# CS222: Computer Architecture

Instructors:
Dr Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

الاحترام - الادب - الاخلاق
الطالب - المعيد - الدكتور

# Number Systems

- Numbers we can represent using binary representations
  - **Positive numbers**
    - Unsigned binary
  - **Negative numbers**
    - Two's complement
    - Sign/magnitude numbers

- What about **fractions**?

# Numbers with Fractions

- Two common notations:
  - **Fixed-point:** binary point fixed
  - **Floating-point:** binary point floats to the right of the most significant 1

# Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

$$01101100$$

$$0110.1100$$

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

  - Binary point is implied
  - The number of integer and fraction bits must be agreed upon beforehand

# Fixed-Point Number Example

- Represent $7.5_{10}$ using 4 integer bits and 4 fraction bits.

# Fixed-Point Number Example

- Represent $7.5_{10}$ using 4 integer bits and 4 fraction bits.

**01111000**

# Signed Fixed-Point Numbers

- **Representations:**
  - Sign/magnitude
  - Two's complement
- **Example:** Represent $-7.5_{10}$ using 4 integer and 4 fraction bits
  - **Sign/magnitude:**
  - **Two's complement:**

# Signed Fixed-Point Numbers

- **Representations:**
  - Sign/magnitude
  - Two's complement
- **Example:** Represent $-7.5_{10}$ using 4 integer and 4 fraction bits
  - **Sign/magnitude:**

    11111000
  - **Two's complement:**

| | |
|---|---|
| 1. +7.5: | 01111000 |
| 2. Invert bits: | 10000111 |
| 3. Add 1 to lsb: | + 1 |
| | 10001000 |

# Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write $273_{10}$ in scientific notation:

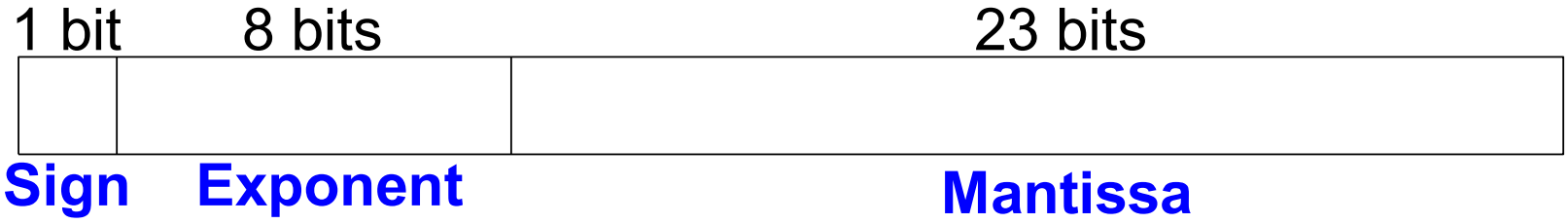
$$273 = 2.73 \times 10^2$$

- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

  - **M** = mantissa
  - **B** = base
  - **E** = exponent
  - In the example, M = 2.73, B = 10, and E = 2

# Floating-Point Numbers




- **Example:** represent the value $228_{10}$ using a 32-bit floating point representation

We show three versions –final version is called the **IEEE 754 floating-point standard**

# Floating-Point Representation 1

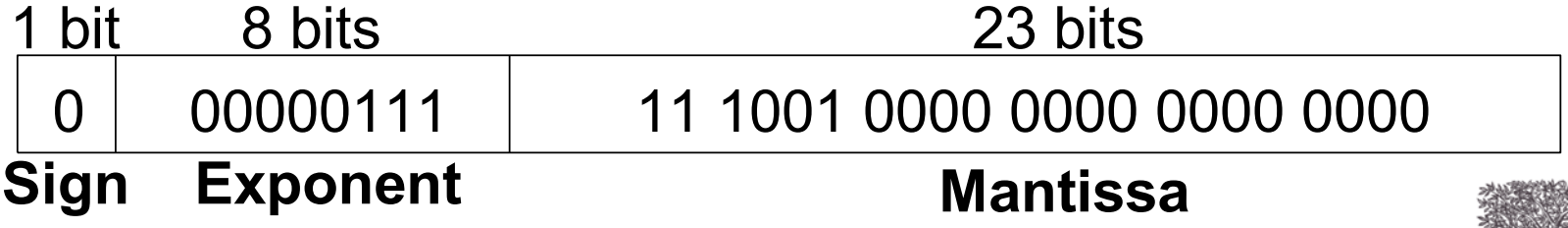
1. Convert decimal to binary (**don't reverse steps 1 & 2!**):

   $228_{10} = 11100100_2$

2. Write the number in "binary scientific notation":

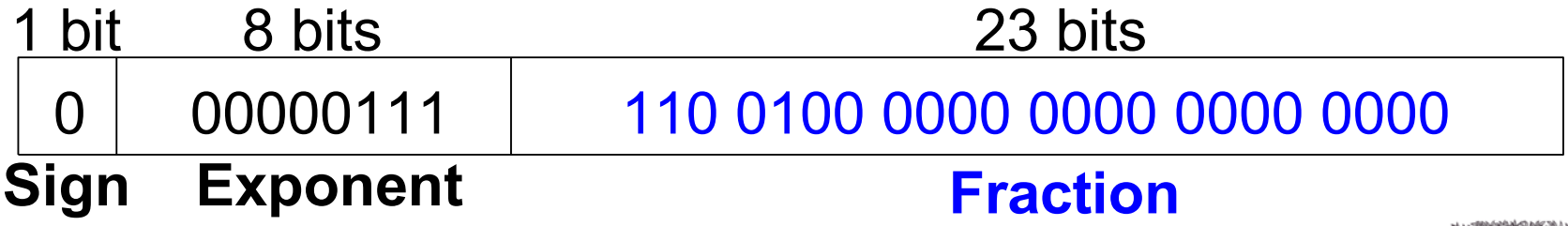
   $11100100_2 = 1.11001_2 \times 2^7$

3. Fill in each field of the 32-bit floating point number:
   - The sign bit is positive (0)
   - The 8 exponent bits represent the value 7
   - The remaining 23 bits are the mantissa



# Floating-Point Representation 2

- First bit of the mantissa is always 1:
  - $228_{10} = 11100100_2 = \mathbf{1}.11001 \times 2^7$
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field



# Floating-Point Representation 3

Positive and Negative exponents.

- *Biased exponent*: bias = 127 ($01111111_2$)
  - Biased exponent = bias + exponent
  - Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$

- The **IEEE 754 32-bit floating-point representation** of $228_{10}$

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 0 | 10000110 | 110 0100 0000 0000 0000 0000 |
| **Sign** | **Biased Exponent** | **Fraction** |

in hexadecimal: **0x43640000**

# Floating-Point Example

Write $-58.25_{10}$ in floating point (IEEE 754)

1. Convert decimal to binary:

   $58.25_{10} = \mathbf{111010.01_2}$

2. Write in binary scientific notation:

   $\mathbf{1.1101001 \times 2^5}$

3. Fill in fields:

   **Sign bit:** 1 (negative)
   **8 exponent bits:** (127 + 5) = 132 = $\mathbf{10000100_2}$
   **23 fraction bits:** **110 1001 0000 0000 0000 0000**

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 1 | 100 0010 0 | 110 1001 0000 0000 0000 0000 |
| **Sign** | **Exponent** | **Fraction** |

in hexadecimal: **0xC2690000**

# Floating-Point: Special Cases

| Number | Sign | Exponent | Fraction |
|---|---|---|---|
| 0 | X | 00000000 | 00000000000000000000000 |
| ∞ | 0 | 11111111 | 00000000000000000000000 |
| - ∞ | 1 | 11111111 | 00000000000000000000000 |
| NaN | X | 11111111 | non-zero |

