Introduction to (High-Performance, Parallel, ...) Computing

CS406 - CS531

Sabancı University

High-Performance Computing

- **High-performance computing (HPC)** refers to systems that,
  - through a combination of processing capability and storage capacity,
  - can **rapidly** solve difficult computational problems
  - across a diverse range of scientific, engineering, and business fields.

IN SHORT

understand your architecture

work on your problem/code/algorithms/data structures

solve the problem on the architecture with minimum execution time/energy consumption

Topic Overview

Part 0:

- Back then…

1800s

The first time the term "Computer" appeared in the New York Times was in May 2, 1892; the ad by the US Civil Service Commission stated:

"A Computer Wanted. [...] The examination will include the subjects of algebra, geometry, trigonometry, and astronomy."

1930s
Alan Turing

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

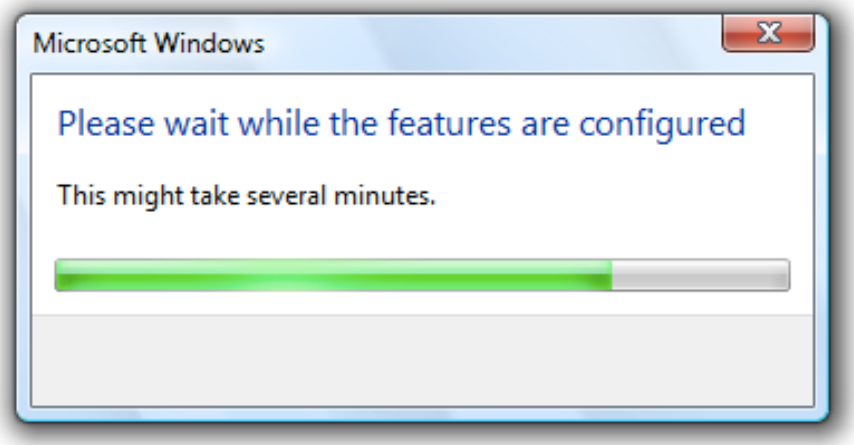
By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

1936

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

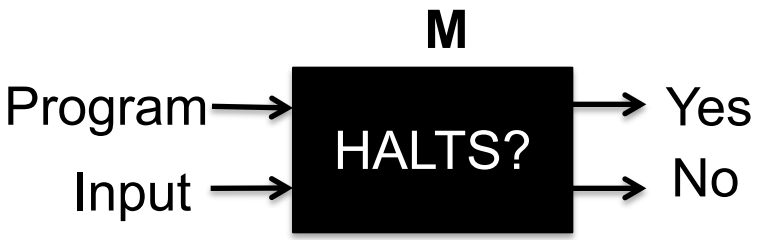
1930s
Alan Turing

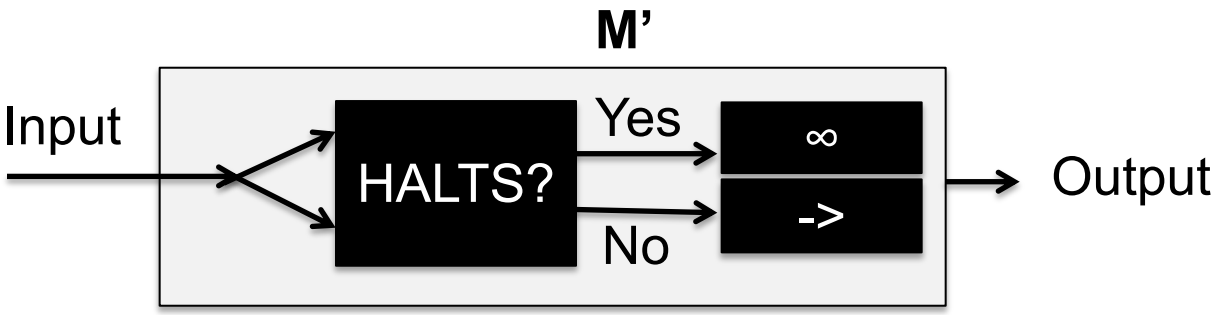


A potentially not halting program

In computability theory, **the Halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.

1930s Alan Turing





What happens when **M'** is fed by itself?

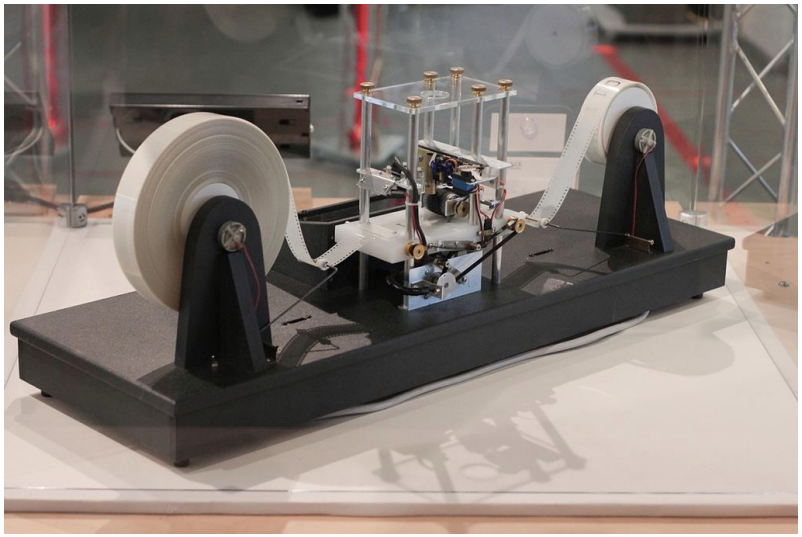
```
def g():
    if halts(g):
        loop_forever()
```

1930s
Alan Turing

Case 1: M' halts on input M'. Hence, by the correctness of the HALT, HALT returns true on input M', M'. Hence, M' loops forever on input M'. Contradiction.

Case 2: M' loops forever on M'. Hence, by the correctness of the HALT, HALT returns false on input M', M'. Hence, program M' halts on input M'. Contradiction.

1930s
Alan Turing



A Turing machine

In computability theory, a system of data-manipulation rules (such as a computer's instruction set, a programming language, or a cellular automaton) is said to be **Turing complete** or **computationally universal** if it can be used to simulate any single-taped Turing machine.

1930s
Alan Turing

Following Hopcroft & Ullman (1979, p. 148), a (one-tape) Turing machine can be formally defined as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Γ is a finite, non-empty set of *tape alphabet symbols*;
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*, that is, the set of symbols allowed to appear in the initial tape contents;
- Q is a finite, non-empty set of *states*;
- $q_0 \in Q$ is the *initial state*;
- $F \subseteq Q$ is the set of *final states* or *accepting states*. The initial tape contents is said to be *accepted* by M if it eventually halts in a state from F.
- $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. If δ is not defined on the current state and the current tape symbol, then the machine halts;[19] intuitively, the transition function specifies the next state transited from the current state, which symbol to overwrite the current symbol pointed by the head, and the next head movement.

(Wikipedia)

1939-1945
World War II


Turing

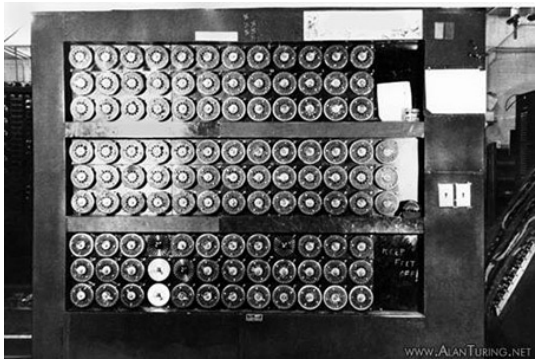

Welchman


Flowers

BAD GUYS

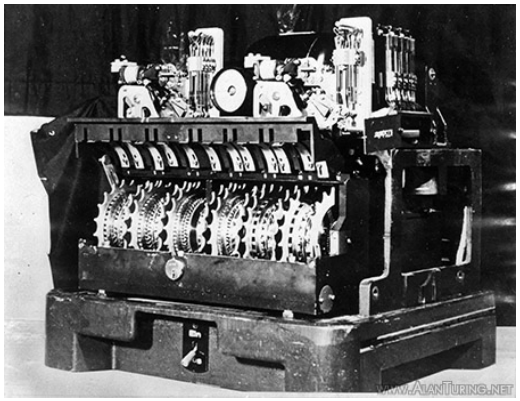
Enigma

vs.



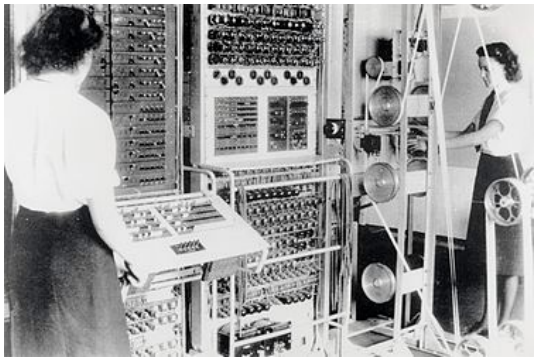
Bombe

GOOD GUYS



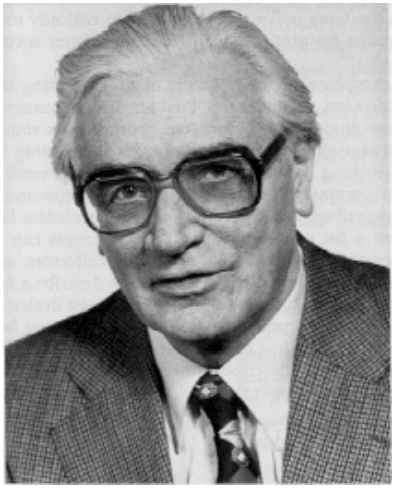
A Lorenz Cipher Machine

vs.

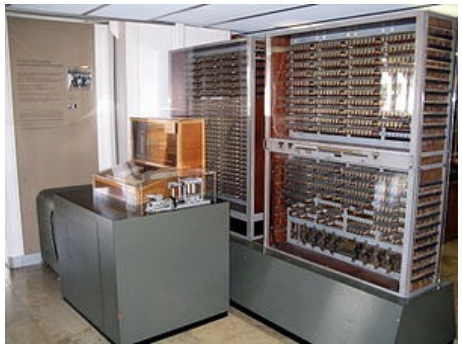

Colossus (1943)

1936-38 Z1 1939-40 Z2 1941… Z3…

(TURING COMPLETE)



Konrad Zuse



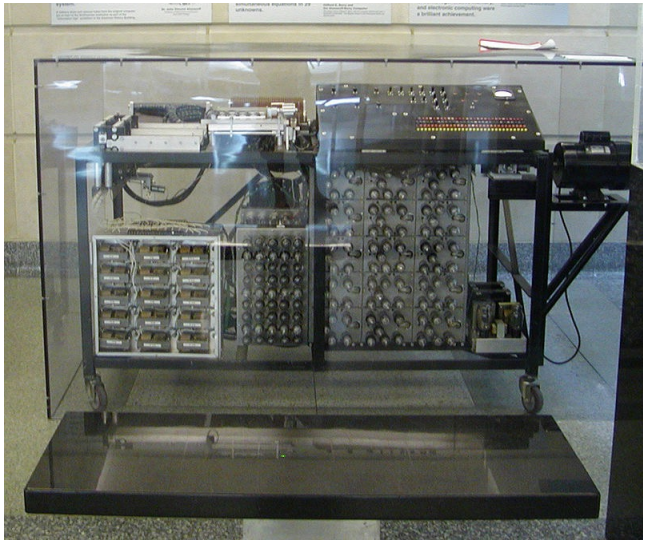
Zuse Z3 replica on display at Deutsches Museum in Munich

Average calculation speed: addition – 0.8 seconds,
multiplication – 3 seconds
Arithmetic unit: Binary floating point, 22 bit,
add, subtract, multiply, divide, square root
Data memory: 64 words with a length of 22 bits
Program memory: Punched celluloid tape
Input: Decimal floating-point numbers
Frequency: 5.3 Hertz
Power consumption: Around 4,000 watts
Weight: Around 1 tonne (2,200 lb)

1939-42
The Atanasoff-Berry Computer (ABC)

Atanasoff

Berry



Atanasoff–Berry computer replica at Durham Center, Iowa State University

Three important ideas:

- Using binary digits to represent all numbers and data
- Performing all calculations using electronics rather than wheels, ratchets, or mechanical switches
- Organizing a system in which computation and memory are separated.

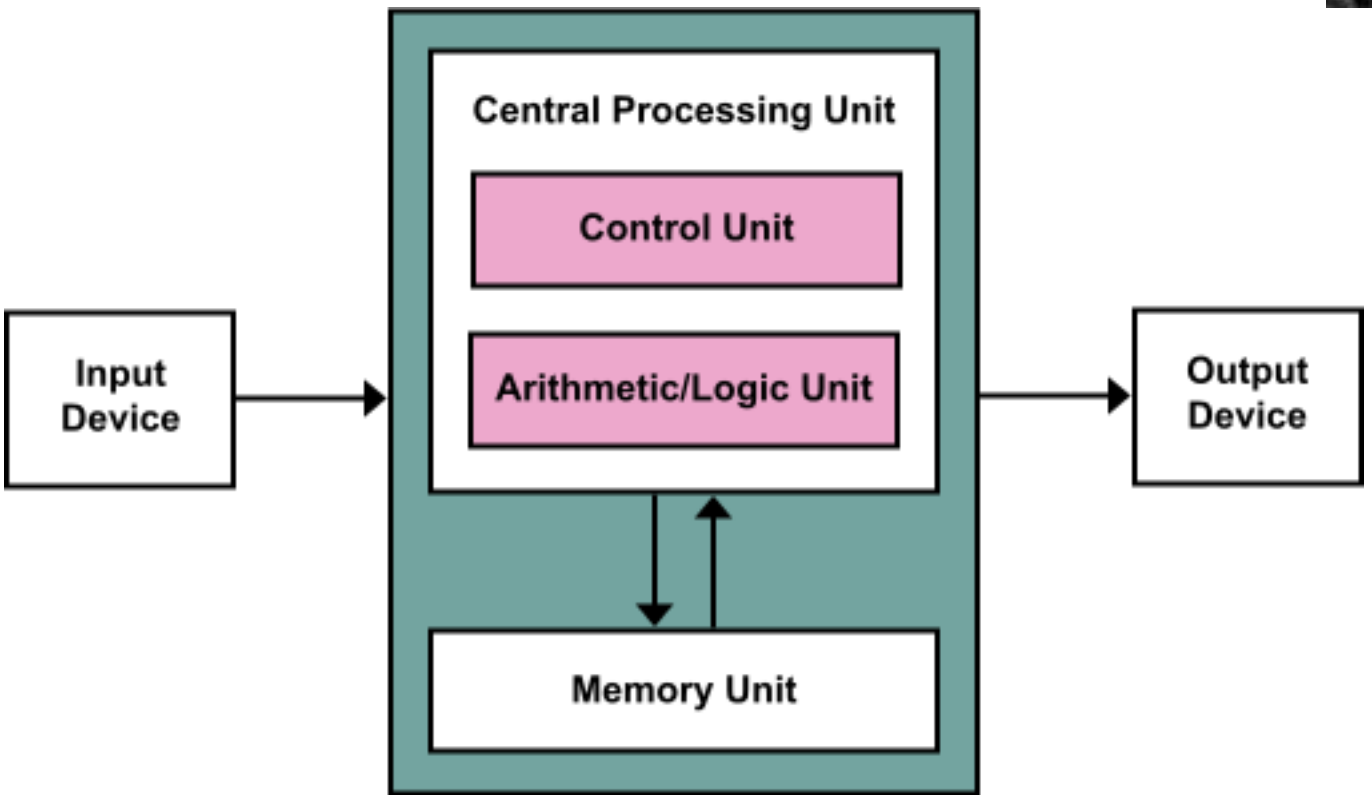
1945-1960s

ENIAC (1945)

IBM 5Mb HD (1956)

1945
Von Neumann Architecture

John von Neumann



1965
Moore's Law

Gordon Moore

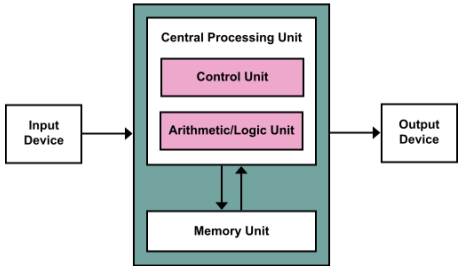
Cramming More Components onto Integrated Circuits

GORDON E. MOORE, LIFE FELLOW, IEEE

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65 000 components on a single silicon chip.

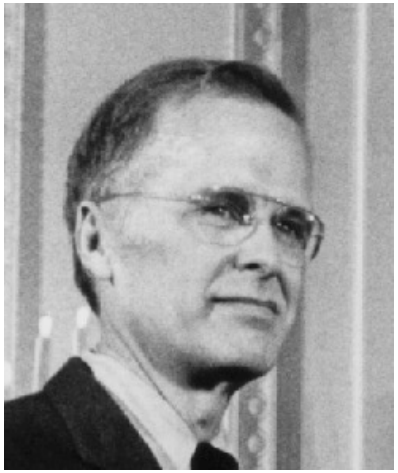
1975: Semiconductor complexity would continue to double annually until about 1980 after which it would decrease to a rate of doubling approximately every two years

1977
von Neumann Bottleneck



- A systems bottleneck: in that the bandwidth between Central Processing Units (CPUs) and Random-Access Memory is much lower than the speed at which a typical CPU can process data internally.

- An 'intellectual bottleneck': programmers at the time spent a lot of time thinking about code optimization to stop 'lots of words' being pushed back and forth between CPU and RAM.



