6.1 SRAM and DRAM

Memory basics

An NxM ***memory*** is a digital component that retains bit values, consisting of N words of M bits each. Each word has a unique address. Ex: A 4096x8 memory has 4096 8-bit words (for a total of 32,768 bits), with word addresses from 0 to 4095. Nearly every computing device requires memory.

PARTICIPATION ACTIVITY 6.1.1: Memory basics.

Animation captions:

1. An NxM memory has N words, each M bits wide (4096x8 here). Word addresses are 0, 1, 2, ..., 4094, and 4095. Input addr requires 12 bits to represent 0-4095.
2. To write, wen (write enable) is set with 1, and the target word's address is provided (2). On the next rising clock, the data (15) on wdata (write data) is stored into that word.
3. Other words can be written similarly. In this case, word 4094 is written with 33. Note that 15 remains stored in word 2.
4. Setting ren (read enable) with 1 causes the addressed word to appear on rdata (read data), typically asynchronously.

Such memory is often called ***random access memory*** (or ***RAM***) because any "random" word can be quickly accessed, in contrast to older sequentially-accessed memory technologies like tape that had to first be spun or moved to access a particular word.

Most RAM is ***volatile memory***, meaning bit values are lost if electrical power is removed.

Note: For a processor, "word" may refer to 4 bytes. But for a memory, word means one address location, however wide.

PARTICIPATION ACTIVITY 6.1.2: Memory basics.

Consider the memory above. Question actions follow the previous question's actions. All words initially contain 0. Answer in decimal, not binary.

1) On rising clock 1, addr is 9, wen is 1, and wdata is 44.
On rising clock 2, addr is 5, wen is 1, and wdata is 77.
What value is in word 9?

Check **Show answer**

2) On rising clock 3, addr is 15, wen is 0, and wdata is 305.

What value is now in word 15?

Check **Show answer**

3) addr is 5, ren is 1.
What soon appears on rdata?

Check **Show answer**

4) How many bits does word 3 contain?

Check **Show answer**

5) How many total bits does the memory store?

Check **Show answer**

6) Based on the general memory design described above, is simultaneously setting both ren and wen with 1 reasonable? Type yes or no.

Check **Show answer**

SRAM and DRAM

Memory comes in two common forms:

- A ***static RAM*** (or ***SRAM***) typically uses 6 transistors to store each bit value, by passing the bit into a loop within those transistors.
- A ***dynamic RAM*** (or ***DRAM***) typically uses 1 transistor and 1 capacitor to store each bit value, by charging the capacitor.

SRAM is faster than DRAM, but DRAM is denser and cheaper. SRAM is thus typically used by processors for small fast on-chip cache, while DRAM is used for the larger main memory off-chip. Also, processors and SRAM are made using different chip design processes than DRAM, so putting DRAM on-chip with a processor is rare.

PARTICIPATION ACTIVITY 6.1.3: SRAM vs. DRAM.

Animation captions:

1. SRAM read/writes are about 10x faster than for DRAM.
2. But DRAM's are about 5x denser (using only 1 transistor/cell). Because of the density and mass production, DRAM's are about 100x cheaper too.
3. Thus, SRAM is used primarily for a processor's small but fast cache or other small on-chip memory, while DRAM is used for a processor's large main memory off-chip.

PARTICIPATION ACTIVITY 6.1.4: SRAM vs. DRAM.

1) Which is faster?
   - SRAM
   - DRAM

2) For a fixed size, which can store more bits?
   - SRAM
   - DRAM

3) Consider a processor that accesses a memory once per instruction. Suppose each instruction's time is 0.5 ns to access SRAM and 0.5 ns to run the instruction, so a 1 billion instruction program takes 1 second to execute. How many seconds would the program take using DRAM instead?
   - 1.2 sec
   - 5.5 sec

4) Consider a processor that requires 1 GB of DRAM, where the DRAM costs $50. About how much would the memory cost if all the DRAM was instead SRAM?
   - $500
   - $5000

Memory size

A memory's size may be specified in various ways.

- *4096x32*: Indicates the number of words, and the bits per word.
- *131,072 bits*: Indicates the total number of bits.
- *16,384 bytes*: Indicates the total number of bytes (a byte is 8 bits).
- *16 KBytes (or 16 KB)*: Approximate number of bytes. The K is the metric kilo, for 1,000. Note: This method is common *but inaccurate*, because 16 Kbytes means 16,000 bytes rather than the actual 16,384 bytes.

- *128 Kbits (or 128 Kb)*: Indicates the total number of bits. Again, this method is common *but inaccurate*.

Memory sizes are commonly measured in MB (megabytes, or 1 million bytes)), GB (gigabytes, or 1 billion bytes), or TB (terabytes, or 1 trillion bytes). Ex: A 16 GB memory. The uppercase B means bytes (like GB), while lowercase b means bits (like Gb).

Memory sizes are powers of 2, so metric prefixes like kilo, mega, and giga, which are powers of 10, are inaccurate. Alternative prefixes, known as ***IEC prefixes***, exist like kibi (2^{10} or 1024), mebi (2^{20} or 1,048,576), gibi (2^{30} or 1,073,741,824), and tebi (2^{40} or 1,099,511,627,776). In kibi, the ki refers to the metric prefix kilo, and the bi to "binary". A kibi is abbreviated Ki, as in 1 KiB for 1 kibibyte. Likewise for other IEC prefixes.

When metric prefixes are used, those prefixes are known to actually refer to the nearest power of 2, so a kilobyte is known to mean 1024 bytes (a kibibyte) and not 1000 bytes.

PARTICIPATION ACTIVITY 6.1.5: Memory sizes.

1) A 512x8 memory has how many *words*?

Check **Show answer**

2) A 64x4 memory has how many *bits*?

Check **Show answer**

3) In 1 KB, does the B mean bytes or bits?

Check **Show answer**

4) For memory, 1K actually refers to what value? Just type a number.

Check **Show answer**

5) What IEC prefix refers to the power of 2 nearest to 1 million?

Check **Show answer**

6) How many bits are in a 1 KB memory? Just type a number.

Check **Show answer**

7) How many bits are in a 2 Kb memory? Just type a number.

Check **Show answer**

Exploring further:

- SRAM (Wikipedia)
- DRAM (Wikipedia)
- SRAM vs. DRAM (diffen.com)
- IEC prefixes

6.2 Chip economics

High NRE cost encourages mass-production

A ***chip*** (aka ***integrated circuit*** or ***IC***) is a digital circuit manufactured on a fingernail-sized piece of silicon, typically placed inside a black or silver insulating package.

Non-recurring engineering (or ***NRE***) cost is the cost to design and set up a computer chip for manufacturing. Due to the complexity of modern chips having billions of transistors, NRE costs may be tens or hundreds of millions of dollars. That cost adds to a chip's cost, depending on the number of chips made. Ex: NRE cost for a chip may be $10,000,000. If 10,000 chips are made, $10,000,000/10,000 = $1000 needs to be added per chip to cover NRE cost. But if 1,000,000 chips are made, only $10,000,000/1,000,000 = $10 need be added. Thus, mass-producing a chip allows for lower chip costs, since NRE cost can be spread.

PARTICIPATION ACTIVITY 6.2.1: High NRE cost favors mass-producing chips.

Animation captions:

1. The non-recurring engineering (NRE) costs for a chip's circuits may be 10 million dollars. Like the engineering costs for a mass-produced car, those costs only occur once.
2. That design is used to create identical chips. If only 10,000 chips are made and sold, then $1000 must be added to each chip to cover the $10M NRE costs.
3. But if 1,000,000 chips are sold, only $10 need be added to cover the $10M NRE. Hence, chip makers strive to design fewer chips and then mass produce those chips.

PARTICIPATION ACTIVITY 6.2.2: NRE and mass-produced chips.

Assume NRE cost is evenly distributed among chips sold.

1) NRE cost for a given chip is $10 million. If 1 million chips are sold, how much is added per chip to cover NRE cost?

Check **Show answer**

2) NRE cost for a given chip is $10 million. If 10 million chips are sold, how much is added per chip to cover NRE cost?

Check **Show answer**

3) A chip maker currently sells a 128 MB chip for $10. A customer wants 10,000 256 MB chips for $50 per chip. The NRE cost for designing a 256 MB chip will be $10 million. Should the chip maker design a new 256 MB chip for the customer? Type yes or no.

Check **Show answer**

Low yield of large chips encourages smaller chips

The chip manufacturing process simultaneously creates multiple identical chips on a silicon wafer (a round silicon slice), and then each chip is cut out. The manufacturing process may take hours or days and is costly. Unfortunately, a wafer may have defects, like a broken wire, that render some chips unusable. ***Yield*** is the percentage of chips that are usable and free from significant defects. Ex: If 40 of 50 chips on a wafer are usable, yield is 40/50 = 80%. Larger chips are more likely to enclose a defect and thus have lower yield.

PARTICIPATION ACTIVITY 6.2.3: Smaller chips are less likely to enclose a defect, giving higher yields, and ultimately lower cost.

Animation captions:

1. Chips are manufactured as one large silicon wafer and then cut out, like baking one large cookie that is then cut into smaller cookies.
2. Customers may want large chips, like a 4 GB memory.
3. Unfortunately, defects (shown as X) occur on wafers, so only some chips are usable. The wafer manufacturing cost is distributed across the usable chips ($30 for this 4 GB chip)
4. With smaller chips, a defect impacts fewer chips, giving a higher "yield" of usable chips, which may lead to lower overall cost ($5 for these 1 GB chips, so only $20 for 4 GB).

For learning purposes, this section shows only a few chips per wafer. However, a real wafer may fit hundreds or even thousands of chips.

PARTICIPATION ACTIVITY 6.2.4: Yield.

