EE310 Embedded Computer Systems

Lecture 8: Memory Architectures

Dr. Suneel Kommuri

Suneel.Kommuri@xjtlu.edu.cn

Outline

- ✓ **Memory Technologies**
  - Random Access Memory
  - Non-Volatile Memory
- ✓ **Memory Hierarchy**
  - Memory Maps
  - Register Files
  - Caches
- ✓ **Memory Models**
  - Memory Addresses
  - Stacks
  - Protection Units
  - DMA
  - Memory Model of C

Introduction

Memory systems have a significant impact on overall system performance.

- There are three main sources of complexity in memory

1. It is usually necessary to mix a variety of memory technologies in the same embedded system.
   1. Some volatile and some non-volatile memory is required.
2. Memory hierarchy is needed.
   1. Memories with larger capacity or lower power consumption are slower
   2. To achieve good performance there needs to be a mix of fast and slow memories.
3. The address space of a processor architecture is divided up to provide access to various kinds of memory.
   1. To provide support for common programming models
   2. To designate addresses for interaction with non-memory devices.

1.0 Memory Technologies

- ✓ Volatile Memory
  - RAM
- ✓ Non-Volatile Memory
  - ROM
  - Flash memory

1.1 Random Access Memory (RAM)

- ✓ Memory where individual bytes or words can be written and read one at a time relatively quickly.
- ✓ Two types: Static RAM (SRAM) and Dynamic RAM (DRAM)
- ✓ SRAM is faster than DRAM, but it is larger
- ✓ DRAM holds data for only a short time and must be periodically refreshed
- ✓ SRAM holds data for as long as the power is maintained.
- ✓ Both are volatile (lose their contents if power is lost)

RAM and Embedded Systems

- ✓ Most Embedded systems include an SRAM memory. Many also include DRAM because it can be impractical to provide enough memory with SRAM technology alone.

- ✓ A programmer concerned with execution times should be aware of whether memory addresses being accessed are mapped to SRAM or DRAM (because DRAM is more volatile).

- ✓ DRAM can introduce variability to access times because it may be busy refreshing at the time access is requested.

  – it may depend on what memory address was last accessed

1.2 Non-volatile Memory

- Non-volatile memories are required as data has to be stored when power is turned off.
- Read Only Memory (basic non-volatile memory today!)
  - Contents fixed during manufacture.
  - Useful for mass produced products that only need to have a program and constant data stored (i.e. Data never changes). Such programs are called firmware.
- EEPROM (electrically-erasable programmable ROM)
  - EEPROM comes in several forms but it is possible to write to all of them. The write time is very slow (much slower than the read time) and the total number of writes during the lifetime of the device is limited. (Flash memory $\longrightarrow$ Next slide)

Flash Memory

- ✓ Flash memory is commonly used to store firmware and user data that needs to be maintained when power is removed.
- ✓ Two types: NOR and NAND flash memories
- ✓ NOR has longer erase and write times – accessed like RAM
- ✓ NAND is less expensive, has faster erase and write times
- ✓ Both types can be erased and written bounded number of times
- ✓ Disk memory also non-volatile – store very large amounts of data

2.0 Memory Hierarchy

- ✓ Many applications require more memory than is available on-chip µC
- ✓ Memory hierarchy – combines various memory technologies
  - to increase overall memory capacity while optimizing cost, energy consumption
  - small amount of on-chip SRAM + large amount of off-chip DRAM + disk drives
- ✓ **Virtual memory** makes technologies look to compiler *address space*
- ✓ The OS/hardware provides *address translation*
  - Converts logical addresses in the address space to physical locations in one of memory technologies
- ✓ These techniques can create *serious problems*
  - because they make very difficult to predict how long memory accesses will take

2.1 Memory Maps

- ✓ A memory map for a processor defines how addresses are mapped to hardware.

