What is the length of a clock cycle in a 2 GHz machine, in picoseconds?

500ps

**Amdahl’s Law:** The opportunity for improvement is affected by how much time an event consumes. It’s a quantitative law of diminishing returns.

**Relative Performance:**

Computer a runs a given program in 10s

Computer b runs a given program in 15s

A is n times as fast as B if:

Performance[a]/Performance[b]

= Execution time[b]/Execution[a]

15/10 = 1.5

“Static” refers to the code in memory or on the filesystem

“Dynamic” refers to what happens at runtime

Response time: How long it takes to do a task.

Throughput: Total work done per unit time.

1 kilobyte (KB) → 1024 bytes (210 bytes)

1 megabyte (MB) → 1024 KB (220 bytes)

1 gigabyte (GB) → 1024 MB (230 bytes)

1 picosecond = 10^-12

1 nanosecond = 10^-9

1 microsecond = 10^-6

1 millisecond = 10^-3

1 KHz = 10^3

1 MHz = 10^6

1 GHz = 10^9

X0 – X7: procedure arguments/results

X8: indirect result location register

X9 – X15: temporaries

X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register

X18: platform register for platform independent code; otherwise a temporary register

X19 – X27: saved

X28 (SP): stack pointer

X29 (FP): frame pointer

X30 (LR): link register (return address)

XZR (register 31): the constant value 0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

ASCII: 128 characters

**Cycles per Instruction**

= CPU Cycles/Instruction Count

**Clock Cycles**

= Instruction Count \* Cycles per Instruction

**CPU Time**

= Instruction Count \* CPI \* Clock Cycle Time

= (Instruction Count \* CPI)/Clock Rate

What is the decimal value of the following signed 8-bit binary: (~0x51 + 1)?

-81

What is the decimal value of (75 >> 2) ?

18

**Floating Point: Decimal 🡪 Binary**

741.625

741 – 512 (2^9) = 229

229 – 128 (2^7) = 101

101 – 64 (2^6) = 37

37 – 32 (2^5) = 5

5 – 4 (2^2) = 1

1 – 1 (2^0) = 0

🡪Whole number: 1011100101

0.625 \* 2

1.25

0.25 \* 2 XXX ‘0’ Not included into Fraction

0.5 \* 2

1.0

🡪 Fraction: 0.101

🡪 Whole + Fraction: 1011100101.101

🡪 Normalized: 1.011100101101 x 2^9

Exponent + bias

= 9+127

= 136

136 – 128 (2^7) = 8

8 – 8 (2^3) = 0

🡪Exponent field: 10001000

Sign | Exponent | Fraction

0 | 10001000 | 011100101101…0

**Binary 🡪 Decimal**

Sign | Exponent | Fraction

1 | 10000001 | 01000000…0

The sign bit is 1, the exponent field contains 129, and the fraction field contains 1 \* 2-2 = ¼ or 0.25. Convert using the equation:

(-1)S \* (1+Fraction) \* 2(Exponent – Bias)

= (-1)S \* (1 + 0.25) \* 2(129 – 127)

= -1 \* 1.25 \* 2(2)