Assume wafer manufacturing cost is evenly distributed among usable chips. Ex: If the wafer manufacturing cost is $50 and a wafer has 10 usable chips, the cost is $50 / 10 = $5 per chip.

1) A wafer holds 50 chips and costs $100. What is the cost per chip, assuming all chips are usable?

Check **Show answer**

2) A wafer holds 50 chips. 10 defects appear on the wafer, making 5 chips unusable. How many usable chips result?

Check **Show answer**

3) A wafer holds 50 chips. 20 defects cause 15 to be unusable. What is the yield? Answer as: 50%

Check **Show answer**

4) An $80 wafer holds 50 small chips. 10 defects cause only 40 chips to be usable (80% yield). What is the wafer cost per usable chip?

Check **Show answer**

5) An $80 wafer holds 20 large chips. 10 defects cause only 10 chips to be usable (50% yield). What is the wafer cost per usable chip?

Check **Show answer**

6) If a wafer just holds one massive chip, and any defect renders a chip unusable, how many defects can be tolerated?

Check **Show answer**

Example 6.2.1: Memory card composed of multiple memory chips.

If one looks inside a computer, one is likely to find memory chips arranged in an array to form a larger memory (described in another section). Below is a memory card from a personal computer, with 16 4 Gb chips (8 on each side of the card). Those 16 chips form a 16*4 = 64 Gb memory, meaning an 8 GB memory. Building a larger memory by composing smaller memory chips is more economical, as described above.

6.3 Composing memory

Composing memory chips horizontally

Because small memory chips may be cheap and widely available due to chip economics, building a memory by composing smaller memory chips is often cheaper than buying a single larger chip.

A designer may wish to create a wider memory, in which case memory chips can be composed horizontally, as below.

PARTICIPATION ACTIVITY 6.3.1: Composing two memory chips horizontally into a memory with twice as many bits per location.

Animation captions:

1. A designer may want a memory with 1 G locations each 16 bits wide, but only have 1G x 8 (1 GB) memory chips available.
2. The designer can use two 1 GB chips, splitting the 16-bit data input and output into the left 8-bits and the right 8-bits.
3. The address and control lines are simply connected to both memory chips. The two 8-bit-wide memories act like one 16-bit memory.

PARTICIPATION ACTIVITY 6.3.2: Composing memory chips horizontally.

1) A designer wishes to build a 32-bit wide memory using 1G x 8 memory chips. How many memory chips are needed?
   - ○ 4
   - ○ 8

2) A memory with 1 G addresses has 30 address inputs. If two 1G x 8 memory chips are composed horizontally to build a 16-bit-wide memory, how many address lines connect to the left chip?
   - ○ 29
   - ○ 30

3) A designer wishes to build a 10-bit wide memory using any number of 8-bit wide memory chips. How many chips are required?
   - ○ 2
   - ○ Not possible

Composing memory chips vertically

A designer may wish to create a memory with more locations, in which case memory chips can be composed vertically, as below.

For this purpose, when read is not enabled, a memory chip outputs neither 0's nor 1's, but electrically outputs nothing, allowing different rows' read data output lines to simply be connected when creating a larger memory.

PARTICIPATION ACTIVITY 6.3.3: Composing two memory chips vertically into a memory with twice as many locations.

Animation content:

undefined

Animation captions:

1. A designer can compose two 1G x 8 memory chips vertically to form a 2G x 8 memory. The 8 wdata inputs are connected to both chips. The 8 rdata outputs also connect.
2. A 2 GB memory has 31 addresses, but each 1 GB memory has just 30. Bit 30 (the 31st bit) controls a 1x2 decoder that activates either the upper or lower chip.
3. If an address is less than 1G (starts with 0), the top memory should be activated.
4. If an address is greater than 1G (starts with 1), the bottom memory should be activated.
5. Each ren and wen enable input is thus ANDed with one of the decoder outputs, so that the one appropriate memory chip will be active based on the address. A 2 GB memory is achieved.
6. An even larger memory uses more of the leftmost address bits and a larger decoder. Ex: A 4 GB memory has 32 address bits. Bits 31 and 30 control a 2x4 decoder that activates one of four 1 GB chips.

PARTICIPATION ACTIVITY 6.3.4: Composing memory chips vertically.

1) Two 1 GB memory chips are composed into a 2 GB memory. What size decoder is needed? Type 1x2, 2x4, or 3x8.

Check **Show answer**

2) Two 1 GB memory chips are composed into a 2 GB memory. Which address bit controls the decoder? Type 33, 32, 31, or 30.

Check **Show answer**

3) 256-byte memory chips (8 address inputs) are composed into a 1 KByte memory (10 address inputs). What size decoder is used? Type 1x2, 2x4, 3x8, or 4x16.

Check **Show answer**

4) 256-byte memory chips (8 address inputs) are composed into a 1 KByte memory (10 address inputs). What address bits control the decoder?
Type either:
10, 9
9, 8
9, 8, 7
9

Check **Show answer**

5) Eight 1 KB memory chips (10 address inputs) named C0, C1, ..., C7 (top to bottom) are composed into an 8 KB memory (13 address inputs). Which chip is active for address 1110110000000?

Check **Show answer**

6) A designer has 1 KB (1K x 8) memory chips available. The designer wants to create a 4K x 24 memory. How many 1 KB chips will be needed?

Check **Show answer**

6.4 Cache basics: Part 1

Off-chip and on-chip memory

Because a processor's chip has limited space, a processor's memory commonly exists on a separate chip (or chips). Ex: A 32-bit processor may have a 4 GB external memory chip (32 addr bits can specify 2^{32} = 4,294,967,296 words). Accesses between chips is slower than within a chip, due to large resistances and capacitances of external chip pins and wires causing signals to switch more slowly, and/or due to using DRAM (which is slower than SRAM) for smaller size and lower cost. For faster access, a processor may have a small on-chip memory, called a ***cache*** (pronounced "cash"), having a copy of likely-to-be-accessed memory items, for fast access, akin to a short list of "favorite" phone contacts. Cache is usually SRAM.

Initially, a cache is empty. Reading items fills the cache with copies of memory items. For later reads, the system checks the cache first; if an item's copy exists in the cache (a ***hit***), the system quickly reads that copy; else (a ***miss***) the system first copies the item from memory to cache.

A hit may require only 1 clock cycle, while a miss may require more, perhaps 10 cycles or more. For 1000 accesses, off-chip memory accesses would require 1000*10 = 10,000 cycles. But once a cache is filled, then if 95% of accesses are cache hits, 1000 accesses would only require 0.95*1000*1 + 0.05*1000*10 = 1,450 cycles.

PARTICIPATION ACTIVITY 6.4.1: Cache is a small fast on-chip memory for quick access to likely-to-be-accessed items.

Animation captions:

1. Memory is large, so may not fit on chip. Accessing off-chip memory is slow.
2. A cache is a small fast on-chip memory (perhaps 1/1000th the size of memory). Copies of items are kept in cache for fast subsequent access (perhaps 10x faster).
3. An item found in cache is a "hit". An item not found is a "miss". If the cache is full, existing items may be replaced.

PARTICIPATION ACTIVITY 6.4.2: Cache basics.

1) Relative to the processor, a cache typically resides _____.
   - ○ on-chip
   - ○ off-chip

2) Cache is usually built using _____.
   - ○ SRAM
   - ○ DRAM

3) Compared to memory, a typical cache might be _____.
   - ○ 10x smaller
   - ○ 1000x smaller
   - ○ 10x larger

4) If 1000 memory accesses require 10,000 cycles, how long might 1000 accesses to cache require (assuming the items being accessed are already in the cache)?
   - ○ 100,000 cycles
   - ○ 10,000 cycles
   - ○ 1,000 cycles

5) If an item is NOT found in cache, a cache ____ occurs.

- hit
- miss
- fault

6) Simple processor X only requires 1 MB of memory, which can fit entirely on-chip with X as SRAM. Is a cache necessary?
   - Probably
   - Probably not

7) Simple processor Y is very slow relative to memory acccesses. Is a cache necessary?
   - Probably
   - Probably not

Tag bits

A ***direct-mapped cache*** directly maps memory addresses to cache addresses using a subset of address bits (called the ***index***), storing the remaining bits (called the ***tag***) in the cache entry. To check if a memory address' item is in cache, the system uses the index bits as the cache address, and then compares that cache entry's tag bits, as below.

Below, each memory item is one byte. Data memory is shown, but the same applies to instruction memory.

PARTICIPATION ACTIVITY 6.4.3: A direct-mapped cache (trivially small, for learning).

Animation content:

undefined

Animation captions:

1. This data memory has 32 addresses, and the cache has 4 addresses. Both are trivially small, for learning purposes.
2. Each memory address can be mapped to a cache address using the rightmost two bits ("index"). Ex: Addresses ending in 00 map to index 00. Likewise for 01, 10, and 11.
3. When a memory item is accessed, a copy is put in cache, for fast access later. The second access to 01000 is fast (a "hit") because the item is already in the cache.
4. The remaining address bits ("tag") are included to indicate which of the 8 possible memory addresses is currently present. If tags don't match, a copy must be obtained.

A cache's design uses a comparator for the tag comparisons. Some additional logic uses the copy in the cache on a hit, or gets a copy from memory first on a miss.

PARTICIPATION ACTIVITY 6.4.4: Direct-mapped cache.

Consider the cache above.

1) Memory address 11100 maps to cache address ___.
   - 11
   - 111
   - 00

2) Memory address 10000 maps to cache address ___.
   - 10
   - 00

3) Memory addresses 00010, 00110, 01010, 01110, 10010, 10110, 11010, and 11110 all map to cache address 10 (the righmost two bits are 10). In a direct-mapped cache, how many of those memory addresses can simultaneously have a copy residing in cache?
   - 0
   - 1
   - 2

4) What is the tag for the data currently at cache address 10?
   - 10
   - 011
   - 6207

5) What is the memory address for the data currently in cache address 00?
   - 000000
   - 999
   - 10100

6) Suppose the next processor access is to memory address 11110. What will appear next at cache address 10? (data / tag)
   - 6207 / 011
   - 42 / 011
   - 42 / 111

PARTICIPATION ACTIVITY 6.4.5: Direct-mapped cache hits and misses.