# Floating-Point Precision

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127
- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

# Floating-Point: Rounding

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest
- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

# Floating-Point Addition

1. Extract exponent and fraction bits
2. Prepend leading 1 to form mantissa
3. Compare exponents
4. Shift smaller mantissa if necessary
5. Add mantissas
6. Normalize mantissa and adjust exponent if necessary
7. Round result
8. Assemble exponent and fraction back into floating-point format

# Floating-Point Addition Example

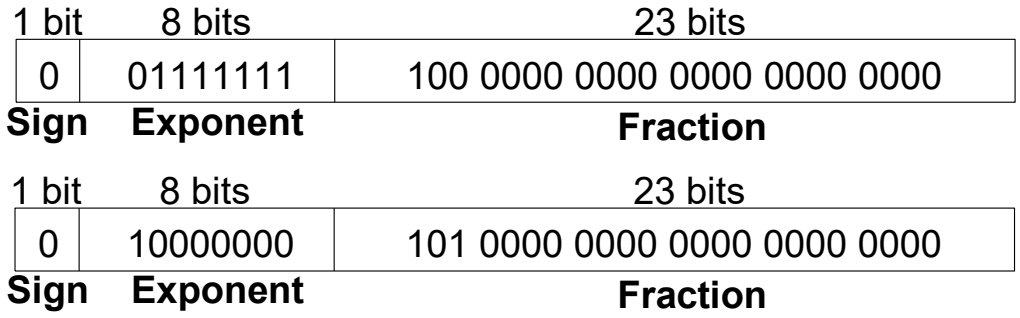
Add the following floating-point numbers:

0x3FC00000

0x40500000

# Floating-Point Addition Example

**1. Extract exponent and fraction bits**




For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

**2. Prepend leading 1 to form mantissa**

N1: 1.1

N2: 1.101

# Floating-Point Addition Example

3. **Compare exponents**

   127 – 128 = -1, so shift N1 right by 1 bit

4. **Shift smaller mantissa if necessary**

   shift N1's mantissa: 1.1 >> 1 = 0.11 $(\times 2^1)$

5. **Add mantissas**

$$\begin{array}{r} 0.11 \times 2^1 \\ +\ 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

# Floating Point Addition Example

6. **Normalize mantissa and adjust exponent if necessary**

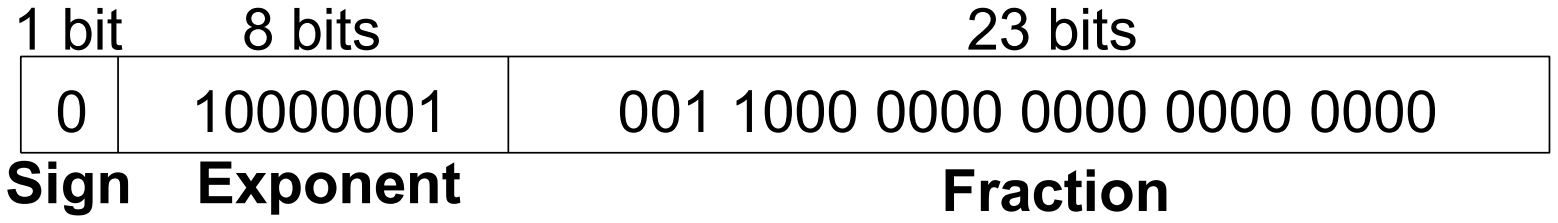
   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. **Round result**

   No need (fits in 23 bits)

8. **Assemble exponent and fraction back into floating-point format**

   $S = 0$, $E = 2 + 127 = 129 = 10000001_2$, $F = 001100..$

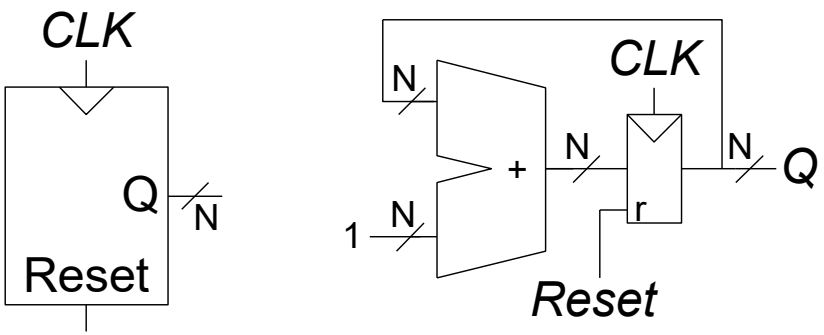



in hexadecimal: **0x40980000**

# Counters

- Increments on each clock edge
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

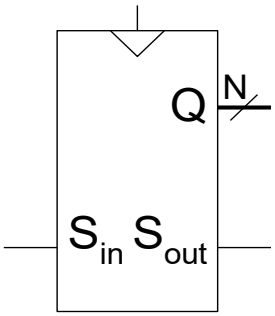
**Symbol** **Implementation**



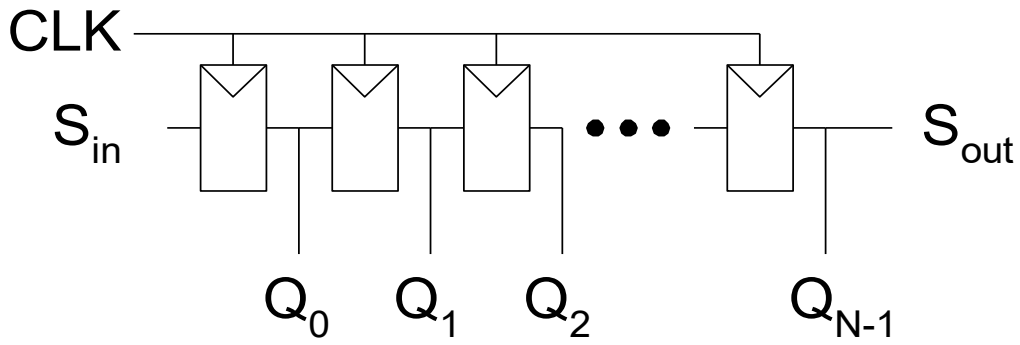
# Shift Registers

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input ($S_{in}$) to parallel output ($Q_{0:N-1}$)

