Embedded Memory Organization

Ke Huang
Department of Electrical and Computer Engineering
San Diego State University

Outline

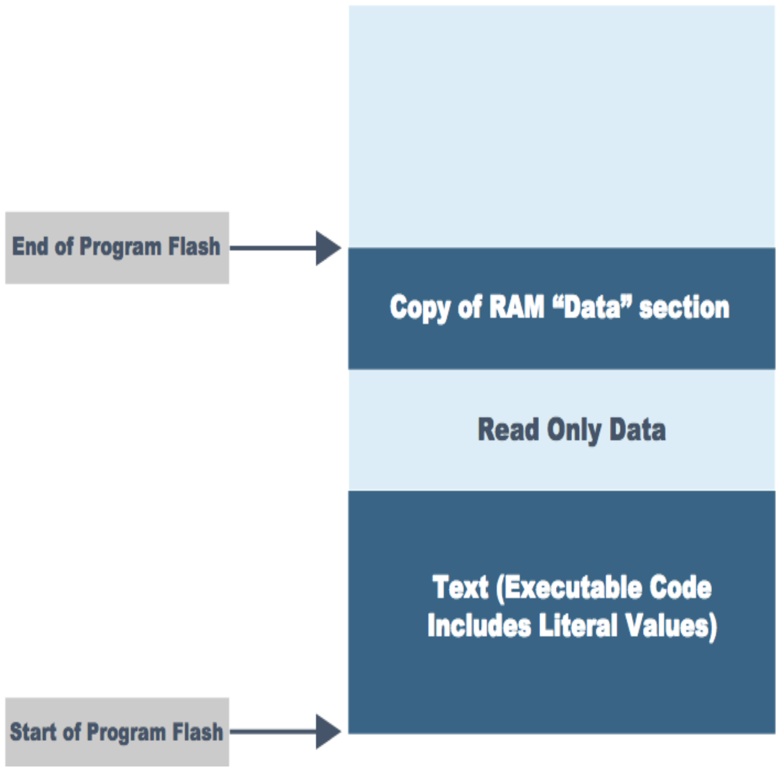
- Memory management
- Memory organization basics
- Examples for different memory types
- Memory timing analysis for read/write
- Example questions

Memory in an Embedded C Program

- Memory in a C program includes code (executable instructions) and data
  - Code is typically read-only and executable
  - Data memory is non-executable, can be either read-only or read-write, and is either statically or dynamically allocated
- In RAM-constrained embedded systems, the memory map is divided in to a section for flash memory (code and read-only data) and a section for RAM (read-write data)

Flash: Code and Read-Only Memory

- Code and read-only data are stored in flash memory
- The beginning of the program (the lowest memory location at the bottom of the diagram) is the text section which includes executable code
- This section also includes numerical values that are not assigned to any specific C variable called "literal values"


End of Program Flash
Copy of RAM "Data" section
Read Only Data
Text (Executable Code Includes Literal Values)
Start of Program Flash

Flash: Code and Read-Only Memory

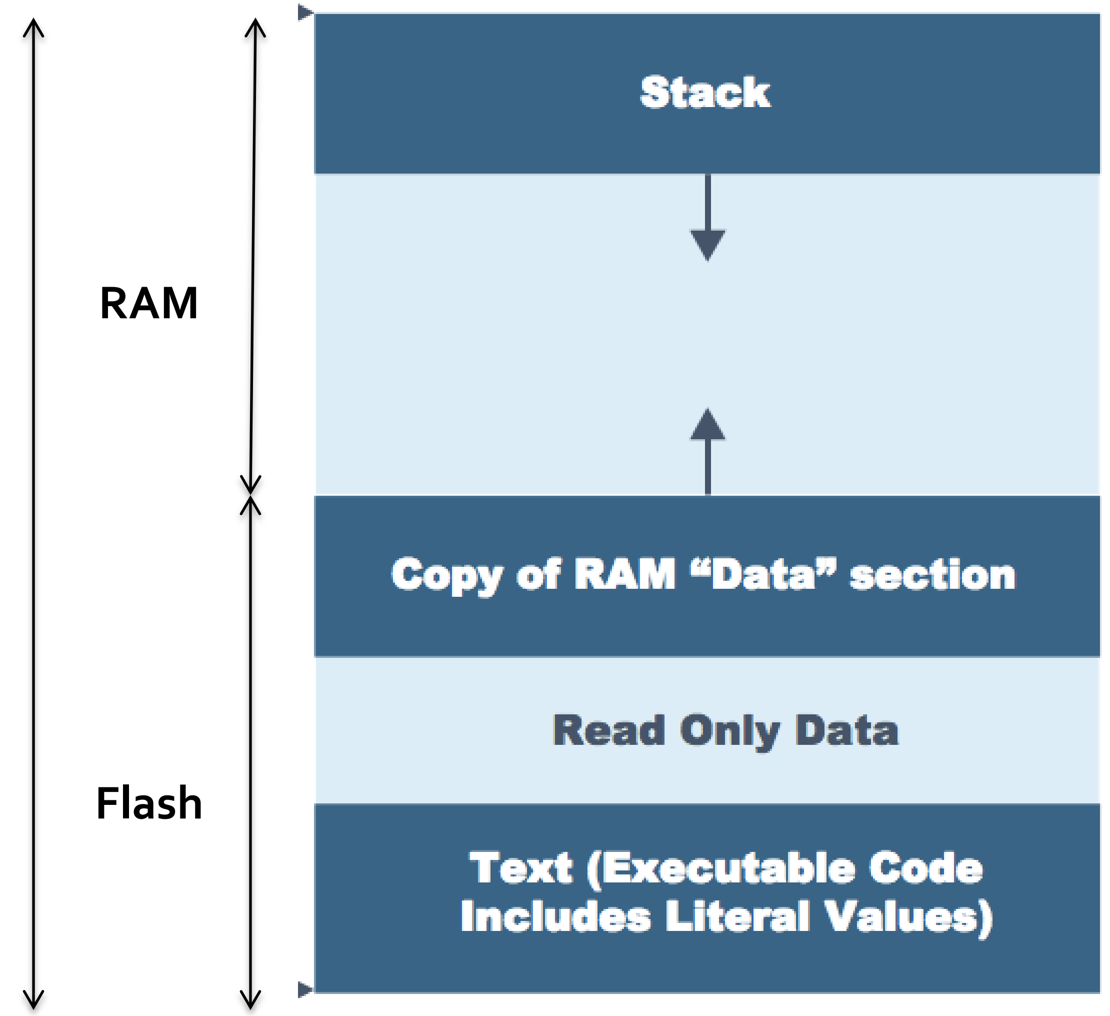
- The "data" section which contains the initial values of global and static variables
  - This section is copied to RAM when the program starts up
- Example:
  - **read_only_variable** is stored in the read-only data section because it is preceded by the **const** keyword
  - The compiler assigns **read_only_variable** a specific address location (in flash) and writes the value of 2000 to that memory location
  - When the variable x within **my_function()** is assigned the literal value 200, it references the value stored in a "literal pool" within the text section
  - A copy of the initial value, 500, assigned to **data_variable** is stored in flash memory and copied to RAM when the program starts

```c
 #include <stdio.h>
const int read_only_variable = 2000;
int data_variable = 500;

void my_function(void){
     int x;
     x = 200;
     printf("X is %d\n", x);
}
```