Consider the cache above, initially empty. The accesses below occur in sequence. Indicate

whether each is a hit or miss.

1) 01000
   - Hit
   - Miss

2) 00111
   - Hit
   - Miss

3) 01100
   - Hit
   - Miss

4) 00111
   - Hit
   - Miss

5) 01100
   - Hit
   - Miss

6.5 Cache basics: Part 2

Valid bits

Upon system startup, a cache's entries are invalid. Thus, each cache entry has a ***valid*** bit that is initially set with 0, then set with 1 upon the first copy of an item into that entry (staying 1 from then on).

PARTICIPATION ACTIVITY 6.5.1: A direct-mapped cache (trivially small, for learning).

Animation captions:

1. Initially, the cache has no copies of memory items, so all valid bits are set with 0.
2. The first access to a cache address sees that Valid is 0 (ignoring tag bits), so the item is copied from memory. The Valid bit is set with 1, staying 1 from then on.
3. If Valid is 0, a copy is made. If Valid is 1, the tag bits are checked to see if a copy is needed. (In this case, the tag matches, so no copy is needed; the access is a hit).

PARTICIPATION ACTIVITY 6.5.2: Valid bits.

A four-address cache has the following entries (Addr: Data, Tag, Valid)

00: 55, 010, 1
01: 99, 111, 0
10: 6207, 011, 1
11: 0, 000, 0

1) An access to cache address 01 will cause a copy from memory.
   - Yes
   - No
   - Maybe

2) After an access to cache address 01, Valid's value is ___.
   - 0
   - 1
   - Unknown

3) An access to cache address 10 will cause a copy from memory.
   - Yes
   - Maybe

4) When will cache address 00's Valid bit be changed from 1 to 0?
   - When tags mismatch
   - Never

Spatial locality and blocks

If a program accesses a memory location, then the program will likely soon access adjacent locations too, known as ***spatial locality***. For accesses to instructions, spatial locality is high due to programs typically executing instructions in sequence. For accesses to data, programs commonly access array elements in sequence (for example).

Thus, if a memory item doesn't exist in cache, the system usually copies that item plus several adjacent items, known as a ***block*** (or ***line***), into the cache. Typical block sizes are 16, 32, or 64 bytes. Copying entire blocks rather than single memory items can reduce cache misses. Some address bits are used to ***offset*** into the block to get the proper item.

PARTICIPATION ACTIVITY 6.5.3: A direct-mapped cache with a block size of 2 bytes.

Animation captions:

1. Typically multiple bytes, called a block, are copied at once. Here, if 01000 is to be copied, both 01000 and 01001 are copied (block size is 2 bytes).
2. The rightmost address bit is the "offset" into the block. The next two bits are the cache address, or "index". The remaining leftmost bits are the "tag".
3. Access of 01001: Index is 00. Tag bits match, so the data is present (a cache hit). Offset is used to get the correct item within the block.

PARTICIPATION ACTIVITY 6.5.4: Cache blocks.

1) For the above cache, in memory address 00110, the 11 is the _____.
   - Index
   - Offset
   - Tag

2) For the above cache, in memory address 11001, the rightmost 1 is the _____.
   - Index
   - Offset
   - Tag

3) Given a cache whose block size is 4 bytes, how many address bits specify the block offset?
   - 1
   - 2
   - 4

4) If a cache has 16-byte block size, and 4K addresses, what is the total capacity of the cache?
   - 16 bytes
   - 4 Kbytes
   - 64 Kbytes

5) Given a 16-bit memory address. Suppose 3 bits are used for offset (for an 8-byte block size), and 8 bits for index (for 256 cache addresses). How many bits is each entry's tag?
   - 3
   - 5
   - 8

6) Given an 8-bit memory address, and a 4-byte block size. If address 11000001 is to be copied from memory to cache, what four locations will be copied for the block?

- 11000000
  11000001
  11000010
  11000011
- 11000001
  11000010
  11000011
  11000100

7) Given a 4-byte block size, how many tags will be present in one cache address' entry?
   - 1
   - 2
   - 4

8) Given a cache with 256 entries and a block size of 32 bytes, how many bits are used for each entry's tag?
   - 5
   - 8
   - Can't be determined

Write policies

The above material only discussed a processor reading memory, but writes may also occur. A straightforward cache write approach is called ***write through***, wherein when a processor writes data to a cache, the system also writes that data to memory.

A faster cache write approach is called ***write back***, wherein when a processor writes data to the cache, the system marks that cache block as ***dirty***, but doesn't write memory. Then, when a dirty block is being replaced, the system writes the block to memory first.

For frequent writes and reads of a block in cache, a write back approach is faster due to writing memory less frequently. Write through is used when the cache and memory cannot be temporarily inconsistent, such as when multiple processors share data in memory.

PARTICIPATION ACTIVITY 6.5.5: Write back is faster than write through but leaves memory out of date for some time.

Animation content:

undefined

Animation captions:

1. The cache has 01100 with data 999, then the processor writes 01100 with 777. The write is only to cache, not DM, so 01100 in DM is now out of date. The dirty bit is set.
2. A later read of 01100 reads 777 from cache (fast, like 1 cycle), which is correct, even though DM location 01100 is out of date.
3. When 01100 is replaced in cache, if Dirty is 1, data is first "written back" to DM. If DM access is 15 cycles, then total is 15 (DM write back) + 15 (DM read) + 1 (cache read).

4. With write through, each write may take 16 cycles (1 to write cache, 15 to write memory). With write back, each write is 1 cycle, until replacement, which is 31 (15 + 15 + 1).

PARTICIPATION ACTIVITY 6.5.6: Cache write approaches.

1) In write through, if an item exists in the cache, a write to that item is made to the _____.
   - cache only
   - memory only
   - cache and memory

2) In write back, if an item exists in the cache, a write to that item is made to the _____.
   - cache only
   - memory only
   - cache and memory

3) If a write to cache takes 1 cycle and a write to memory takes 5 cycles, how many cycles would 10 writes require using write back, assuming the item remains in the cache?
   - 5
   - 10
   - 50

4) If a write to cache takes 1 cycle and a write to memory takes 5 cycles, how many cycles would 10 writes require using write through, assuming the item remains in the cache?
   - 10
   - 50
   - 60

5) Assume a write back approach, 1 cycle cache reads, and 5 cycles to read or write a block in memory. A read of an item yields a miss, and the existing block to replace has a dirty bit of 0. What is the total cycles for the read?
   - 1
   - 6
   - 11

6) Assume a write back approach, 1 cycle cache reads, and 5 cycles to read or write a block in memory. A read of an item yields a miss, and the existing block to replace has a dirty bit of 1. What is the total cycles for the read?

- 1
- 6
- 11

7) Assume a write through approach, 1 cycle cache reads, and 5 cycles to read or write a block in memory. A read of an item yields a miss. What is the total cycles for the read?

- 6
- 11
- Need to know dirty bit's value

8) If two processors access the same memory, those processors' caches should most likely use a _____ approach.

- write back
- write through

6.6 Set-associative cache

Set-associative cache

When a memory block doesn't exist in the cache, the system copies the block into the appropriate cache address, replacing whatever previous block existed there. In some programs, two blocks may keep replacing each other in the cache, known as ***thrashing***.

To reduce thrashing, a cache can be designed to allow two blocks per cache address, known as a ***2-way set-associative cache***. To check if a memory block exists, the system checks the tags of both blocks.

PARTICIPATION ACTIVITY 6.6.1: A 2-way set-associative cache (trivially small).

Animation content:

undefined

Animation captions:

1. Sometimes the processor accesses memory addresses in a pattern that results in frequently-accessed items mapping to the same cache address, causing thrashing.
2. A solution is to have 2 blocks per cache entry, known as a "2-way set-associative" cache. For an access, if one of the block tags match, that block's data is used.
3. If cache size of the 2-way cache should be the same as direct mapped, the number of cache addresses is halved, meaning one fewer index bit, and one more tag bit.

Common associativities are 1-way (direct-mapped), 2-way, 4-way, or 8-way, and higher associativities (like 64) are sometimes seen too. Higher associativities yield better hit rates but require more comparators (one per way) plus more wires and internal logic, which may slow cache access.

A cache where all memory blocks are looked up just by tags (so the cache essentially has only one cache address) is called a ***fully associative cache***. While having the best hit rate, such caches are rare due to being expensive to build and/or slow.

PARTICIPATION ACTIVITY 6.6.2: 2-way set-associative cache hits and misses.

Consider the above 2-way set-associative cache (in blue) with two cache addresses (0 and 1), initially empty. The accesses below occur in sequence. Indicate whether each is a hit or miss

1) 01000
   - ○ Hit
   - ○ Miss

2) 00111
   - ○ Hit
   - ○ Miss

3) 01100
   - ○ Hit
   - ○ Miss

4) 00111
   - ○ Hit
   - ○ Miss

5) 01100
   - ○ Hit
   - ○ Miss

6) 01000
   - ○ Hit
   - ○ Miss

The above caches and memory were trivially small, for learning purposes. Below is a more typical cache and memory arrangement.

PARTICIPATION ACTIVITY 6.6.3: A more typical memory and cache arrangement: 4 GB memory, 8 KB 2-way set-associative cache with 16-byte blocks.