**Symbol:**

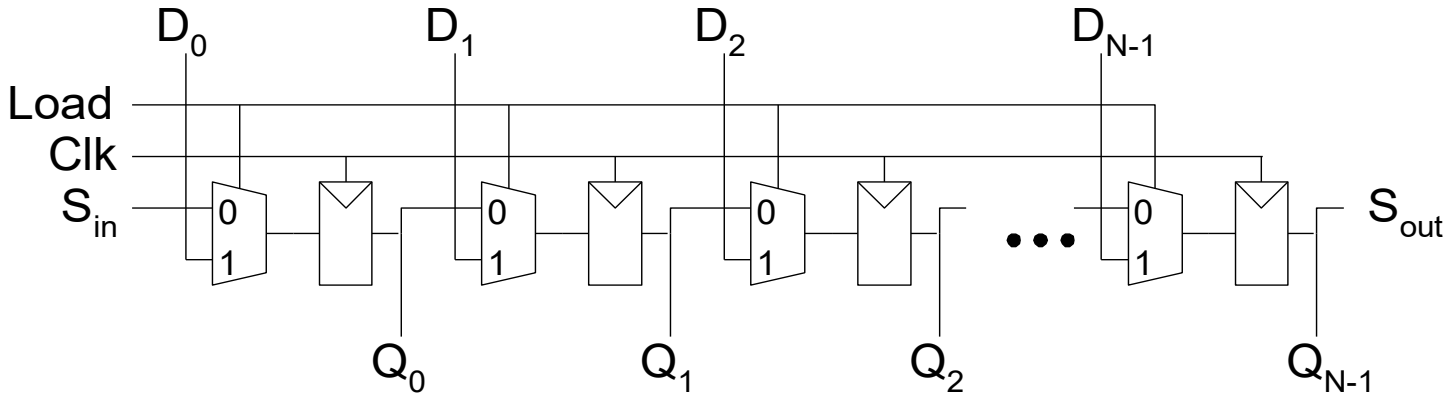



**Implementation:**



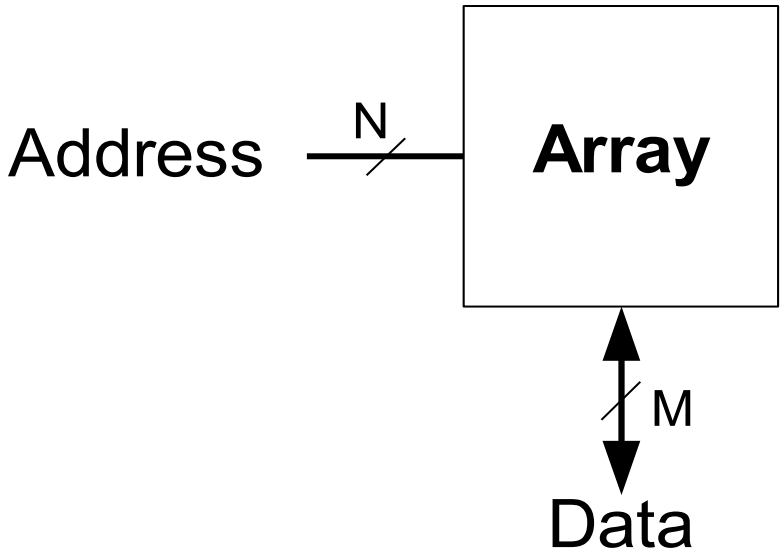
# Shift Register with Parallel Load

- When *Load* = 1, acts as a normal *N*-bit register
- When *Load* = 0, acts as a shift register
- Now can act as a *serial-to-parallel converter* ($S_{in}$ to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to $S_{out}$)



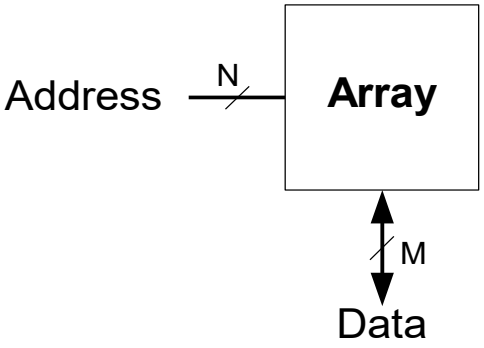
# Memory Arrays

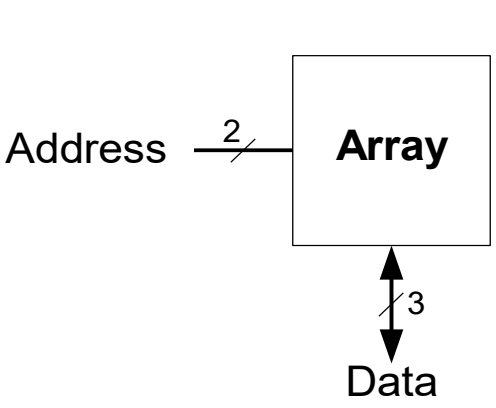
- Efficiently store large amounts of data
- 3 common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- *M*-bit data value read/ written at each unique *N*-bit address

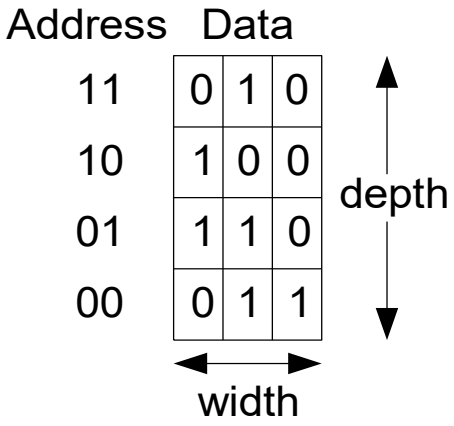


# Memory Arrays

- 2-dimensional array of bit cells
- Each bit cell stores one bit
- $N$ address bits and $M$ data bits:
  - $2^N$ rows and $M$ columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$

