Today's Topics

- Memory organization and byte ordering
- Using parity to detect errors in memory
- Detection of errors vs. correction of errors
- Hamming Codes
- Using cache memory to improve performance
- Cache effectiveness, performance, policies
- Boolean Logic Operators
- Solving Problems using Boolean Logic
- Equivalence of Boolean Expressions
- Negative Logic

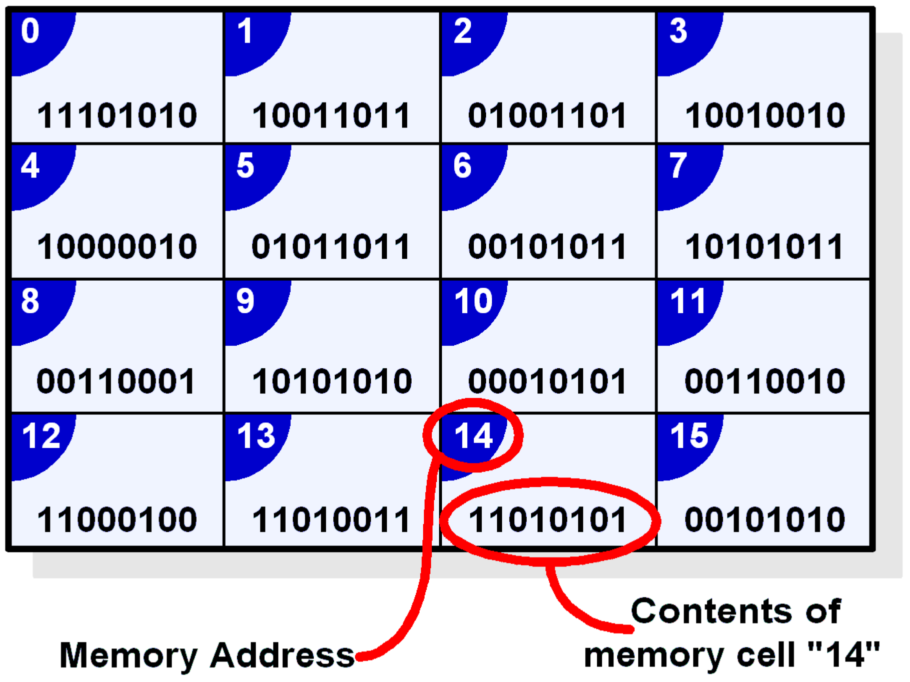
Primary Memory

The information in memory is organized into groups of bits. Information is read from or written to memory a group at a time. For example, if memory is organized as bytes (groups of 8 bits), then every memory read or write transfers 8 bits at the same time.

A group of memory bits is called a memory cell or memory location.

When writing information to memory or reading it from memory, you need to identify which memory cell is being accessed. This is done using a memory address.

Every cell in memory has a unique memory address which starts with “0” for the very first cell in memory and goes up by one for each additional cell.

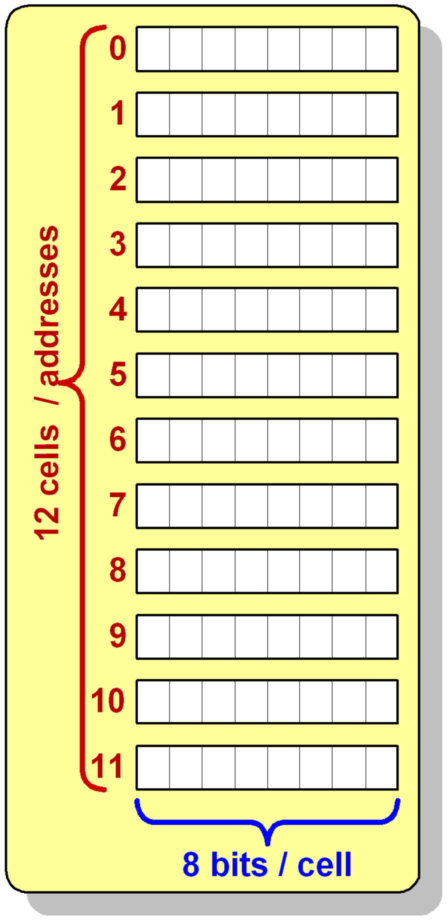


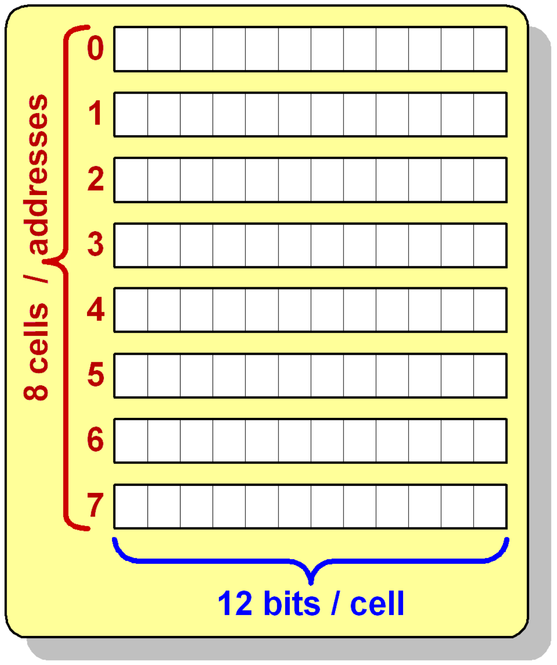
Note the difference between memory *addresses* and memory *contents*. Addresses are not actually *stored* in memory. An address is like a locker number – it tells which locker is which, not what’s inside the locker.

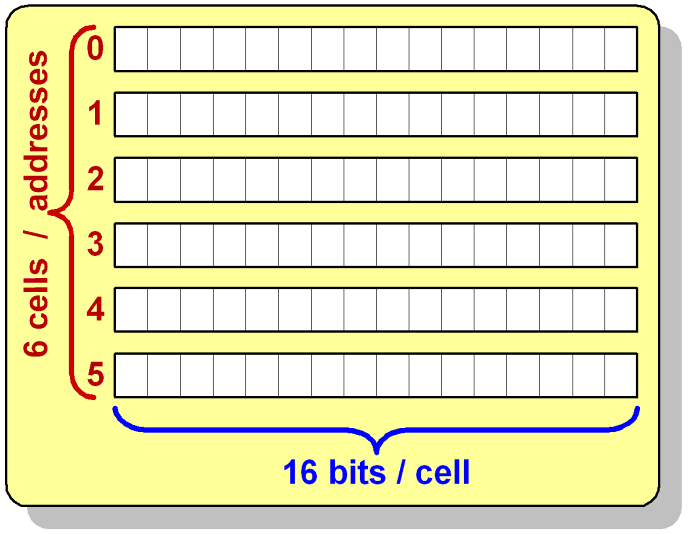
Whenever the CPU accesses memory, it passes the address of the information it’s interested in so that the memory will select the correct group of bits.

Memory Organization

The number of bits stored in each memory cell can vary from one computer to another, as can the number of memory cells. Here are some examples of different ways to organize a memory which holds 96 bits:



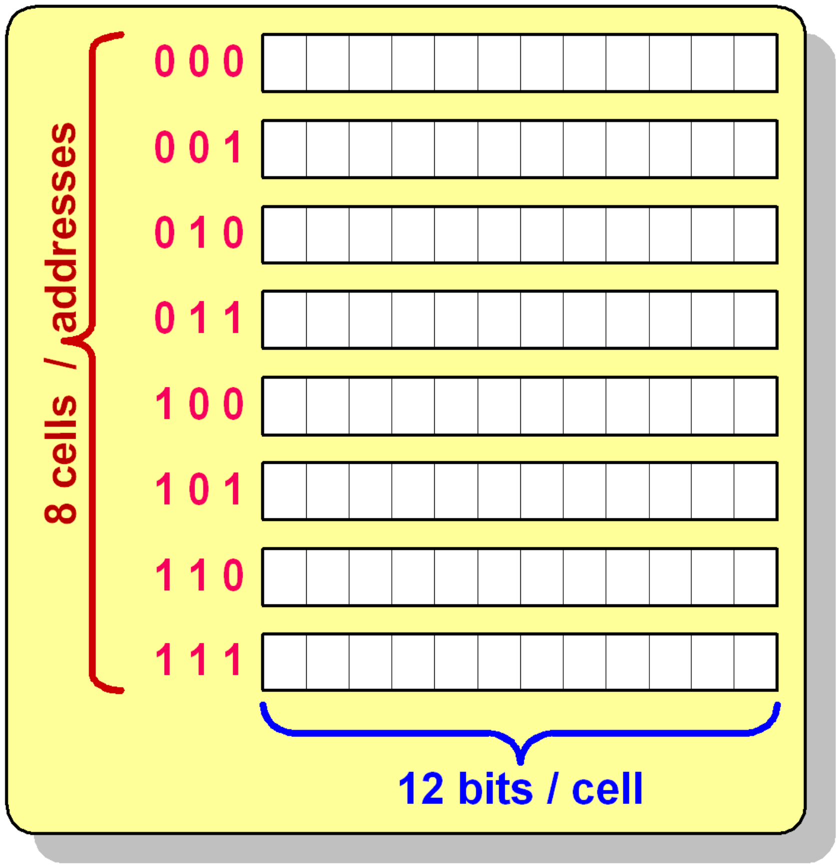




Memory organization is written as number of cells X cell size.
For example the memories above are organized as (left to right) 12 x 8, 8 x 12 and 6 x 16.

Memory Address Sizes

Memory addresses, like everything else in a computer system, are sent between the CPU and memory using binary signals. So memory addresses are actually binary numbers. For example, the "8 x 12" memory we showed on the previous page actually uses binary addresses like these:



This example shows that a memory with 8 cells can use a 3-bit memory address, because three bits can have 8 different combinations ($2^3 = 8$).

Address Sizes vs. Cell Sizes

The number of bits in each memory cell and the number bits in the memory address are completely independent of each other. The computer designer trades off various advantages and disadvantages when selecting the sizes of each cell and the number of addresses:

	Advantages of a large size	Advantages of a small size
Cell Size	• Larger memory capacity with same address size • Faster performance – each read or write transfers a larger number of bits	• Finer resolution – can read/write smaller units of data • No need to read/write extra data when only a small amount is needed
Address Size	• The memory contains more cells, and thus can store more information	• The CPU does not need to use as many bits when handling memory addresses

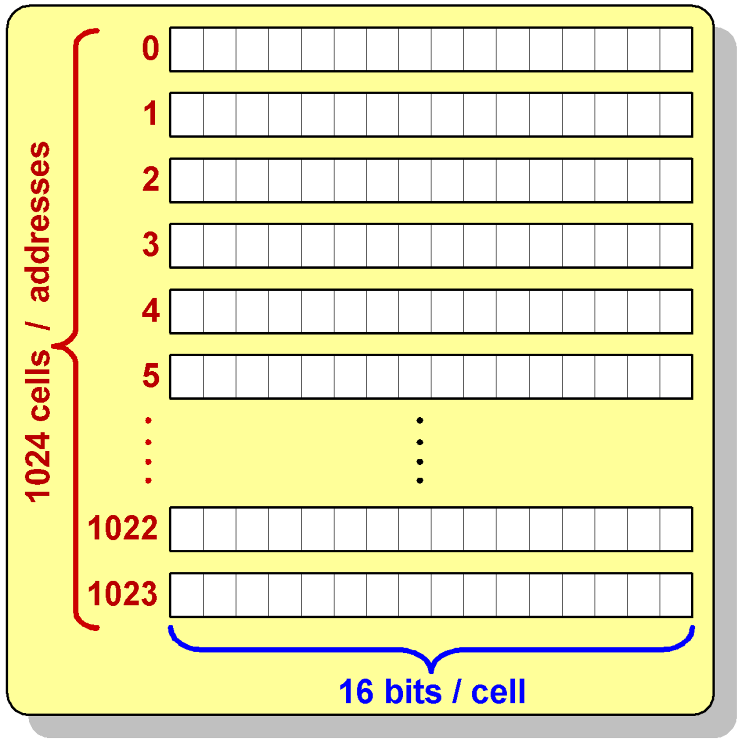
Most modern computers use a memory cell size of 8 bits – this unit of storage is called a byte.

Exercise 1 - Memory Organization

A memory holds 1024 cells with 16 bits stored in each cell:

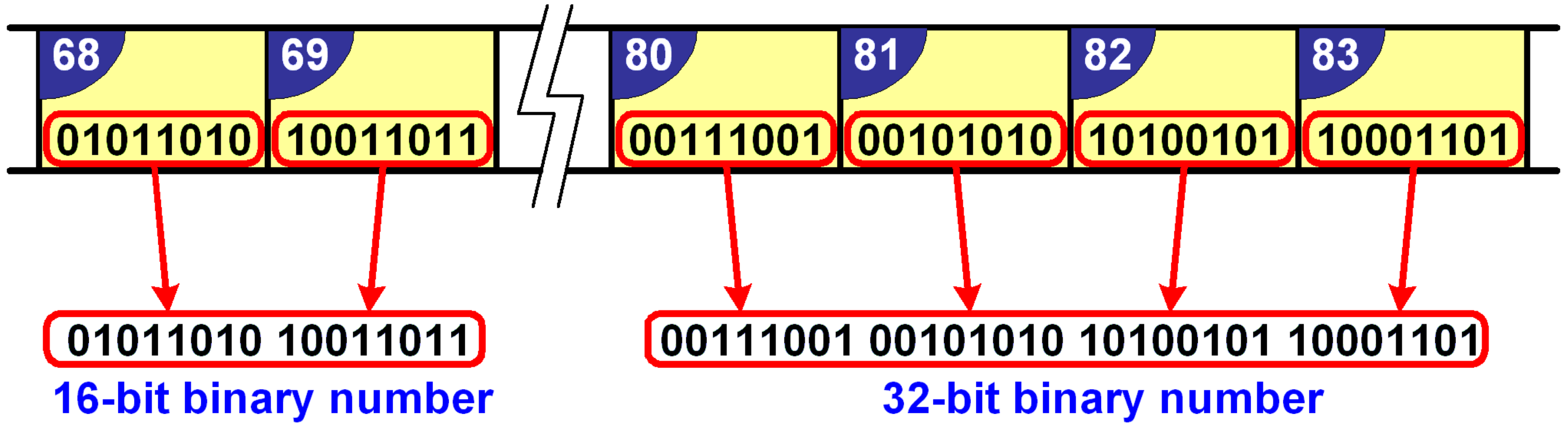
1. **How many bytes can the memory hold?**
   - **A – 1024 bytes**
   - **B – 2048 bytes**
   - **C – 16384 bytes**
   - **D – 32768 bytes**

2. **How many bits is needed for the memory address?**
   - **A – 8 bits**
   - **B – 10 bits**
   - **C – 16 bits**
   - **D – 1024 bits**



Storing Multibyte Numbers

CPUs can usually handle numbers with more bits than the memory cell size. It's possible store numbers larger than an individual memory cell by using two or more memory cells to hold the data. For example, a computer system that uses 8-bit memory cells (each memory address holds exactly one byte of data) can store 16-bit numbers using two consecutive memory cells, or 32-bit numbers using four consecutive memory cells:



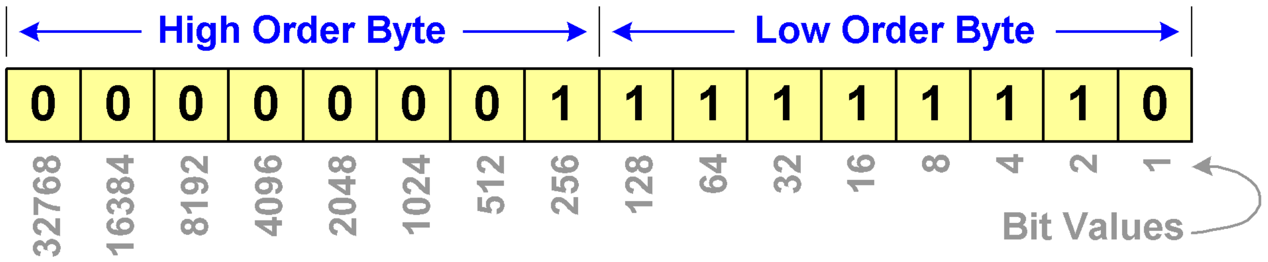
When the CPU wants to move a multi-cell number to or from memory, it indicates the address of the first cell along with the number of cells it wants to read or write. For the numbers in the above example:

	Address	No. of Cells
16-bit number	68	2
32-bit number	80	4

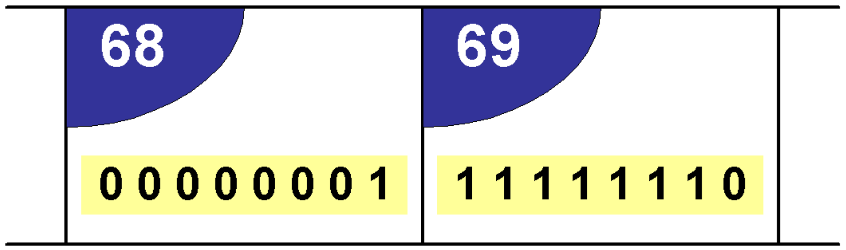
Byte Ordering

When a binary number is larger than the memory cell size, there is a choice as to which part of the binary number should be stored in which memory cells. There are two different ways to this. The methods are called Big Endian and Little Endian, after the fictional countries in "Gulliver's Travels".

The examples below show how each of these methods stores a 16-bit number with the decimal value 510 at memory address 68:

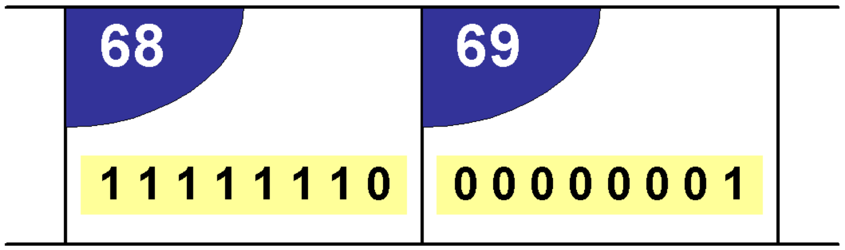


Big Endian



The high-order byte is stored first

Little Endian

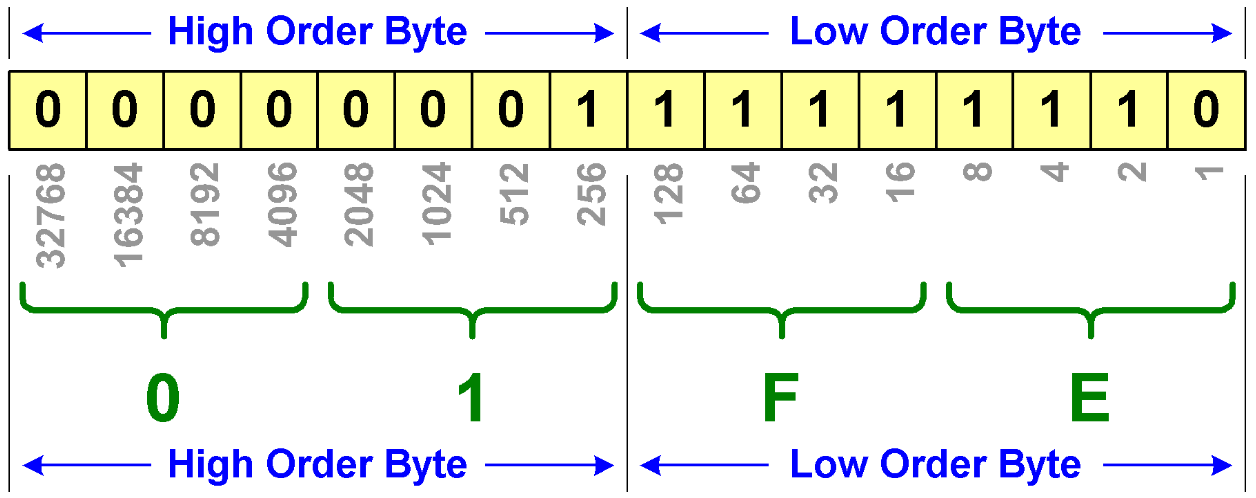


The low-order byte is stored first

Note that the order of the bits within each byte do not change – only the order of the bytes themselves.

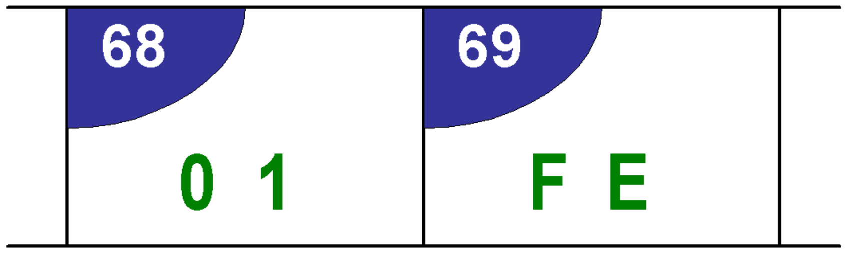
Byte Ordering (in Hex)

We can use hexadecimal notation to show exactly the same thing without having to deal with so many "1"s and "0"s. First, we convert the binary value for 510 into hexadecimal:



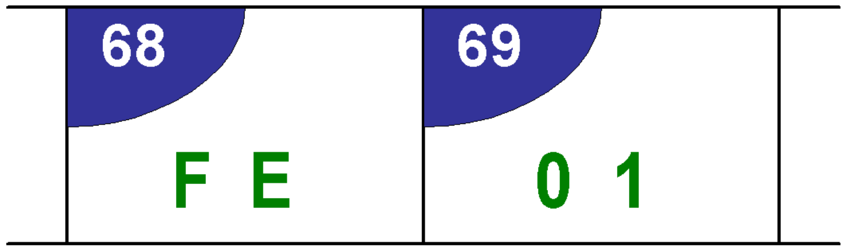
The hexadecimal value is "01FE", with "01" being the high-order byte and "FE" being the low order byte. As before, we can store this in memory in Big or Little Endian format:

Big Endian



The high-order byte is stored first

Little Endian



The low-order byte is stored first

Byte Ordering

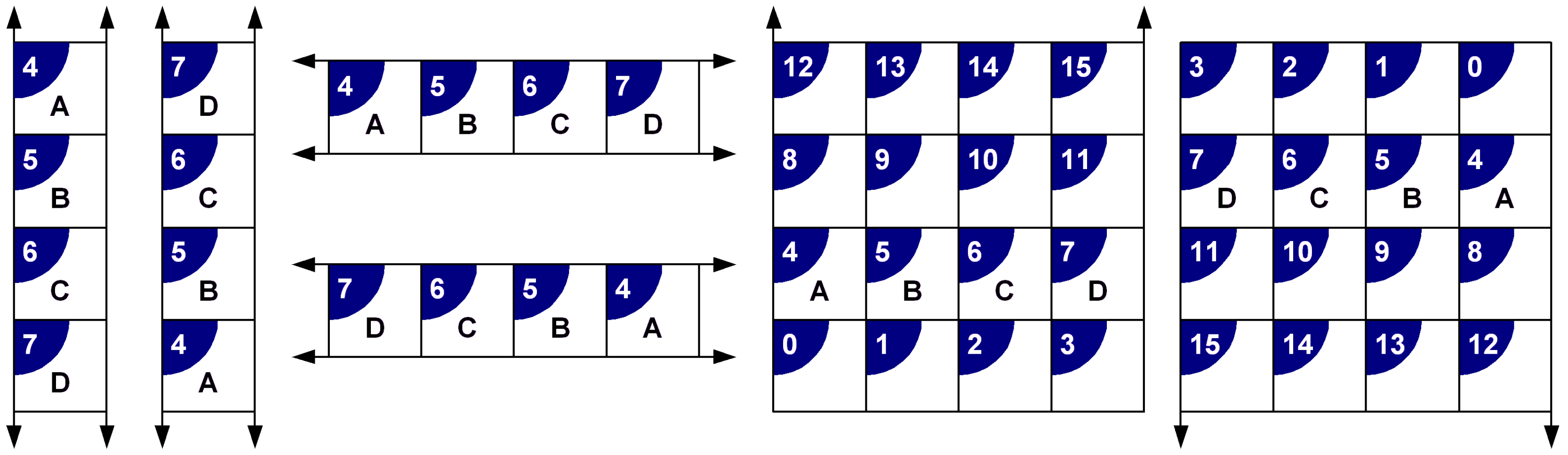
There's no particular advantage to either Big or Little Endian byte ordering. Many people find Big-Endian to be easier to read because it shows the high-order byte of a number on the left side when memory diagrams are written from left-to-right. On the other hand, each byte of a Little Endian binary number is more consistent with it's memory address (the part of the number stored at the higher address is more significant).

