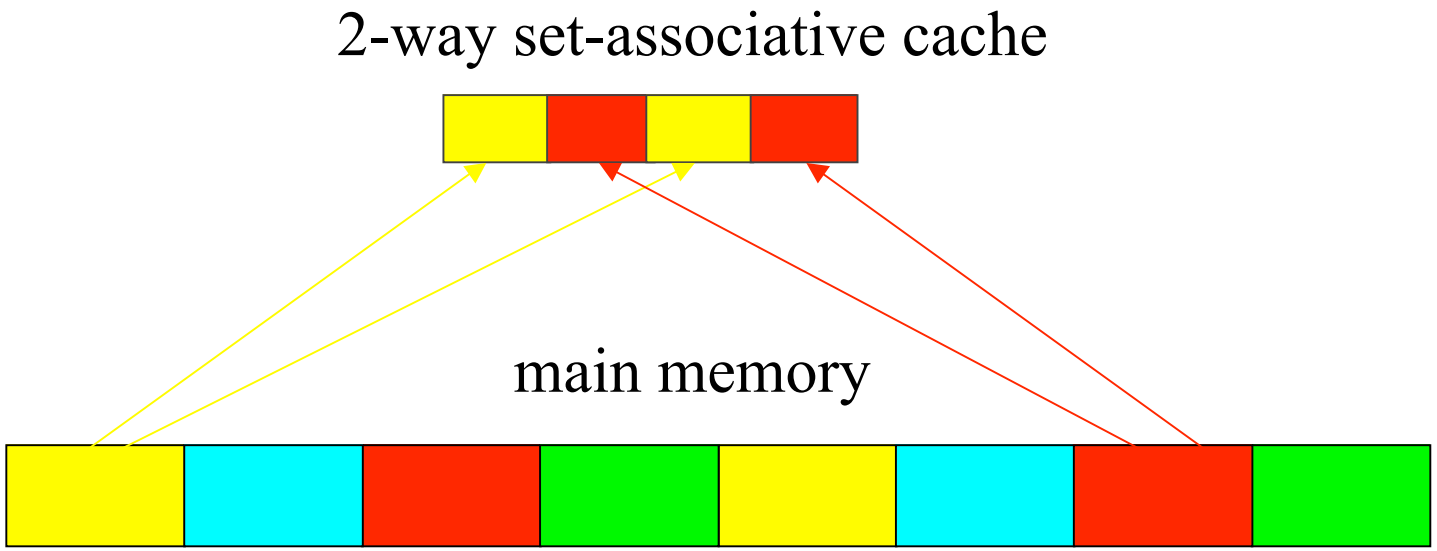


Direct Mapped Caches

- If the cache size is N_c and it is divided into k lines, then each cache line is N_c/k in size
- If the main memory size is N_m, memory is then divided into $N_m/(N_c/k)$ blocks that are mapped into each of the k cache lines
- Means that each cache line is associated with particular regions of memory

Set Associative Caches

Set associative cache : The middle range of designs between direct mapped cache and fully associative cache is called set-associative cache. In a n-way set-associative cache a block from main memory can go into n (n at least 2) locations in the cache.