= -1.25 \* 4

= -5.0

**Leg V8 Pipeline Stages:**

IF: Instruction fetch from memory

ID: Instruction decode and memory read

EX: Execute operation or calculate address

MEM: Access memory operand

WB: Write result back to register

**Associative Caches:**

Larger blocks should reduce miss rate

Due to spatial locality

But in a fixed-sized cache

Larger blocks ⇒ fewer of them

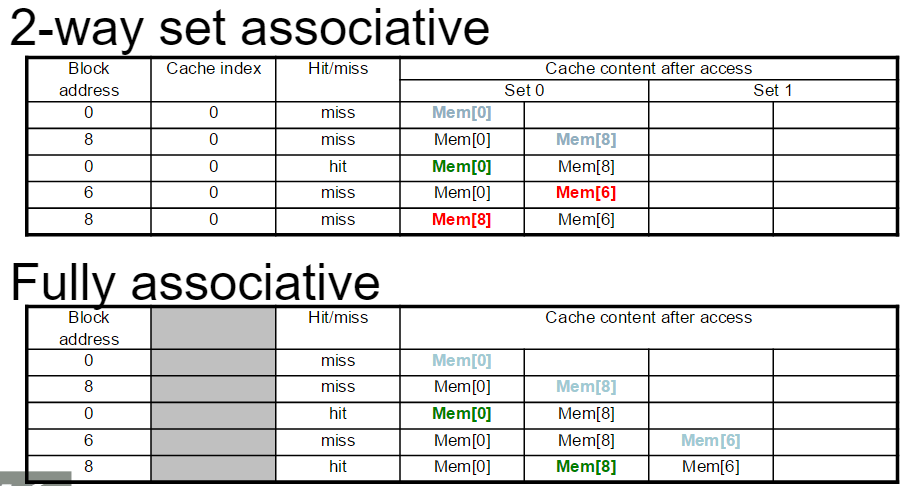
More competition ⇒ increased miss rate

Larger blocks ⇒ pollution

Larger miss penalty

Can override benefit of reduced miss rate

Early restart and critical-word-first can help



**n-way set associative**

Each set contains n entries

Block number determines which set

(Block number) modulo (#Sets in cache)

Search all entries in a given set at once

n comparators (less expensive)

**Fully Associative:**

Allow a given block to go in any cache entry

Requires all entries to be searched at once

Comparator per entry (expensive)

Block address index is 0;

Tag: 27 bits; Offset: 5 bits

32 Bytes

256 rows

**4-way set associative:**

256 rows = 64 sets

Log2 (64) = 6

Tag: 21; Index: 6; Offset: 5

**Direct Mapped:** 1-way set associative

Tag: 19; Index: 8; Offset: 5

**Cache Performance Example:**

**Given**

I-cache miss rate = 2%

D-cache miss rate = 4%

Miss penalty = 100 cycles

Base CPI (ideal cache) = 2

Load & stores are 36% of instructions

**Miss cycles per instruction**

I-cache: 0.02 × 100 = 2

D-cache: 0.36 × 0.04 × 100 = 1.44

**Actual CPI = 2 + 2 + 1.44 = 5.44**

Ideal CPU is 5.44/2 =2.72 times faster

**Average memory access time (AMAT)**

AMAT = Hit time + Miss rate × Miss penalty

**Example**

CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%

AMAT = 1 + 0.05 × 20 = 2ns

2 cycles per instruction

**Cache Performance Summary:**

When CPU performance increased

Miss penalty becomes more significant

Decreasing base CPI

Greater proportion of time spent on memory stalls

Increasing clock rate

Memory stalls account for more CPU cycles

**RAID:** **Redundant Array of Inexpensive (Independent) Disks**

Use multiple smaller disks (c.f. one large disk)

Parallelism improves performance

Plus extra disk(s) for redundant data storage

Provides fault tolerant storage system

Especially if failed disks can be “hot swapped”

**RAID 0**

No redundancy (“AID”?)

Just stripe data over multiple disks

But it does improve performance

**RAID 1:** Mirroring

N + N disks, replicate data

Write data to both data disk and mirror disk

On disk failure, read from mirror

**RAID 2:** Error correcting code (ECC)

N + E disks (e.g., 10 + 4)

Split data at bit level across N disks

Generate E-bit ECC

Too complex, not used in practice

**RAID 4:**

N + 1 disks

Data striped across N disks at block level

Redundant disk stores parity for a group of blocks

Read access

Read only the disk holding the required block

Write access

Just read disk containing modified block, and parity disk

Calculate new parity, update data disk and parity disk

On failure

Use parity to reconstruct missing data

Not widely used

**RAID 5:**

N + 1 disks

Like RAID 4, but parity blocks distributed across disks

Avoids parity disk being a bottleneck

Widely used

**RAID 6:**

N + 2 disks

Like RAID 5, but two lots of parity

Greater fault tolerance through more redundancy

Multiple RAID

More advanced systems give similar fault tolerance with better performance

**3 step process on 4KB of data:**

4KB = 4096 bytes = 32,768 bits

Transfer rate: 20,000,000 bits/sec

Seek time: 8 ms

Controller overhead: 2 ms

Rotational latency: ½ / (7,200/60) = ~0.0042 sec = 4.2 ms

Time it takes to read 4KB of data: 1.64 ms

32,768 bits / 20,000,000 bits/sec = ~0.00164 sec = 1.64 ms

Time it takes to process the data: 50 ms

20,000,000 cycles / 400,000,000 cycles/sec = 0.05 sec = 50 ms

Time it takes to write the data: 1.64 ms

32,768 bits / 20,000,000 bits/sec = ~0.00164 sec = 1.64 ms

    8.00 ms (Seek)

    4.20 ms (Rotational)

+  1.64 ms (Read)

   2.00 ms (Overhead)

 50.00 ms (Process)

8.00 ms (Seek)

    4.20 ms (Rotational)

 1.64 ms (Read)

    1.64 ms (Write)