RAM: Read-Write Data

- After the program starts running



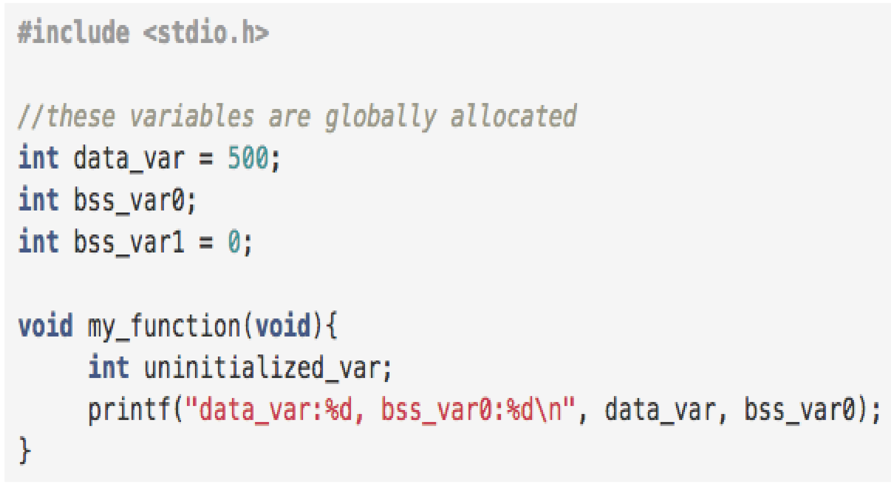
RAM
Flash
Stack
Copy of RAM "Data" section
Read Only Data
Text (Executable Code Includes Literal Values)

Memory requirement of the entire C program

The read-write data that is stored in RAM is further categorized as statically or dynamically allocated

Statically Allocated Data

- Statically allocated memory means that the compiler determines the memory address of the variable at compile time
- Static data is divided in two sections: data and bss
  - Data is assigned an initial, non-zero value when the program starts
  - All variables in the bss section are initialized to zero
- When the program starts:
- C runtime (CRT) start function loads the memory location assigned to **data_var** with 500
  - Copy the value from Flash to RAM
- The CRT start function then sets the memory locations for bss_var0 and bss_var1 to zero which does not require any space in flash memory



```c
#include <stdio.h>

//these variables are globally allocated
int data_var = 500;
int bss_var0;
int bss_var1 = 0;

void my_function(void){
     int uninitialized_var;
     printf("data_var:%d, bss_var0:%d\n", data_var, bss_var0);
}
```

Dynamically Allocated Data

- The locations of dynamically allocated variables are determined while the program is running
- Heap vs. Stack
- The stack grows down (from higher memory address to lower ones) and the heap grows up
- If memory usage is ignored in the design, the stack and heap can collide causing one or both to become corrupted
- The heap is managed by the programmer while the compiler takes care of the stack

Stack vs. Heap

- **Stack**: a "LIFO" (last in, first out) data structure,. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are popped off (deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

- Memory is managed by the compiler => no need to manually allocate memory

- Local in nature => only exist while the function is running. A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited).

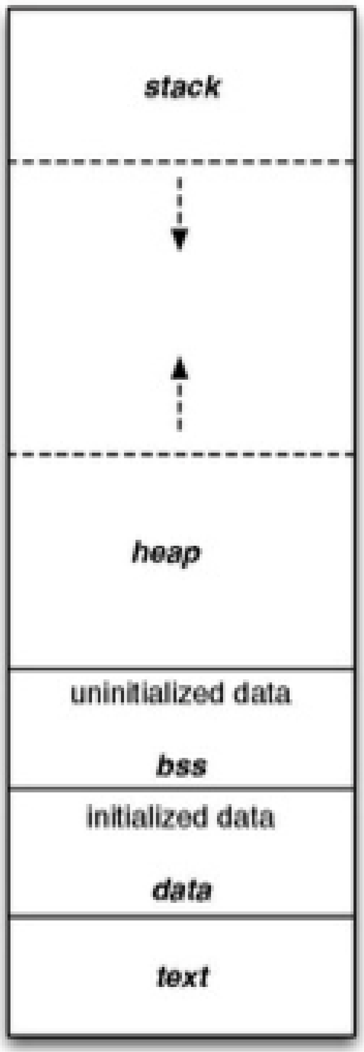
- Limit on the size of variables that can be stored on the stack. Not the case for variables allocated on the heap.