John Backus

"… programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it." [1977 Turing Award Lecture]

Codes

See the array sum code with perf

(VonNeumann)

Topic Overview

Part 1:

- Implicit Parallelism: Trends in Microprocessor Architectures
- Limitations of Memory System Performance

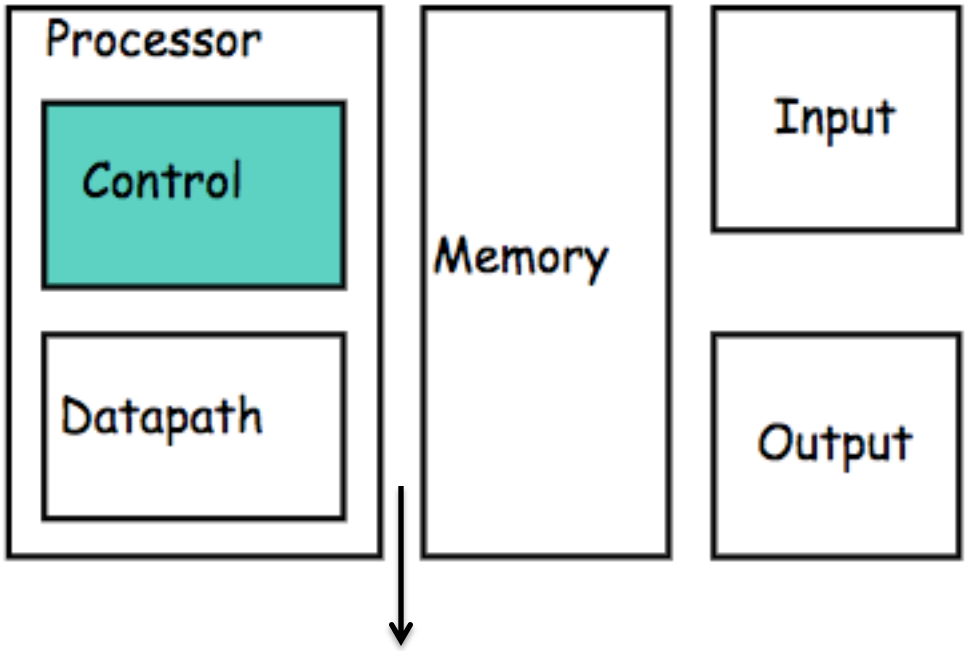
Scope of Parallelism

- Conventional computers coarsely comprise of a **processor**, and a **memory system**.
  - Each of these components present significant performance bottlenecks.
  - Parallelism addresses each of these components in significant ways.

- Computer = Processor + Memory system
- Processor = Control + Datapath
- Control can be considered as a finite state machine
  - **Input** = **Instructions** + **Datapath conditions**
  - **Outputs** = **Transfer control signals, ALU operation codes**
- Datapath = Functional units (ALU, multipliers, dividers, etc.) + Registers (Program counters, shifters, storage) + Buses

Scope of Parallelism

- Conventional architectures coarsely comprise of a **processor**, **memory system**, and the **datapath**.
  - Each of these components present significant performance bottlenecks.
  - Parallelism addresses each of these components in significant ways.



The (physical) von Neumann Bottleneck

Scope of Parallelism

- Different applications utilize different aspects of parallelism - e.g.,
  - data intensive applications utilize **high aggregate throughput,**
  - server applications utilize **high aggregate network bandwidth**, and
  - scientific applications typically utilize **high processing** and **memory system performance.**

- It is important to understand each of these performance bottlenecks.

- In this lecture, we mainly focus on the ones requiring **data and compute intensive applications**

Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
  - Higher levels of device integration have made available a large number of transistors.
- The question of how best to utilize these resources is important.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.

Pipelining and Superscalar Execution

- Pipelining overlaps various stages of instruction execution to achieve performance.

- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.

Pipelining and Superscalar Execution

Five stages of RISC pipeline

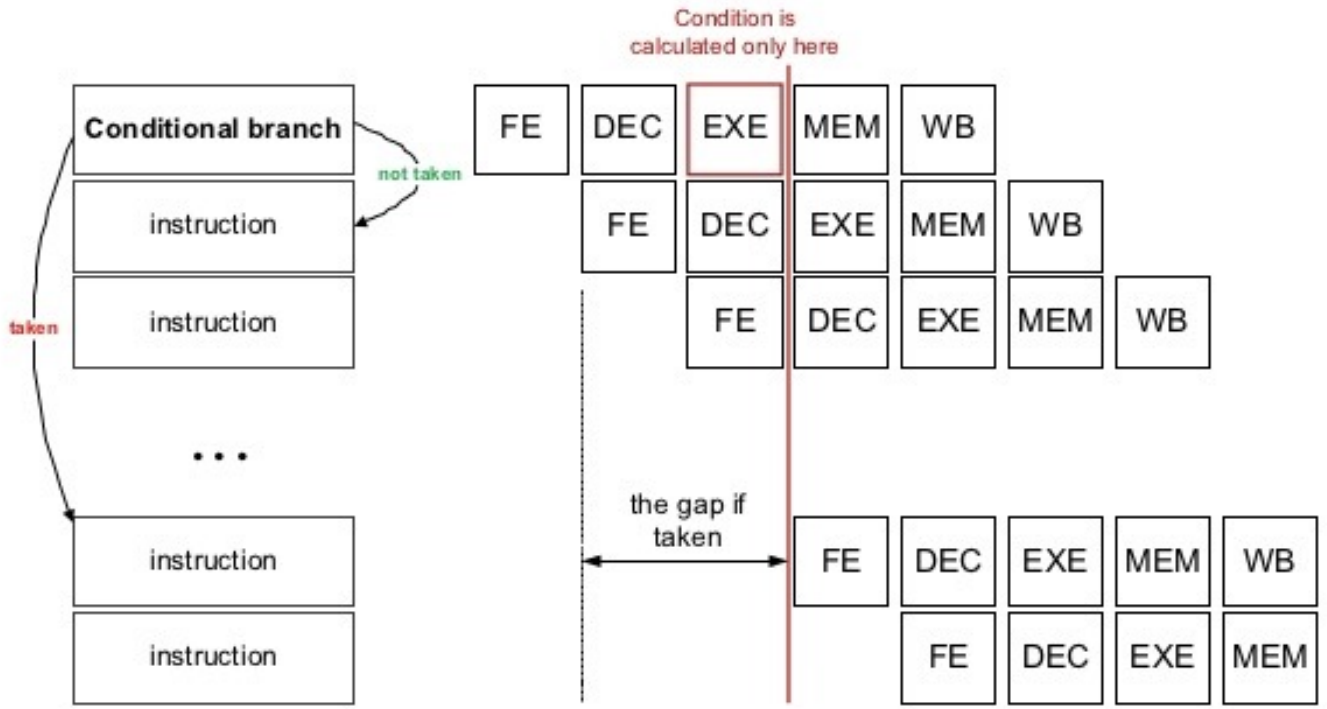
- **Fetch:** instruction is being fetched from the memory.
- **Decode:** we decode the instruction and fetch the source operands
- **Execute:** the computer performs the operation specified by the instruction
- **Memory:** if there is any data that needs to be accessed, it is done in the memory stage
- **Write:** if we need to store the result in the destination location, it is done during the writeback stage

Pipelining and Superscalar Execution

- Pipelining, however, has several limitations.
- The speed of a pipeline is eventually limited by the slowest stage.
- For this reason, conventional processors rely on very deep pipelines (e.g., 20 stage pipelines in Pentium processors).
- However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.
- The penalty of a mis-prediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

Pipelining and Superscalar Execution

- One simple way of alleviating these bottlenecks is to use multiple pipelines.
- The question then becomes one of selecting these instructions.

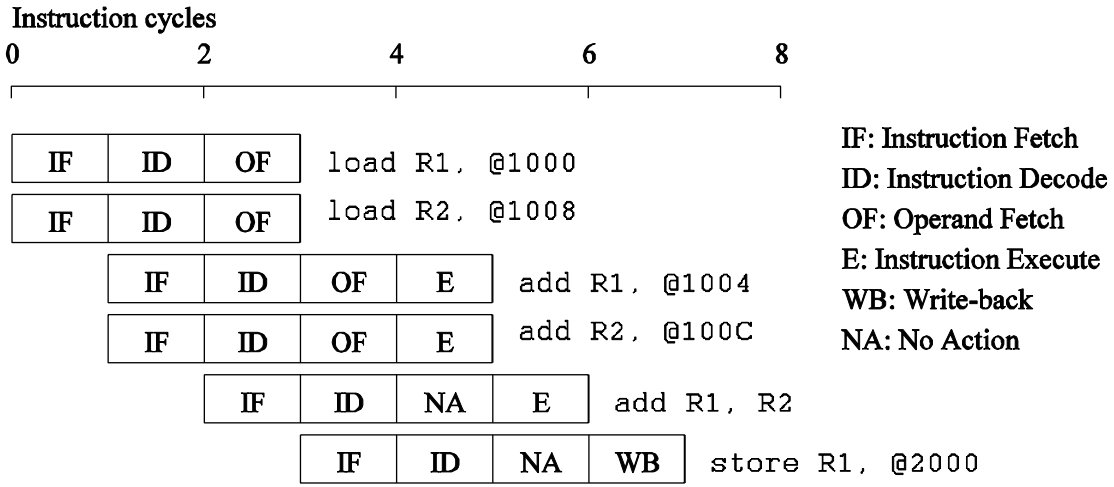


Superscalar Execution: An Example

```
1. load R1, @1000        1. load R1, @1000        1. load R1, @1000
2. load R2, @1008        2. add R1, @1004         2. add R1, @1004
3. add R1, @1004         3. add R1, @1008         3. load R2, @1008
4. add R2, @100C         4. add R1, @100C         4. add R2, @100C
5. add R1, R2            5. store R1, @2000       5. add R1, R2
6. store R1, @2000                                6. store R1, @2000
```

(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.

Example of a two-way superscalar execution of instructions.

Superscalar Execution

- Scheduling of instructions is determined by a number of factors:
  - **True Data Dependency**: The result of one operation is an input to the next.
  - **Resource Dependency**: Two operations require the same resource.
  - **Branch Dependency**: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.

Issue Mechanisms

- In the simpler model, instructions can be issued only in the order in which they are encountered.

- That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle.

- This is called ***in-order* issue**.
  - Performance of in-order issue is generally limited.

In-order Problems
(from Dr. Onur Mutlu's slides)

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2
ADD   R3 ← R3, R1
ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5
```

```
LD    R3 ← R1 (0)
ADD   R3 ← R3, R1
ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5
```

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed

- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)

Issue Mechanisms

- In a more aggressive model, instructions can be issued ***out of order***.

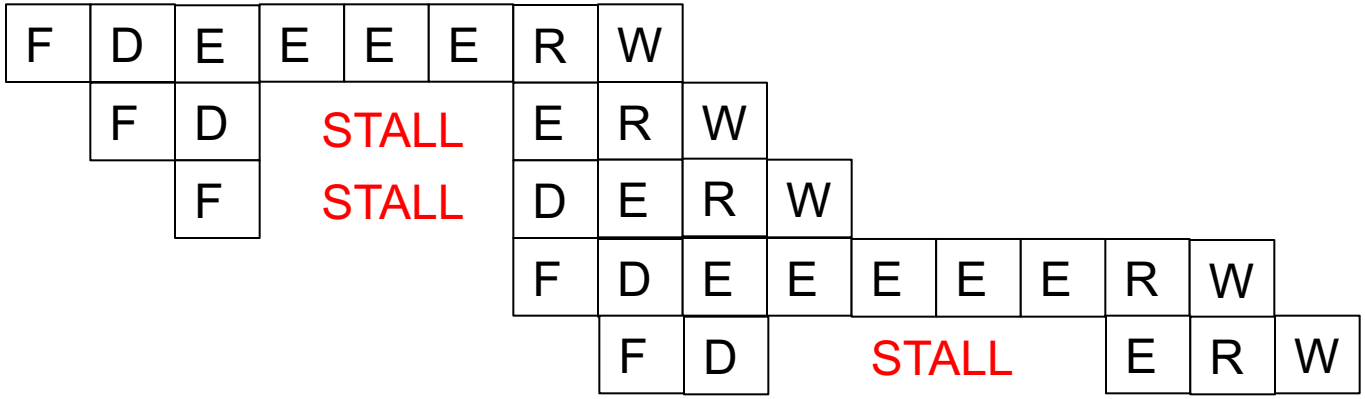
- In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled.

- This is also called dynamic issue.

Out-of-order Dispatch (from Dr. Onur Mutlu's slides)

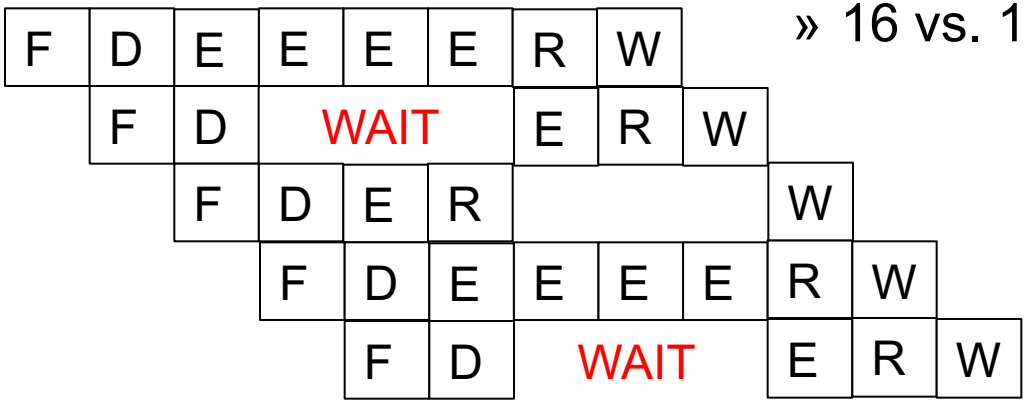
- In order dispatch + precise exceptions:

IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5



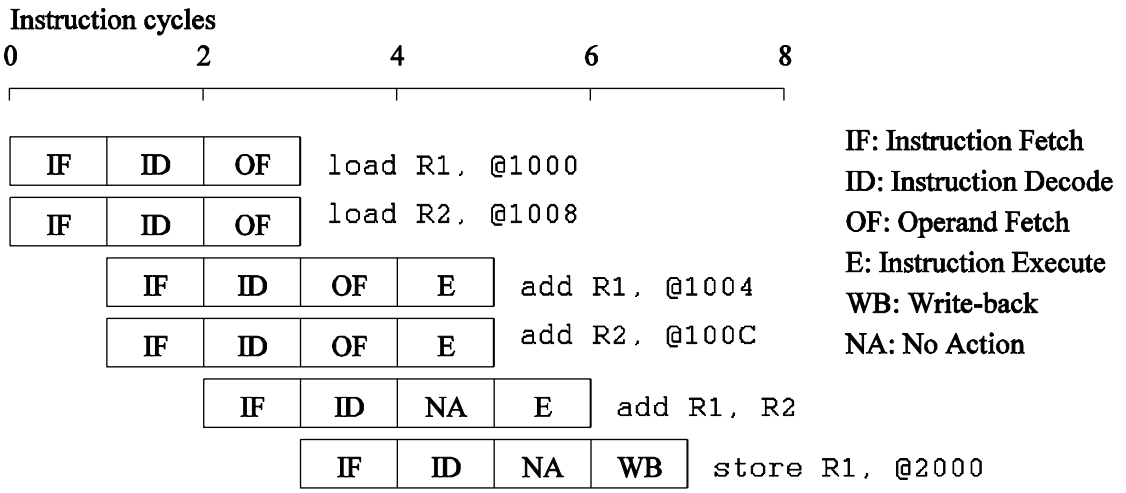
- Out-of-order dispatch + precise exceptions:

» 16 vs. 12 cycles

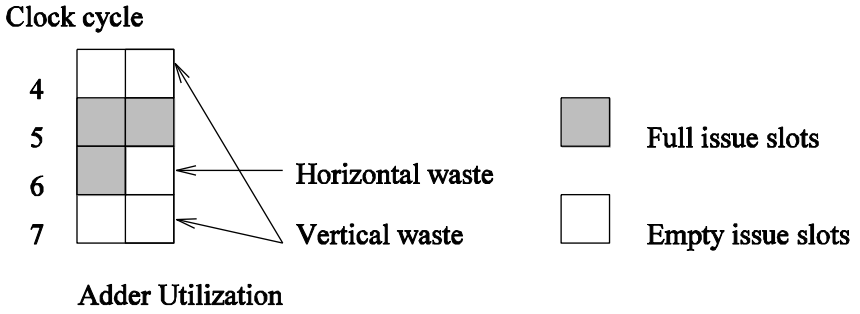


Superscalar Execution: Efficiency Considerations

- Not all functional units can be kept busy at all times.
- If during a cycle, no functional units are utilized, this is referred to as **vertical waste**.
- If during a cycle, only some of the functional units are utilized, this is referred to as **horizontal waste.**



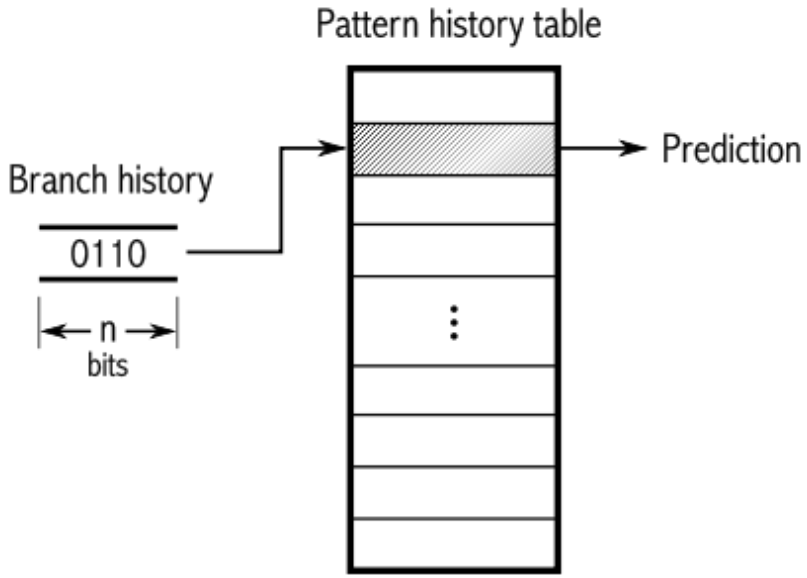
(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

Instructions, Pipelining, Reordering, etc…

- A typical branch predictor



Branch predictor

Instructions, Pipelining, Reordering, etc…