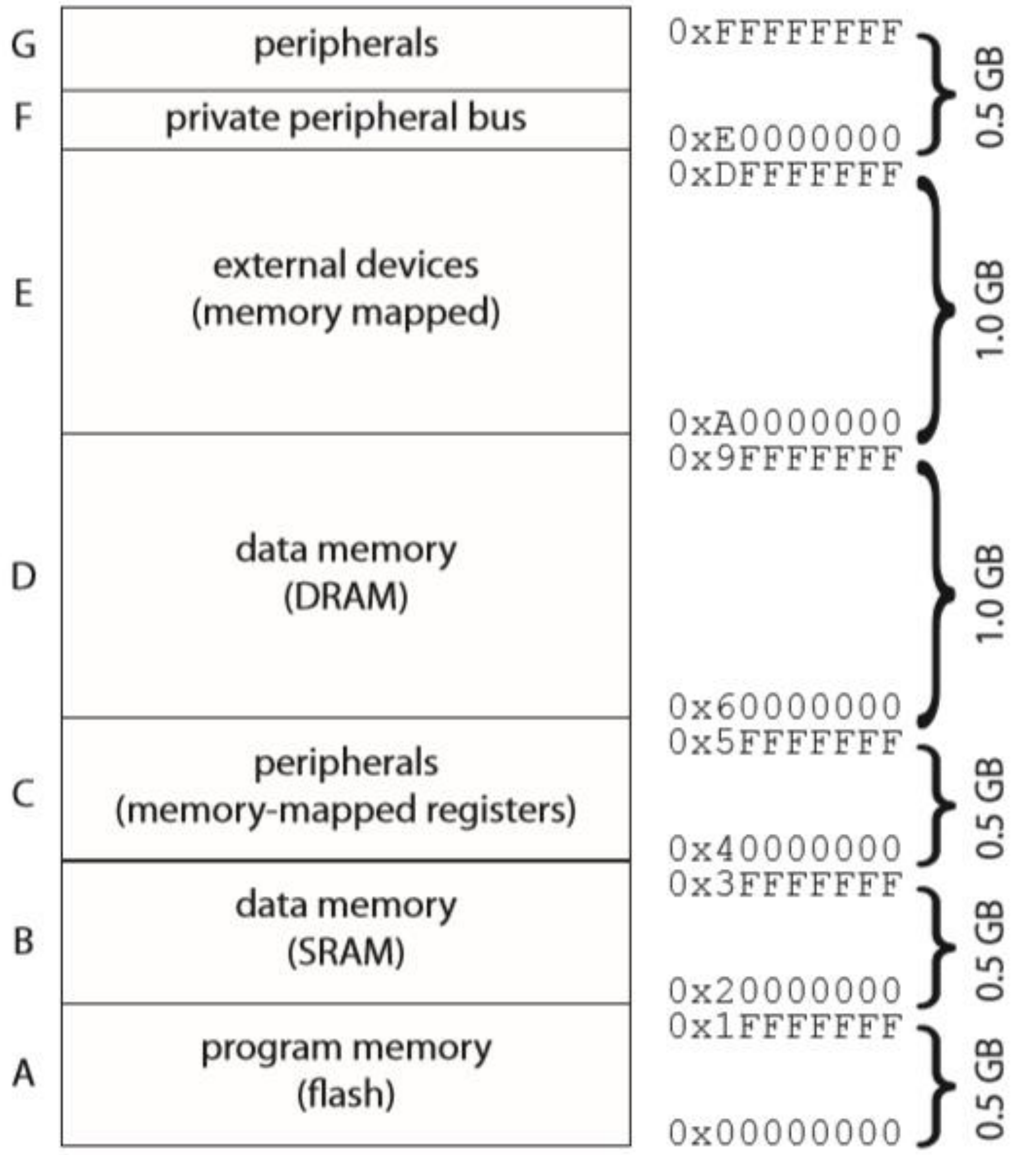
- ✓ Total size of the address space is constrained by width of processor

- ✓ How many memory locations can be addressed by a 32-bit proc.?

  Can address 2^{32} locations, or 4 GB

- ✓ The address width matches the word width, except 8-bit processors – where the address width is typically higher (often 16 bits)

Memory map of an ARM Cortex^{TM}



There are separate addresses used for program memory (A) from those used for data memory (B and D).
This allows these memories to be accessed via separate buses, permitting data and instructions to be fetched simultaneously. This effectively doubles the memory bandwidth.
This is known as **Harvard architecture.**
Classical **von Neumann architecture**, stores program and data in the same memory.

2.2 Register Files

- ✓ The register file is tightly integrated memory in a processor.
- ✓ Each register stores a **word**.
  - The size of the word is a key property of a processor architecture (four bytes for a 32-bit architecture, 8 bytes on 64-bit).
- ✓ The number of registers in a processor is usually small.
  - This is to do with the bits in an instruction word.
- ✓ If the register file has 16 registers, each reference to register requires 4 bits

2.3 Scratchpads and Caches

- ✓ As we have discussed many embedded applications will use a mix of memory technologies.
- ✓ Some memories are accessed before others (close memory).
- ✓ If the close memory (SRAM) has a distinct set of addresses, and the program is responsible for moving data into it or out to the distant memory, then it is termed a **scratchpad.**
- ✓ If the close memory duplicates data in the distant memory with the hardware handling all the copying, it is called a **cache.**
- ✓ Cache is more normal, but presents obstacles for embedded applications, because their timing behavior can vary unpredictably.