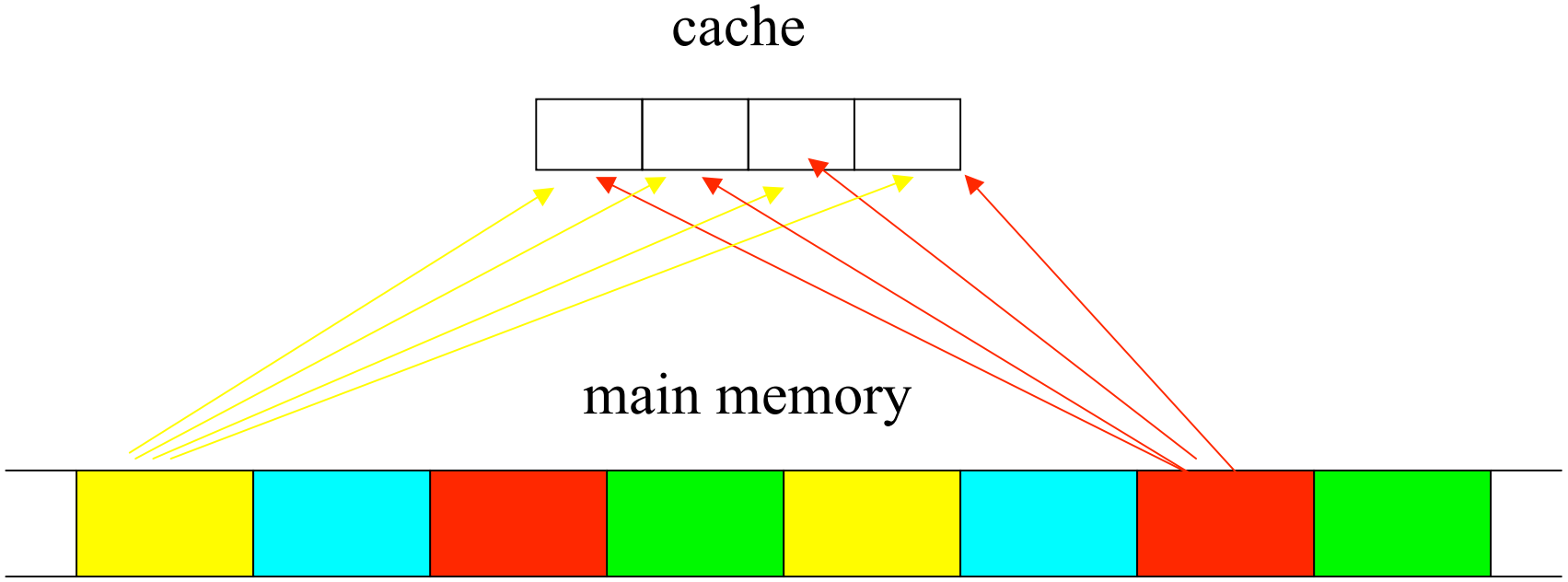


Set Associative Caches

- Direct-mapped caches are 1-way set-associative caches
- For a k-way set-associative cache, each memory region can be associated with *k* cache lines

Fully Associative Caches

Fully associative cache : A block from main memory can be placed in any location in the cache. This is called fully associative because a block in main memory may be associated with any entry in the cache.



Fully Associative Caches

- Ideal situation
- Any memory location can be associated with any cache line
- Cost prohibitive

Intel Woodcrest Caches

- L1
  - 32 KB
  - 8-way set associative
  - 64 byte line size
- L2
  - 4 MB
  - 8-way set associative
  - 64 byte line size

TLB

- Translation Look-aside Buffer
- Translates between logical space that each program has and actual memory addresses
- Memory organized in ‘small pages’, a few Kbyte in size
- Memory requests go through the TLB, normally very fast
- Pages that are not tracked through the TLB can be found through the ‘page table’: much slower
- => jumping between more pages than the TLB can track has a performance penalty.
- This illustrates the need for spatial locality.

Data reuse

- Performance is limited by data transfer rate
- High performance if data items are used multiple times
- Example: vector addition $x_i=x_i+y_i$: 1op, 3 mem accesses
- Example: inner product $s=s+x_i*y_i$: 2op, 2 mem access (s in register; also no writes)

Data reuse: matrix-vector product

- Matrix vector product: $2n^2$ ops, n^2+2n data
- However, only theoretical: in naïve implementation the source vector gets reloaded lots of times
- Loop exchange doesn't help: output gets written many times
- Solution: blocking (see below)

```
for (i) {
  s = 0;
  for (j)
    s = s+A_ij*x_j;
  y_i  = s;
}
```

```
for (j) {
  for (i)
    y_i = y_i+A_ij*x_j;
}
```

Data reuse: matrix-matrix product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data
- If it can be programmed right, this can overcome the bandwidth/cpu speed gap
- Again only theoretically: naïve implementation inefficient