   2.00 ms (Overhead)

\_\_\_\_\_\_\_\_\_

83.32 ms/block

1 block              x blocks

--------------  =  --------------

0.08332sec       1 sec

= ~12 blocks/sec

**Spatial locality:**

Items near those accessed recently are likely to be accessed soon

E.g., sequential instruction access, array data

**Temporal locality:**

Items accessed recently are likely to be accessed again soon

e.g., instructions in a loop, induction variables

SMP: shared memory multiprocessor

Static RAM (SRAM)

0.5ns – 2.5ns, $2000 – $5000 per GB

Dynamic RAM (DRAM)

50ns – 70ns, $20 – $75 per GB

Magnetic disk

5ms – 20ms, $0.20 – $2 per GB

**Write Through:**

On data-write hit, could just update the block in cache

But then cache and memory would be inconsistent

Write through: also update memory

But makes writes take longer

e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles

Effective CPI = 1 + 0.1×100 = 11

Solution: write buffer

Holds data waiting to be written to memory

CPU continues immediately

Only stalls on write if write buffer is already full

**Write Back:**

Alternative: On data-write hit, just update the block in cache

Keep track of whether each block is dirty

When a dirty block is replaced

Write it back to memory

Can use a write buffer to allow replacing block to be read first

**Write Allocation:**

What should happen on a write miss?

Alternatives for write-through

Allocate on miss: fetch the block

Write around: don’t fetch the block

Since programs often write a whole block before reading it (e.g., initialization)

For write-back

Usually fetch the block

**Cache Design Trade-offs:**

Design change

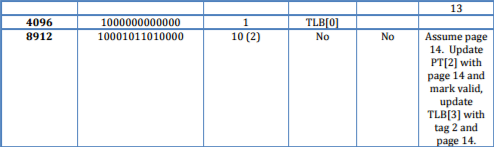
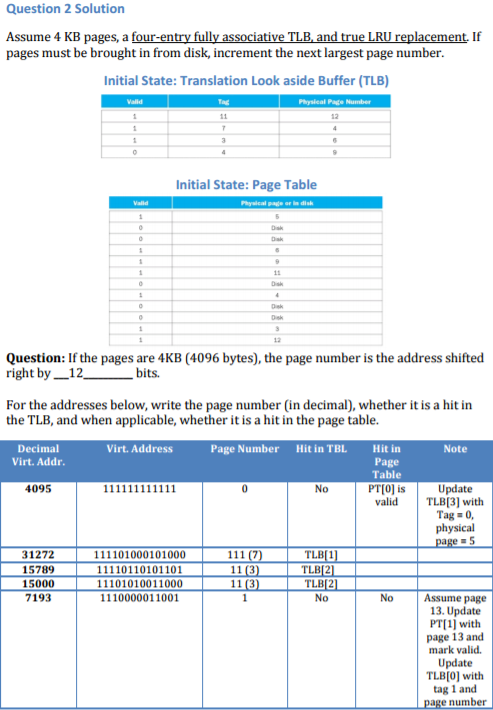
1. Increase cache size
2. Increase associativity
3. Increase block size

Effect on miss rate

1. Decrease capacity misses
2. Decrease conflict misses
3. Decrease compulsory misses

Negative performance effect

1. May increase access time
2. May increase access time
3. Increases miss penalty. For very large block size, may increase miss rate due to pollution.



**For 4-word block, 1-word-wide DRAM**

Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles

Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

**4-word wide memory**

Miss penalty = 1 + 15 + 1 = 17 bus cycles

Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle

**4-bank interleaved memory**

Miss penalty = 1 + 15 + 4×1 = 20 bus cycles

Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

**Replacement Policy:**

Direct mapped: no choice

Set associative

Prefer non-valid entry, if there is one

Otherwise, choose among entries in the set

Least-recently used (LRU)

Choose the one unused for the longest time

Simple for 2-way, manageable for 4-way, too hard beyond that

Random

Gives approximately the same performance as LRU for high associativity

**Multilevel Cache Example:**

Given

CPU base CPI = 1, clock rate = 4GHz

Miss rate/instruction = 2%

Main memory access time = 100ns

With just primary cache

Miss penalty = 100ns/0.25ns = 400 cycles

Effective CPI = 1 + 0.02 × 400 = 9

Now add L-2 cache

Access time = 5ns

Global miss rate to main memory = 0.5%

Primary miss with L-2 hit

Penalty = 5ns/0.25ns = 20 cycles

Primary miss with L-2 miss

CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4

Performance ratio = 9/3.4 = 2.6