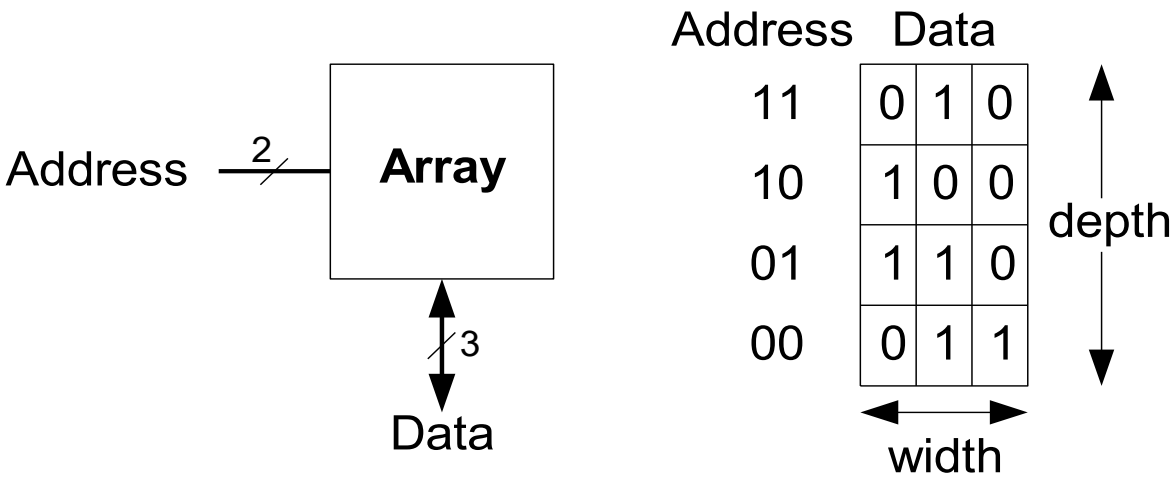


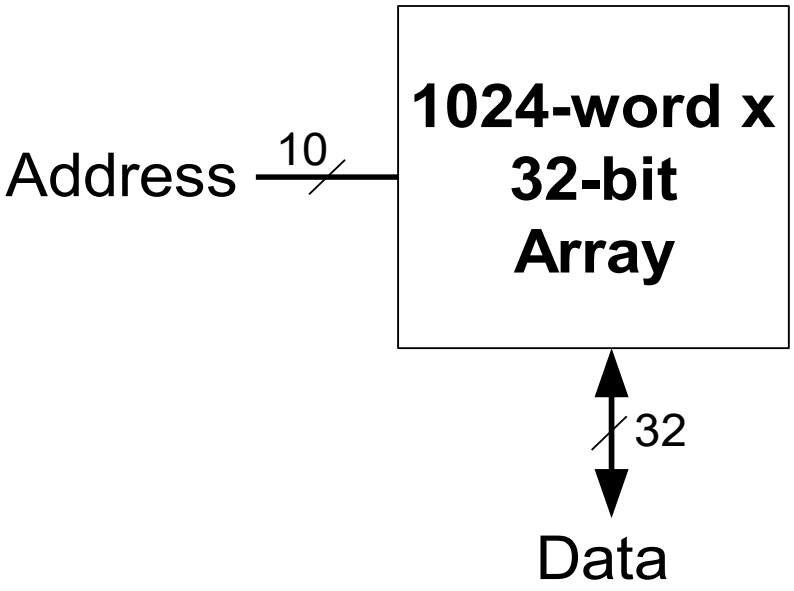



# Memory Array Example

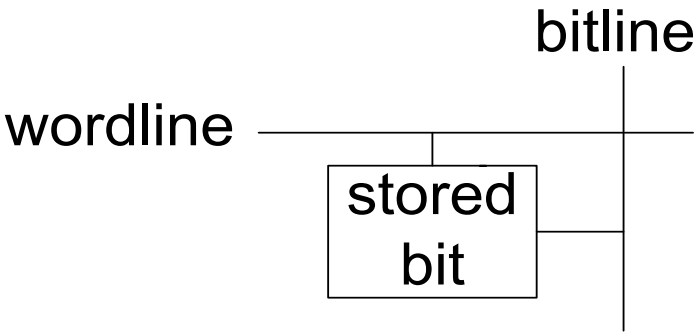
- $2^2 \times 3$-bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

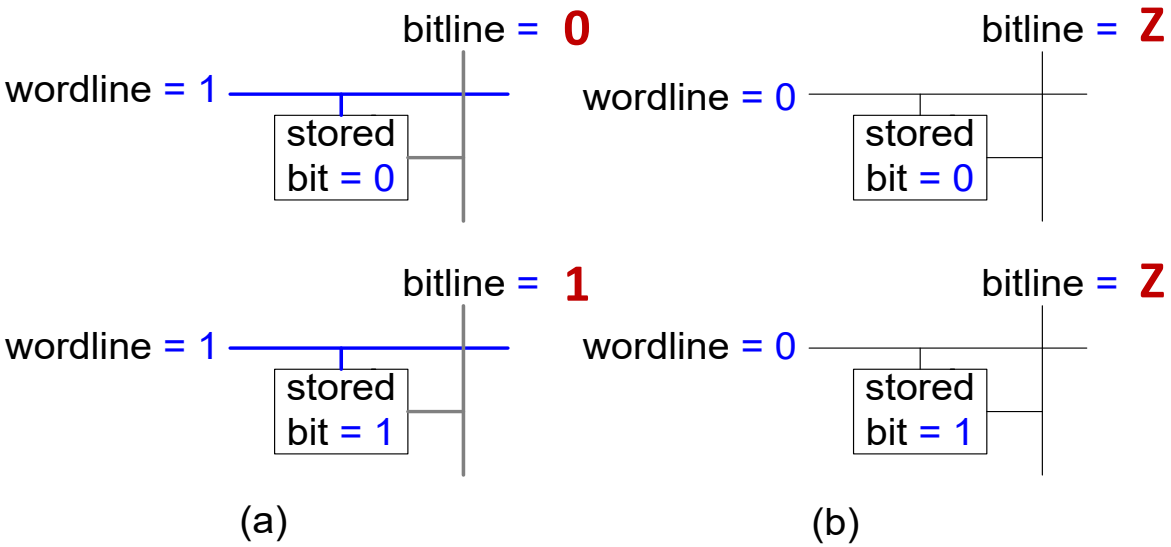


# Memory Arrays



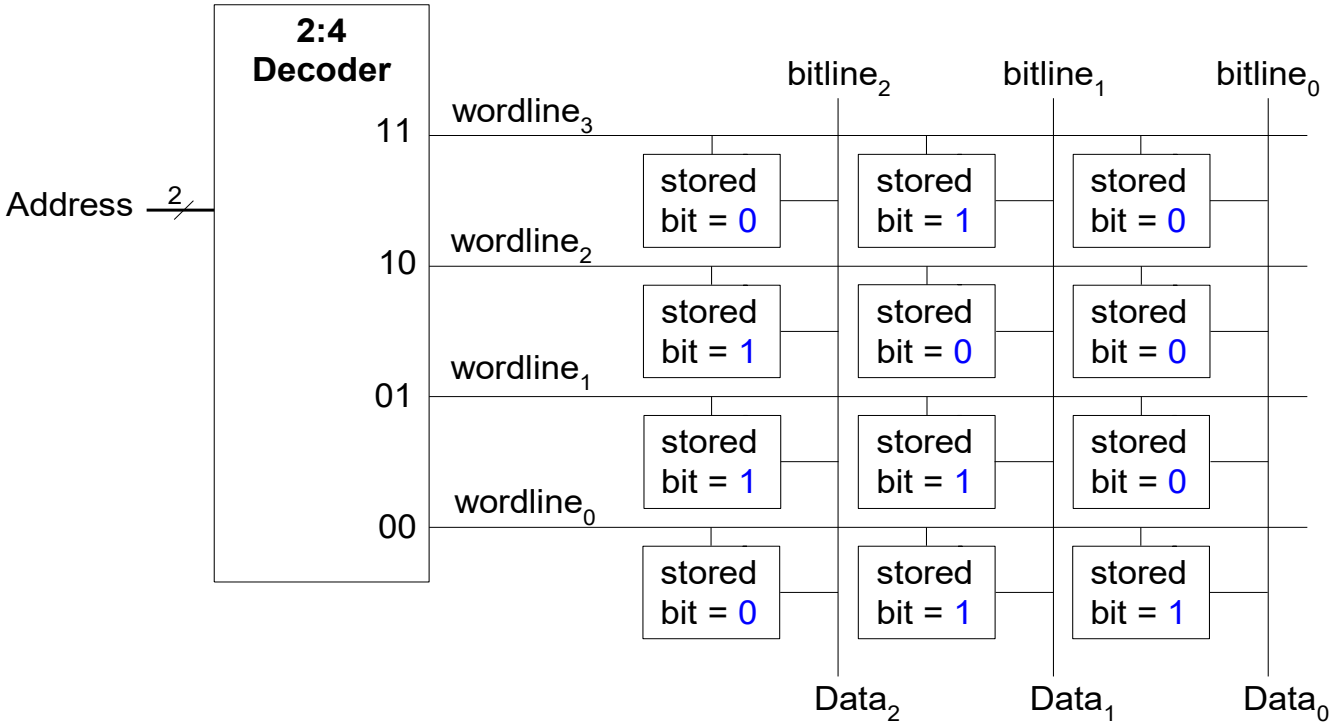
# Memory Array Bit Cells



# Memory Array

- **Wordline:**
  - like an enable
  - single row in memory array read/written
  - corresponds to unique address
  - only one wordline HIGH at once



# Types of Memory

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

# RAM: Random Access Memory

- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)

# ROM: Read Only Memory

- **Nonvolatile:** retains data when power off
- Read quickly, but writing is impossible or slow
- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