Stack vs. Heap

- **Heap**: Memory region not automatically managed by the compiler, but by the programmer.
- Use built-in functions malloc() or calloc() to allocate memory. Use free() to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a memory leak (a failure in a program to release discarded memory, causing impaired performance or failure).
- No size restrictions on variable size. Slightly slower since one has to use pointers to access memory on the heap.
- Variables created on the heap are accessible by any function, anywhere in the program. Heap variables are essentially global in scope.

Stack vs. Heap

- The beginning of the heap is just above the last bss variable
- To manage the heap: malloc() and free()
- Variables that are declared within a function, known as local variables, are either allocated on the stack or simply assigned a register value
- Whether a variable is allocated on the stack or assigned to a register depends on many factors
  - Compiler, the microcontroller architecture, number of variables already assigned to registers, etc.



C Code Example for Heap

- Heap is used if one needs to allocate a large block of memory (e.g. a large array), and one needs to keep that variable around a long time (like a global).
- Stack is used when dealing with small variables that only need to persist as long as the function using them is alive.

```c
double *multiplyByTwo (double *input) {
  double *twice = malloc(sizeof(double));
  *twice = *input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n", *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

Outline

- Memory management
- **Memory organization basics**
- Examples for different memory types
- Memory timing analysis for read/write
- Example questions

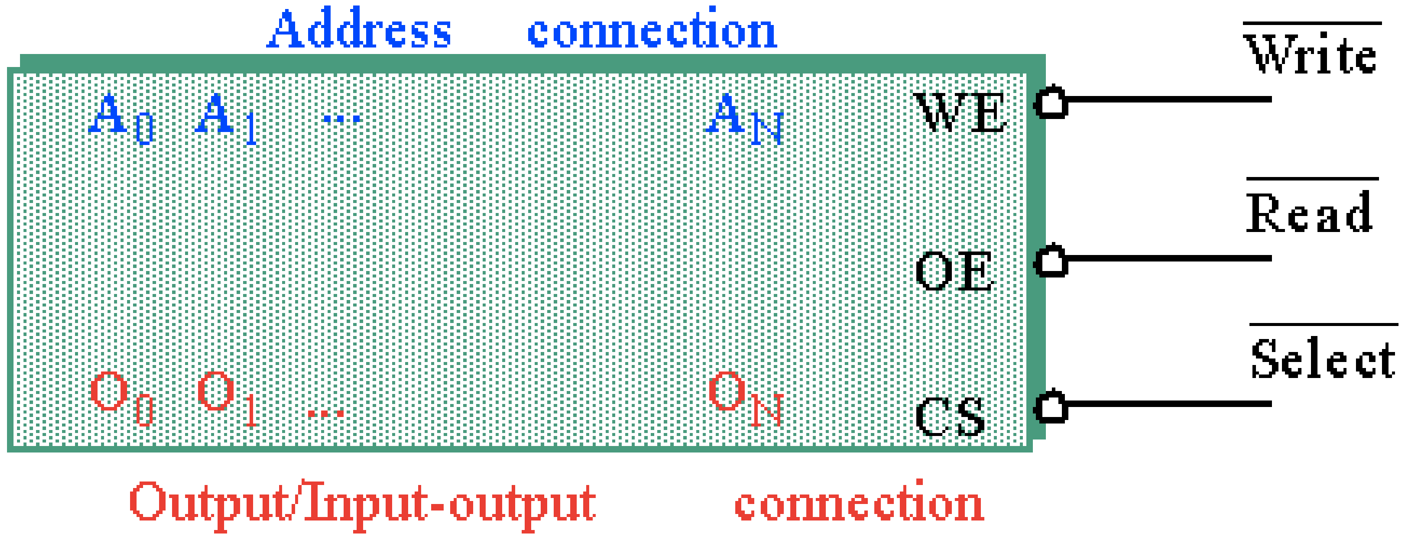
Memory Organization

- N locations by M bits per location:
  - For n address bits, $N = 2^n$ locations
  - Ex: 20 address lines, 8 data bits per location
    - 20 address bits = 2^{20} = 1M locations