When multi-byte binary data is transferred from a Big-Endian system to a Little-Endian system, the order of bytes must be reversed so that the binary numbers are interpreted correctly.

Note that text or character data is always stored with the leftmost byte of the text string at the lowest-numbered memory address. So, data that is stored as text never has to be adjusted when it's moved from one computer system to another.

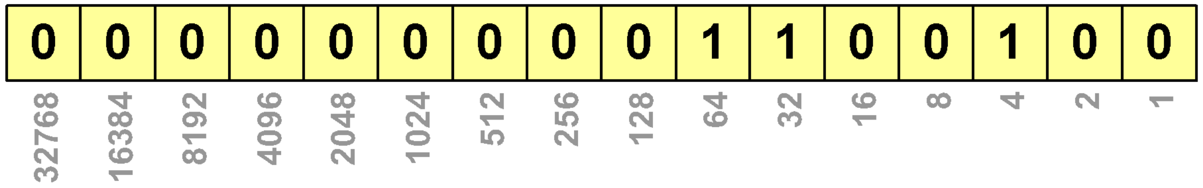
Note on memory diagrams:

A diagram showing the contents of memory can be drawn many different ways. All of the diagrams below show the same memory contents:

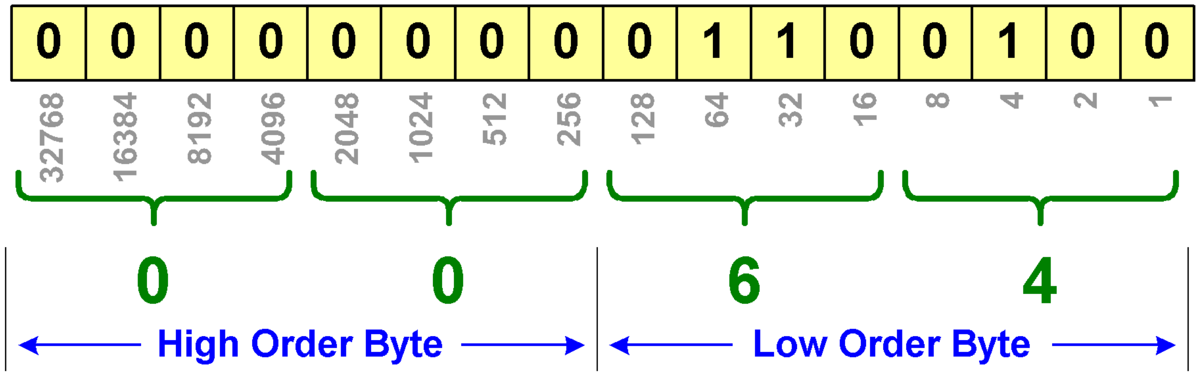


Exercise 2 - Byte Ordering

We're going to store a decimal 100 in memory as a 16-bit value. In binary, it's value is:



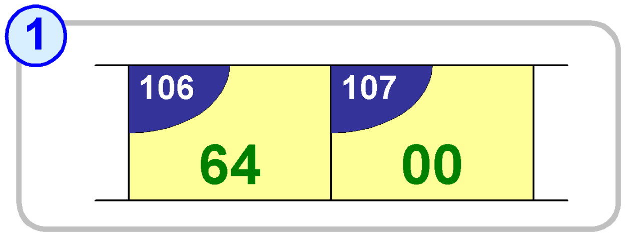
Converting to hexadecimal gives us:

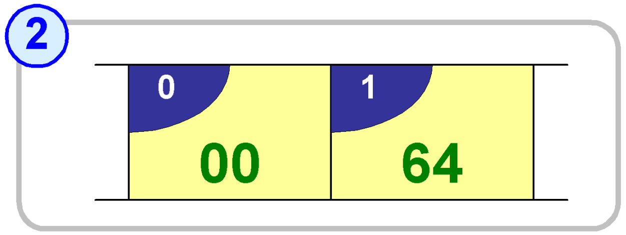


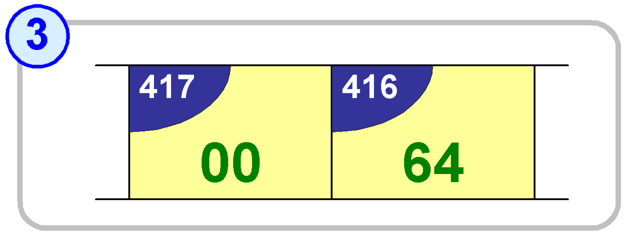
For each of these memory diagrams, click:

A for **Little-Endian**

B for **Big-Endian**



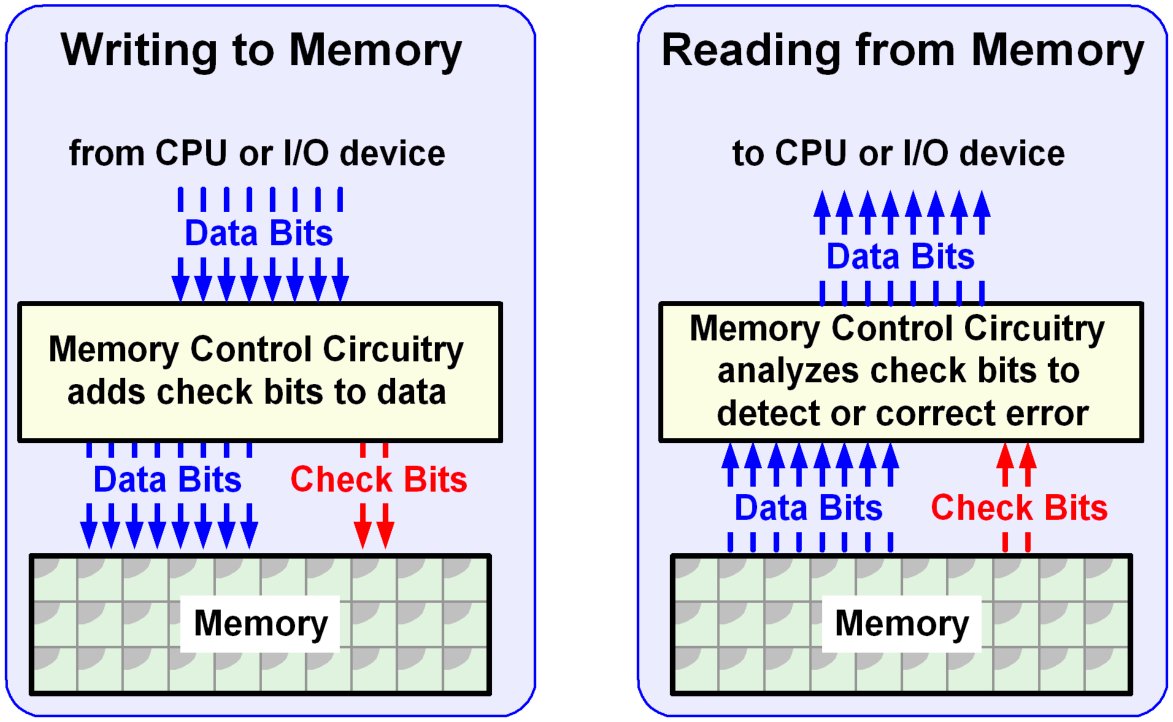




Error Correcting Codes

Error correcting codes are used to detect or correct data in memory when one or more of the bits change because of an error.

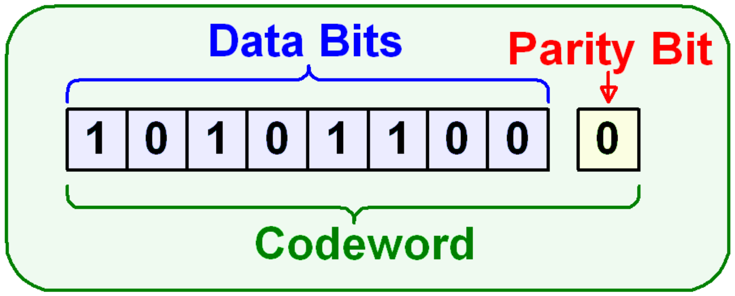
To detect or correct errors, extra bits are added to the data when it's written into memory. When the data is read back from memory the extra bits are then used to see if there is an error.

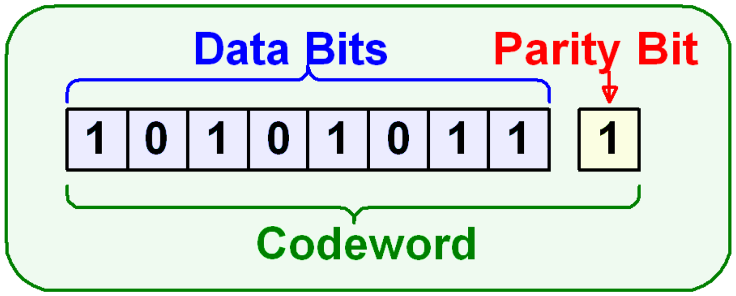


The combination of the data bits <u>and</u> check bits that is written to memory is called a "<u>codeword</u>".

Simple Parity

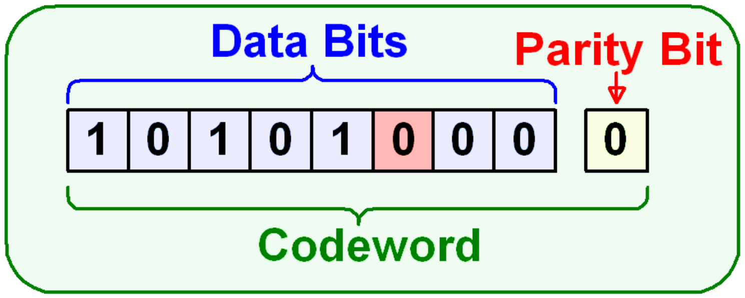
Parity checking is the simplest form of error detection. It uses one extra bit which is set so that the total number of "1" bits in the codeword are <u>even</u>. Here are two examples:

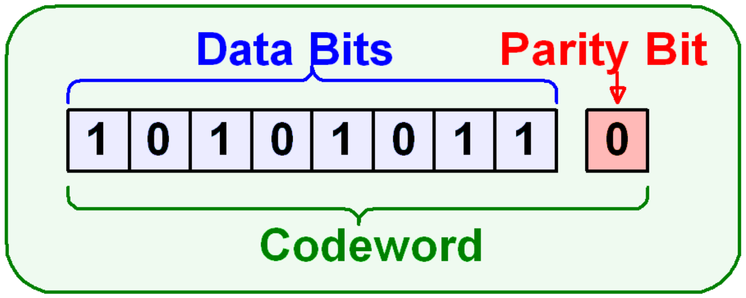




("Odd parity" is also possible, but in this course we'll use "even parity" for all our examples)

If a memory bit is accidentally changed, then the parity sum will no longer be valid. When the memory controller reads the data and parity back from memory and checks it, it will detect the problem and signal an error condition.





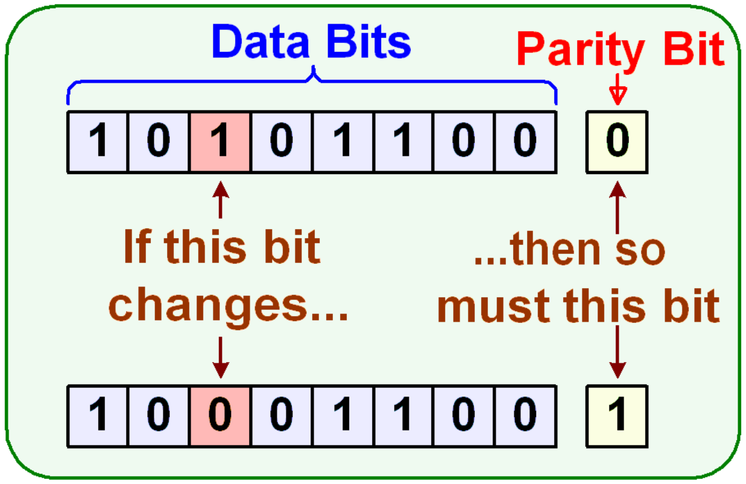
Simple parity can <u>detect</u> errors, but it cannot <u>correct</u> errors because there's no way for the memory controller to tell <u>which</u> bit is wrong. And parity cannot detect <u>multiple-bit</u> errors. For example, if exactly two bits are accidentally changed, then the codeword will still have the correct parity.

Hamming Distance

An error detection scheme like parity depends on being able to detect errors in the codewords stored in memory. The memory controller recognizes which codewords are valid and which are not according to the rules of the scheme. For simple even parity sums, the rule is that the total number of "1" bits in the codeword must be even.

The capability of the memory controller to be able to detect or correct errors depends on the Hamming Distance of valid codewords. The Hamming Distance is the minimum number of bits that are different between two valid codewords.

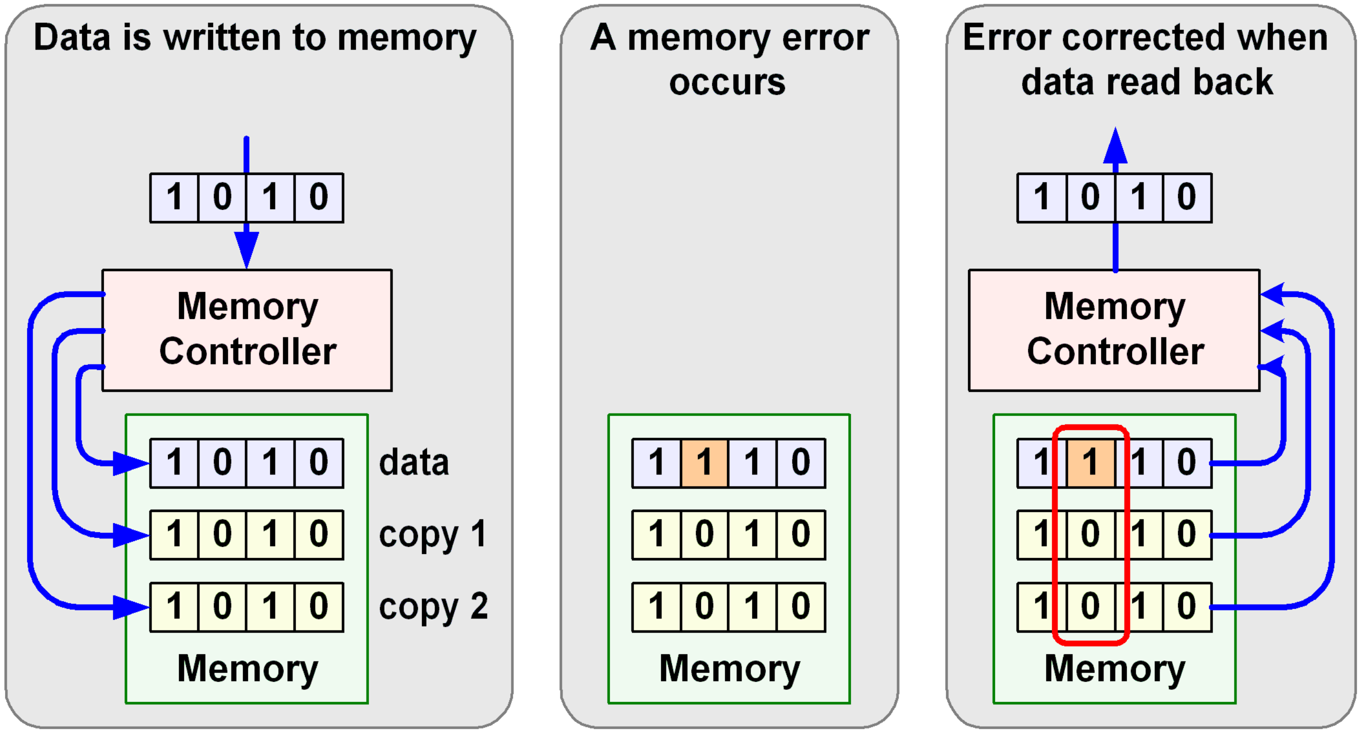
For example, simple parity has a Hamming Distance of 2, because every time a data bit changes, the parity bit must also change in order for the entire codeword to remain valid:



This type of simple parity code can detect single-bit errors, but it cannot correct them.

Correcting Errors

A code which can detect and correct errors has to have a Hamming Distance greater than 2. The simplest way of doing this is to simply store two extra copies of each data bit. This example shows how this can be done for four data bits:

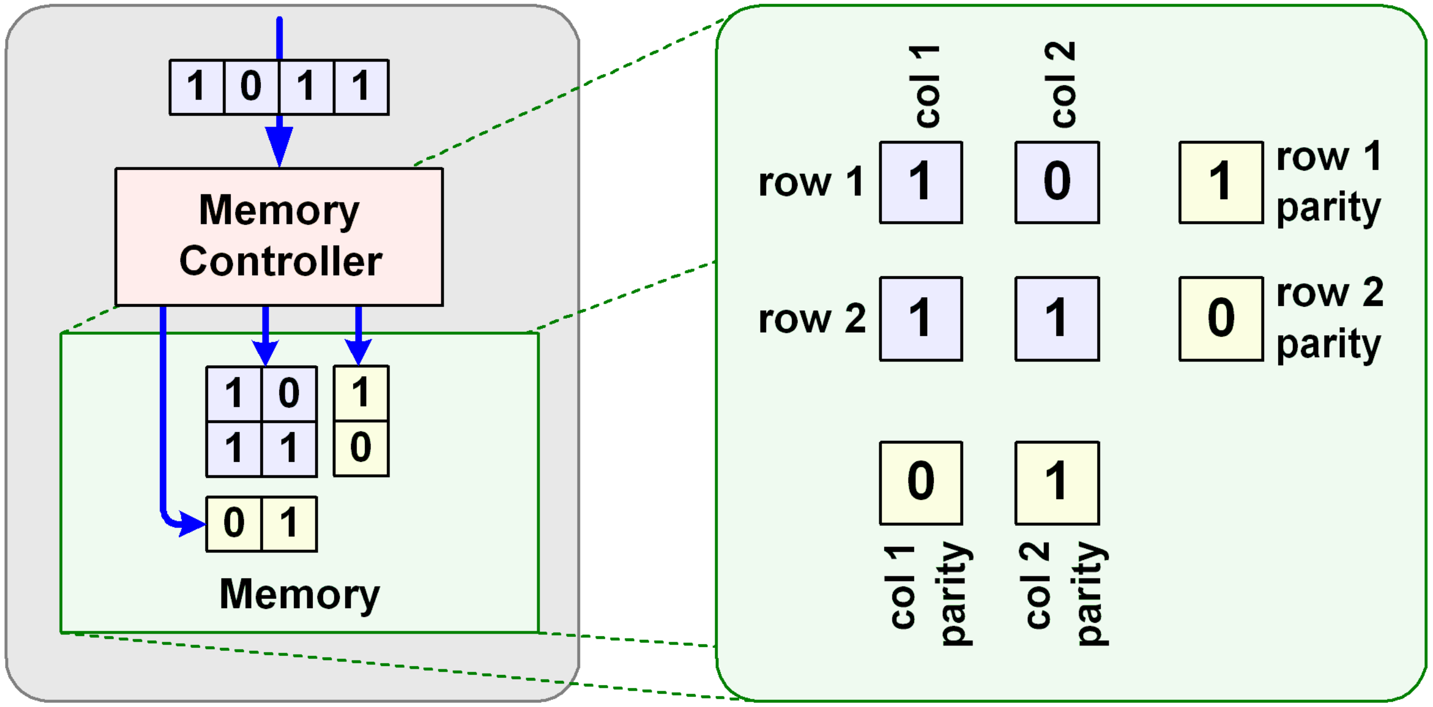


The memory controller can use a "best two out of three" rule when reading the data back from memory. If one of the three bits is different than the others, it is assumed to be the one in error and the memory controller returns the correct value.

This type of scheme can correct errors because it has a Hamming Distance of 3 (because if a data bit changes then the two other copies of the bit must also change).

Row / Column Parity

A more efficient way to create a codeword with a Hamming Distance of 3 is to arrange the bits in rows and columns and use parity sums for each row and column:

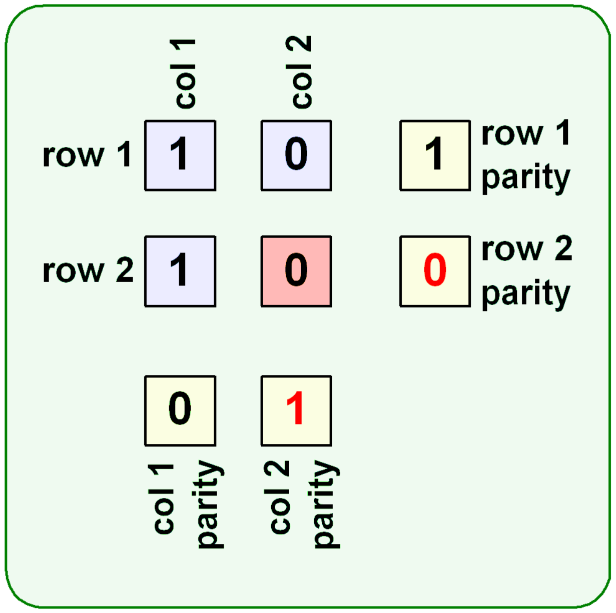


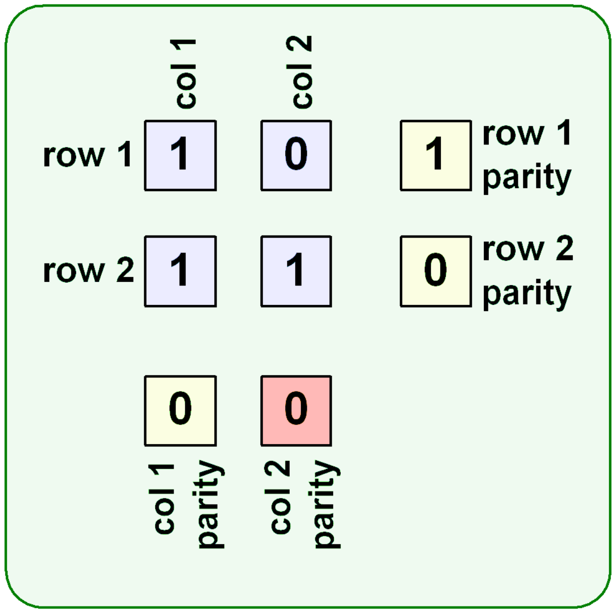
Row / column parity also has a Hamming Distance of 3. For example, if the upper-left data bit changes, then the row 1 parity and column 1 parity bits must also change. So three bits must be different between any two valid codewords stored in memory.

Since the Hamming Distance is three, this type of code can also detect and correct memory errors. But it requires much less memory than keeping two complete replicated copies of all the bits.

Correcting Errors using Row / Column Parity

Here are a couple of examples of how errors can be corrected using Row / Column Parity:





The bit in Row 2 / Col 2 is bad

The parity sums for Row 2 and Col 2 are both bad. The only bit that could cause both these sums to be bad is the Row 2 / Col 2 bit.