Increasing bandwidth using a cache

- ✓ Typically programs run using only small regions of memory for periods that are long compared with the memory access time.
- ✓ We can increase bandwidth using a cache memory.
- ✓ Cache: What does it mean?
  - noun = hiding place (usually for valuables)
  - or verb = to put into a hiding place (from French cacher - to hide)

Cache memory

- ✓ A cache memory keeps a copy of parts of the main memory so that the CPU can access (read or write) that data more quickly.

- ✓ The operation of a cache relies on two features of computer programs:
  - temporal
  - and spatial locality

Temporal / spatial locality

- ✓ Temporal locality occurs because data accessed once is likely to be accessed again soon
  - e.g. an instruction in a program loop

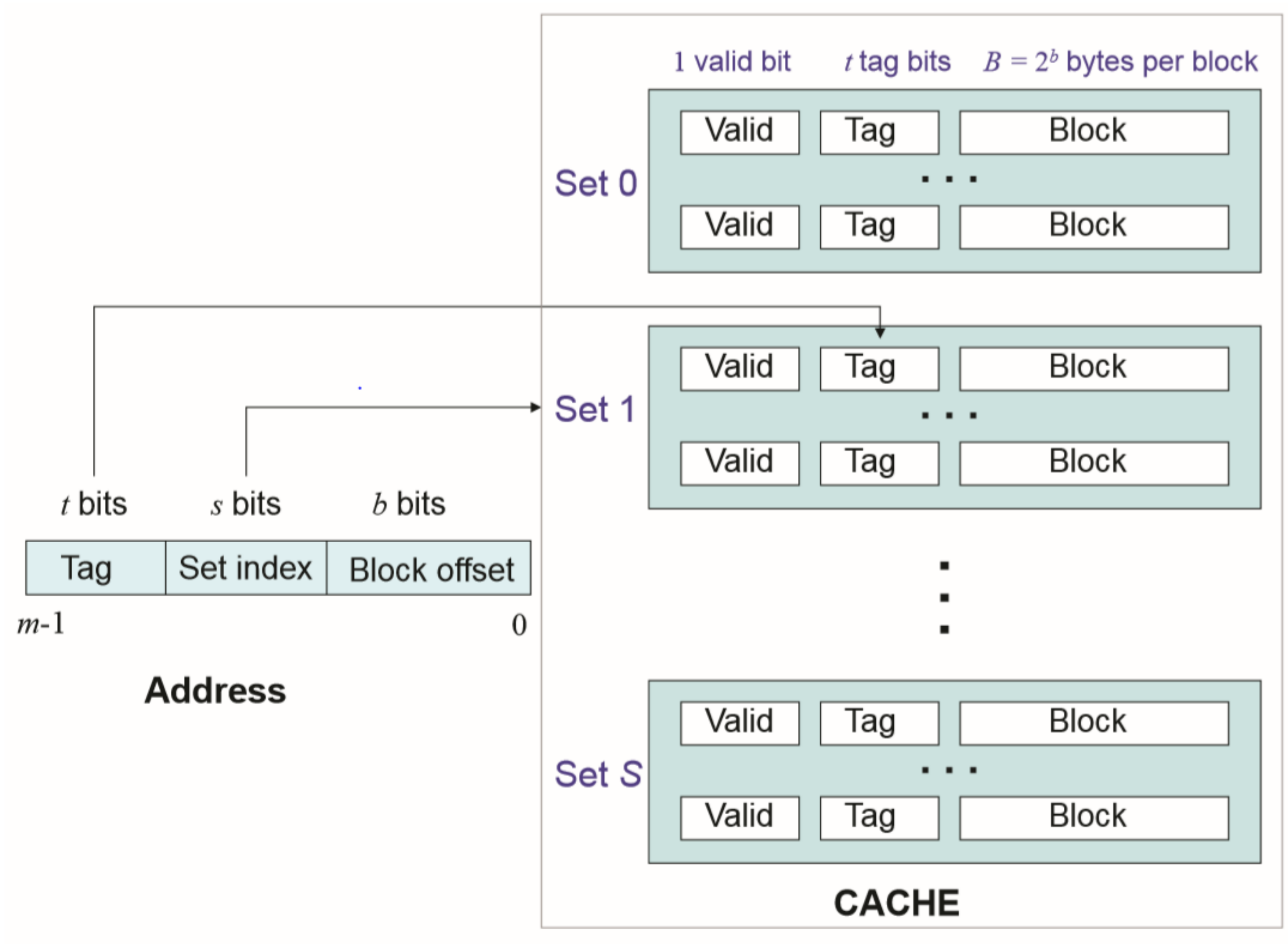
- ✓ Spatial locality occurs because data from one memory location is more likely to be accessed if data in an adjacent memory location has recently been accessed.
  - (Typically the data stored in the cache may be 16 or more adjacent bytes.)