    - 8 bits per location = 1M x 8 = 1MB (1 megabyte)
  - K = 2^{10} = 1,024
  - M = 2^{20} = 1,048,576
  - b=bit, B=Byte (8 bits)

Memory Types

- In the previous lecture, we saw that there are two basic types of memory:
  - Read-only memory (ROM)
  - Read-write memory (RAM)
- Four commonly used memory types
  - ROM
  - Flash and EEPROM
  - Static RAM (SRAM)
  - Dynamic RAM (DRAM)

Generic Pin Configuration



- Each memory device has at least one chip select (CS) or chip enable (CE) or select (S) pin that enables the memory device
  - This enables read and/or write operations

ource: Plusquellic, Systems Design and Programming

Memory Chips

- The number of address pins is related to the number of memory locations
  - Example sizes are 1K to 256M locations (depending on the device)
  - 1K → 10 pins, 256M → 28 pins
- The data pins are typically bi-directional in read-write memories
  - The number of data pins is related to the size of the memory location
  - For example, an 8-bit wide (byte-wide) memory device has 8 data pins
  - Catalog listing of 1K X 8 indicate a byte addressable 8K memory

Memory Chips

- Each memory device has at least one control pin
- For ROMs, an output enable (OE) or gate (G) is present
  - The OE pin enables and disables a set of tri-state buffers
- For RAMs, a read-write (R/W) or write enable (WE) and read enable (OE) are present
  - For dual control pin devices, it must be hold true that both are not 0 at the same time

Outline

- Memory management
- Memory organization basics
- **Examples for different memory types**
- Memory timing analysis for read/write
- Example questions

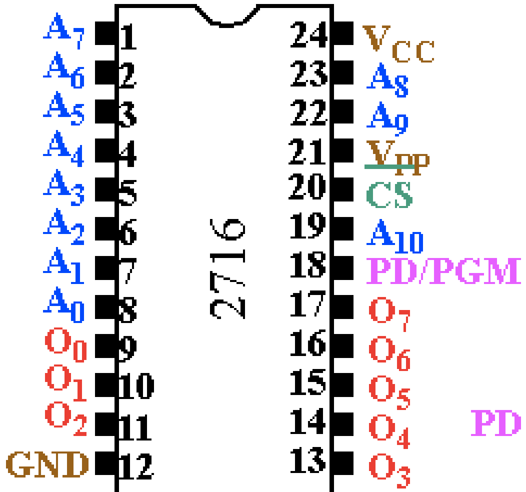
Example ROM Types

- Non-volatile memory: Maintains its state when powered down
- **ROM** : Factory programmed, cannot be changed. Older style
- **PROM** : Programmable Read-Only Memory
  - Field programmable but only once. Older style
- **EPROM** : Erasable Programmable Read-Only Memory