Animation captions:

1. A typical memory size is 4 GB, which has 32-bit addresses.
2. This typical cache has size 8 KB. The cache has 16-byte blocks, and is 2-way set-associative.
3. For address 00..0011111111110000: 16-byte blocks means the right 4 bits (0000) are the block offset. 8192 / 16 / 2 ways means 256 cache addresses, so 8 index bits. The remaining leftmost 20 bits are the tag.
4. An access first uses the index as the cache address. If a matching tag is found in one of the 2 ways, the offset is used to get the appropriate byte (or word).

PARTICIPATION ACTIVITY 6.6.4: Typical memory and cache arrangement.

Consider the above 4 GB memory and 8 KB cache.

1) Considering data bytes, the cache size is about what fraction of the memory size? Type either 0.2, 0.02, or 0.000002.

Check **Show answer**

2) A memory address is how many bits?

Check **Show answer**

3) A tag is how many bits?

Check **Show answer**

4) How many memory address bits are relevant to the cache?

Check **Show answer**

5) How many more bits were needed in the tag due to having a 2-way set-associative cache rather than a

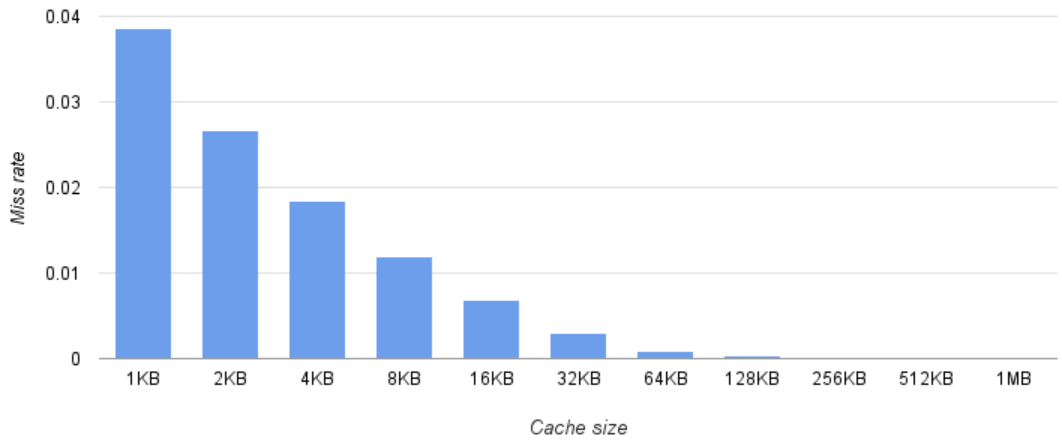
direct-mapped cache?

Check **Show answer**

Configuring a cache

A processor designer must select a size for the cache. A larger size yields fewer misses, but occupies costly chip space. The figure below shows average miss rates for 26 common "benchmark" programs for cache sizes ranging from 1 KB to 1 MB. For those programs, a cache larger than 64 KB has little additional improvement.

Figure 6.6.1: Average cache miss rate for 26 common programs on different sized direct-mapped caches.



Source: Univ. of Wisconsin.

PARTICIPATION ACTIVITY 6.6.5: Cache miss rate for different cache sizes.

Consider the above chart showing average miss rates for 26 common programs on different sizes of direct-mapped caches.

1) What is the approximate miss rate on a 1 KB cache?
   - 4%
   - 0.04%
   - 0%

2) What is the approximate miss rate on an 8 KB cache?

- 1%
- 0.01%

3) Suppose a hit takes 1 clock cycle, and a miss takes 20 cycles. How many total cycles would 1000 accesses take with a miss rate of 4%?

- 1,000
- 1,760
- 20,000

4) Suppose a hit takes 1 clock cycle, and a miss takes 20 cycles. How many total cycles would 1000 accesses take with a miss rate of 1%?

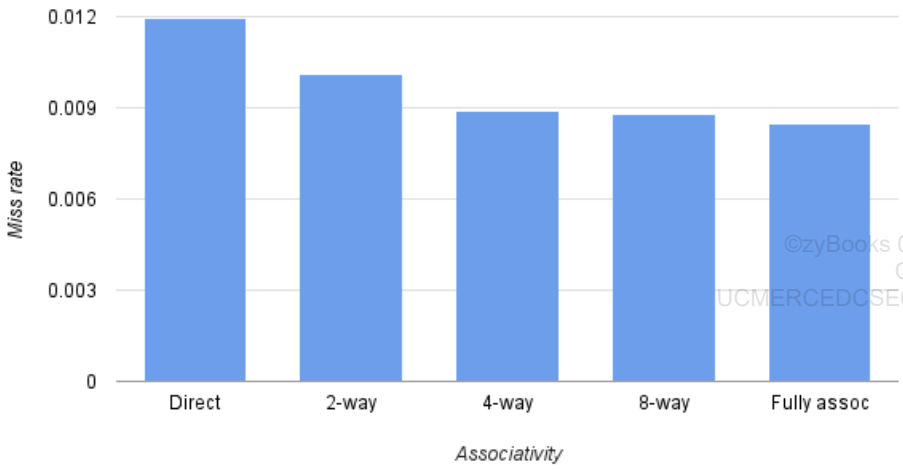
- 1,000
- 1,190

5) If one cache yields a program execution of 1760 cycles and a larger cache yields 1190 cycles, what is the speedup of the larger cache (speedup = slower/faster).

- 570
- 1.5

For a given cache size, a designer must select the associativity. More associativity yields fewer misses, but the extra comparison logic occupies costly chip space.

Figure 6.6.2: Average cache miss rate for 26 common programs on an 8 KB cache with different associativities.



Source: Univ. of Wisconsin.

PARTICIPATION ACTIVITY 6.6.6: Cache associativity.

Consider the above bar chart for 26 common programs on an 8 KB cache with different associativities.

1) The miss rate for a direct-mapped cache is about 1.2%, and for a 4-way cache is about 0.9%.
   - ○ True
   - ○ False

2) Increasing associativity from 4-way to 8-way yields a substantial improvement in miss rate.
   - ○ True
   - ○ False

3) The fully-associative cache provides the lowest possible miss rate for the given cache.
   - ○ True
   - ○ False

4) Any given program will exhibit no worse than 1.2% miss rate on the 8 KB cache.
   - ○ True
   - ○ False

A similar bar chart can be drawn for different block sizes. Miss rates improve slightly from 16 bytes to 32 bytes to 64 bytes, then become worse for larger block sizes. Importantly, the number of cycles for a miss increases for larger block sizes too.

A processor designer typically can choose from various cache sizes, associativities, and block sizes. A tool then generates the cache component to match the designer's choices, and that component can then be placed on the chip.

Larger cache sizes and higher associativities not only use costly chip space, but may also increase cache access times due to more circuitry and longer wires. That increased time may result in a slower clock cycle, thus hurting overall performance. Ex: A program may require 5000 cycles on small cache A and only 4000 cycles on larger cache B, but if the cycle time increase from 8 ns to 10 ns, then total time for A is 5000 * 8 = 40,000 ns vs. B's 4000 * 10 = 40,000 ns, meaning cache A is preferable due to being smaller.

PARTICIPATION ACTIVITY 6.6.7: Cache size and access time.

1) A program runs 200 cycles on cache X. Cache X yields a cycle time of 3 ns. What is the total time in ns?

Check **Show answer**

2) A program runs 150 cycles on cache Y. Cache Y yields a cycle time of 5 ns. What is the total time in ns?

Check **Show answer**

Temporal locality and replacement policy

When copying a block into a 2-way associative cache, the system must decide which of the two existing blocks to replace, known as the ***cache replacement policy***. If a memory address was recently accessed, that word will likely be accessed again soon, known as ***temporal locality***. Thus, a common cache replacement policy is ***LRU***, which replaces the ***least recently used*** block.

To implement LRU for a 2-way set-associative cache, a single bit can be associated with each cache address, indicating which way was least recently accessed. An access to way 0 sets the bit with 1. An access to way 1 sets the bit with 0. Then, if a block must be replaced, that bit indicates which way's block is the least recently used so should be replaced. More complex schemes exist for 4-way or higher associativities.

PARTICIPATION ACTIVITY 6.6.8: Implementing an LRU replacement policy using one bit per cache address.

Animation captions:

1. One LRU bit is added per cache address. (The initial value is arbitrary).
2. When a way is accessed, the bit is set to point to the other way.
3. If an access requires replacing an item, the way pointed to by the LRU bit is chosen for replacement. The bit is then set to point to the other way.

PARTICIPATION ACTIVITY 6.6.9: Cache replacement policy: LRU.

Refer to the above animation.

1) After the first access (10000), what is cache addr 0's LRU bit set with?
   - ○ 0
   - ○ 1

2) After the second access (11110), what is cache addr 0's LRU bit set with?
   - ○ 0
   - ○ 1

3) Does the third access (10000) have to replace a cache item?

- Yes
- No

4) Does the access to 01100 have to replace a cache item?
   - Yes
   - No

5) Suppose an additional access should be made, this time to 00010. Which way's item should be replaced?
   - 0
   - 1

6) How many total LRU bits exist in a 2-way set-associative cache having 256 addresses?
   - 256
   - 512

7) How many total LRU bits exist in a direct-mapped cache having 256 addresses?
   - 0
   - 256

6.7 Memory hierarchy

Computers often have multiple levels of memory/storage devices, known as a ***memory hierarchy***, with big and slow memory/storage for long-term storage, and smaller and faster memory for copies of currently-accessed items.

A ***drive*** (or ***hard drive***) is non-volatile storage (typically disk or flash) that holds a computer's program instructions and data files even when the computer is off. ***Non-volatile storage*** maintains bit values even without electrical power.

In a computer, ***memory*** is a volatile device (typically DRAM) that holds a copy of just the currently active program and data parts.