```
for (unsigned c = 0; c < arraySize; ++c) {
  data[c] = std::rand() % 256;
}
```

```
long long sum = 0;
for (unsigned i = 0; i < 100000; ++i) {
  for (unsigned c = 0; c < arraySize; ++c) {
    if (data[c] >= 128)
      sum += data[c];
  }
}
```

Instructions, Pipelining, Reordering, etc…

```
for (unsigned c = 0; c < arraySize; ++c) {
  data[c] = std::rand() % 256;
}

std::sort(data.begin(), data.end());

long long sum = 0;
for (unsigned i = 0; i < 100000; ++i) {
  for (unsigned c = 0; c < arraySize; ++c) {
    if (data[c] >= 128)
      sum += data[c];
  }
}
```

Codes

See the branch prediction code with perf

(Branches)

Also check vf_calls.cpp in the ArchBenchmarks/branch_prediction folder

Instructions, Pipelining, Reordering, etc…

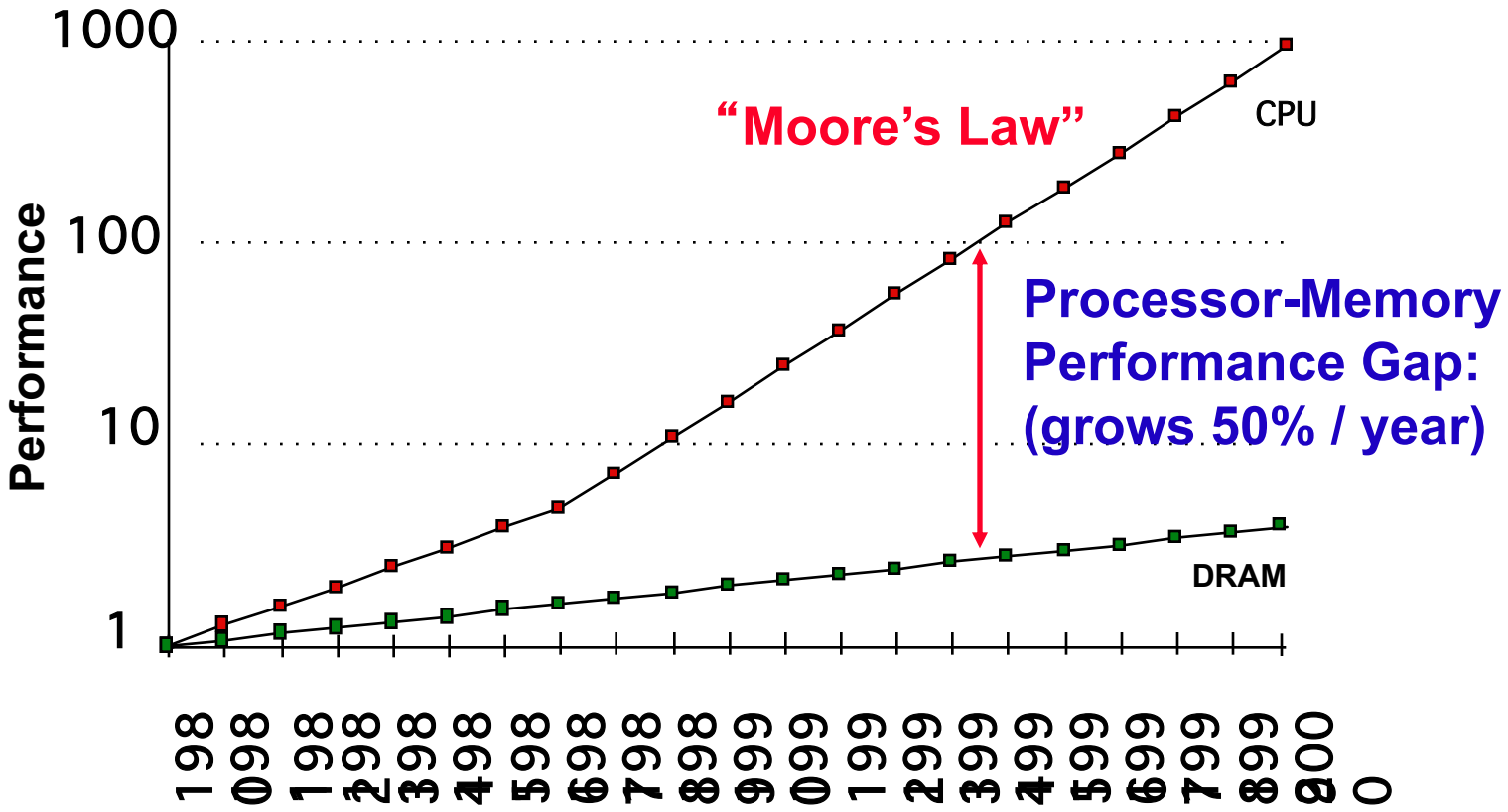
- **Compilers can solve some of these problems**
  - See the example in the codes with –O1-3 instead of –O0
  - But this is NOT always true …

- More on branch_prediction example
  - Basic conditional move example:

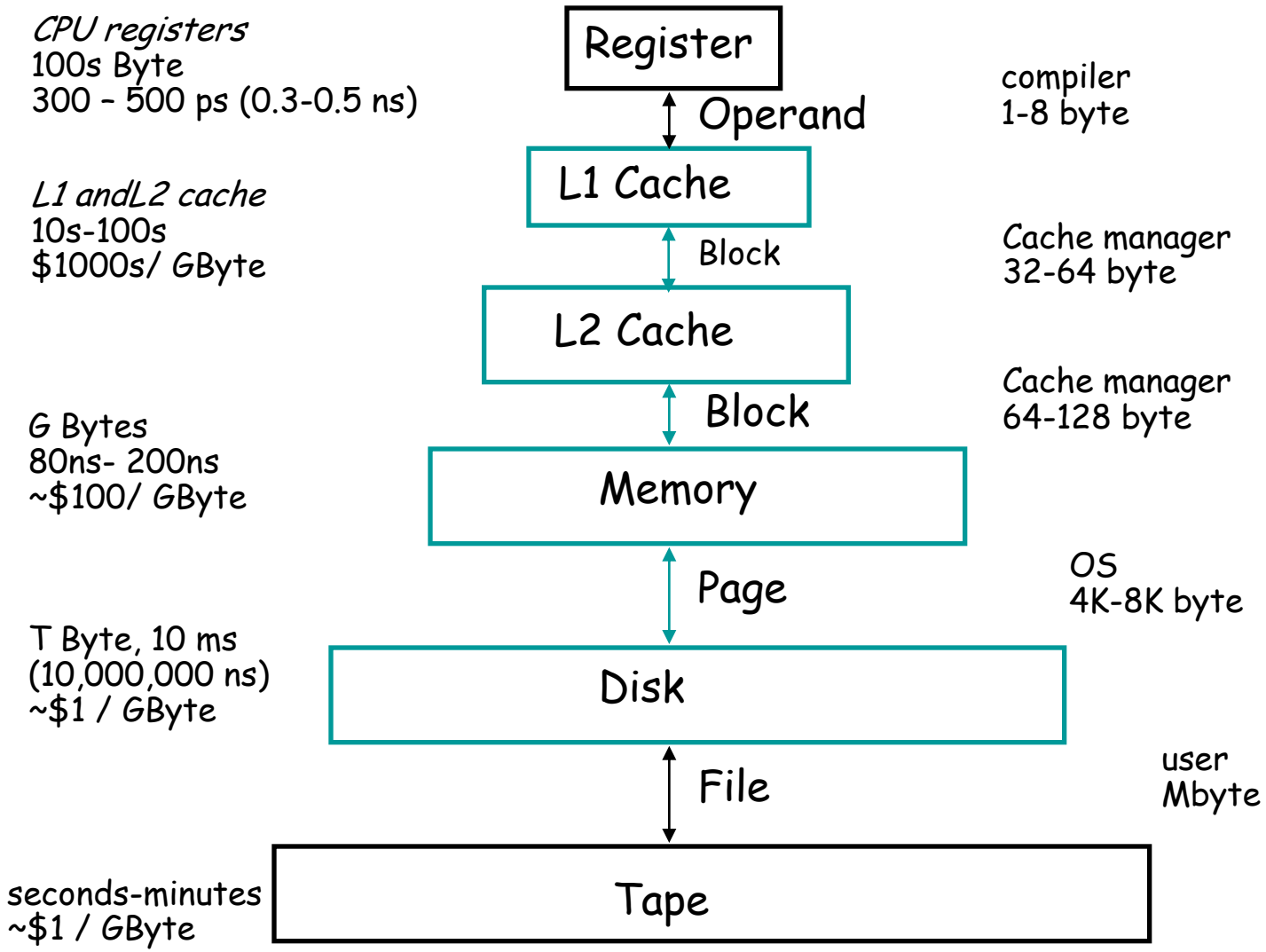
```
...
cmp eax, ebx
jne skip
    mov ecx, edx
skip:
```