  - Reprogramming requires up to 20 minutes of high-intensity UV light exposure
- **EEPROM** : Electrically Erasable Programmable ROM
  - Also called EAROM (Electrically Alterable ROM) and NOVRAM (NOn-Volatile RAM)
  - Writing is much slower than a normal RAM
  - Used to store setup information, e.g. video card, on computer systems

Source: Plusquellic, Systems Design and Programming

Example EPROM Organization

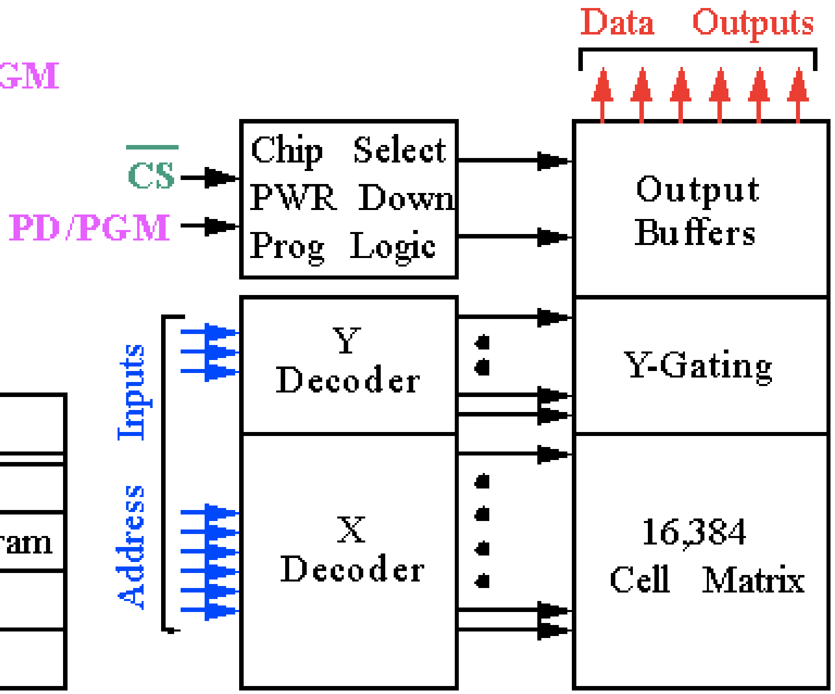
- Intel 2716 EPROM (2K X 8)



2K x 8 EPROM

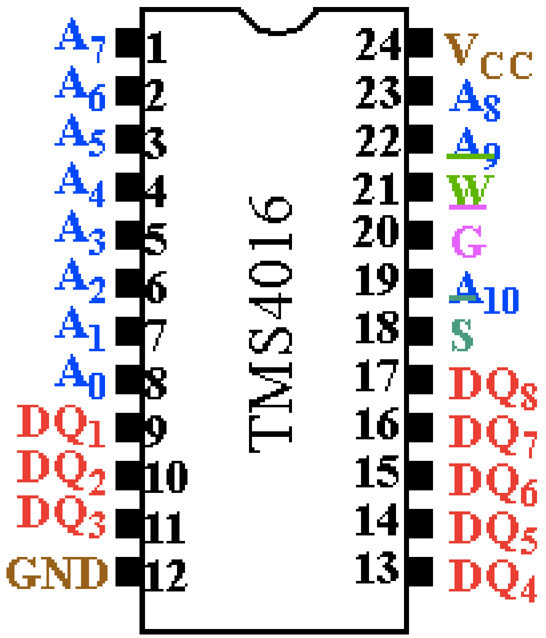
Pin(s)	Function
A_0-A_{10}	Address
PD/PGM	Power down/Program
$\overline{CS}$	Chip Select
O_0-O_7	Outputs

V_{PP} is used to program the device by applying 25V and pulsing PGM while holding $\overline{CS}$ high.



Example SRAM Organization

- TI TMS 4016 SRAM (2K X 8):



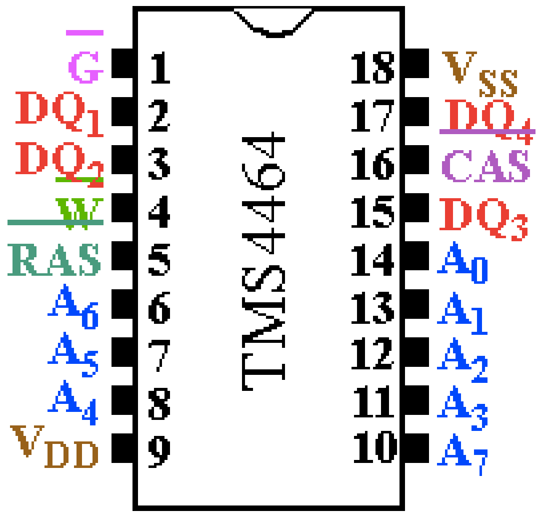
2K x 8 SRAM

Pin(s)	Function
A_0-A_{10}	Address
DQ_0-DQ_7	Data In/Data Out
$\overline{S}$ ($\overline{CS}$)	Chip Select
$\overline{G}$ ($\overline{OE}$)	Read Enable
$\overline{W}$ ($\overline{WE}$)	Write Enable

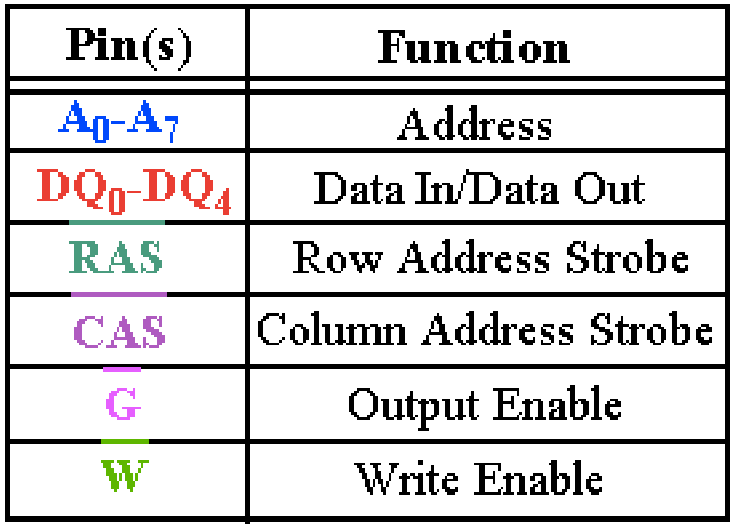
Source: Plusquellic, Systems Design and Programming

Example DRAM Organization

- TI TMS4464 DRAM (64K X 4):



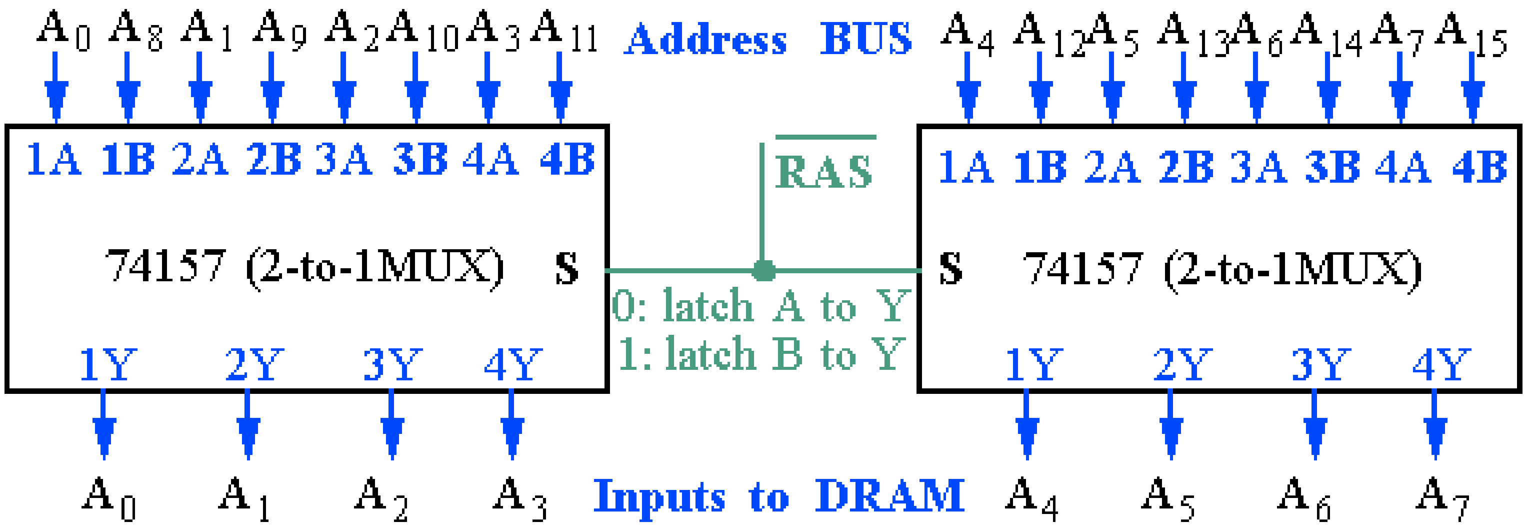
64K x 4 DRAM



Pin(s)	Function
A_0-A_7	Address
DQ_0-DQ_4	Data In/Data Out
$\overline{RAS}$	Row Address Strobe
$\overline{CAS}$	Column Address Strobe
$\overline{G}$	Output Enable
$\overline{W}$	Write Enable

Example DRAM Organization

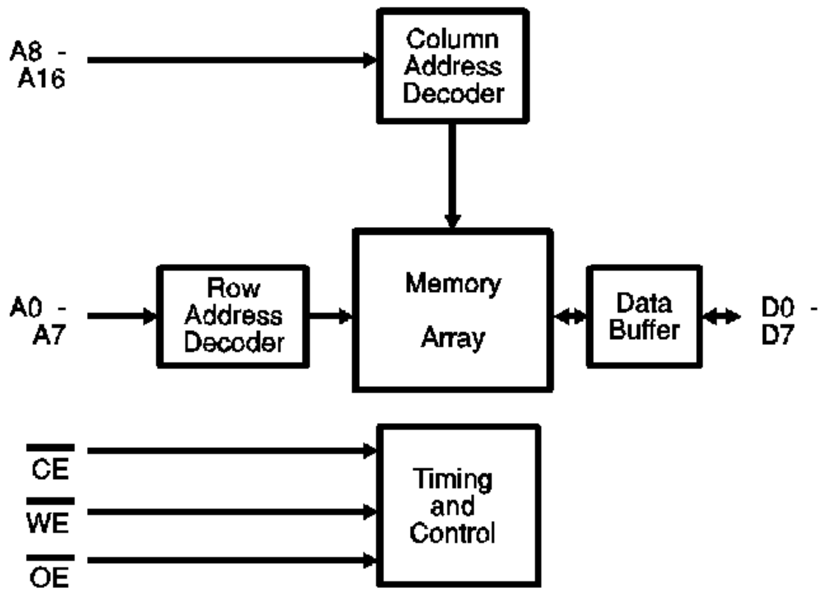
- There are 64K addressable locations which means it needs 16 address inputs, but it has only 8
- The row address (A 0 through A 7) are placed on the address pins and strobed into a set of internal latches
- The column address (A 8 through A 15) is then strobed in using CAS
- This is achieved by multiplexing:



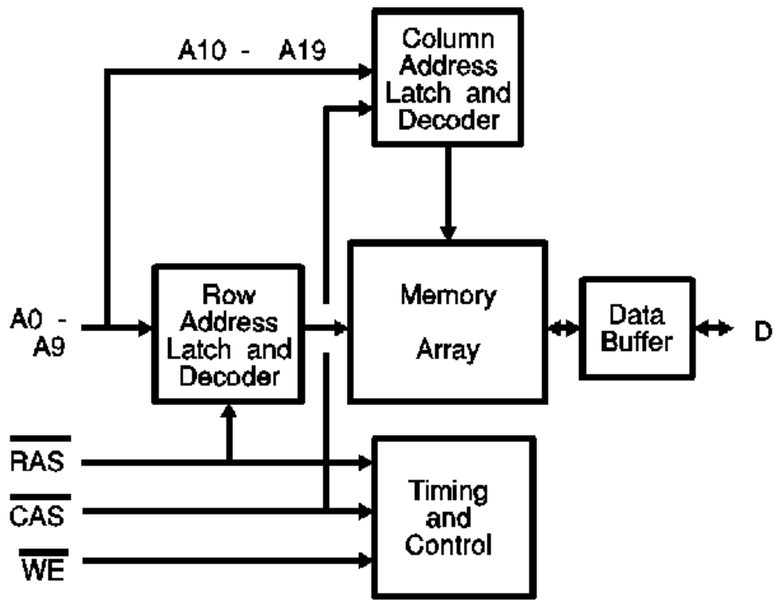
Source: Plusquellic, Systems Design and Programming

Examples