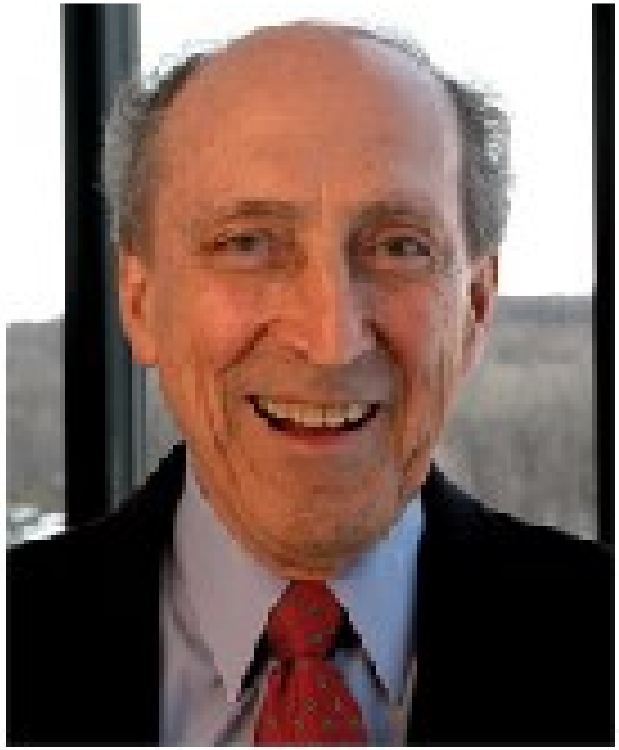
Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

# Types of RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
  - DRAM uses a capacitor
  - SRAM uses cross-coupled inverters

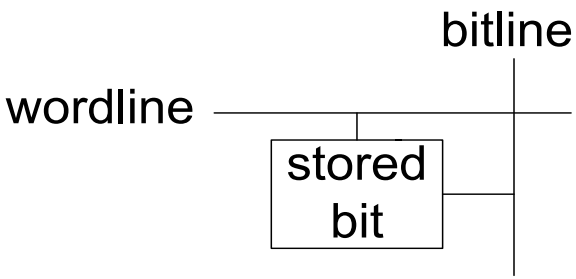
# Robert Dennard, 1932 -

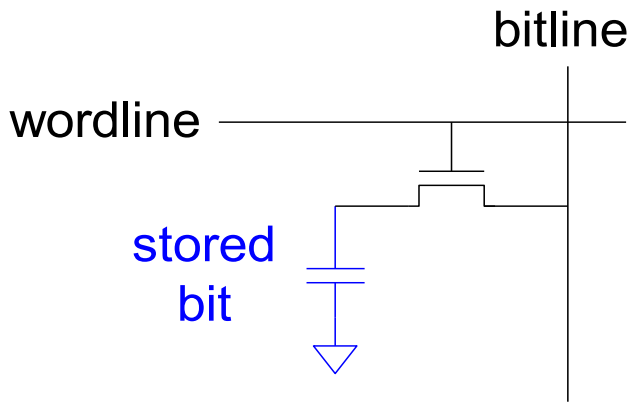
- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers



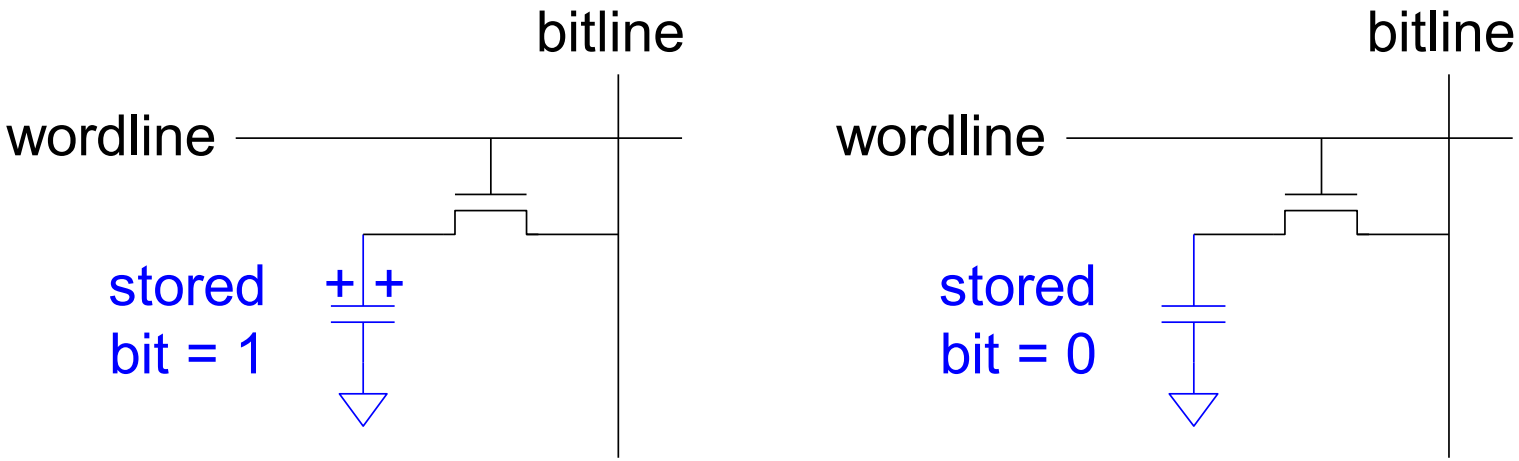
# DRAM

- Data bits stored on capacitor
- *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
  - Charge leakage from the capacitor degrades the value
  - Reading destroys the stored value