```
…
cmp eax, ebx
cmove ecx, edx
```

Memory System Performance: The Memory Wall



The Memory System: Hierarchy



Limitations of Memory System Performance

- **Memory system, and not processor speed, is often the bottleneck for many applications**.
- Memory system performance is largely captured by two parameters, **latency** and **bandwidth**.
  - **Latency** is the time from the issue of a memory request to the time the data is available at the processor.
  - **Bandwidth** is the rate at which data can be pumped to the processor by the memory system.

Memory System Performance: Bandwidth and Latency

- **It is very important to understand the difference between latency and bandwidth.**
- Consider the example of a fire-hose (or an assembly line). If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want **immediate response** from the hydrant, it is important to reduce **latency**.
- If you want to **fight big fires**, you want high **bandwidth**.

Memory Latency: An Example

- Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and hence is capable of executing four instructions (in total) in each cycle of 1 ns.
- The following observations follow:
  - The peak processor rating is
    - **4 GFLOPS.**
  - Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made (i.e., for each word), the processor must wait each data for
    - **100 cycles**

Memory Latency: An Example

milli	m	1000^{-1}	10^{-3}	0.001
micro	μ	1000^{-2}	10^{-6}	0.000 001
nano	n	1000^{-3}	10^{-9}	0.000 000 001

- On the above architecture, consider the problem of computing a dot-product of two vectors.
  - A dot-product computation performs one multiply-add (2 flops) on a single pair of vector elements, i.e., each floating-point operation requires
    - **one**

    data fetch.
  - It follows that the peak speed of this computation is limited to one floating point operation every
    - **100 ns, or a speed of**
    - **10 MFLOPS**

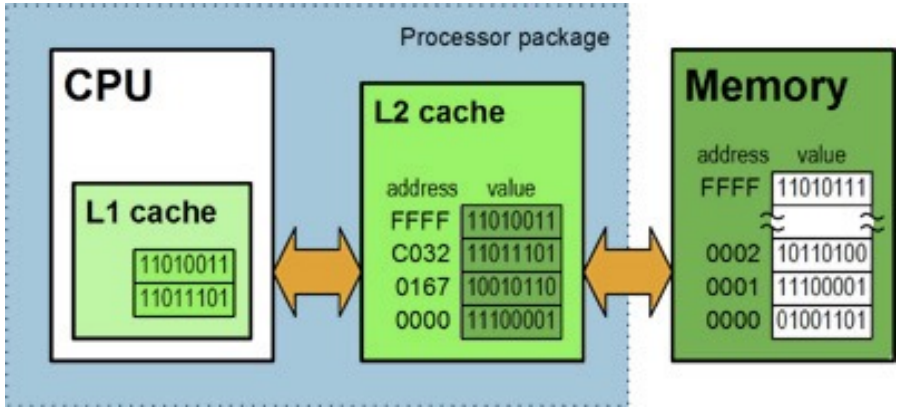
    which is a very small fraction of the peak processor rating!

Latency Numbers You Should Know

Operation	ns	µs	ms	note
L1 cache reference	0.5 ns			
Branch mispredict	5 ns			
L2 cache reference	7 ns			14x L1 cache
Mutex lock/unlock	25 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	3 µs		
Send 1K bytes over 1 Gbps network	10,000 ns	10 µs		
Read 4K randomly from SSD*	150,000 ns	150 µs		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 µs		
Round trip within same datacenter	500,000 ns	500 µs		
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 µs	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	10,000 µs	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 µs	20 ms	80x memory, 20X SSD
Send packet CA -> Netherlands -> CA	150,000,000 ns	150,000 µs	150 ms	

Improving Effective Memory Latency Using Caches

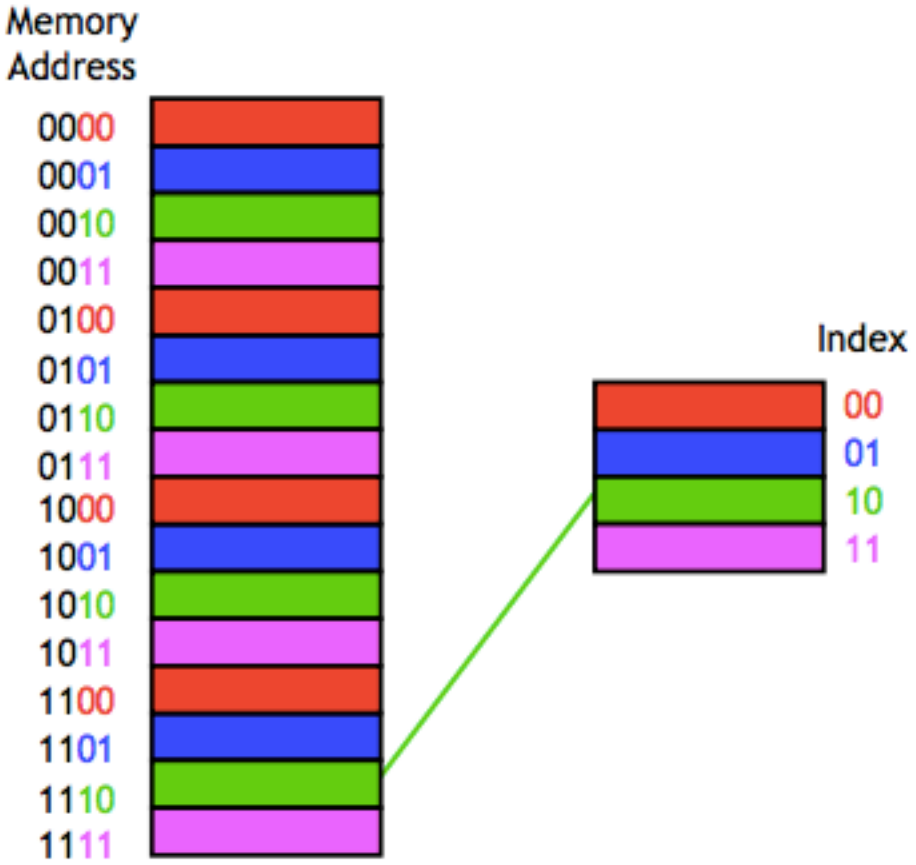
- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.



- Can we improve the dot-product example with a cache?

Interlude: How Caches Work?

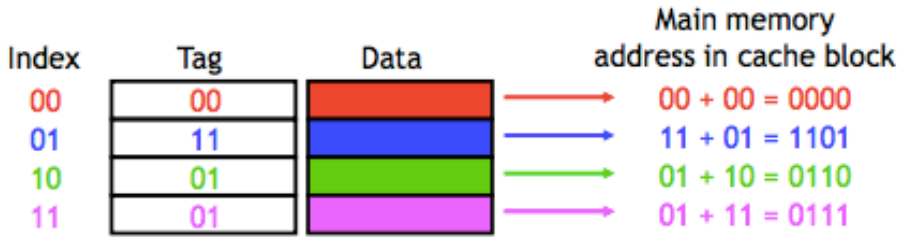
- A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block.
  - For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks)



Interlude: How Caches Work?

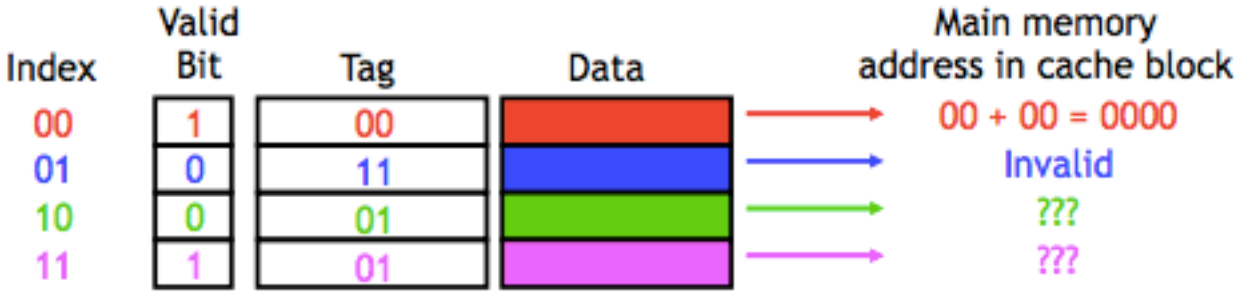
- We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.

- We can tell exactly which addresses of main memory are stored in the cache.



Interlude: How Caches Work?

- When started, the cache is empty and does not contain valid data. We should account for this by adding a **valid bit** for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



Improving Effective Memory Latency Using Caches

- The fraction of data references satisfied by the cache is called the **cache hit ratio** of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.

Impact of Caches: Example

Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions 32×32. So, the cache is large enough to store matrices A and B, as well as the result matrix C.

What is the peak performance **without the cache**?

10 MFlops

Impact of Caches: Example (continued)

- The following observations can be made about the problem (100ns per word, 1 ns cycle, 4 instructions/cycle):

milli	m	1000^{-1}	10^{-3}	0.001
micro	μ	1000^{-2}	10^{-6}	0.000 001
nano	n	1000^{-3}	10^{-9}	0.000 000 001

  - Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately
    - **200 µs.**

  - Multiplying two $n \times n$ matrices takes $2n^3$ operations which can be performed in
    - **16K cycles (or 16 µs) at four instructions per cycle.**

Impact of Caches: Example (continued)

- The following observations can be made about the problem:
  - The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e.,
    - **200 + 16 µs.**
  - This corresponds to a peak computation rate of
    - **64K FLOP/216 µs or 303 MFLOPS**.

Impact of Caches

- Repeated references to the same data item is related to **temporal locality**.
- In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. This asymptotic difference makes the above example particularly desirable for caches.
  - Temporal locality in the example does not have an impact since the cache can store everything.
  - If this is not the case, it depends on the implementation (the intellectual part Von Neumann Bottleneck).
- **Data reuse is critical for cache performance.**

We will come to that later.

Impact of Memory Bandwidth

- **Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.**
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The underlying system takes
  - *l* time units (where *l* is the latency of the system) to deliver
  - *b* units of data (where *b* is the block size).

Impact of Memory Bandwidth: Example

- Consider the same setup as before (no cache), except in this case, the block size is 4 words instead of 1 word. We repeat the dot-product computation in this scenario:
  - Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in
    - **200 cycles**.
  - This is because a single memory access fetches four consecutive words in the vector.
  - This corresponds to a FLOP every
    - **25 ns, for a peak speed of**
    - **40 MFLOPS.**

Impact of Memory Bandwidth

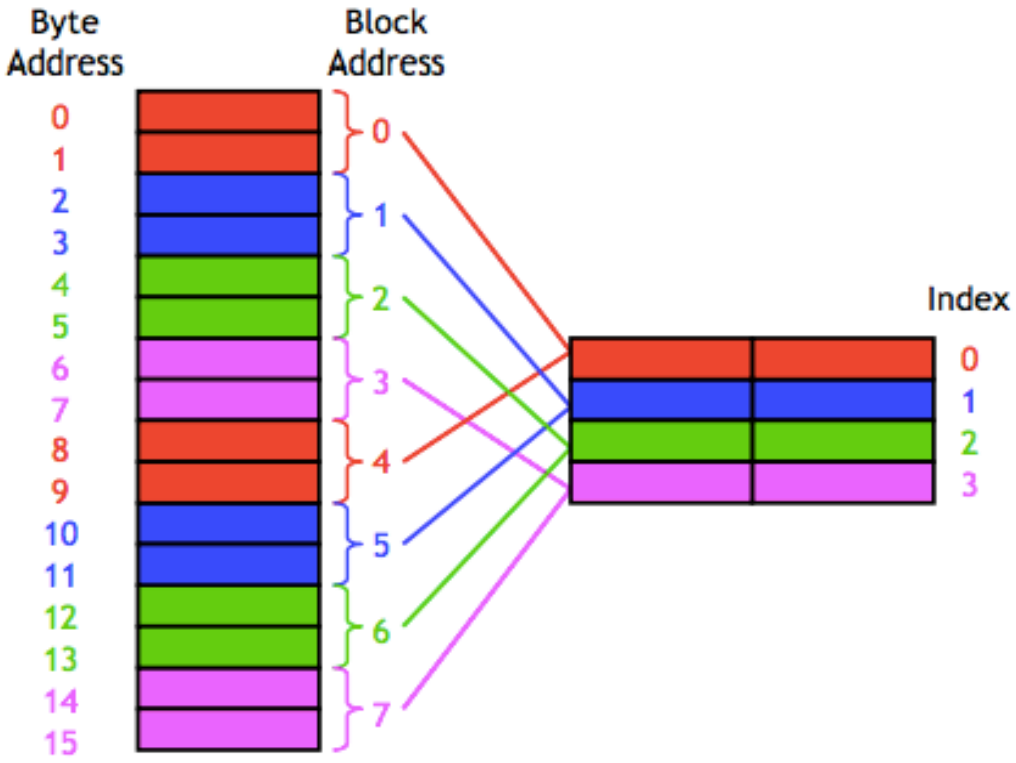
- **It is important to note that increasing block size does not change latency of the system**.
- Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128/256 bits) connected to multiple memory banks.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.

Impact of Memory Bandwidth

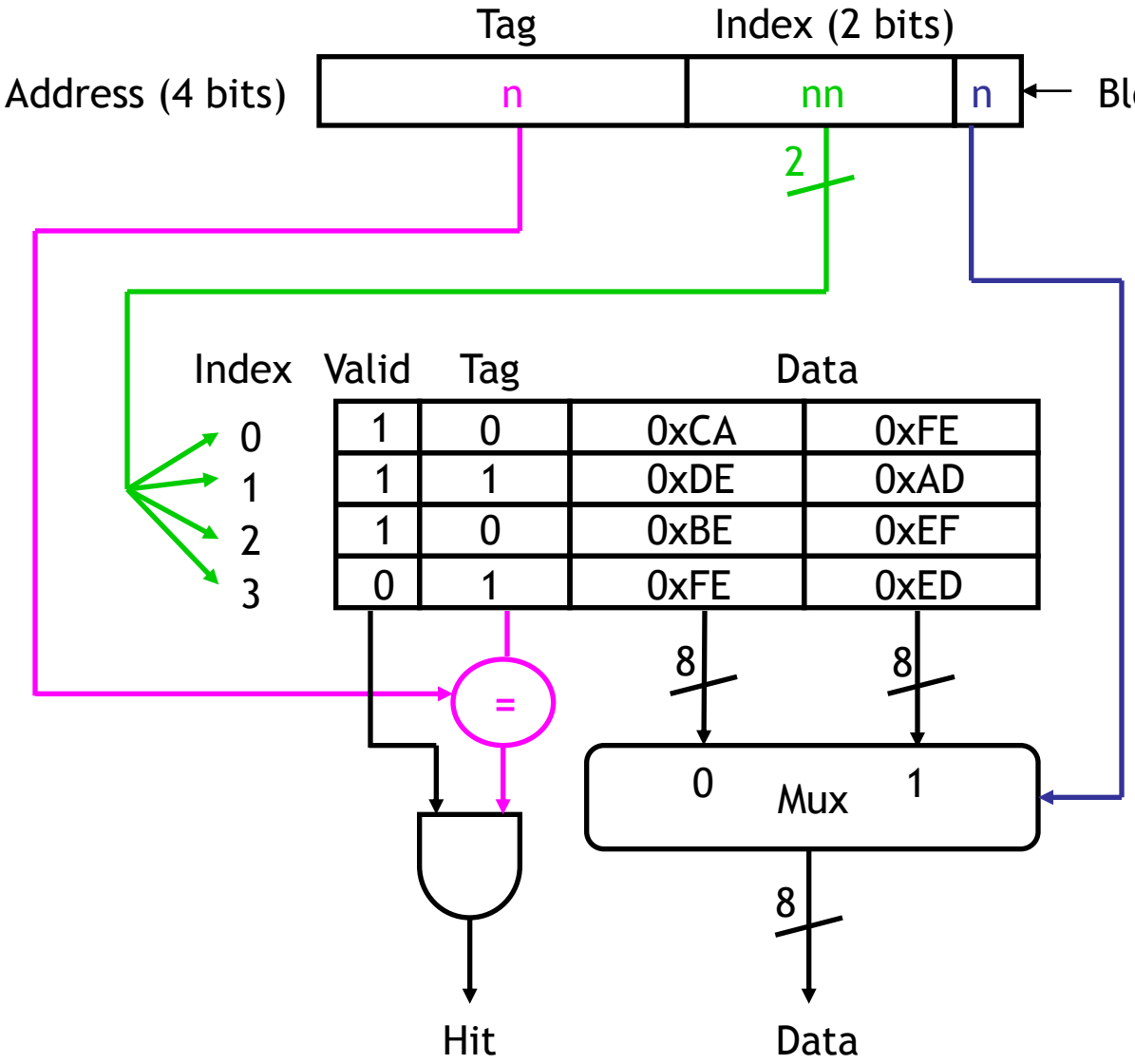
- The example clearly illustrated how increased bandwidth results in higher peak computation rates.
- The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions (**spatial locality of reference**).
- If we take a **data-layout centric view**, computations must be reordered to enhance spatial locality of reference.

Impact of Memory Bandwidth

- One-byte cache blocks don't take advantage of spatial locality, which predicts that an access to one address will be followed by an access to a nearby address.
- What we can do is make the cache block size larger than one byte.



Impact of Memory Bandwidth

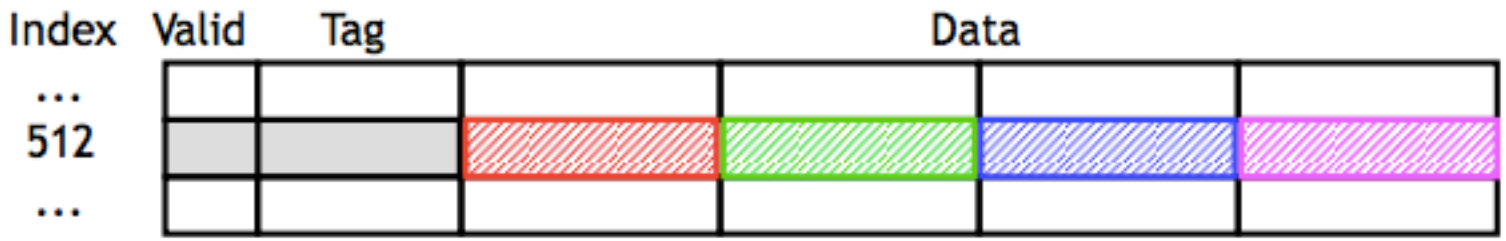


For the addresses below, what is the byte read from the cache (or is there a miss)?

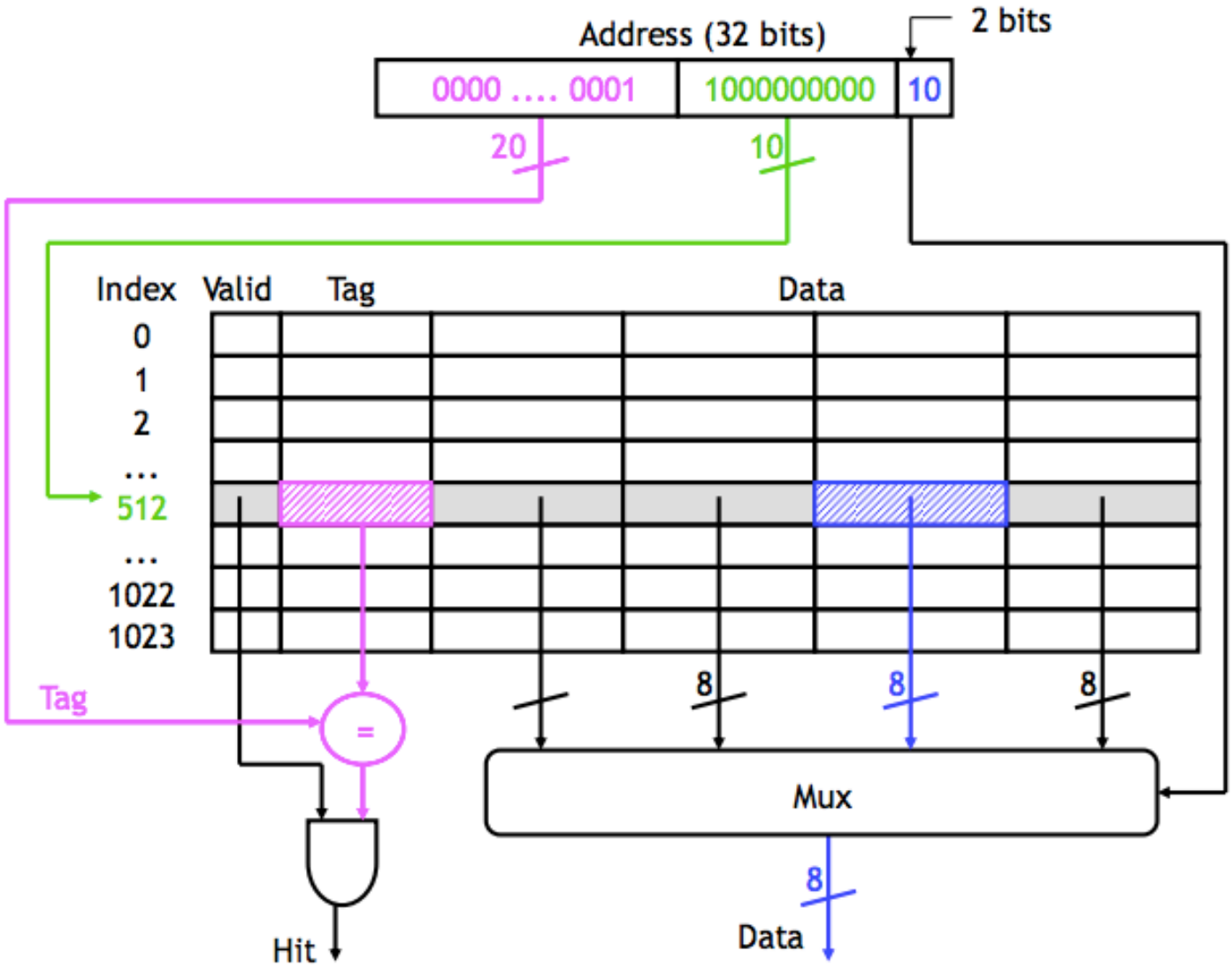
- 1010
- 1110
- 0001
- 1101

Impact of Memory Bandwidth

Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147



Impact of Memory Bandwidth



Impact of Memory Bandwidth: Example

Consider the following code fragment:

```
for (i = 0; i < 1000; i++)
     for (j = 0; j < 1000; j++)
          column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix b into a vector `column_sum`.

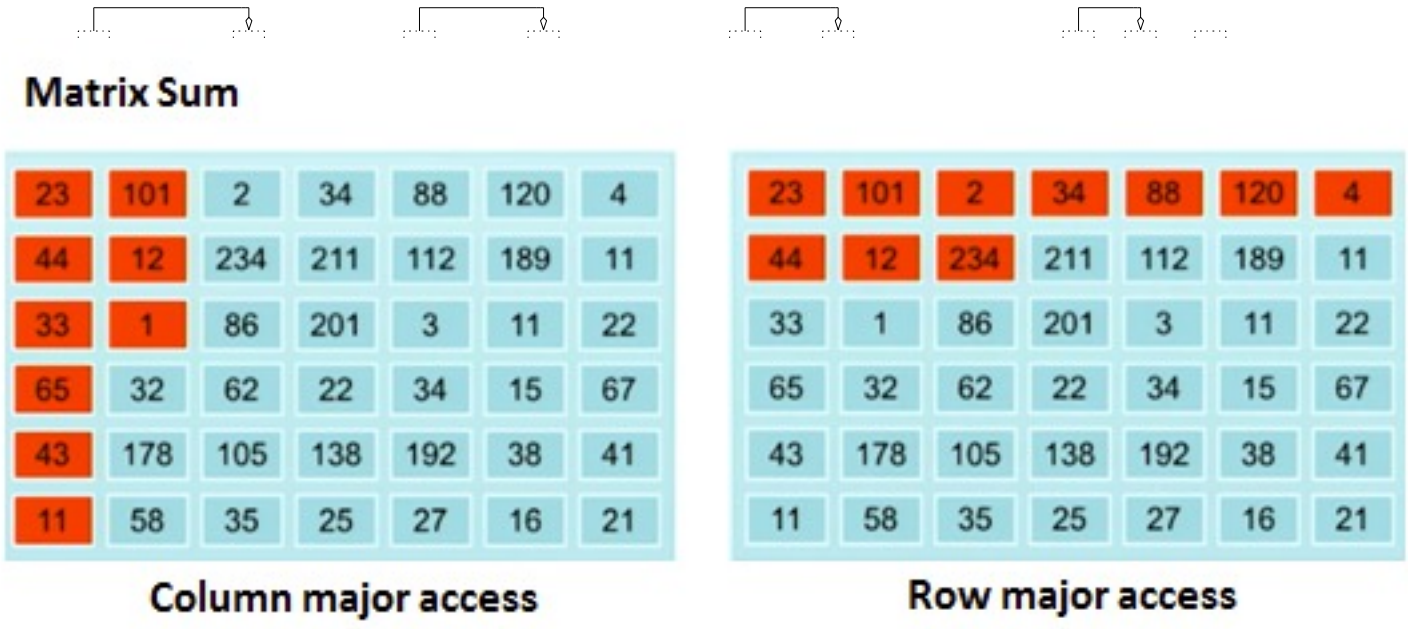
Codes

See the column sum codes

(Spatial)

Impact of Memory Bandwidth: Example

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order.
- The strided access results in very poor performance.