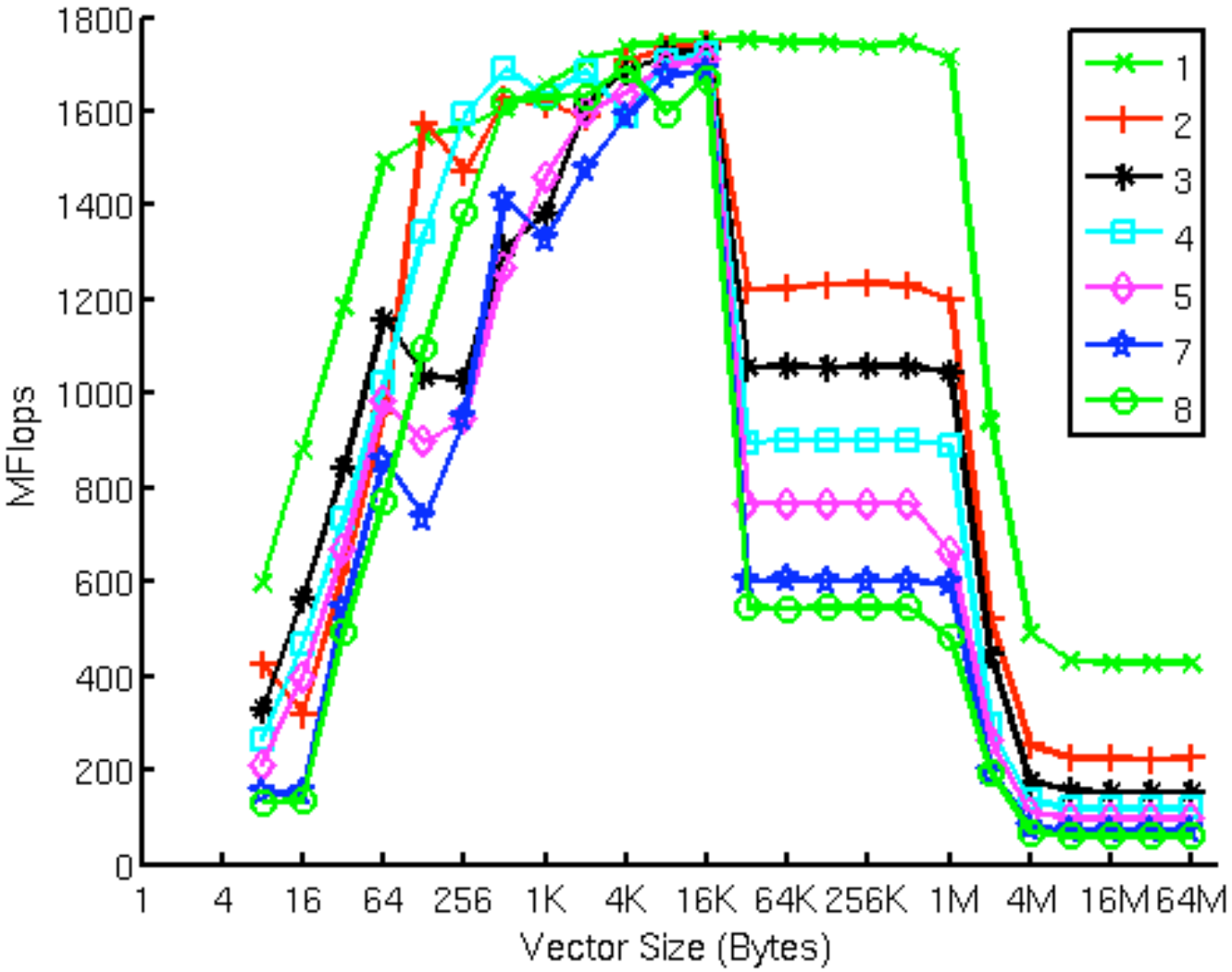
Programming strategies: contiguous access

- Avoid strides: cache lines contain 4 (or so) words, might as well use them all
- Example: dot product, sequential access of the vectors
- Strided dot product:

```
sum=0.;
for (j=0; j < stride; ++j)
  for(i=j; i < n; i+=stride)
    sum += a[i]*b[i];
```

Not all elements on a cache line used.