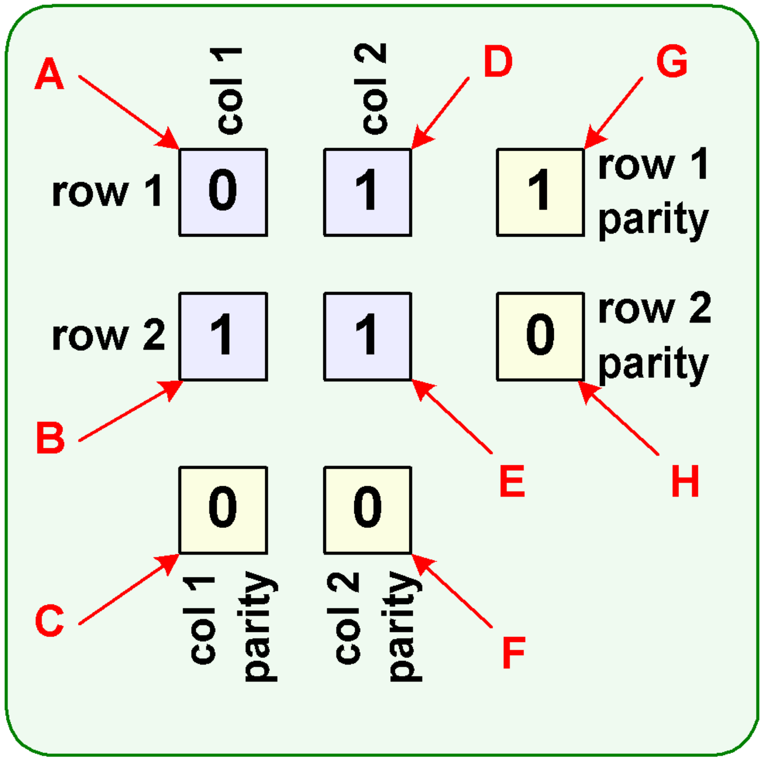
The Col 2 parity bit is bad

All parity sums are valid except Col 2. If a data bit was bad then there would be another parity sum that was also bad. Since there isn't the bad bit must be the Col2 parity bit.

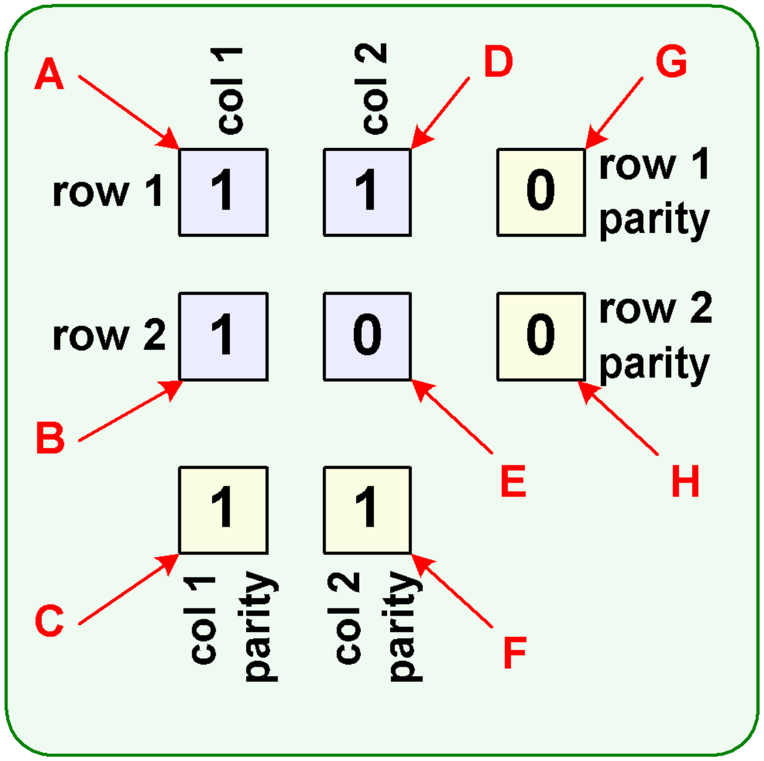
Exercise 3 - Row / Column Parity

Click the letter for the bit that's in error for each of these configurations:

1.



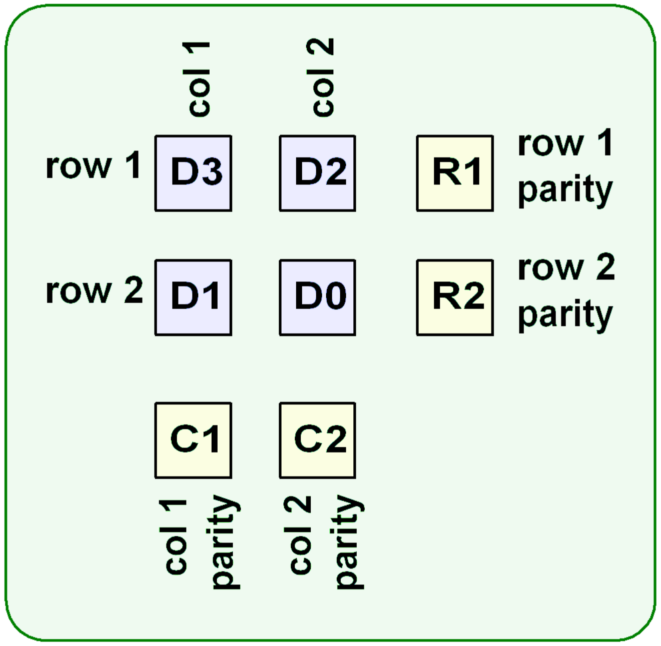
2.

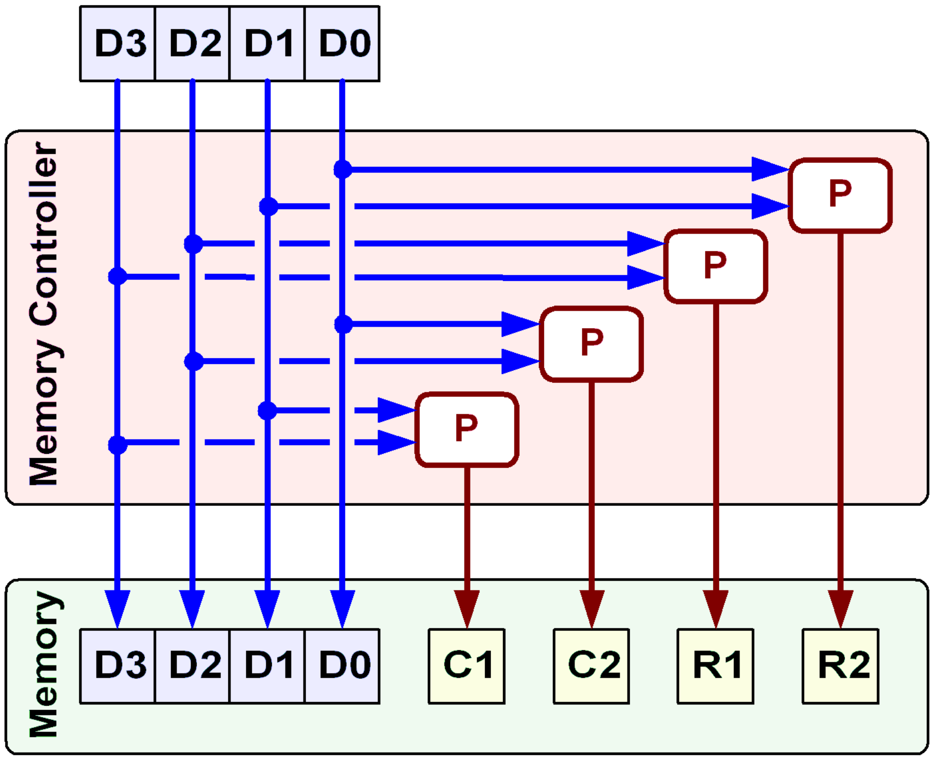


A Note on Row / Column Parity Memory Layout

The diagrams of Row / Column parity have been arranged to make it clear which data bits go with which parity sums – but they don't show the physical layout of the data in memory.

As data is written to memory, the memory controller creates four parity sums from pairs of bits that fall logically into each row and column:

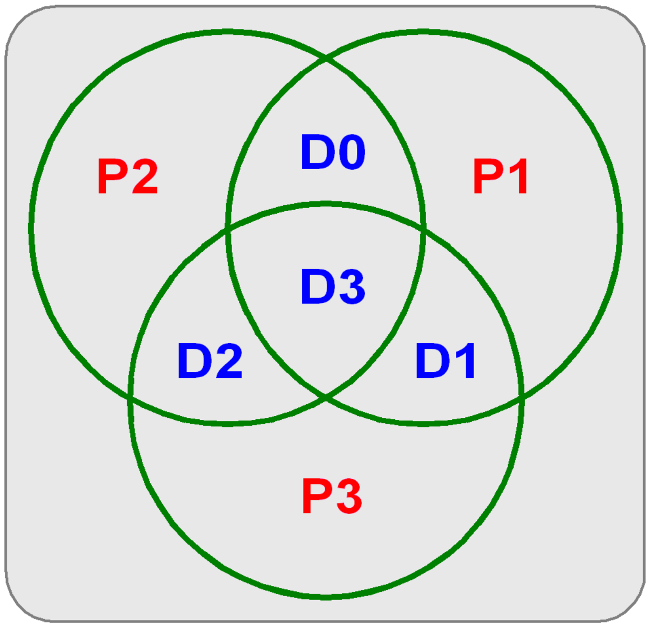


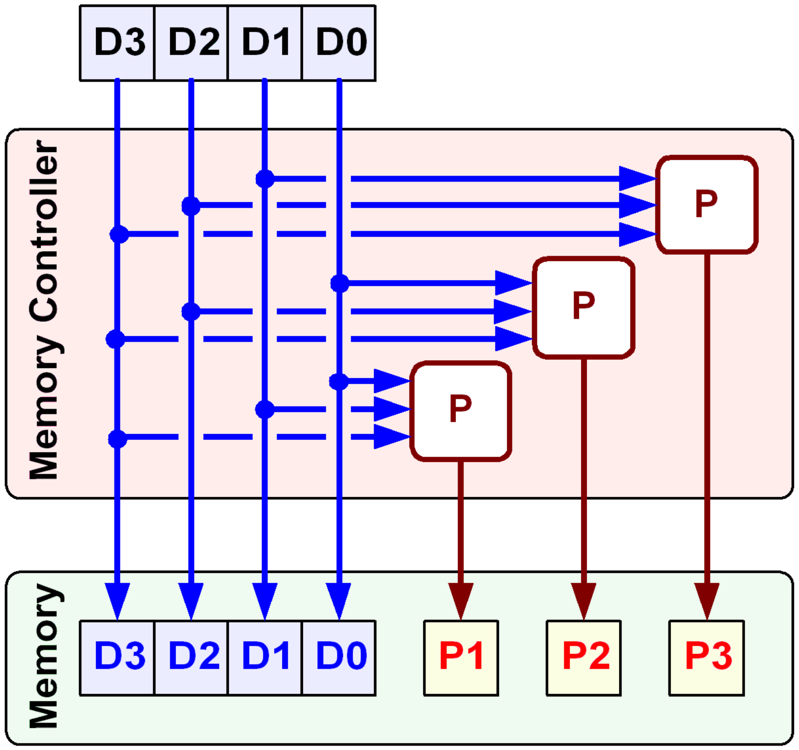


The important thing about this scheme isn't the Row / Column format, but that the parity sums are carefully chosen so that the Hamming Distance is 3 (i.e., if any data bit changes then two parity bits must also change in order for the codeword to remain valid)

Hamming Code for 4 data bits

There's an even more clever way to arrange the parity sums using a method developed by Richard Hamming in 1950. With this method we can protect our 4-bit data using a codeword that contains only three parity bits. This is easiest to visualize using a Venn diagram as shown at left below:



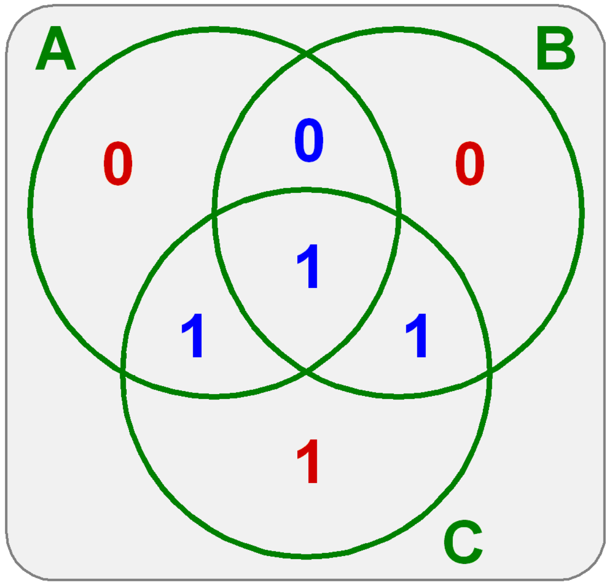


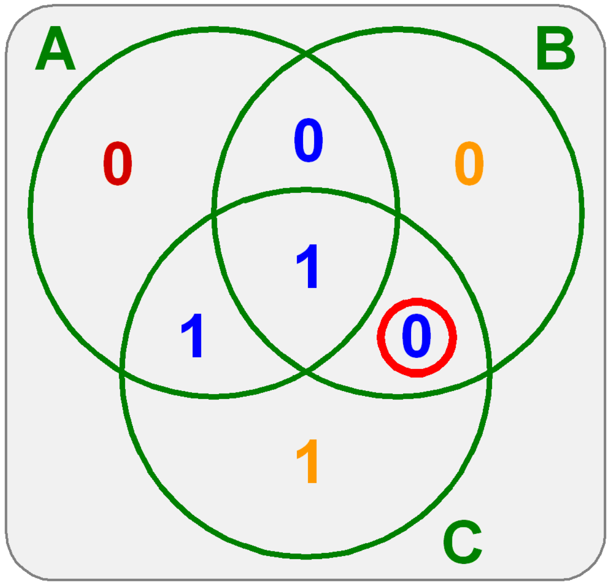
The four blue data bits appear in the intersections of the circles, and the red parity bits appear at the outside. The parity bits are set so that there's an even number of "1" bits in each circle.

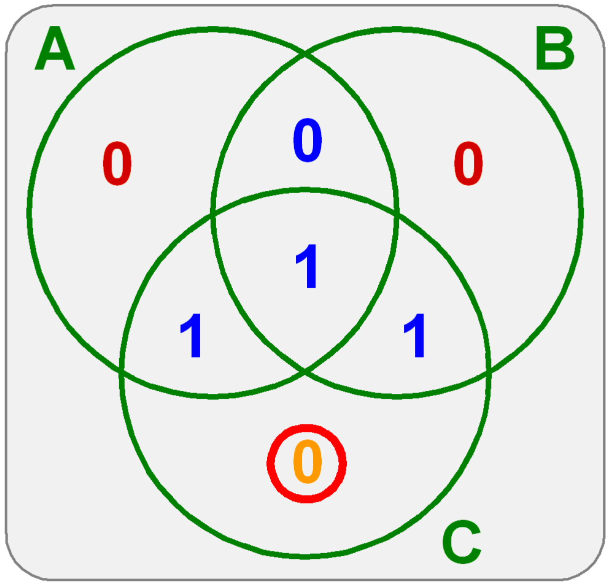
As before, the Venn diagram is only a handy way to visualize the parity sums. Actually, the memory controller stores all the bits in memory as a codeword as shown above at right.

Errors in a Hamming Code for four data bits

Here are some examples of correct and invalid combinations of data and parity bits in a Hamming Code for four data bits:







Correct

Circle "A" has two "on" bits – an even number.

Circle "B" has two "on" bits

Circle "C" has four "on" bits

Bad Data Bit

"B" has one "on" bit and "C" has three – both are wrong.

Only the bit that is common to circles B and C (and not A) could have caused this error.

Bad Parity Bit

Circle "C" has three "on" bits, the others are OK.

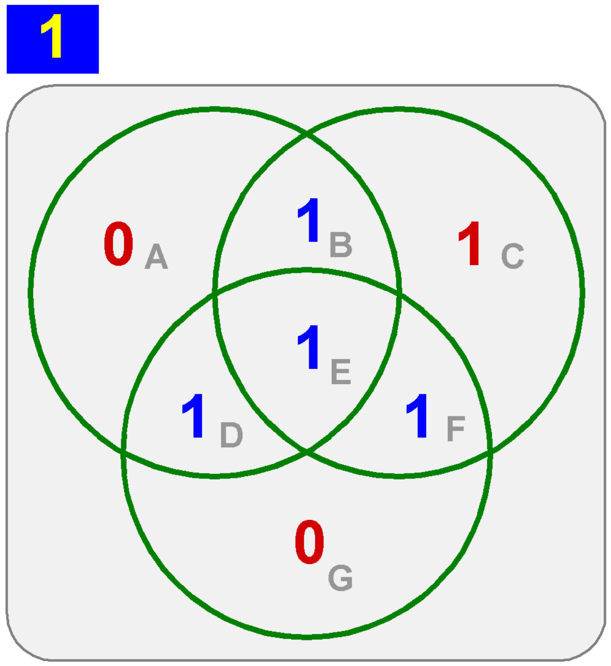
Only the parity bit in circle C could have caused this error.

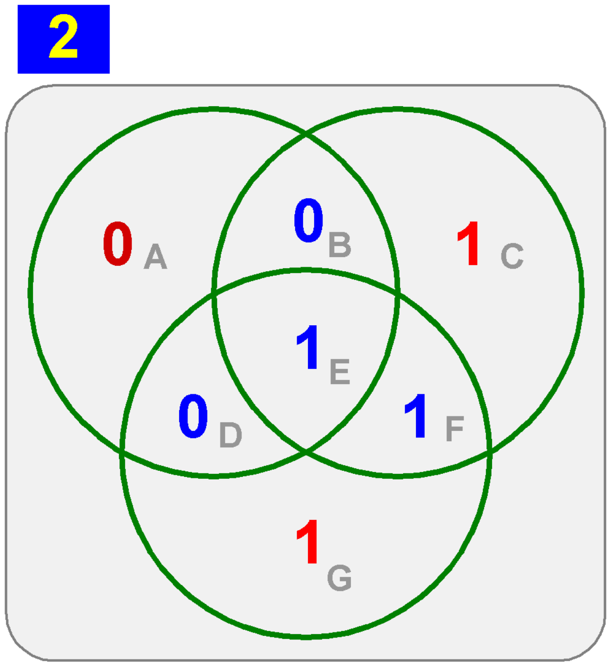
Note that this code also has a Hamming Distance of 3 – when any data bit changes at least two of the parity bits must also change.

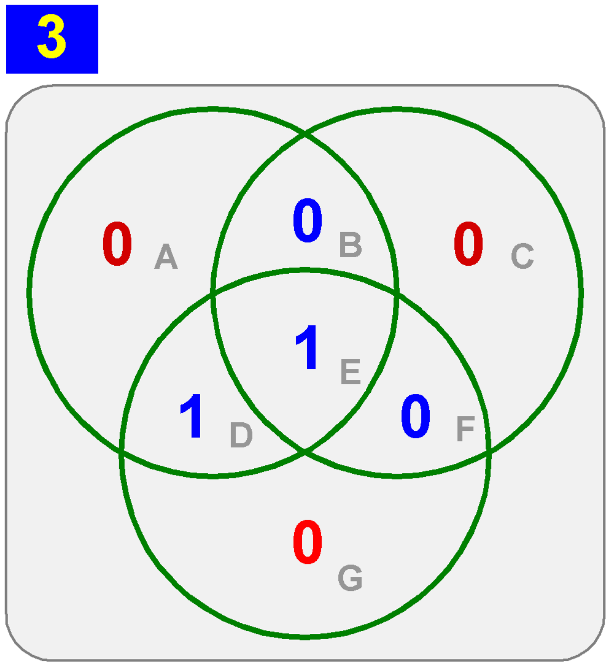
Exercise 4 – Correcting Errors in a 4-bit Hamming Code

Identify which bit (if any) is incorrect in each of the following 4-bit Hamming Code diagrams:

(use "H" if no bits are in error)







(use "H" if no bits are in error)

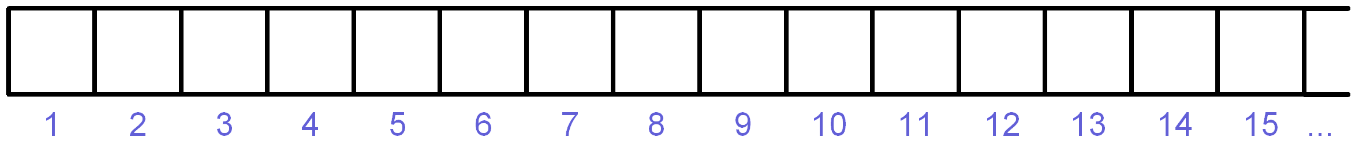
Hamming Codewords for Larger Data Sizes

Hamming Codes are widely used in ECC (Error Checking and Correcting) memory. A Hamming Codeword can be developed to detect and correct errors for data words of any size. The larger the data word, the more efficient the code is.

To create a Hamming Codeword, follow these steps:

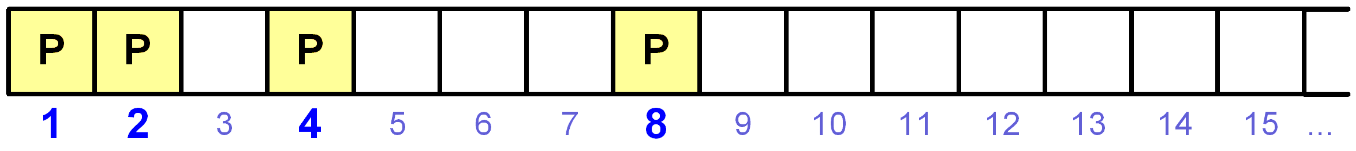
Step 1:

Draw a series of bits to hold the codeword. Number each bit, starting from the left with "1" (<u>don't</u> start with zero, and <u>don't</u> label starting from the right end)



Step 2:

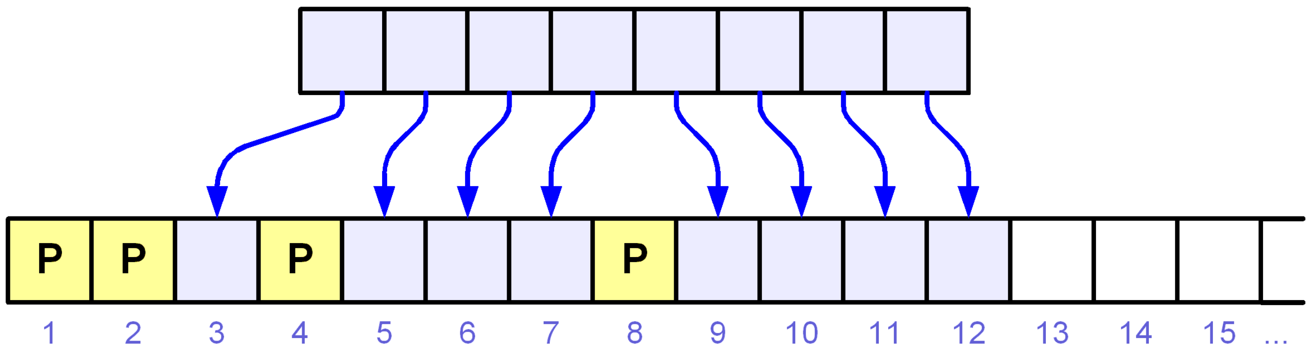
Label each bit whose number is a power of 2 as a "Parity" bit



Hamming Codewords for Larger Data Sizes

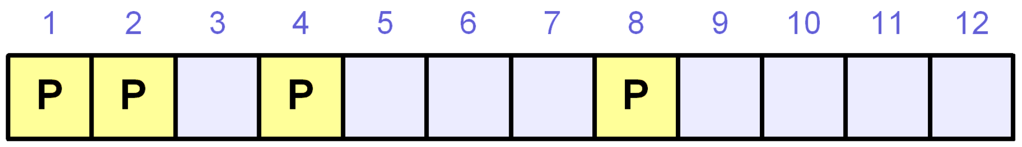
Step 3:

Insert the data bits into the non-parity positions in the codeword. The following example shows how 8 data bits would be inserted into the codeword:



Step 4:

Discard the unused bits:



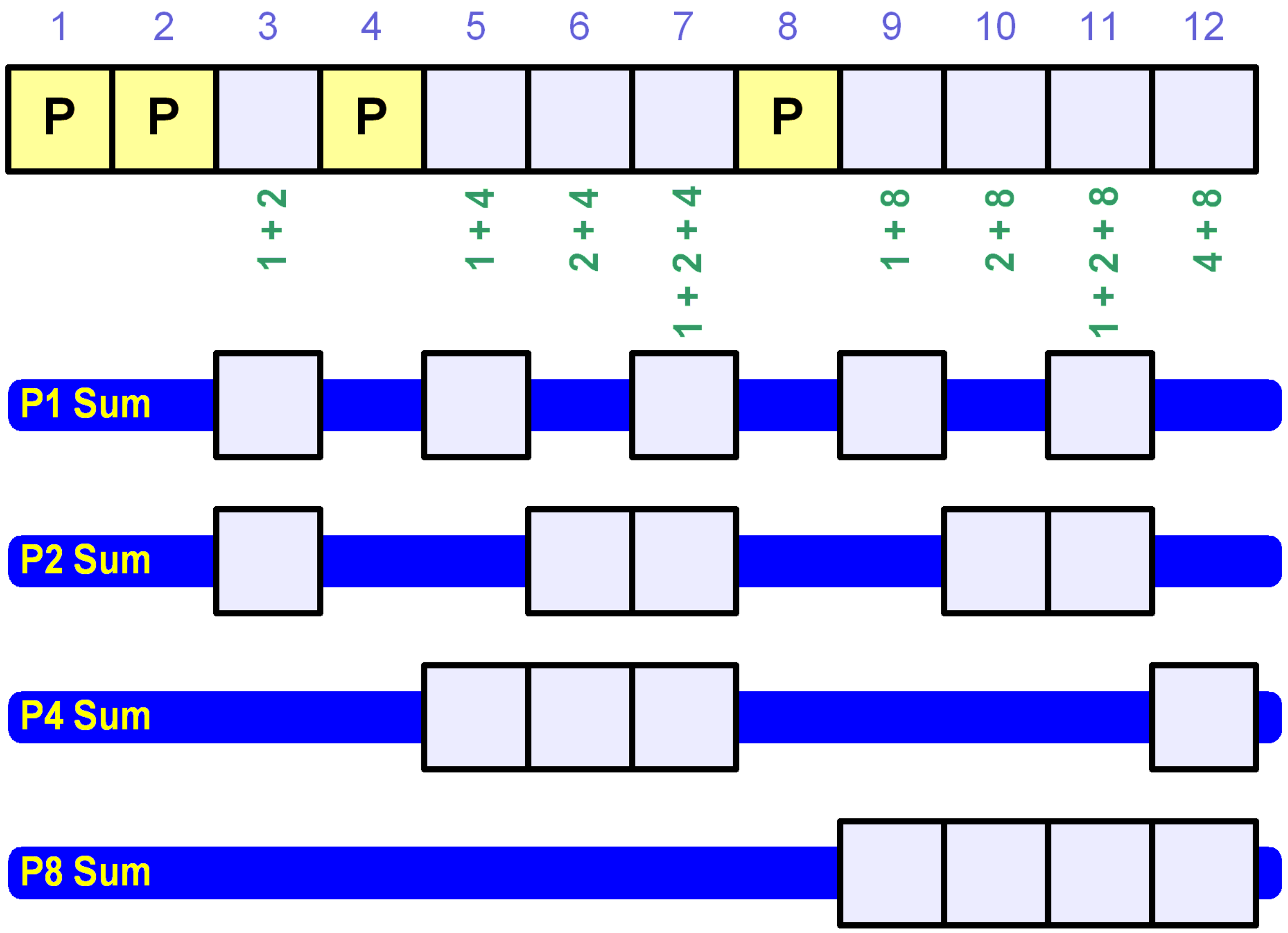
The Hamming Code to handle 4 data bits used 3 parity bits. Here we've been able to increase the "payload" to 8 data bits, but we only needed to add one more parity bit!