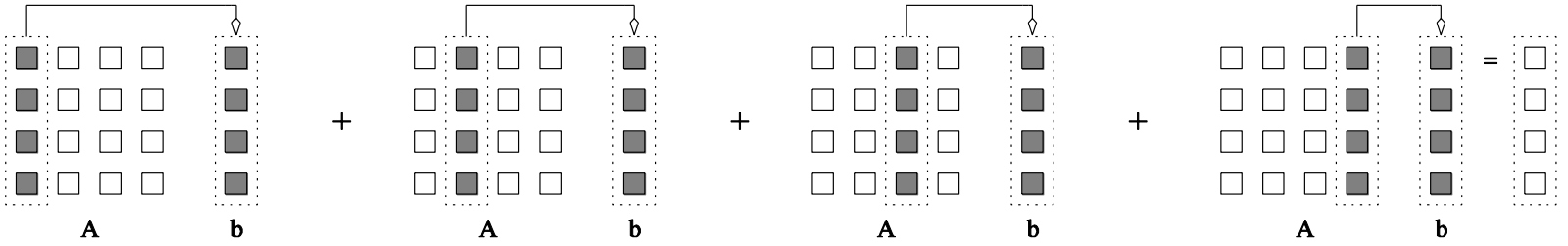
Impact of Memory Bandwidth: Example

We can fix the above code as follows:

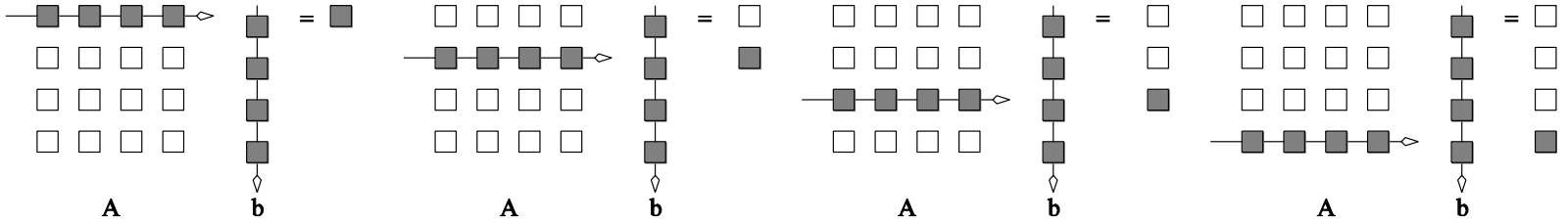
```
for (j = 0; j < 1000; j++)
     for (i = 0; i < 1000; i++)
          column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

Impact of Memory Bandwidth: Another Example – Matrix x Vector



(a) Column major data access



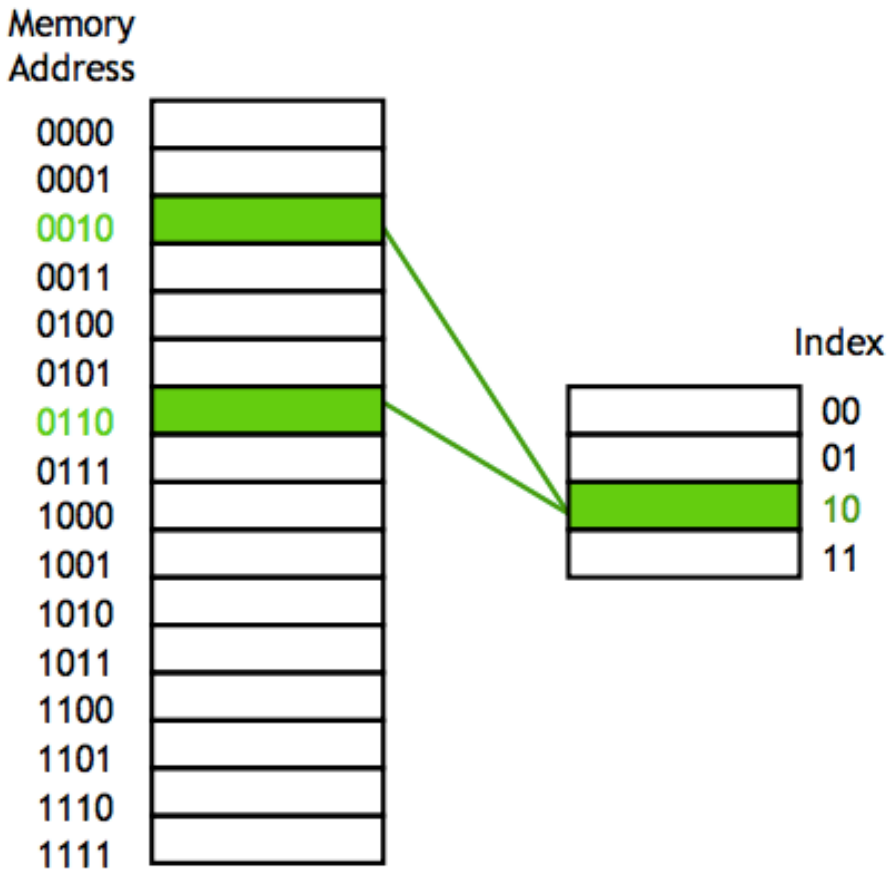
(b) Row major data access.

Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

Interlude: Set-associativity

The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.

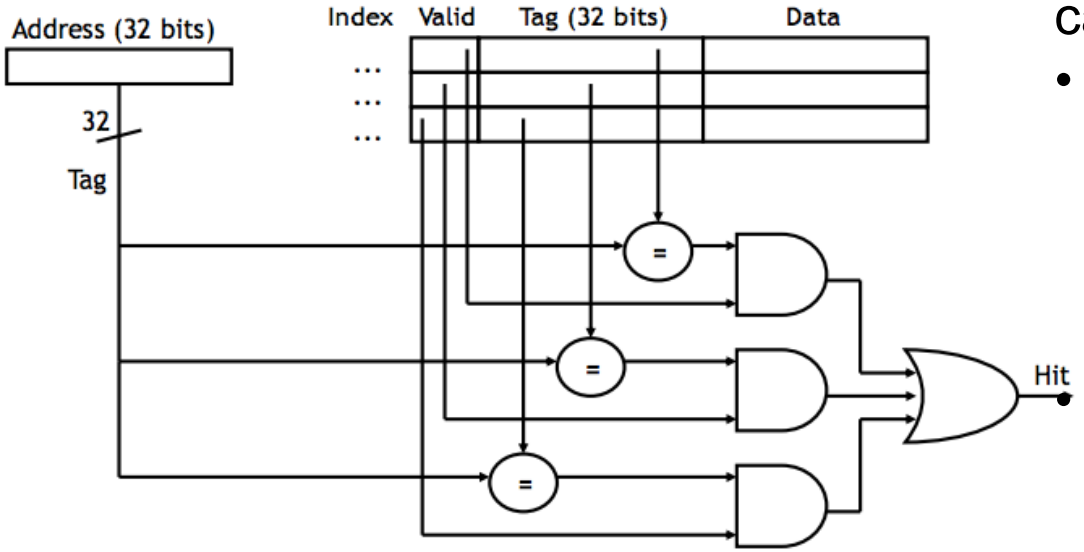
But, what happens if a program uses addresses 2, 6, 2, 6, 2, …?



Interlude: Set-associativity

A fully associative cache permits data to be stored in any cache block, instead of forcing each memory address into one particular block.

- When data is fetched from memory, it can be placed in any place.
- We'll never have a conflict between two or more memory addresses which map to a single cache block



However, a fully associative cache is expensive to implement.

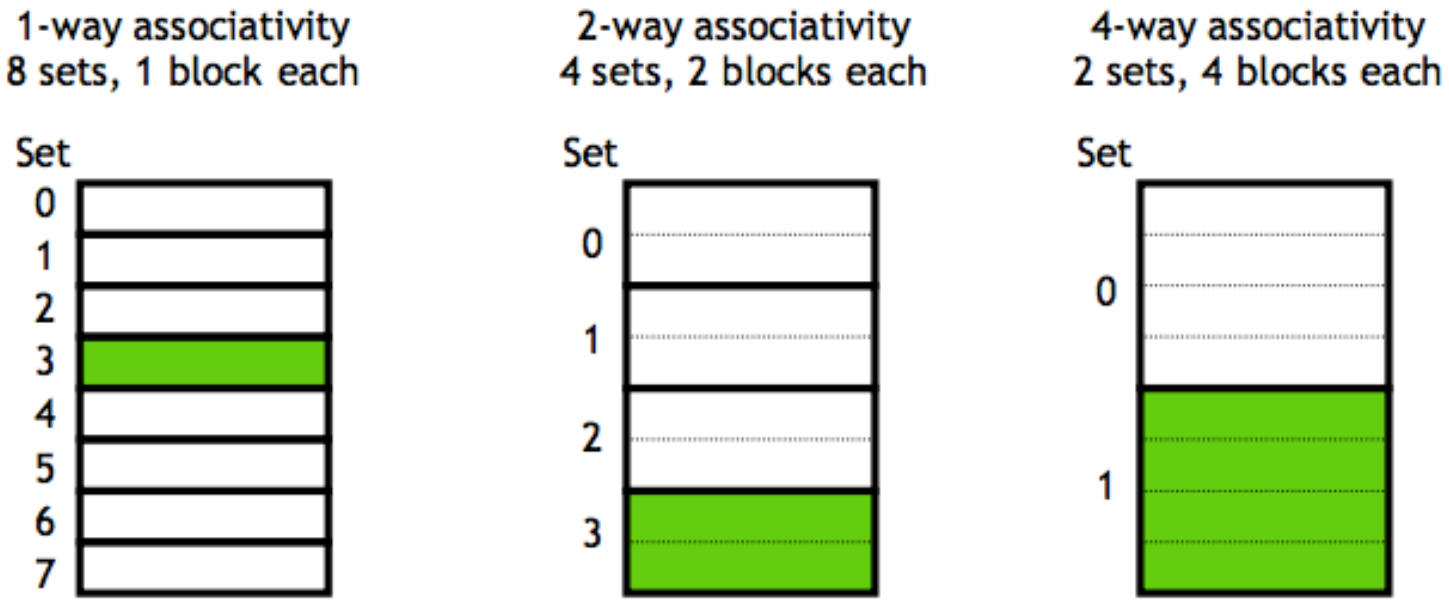
- There is no index field in the address anymore, the entire address must be used as the tag, increasing the total cache size.
  Data could be anywhere in the cache, so we must check the tag of every cache block.
  That's a lot of comparators!

Interlude: Set-associativity

An intermediate possibility is a set-associative cache.

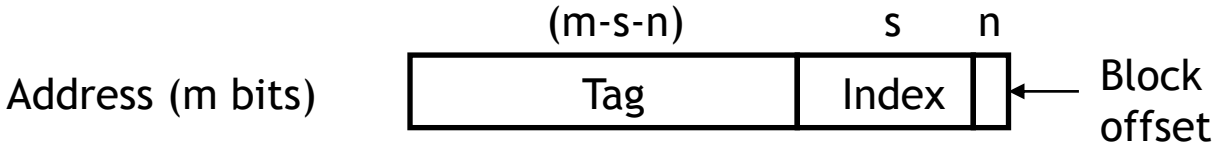
- The cache is divided into groups of blocks, called sets.
- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.

If each set has 2^x blocks, the cache is 2^x-way associative cache.



Interlude: Set-associativity

- We can determine where a memory address belongs in an associative cache in a similar way as before.
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

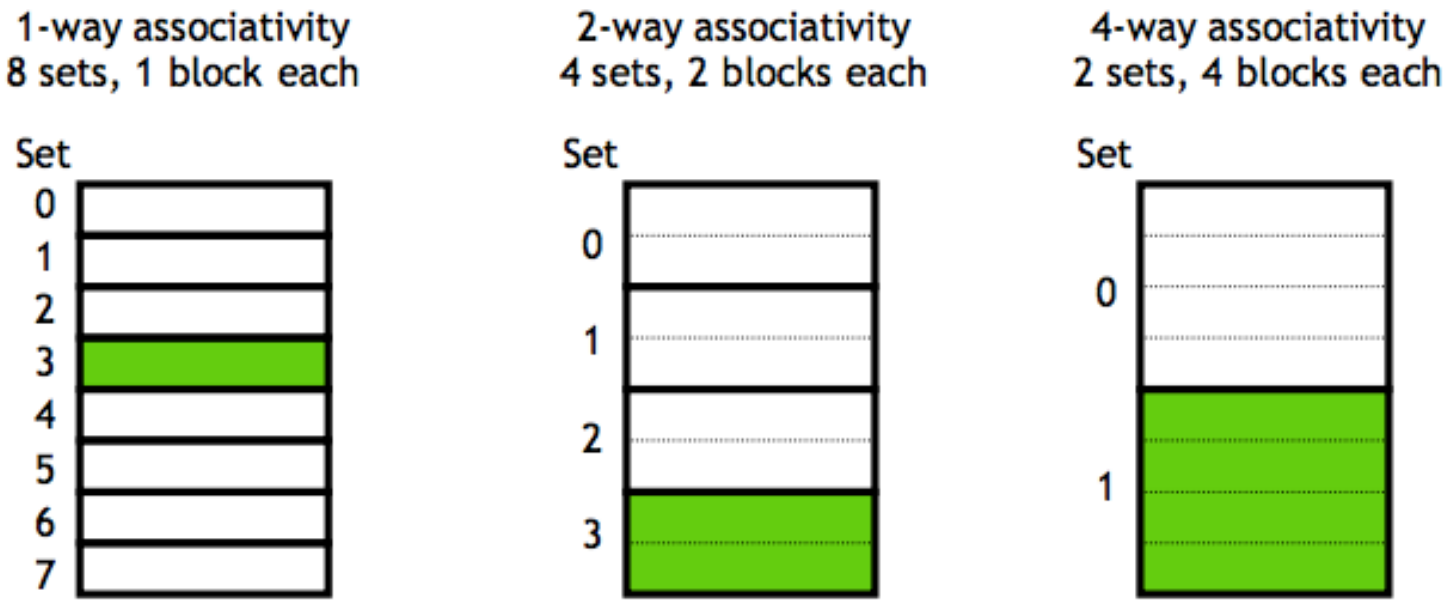
Block Offset = Memory Address mod 2^n

Block Address = Memory Address / 2^n

Set Index = Block Address mod 2^s

Interlude: Set-associativity

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in *any* block, shown in green, within the correct set.



Interlude: Set-associativity

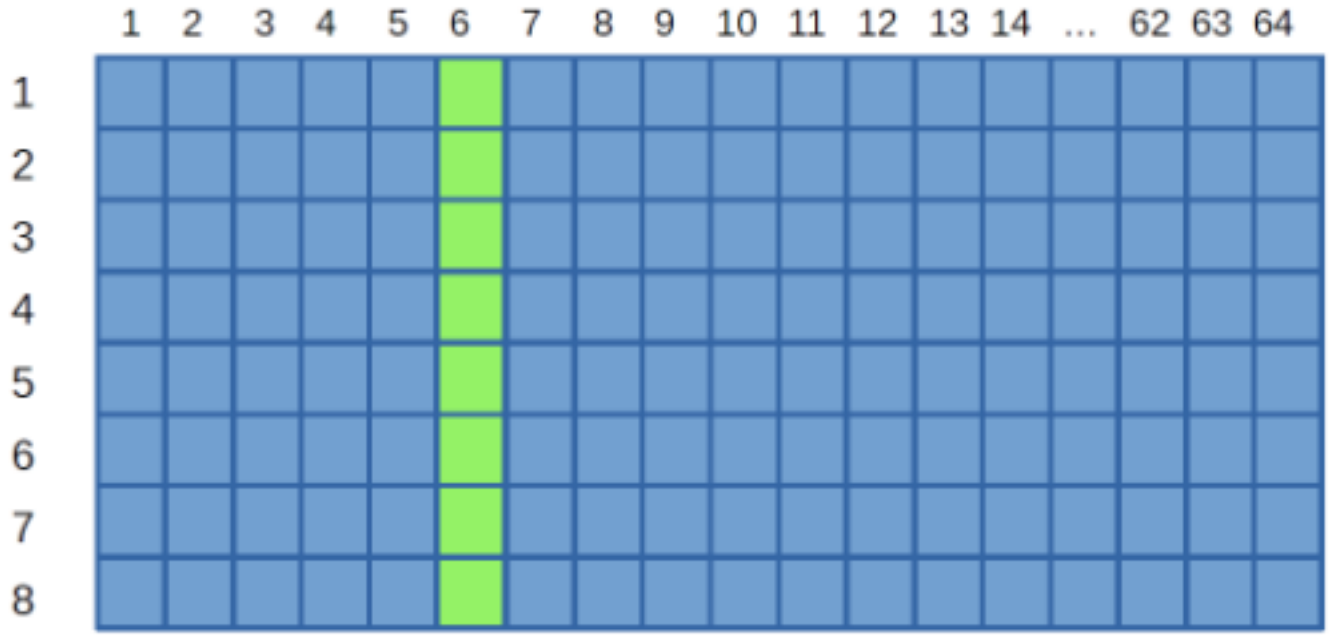
```
getconf -a | grep CACHE
```

```
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  20971520
LEVEL3_CACHE_ASSOC                 20
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Interlude: Set-associativity

- The 32KB of L1 data cache in a core can therefore be envisioned as a three-dimensional box, where:
  - Depth represents the size of a cache line, e.g., 64 bytes
  - Height represents the extent of a cache set
  - Width represents the number of sets that are available
- After doing a few quick calculations, we can find the relevant properties for the L1d cache of a core, which holds 32 KB divided into 64-byte cache lines and is 8-way set associative:
  - Bytes in L1d = 32 KB * 1024 (bytes/KB) = 32768 bytes
  - Cache lines in L1d = 32768 / (line size) = 32768 / 64 = 512
  - Number of sets = 512 / 8 = 64

Interlude: Set-associativity



Recall that each square above represents an entire cache line (64 bytes in our case).
When data at a particular address is requested, the congruence class of the address is computed, determining the cache set of the cache line containing the data.
Then the entire line is fetched into one of the eight slots for that cache set.

Codes

What is the worst performance pattern for a cache like this?

Lets see the answer in the assoc.cpp

(ArchBenchmarks/associativity)

Memory System Performance: Summary

- The series of examples presented in this section illustrate the following concepts:
  - Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
  - The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
    - Memory bound…
    - Compute bound…
  - Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

Alternative Approaches for Hiding Memory Latency

- Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:
  - we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
  - we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
- The first approach is called **prefetching** and the second **multithreading**

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program.