Cache memories

- ✓ Cache memory is
  - small compared with main memory
  - built with devices that are much faster than the ones used in the main memory
  - connected between the main memory and the CPU
  - **hidden** because the CPU operates as if it is manipulating the main memory and programs only see main memory.

Basic Cache Organization

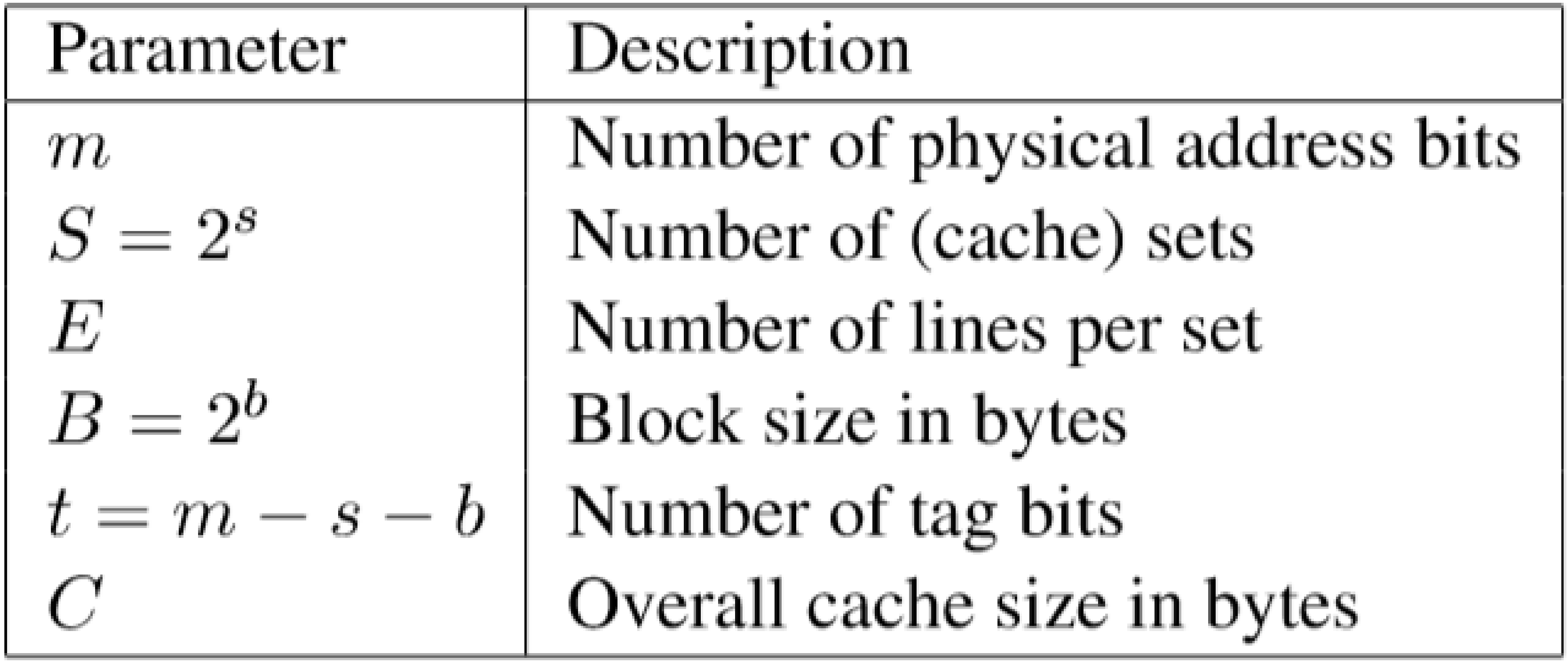
- Each address in a memory system comprises of *m* bits
  - Maximum of M = 2^m unique addresses.
- A cache memory is organized as an array of S=2^s cache sets.
- Each set comprises of *E* cache lines.
  - A cache line stores a single block of $B = 2^b$ bytes of data, along with valid and tag bits.
  - The valid bit indicates if the cache line stores meaningful information
  - The tag (t=m-s-b bits) will uniquely identify the block that is stored on the cache line.