Hamming Codeword Parity Sums

The magic of the Hamming Code is in how the parity sums are calculated. Each data bit is included in two or more parity sums. Each data bit's bit number tells which sums it's part of. Write the bit number down as a sum of powers of two. For example:

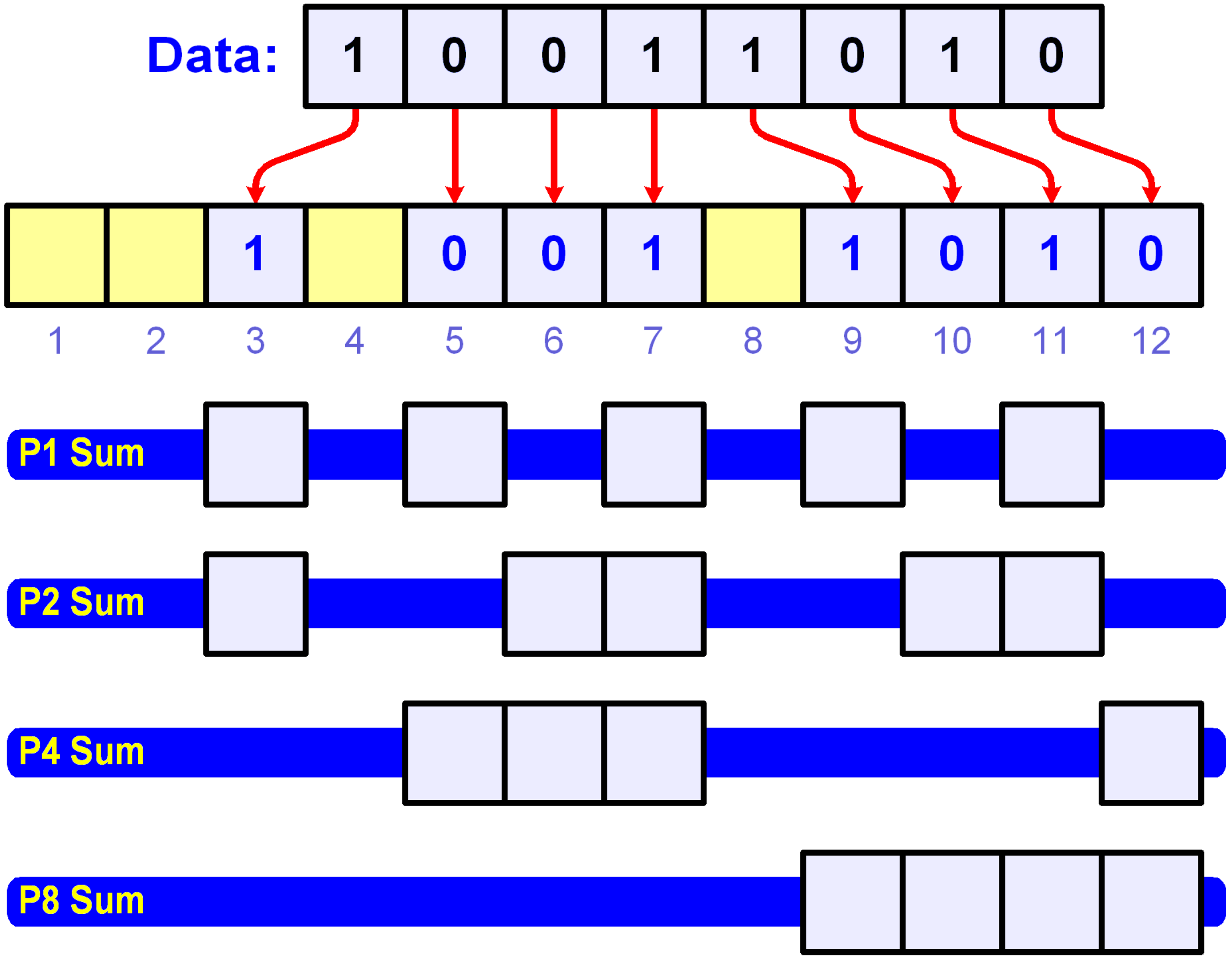
Bit number 11: 8 + 2 + 1 **Data bit 11 is included in the sums for parity bits 8, 2 and 1.**

This diagram shows the parity sums for a Hamming Codeword that holds 8 data bits:



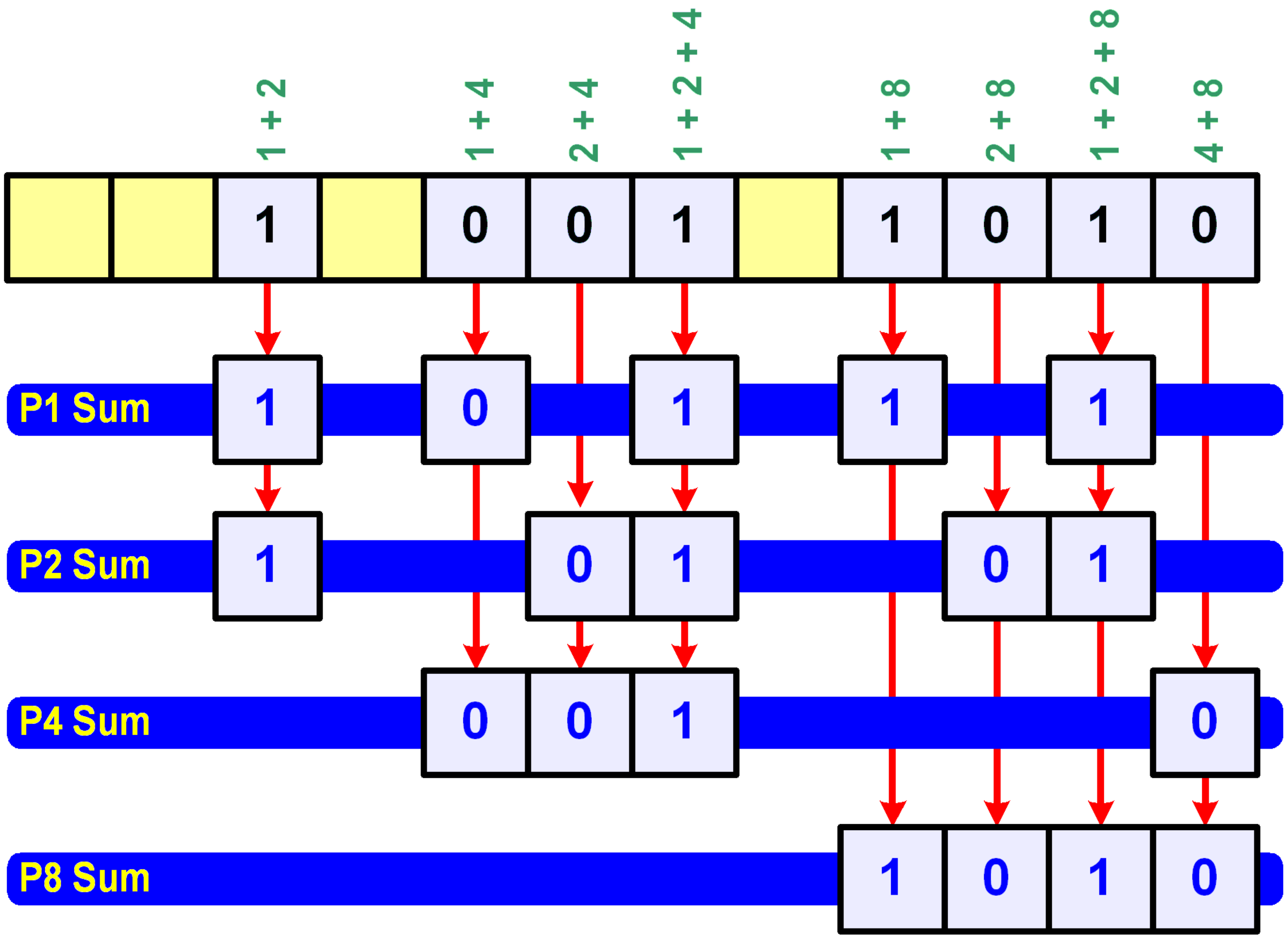
Building a Valid Hamming Codeword – Part 1

To build a valid Hamming Codeword from a given set of data bits, start by taking each data bit and put it into the corresponding data bit position in the codeword:



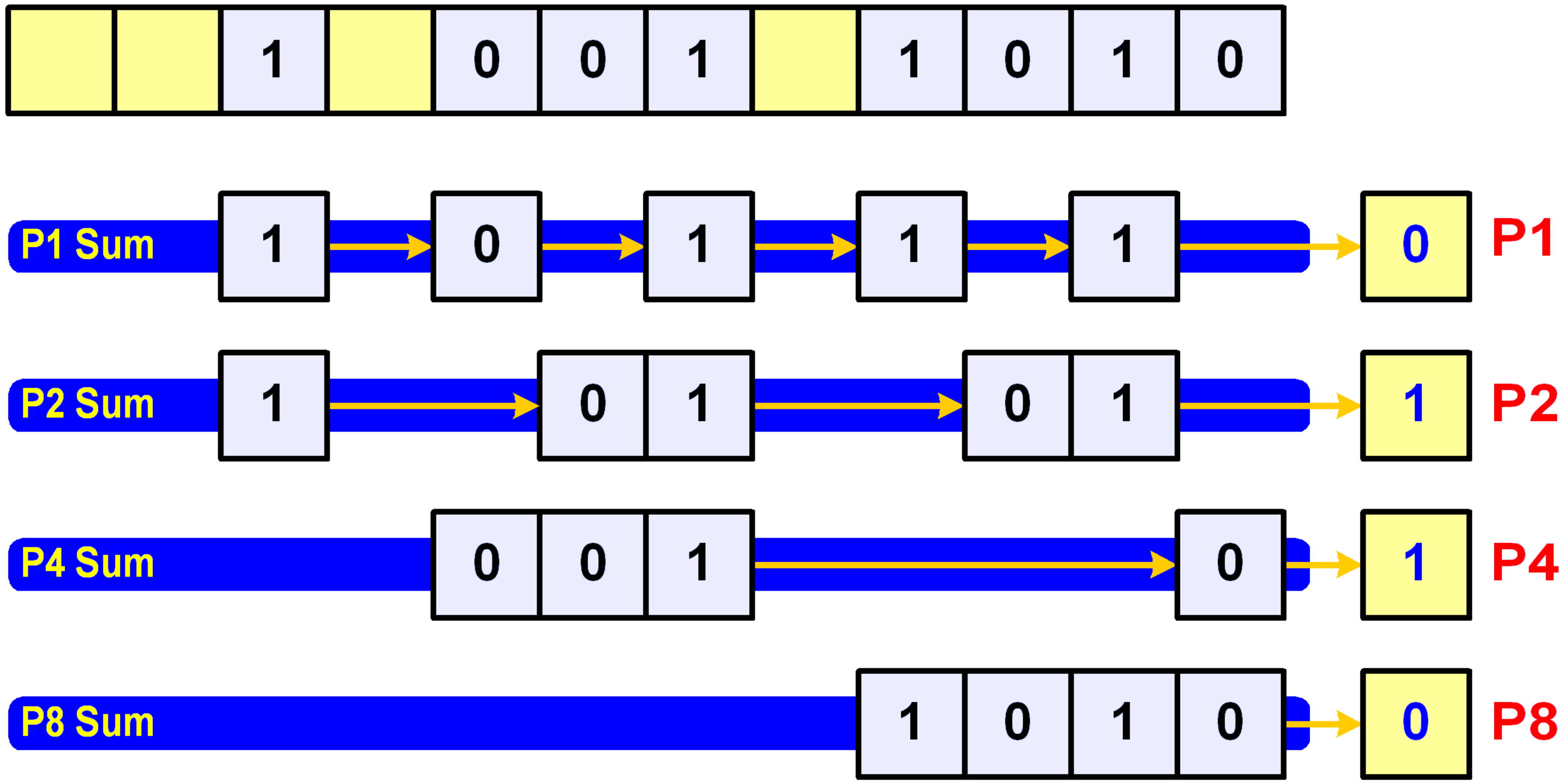
Building a Valid Hamming Codeword – Part 2

Next, put a copy of each data bit into the appropriate parity sums:



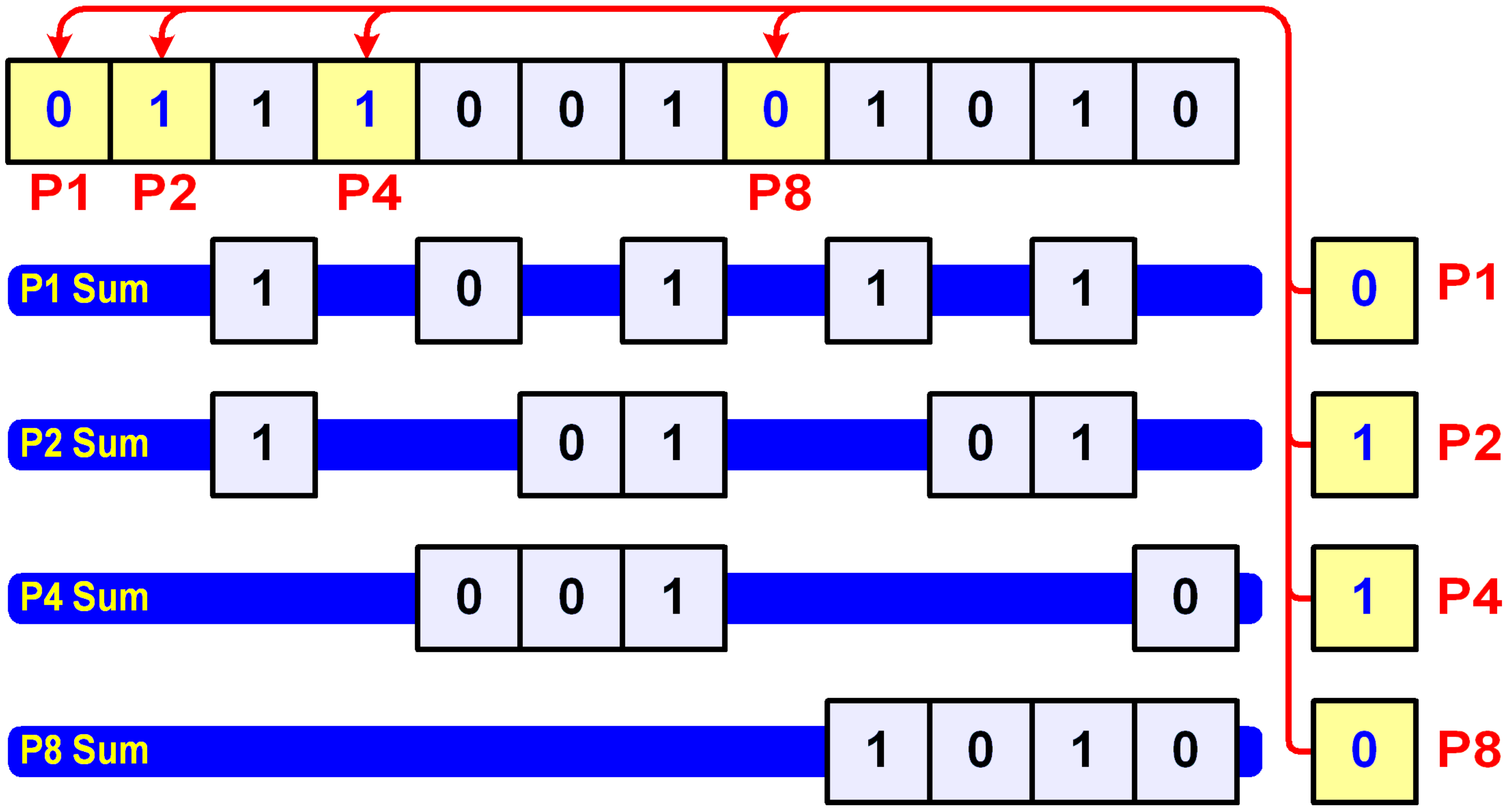
Building a Valid Hamming Codeword – Part 3

Then calculate what each parity bit should be (the total number of ON bits must be EVEN for each parity sum):



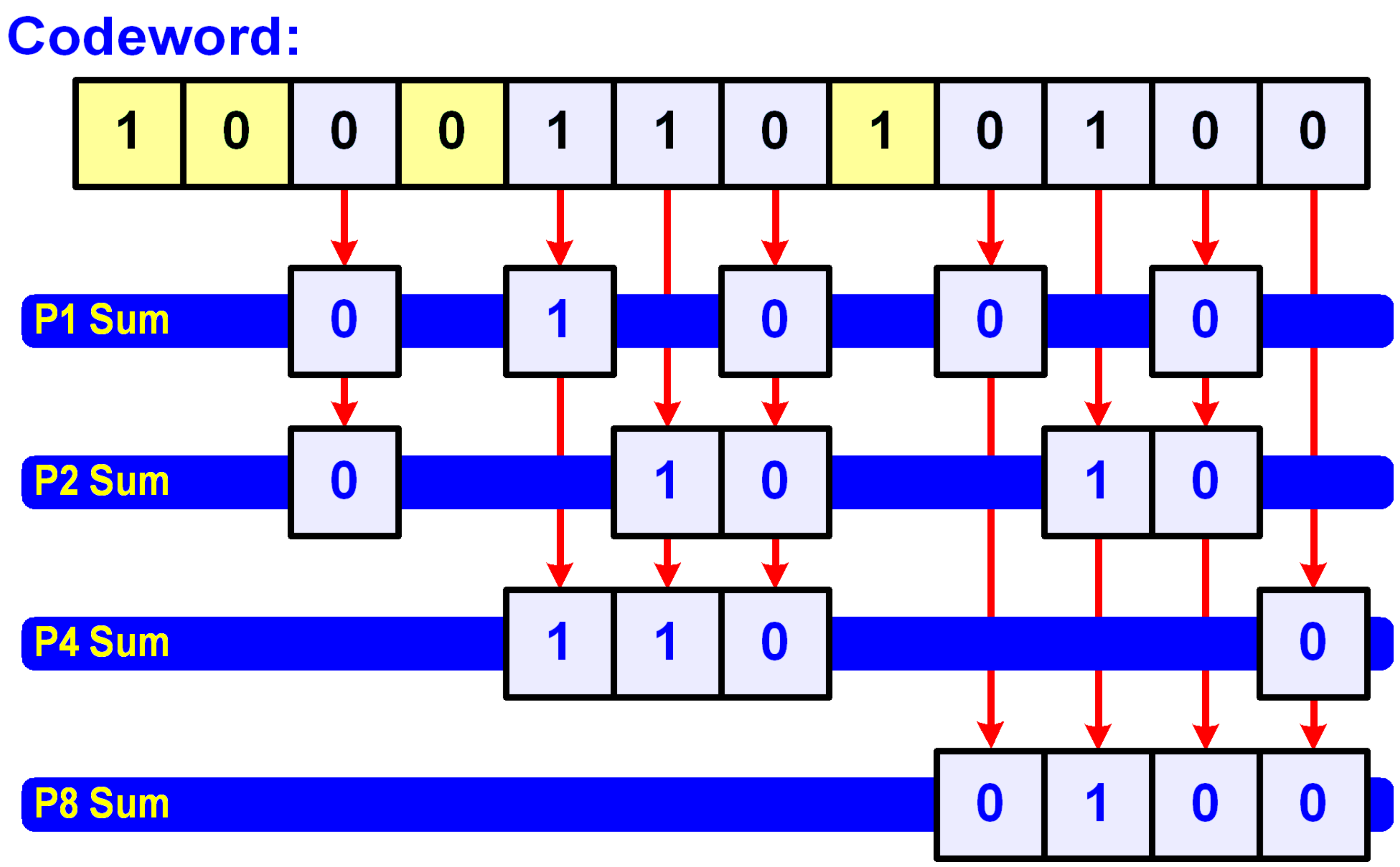
Building a Valid Hamming Codeword – Part 4

Finally, insert the resulting parity bits into the corresponding bits of the codeword:



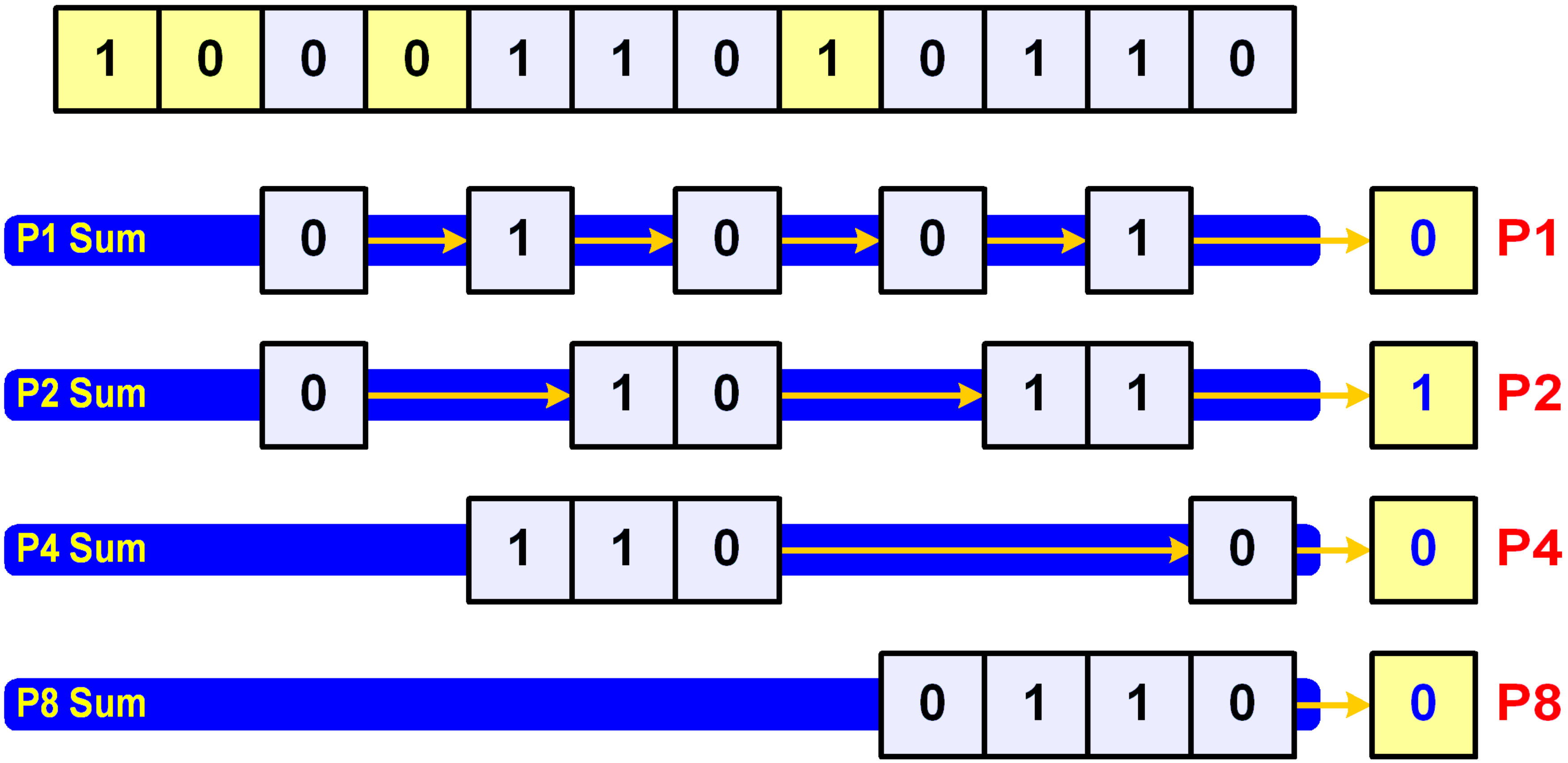
Checking a Hamming Codeword - 1

To check a Hamming Codeword (data bits plus parity bits) to see if it is valid or not, start by copying each of the data bits into the appropriate parity sums:



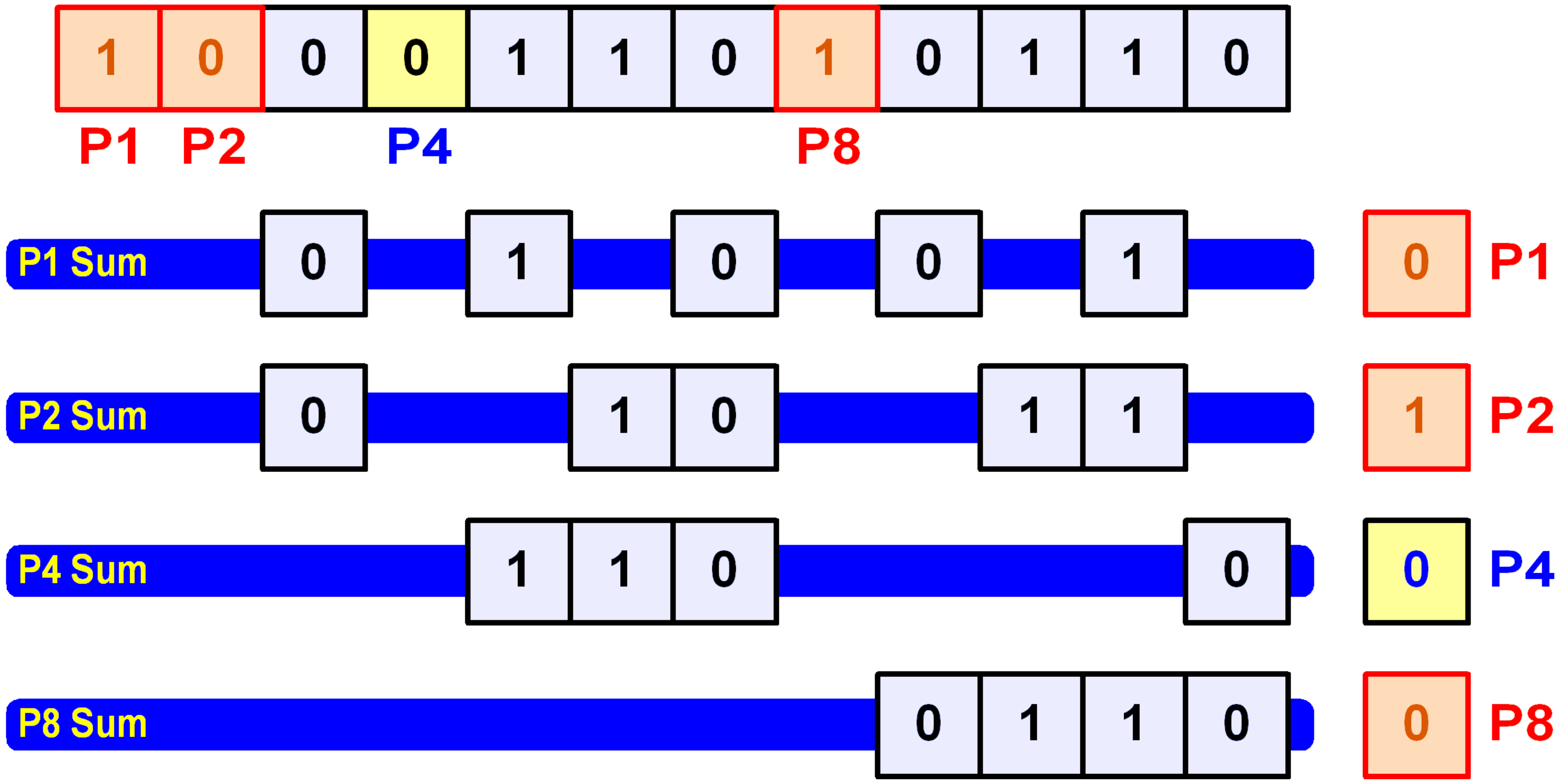
Checking a Hamming Codeword - 2

Then calculate what each parity bit should be (the total number of ON bits must be EVEN for each parity sum):



Checking a Hamming Codeword - 3

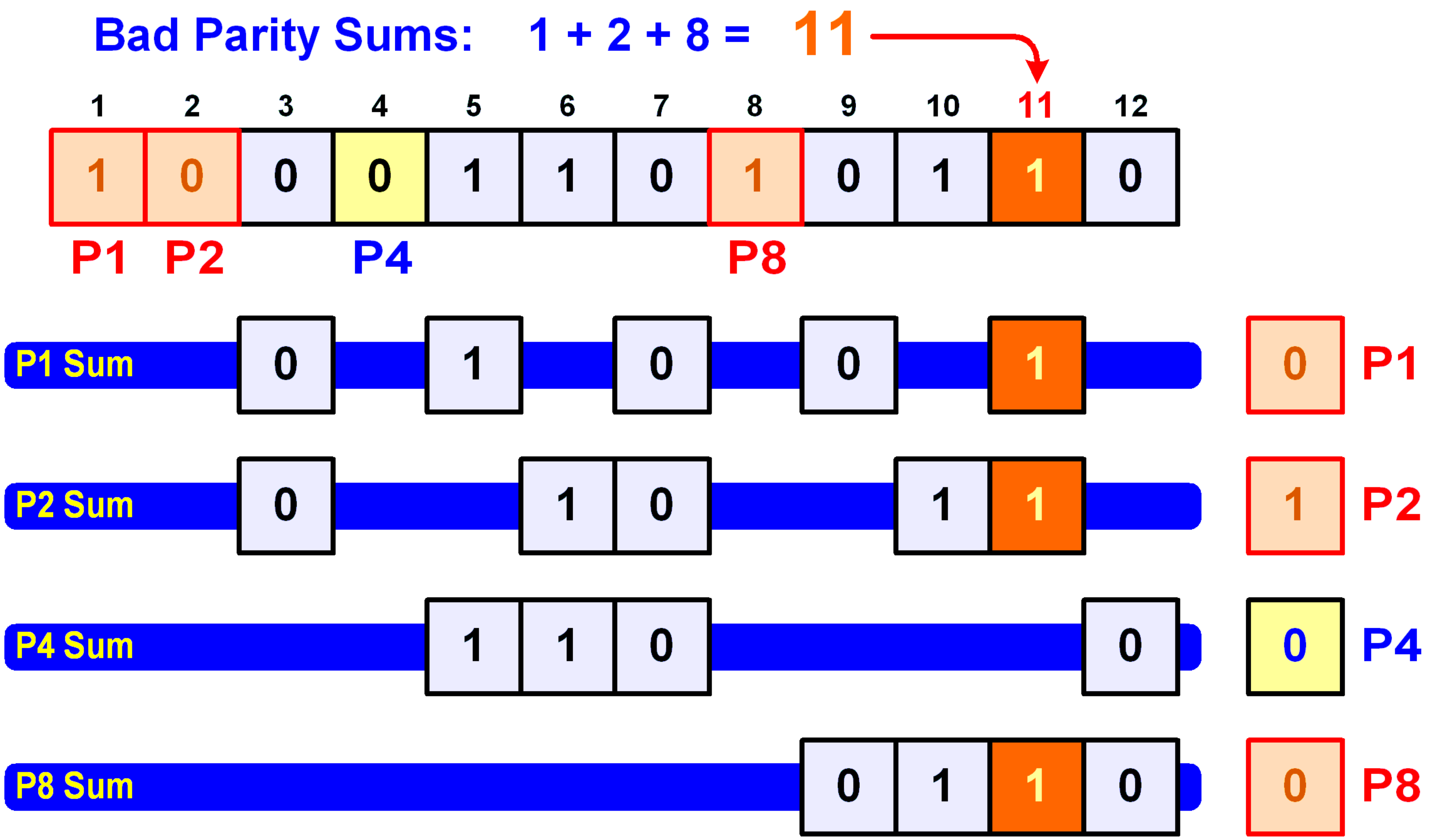
Compare each resulting parity bit with the corresponding bit in the original codeword to see if any of them are different:



A difference between what the parity bit <u>should</u> be and what the codeword <u>actually contains</u> means that the parity sum is invalid - one of the bits in that parity sum must be bad.

Checking a Hamming Codeword - 4

Add up the bit numbers of the invalid parity sums – the result tells you which bit is bad:



Exercise 5 – Layout of a Hamming Codeword for 16 Data Bits

Which of the following is the correct layout for a Hamming Codeword which can hold 16 data bits?

A

P1	P2		P4				P8								P16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

B

P1	P2		P4				P8								P16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

C

P0	P1		P2				P3								P4					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

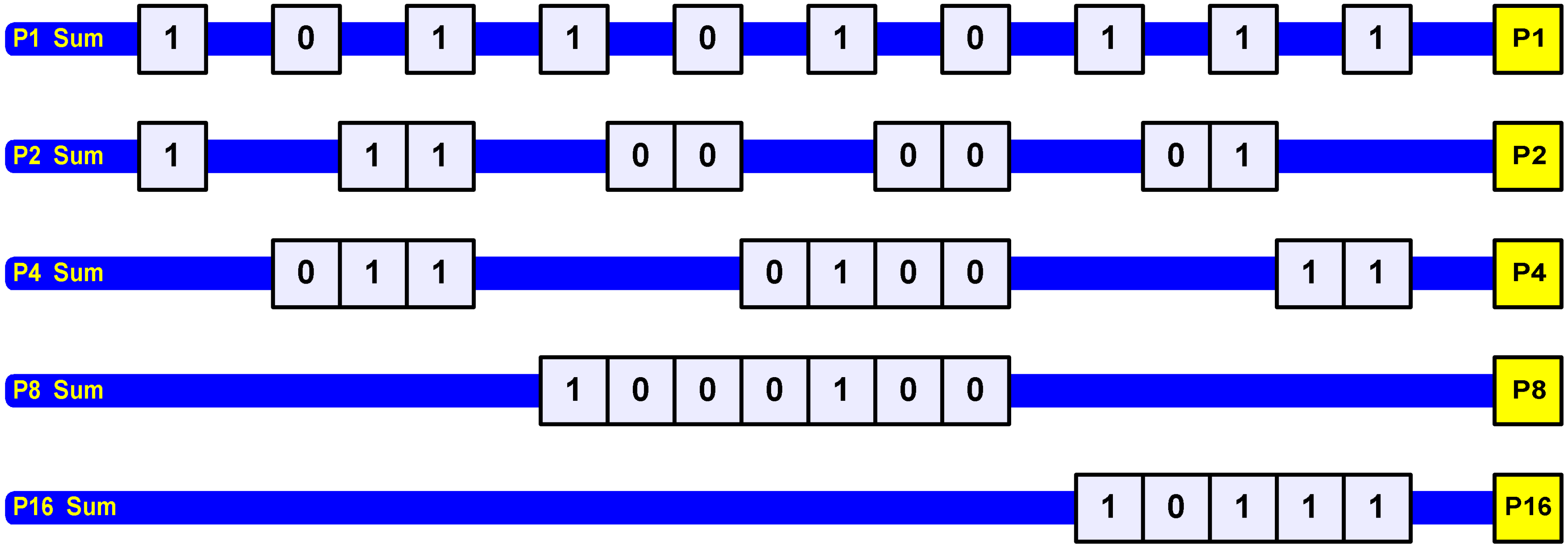
D

P1	P2		P4				P8								P16					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Exercise 6 - Hamming Codeword Validation

A Hamming Codeword holding 16 data bits is being checked for errors:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	1	1	0	1	1	0	1	0	0	0	1	0	0	1	1	0	1	1	1



1. **Is Parity Sum 1 correct (A) or incorrect (B)?**
2. **Is Parity Sum 2 correct (A) or incorrect (B)?**
3. **Which codeword bit is in error: none (A) Bit 5 (B) Bit 8 (C) Bit 19 (D) Bit 20 (E)**

Detecting and Correcting Multi-bit errors

Simple parity can detect a single-bit error, while a Hamming code and detect and correct a single-bit error. This capability is usually sufficient for memory, since memory errors are rare and different bits for the same word are usually stored in different chips (thus reducing the chance of a multi-bit error).

Other media such as disks, tapes, and networks are prone to error bursts which can damage several bits. To detect and/or correct errors in multiple bit errors you need a codeword with a Hamming distance larger than 3.

The general formulas for the required Hamming distances are:

Hamming distance needed to detect an "n-bit" error:

$$n + 1$$

Example: To detect a 3-bit error you need a code with a Hamming distance of 4

Hamming distance needed to correct an "n-bit" error:

$$2n + 1$$

Example: To correct a 3-bit error you need a code with a Hamming distance of 7

More advanced techniques such as Reed-Solomon codes and BCH codes can have much larger Hamming distances, but we won't be discussing them in this course.

Exercise 7 - Hamming Distance

What is the Hamming Distance of an error-correcting code?

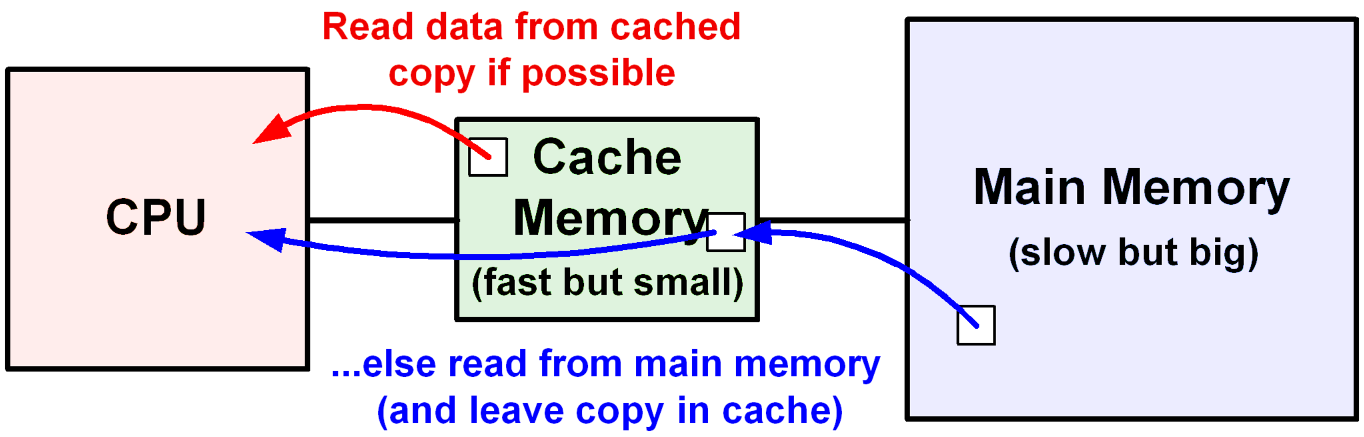
- **A** It’s the minimum number of parity bits in the codeword
- **B** It’s the maximum number of parity bits in the codeword
- **C** It’s the minimum number of bits that are different between valid codewords
- **D** It’s the maximum number of bits that are different between valid codewords
- **E** It’s the minimum number of bits that are different between invalid codewords
- **F** It’s the maximum number of bits that are different between invalid codewords

Cache Memory

Memory is a bottleneck for modern CPUs. A CPU that can execute 1 billion instructions per second must be able to fetch 1 billion instructions per second from memory. On top of that, many programs depend heavily on manipulating data in memory in order to operate.

Dynamic RAM (DRAM) that is used as main memory in most computer systems is relatively cheap but quite slow, with typical access times in the order of 20-60ns. That means the CPU can issue no more than about 20-50 million memory accesses per second – much slower than typically needed.

Cache memory is used to solve the problem. Cache is made from much faster (and more expensive) Static RAM (SRAM). A small amount of fast SRAM memory is used to hold a copy of recently-accessed data. Because the CPU often references the same memory areas over and over again, the memory subsystem can often find the information in the SRAM cache and deliver it immediately, rather than having to wait for it to be read from slower DRAM.


Read data from cached copy if possible
CPU
Cache Memory
(fast but small)
Main Memory
(slow but big)
...else read from main memory (and leave copy in cache)

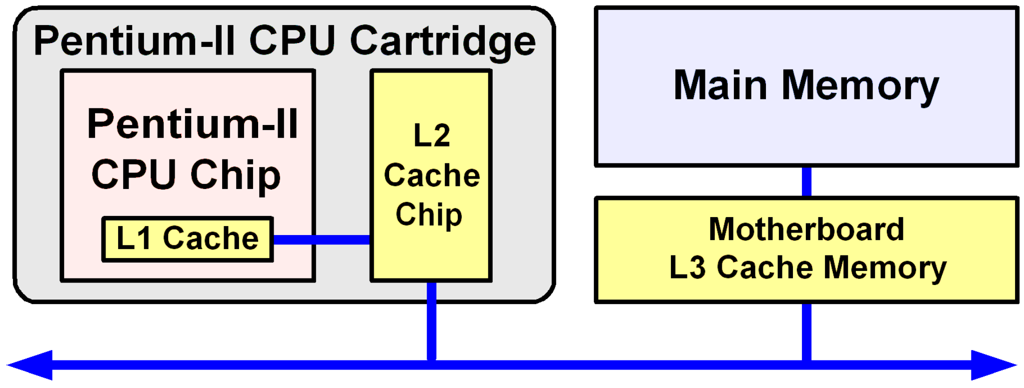
We'll be talking about exactly how cache memory works later in the course. Right now we'll discuss some general characteristics about cache memory.

Location of Cache Memory

Cache memory may be located on the same piece of silicon as the CPU chip, or as separate chips in the system, or in both places. Modern CPU chips can hold millions of transistors, and almost all of them now include at least some on-chip cache.

At one time it was common for motherboards to have sockets for cache memory, but the caches in modern CPUs are now so large that it's become unusual to see motherboards which include additional cache, except in certain very expensive, high performance systems.

Different cache locations in a system are known as "cache levels". "Level 1" cache is the cache that is closest to the CPU (it's usually right on the CPU chip). "Level 2" cache is the next closest, followed by "Level 3", etc.



This example shows the Pentium-II, which had a separate L2 cache chip within the CPU cartridge. Current Pentium chips have both L1 and L2 caches on the CPU chip itself.

The caches closer to the CPU are smaller and faster, while the caches closer to main memory are larger and slower.

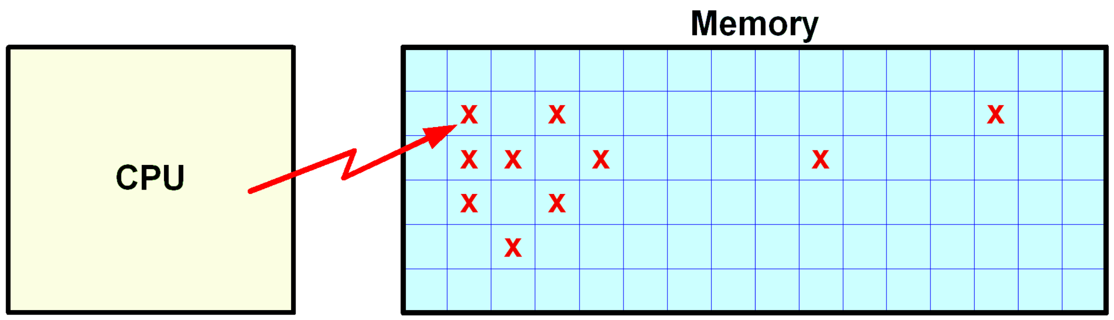
Cache Effectiveness

The effectiveness of cache memory is measured by the percentage of memory accesses that can be handled by the cache instead of having to go all the way to main memory. This is known as the cache "hit rate". Hit rates of over 95% are common.

There are two reasons why cache memory is so effective:

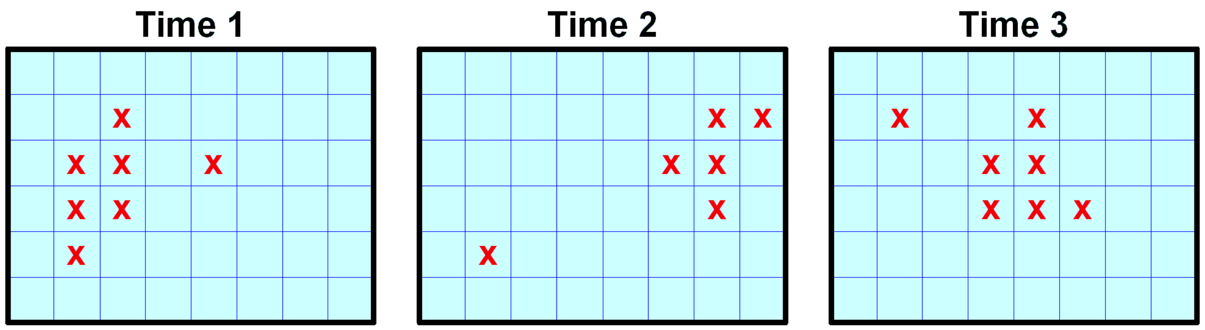
Spatial Locality

Most programs do most of their memory accesses to a relatively small set of memory addresses (the 80/20 rule):



Temporal Locality

The CPU tends to access a certain set of memory locations at a given time, then moves on to access other locations:



A Note on the "80/20 Rule"

The "80/20 rule" is a general statement that can be made about a lot of different situations where a minority of one thing is associated with the majority of something else. Some examples:

- **20% of the people own 80% of the wealth**
- **80% of the people live in 20% of the cities**
- **80% of car trips are made in just 20% of the hours of the day**
- **20% of the code in a program is used 80% of the time**

The "80/20 rule" isn't intended to be precise – in some of the examples above the actual ratios might range anywhere from 60/40 to 99/1.   But when you hear "80/20 Rule" the general idea is that most of thing "A" is associated with only a small amount of thing "B".

The "80/20 rule" was originally postulated in 1897 by Italian economist Vilfredo Pareto, and is sometimes known as "Pareto's Law".

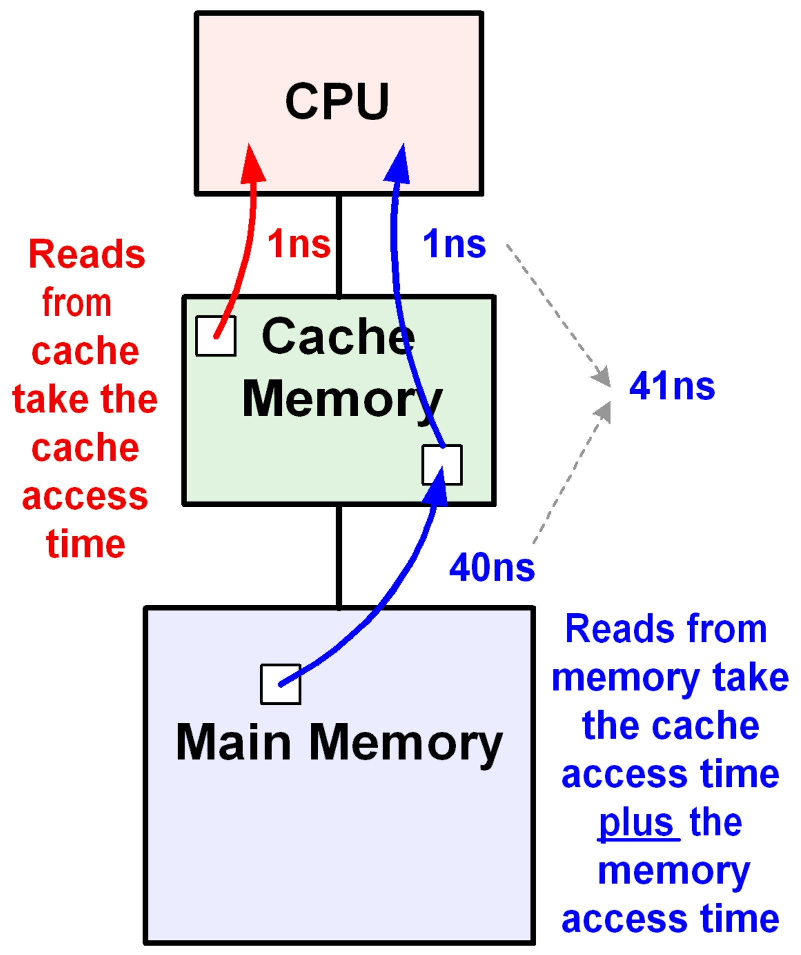
Effective Access Times

In a system with cache memory the time it takes to perform a memory read varies depending on whether the information is found in cache or not. Memory performance is influenced by the cache access time, the main memory access time, and the hit rate for the cache. You can calculate the average, or Effective Access Time for the memory subsystem using these factors.

The access time for the cache is used up for every memory access, because the cache must check every read to see if it contains the data. The main memory access time is only used for data that is not found in the cache.

If the cache hit rate is 80%, it means that only 20% of the reads need to wait for the main memory access time. This 20% is known as the cache miss rate.

The cache miss rate is equal to 100% minus the hit rate.


CPU
Cache Memory
Main Memory
Reads from cache take the cache access time
1ns
1ns
41ns
40ns
Reads from memory take the cache access time plus the memory access time

Effective Access Times

Let's look at an example of a cached memory system with the following characteristics:

Cache access time: 1ns
Cache hit rate: 90% (therefore, the miss rate is 10%, because 100% - 90% = 10%)
Main memory access time: 40ns

The 1ns cache access time is used up for every memory read. An additional 40ns memory access time is used up for only the 10% of reads that aren't found in the cache.

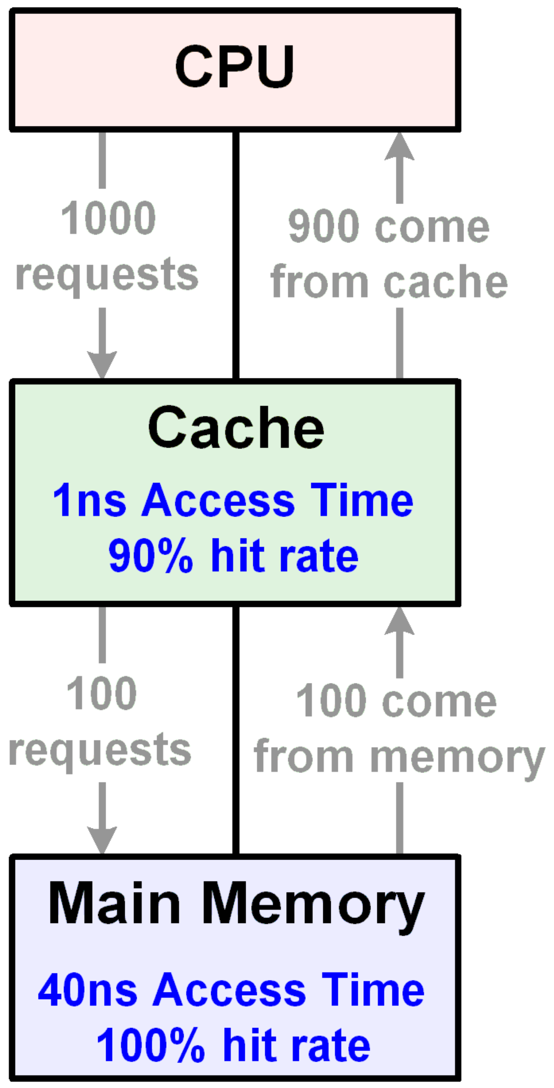
If the CPU issued 1000 memory reads, then they would break down as follows:

1000 reads – every read takes 1ns to see if the data is in the cache or not
100 reads – data not in cache – an additional 40ns are needed to read the data from main memory

So the total time used by all 1000 memory reads would be:

(1000 x 1ns) + (100 x 40ns) = 1000 + 4000 = 5000ns

If it takes 5000ns for 1000 reads, then the average time per read is 5000/1000 = 5ns. This is the Effective Access Time.



Effective Access Times

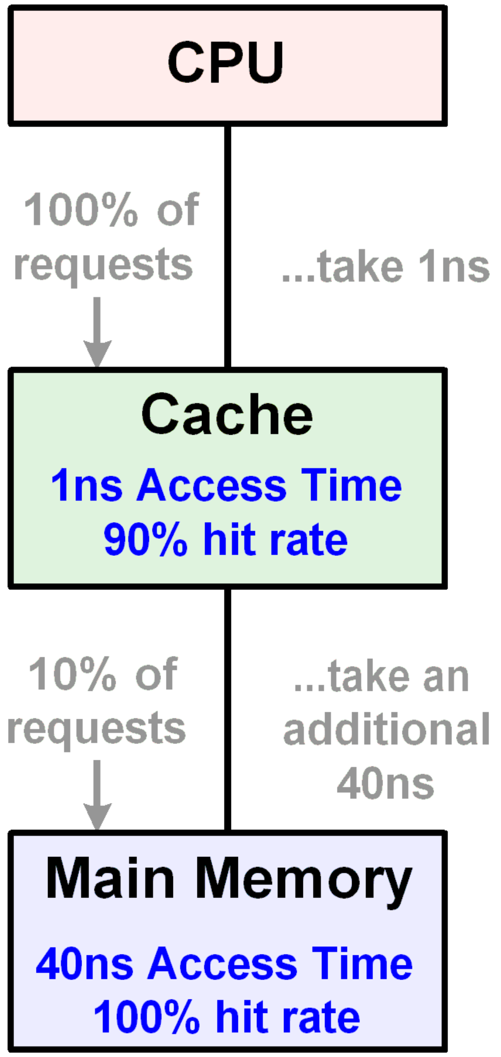
A simpler way to calculate the Effective Access Time is to factor the percentage of reads for each access time and their contribution to the overall average access time.

The access time of the first level of cache is <u>always</u> required for <u>every</u> read (ie, 100% of the reads).

For our sample cache, the access time of the memory is only required for 10% of the reads (the ones not found in cache)

So we can calculate the Effective Access Time as:

Cache:	**100% x 1ns =**	**1ns**
Memory:	**10% x 40ns =**	**<u>4ns</u>**
Total:		**5ns**

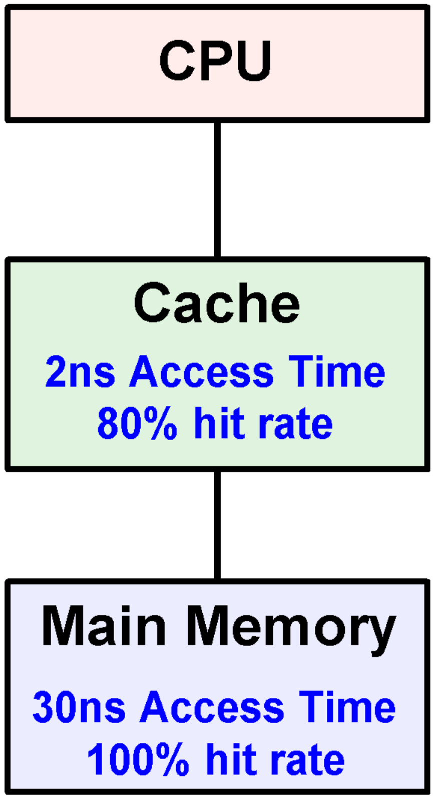


Since the 100% of the reads always take the cache access time, you can simplify the calculation for a 1-level cache system to:

<u>Cache access time</u> + ((100% - <u>hit rate</u>) x <u>Memory access time</u>)

Exercise 8 - Cache Access Times

A cache memory subsystem has the characteristics shown at the right:



1. **What is the percentage of reads that reach main memory?**

   20% (A) 80% (B) 100% (C)

2. **What is the effective access time for the memory subsystem?**

   2ns (A) 8ns (B) 26ns (C)

   30ns (D) 32ns (E)

Multilevel Cache Access Times

When there's more than one cache level, the Effective Access Time calculation needs to take account of the hit rates and access times all of the levels. Let's look at this example:

Level 1 cache – Access time = 1ns, hit rate = 90%

Every memory read needs to check the level 1 cache, so every read takes 1ns

Level 2 cache – Access time = 5ns, hit rate = 95%

Only 10% of the memory reads get to the level 2 cache (10% is the L1 cache miss rate). So 10% of the reads need an additional 5ns.

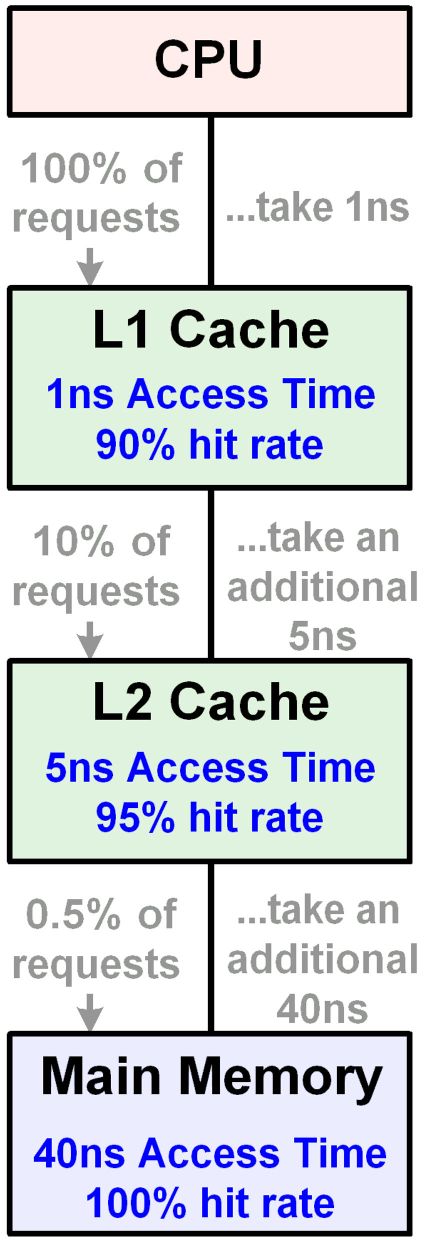
Main memory – Access time = 40ns

Only 10% of reads reach the L2 cache, and only 5% of those (the L2 cache miss rate) go on to main memory.

So 0.5% of the memory reads need an additional 40ns (because 10% x 5% = 0.5%)

So the Effective Access Time for this memory system is:

L1 Cache:	100% x 1ns =	1.0ns
L2 Cache:	10% x 5ns =	0.5ns
Memory:	0.5% x 40ns =	0.2ns
Total:		1.7ns



Multilevel Cache Access Times

With a multilevel cache, the most challenging problem is to calculate the percentage of reads that reach each level of the cache.

Remember that, for any given cache level, the number of reads that go on to the next level is:

the percentage of reads that reached this level

times

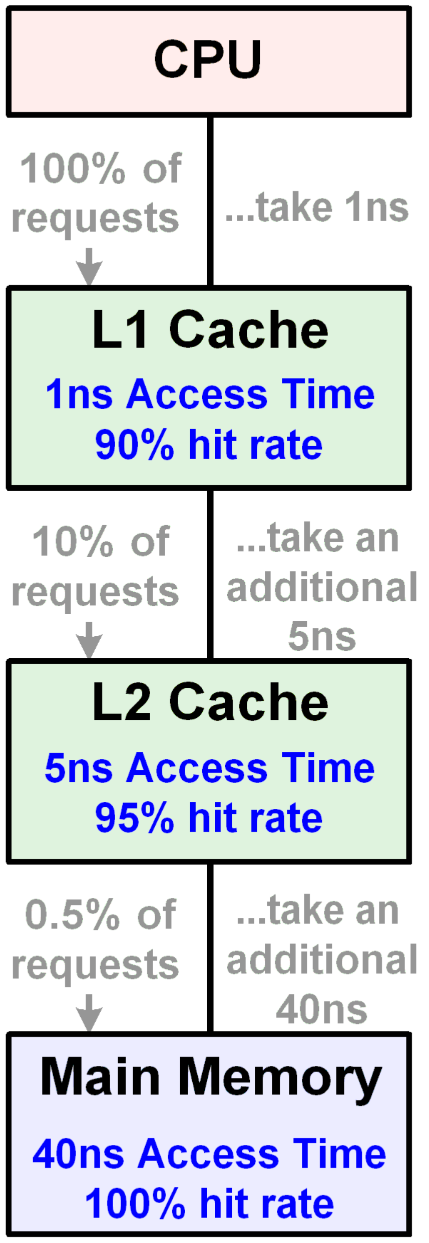
the miss rate of this level

In the previous example, this explains why main memory sees only 0.5% of all reads. It's because the L2 cache receives 10% of all memory requests, and it's miss rate is 5%. So the number of reads that go on to Main memory is:

10% (percent of reads that reached the L2 cache)

times

5% (miss rate of the L2 cache, 100% - 95% hit rate)



A Note on Percentages

A percentage is number that represents a fraction of 100. When you use a percentage in a calculation, you need to use 1/100th of the given number.

Some examples:

A 50% share of a $1,000,000 lottery:	$1,000,000 x 0.50 = $500,000
7% sales tax on $100:	$100 x 0.07 = $7.00
125% of rated capacity of 1000 kg:	1000kg x 1.25 = 1250kg

For calculations that include cache hit or miss rates given as percentages, divide the given percentage figure by 100 to use it as a multiplier value.

So in our multilevel cache problem:

0.5% x 40ns
= 0.005 x 40ns
= 0.2ns

Exercise 9 - Cache Access Times

A cache memory subsystem has the characteristics shown at the right:

	Hit Ratio	Access Time
Level 1 Cache	85%	2ns
Level 2 Cache	95%	6ns
Main Memory	100%	40ns

1. **What is the percentage of reads that reach the L1 cache?**

   15% (A) 85% (B) 100% (C)

2. **What is the percentage of reads that reach the L2 cache?**

   15% (A) 85% (B) 95% (C)

3. **What is the percentage of reads that reach main memory?**

   5% (A) 0.75% (B) 0% (C)

4. **What is the effective access time (i.e., average access time) for the cache?**

   2.3ns (A) 3ns (B) 3.2ns (C) 5.9ns (D) 14ns (E)

Boolean Logic Values in a Circuit

Computer systems are built from circuitry which manipulates Boolean values. A Boolean value is something that has only two possible states. For example:

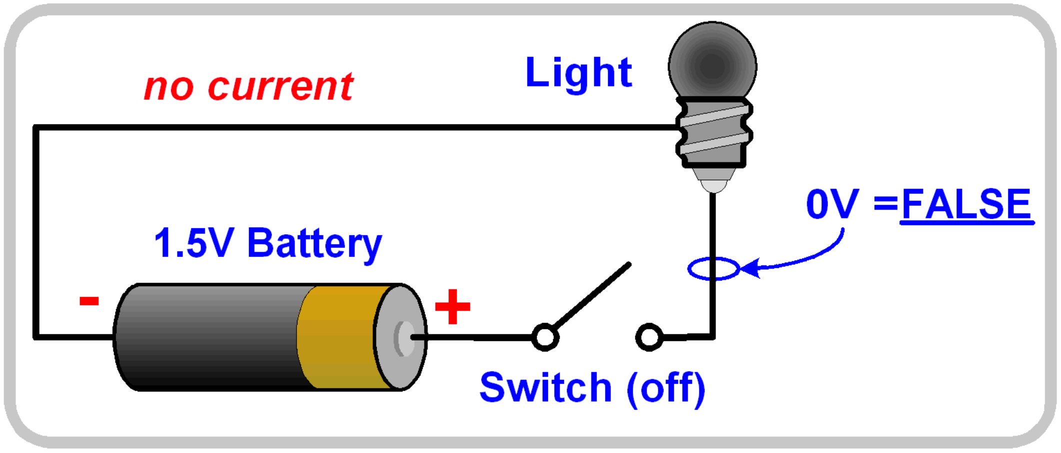
TRUE or FALSE

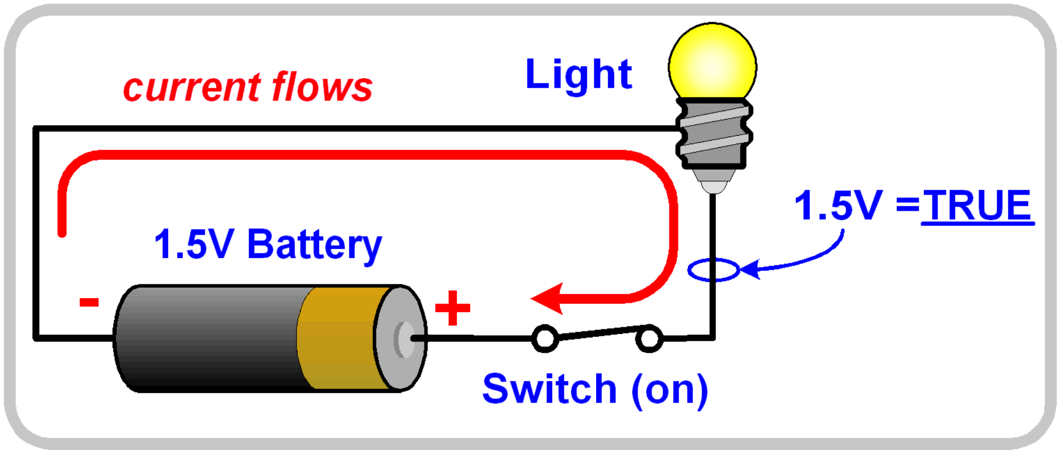
ON or OFF

YES or NO

1 or 0

In computer circuits, the two possible states are represented by the presence or absence of a particular voltage in a circuit. In the example at right, a 1.5V (V = "volts") battery will either light up a light bulb or not depending on whether the switch is closed.





In digital logic circuits the voltages that represent TRUE and FALSE are often 5V and 0V. TRUE and FALSE are usually written in shorthand form as "1" and "0". So therefore:

0 volts = logical "0"
5 volts = logical "1"

Boolean Logic Operations

Several Boolean logic operations are needed in order to build circuits that can perform work. The Boolean logic operations are the Boolean equivalent of arithmetic operations such as addition, multiplication, and so on. Like the arithmetic addition operation, most Boolean operations accept two inputs and produce one output.

A "truth table" can be used to show what a Boolean operation does. The truth table shows all of the possible input combinations and, for each, the result.

Here are truth tables for the basic Boolean operations:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

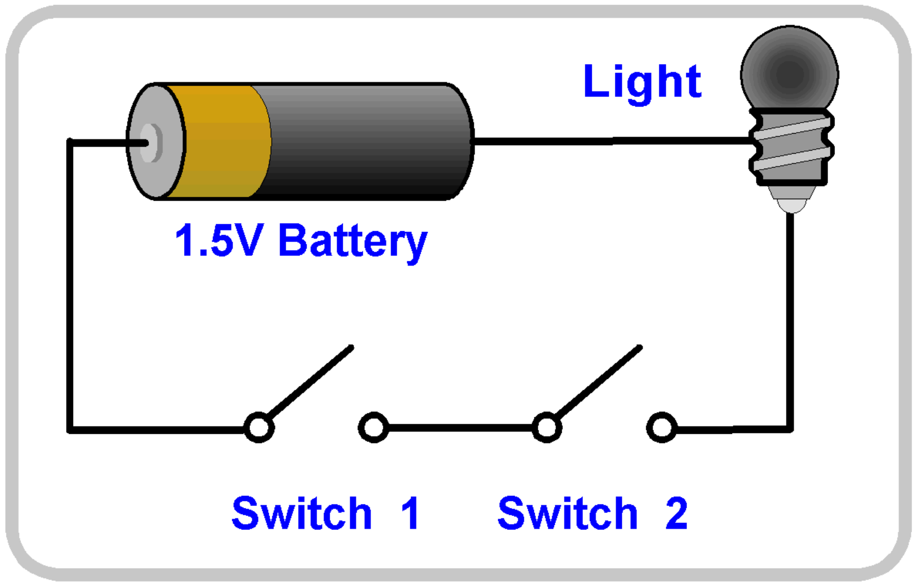
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

A	NOT A
0	1
1	0

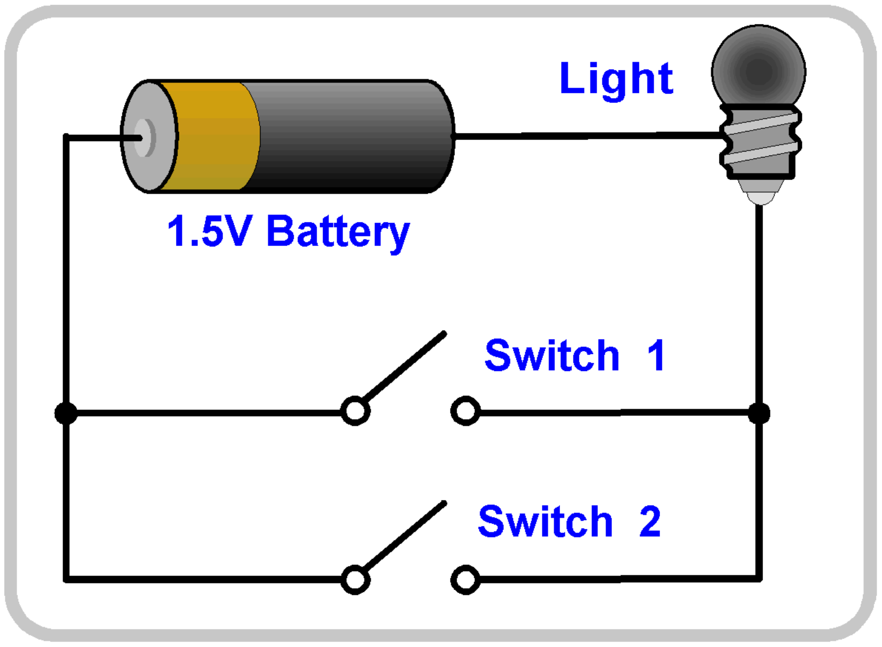
XOR is read as "Exclusive OR". Note that NOR produces exactly the opposite result of OR, and NAND produces exactly the opposite result of AND.

Building Boolean Logic Operators

The Boolean logic operators can be built using transistors or other switching devices. By wiring the switches in serial or parallel we can make the circuits perform Boolean "AND", "OR", and other operations. The examples below show how a couple of these work:



Switch 1 AND Switch 2 must be turned on in order to turn on the light.