We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)
      c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)
  c[i] = create_thread(dot_product,get_row(a, i), b);
```

Multithreading for Latency Hiding: Example

- In the code, the first instance of this function accesses a pair of vector elements and waits for them.
- In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.
- After ***l*** units of time, where ***l*** is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.
- In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.

Multithreading for Latency Hiding

- The execution schedule in the previous example is predicated upon two assumptions: the memory system is capable of servicing multiple memory requests, and the processor is capable of switching threads at every cycle.

Prefetching for Latency Hiding

- **The same problem**: Misses on loads cause programs to stall.
- **A different solution**: Why not advance the loads so that by the time the data is actually needed, it is already there!

- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

Prefetching for Latency Hiding

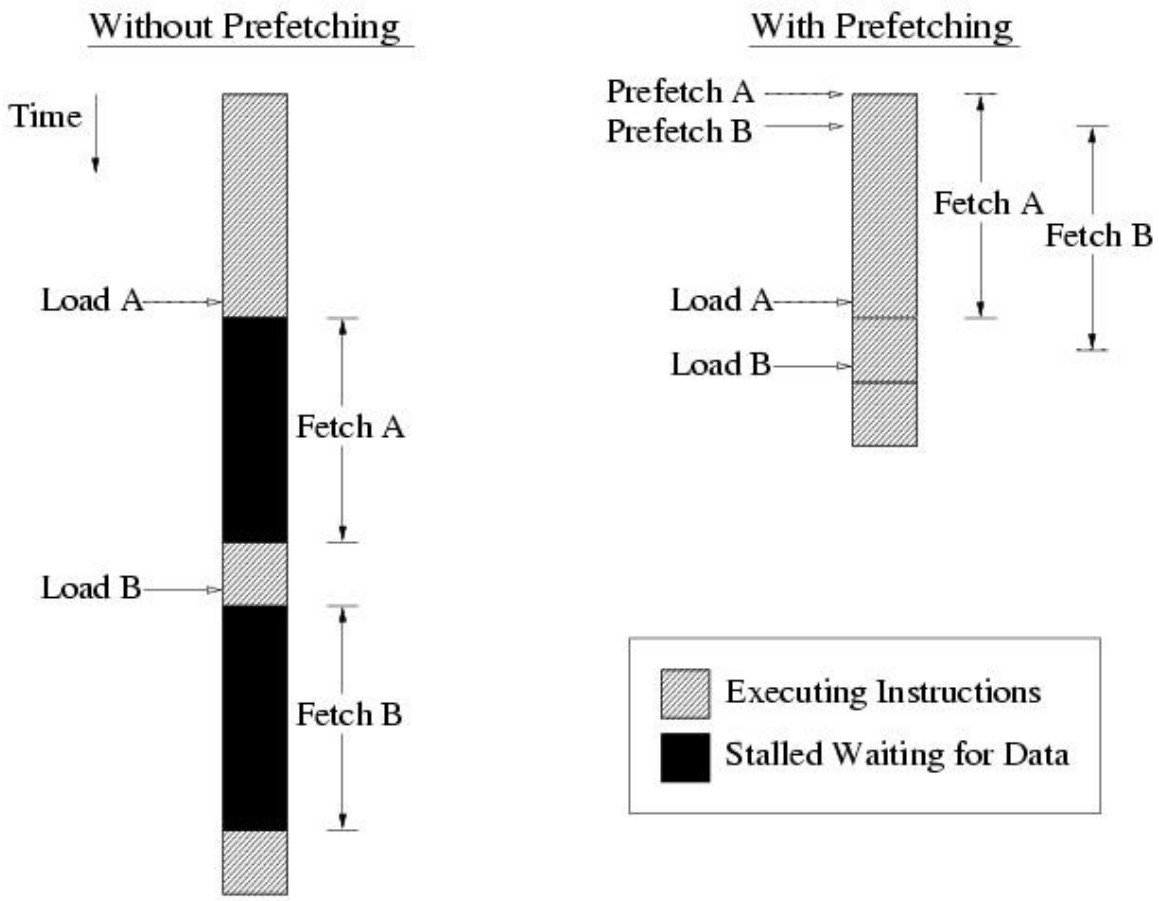


Figure 1.4: Illustration of how prefetching tolerates memory latency.

Example: Vector Product

- No prefetching

```
for (i = 0; i < N; i++) {
   sum += a[i]*b[i];
}
```

- Assume each cache block holds 4 elements

  → 2 misses/4 iterations

- Simple prefetching

```
for (i = 0; i < N; i++){
   fetch (&a[i+1]);
   fetch (&b[i+1]);
   sum += a[i]*b[i];
}
```

- **Problem**
  - **Unnecessary prefetch operations**

Example: Vector Product (Cont.)

- Prefetching + loop unrolling

```
for (i = 0; i < N; i+=4) {
   fetch (&a[i+4]);
   fetch (&b[i+4]);
   sum += a[i]*b[i];
   sum += a[i+1]*b[i+1];
   sum += a[i+2]*b[i+2];
   sum += a[i+3]*b[i+3];
}
```

- **Problem**
  - **First and last iterations**

```
fetch (&sum);
fetch (&a[0]);
fetch (&b[0]);
for (i = 0; i < N-4; i+=4){
   fetch (&a[i+4]);
   fetch (&b[i+4]);
   sum += a[i]*b[i];
   sum += a[i+1]*b[i+1];
   sum += a[i+2]*b[i+2];
   sum += a[i+3]*b[i+3];
}
for (i = N-4; i < N; i++)
   sum = sum + a[i]*b[i];
```

Example: Vector Product (Cont.)

- Previous assumption: prefetching 1 iteration ahead is sufficient to hide the memory latency
- When loops contain small computational bodies, it may be necessary to initiate prefetches d iterations before the data is referenced

$$d \geq \left\lceil \frac{l}{s} \right\rceil$$

- d: prefetch distance, l: avg memory latency, *s* is the estimated cycle time of the shortest possible execution path through one loop iteration

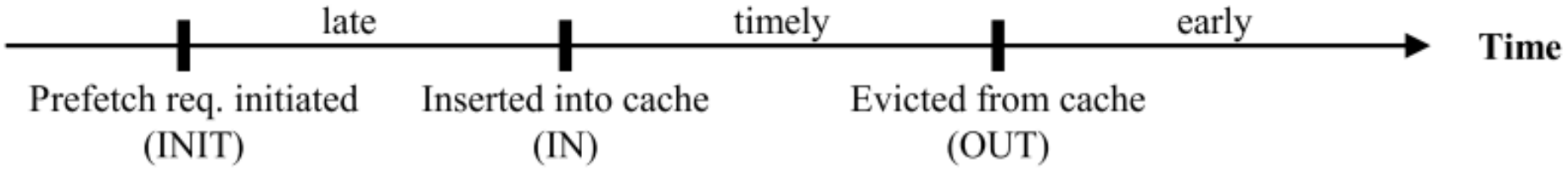
```
fetch (&sum);
for (i = 0; i < 12; i += 4){
    fetch (&a[i]);
    fetch (&b[i]);
}

for (i = 0; i < N-12; i += 4){
    fetch(&a[i+12]);
    fetch(&b[i+12]);
    sum = sum + a[i] *b[i];
    sum = sum + a[i+1]*b[i+1];
    sum = sum + a[i+2]*b[i+2];
    sum = sum + a[i+3]*b[i+3];
}