Cache Organization and Address Format

A cache can be viewed as an array of sets.
Each set comprises of one or more cache lines
Each cache line includes a valid bit, tag bits and a cache block.

Summary of cache parameters



Parameter	Description
m	Number of physical address bits
$S = 2^s$	Number of (cache) sets
E	Number of lines per set
$B = 2^b$	Block size in bytes
$t = m - s - b$	Number of tag bits
C	Overall cache size in bytes

Features affecting Cache performance

- ✓ Cache and main memory access time must differ by a reasonable amount.
- ✓ Relative sizes of cache and main memory.
- ✓ Selection of block size - absolute and relative to total cache size.
- ✓ Mechanism for transfer of blocks.
- ✓ How system handles write operations.
- ✓ Method of selecting which block to remove from cache when it is full and a new block is to be put in the cache.

Direct-mapped caches

- ✓ Direct-mapped cache – cache with exactly one line per set ($E = 1$)

- ✓ For such cache, given a word "w" requested from memory, where "w" is stored at address "a". Steps to determine w cache hit or miss
  - *Set selection:* The s bits encoding the set are extracted from address a and used as an index to select the corresponding cache set
  - *Line matching:* Check whether copy of w is present in the unique cache line. This can be done by checking valid and tag bits for that cache line.
  - *Word selection:* If the word present in cache block, use *b* bits of address *a* encoding the words position within the block to read that data word

- ✓ **Cache miss**, the word w request from next level memory hierarchy

- ✓ **Conflict misses** can be resolved using set-associative caches

Set-Associative caches

- ✓ Can store more than one cache line per set
- ✓ If each set in cache stores E lines, $(1 < E < C/B)$, *E-way cache*
- ✓ *Associative memory* – each word in the memory stores along with a unique key and retrieved using key
- ✓ Accessing a word w at address a consists following steps:
  - *Set selection:* Similar to the direct-mapped cache
  - *Line matching:* tag bits of a could match the tag bits of any of the lines in its cache set
  - *Word selection:* If cache line matches, performs as in direct-mapped cache