A ***cache*** is fast volatile memory (typically SRAM) on a processor chip that holds a copy of the most frequently or recently accessed pieces of a program and data. The cache concept can itself be repeated, like with a small/fast level 3 (***L3***) cache, a smaller/faster ***L2*** cache, and an even smaller/faster ***L1*** cache that the processor directly accesses.

PARTICIPATION ACTIVITY 6.7.1: Memory hierarchy.

Animation captions:

1. Programs and data files are stored on non-volatile drive. A program may be loaded into memory to start running. Loading from drive is very slow.
2. A program may run from memory, but accesses are slow. Instead, copies of currently-needed

parts reside in fast on-chip cache.
3. Multiple levels of cache may exist, with each closer level to the processor being even smaller and faster.
4. Because programs access the same and nearby items repeatedly (aka temporal and spatial locality), most accesses are fast. Accesses to memory or drive are relatively rare.

PARTICIPATION ACTIVITY 6.7.2: Memory hierarchy.

Consider the memory hierarchy above, having a drive, memory, L2 cache, and L1 cache.

1) A user installs a program called PacMan on a computer, adding to 50 already-installed programs. On what storage do those programs reside? Type drive, memory, L2, or L1.

Check **Show answer**

2) When a program is run, the program is first loaded from drive into what storage? Type memory, L2, or L1.

Check **Show answer**

3) A processor running a program wishes to access memory location 6000. Where does the system first look? Type drive, memory, L2, or L1.

Check **Show answer**

4) A processor running a program wishes to access memory location 7000. A copy is found in L1. How many cycles did the access take?

Check **Show answer**

5) A processor running a program makes 100 accesses to memory addresses. 90% are to L1 only, 9%

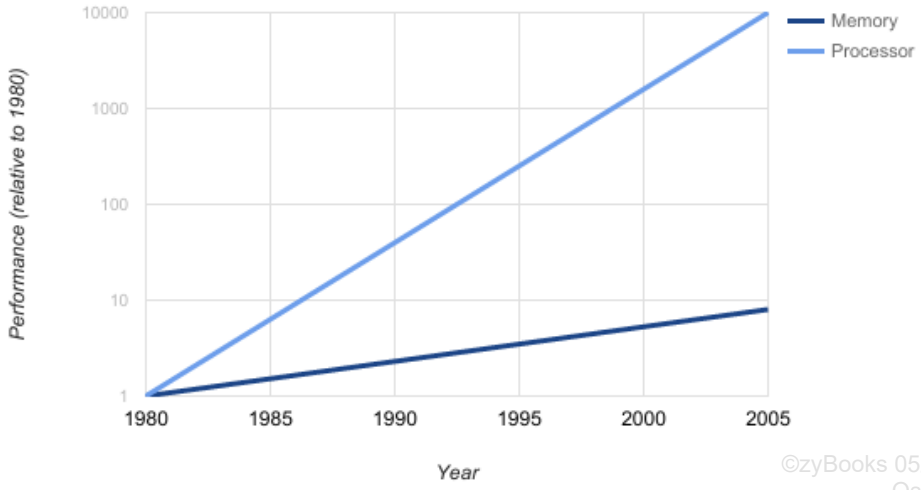
are to L2 and L1, and 1% are to memory, L2, and L1. What is the total number of cycles? Assume cycles simply add, so the 1 access that involves memory takes 10 + 2 + 1 = 13 cycles.

Check **Show answer**

6) Redo the above question assuming no L2 cache exists, so 90% of accesses are L1 hits, and 10% of accesses involve memory plus L1.

Check **Show answer**

Caches weren't needed in the 1980's when processors and memory were equal speeds. But then processor speed improvements outpaced memory speed improvements, meaning processors spent a lot of time just waiting for memory accesses to complete. Caches were introduced to solve that problem.



Source: Computer Architecture: A Quantitative Approach (Hennessy/Patterson)

6.8 Review: nMOS transistors

An ***nMOS*** transistor is a popular transistor type that conducts electricity when its control input is 1 and doesn't conduct when its control input is 0. An nMOS transistor is like a light switch, which conducts when in the on position, and doesn't conduct when in the off position.

PARTICIPATION ACTIVITY 6.8.1: A switch is either on (1) or off (0) (click the switch).

PARTICIPATION ACTIVITY 6.8.2: nMOS transistor.

Animation captions:

1. nMOS transistor conducts when control input is 1.
2. nMOS transistor does not conduct when control input is 0.

PARTICIPATION ACTIVITY 6.8.3: nMOS transistors.

1) Does an nMOS transistor conduct if the control input is 1?
   - Yes
   - No

2) Does an nMOS conduct if the control input is 0?
   - Yes
   - No

Exploring further:

- Transistor (Wikipedia)

6.9 RAM design

SRAM

An NxM SRAM consists of N locations each M bits wide. An SRAM could be built from N M-bit registers, with a decoder connecting to each register's load input for writes, and the decoder controlling a large mux for reads, but such

a design has excessive wiring and mux logic for large RAMs. Instead, each SRAM location is typically built from an array of cells, with each cell passing a word enable line and data line through.

PARTICIPATION ACTIVITY 6.9.1: SRAM design: Array of cells, and decoder.

Animation captions:

1. Conceptually, SRAM is an array of cells that each stores one bit. A decoder sets one word's enable (we) with 1 when write enable (wen) is 1. (Read enable not shown)
2. This address causes decoder output 4095 to be 1, causing the write data 10110001 to be stored in word 4095.

A common ***SRAM cell*** design uses two inverters to store a bit, and two transistors to let a 1 or 0 into that loop. Because each inverter is two transistors, each cell has six transistors.

PARTICIPATION ACTIVITY 6.9.2: SRAM 6-transistor cell basics.

Animation captions:

1. The basic idea of an SRAM cell is to store a bit in a two-inverter loop. The 1 becomes a 0, which becomes a 1 again, and the process repeats.
2. The question is how to get a bit into that loop. A transistor can allow a new bit into the loop. When word enable is 1, wdata's value enters the loop.
3. A detail: The previous value on the right may interfere with the new value. Another transistor is added, with wdata' (wdata's complement).
4. The word enable is fed through for the cell to the right. The data lines are fed through for the cell below.

Reads use an electrical technique beyond this section's scope. In short, when read-enable (ren) is 1, both data lines are set with 1. we is also set with 1, so the transistors on both sides conduct, causing either the left or right data line's voltage to be pulled down slightly. A special "sense amplifier" circuit detects which side was slightly pulled down, to determine whether a 0 or 1 was stored.

PARTICIPATION ACTIVITY 6.9.3: SRAM design.

Consider the above SRAM design.

1) How many cells exist in one word?

Check **Show answer**

2) How many total cells exist?

Check **Show answer**

3) For one cell, how many data lines enter from the top?

Check **Show answer**

4) In the animation, for one cell, a word enable (we) line enters from the left, and exits from the _____.

Check **Show answer**

5) For a cell, we is 1 and wdata is 1, causing 1 to pass through the left transistor and enter the left side of the inverter loop. What value will appear on the right side of the inverter loop?

Check **Show answer**

6) we is 1, wdata is 1, and 1 is on the left side of the inverter loop, and 0 on the right. Then we and wdata are set with 0. What value is on the left side of the inverter loop?

Check **Show answer**

7) A cell has 0 on the left and 1 on the right of the inverter loop, with we = 0. What value is presently stored in the cell?

Check **Show answer**

DRAM

A ***DRAM cell*** stores a bit as a charge on a capacitor, with one transistor that can pass current to the capacitor. With only 1 transistor and 1 capacitor, a DRAM cell is much smaller than an SRAM cell. But the capacitor in a DRAM cell leaks, requiring repeated reads and writes to ***refresh*** the cell, which is one reason DRAM access is slower than SRAM.

PARTICIPATION ACTIVITY 6.9.4: DRAM 1-transistor 1-capacitor cell basics.

Animation captions:

1. The basic idea of a DRAM is to store a 1 or 0 as a charge on a capacitor. Writing a 1 charges the capacitor's top plate, causing positive voltage there.
2. But capacitors leak. So the charge slowly leaks to ground. Thus, each DRAM cell must be "refreshed" (read, and written right back), typically within 60 ms.

PARTICIPATION ACTIVITY 6.9.5: DRAM cells.

1) How many transistors are in a DRAM cell?
   - 1
   - 2

2) Where is a 1 stored in a DRAM cell?
   - Transistor
   - Capacitor

3) To prevent a stored bit from leaking completely away, a DRAM repeatedly reads and then writes each cell, called a _____.
   - replenish
   - refresh

4) A DRAM is non-volatile.
   - True
   - False

5) A DRAM requires a controller to handle refreshes.
   - True
   - False

Variations of SRAM and DRAM cells exist, such as SRAM cells with 4 transistors and two resistors, but the above are the most common forms.

Exploring further:

- SRAM (Wikipedia)
- DRAM (Wikipedia)

6.10 ROM design

ROM using floating-gate transistors

Flip-flops, SRAMs, and DRAMs lose stored bits when electrical power is removed (*volatile* memory). In contrast, ***ROM*** (read-only memory) retains stored bits when power is removed (*non-volatile* memory), by using technology that usually has slow writes, so writes are less frequent than reads; hence the term "read-only" (a misnomer when writing is in fact possible). Writing to a ROM is called ***programming*** the ROM.

Bit storage in a ROM commonly uses a ***floating-gate transistor*** having a special region where electrons can be trapped, staying there even without power. Applying a large positive voltage traps the electrons (programming). A large negative voltage frees the electrons (erasing). Applying such large voltages long enough to trap or free electrons is slow, which is why writing a ROM is slow.

PARTICIPATION ACTIVITY 6.10.1: ROM using floating-gate transistors.