1 Megabit Memories: SRAM and DRAM



128K x 8 SRAM

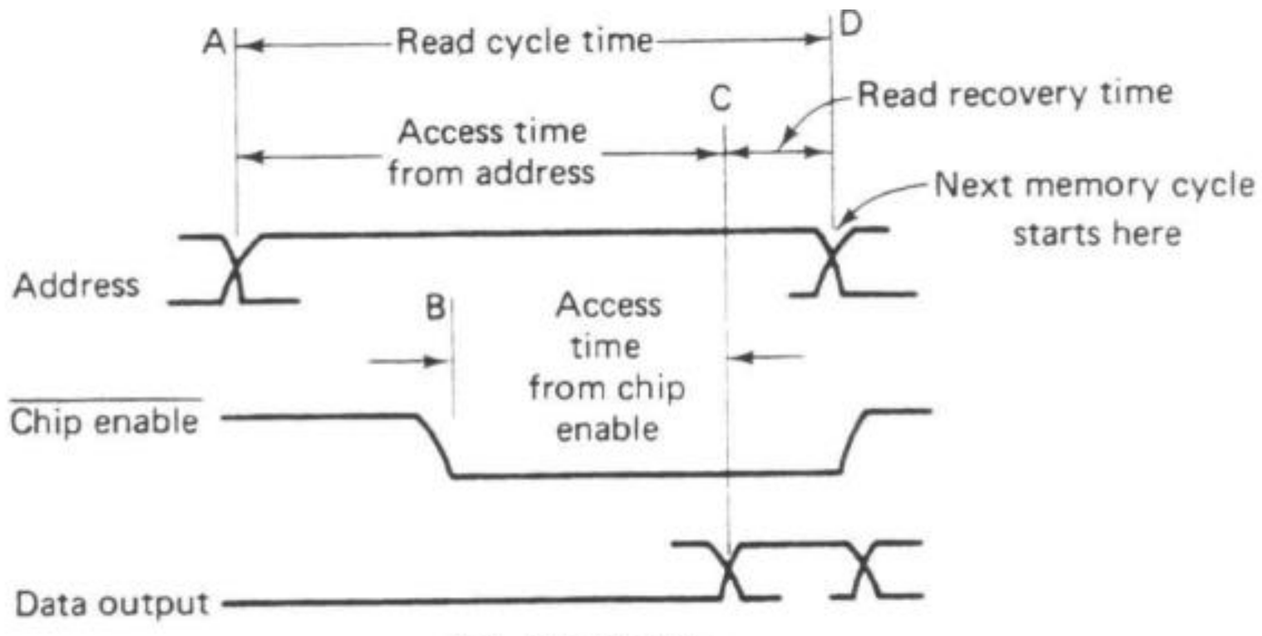


1M x 1 DRAM

Outline

- Memory management
- Memory organization basics
- Examples for different memory types
- **Memory timing analysis for read/write**
- Example questions

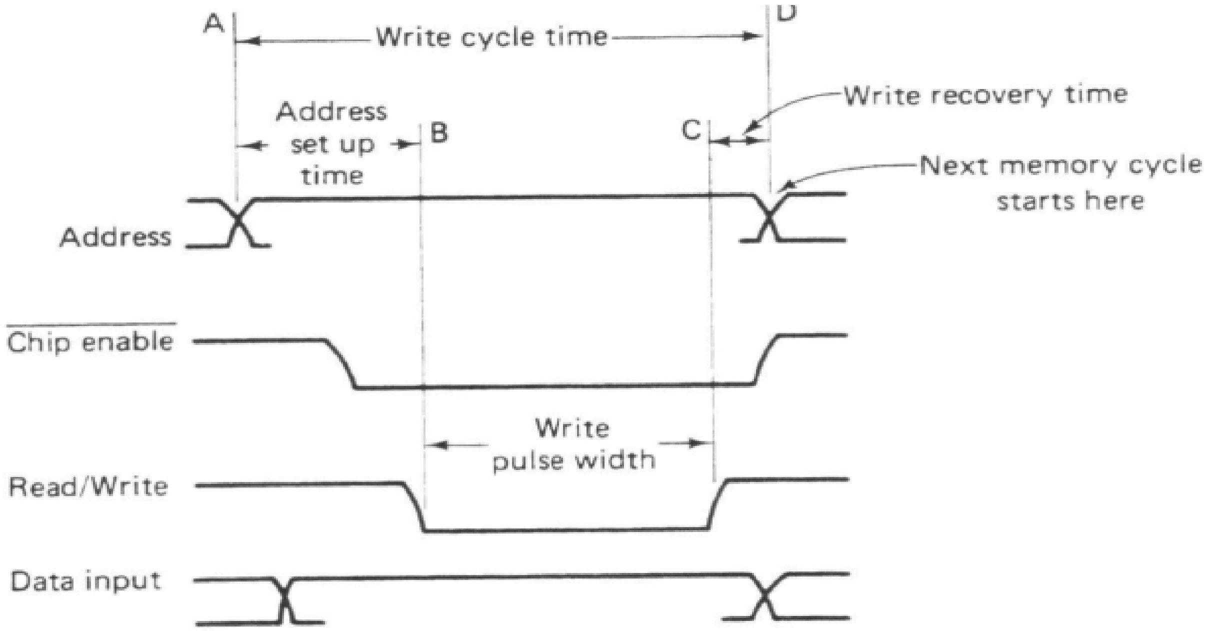
Memory Read Timing



(a) Read cycle.

- Once the output data are valid, the address input cannot be changed immediately to start another read operation. This is because the device needs a certain amount of time, called read recovery time, to complete its internal operations before the next memory operation.
- The sum of the access time and read recovery time is the memory read cycle time. This is the time needed between the start of a read operation and the start of the next memory cycle.

Memory Write Timing



(b) Write cycle.

- In addition to the address and chip enable inputs, an active low write pulse on the R/W line and the data to be stored must be applied during the write cycle.
- The timing of data input is less restrictive and can be satisfied simply by holding the data input stable during the entire cycle. However, the application of the write pulse has two critical timing parameters:
  - The address setup time and the write pulse width
- The address setup time is the time required for the address to stabilize and is the time that must elapse before the write pulse can be applied

Memory Errors

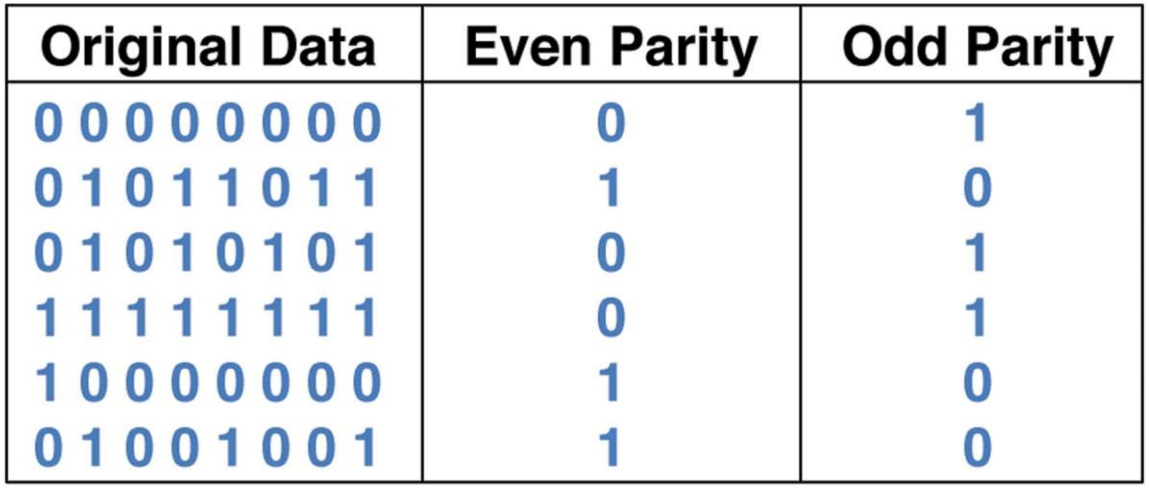
- Error detection
  - Parity, checksum, CRC (Cyclic Redundancy)
- Error correction
  - Hamming codes, Block error correcting codes
- Soft error
  - A transient error, that does not repeat
- Hard error
  - An error that is permanent, reproducible

Memory Error Checking

- Parity checks:
  - Used to detect single bit errors in the memory
- Odd parity
  - Parity bit maintains an odd # of 1’s in word
  - Even number of ones indicates an error
- Even parity
  - Parity bit maintains an even # of 1’s in word