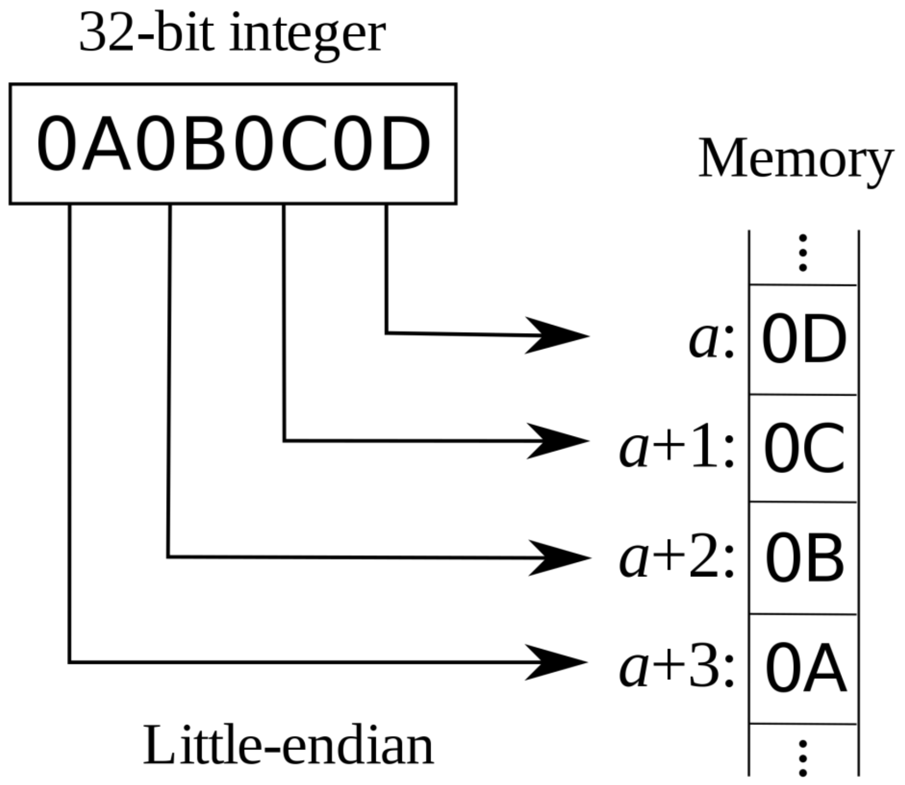
3.0 Memory Models

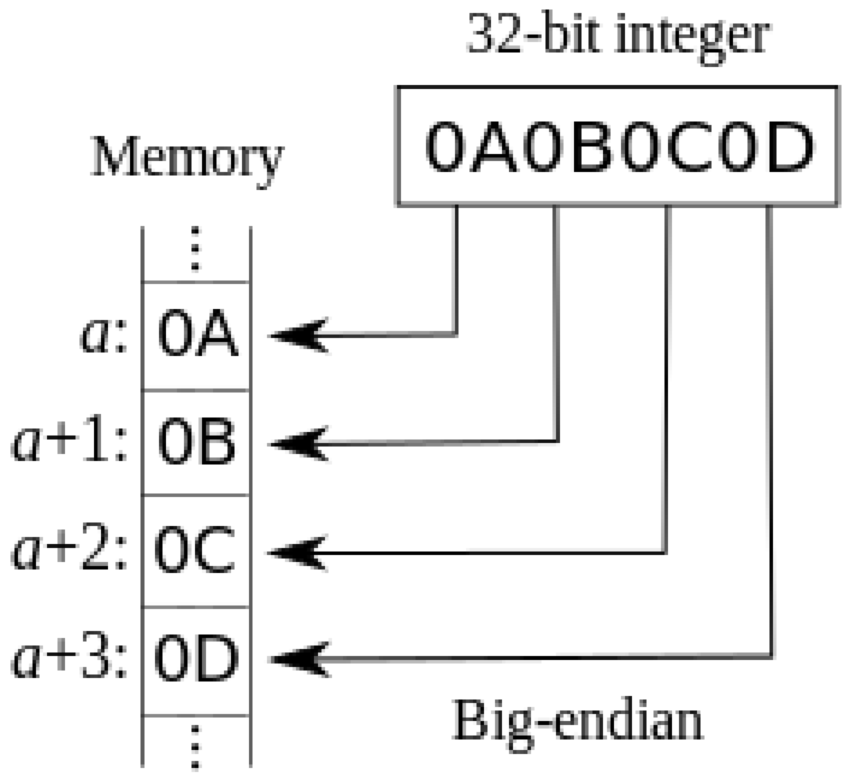
- ✓ A memory model defines how memory is used by programs
- ✓ The hardware, OS, and programming language all contribute to memory model

3.1 Memory Addresses

- ✓ At minimum, a memory model defines a range of memory addresses accessible to the program
  - – In C, these are stored in pointers
  - – In 32-bit architecture, memory addresses are 32-bit unsigned integers, representing addresses 0 to $2^{32} - 1$
  - – each address refers to byte in memory
- ✓ Two critical compatibility concerns when writing a program
  - – alignment of data: *int* will occupy 4 bytes, multiple of 4 (0,4,8)
  - – byte order: byte may represent 8 low/high order bits of *int*

- ✓ **8 low order bits of the *int* (Little Endian) – ARM**
- ✓ **8 high order bits of the *int* (Big Endian) – IBM**





Which is better?

- ✓ This is for you to find out as part of your assignment !

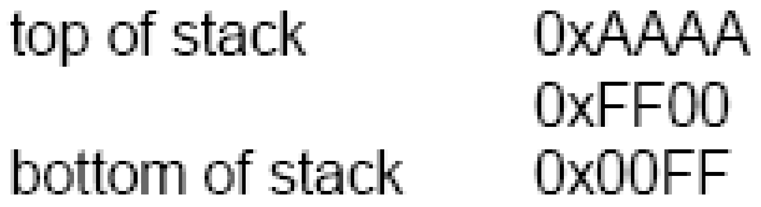
3.2 The Stack (Nested subroutines)

- ✓ Well structured programs will have several subroutines and some subroutines will call other subroutines – this is standard practice and it is known as *nesting*.

- ✓ If subroutine A calls subroutine B, the link register can not store the return address for both subroutines at the same time.

- ✓ In order to preserve the return address of all subroutines an area of computer memory called the ***stack*** is used. The stack is a ‘last-in first-out’ pattern.

- ✓ The stack is a last in first out queue - that means whatever data was added to the stack (‘pushed’) last is taken from the stack (‘popped’) first.
  - E.g. if the values pushed onto a stack where 0x00FF, 0xFF00, 0xAAAA in that order then they would be popped from the stack in the reverse order.
- ✓ In memory the stack is held as a list:



top of stack	0xAAAA
	0xFF00
bottom of stack	0x00FF

Ascending and descending stacks

- ✓ Stacks can also be either ascending or descending.
- ✓ For an *ascending stack*, the memory address of the top of the stack is greater than the memory address for the bottom of the stack.
- ✓ When data is pushed onto an ascending stack, the value held by the stack pointer increases and when data is popped from an ascending stack it decreases.
- ✓ *Descending stacks* work in the opposite way so that the top is at a lower memory address than the bottom.