Animation content:

undefined

Animation captions:

1. A normal transistor does not conduct when the gate input is 0, and conducts when the gate is 1.
2. A floating-gate transistor has a special region called a floating-gate. The transistor can work like a normal transistor.
3. A large positive voltage causes electrons to tunnel into the special region. When that voltage is removed, the electrons are trapped. The transistor has been "programmed".
4. Now, the transistor won't conduct even with a 1 at the gate. The negative electrons "block" the positive 1 from switching the transistor to conduct.
5. In a ROM, for an address, one word line will be 1. The data lines become 1 for unprogrammed transistors, and 0 for programmed transistors (uses special circuitry).

Note: The above is a "logical" view of a ROM, oversimplifying electrical features beyond this material's scope.

PARTICIPATION ACTIVITY 6.10.2: ROM using floating-gate transistors.

1) An unprogrammed floating-gate transistor (no electrons trapped in the floating gate) ____.
   - acts as a normal transistor
   - acts opposite a normal transistor.
   - won't conduct
2) A programmed floating-gate transistor (electrons trapped in the floating gate)

____.

- acts as a normal transistor
- acts opposite a normal transistor.
- won't conduct

3) Programming a floating-gate transistor is done via a large _____.

- positive voltage
- negative voltage
- hammer

4) Erasing a floating-gate transistor is done via a large _____.

- positive voltage
- negative voltage

5) A 256x16 ROM would have how many word lines?

- 8
- 16
- 256

6) A 256x16 ROM would have how many floating-gate transistors?

- 16
- 256
- 4096

7) A ROM has 8 bits per location. Location 99 has only the first two transistors programmed (electrons trapped). What value is stored at location 99?

- 11000000
- 00111111
- 11111111

8) The ROM shown above would use _____ to achieve the 1's and 0's at the data output.

- normal logic
- special circuitry

ROM types

Numerous ROM types exist.

- ***Mask-programmed ROM***: The word line to bit line connections are hardwired during chip manufacturing, and can

never be changed.

- ***OTP ROM*** (one-time programmable ROM): The word line to bit line connections have a fuse that can be "blown" to break the connection. A user can program the device only once.
- ***EPROM*** (erasable programmable ROM): Programming uses large positive voltage to trap electrons in a floating-gate transistor, but erasing is done by placing the chip under ultraviolet light to provide the energy to free the trapped electrons. This ROM type preceded the more convenient EEPROM.
- ***EEPROM*** (electrically-erasable programmable ROM): Programming uses large positive voltage to trap electrons in a floating-gate transistor, and erasing uses a large negative voltage.
- ***Flash***: Programming uses large positive voltage to trap electrons in a floating-gate transistor, and erasing uses a large negative voltage to quickly erase entire blocks of locations at one time, like a "flash".

PARTICIPATION ACTIVITY 6.10.3: ROM types.

Animation captions:

1. A mask-programmed ROM has connections for 1's and 0's made during manufacturing, which involves passing light through "masks" to create wires on a chip.
2. A one-time programmable (OTP) ROM has fuses. Blowing some fuses but leaving others intact programs the 1's and 0's. Once blown, fuses can't be reconnected (hence "one-time").
3. EPROM uses a large positive voltage to trap electrons in a floating-gate transistor (FGT above). Erasing is done by placing chip package (having a window) under UV light.
4. EEPROM is like EPROM, but conveniently erased using a large negative voltage (no need to place chip under UV light).
5. Flash is like EEPROM, except entire blocks (thousands of words or more) can be quickly erased at once (in a "flash"), rather than one word at a time like EEPROM.

PARTICIPATION ACTIVITY 6.10.4: ROM types.

Match the ROM type with the most likely usage scenario.

If unable to drag and drop, refresh the page.

Flash **Mask-programmed ROM** **EPROM** **OTP ROM** **EEPROM**

Storing program instructions in a calculator that will be produced in the hundreds of millions.

Storing a unique ID number for a secure card key, which should never be changed.

Storing a program in a microprocessor chip being used for prototyping in an engineering lab, with the chip being reprogrammed a few times a day, in the 1980s.

Storing 256 phone numbers in a portable keychain device, each number reprogrammable by some button clicks.

A digital photo frame to which 8 photos can be uploaded, with the frame showing a different photo every minute.

Reset

Exploring further:

- ROM (Wikipedia)
- Flash (Wikipedia)

6.11 Virtual memory

Basic virtual memory

Memory is expensive. Thus, a computer may not physically have all the memory that a program requires. Ex: 32-bits can address 4 G locations, but a computer may only have 1 G of physical memory. ***Virtual memory*** makes a physical memory appear larger to a program, by keeping items on a drive and copying only a subset into physical memory as needed.

PARTICIPATION ACTIVITY 6.11.1: Virtual memory (sizes are small, for learning).

Animation content:

undefined

Animation captions:

1. Desired ("virtual") memory might be 8 KB (13-bit address), but physical memory may be 4 KB (12-bit address).
2. One can divide virtual memory into pieces called pages (here 8 pages, each 1 KB). An address' first three bits denote the page.
3. When the processor accesses a location, the location's entire page is copied into an arbitrary page in physical memory, which can hold 4 pages. A page table tracks that page's place in the PM.
4. That page's Valid bit is set with 1 in the page table to indicate the page is in physical memory.
5. The rest of the address is the offset into that page in physical memory. A memory management unit (MMU) manages the page table and pages.

For learning purposes, the above animation's memory sizes are unusually small. Typical sizes might be a 4 GB virtual

memory, 1 GB physical memory, and 4 KB page sizes. Thus, the virtual memory is divided into 4 GB / 4 KB = 1,000,000 pages.

A ***memory management unit*** (***MMU***) translates virtual addresses to physical addresses, keeping translations in a ***page table***. For each page, the page table indicates whether the page is in physical memory (using a Valid bit) and where the page is located in physical memory, as above. The table also indicates the page's location in the drive (not shown above). The page table may reside in the MMU or in physical memory.

If an access is to a page currently in physical memory, a ***page hit*** occurs; else a ***page fault*** occurs. For a page fault, the MMU copies the requested page from the drive to physical memory. Each page is placed into physical memory at one of several specific locations known as a ***page frame***.

PARTICIPATION ACTIVITY 6.11.2: Virtual memory.

Consider the virtual memory arrangement shown above.

1) How much smaller was physical memory than desired "virtual" memory?
   - 1/2
   - 1/8

2) Virtual memory was implemented in DRAM.
   - True
   - False

3) The sizes of virtual memory and physical memory necessitated page sizes of 1 KB.
   - True
   - False

4) The page table has 1 row per page.
   - True
   - False

5) If a processor wishes to read memory location 5000, the MMU first checks if location 5000's page is in physical memory. If so, the MMU finds that location and sends the value to the processor.
   - True
   - False

6) If a virtual memory is divided into 256 pages, how many of the leftmost virtual address bits indicate the page?

- 4
- 8

7) In the animation above, suppose the requested address is 0100001110000. What is the page?
   - 2
   - 1

8) In the animation above, the physical memory has how many page frames?
   - 4
   - 8

9) In the animation above, suppose the requested address is 1110000000100. Where is the location in physical memory?
   - 0000000100
   - To be determined

10) In the animation above, suppose the processor wishes to access 0000000000011. At what physical memory address is that item found?
    - 000000000011
    - 100000000011

Page replacement

If a program accesses an address (whether an instruction or data), the program is likely to soon access nearby addresses, known as ***spatial locality***. Spatial locality thus prefers larger pages. But larger pages take more space, requiring more frequent replacements. And larger pages take more time to replace. Designers choose a page size that strikes a balance.

When copying a new page to full physical memory, the MMU replaces an existing page, typically the ***least recently used*** (***LRU***) page. The MMU keeps recently accessed pages due to ***temporal locality***: Recently-accessed addresses are likely to be accessed again soon.

A ***dirty bit*** indicates whether any location within a page has been written while in physical memory. Upon being replaced, if the page's dirty bit is 1, the page must be copied to the drive. Else, the page can just be discarded.

PARTICIPATION ACTIVITY 6.11.3: Page replacement.

Animation captions:

1. As items are accessed, pages are copied into physical memory. Accesses (shown as green dot) will mostly be hits, due to spatial and temporal locality.
2. Eventually, all available page frames will be filled. The peach-colored frame was least recently used (used longest ago), so gets replaced.

PARTICIPATION ACTIVITY 6.11.4: Page replacement.

1) Increasing page size improves performance.
   - ○ True
   - ○ Maybe

2) Decreasing page size improves performance.
   - ○ True
   - ○ Maybe

3) Suppose page size is 4 KB. If a virtual address doesn't exist in physical memory, copying the page into physical memory might be expected to take 5 - 10 cycles.
   - ○ True
   - ○ False

4) Suppose a page X resides in physical memory, and has been read and written. Later, a new page must be loaded and will replace page X. Does X need to be copied back to the drive?
   - ○ Yes
   - ○ No

5) A page has a single dirty bit that is 1. When the page is replaced, the MMU can copy to the drive just the items that were written, rather than the entire page.
   - ○ True
   - ○ False

6) When replacing a page, the MMU is likely to choose the ___-recently used page for replacement.
   - ○ least
   - ○ most

©zyBooks 05/17/23 13:50 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

Virtual memory with multiple programs

Some computers run multiple programs simultaneously (seemingly; actually running each a little at a time), each such program known as a ***process***. Virtual memory enables an operating system to let each process think the process

owns the physical memory. The MMU maintains a page table for each process. When running a particular process, the OS informs the MMU to use that process' page table.

PARTICIPATION ACTIVITY 6.11.5: Virtual memory allows multiple processes to share one physical memory without interference.

Animation captions:

1. These two processes each have 8 KB virtual addresses from 0 to 8188, and reside on the drive (in specific addresses not shown).
2. The OS runs Process 1 for some time, then Process 2, then Process 1 again. Pages from both processes co-exist in physical memory. The MMU has a page table for each process.