  - Odd number of ones indicates an error

Example: Parity Checking

- In an n-bit system:
  - Parity checking adds 1 bit for every n data bits
  - For EVEN parity, the $(n+1)$-th bit is set to yield an even number of 1's in all $(n+1)$-bits
  - For ODD parity, the $(n+1)$-th bit is set to make this number odd
- Example even vs. odd parity checker:



Original Data	Even Parity	Odd Parity
0 0 0 0 0 0 0 0	0	1
0 1 0 1 1 0 1 1	1	0
0 1 0 1 0 1 0 1	0	1
1 1 1 1 1 1 1 1	0	1
1 0 0 0 0 0 0 0	1	0
0 1 0 0 1 0 0 1	1	0

Block Parity

- Horizontal and vertical parity bits
  - One parity bit for each row of bits
  - One parity bit for each column of bits
- Single bit error will be detected
  - Row and column error indicates location
  - Complementing the bit fixes the error

Example Using Even Parity

- data: Horizontal (even) parity:
  - 1 0 1 1 p=1 odd horizontal parity
  - 1 1 1 1 p=0 even horizontal parity
  - 1 0 0 1 p=0 even horizontal parity
  - 1 0 1 1 p=1 odd horizontal parity
  - 0 1 1 0 <- The vertical parity for these 4 words

Cyclic Redundancy Code (CRC)

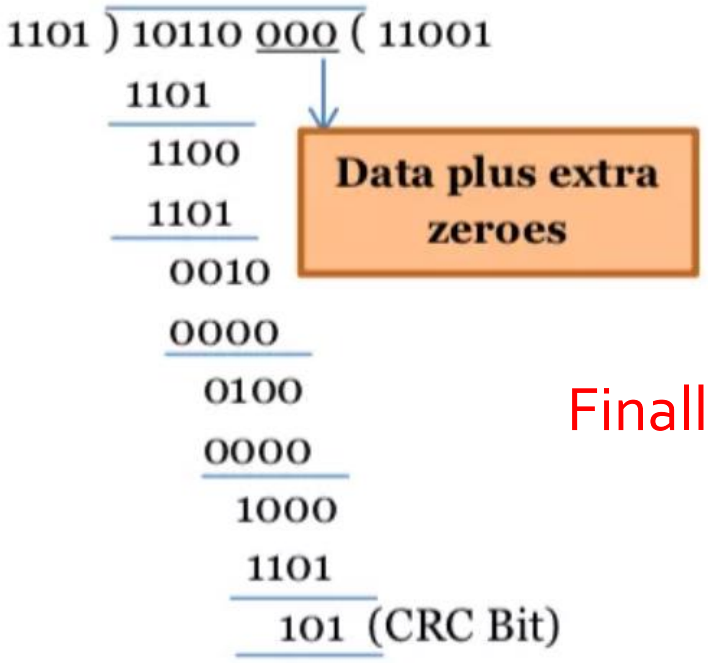
- CRC is based on a binary polynomial
- Can be calculated using only shift & XOR
- Better than checksum:
  - Detects data in wrong order or byte swapped
  - Detects missing or extra zeros in a block
- Standardized polynomials, e.g. CCITT-16
- Standardized initial value of polynomial

Cyclic Redundancy Code (CRC)

- At Sender Side
  - Sender has a generator $G(x)$ polynomial.
  - Sender appends $(n - 1)$ zero bits to the data.
    Where $n =$ # of bits in generator
  - Dividend appends the data with generator $G(x)$
    using modulo 2 division (arithmetic).
  - Remainder of $(n - 1)$ bits will be CRC.
- Codeword: It is combined form of Data bits and CRC bits, i.e.
  Codeword = Data bits + CRC bits

Cyclic Redundancy Code (CRC)

- Assume that
  - Data is 10110.
  - Code generator is 1101 (represented by the polynomial $x^3 + x^2 + 1$).



Finally, the sender sends 10110 110