for (i = N-12; i < N; i++)
   sum = sum + a[i]*b[i];
```

Prefetching: Summary

- Prefetching can be done late, timely and early
- The following figure classifies this w.r.t. to the actual demand of the data item on a timeline.



- For more information: read “When Prefetching Works, When It Doesn’t, and Why”, Jaekyu Lee, Hyesoon Kim, Richard Vuduc. ACM Transactions on Architecture and Code Optimization (2012)

Tradeoffs of Multithreading and Prefetching

- Multithreading and prefetching are critically impacted by the memory bandwidth. Consider the following example:
  - Consider a computation running on a machine with a 1 GHz clock, 4-word cache line, single cycle access to the cache, and 100 ns latency to DRAM. **The computation has a cache hit ratio at 1 KB of 25% and at 32 KB of 90%**. Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB.
  - If the computation makes one data request in every cycle of 1 ns, and 1 word is 4 bytes, to work efficiently the first scenario **requires**
    - **400MB/s (we need 1 word per 10ns)**

    of memory bandwidth and the second,

    3GB/s (we need 24 words per 32ns / 0.75 word per ns)

Codes

See prefetching for binary search. But lets first discuss how can you use it?

(ArchBenchmarks/prefetch)

Tradeoffs of Multithreading and Prefetching

- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
- Multithreaded systems become **bandwidth bound** instead of **latency bound**.
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.

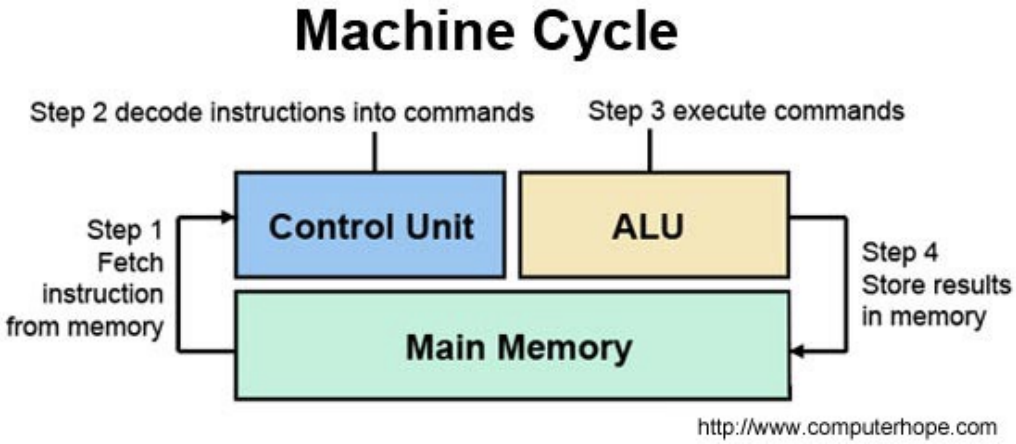
Topic Overview

Part 2

- Dichotomy of Parallel Computing Platforms

Control Structure of Parallel Programs

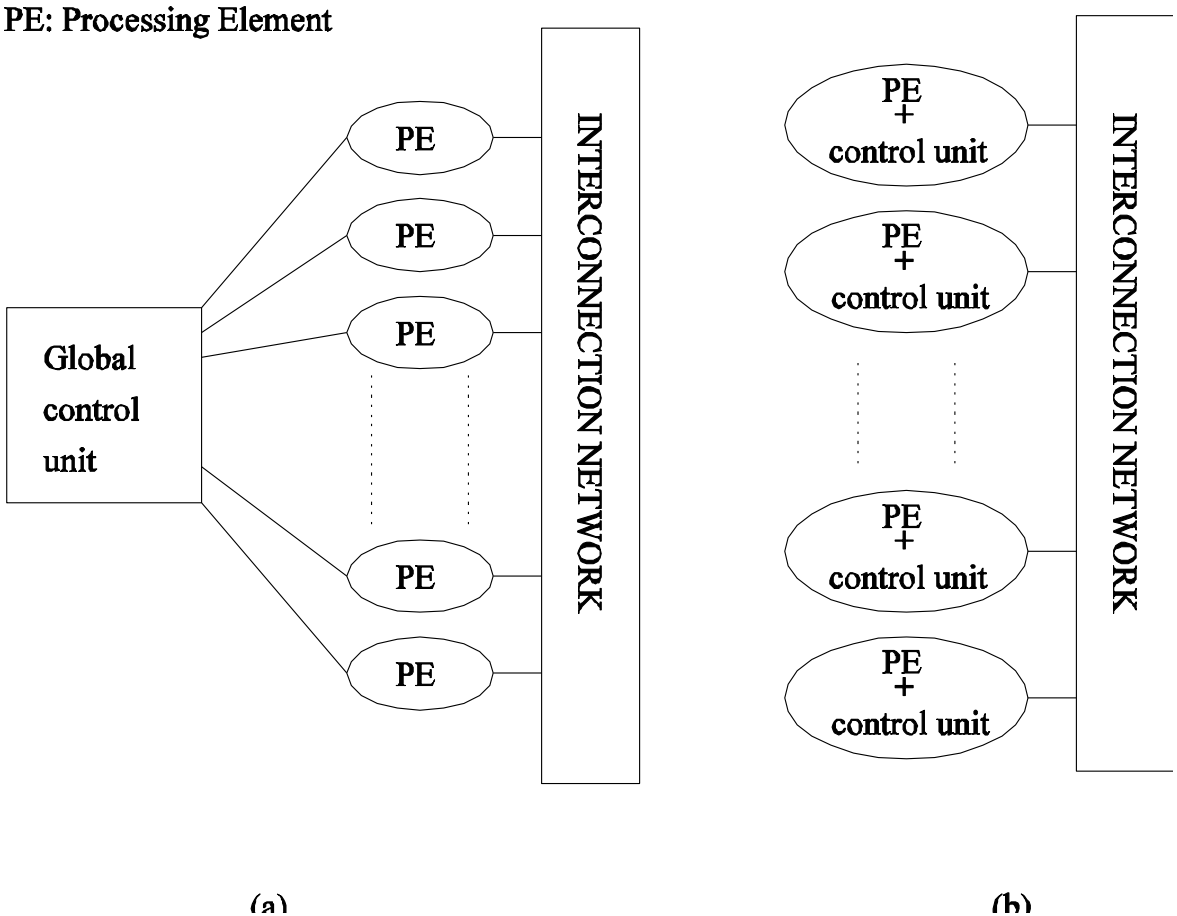
- **The control unit** is a component of a computer's central processing unit (CPU) that directs operation of the processor.
- It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions.



Control Structure of Parallel Programs

- Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.
- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as **single instruction stream, multiple data stream (SIMD)**.
- If each processor has its own control control unit, each processor can execute different instructions on different data items. This model is called **multiple instruction stream, multiple data stream (MIMD).**

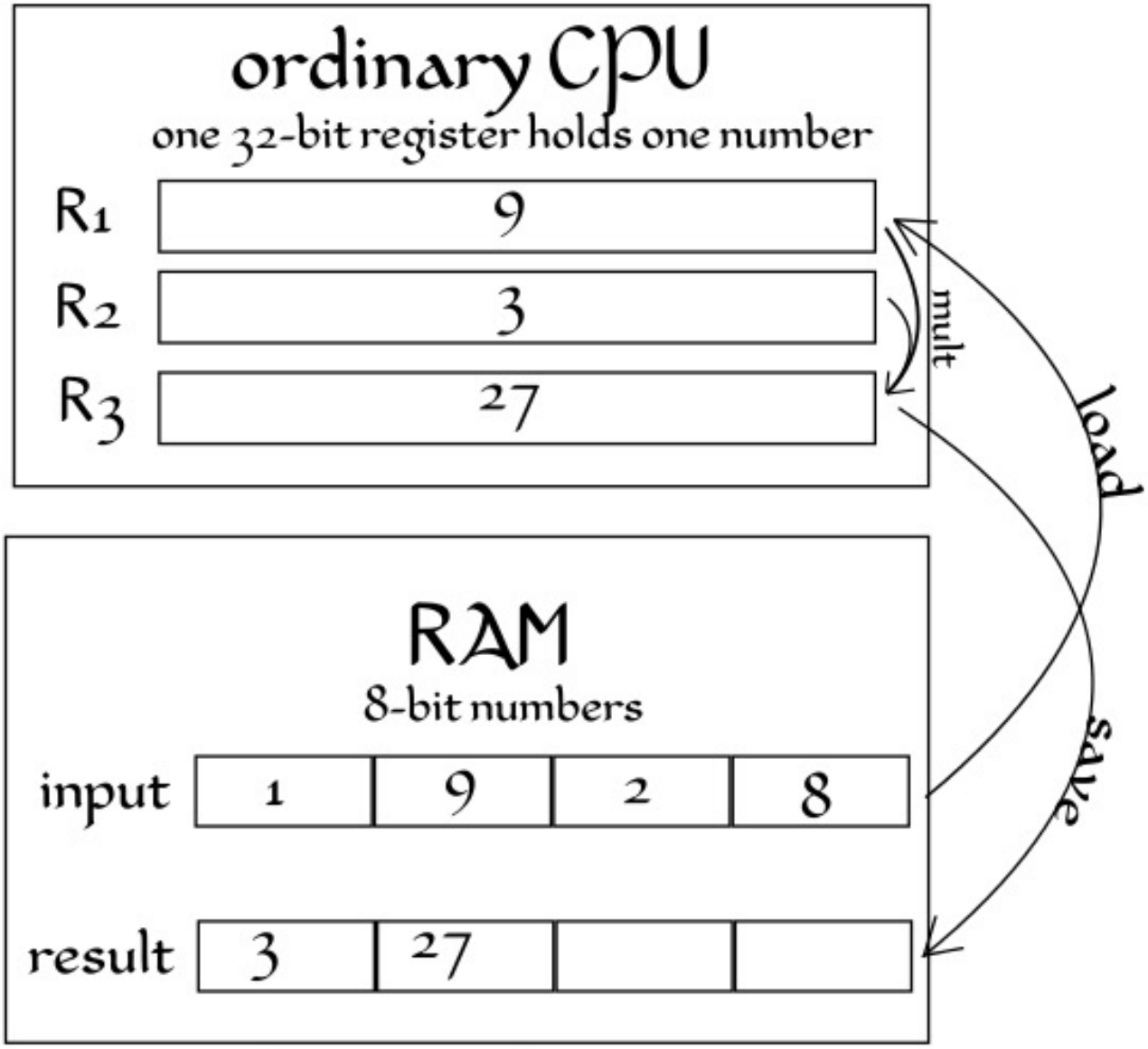
SIMD and MIMD Processors



A typical SIMD architecture (a) and a typical MIMD architecture (b).

MIMD Processors

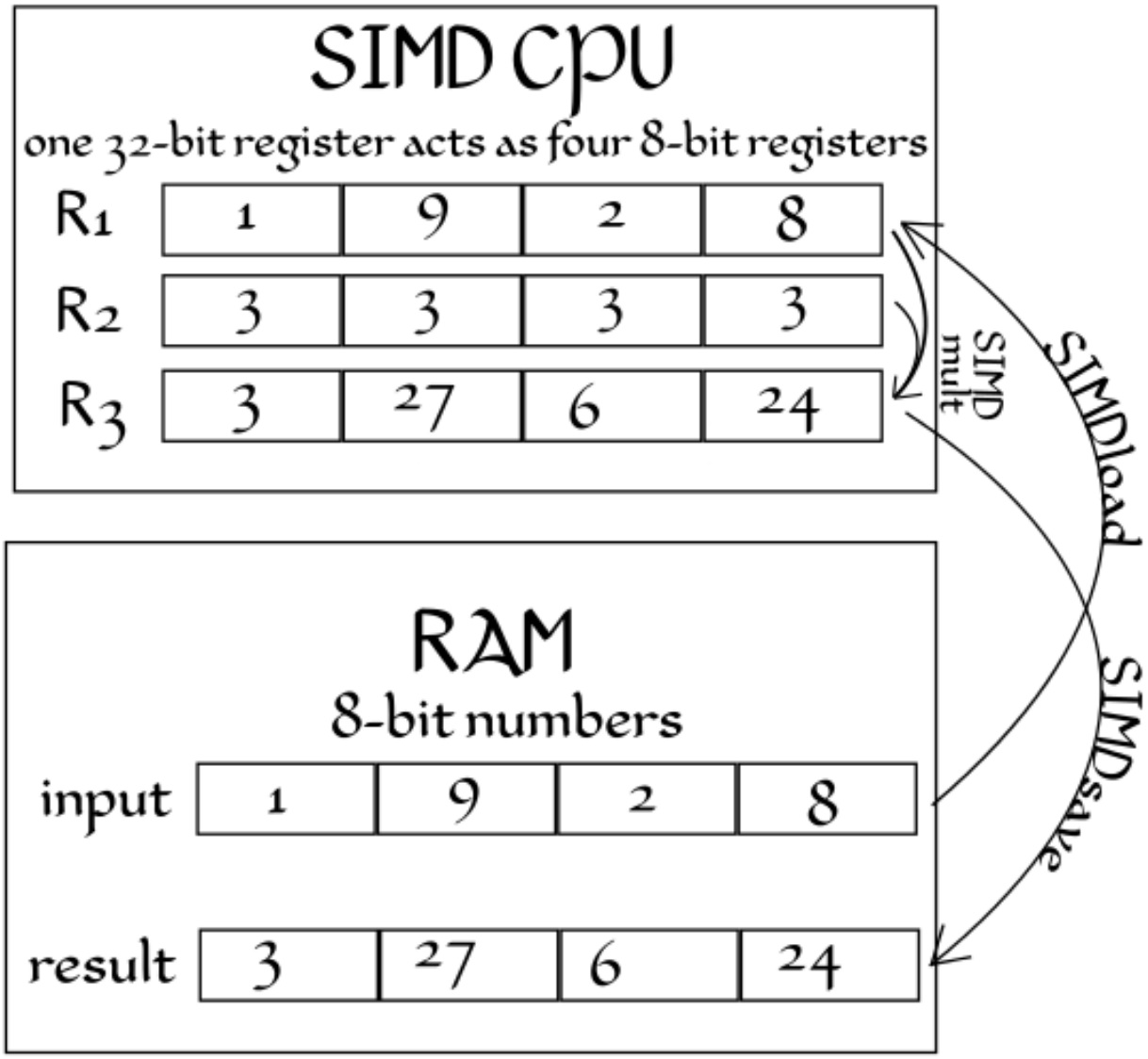
- In contrast to SIMD processors, MIMD processors can execute different programs on different processors.
- Examples of such platforms include current generation multiprocessor PCs and workstation clusters.



Operation Count:
4 loads, 4 multiplies, and 4 saves

SIMD Processors

- Some of the earliest parallel computers belonged to this class of machines.
- Variants of this concept have found use in co-processing units such as the MMX, SSE, AVX, AVX-512 units in Intel processors.
- SIMD relies on the regular structure of computations (such as those in image processing).

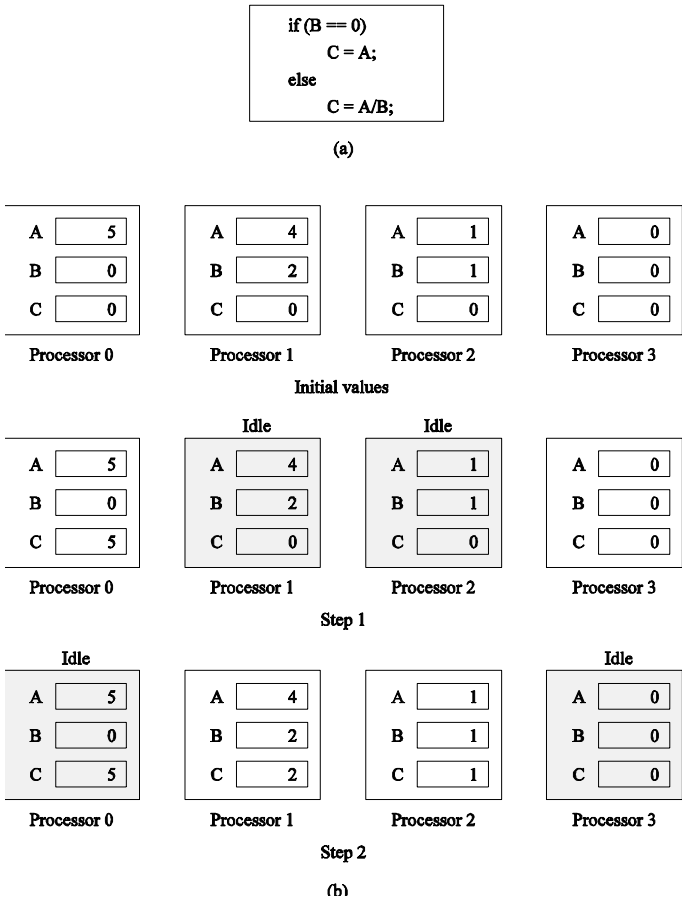


Operation Count:
1 load, 1 multiply, and 1 save

SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles.
- Not all applications are naturally suited to SIMD processors.

Conditional Execution in SIMD Processors



Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

Practical SIMD programming

- Vector registers support more than one data types:
  - Integer (16 bytes, 8 shorts, 4 int, 2 long long int, 1 dqword)
  - single precision floating point (4 floats)
  - double precision float point (2 doubles).

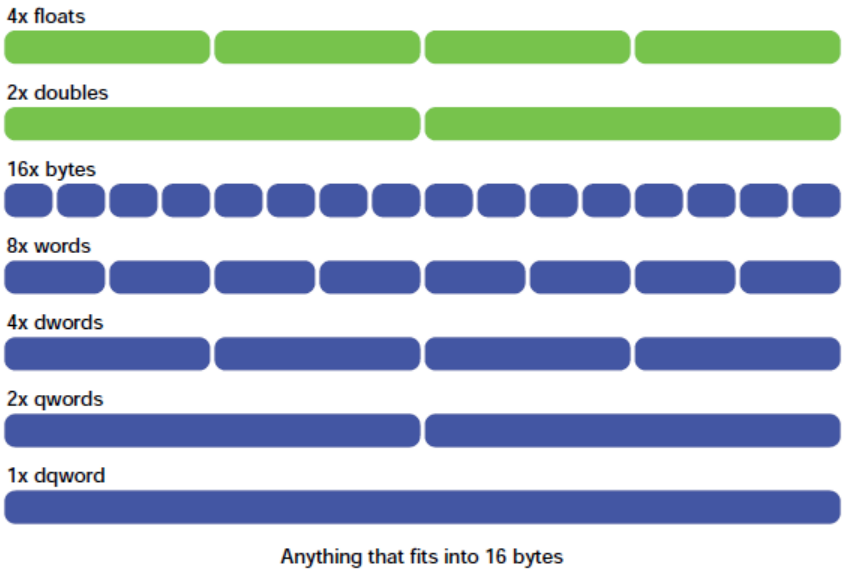


Figure 1. SSE/SSE2 data types

[Klimovitski 2001]

Practical SIMD programming

- Both current AMD and Intel's x86 processors have ISA (Industry Standard Architecture) and microarchitecture support SIMD operations.
- ISA SIMD support
  - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX, AVX2
    - See the flag field in /proc/cpuinfo
  - SSE (Streaming SIMD extensions): a SIMD instruction set extension to the x86 architecture
    - Instructions for operating on multiple data simultaneously (vector operations).
- Micro architecture support
  - Many functional units
  - 8 128-bit vector registers, XMM0, XMM1, …, XMM7

Practical SIMD programming

- Assembly instructions
  - Data movement instructions
    - moving data in and out of vector registers
  - Arithmetic instructions
    - Arithmetic operation on multiple data (2 doubles, 4 floats, 16 bytes, etc)
  - Logical instructions
    - Logical operation on multiple data
  - Comparison instructions
    - Comparing multiple data
  - Shuffle instructions
    - move data around SIMD registers
  - Miscellaneous
    - Data conversion: between x86 and SIMD registers
    - Cache control: vector may pollute the caches
    - State management:

See: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Practical SIMD programming

- Map to *intrinsics*
  - An *intrinsic* is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions.
  - Intrinsics provides a C/C++ interface to use processor-specific enhancements
  - Supported by major compilers such as gcc

Practical SIMD programming

- Header files to access SEE intrinsics
  - #include <mmintrin.h> // MMX
  - #include <xmmintrin.h> // SSE
  - #include <emmintrin.h> //SSE2
  - #include <pmmintrin.h> //SSE3
  - #include <tmmintrin.h> //SSSE3
  - #include <smmintrin.h> // SSE4
- MMX/SSE/SSE2 are mostly supported
- When compile, use –msse, -mmmx, -msse2 (machine dependent code)
  - Some are default for gcc.

SSE intrinsics

- Data types (mapped to an xmm register)
  - __m128: float
  - __m128d: double
  - __m128i: integer
- Data movement and initialization
  - _mm_load_ps, _mm_loadu_ps, _mm_load_pd, _mm_loadu_pd, etc
  - _mm_store_ps, …
  - _mm_setzero_ps

Practical SIMD programming

- Arithmetic intrinsics:
  - _mm_add_ss, _mm_add_ps, …
  - _mm_add_pd, _mm_mul_pd

Practical SIMD programming

- Data alignment issue
  - Some intrinsics may require memory to be aligned to 16 bytes.
    - May not work when memory is not aligned.
- Writing more generic SSE routine
  - Check memory alignment
  - Slow path may not have any performance benefit with SSE

Codes

See dot product examples, then other simple examples, and then finally matrix multiply

(SIMD)

Topic Overview

Part 3

- Communication Costs in Parallel Machines
- Messaging Cost Models and Routing Mechanisms

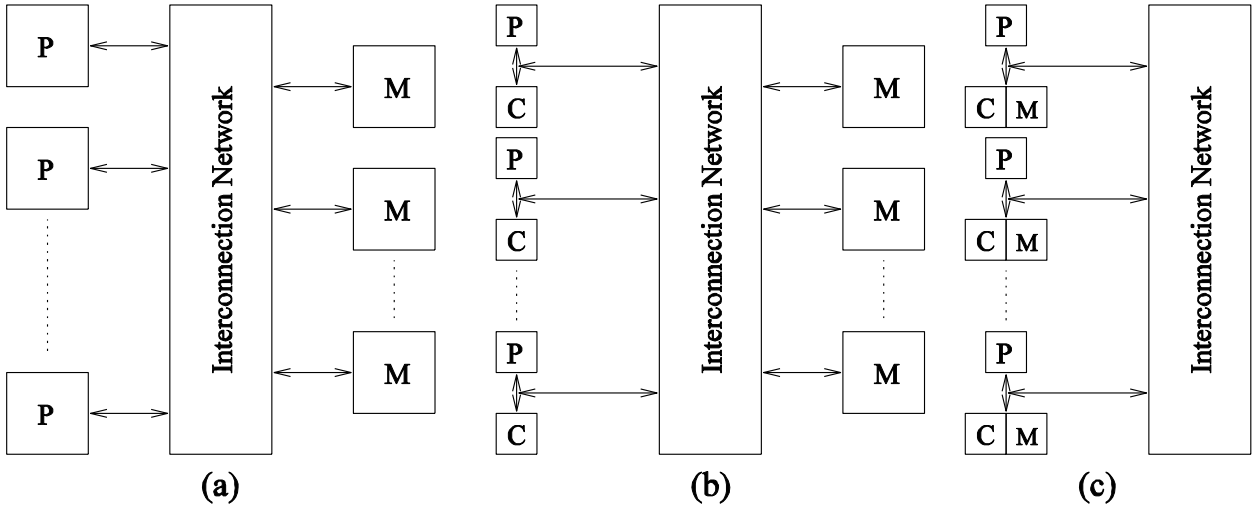
Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks - accessing a shared data space and exchanging messages.
  - Platforms that provide a shared data space are called **shared-address-space** machines or multiprocessors.
  - Platforms that support messaging are also called **message passing** platforms.

Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system global or local is identical, the platform is classified as a **uniform memory access (UMA)**, else, a **non-uniform memory access (NUMA)** machine.

NUMA and UMA Shared-Address-Space Platforms



Typical shared-address-space architectures: (a) Uniform-memory access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

Shared-Address-Space vs. Shared Memory Machines

- It is important to note the difference between the terms shared address space and shared memory.
- We refer to the former as a programming abstraction and to the latter as a physical machine attribute.
- It is possible to provide a shared address space using a physically distributed memory.

Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
- Instances of such a view come naturally from clustered workstations
- These platforms are programmed using (variants of) send and receive primitives.
- Libraries such as **MPI** provide such primitives.

Message Passing vs. Shared Address Space Platforms

- Message passing requires little hardware support, other than a network.
- Shared address space platforms can easily emulate message passing. The reverse is more difficult to do (in an efficient manner).

Interconnection Networks for Parallel Computers

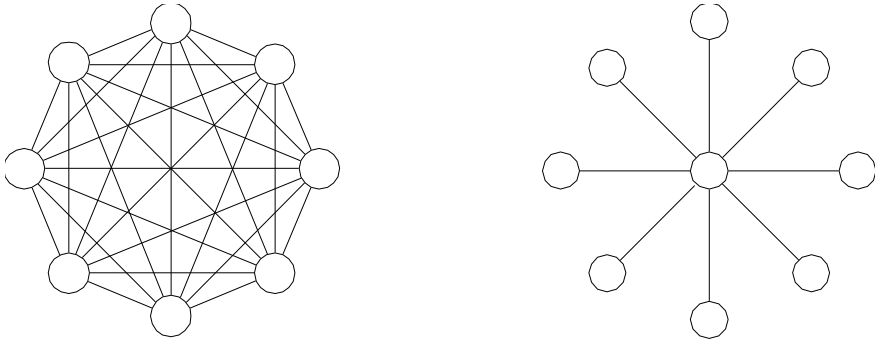
- Interconnection networks carry data between processors and to memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as **static** or **dynamic**.
  - **Static networks** consist of point-to-point communication links among processing nodes and are also referred to as ***direct*** networks.
  - **Dynamic networks** are built using switches and communication links. Dynamic networks are also referred to as ***indirect*** networks.

Network Topologies: Completely Connected Network

- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of p.

Network Topologies: Completely Connected and Star Connected Networks

Example of an 8-node completely connected network.



(a) A completely-connected network of eight nodes;