More about Stacks

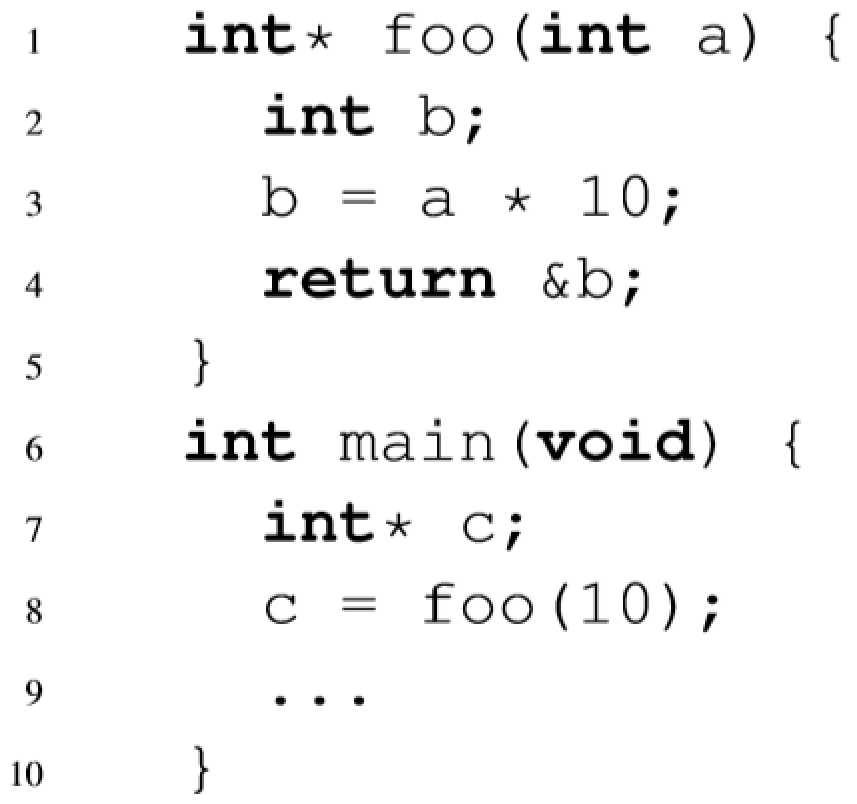
- ✓ For embedded software, it can be disastrous if the stack pointer is incremented beyond the memory allocated for the stack – called as *stack overflow*

- ✓ Bounding the stack usage is an important goal.

- ✓ It becomes difficult with *recursive programs*, where a procedure calls itself

More about Stacks

- ✓ More errors can arise as a result of misuse of the stack



```c
int* foo(int a) {
  int b;
  b = a * 10;
  return &b;
}
int main(void) {
  int* c;
  c = foo(10);
  ...
}
```

- ✓ C provides no protection against such errors

3.3 Memory Protection Units

- ✓ It is important, where multiple simultaneous tasks are being carried out, to prevent one task disrupting another
  - This proves to be very important in embedded applications that permit the downloading of third party software.
  - It can also provide a defense against software bugs in safety-critical applications.
- ✓ Many processors provide memory protection hardware
  - Tasks are assigned their own address space.
  - If a task attempt to access memory outside its address space, a segmentation fault (or some other exception) will result.

3.4 Dynamic Memory Allocation

- ✓ Software applications often have indeterminate memory requirements, depending on parameters or the user input

- ✓ To support such applications, there are dynamic memory allocation schemes
  - A program can request that the operating system allocate more memory.
  - The memory is allocated from a data structure known as a **heap**, this facilitates tracking which portions of memory are in use by which application.