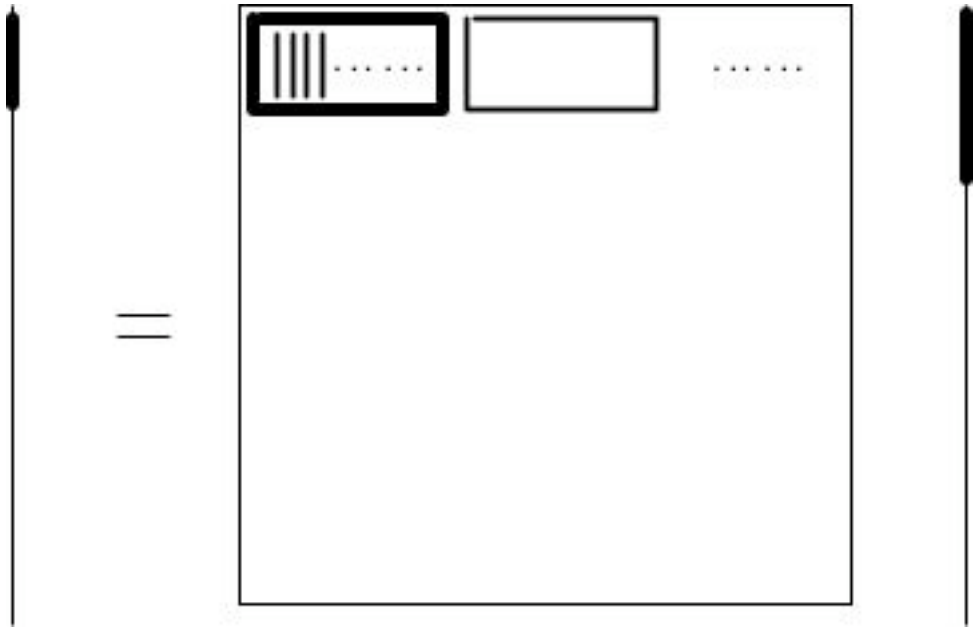
Dot Product Performance



TACC

Programming strategies: blocked algorithms

- Long vectors are flushed from cache: break up in smaller blocks
- Reuse of input vector (limited)
- Use of cache lines in matrix



More on blocked algorithms

- This gets tricky fast
- Matrix-matrix multiply is triple loop; blocked is 6-deep loop
- Choice of blocking sizes is complicated
- Loop exchange to aim for L1 or L2 reuse (Atlas vs Goto approach)