(b) a star connected network of nine nodes.

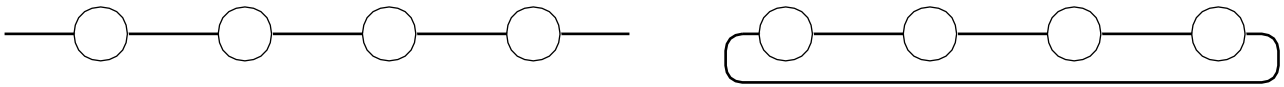
Network Topologies: Star Connected Network

- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is *O(1).*
  - **However, the central node becomes a bottleneck.**

Network Topologies: Linear Arrays, Meshes, and *k-d* Meshes

- In a linear array, each node has two neighbors, one to its left and one to its right. If the nodes at either end of the array are connected, we refer to it as a 1-D torus or a ring.
- A generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.
- A further generalization to *d* dimensions has nodes with *2d* neighbors.
- A special case of a *d*-dimensional mesh is a hypercube. Here, $d = \log p$, where p is the total number of nodes.

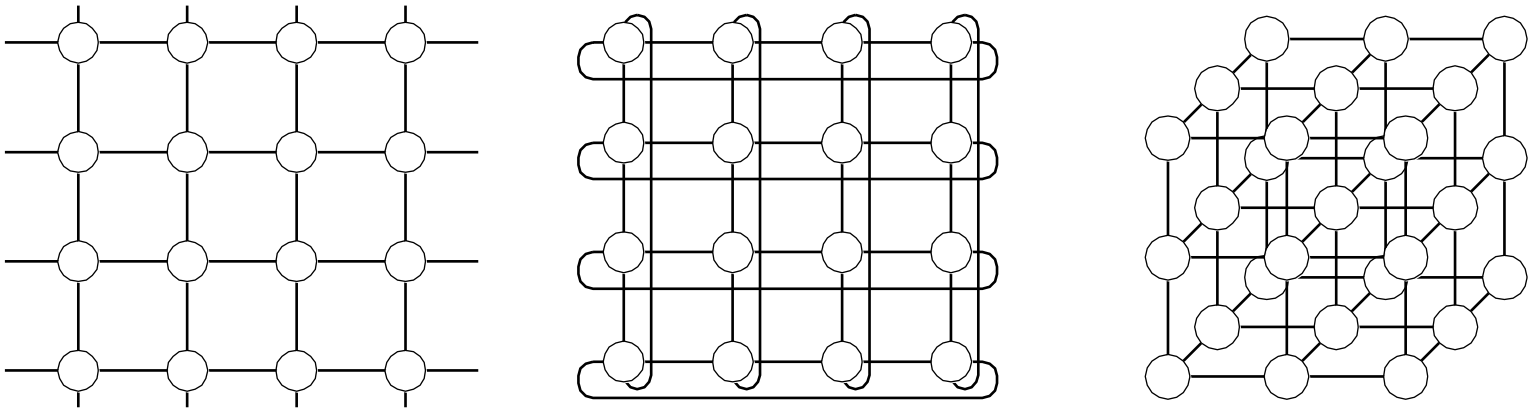
Network Topologies: Linear Arrays



(a) (b)

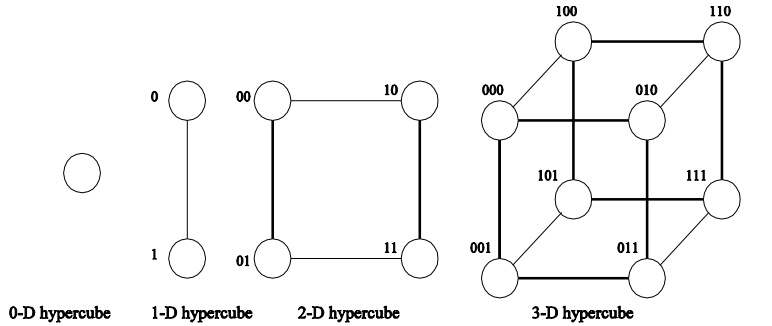
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

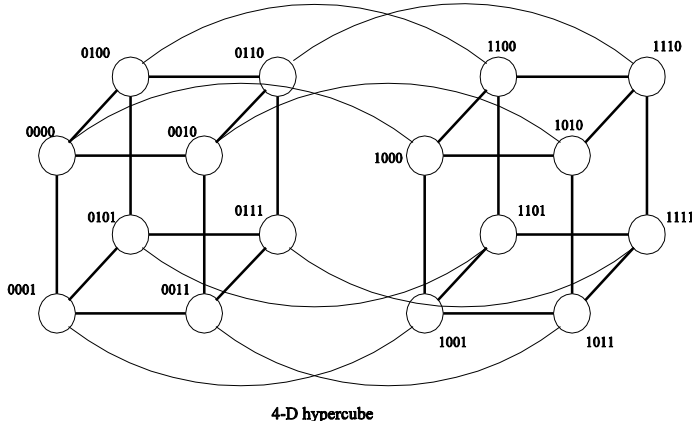
Network Topologies: Two- and Three Dimensional Meshes



Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

Network Topologies: Hypercubes and their Construction





Construction of hypercubes from hypercubes of lower dimension.

Network Topologies: Properties of Hypercubes

- The distance between any two nodes is at most *log p*.
- Each node has *log p* neighbors.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.

Evaluating Static Interconnection Networks

- ***Diameter****:* The distance between the farthest two nodes in the network. The diameter of a linear array is $p - 1$, that of a mesh is $2(\sqrt{p} - 1)$, that of a hypercube is $\log p$, and that of a completely connected network is $O(1)$.
- ***Bisection Width****:* The minimum number of wires you must cut to divide the network into two equal parts. The bisection width of a linear array is 1, that of a mesh is $\sqrt{p}$, that of a hypercube is $p/2$ and that of a completely connected network is $p^2/4$.
- ***Cost****:* The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.

Evaluating Static Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound *k*-ary *d*-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Communication Costs in Parallel Machines

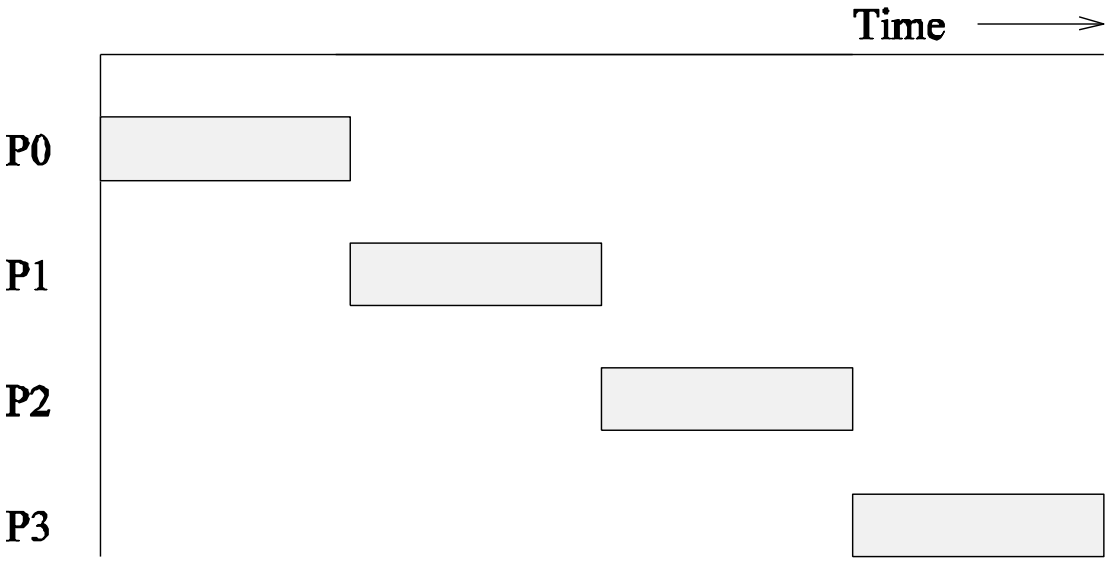
- Along with idling and contention, communication is a major overhead in parallel programs.
- The cost of communication is dependent on a variety of features including the **programming model semantics**, the **network topology**, **data handling** and **routing**, and **associated software protocols**.

Message Passing Costs in Parallel Computers

- The total time to transfer a message over a network comprises of the following:
  - ***Startup time*** **(t_s)**: Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  - ***Per-hop time*** **(t_h)**: This time includes factors such as switch latencies, network delays, etc.
  - ***Per-word transfer time*** **(t_w)**: This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

Store-and-Forward Routing

A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.



(a) A single message sent over a store-and-forward network

Store-and-Forward Routing

- The total communication cost for a message of size *m* words to traverse *l* communication links is

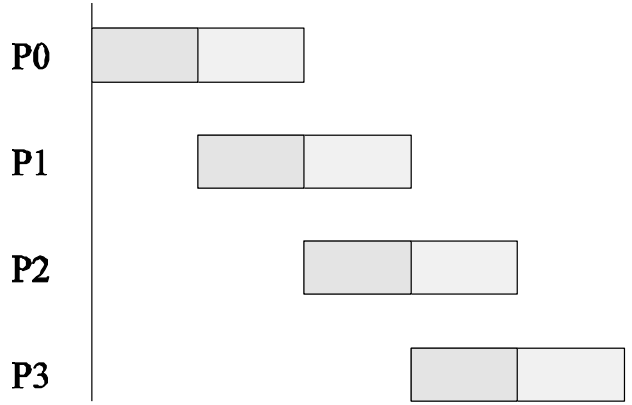
$$t_{comm} = t_s + (mt_w + t_h)l.$$

- In most platforms, t_h is small and the above expression can be approximated by

$$t_{comm} = t_s + mlt_w.$$

Packet Routing

- Store-and-forward makes poor use of communication resources.
- Packet routing breaks messages into packets and pipelines them through the network.



(b) The same message broken into two parts and sent over the network.

Packet Routing

- **m** word message
- **r** is the packet size (in words)
- **s** is the additional information carried in the packet.

Time for packetizing the message

Time for the first packet

Time for later packets

$$t_{comm} = t_s + t_{w1}m + t_h l + t_{w2}(r+s) + \left(\frac{m}{r} - 1\right) t_{w2}(r+s)$$

$$= t_s + t_{w1}m + t_h l + t_{w2}m + t_{w2}\frac{s}{r}m$$

$$= t_s + t_h l + t_w m,$$

where $t_w = t_{w1} + t_{w2}\left(1 + \frac{s}{r}\right)$

Packet Routing (summary)

- Since packets may take different paths, each packet must carry routing information, error checking, sequencing, and other related header information.
- The total communication time for packet routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

- The factor t_w accounts for overheads in packet headers.

Cut-Through Routing

- Takes the concept of packet routing to an extreme by further dividing messages into basic units called flits.
- Since flits are typically small, the header information must be minimized.
- This is done by forcing all flits to take the same path, in sequence.
- A tracer message first programs all intermediate routers. All flits then take the same route.
- Error checks are performed on the entire message, as opposed to flits.
- No sequence numbers are needed.

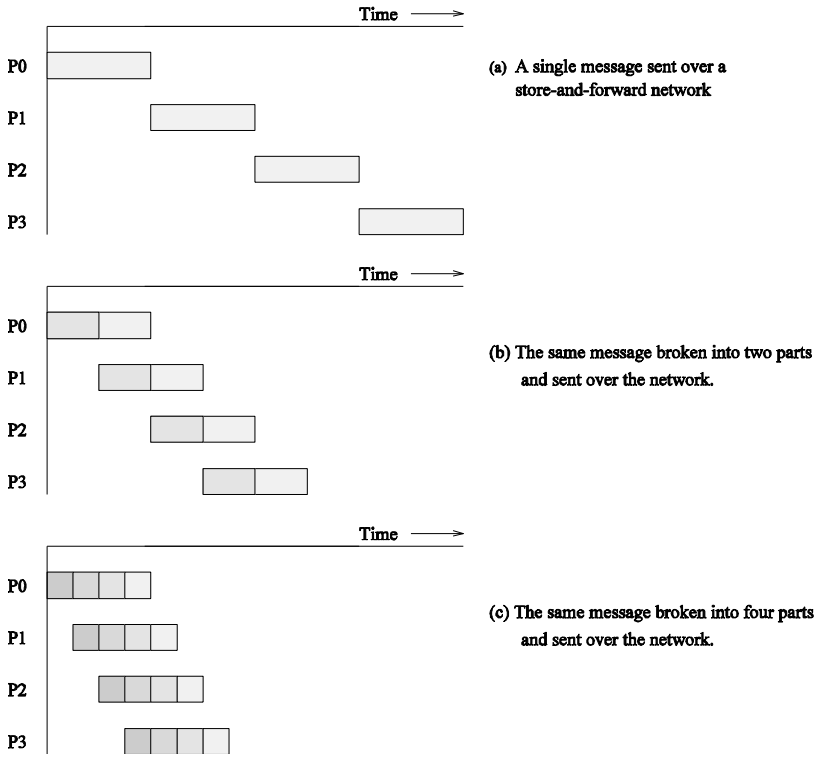
Cut-Through Routing

- The total communication time for cut-through routing is approximated by:

$$t_{comm} = t_s + t_h l + t_w m.$$

- This is identical to packet routing, however, t_w is typically much smaller.

Routing Techniques: Summary



Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

Simplified Cost Model for Communicating Messages

- The cost of communicating a message between two nodes *l* hops away using cut-through routing is given by

$$t_{comm} = t_s + lt_h + t_w m.$$

- In this expression, t_h is typically smaller than t_s and t_w. For this reason, the second term in the RHS does not show, particularly, when m is large.
- Furthermore, it is often not possible to control routing and placement of tasks.
- For these reasons, we can approximate the cost of message transfer by

$$t_{comm} = t_s + t_w m.$$

Simplified Cost Model for Communicating Messages

- It is important to note that the original expression for communication time is valid for only uncongested networks.
- If a link takes multiple messages, the corresponding t_w term must be scaled up by the number of messages.
- Different communication patterns congest different networks to varying extents.
- It is important to understand and account for this in the communication time accordingly.

Rules of Thumb for Communication

Communicate in bulk. Instead of sending small messages and paying t_s for each, aggregate them into a large message and amortize the startup latency. This is because on typical platforms such as clusters, t_s is much larger than t_h or t_w.

Minimize the volume of data. To minimize the overhead paid in terms of per-word transfer time t_w, it is desirable to reduce the volume of data communicated as much as possible.

Minimize distance of data transfer. Minimize the number of hops l that a message must traverse.

Effects of Congestion on the Model

Consider a $\sqrt{p} \times \sqrt{p}$ mesh in which each node is only communicating with its nearest neighbor. Since no links in the network are used for more than one communication, the time for this operation is $\mathbf{t_s} + \mathbf{t_w m}$, where **m** is the number of words communicated. This time is consistent with our simplified model.

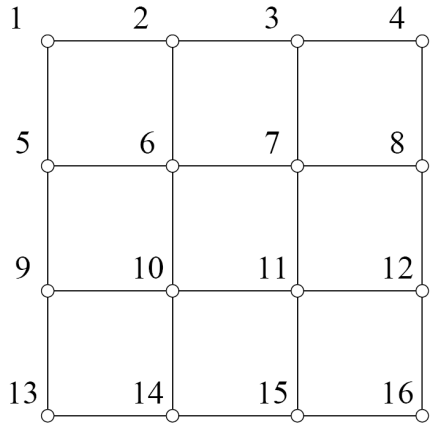
Effects of Congestion on the Model

Consider a scenario in which each node is communicating with a random node. Randomness implies that there are **p/2** communications (or **p/4** bi- directional communications) occurring across any equi-partition of the machine (since the node being communicated with could be in either half with equal probability).

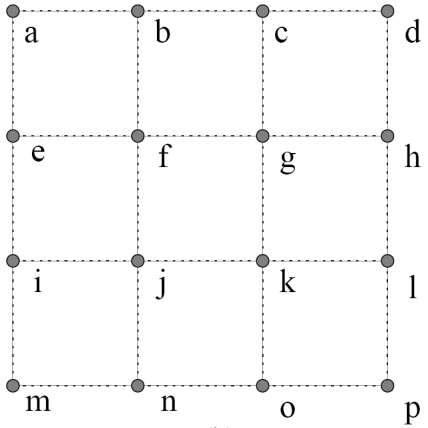
We know that a 2-D mesh has a bisection width of $\sqrt{p}$ We can infer that some links would now have to carry at least $\frac{p/4}{\sqrt{p}} = \sqrt{p}/4$ messages

These messages must be serialized over the link. If each message is of size **m**, the time for this operation is at least $t_s + t_w m \times \sqrt{p}/4$

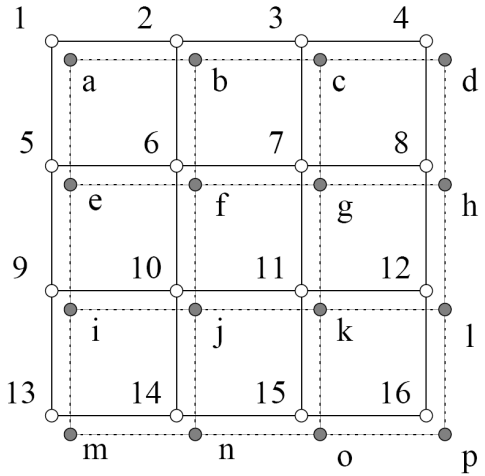
Impact of Process-Processor Mapping and Effective Bandwidth



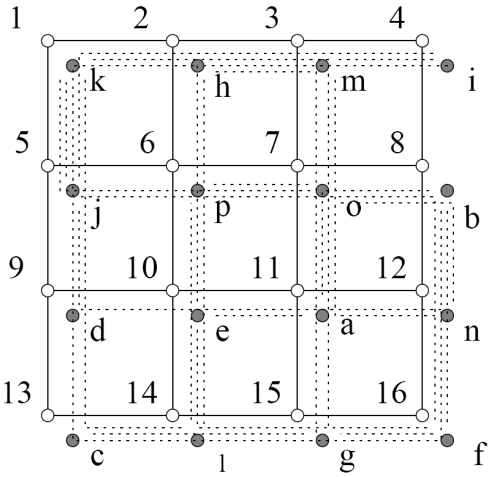
(a)



(b)



(c)

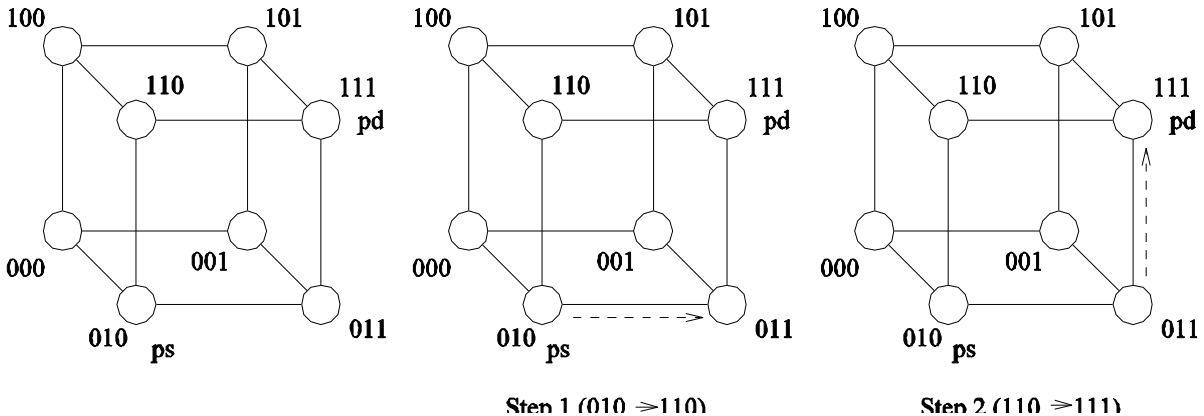


(d)

Routing Mechanisms for Interconnection Networks

- How does one compute the route that a message takes from source to destination?
  - Routing must prevent deadlocks - for this reason, we use dimension-ordered or e-cube routing.
  - Routing must avoid hot-spots - for this reason, two-step routing is often used. In this case, a message from source *s* to destination *d* is first sent to a randomly chosen intermediate processor *i* and then forwarded to destination *d*.

Routing Mechanisms for Interconnection Networks



Routing a message from node P_s (010) to node P_d (111) in a three-dimensional hypercube using E-cube routing.