Cyclic Redundancy Code (CRC)

- At Receiver Side
    - Receiver has same generator $G(x)$.
    - Receiver divides received data (data + CRC) with generator.
    - If remainder is zero, data is correctly received.
    - Else, there is error.

Checksum and Hamming

- Hamming Code
    - Redundant code - extra code bits
    - Detects and corrects single bit errors
- Checksum
    - Sum of all the bytes in a block of memory
    - Use LS 8 or 16 bits of the sum
    - Will not detect errors:
        - Extra/missing zeros, wrong order, swapped bytes

Outline

- Memory management
- Memory organization basics
- Examples for different memory types
- Memory timing analysis for read/write
- **Example questions**

Question 1

- What is the largest byte-wide SRAM that will fit in a 32 pin package?
  - 1M x 8 = 1MB = 8 Mb
  - 512K x 8 = 512KB = 4Mb
  - 256K x 8 = 256KB = 2Mb
  - 128K x 8 = 128KB = 1Mb

Question 2

- What is the maximum number of bits that can be stored in a byte-wide ROM in a 32 pin package?
  - 2M x 8 = 2MB = 16Mb
  - 128K x 8 = 128KB = 1Mb
  - 1M x 8 = 1MB = 8Mb
  - 4M x 8 = 4MB = 32Mb

Question 3

- Access time from address is:
  - The delay from a valid address to output enable active
  - The delay from enable active to valid memory read data
  - The delay from valid address to valid memory read data
  - The address access time when address must be held after the data is read

Question 4

- Using 4Mx4 DRAMs, how many memory chips will be required to implement a 16 MB memory organized in 32 bit words?
  - 4
  - 8
  - 16
  - 32

Question 5

- A CRC code is:
  - An error detection and correction code
  - A checksum that detects data out of order
  - A block error detection code, that detects data which are out of order
  - A block error detection code which detects missing zeros and data which are out of order

Question 6

- For a DRAM, what is the largest memory that can be packaged in a 24 pin package?
  - 32 Mb
  - 16 Gb
  - 64 Gb
  - 4 Gb