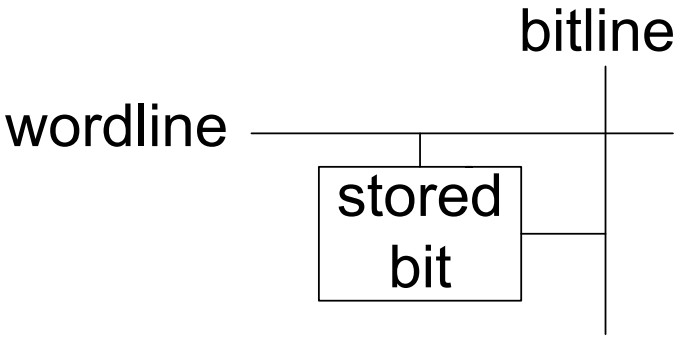


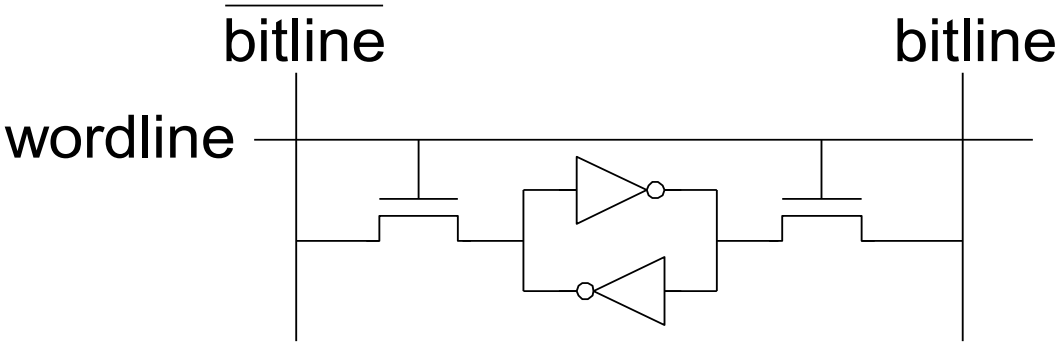



# DRAM

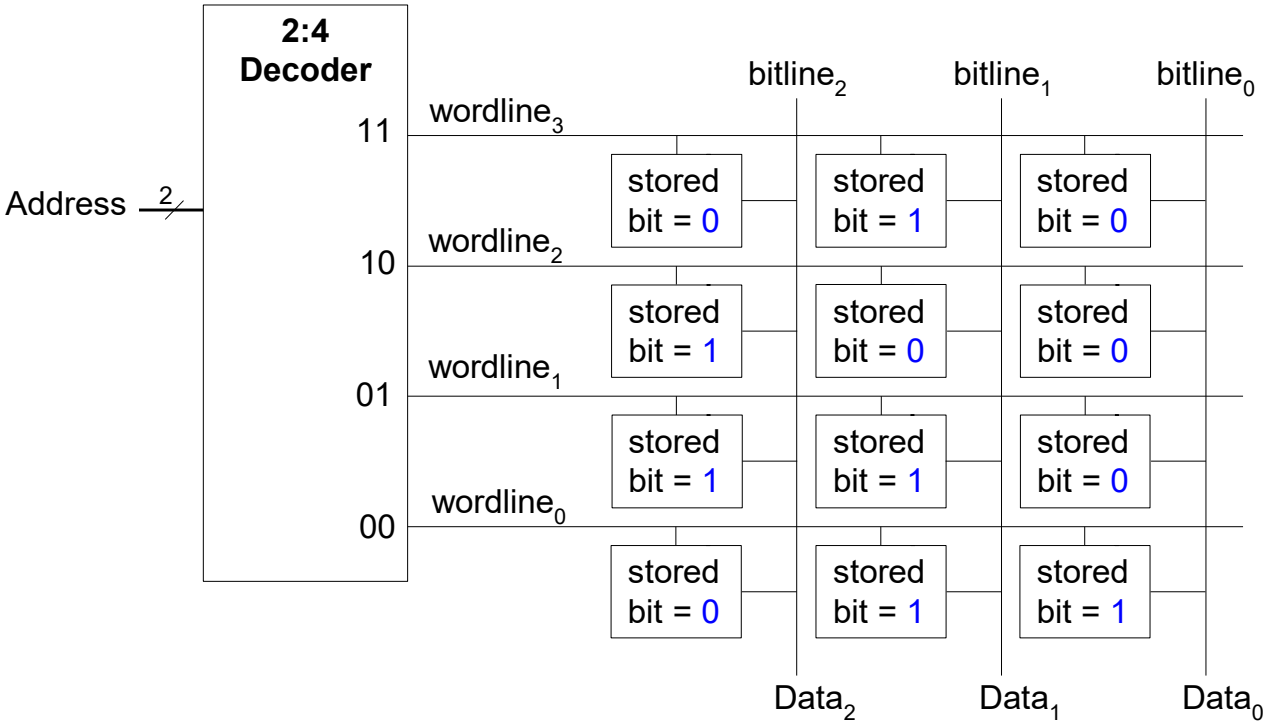


# SRAM

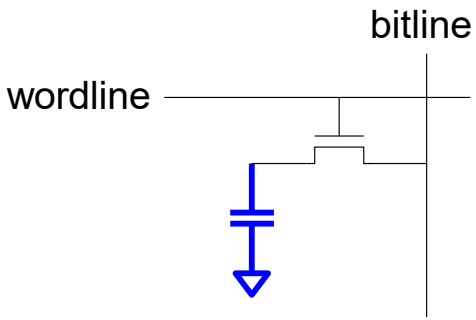


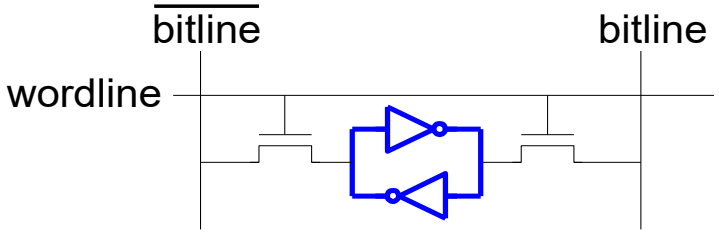
# Memory Arrays Review



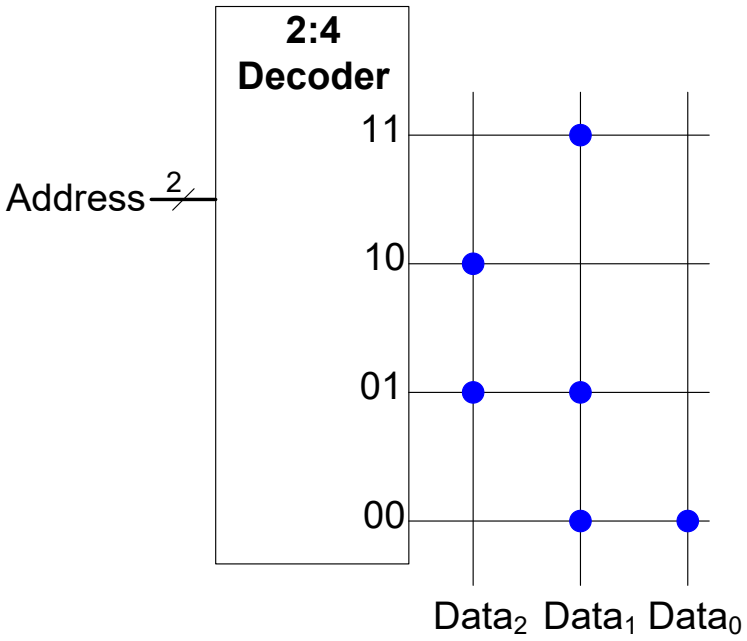

**DRAM bit cell:**

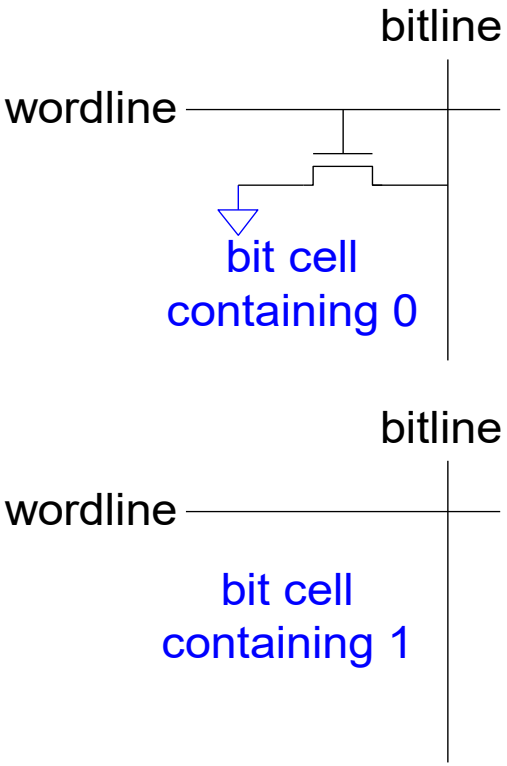



**SRAM bit cell:**



# ROM: Dot Notation

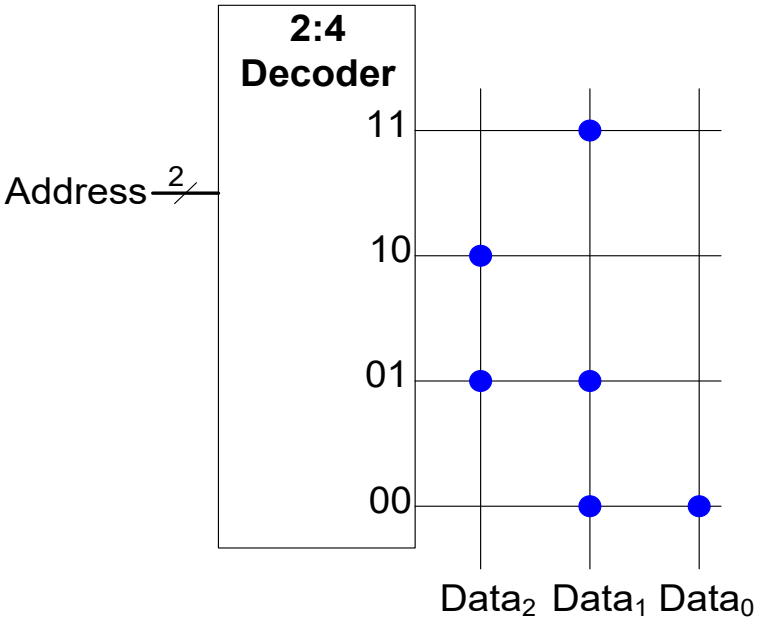


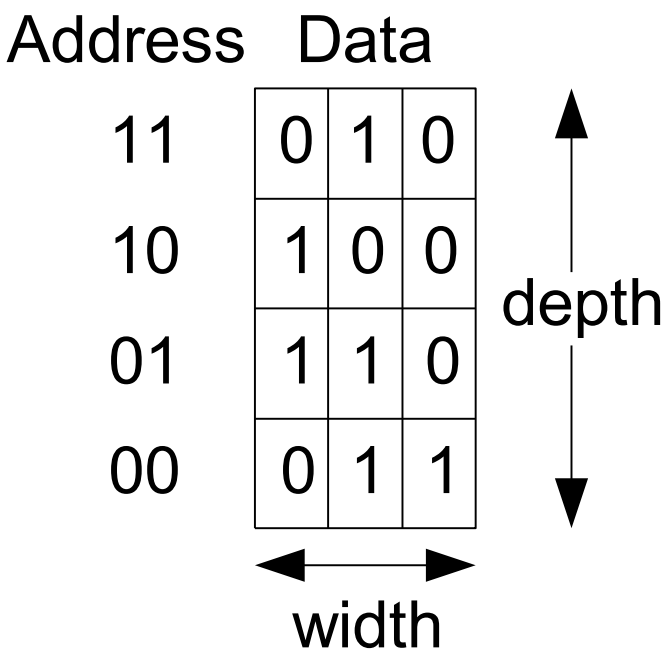



