Parity Checking RAM

- These chips have an extra bit for every 8 bits of data
- When data is being written to the memory, the chip adds up the total number of 1s in each byte in the memory
  - If the total is odd, the parity bit is set to 1
  - If the total is even, the parity bit is set to 0
- Because this process always results in an **even** number of 1s (including the parity bit), it is known as an **Even Parity Scheme**
- Likewise, an **Odd Parity Scheme** would result from setting the parity bit to 1 if the number of 1s in the byte was e even and to 0 if the number was odd.
- When data is being read from the memory, the chip will determine that the number of 1s in each byte conforms to the parity scheme is operation. If so, no errors are reported, if not, something has gone wrong and the chip reports an error.

Parity Checking Example

- Assume an even parity scheme
- If the data byte is 11011011, the parity bit would be set to 0 so that the total number of 1s would be even.

110110110 — Parity

- Similarly, for the byte 11010011, the parity bit would be set to 1, again to ensure that the total number of 1s is even

110100111 — parity

Parity Example

- Alternatively, if an odd parity scheme were to be used, the parity bit for 11011011 would be 1 and for 11010011 would be 0.

Limitations of Parity Schemes

- It is possible to detect an error using parity checks but it is not possible to identify the error, nor to correct it.
- For example, if in an even parity scheme, a 9-bit data byte has an odd number of 1s – an error has occurred – but which bit(s) is wrong?
- Multiple bits errors are statistically unlikely to occur within the same byte – but possible.
  - So an error may go unnoticed.
- In the previous description we described parity at the byte level. Parity checks can be equally applied instead at the word level, if appropriate
- Parity checking is used, not only in memory, but in many communication schemes

Mission-Critical Systems and Error Correction Codes

- Mission-critical systems need a higher level of fault tolerance than parity checking can give.
- Error Correcting Codes (ECCs)
  - These use multiple extra bits to identify multi-bit errors within a byte and can correct single bit errors.

The table shows the number of check bits needed for words of different sizes in order to be able to correct a single bit error

Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

69

Words and Word Alignment

- A word is a fixed-size piece of data that is manipulated as single unit by the instruction set or by the hardware of the processor
- The number of bits in a word is called the word length
- The word length reflects the structure and operator of a computer
  - Register sizes
  - Maximum data transfer between processor and memory in a single operation
    - Memory characteristics
    - Data bus size
  - Largest possible physical address size

70

Words and Word Alignment

- In modern computers, bytes are generally grouped into 4-byte or 8-byte words and instructions manipulate an entire word as a single unit.
- In addition, to simplify design and to speed-up execution, many architectures require words to be **aligned** in memory at addresses that are multiples of the word size. This is called **aligning on natural boundaries**

71

Words and Word Alignment

- For example
  - A 4 byte word would be found in memory beginning at addresses 0, 4, 8 etc but not beginning at addresses 1, 2, 3, 5, 6, 7, etc
  - Similarly, 8-byte words are aligned at addresses 8, 8, 16, etc and not at addresses 1, 2, 3, 4, 5, 6, 7, 9, etc

72

Words and Word Alignment

- Sometimes, data in a word does not take up the entire word.
- Consider a machine with a 4-byte word size. In this word, we could use all 32 bits to store an integer in the range from -1 to -2^{31} and from 0 to 2^{31} -1, for example.
- However, suppose we only wanted to store a single ascii character. In that case 3 bytes of the word would go unused.
- There is therefore a trade-off between having a uniform word size, and the simplicity the that brings, and the amount of memory that might go unused.
- This trade-off also applies when different instructions in the instruction set have different lengths – we will see this later

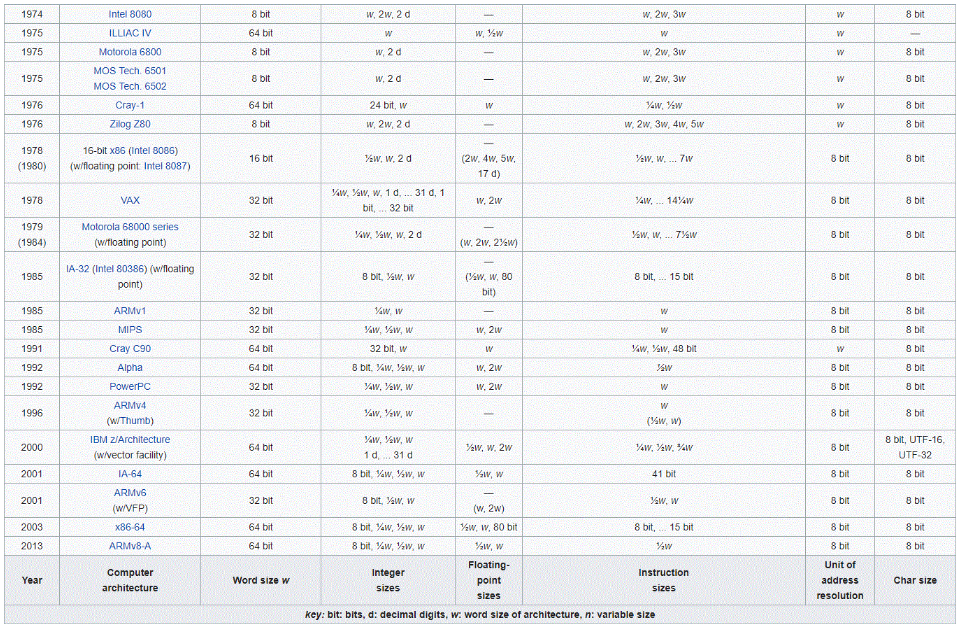
73

Words and Word Alignment

- One option, to use the memory more fully is to use multiple word lengths for each kind of data or to adopt the length of the smallest data type and to encode other data types as multiples of this minimum size (usually 1 byte)
- Both schemes can adversely affect performance by either complicating memory address calculations to account for non-alignment or by increasing the number of memory accesses when fetching data > 1 byte in size

- Thus the trade-off is ultimately in terms of speed versus space
- This trade-off is everywhere in computer architecture and in software algorithms

74



Year	Comp. Arch	Word Size	Integer Sizes	FP Sizes	Instruction Sizes	Addr	Char Sizes
1974	Intel 8080	8 bit	*w*, 2*w*, 2 d	—	*w*, 2*w*, 3*w*	*w*	8 bit
1975	ILLIAC IV	64 bit	*w*	*w*, ½*w*	*w*	*w*	—
1975	Motorola 6800	8 bit	*w*, 2 d	—	*w*, 2*w*, 3*w*	*w*	8 bit
1975	MOS Tech. 6501 MOS Tech. 6502	8 bit	*w*, 2 d	—	*w*, 2*w*, 3*w*	*w*	8 bit
1976	Cray-1	64 bit	24 bit, *w*	*w*	¼*w*, ½*w*	*w*	8 bit
1976	Zilog Z80	8 bit	*w*, 2*w*, 2 d	—	*w*, 2*w*, 3*w*, 4*w*, 5*w*	*w*	8 bit
1978 (1980)	16-bit x86 (Intel 8086) (w/floating point: Intel 8087)	16 bit	½*w*, *w*, 2 d	— (2*w*, 4*w*, 5*w*, 17 d)	½*w*, *w*, ... 7*w*	8 bit	8 bit
1978	VAX	32 bit	¼*w*, ½*w*, *w*, 1 d, ... 31 d, 1 bit, ... 32 bit	*w*, 2*w*	¼*w*, ... 14¼*w*	8 bit	8 bit
1979 (1984)	Motorola 68000 series (w/floating point)	32 bit	¼*w*, ½*w*, *w*, 2 d	— (*w*, 2*w*, 2½*w*)	½*w*, *w*, ... 7½*w*	8 bit	8 bit
1985	IA-32 (Intel 80386) (w/floating point)	32 bit	8 bit, ½*w*, *w*	— (½*w*, *w*, 80 bit)	8 bit, ... 15 bit	8 bit	8 bit
1985	ARMv1	32 bit	¼*w*, *w*	—	*w*	8 bit	8 bit
1985	MIPS	32 bit	¼*w*, ½*w*, *w*	*w*, 2*w*	*w*	8 bit	8 bit
1991	Cray C90	64 bit	32 bit, *w*	*w*	¼*w*, ½*w*, 48 bit	*w*	8 bit
1992	Alpha	64 bit	8 bit, ¼*w*, ½*w*, *w*	*w*, 2*w*	½*w*	8 bit	8 bit
1992	PowerPC	32 bit	¼*w*, ½*w*, *w*	*w*, 2*w*	*w*	8 bit	8 bit
1996	ARMv4 (w/Thumb)	32 bit	¼*w*, ½*w*, *w*	—	*w* (½*w*, *w*)	8 bit	8 bit
2000	IBM z/Architecture (w/vector facility)	64 bit	¼*w*, ½*w*, *w* 1 d, ... 31 d	½*w*, *w*, 2*w*	¼*w*, ½*w*, ¾*w*	8 bit	8 bit, UTF-16, UTF-32
2001	IA-64	64 bit	8 bit, ¼*w*, ½*w*, *w*	½*w*, *w*	41 bit	8 bit	8 bit
2001	ARMv6 (w/VFP)	32 bit	8 bit, ½*w*, *w*	— (w, 2w)	½*w*, *w*	8 bit	8 bit
2003	x86-64	64 bit	8 bit, ¼*w*, ½*w*, *w*	½*w*, *w*, 80 bit	8 bit, ... 15 bit	8 bit	8 bit
2013	ARMv8-A	64 bit	8 bit, ¼*w*, ½*w*, *w*	½*w*, *w*	½*w*	8 bit	8 bit
Year	**Computer architecture**	**Word size *w***	**Integer sizes**	**Floating-point sizes**	**Instruction sizes**	**Unit of address resolution**	**Char size**

***key:* bit: bits, d: decimal digits, *w*: word size of architecture, *n*: variable size**

75

Endianness

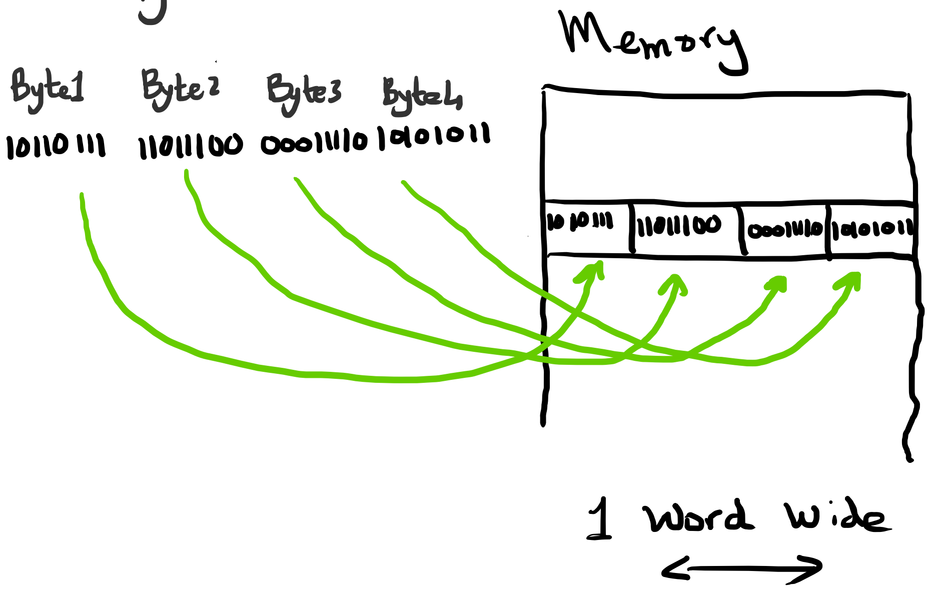
- Endianness refers to how the bytes of a word are stored in Memory.
  - There are 2 options: Big Endian and Little Endian
- Suppose we had a 4-byte word:

Byte 1	Byte 2	Byte 3	Byte 4
10110111	11011100	00011110	10101011

- In Big Endian, the most significant byte (i.e., Byte 1) is stored in the lowest memory address, Byte 2 is stored in the next lowest address, etc.
- In Little Endian, the least significant byte (.=i.e., Byte 3) is stored at the lowest address, Byte 2 at the next lowest, etc.

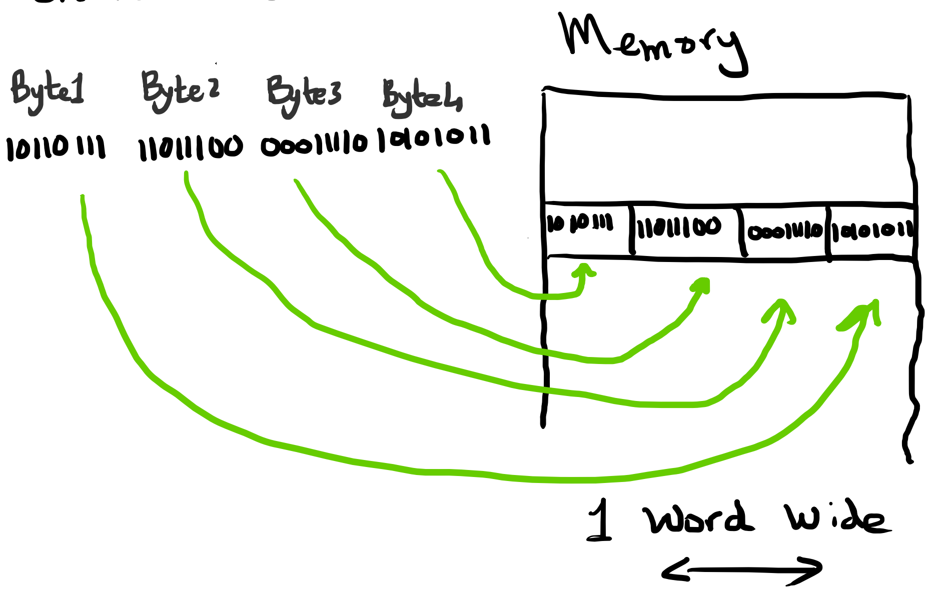
76

Big Endian



77

Little Endian



78

Locality of Reference

The tendency for a set of Memory locations to be accessed multiple times over a short period of time.

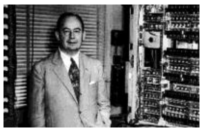
- There are two kinds:
  - locality in time, **temporal** locality
    - The same data is referred to multiple times over a short period of time
    - This data should be in fast access storage to speed up subsequent accesses
  - locality in space, **spatial** locality
    - A particular set of addresses are accessed multiple times over a short period of time
    - Nearby data should be stored in fast access storage to speed up subsequent accesses
  - Knowledge of locality offers a chance for predictability and thus performance optimization using **prefetching** and **caching**.
  - Locality follows from program structure: programming language constructs (for, while, if, etc) and data structures.

  - Thus proper software organization creates opportunities for locality of reference and proper memory hierarchies exploit this locality

79

Aside: von Neumann Machines

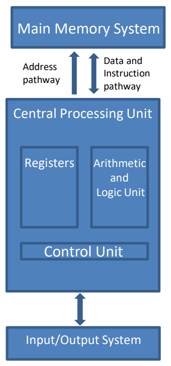
- Probably all of the computers with which you are familiar are von Neumann machines
- A computer is a von Neumann machine if
  - It has the 3 basic hardware systems already mentioned
    - A CPU
    - A main-memory
    - An I/O system
  - It is a stored–program computer
    - The main memory system holds the program that controls its operation, and the computer can manipulate its own program more or less as it can any other data in memory
  - It carries out instructions sequentially
    - The CPU executes, or at least appears to execute one operation at a time
  - It has, or at least appears to have, a single path between the main memory system and the control unit of the CPU; this is often referred to as the "von Neumann bottleneck".



80

Aside: von Neumann Machines

- Conventional von Neumann machines provide one pathway for addresses and a second pathway for data and instructions.



Classical von Neumann Architecture

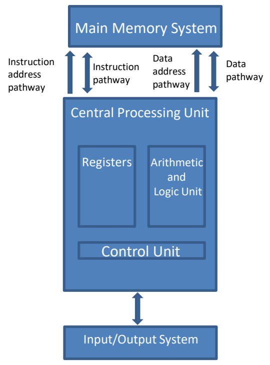
81

Aside: von Neumann Machines

Howard H. Aiken

- **Harvard Architectures** are a class of von Neumann machines except that they provide independent pathways for data addresses, data, instruction addresses and instructions – allowing the CPU to access instructions and data simultaneously.

- Harvard architectures are prevalent in microcontrollers.



Harvard Architecture

82