- ✓ Memory allocation occurs via an operating system call

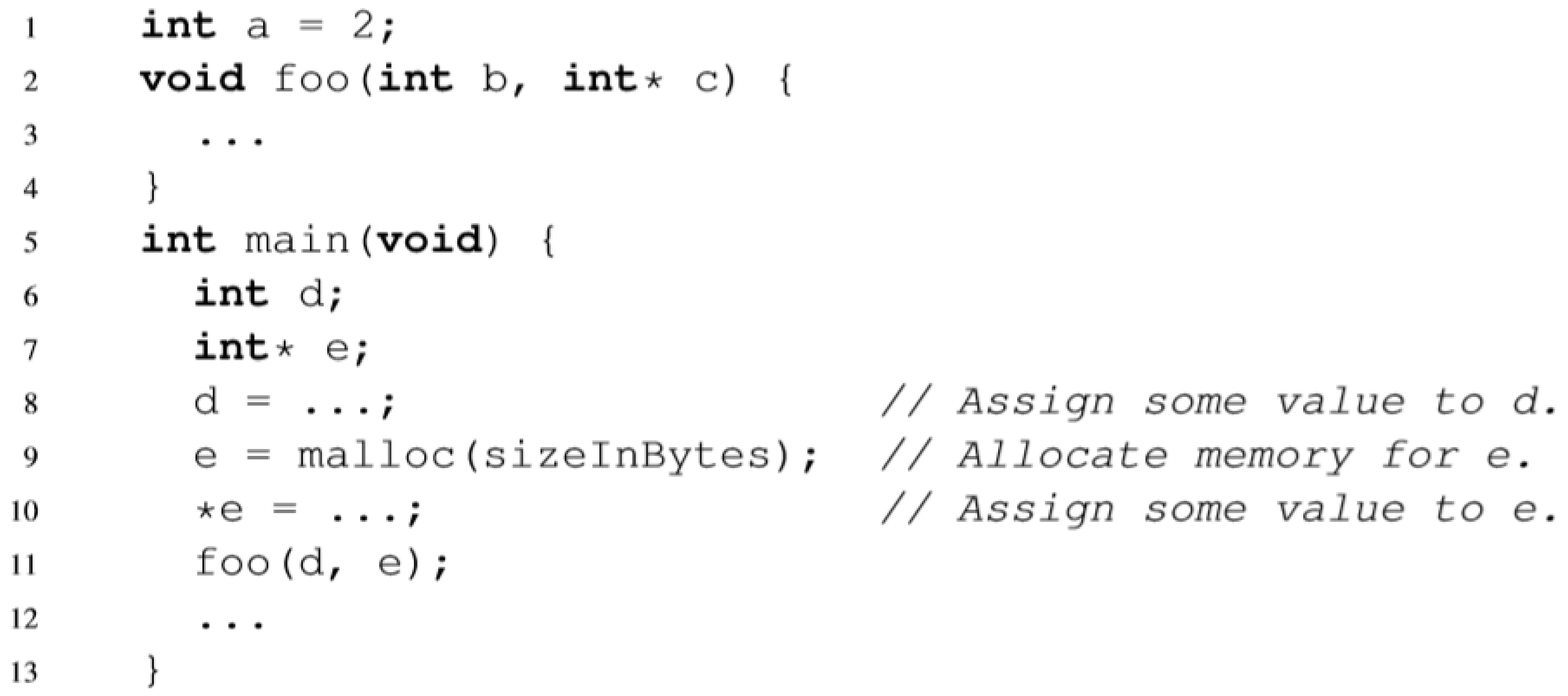
- ✓ Support for memory allocation usually includes **garbage collection**
  - A garbage collector is a task that runs periodically or when there is a lack of memory.
  - It analyzes the data structures that a program has allocated and frees any portion of memory that are no longer referenced within the program.
- ✓ Nevertheless, it is still possible to inadvertently accumulate memory that is never freed. This is termed as a **memory leak**
  - For embedded systems that must run for long periods of time memory leaks are very bad news, as physical memory will be exhausted and the program execution fail.

Memory Fragmentation

- ✓ This occurs when a program chaotically allocates and deallocates memory in varying sizes
- ✓ The free memory chunks, interspersed with allocated ones, become to small to use – *defragmentation* is required
- ✓ Defragmentation and garbage collection are very problematic for real-time systems (effects)
    - Executing tasks are stopped while they take place.
    - This can create substantial pause times, often milliseconds.

3.5 Memory model and C programs

- ✓ C programs store data on the stack, heap, memory locations



```c
int a = 2;
void foo(int b, int* c) {
  ...
}
int main(void) {
  int d;
  int* e;
  d = ...;                    // Assign some value to d.
  e = malloc(sizeInBytes);    // Allocate memory for e.
  *e = ...;                   // Assign some value to e.
  foo(d, e);
  ...
}
```

Summary

- ✓ Embedded designer needs to know memory architecture and memory model of programming language

- ✓ Incorrect uses of memory can lead to extremely subtle errors

- ✓ Needs to know which portions of address space refer to *volatile* and non-volatile memory

  - a memory whose contents are lost when power is cut-off

- ✓ *Memory hierarchy* combines different memory technologies to increase overall memory capacity

- ✓ Programmer needs to be careful with DMA, where systems run long

See you in the next class (May 14th)

The End