PARTICIPATION ACTIVITY 6.11.6: Virtual memory and multiple processes.

Consider the animation above.

1) Process 1 uses 13-bit addresses, for addressing 8 KB. How many bits does Process 2 use for addresses?
   - ○ 12
   - ○ 13

2) The OS runs Process 1 for a short time. Then the OS runs Process 2 for a short time. Before doing so, what does the OS do with Process 1's pages in physical memory?
   - ○ Copies those pages back to drive.
   - ○ Keeps those pages in physical memory.
   - ○ Discards those pages.

3) Suppose Process 1 fills physical memory. When Process 2 runs, what happens?
   - ○ The MMU will replace one of Process 1's pages.
   - ○ The OS skips Process 2, resuming Process 1.

4) If replacing a dirty page of Process 1 by a page of Process 2, the Process 1 page might overwrite a Process 2 page on drive.
   - ○ True
   - ○ False

More virtual memory details

When an MMU converts a virtual address to a physical address, accessing a page table in physical memory (which is typically DRAM) may take several clocks cycles. Due to temporal locality, the same address may be accessed repeatedly. Thus, an MMU maintains a small buffer listing recent translations, called a ***translation lookaside buffer*** (***TLB***), whose access is much faster than accessing a page table. For a given address (say 5000), the MMU first checks the TLB (fast); if the address' translation isn't there, then the MMU accesses the page table.

Virtual memory also provides some security. OS processes are known as ***kernel*** processes, and can configure the MMU and certain special memory addresses like locations that configure a processor chip. In contrast, user processes cannot write to the MMU and may be restricted by the MMU from writing to special addresses, thus preventing user processes from altering a processor chip's configuration. Virtual memory can also be used to prevent a user processor from accessing another process' memory locations.

The MMU typically resides on a chip near a processor. If a cache exists, the MMU is between the processor and cache.

If one has studied caches, one may notice some similarities between virtual memory (VM) and caching. Both make copies of a subset of storage for faster access to frequently-accessed items. A VM's page is like a cache's index. A VM's page size is like a cache's block size, with part of the address in each being used as an offset. However, in a VM a page may be placed anywhere in physical memory, in contrast to direct-mapped or set-associative cache where each memory address maps to one cache address (thus, VM is more similar to a fully-associative cache). In VM, no tag is stored; instead, the mapping is kept in the page table.

PARTICIPATION ACTIVITY 6.11.7: Virtual memory details.

1) A TLB keeps a list of recent _____ for faster translations.
   - dirty bits
   - translations

2) A VM provides _____ security when executing multiple processes.
   - more
   - less

3) Relative to a processor, an MMU typically resides _____.
   - on-chip
   - off-chip

4) Virtual memory and cache have _____ similarities.
   - no
   - several

5) Virtual memory page sizes and cache block sizes are typically similar.
   - True
   - False

6.12 Error detection

Parity

When bit values are sent across wires, especially between chips, electrical interference can erroneously change a 0 to 1 or a 1 to 0.

Even parity is a method of detecting that a single bit error in a group of sent data bits, by sending a bit whose value makes the total number of 1's even. Ex: A sender (like a memory) wants to send 0111, which has three 1's. The sender appends a 1 to make four 1's total, yielding 10111. A receiver (like a processor) ensures the number of 1's is even, then discards the parity bit.

PARTICIPATION ACTIVITY 6.12.1: Even parity detects a single bit error.

Animation captions:

1. For 1110, the send sets the parity bit with 1 to yield an even number of 1's (four). The receiver ensures the total 1's are even, then discards the parity bit.
2. If the number of 1's is already even as for 0110, the sender sets the parity bit's value with 0. Zero 0's is even, so also gets a parity bit of 0.
3. If a single bit error occurs, the receiver notices an odd number of 1's, so discards the data. (The receiver may then ask the sender to resend the data).
4. The receiver doesn't know which bit is erroneous, just that a bit is wrong. The error may in fact be in the parity bit.
5. Two errors won't be detected because the number of 1's remain even.

Parity can detect 1, 3, 5, ..., errors. Parity cannot detect 2, 4, 6, ..., errors. Ex: For even parity, 0110 would be sent as 00110. Two errors might be 01111 or 00000; the receiver notes an even number of 1's (two) so believes the data is correct. Likewise, four errors might be 11000 or 01001, each having an even number of 1's.

Parity is popular due to only requiring one extra bit, and to being easy to implement in hardware (computable using a single XOR or XNOR gate).

Odd parity sets a parity bit with 0 or 1 to make total bits odd, rather than even. Both sender and receiver must agree on whether to use odd or even parity.

Example 6.12.1: ASCII codes and parity.

ASCII code is often said to use 8 bits, but in fact the leftmost bit is always 0. Ex: 'A' is 01000001. ASCII code actually uses just 7 bits; the 8th bit can be set as a parity bit. Below are a few ASCII encodings showing an even parity bit as the leftmost bit.

Parity bit	Bit code	Dec	Char
1	100 0000	64	@
0	100 0001	65	A
0	100 0010	66	B
1	100 0011	67	C
0	100 0100	68	D
1	100 0101	69	E
1	100 0110	70	F
0	100 0111	71	G

PARTICIPATION ACTIVITY 6.12.2: Parity bit.

Assume the parity bit is placed in the leftmost digit.

1) To send 1011, the even parity bit should be ___.
   - ○ 0
   - ○ 1

2) To send 1111, the even parity bit should be ___.
   - ○ 0
   - ○ 1

3) To send 1111001, the even parity bit should be ___.
   - ○ 0
   - ○ 1

4) Received bits are 10000111. Even parity is used. What is the received data?
   - ○ 0000111
   - ○ Error

5) Received bits are 11100000. Even parity is used. What is the received data?

- 1100000
- Error

6) Received bits are 11000110. Even parity is used. What ASCII character was received?

- F
- Error

7) To send 1011 using odd parity, the parity bit should be ___.

- 0
- 1

8) The sender and receiver must know how many bits are in a group of bits that includes a parity bit.

- True
- False

9) If a parity error is detected, the receiver knows which bit contains the error.

- True
- False

10) Parity detects any number of errors.

- True
- False

11) A sender can use a(n) _____ gate to compute an even parity bit.

- XOR
- XNOR

Checksum

A ***checksum*** is an error detection approach wherein data to be sent is divided into chunks whose sum is sent along; if the received chunks' sum doesn't match the sent sum, one or more errors occurred. A checksum is more costly to compute than parity, but can detect more errors.

PARTICIPATION ACTIVITY 6.12.3: A checksum approach can detect multiple errors.

Animation captions:

1. 16 bits are to be sent. The sender sums chunks, in this case 4-bit chunks, ignoring overflow. Ex:

1110 + 0110 + 0000 + 0110 is 1010.
2. The data and checksum are sent. Four errors occur.
3. The receiver calculates the data's checksum. If the calculated checksum differs from the received checksum, at least one error occurred.

PARTICIPATION ACTIVITY 6.12.4: Checksum.

1) 8 bits 00010011 are to be sent. The checksum computed on 4-bit chunks is ____ .
   - 0111
   - 0100

2) 12 bits are received in 4-bit chunks: 0000, 1111, and 0001 (checksum). Did an error occur?
   - Yes
   - No

3) Does checksum detect all errors?
   - Yes
   - No

4) Checksum is easier to calculate than parity.
   - True
   - False

An even more sophisticated form of checksum (not discussed here), known as a cyclic redundancy check (CRC), is even less likely to miss errors, at the expense of more calculation. The term "checksum" is used for nearly any way of combining data for error checking, even if no sum is involved.

Exploring further:

- Checksum (Wikipedia)
- CRC (Wikipedia)

6.13 Data compression

Basic compression idea

Given data represented as some quantity of bits, ***compression*** transforms the data to use fewer bits. Compressed data uses less storage, and can be communicated faster too.

The basic idea of compression is to encode frequently-occurring items using fewer bits. Ex: ASCII characters use 8 bits each, but instead more-frequently-occurring characters could use fewer bits and other characters use more bits.

PARTICIPATION ACTIVITY 6.13.1: The basic ideas of compression is to use fewer bits for frequent items (and more bits for less-frequent items).

Animation captions:

1. The text "AAA Go" as ASCII would use 6 * 8 = 48 bits. Such data is uncompressed.
2. Compression uses a dictionary of codes specifically for the data. Frequent items get shorter codes. Here, A (which is most frequent) is 0, space 10, G 110, and o 111.
3. Thus, "AAA Go" is compressed as 0 0 0 10 110 111. The compressed data uses only eleven bits, much fewer than the 48 bits uncompressed.

The example above has only four distinct characters (A, space, G, and o) so could be encoded using 2 bits (fixed-length code). However, the example is trivially simple, for learning. Actual text may use all ASCII characters so a fixed-length code would require 8 bits per character, but with varying-length codes as above where the frequent characters in the data might use fewer bits (like 4).

PARTICIPATION ACTIVITY 6.13.2: Basic compression.

Given the following dictionary:
00000000: 00
11111111: 01
00000010: 10
00000011: 110
00000100: 111

1) Compress the following: 00000000 00000000 11111111 00000100

Check **Show answer**

2) Compress the following: 00000011 00000010

Check **Show answer**

3) Decompress the following: 00 01 00

Check **Show answer**

4) Does any code in the dictionary contain another code starting from

the left of each code? Type yes or no. Ex: Consider a different dictionary having codes 1110 and 111; 1110 contains 111.

Check **Show answer**

5) Decompress the following, in which the spaces that were inserted above for reading convenience are absent: 0011000.

Check **Show answer**

Huffman coding

Huffman coding is a common compression technique that assigns fewer bits to frequent items, using a binary tree.