Pipelining

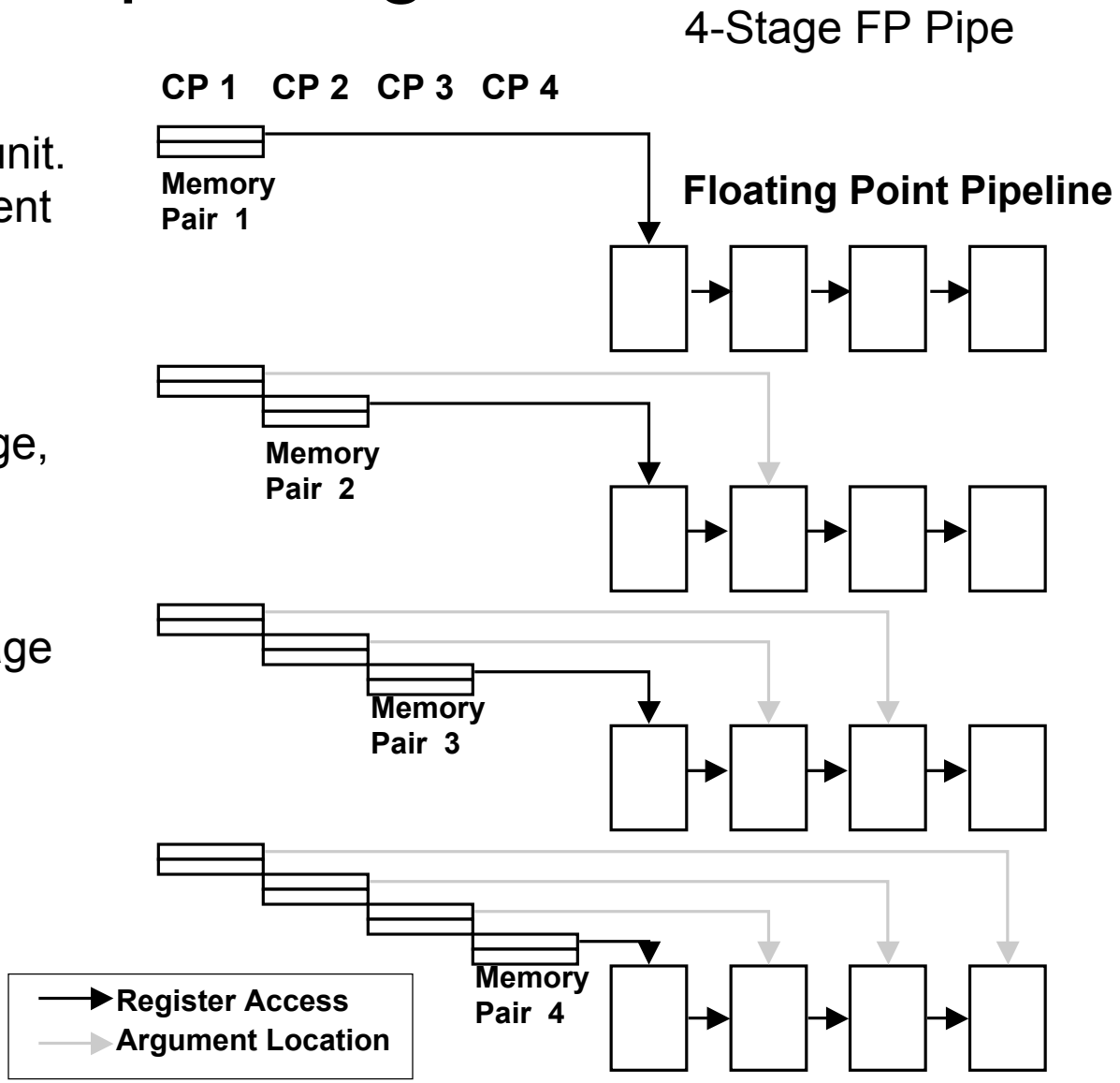
Pipeline

A serial multistage functional unit. Each stage can work on different sets of independent operands simultaneously.

After execution in the final stage, first result is available.

Latency = # of stages * CP/stage

CP/stage is the same for each stage and usually 1.



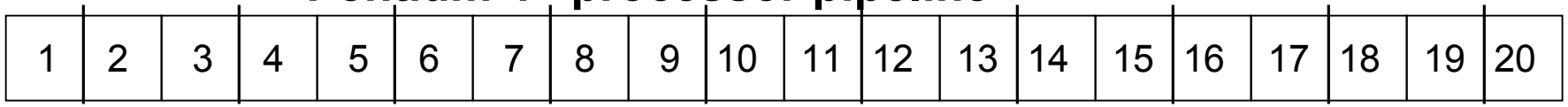
TACC

Branch Prediction

- The “instruction pipeline” is all of the processing steps (also called segments) that an instruction must pass through to be “executed”.
- Higher frequency machines have a larger number of segments.
- Branches are points in the instruction stream where the execution may jump to another location, instead of executing the next instruction.
- For repeated branch points (within loops), instead of waiting for the loop to branch route outcome, it is predicted.

Pentium III processor pipeline

Pentium 4 processor pipeline



Misprediction is more “expensive” on Pentium 4’s.