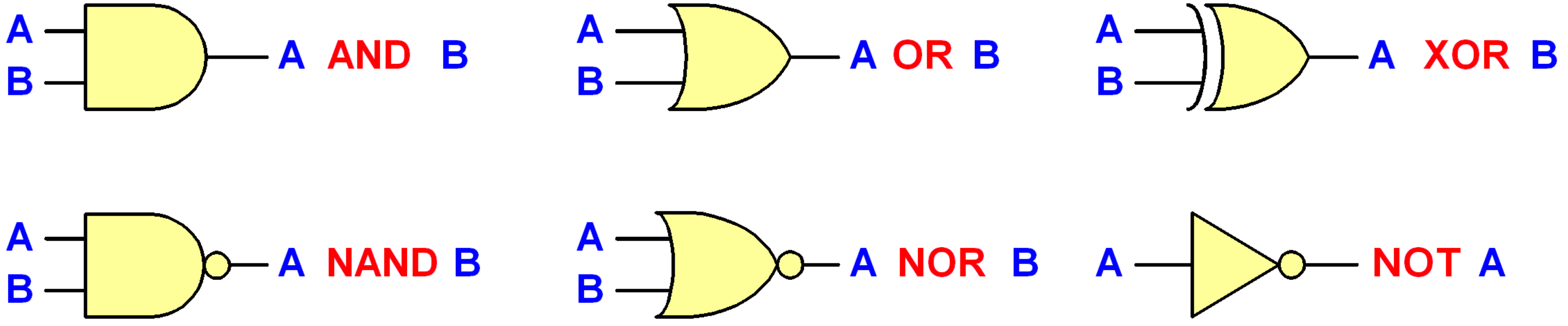


Switch 1 OR Switch 2 must be turned on in order to turn on the light.

In a computer system the switching devices are transistors, so you can see that it takes only a couple of transistors to build a Boolean logic function.

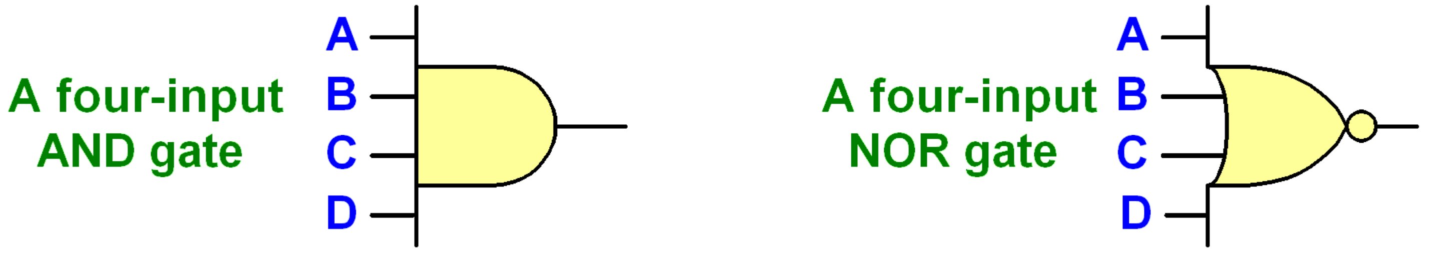
Symbols for Logic Diagrams

All the switches are tedious to draw and what they do isn't very obvious, so when drawing digital circuits there are several shorthand symbols that are used to show the various Boolean logic operations:



Note that the lower row of symbols has a small circle at the output. All of these functions produce a "negative" output (in other words, the opposite value that a normal gate would produce). The circle is an indication of this negative value, and is sometimes called an "inversion bubble".

In a circuit, the Boolean operators are often called "logic gates". It's also possible to have gates with more than two inputs:



Boolean Algebraic Symbols

When writing Boolean operations in algebraic form, the following symbols are used to represent the various operators:

A AND B	A OR B	A XOR B
$A \cdot B$	$A + B$	$A \oplus B$
A NAND B	A NOR B	NOT A
$\overline{A \cdot B}$	$\overline{A + B}$	$\overline{A}$

Note the overbar that is used for all of the negative operators. This overbar is the equivalent of the "inversion bubble" used in the logic diagramming symbols.

Be careful not say "AND" when reading the "+" symbol for the Boolean OR operation.

A shorthand notation can also be used for the AND and NOT operators:

A AND B	NOT A
$A\,B$	A'

The "AB" shorthand form is similar to the algebraic notation where "AB" means "A times B".

Precedence of Operations

In algebraic form the operations are normally performed from left to right, except that the NOT operator has the highest precedence followed by AND / NAND, and then by OR / NOR and XOR. So for example, the following expression:

$$A + B \cdot C$$

Should be read as "B is ANDed with C and the result is ORed with A". However it can be confusing to rely on the implicit precedence of operators, so it's best to show precedence explicitly using parenthesis:

$$A + (B \cdot C)$$

It's not necessary to use parenthesis with the NOT operator. The NOT "overbar" is applied to whatever falls below it *after* the operations below have been completed. Note the difference between the following two expressions:

$$\overline{A} + \overline{B}$$

NOT-A is ORed with NOT-B

$$\overline{A + B}$$

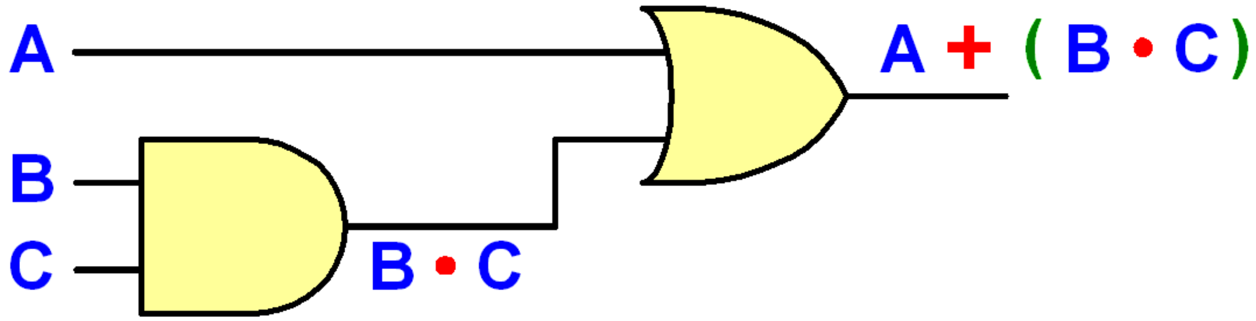
A is ORed with B and *then* the result is NOTed (same as NOR)

Algebraic vs. Logic Circuit Notations

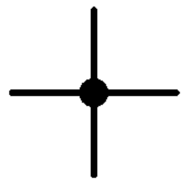
Every Boolean expression written in Algebraic form can also be written as a logic circuit using the logic symbols shown earlier. For example, the following Boolean expression:

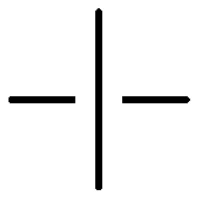
$$A + (B \cdot C)$$

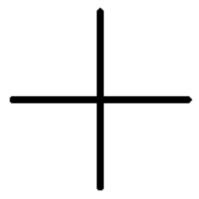
Can be drawn in circuit form using AND and OR logic gates as follows:



Logic diagrams are usually written so that the flow of signals runs from left-to-right and top-to-bottom where possible. If you can't avoid having lines cross over each other, use the following notation to show whether the lines are joined (connected) or separate:

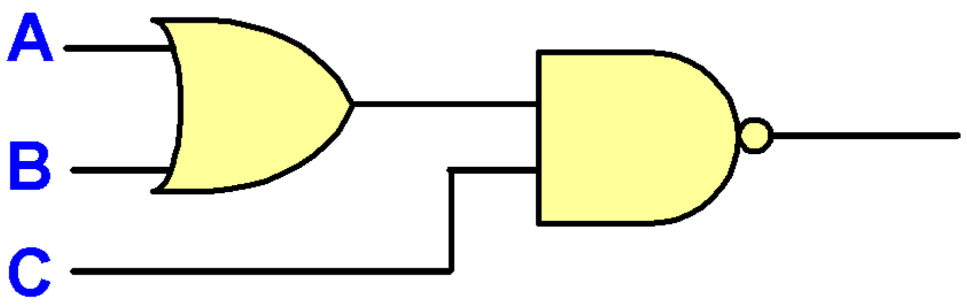

Joined


Unjoined (best)


Unjoined (alternate)

Exercise 10 – Circuit vs. Algebraic Expressions

Which of the algebraic expressions corresponds to this circuit diagram?



A $(A \cdot B) + C$

B $(A \cdot B) + \overline{C}$

C $(\overline{A \cdot B}) + C$

D $\overline{(A \cdot B) + C}$

E $(A+B) \cdot C$

F $(A+B) \cdot \overline{C}$

G $(\overline{A+B}) \cdot C$

H $\overline{(A+B) \cdot C}$

Why are they called “Gates”?

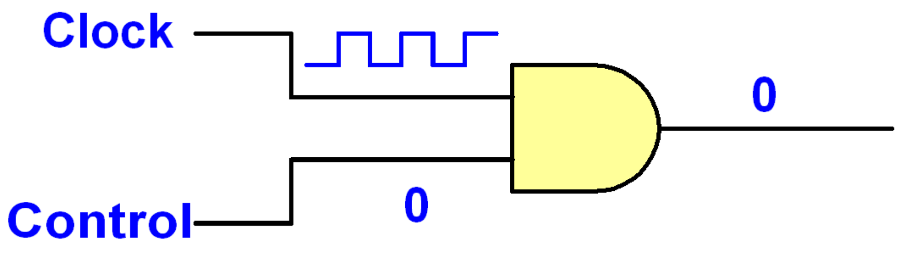
You will often hear the word “gate” to describe circuits which implement Boolean functions. You may be curious as to this word is used.

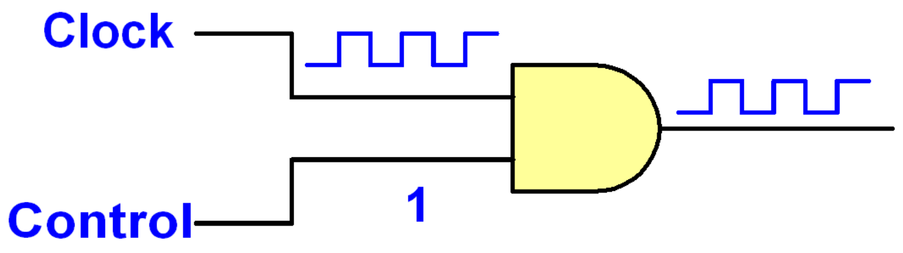
The reason is that AND and OR logic elements can be used to “gate” a signal – that is, to allow the signal to pass or to block it.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Remembering the truth table for the AND operation, consider an AND gate with two inputs:

- A “clock” signal which regularly pulses on and off
- A control signal





When the control signal is FALSE, the clock signal is blocked from passing through the AND gate because an AND gate always produces a FALSE output when either input is FALSE.

When the control signal is TRUE, the clock signal passes through the AND gate unchanged.

It is because of this ability of a Boolean logic element to allow a signal to pass through or be blocked that we call them “gates”.

Using Boolean Logic to Solve a Problem

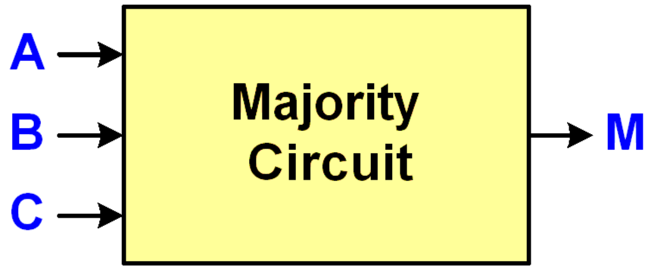
Boolean logic can be used to solve problems that involve Boolean (two-state) values. To see how this is done, let's take the following problem:

> **Create a Boolean logic circuit which will accept three inputs labeled A, B, C and produce a single output labeled M. The output should be true whenever most of the inputs are true.**

This problem will produce a circuit that implements a 3-input "majority function". It could be used by a three-person panel of judges to show if most of the judges voted "Yes" or not.

Without knowing how the logic will operate, we can draw a generic Boolean expression or a generic circuit diagram:

M = majority(A, B, C)



Our task is to find out what combination of Boolean operations need to be applied to the A, B, and C inputs to get the correct output. Once we know the Boolean operations we can devise a logic circuit to perform the work.

Creating a Truth Table

The first step is to draw a truth table to show each possible combination of inputs, and for each combination the expected result.

Since we have three Boolean inputs, there are a total of 2^3 = 8 possible input combinations. So our truth table will need 8 rows.

A	B	C	
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Once we've drawn 8 combinations of the three input variables, we fill in the result column with the TRUE (1) or FALSE (0) result required by the problem we're trying to solve.

In this problem, we need to produce a TRUE output whenever most of the inputs are true. So we look for rows in the truth table where two or more of the three inputs are TRUE. For each such row we draw a "1" in the result column to indicate that the result is TRUE for this combination of inputs.

Then we draw a "0" in all of the other rows of the truth table to indicate that the result is FALSE for those combinations of inputs.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Deriving a "Sum of Products" Expression

The next step is look at our truth table and identify what the input values are for each TRUE output.

A	B	C	M	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	Case 1
1	0	0	0	
1	0	1	1	Case 2
1	1	0	1	Case 3
1	1	1	1	Case 4

Our table has four cases where the result is TRUE:

Case 1: A=0 and B=1 and C=1 (in other words: $\overline{A} \bullet B \bullet C$)

Case 2: A=1 and B=0 and C=1 (in other words: $A \bullet \overline{B} \bullet C$)

Case 3: A=1 and B=1 and C=0 (in other words: $A \bullet B \bullet \overline{C}$)

Case 4: A=1 and B=1 and C=1 (in other words: $A \bullet B \bullet C$)

Our expression must produce a true result when the inputs correspond to Case 1, OR when they correspond to Case 2, OR Case 3, OR Case 4. So we take the input values for each case and OR them together:

$$M = \underbrace{(\overline{A} \bullet B \bullet C)}_{\text{Case 1}} + \underbrace{(A \bullet \overline{B} \bullet C)}_{\text{Case 2}} + \underbrace{(A \bullet B \bullet \overline{C})}_{\text{Case 3}} + \underbrace{(A \bullet B \bullet C)}_{\text{Case 4}}$$

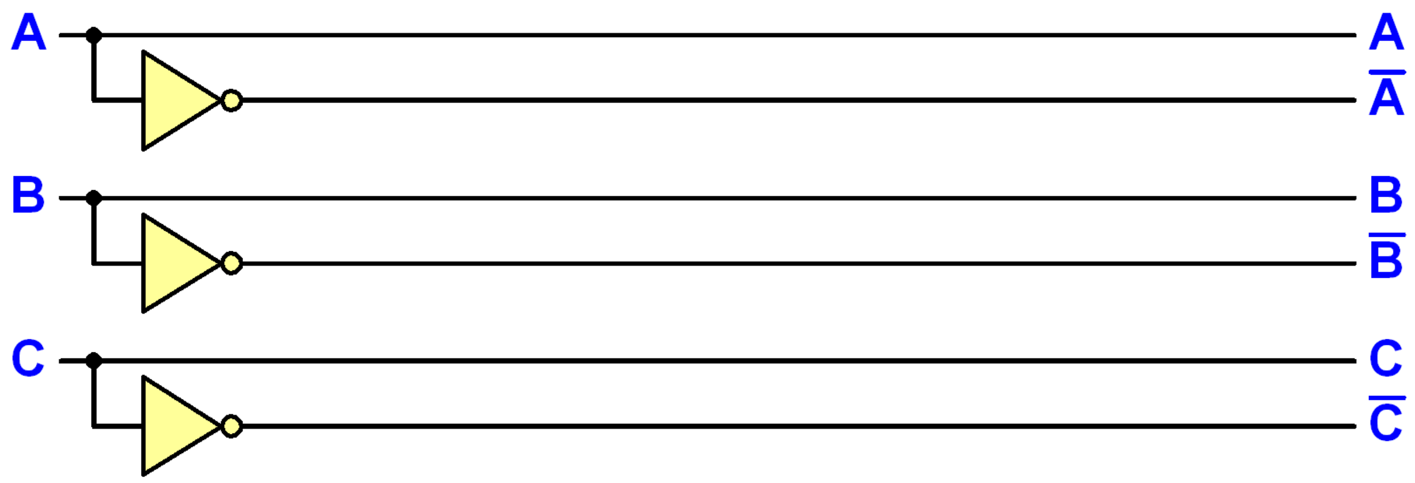
This form of expression is known as "Sum of Products" form, because each "product" (variables joined by AND) is "summed" (joined by OR).

Drawing the Boolean Logic Circuit - 1

The Sum of Products expression requires A, B and C inputs and also their negative values.

$$M = (\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C}) + (A \bullet B \bullet C)$$

An easy way to draw our Boolean Logic Circuit is to start off with the three input values and use NOT logic elements to create the negative values that we will need:



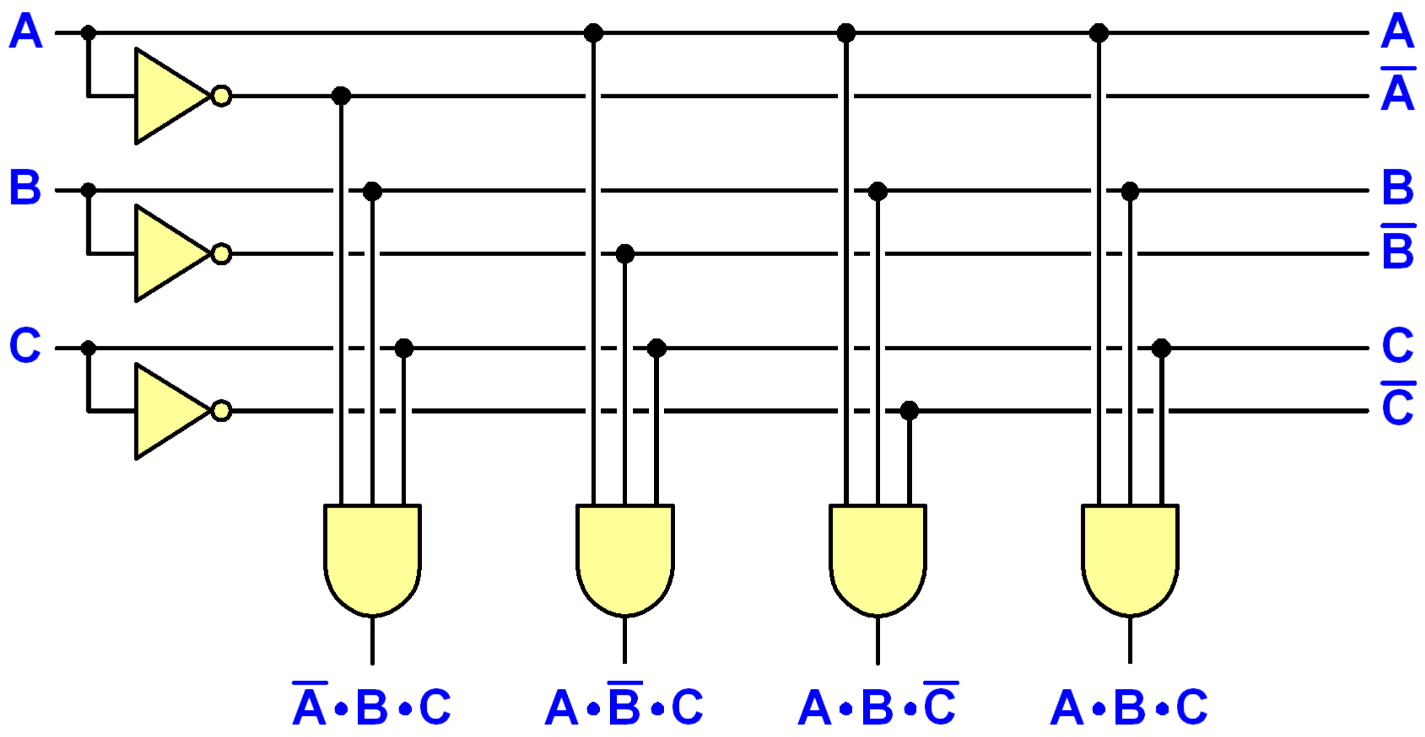
We now have all of the input variables as well as their negated forms ready for the next step.

Drawing the Boolean Logic Circuit - 2

$$M = (\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C}) + (A \bullet B \bullet C)$$

The next step is to add AND gates which correspond to the product terms in the expression. There are four products, so we'll need four gates. And each product has three variables, so we'll need to use three-input AND gates.

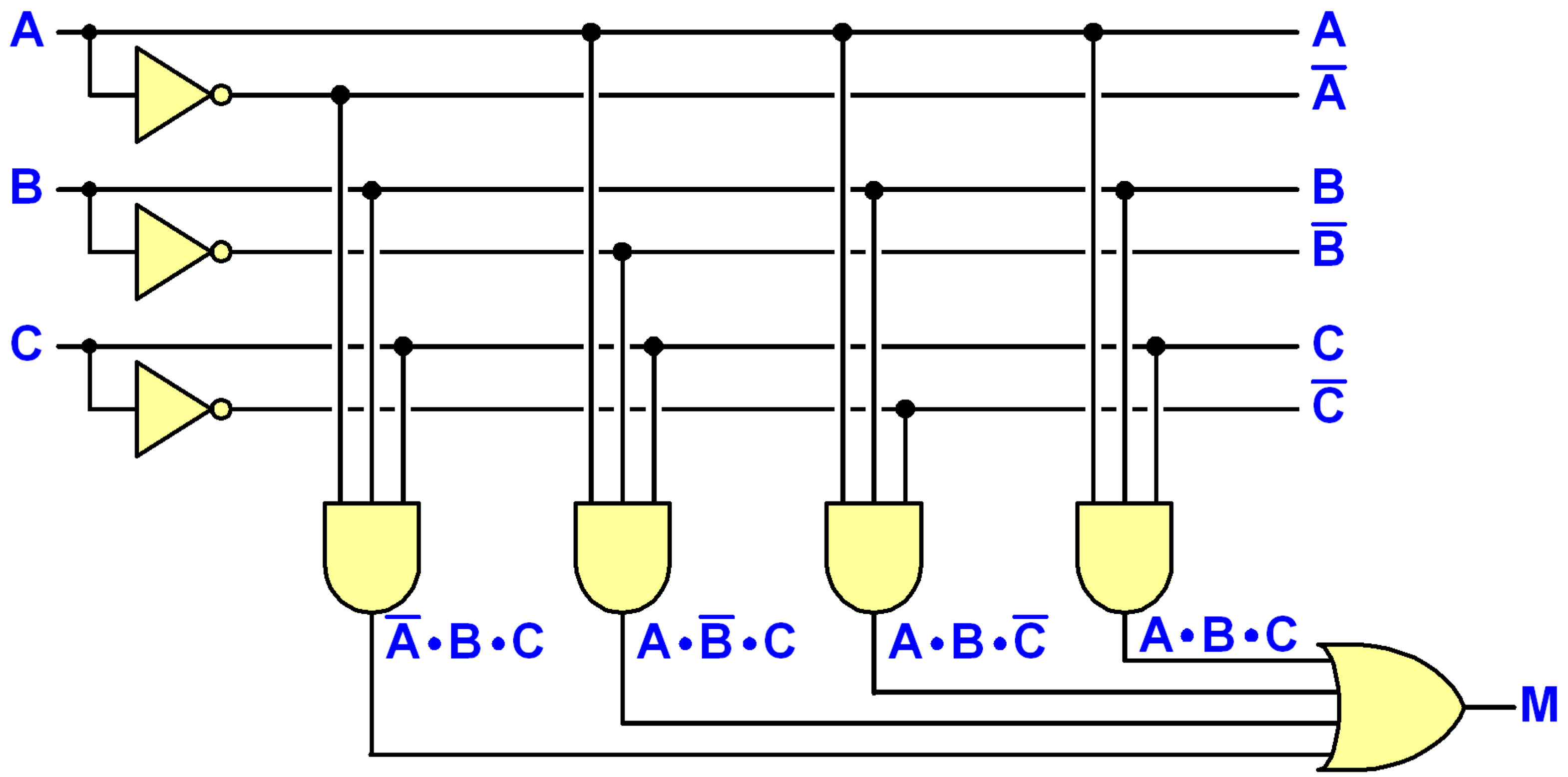
We connect the three inputs for each AND gate to the equivalent A, B and C values in each of the product terms:


A
B
C
A
$\overline{A}$
B
$\overline{B}$
C
$\overline{C}$
$\overline{A} \bullet B \bullet C$
$A \bullet \overline{B} \bullet C$
$A \bullet B \bullet \overline{C}$
$A \bullet B \bullet C$

Drawing the Boolean Logic Circuit - 3

$$M = (\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet \overline{C}) + (A \bullet B \bullet C)$$

The last step is to add a final four-input OR gate which will accept the outputs of the four AND gates:


A
B
C
A
$\overline{A}$
B
$\overline{B}$
C
$\overline{C}$
$\overline{A} \bullet B \bullet C$
$A \bullet \overline{B} \bullet C$
$A \bullet B \bullet \overline{C}$
$A \bullet B \bullet C$
M

And that’s it – this is a Boolean logic circuit which implements a 3-input majority function.