PARTICIPATION ACTIVITY 6.13.3: A binary tree can be used to determine the Huffman coding.

Animation captions:

1. Huffman coding first determines the frequencies of each item. Here, a occurs 4 times, b 3, c 2, and d 1. (Total is 10).
2. Each item is a "leaf node" in a tree. The pair of nodes yielding the lowest sum is found, and merged into a new node formed with that sum. Here, c and d yield 2 + 1 = 3.
3. The merging continues. The lowest sum is b's 3 plus the new node's 3, yielding 6. (Note that c and d are no longer eligible nodes). The merging ends when only 1 node exists.
4. Each leaf node's encoding is obtained by traversing from the top node to the left. Each left branch appends a 0, and each right branch appends a 1, to the code.

When merging, if two (or more) different node pairs would yield the same sum, the choice among those pairs is arbitrary.

PARTICIPATION ACTIVITY 6.13.4: Huffman coding example: Frequency counts.

Given the text "seems he fleed". Indicate the frequency counts.

1) s

Check **Show answer**

2) e

Check **Show answer**

3) Each of m, h, f, l, and d

Check **Show answer**

4) (space)

Check **Show answer**

PARTICIPATION ACTIVITY 6.13.5: Huffman coding example: Merging nodes.

A 100-character text has these character frequencies:
A: 50
C: 40
B: 4
D: 3
E: 3

1) What is the first merge?
   - D and E: 6
   - B and D: 7
   - B and D and E: 10

2) What is the second merge?
   - B and D: 7
   - DE and B: 10
   - C and A: 90

3) What is the third merge?
   - DEB and C: 40
   - DEB and C: 50

4) What is the fourth merge?
   - None
   - DEBC and A: 100

5) What is the fifth merge?
   - None
   - DEBCA and F

6) What is the code for A?

- 0
- 1

7) What is the code for C?

- 1
- 01
- 10

8) What is the code for B?

- 001
- 110

9) What is the code for D?

- 1110
- 1111

10) What is the code for E?

- 1110
- 1111

11) 5 unique characters (A, B, C, D, E) can each be uniquely encoded in 3 bits (like 000, 001, 010, 011, and 100). With such a fixed-length code, how many bits are needed for the 100-character text?

- 100
- 300

12) For the Huffman code determined in the above questions, the number of bits per character is A: 1, C: 2, B: 3, D: 4, and E: 4. Recalling the frequencies in the instructions, how many bits are needed for the 100-character text?

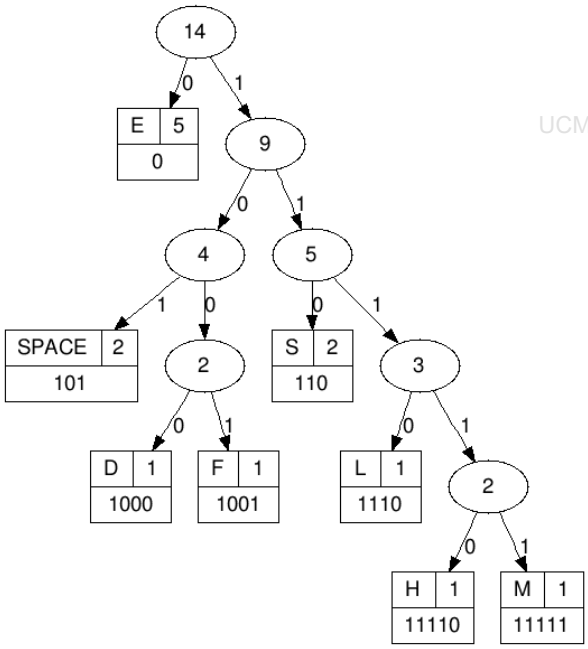
- 14
- 166
- 300

Note: For Huffman encoded data, the dictionary must be included along with the compressed data, to enable decompression. That dictionary adds to the total bits used. However, typically only large data files get compressed, so the dictionary is usually a tiny fraction of the total size.

Huffman tree web tools

This site has a tool that converts given text into a Huffman tree. For the earlier example of "seems he fleed", the site generated the following tree.

Figure 6.13.1: Huffman tree for the text: SEEMS HE FLEED.



Source: Huffman tree generator. No copyright held on generated images.

Table 6.13.1: Huffman and ASCII code table for the text: SEEMS HE FLEED.

Frequency	Chars	Huffman	ASCII
5	'E'	0	01000101
2	' '	101	00100000
2	'S'	110	01010011
1	'D'	1000	01000100
1	'F'	1001	01000110
1	'L'	1110	01001100
1	'H'	11110	01001000
1	'M'	11111	01001101

Note: CrypTool Project provides tools to generate a similar table comparing the sizes of Huffman and ASCII code.

For the text "SEEMS HE FLEED", Huffman code requires 39 bits while the ASCII code requires 112 bits.

- Huffman: 110 0 0 11111 110 101 11110 0 101 1001 1110 0 0 1000
- ASCII: 01010011 01000101 01000101 01001101 01010011 00100000 01001000 01000101 00100000 01000110 01001100 01000101 01000101 01000100

PARTICIPATION ACTIVITY 6.13.6: Huffman and ASCII code.

1) E's Huffman code is _____ .
   - 0
   - 01000101

2) _____ bits are needed to encode the 'SEEMS HE FLEED' using Huffman code.
   - 39
   - 112

Decompressing Huffman coded data

To decompress Huffman code data, one can use a Huffman tree and trace the branches for each bit, starting at the root. When the final node of the branch is reached, the result has been found. The process continues until the entire item is decompressed.

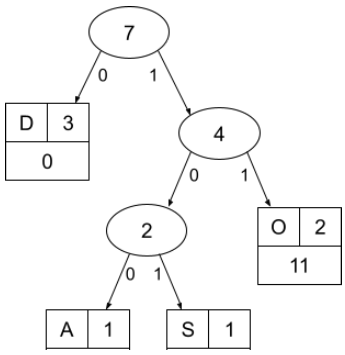
PARTICIPATION ACTIVITY 6.13.7: Decompressing Huffman code.

Animation captions:

1. The Huffman code is decompressed by first starting at the root. The branches are followed for each bit.
2. When the final node of the branch is reached, the result has been found.
3. Once the final node is reached decoding restarts at the root node.
4. The process continues until the entire item is decompressed.

PARTICIPATION ACTIVITY 6.13.8: Decompressing Huffman code.

Use the tree below to decompress 0111101000101.



100 101

1) What is the first decoded character?

Check **Show answer**

2) What is the second decoded character?

Check **Show answer**

3) 11 yields the third character O.
0 yields the fourth character D.

What is the next decoded character?

Check **Show answer**

4) What is the decoded text?

Check **Show answer**

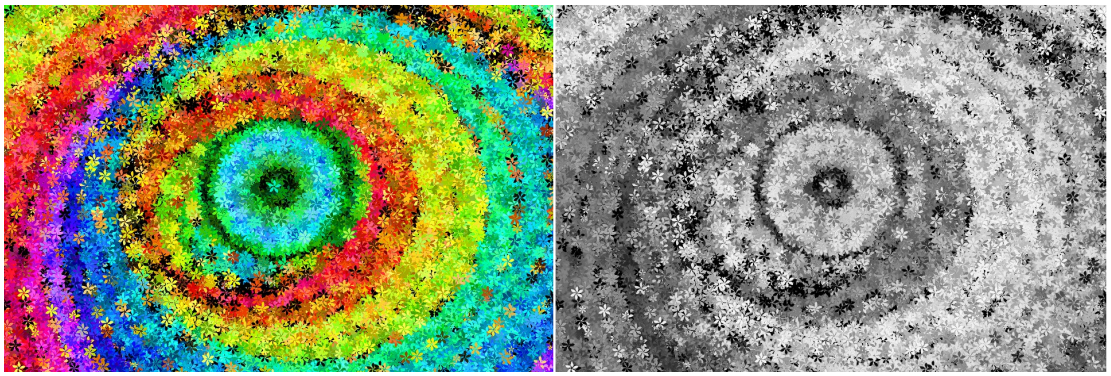
Text files, images, and videos

When compressing text files, an additional compression trick is used, wherein common sequences are treated as a pattern. Ex: "the box there is the right box" has the pattern "the" appear three times, and "box" twice. Thus, the dictionary may define a code for that pattern, so a code like 0010 might be listed not just as a letter like 't' but as a string like "the". Such a technique may be found in compression used to make ZIP files, for example.

Images may occupy much storage. Ex: An image with 1 million pixels and 3 bytes per pixel (for red, green, blue), may require 3 MB of storage. Each pixel is a number from 0 to 255. Some colors are much more common, like white (255, 255, 255) and black (0, 0, 0), so some numbers are much more common than others. Huffman coding is part of the common image compression technique known as JPEG. Image compression uses other techniques as well, which may lose some information (rounding, and discrete cosine transform, not discussed here) to achieve even greater compression.

Figure 6.13.2: Uncompressed image vs. a compressed image and color image vs. black and white image.

Source: zyBooks



Source: Pixabay

Video is a series of images known as frames. Thus, video compression techniques like MPEG include Huffman coding as well. Video compression also uses another compression technique: Because successive frames have only small differences, after an image, several successive frames may be represented just by the differences from the previous frame.

PARTICIPATION ACTIVITY 6.13.9: Text files, image, and video compression.

1) When compressing text, a dictionary may produce a code for a single character or a string.
   - True
   - False

2) A compressed image takes up the same amount of storage as an uncompressed image.
   - True
   - False

3) A common video compression technique involves only changing components that are different from the previous video frame.

- ○ True
- ○ False

Exploring further:

- Huffman coding (Wikipedia)
- Huffman tree generator (huffman.ooz.ie)
- Comparison of Huffman code and ASCII code (cyrptool-online.org)

©zyBooks 05/17/23 13:50 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023