# Fujio Masuoka, 1944 -

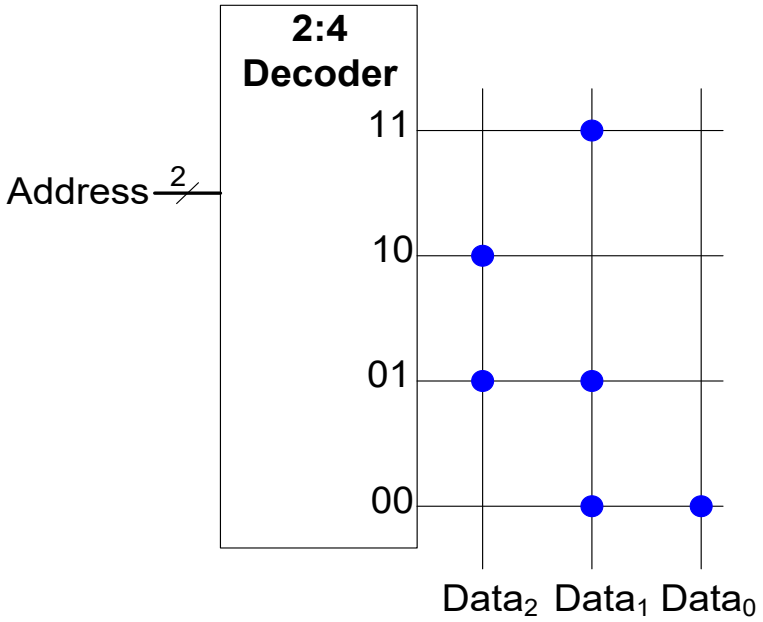
- Developed memories and high speed circuits at Toshiba, 1971-1994
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a $25 billion per year market

# ROM Storage


2:4
Decoder
Address
2
11
10
01
00
Data2 Data1 Data0


Address
Data
11
10
01
00
0 1 0
1 0 0
1 1 0
0 1 1
depth
width

# ROM Logic




$$Data_2 = A_1 \oplus A_0$$

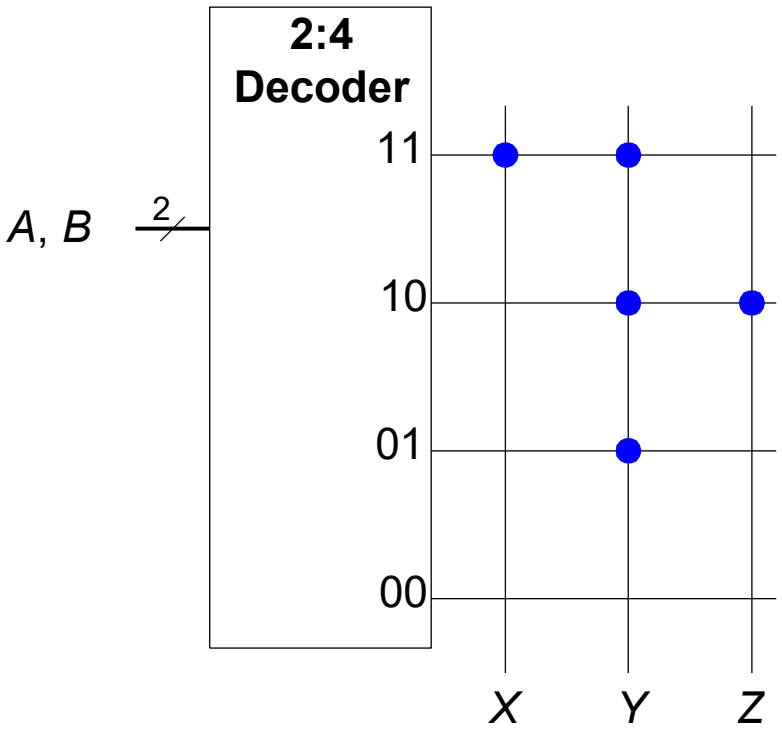
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