Exercise 11 – Boolean Logic Circuit

Create a Boolean logic circuit which will determine whether or not to sound the warning buzzer in a car. The circuit must accept three inputs:

- **K – Key in ignition. If this input is TRUE then the key is in the ignition.**
- **H – Headlights are on. If this input is TRUE then the headlights are turned on.**
- **D – Door is open. If this input is TRUE then the car door is open.**

The circuit should produce one output:

- **B – Turn on Buzzer. This output should be TRUE if the car door is open with the headlights are on, or if the car door is open with the key in the ignition.**

Which of the following truth tables shows the needed results?

A

D	K	H	B
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

B

D	K	H	B
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

C

D	K	H	B
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

D

D	K	H	B
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Exercise 12 – Boolean Logic Circuit

Which of the Boolean expressions corresponds to the truth table?

D	K	H	B	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	Case 1
1	1	0	1	Case 2
1	1	1	1	Case 3

A: $(D \bullet \overline{K} \bullet H) + (D \bullet K \bullet \overline{H}) + (D \bullet K \bullet H)$

B: $(D + \overline{K} + H) \bullet (D + K + \overline{H}) \bullet (D + K + H)$

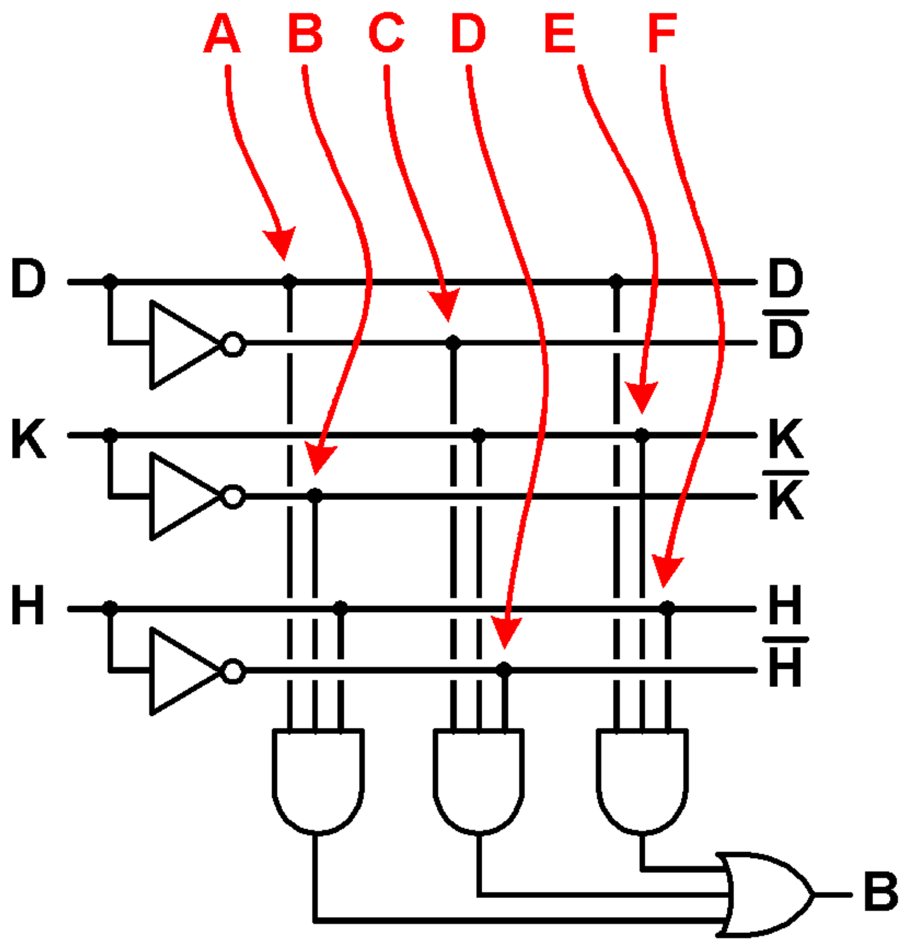
C: $(\overline{D} \bullet K \bullet \overline{H}) + (\overline{D} \bullet \overline{K} \bullet H) + (\overline{D} \bullet \overline{K} \bullet \overline{H})$

D: $(\overline{D} + K + \overline{H}) \bullet (\overline{D} + \overline{K} + H) \bullet (\overline{D} + \overline{K} + \overline{H})$

Exercise 13 – Boolean Logic Circuit

For the expression given, which of the circuit connections is wrong?

$$(D \cdot \overline{K} \cdot H) + (D \cdot K \cdot \overline{H}) + (D \cdot K \cdot H)$$



Boolean Identities

An important design goal is to simplify the design of the circuit as much as possible. There are a number of Boolean identities that allow a complex Boolean expression to be reduced to a much simpler expression. Each identity has an AND form and an OR form.

Here is a list of some of the more important identities:

	AND form	OR form
Identity law	$A \bullet 1 = A$	$A + 0 = A$
Null law	$A \bullet 0 = 0$	$A + 1 = 1$
Idempotent law	$A \bullet A = A$	$A + A = A$
Inverse law	$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$A \bullet B = B \bullet A$	$A + B = B + A$
Associative law	$(A \bullet B) \bullet C = A \bullet (B \bullet C)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + (B \bullet C) = (A + B) \bullet (A + C)$	$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$
Absorption law	$A \bullet (A + B) = A$	$A + (A \bullet B) = A$
De Morgan's law	$\overline{A \bullet B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \bullet \overline{B}$

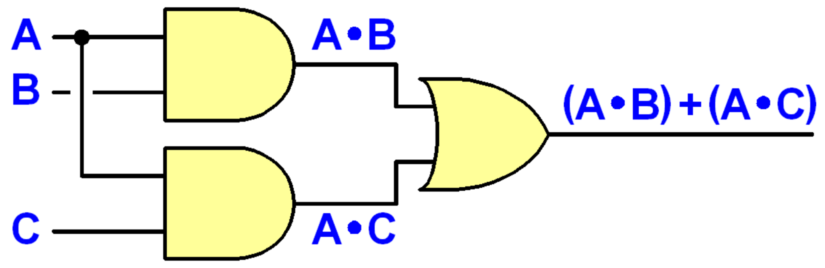
Note that De Morgan’s law can be extended to apply to 3 or more variables.

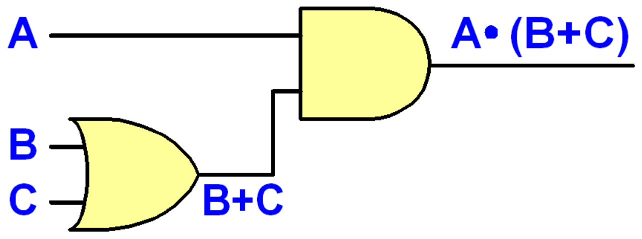
Boolean Identities

The Boolean identities ought to allow us to substitute simpler circuits in place of more complex ones. For example, the OR form of the distributive law:

$$A \cdot (B+C) = (A \cdot B)+(A \cdot C)$$

Suggests that we can replace the circuit at left below with the one on the right:





But how can we know for sure that these two circuits really do produce the same results?

One way to check is to draw the truth table for both expressions and see if they produce the same results:

A	B	C	(A•B)+(A•C)
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	A• (B+C)
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Circuit Equivalence

The expressions are a little too complex to simply write down the results directly, so we've added some intermediate columns with partial results to the truth tables. This allows us to solve each expression a little bit at a time:

A	B	C	A•B	A•C	(A•B)+(A•C)
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

A	B	C	B+C	A•(B+C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

same

Since both results are the same for all combinations of inputs, the two circuits are equivalent.

Exercise 14 – Circuit Equivalence

Which of the following pairs of truth tables correctly show the equivalence between these expressions?

$$A \bullet (A+B) = A + (A \bullet B)$$

A

A	B	A+B	A•(A+B)
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

A	B	A•B	A+(A•B)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

B

A	B	A+B	A•(A+B)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

A	B	A•B	A+(A•B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

C

A	B	A+B	A•(A+B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

A	B	A•B	A+(A•B)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Simplifying Complex Expressions

The Boolean identities can be used to simplify expressions – in an actual circuit this means fewer transistors and lower cost.   Here's an example:

$$(\overline{A} \bullet B \bullet \overline{C}) \;+\; (A \bullet \overline{B} \bullet \overline{C}) \;+\; (A \bullet B \bullet \overline{C})$$

Commutative Law

$$(\overline{C} \bullet (\overline{A} \bullet B)) \;+\; (\overline{C} \bullet (A \bullet \overline{B})) \;+\; (\overline{C} \bullet (A \bullet B))$$

Distributive Law

$$\overline{C} \bullet (\,(\overline{A} \bullet B) \;+\; (A \bullet \overline{B}) \;+\; (A \bullet B)\,)$$

Commutative Law

$$\overline{C} \bullet (\;(\,(A \bullet B) \;+\; (A \bullet \overline{B})\,) \;+\; (\overline{A} \bullet B)\,)$$

Distributive Law

$$\overline{C} \bullet (\;(\,A \bullet (B + \overline{B})\,) \;+\; (\overline{A} \bullet B)\,)$$

Inverse Law

$$\overline{C} \bullet (\;(\,A \bullet 1\,) \;+\; (\overline{A} \bullet B)\,)$$

Identity Law

$$\overline{C} \bullet (\;A \;+\; (\overline{A} \bullet B)\,)$$

Distributive Law

$$\overline{C} \bullet (\,(A \bullet \overline{A}) + (A + B)\,)$$

Inverse Law

$$\overline{C} \bullet (\,0 + (A + B)\,)$$

Identity Law

$$\overline{C} \bullet (A \;+\; B)$$

Karnaugh Diagrams

You can see that using the Boolean Identities to simplify an expression is tedious and error-prone. There must be a better way…

There is a better way! It's called a "Karnaugh Diagram". A Karnaugh Diagram is a two-dimensional truth table drawn in a way so as to make it easy to visualize the simplified Boolean expression directly.

Here's our car buzzer truth table redrawn in Karnaugh Diagram format:

D	K	H	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

K H D	0 0	0 1	1 0	1 1
0	0	0	0	0
1	0	1	1	1

The Karnaugh diagram shows the same thing as the truth table, but the results are arranged in a matrix instead of as a column.

Using the Karnaugh Diagram

To use a Karnaugh diagram, first you identify the result squares that contain TRUE results as shown at right:

K	0	0	1	1
D H	0	1	0	1
0	0	0	0	1
1	0	1	1	1

Then you identify the expressions which identify the rows and the columns where the TRUE results lie.

The true results occur only in the row where D has a true value – so the row expression is simply "D".

The results are also in the columns where H is true OR K is true, so the column expression is "H + D".

K=1 or H=1

(K+H)

K	0	0	1	1
D H	0	1	0	1
0	0	0	0	0
1	0	1	1	1

D = 1

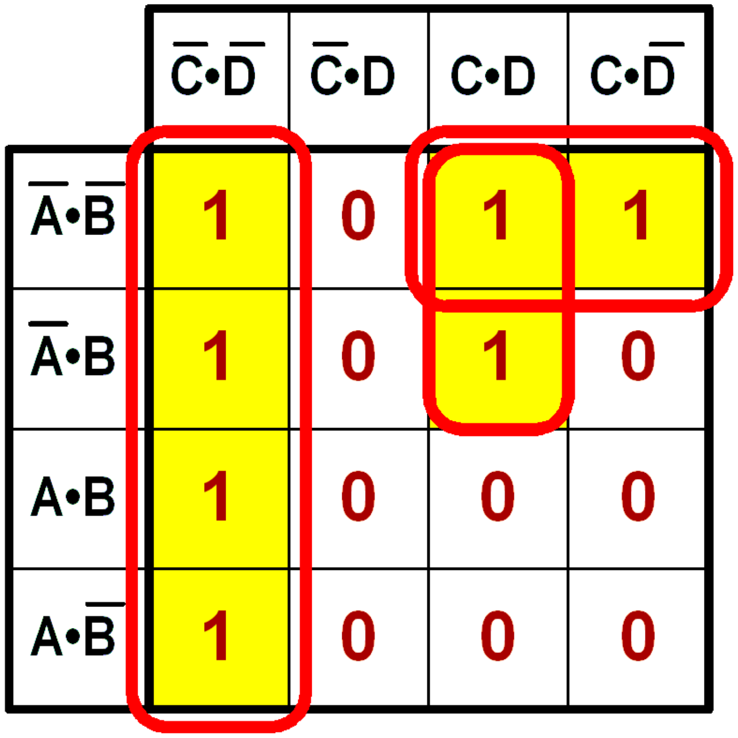
(D)

Finally, join the row and column expressions with AND, because the true results only apply when the row conditions AND the column conditions are true.

$D \cdot (K+H)$

Karnaugh Diagrams

Karnaugh diagrams are effective for simplifying Boolean expressions with up to four input variables.



	$\overline{C}\cdot\overline{D}$	$\overline{C}\cdot D$	$C\cdot D$	$C\cdot\overline{D}$
$\overline{A}\cdot\overline{B}$	1	0	1	1
$\overline{A}\cdot B$	1	0	1	0
$A\cdot B$	1	0	0	0
$A\cdot\overline{B}$	1	0	0	0

Beyond four variables, circuit designers use specialized software to simplify expressions.

Negative Logic

Any circuit or expression can also use "Negative Logic". Negative logic uses "0" values to represent logical TRUE conditions and vice versa. This is just as valid as positive logic, but it swaps AND and OR operators to achieve the same result.

As an example, consider the following "eating" problem. It accepts two inputs:

H – I am hungry. TRUE if I feel like eating something.
C – I have cash. TRUE if I have money to buy food.

And it produces one output:

E – I will go and eat in a restaurant.

This problem is solved with a simple AND operation:

E = H AND C **(I will eat a meal if I am hungry AND I have cash)**

Now watch what happens when we reverse the meaning of all the values. The inputs become:

F – I am full. TRUE when I don't feel like eating. (Same as "NOT-Hungry" above)
B – I am broke. TRUE if I have no money to buy food. (Same as "NOT-Cash" above)

And the output becomes:

S – I will skip the meal. (Same as "NOT-Eat" above)

Now the problem is solved with an OR operation:

S = F OR B **(I will skip a meal if I am full OR I am broke)**

Note that this is the same as: $\overline{E} = \overline{H}$ OR $\overline{C}$

Negative Logic and "Product of Sums Form"

We can use negative logic to create an alternate form of Boolean expression. This alternate form is called "Product of Sums" form, and we create it by looking at the zero results in the truth table. Let's look again at the truth table for the "majority function":

A	B	C	M	
0	0	0	0	Case 1
0	0	1	0	Case 2
0	1	0	0	Case 3
0	1	1	1	
1	0	0	0	Case 4
1	0	1	1	
1	1	0	1	
1	1	1	1	

For "product of sums" form, we write the sum terms of the expression by looking for the cases with zero results, using the negative value of the input conditions, and joining them with OR operators:

Case 1: $A+B+C$

Case 2: $A+B+\overline{C}$

Case 3: $A+\overline{B}+C$

Case 4: $\overline{A}+B+C$

Then we join each sum term with AND:

$$(A+B+C)\bullet(A+B+\overline{C})\bullet(A+\overline{B}+C)\bullet(\overline{A}+B+C)$$

If your Boolean problem produces a truth table with lots of "1" results and only a few "0" results, you can create a simpler Boolean expression by using the "Product of sums" form.

Showing that Product of Sums Form Works

It’s not very obvious that a Boolean expression written in “Product of Sums” form produces the same result as one written in “Sum of Products” form. Let’s look at the “majority function” truth table and see how the “Product of Sums” for works out for it:

$$(A+B+C) \bullet (A+B+\overline{C}) \bullet (A+\overline{B}+C) \bullet (\overline{A}+B+C)$$

A	B	C	M	$A+B+C$	$A+B+\overline{C}$	$A+\overline{B}+C$	$\overline{A}+B+C$	
0	0	0	0	0	1	1	1	0
0	0	1	0	1	0	1	1	0
0	1	0	0	1	1	0	1	0
0	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	0	0
1	0	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

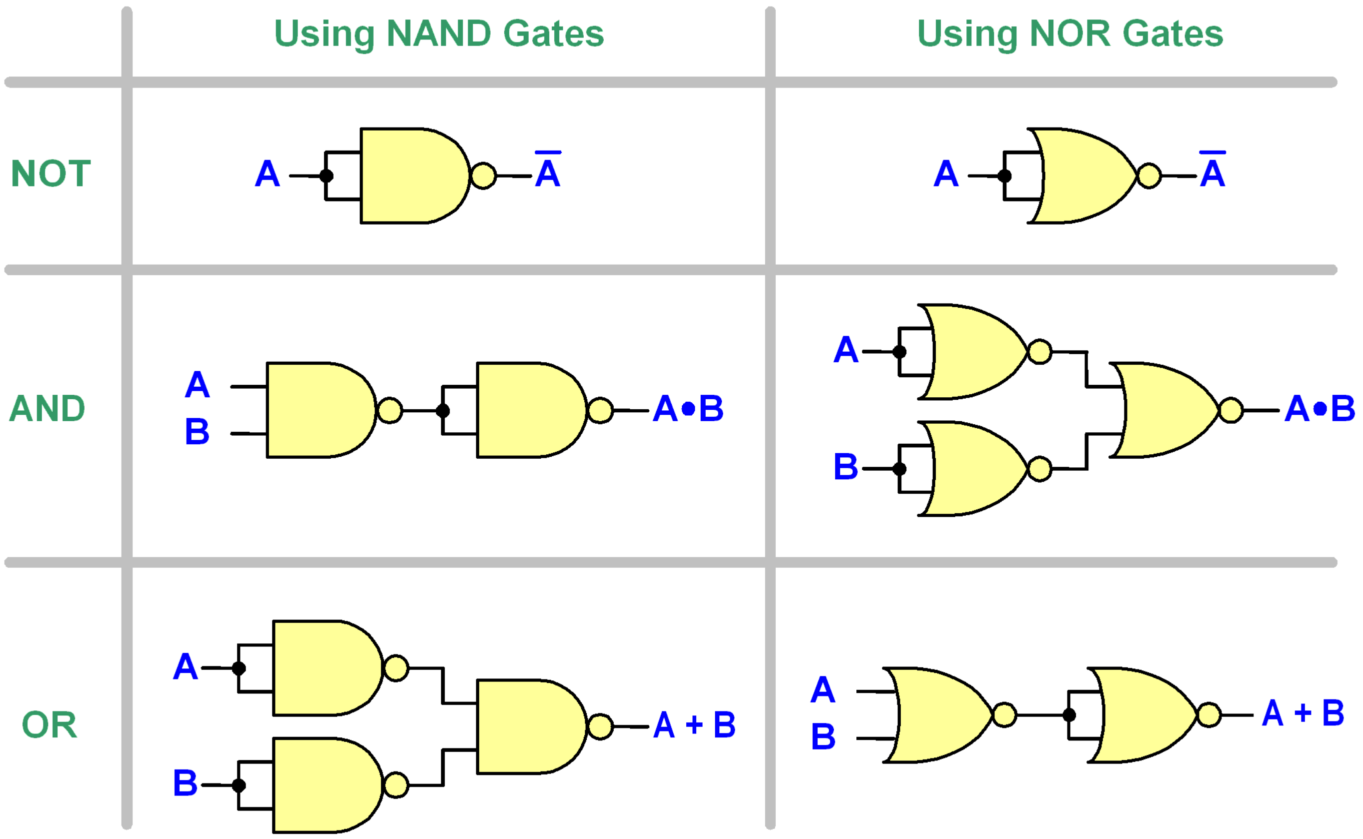
same

The truth table shows that the Product of Sums form does indeed give the correct result.

Reducing Gate Types

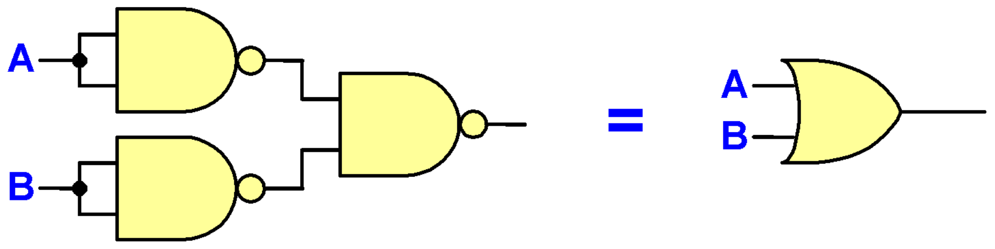
When building real-world circuits it's often very convenient if you can create a circuit that uses only one <u>type</u> of logic gate. This is because many logic chips hold several gates of the same type. If you can create a circuit that uses only type of gate, even if it means using more gates, you can often build the circuit using only a single chip.

NAND gates and NOR gates are used for this because they can be combined to do the work of any Boolean Logic function:



Reducing Gate Types

How can we tell that the NAND only circuit really produces the same result as an OR function?



$$\overline{\overline{A \cdot A} \cdot \overline{B \cdot B}} = A + B$$

We can draw a truth table to confirm that they are both equivalent. The truth table below shows the results for both Boolean expressions. The complex expression has been broken down into it's constituent parts to make it easier to fill in the truth table.

A	B	$A \cdot A$	$\overline{A \cdot A}$	$B \cdot B$	$\overline{B \cdot B}$	$\overline{A \cdot A} \cdot \overline{B \cdot B}$	$\overline{\overline{A \cdot A} \cdot \overline{B \cdot B}}$	$A + B$
0	0	0	1	0	1	1	0	0
0	1	0	1	1	0	0	1	1
1	0	1	0	0	1	0	1	1
1	1	1	0	1	0	0	1	1

same

Since both expressions produce the results for each input combination, they are equivalent.

Boolean Expressions in Your Programs

The Boolean operators that we've just talked about are the same as those used in conditional tests in the programs you write. Each expression in a conditional test yields a TRUE or FALSE result which can be combined using AND and OR operators. This means that the principles we've just discussed can also be applied to your programs.

For example, the equivalence of negative and positive logic mean that the following program statements will produce the same results:

```
if (   (Cheque_Amt > Acct_Balance)   &&
       (  (Overdraft_Permitted == FALSE)  | |  (Cheque_Amt > Overdraft_Limit)  )  )
              Deny_Cheque();
        else Process_Cheque();
```

…and

```
if (   (Cheque_Amt <= Acct_Balance)   | |
       (  (OverDraft_Permitted == TRUE)   &&  (Cheque_Amt <= Overdraft_Limit) ) )
              Process_Cheque();
        else Deny_Cheque();
```

You can use these principles to help make your programs simpler and easier to understand.

Key Concepts

- Memory organization and byte ordering
- How simple parity works
- The difference between data bits and a codeword
- The Hamming Distance of a codeword
- Hamming Distance for multi-bit error detection and error correction
- Cache systems and organization
- Calculating the effective memory access time for a system with cache memory
- Boolean logic operators and symbols
- Solving Boolean logic problems using Sum of Products form
- Equivalence of Boolean logic expressions and circuits
- Negative logic and Product of Sums form

What’s Next

- Look at Review Questions for this week
- Sign on to WebCT and do Module 2 – “Electronic Technology”
- Study for Quiz 1, which covers:
  - Week 1 and 2 lectures
  - WebCT modules 1 and 2
- Start working on Assignment 1 (for the material covered so far)
- Pre-read week 3 material