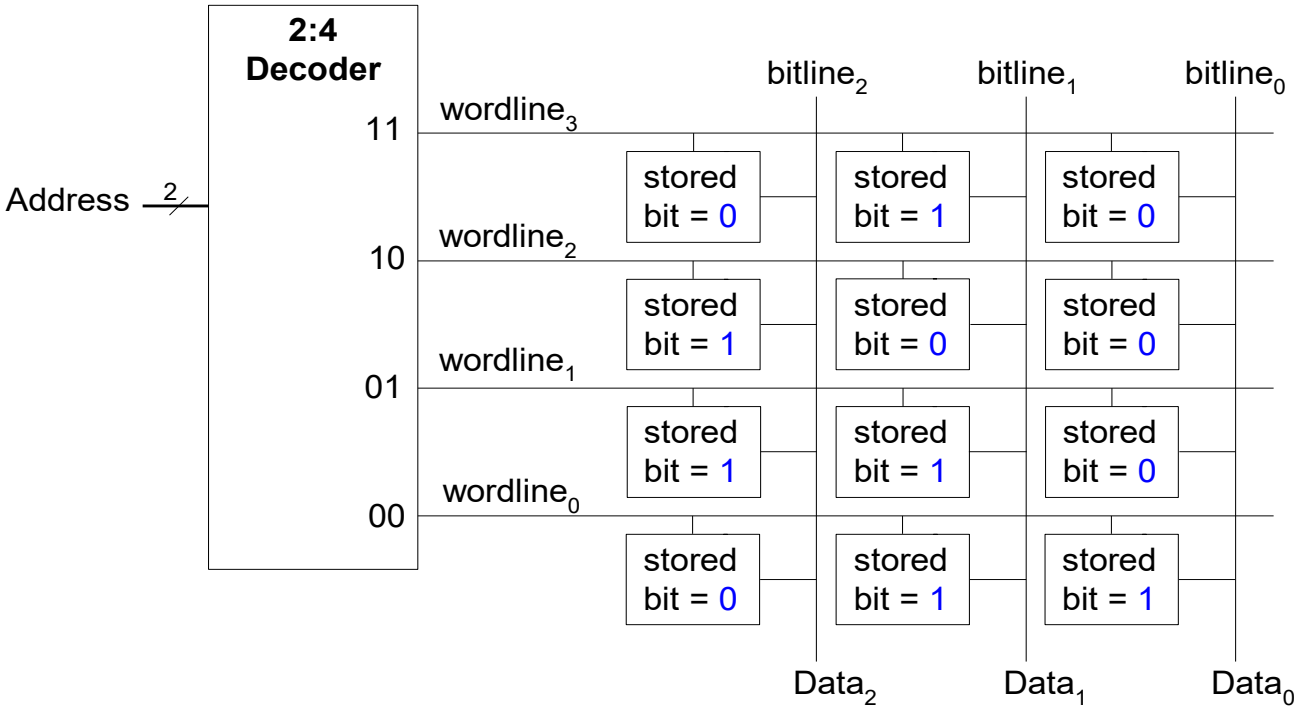
# Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$-bit ROM:

- $X = AB$
- $Y = A + B$
- $Z = A\overline{B}$



# Logic with Any Memory Array




$$Data_2 = A_1 \oplus A_0$$

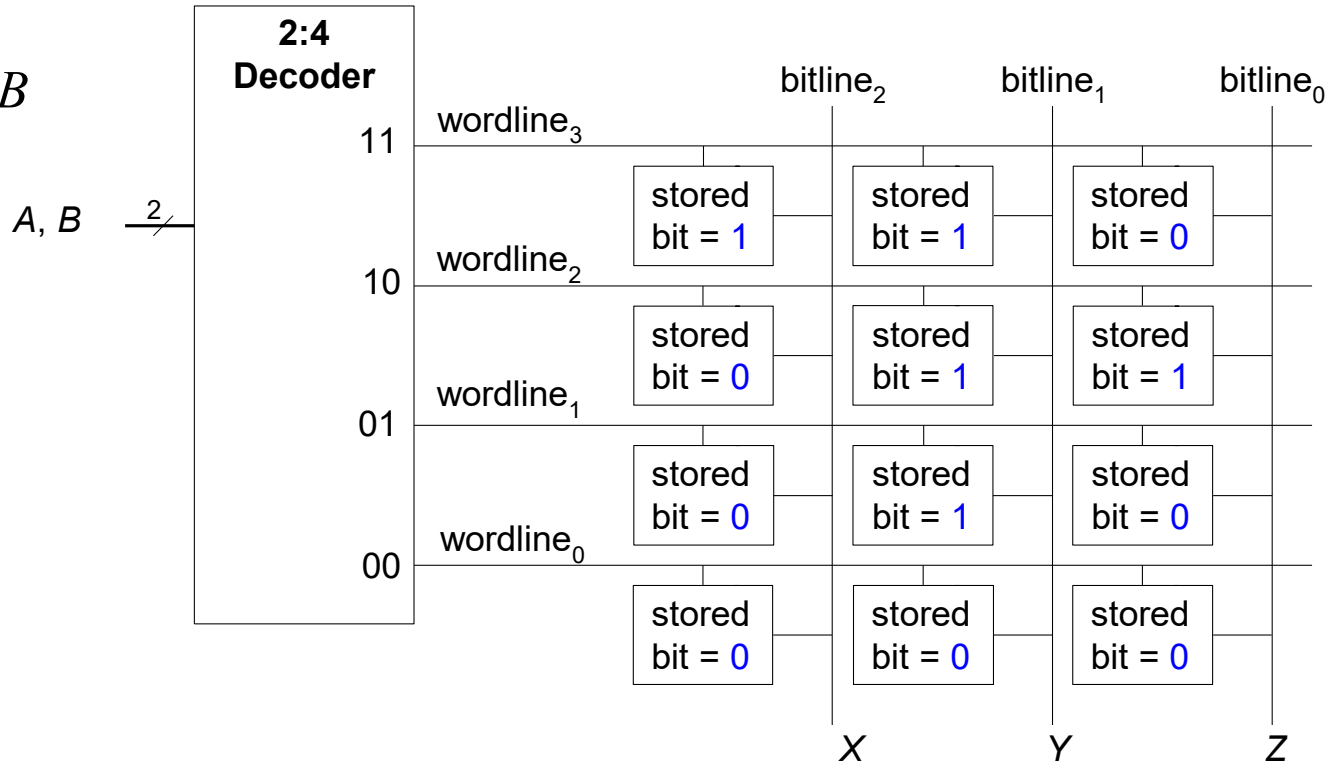
$$Data_1 = \overline{A}_1 + A_0$$

$$Data_0 = \overline{A}_1 \overline{A}_0$$

# Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$-bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = A\overline{B}$

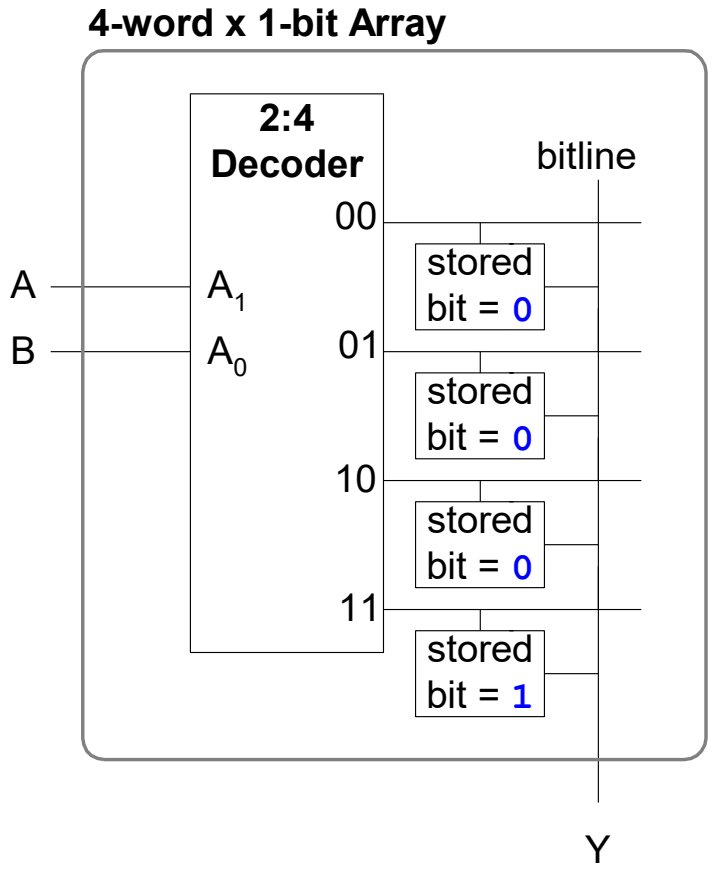


# Logic with Memory Arrays

Called *lookup tables* (LUTs): look up output at each input combination (address)

**Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | **0** |
| 0 | 1 | **0** |
| 1 | 0 | **0** |
| 1 | 1 | **1** |



# Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory