Memory Architectures

01266212
CYBER PHYSICAL SYSTEM DESIGN

SEMESTER 1-2022

Original contents from
Edward A. Lee and Prabal Dutta, UC Berkeley, EECS 149/249A

Role of Memory in Embedded Systems

- Traditional roles: Storage and Communication for Programs
- Communication with Sensors and Actuators
- Often much more constrained than in general-purpose computing
  - Size, power, reliability, etc.
- Can be important for programmers to understand these constraints

Memory Architecture

- Volatile (lose data when power off) vs Non-volatile (keep data when power off)
- Most embedded systems need at least some non-volatile memory and some volatile memory.
- Non-volatile memory in embedded systems is used to store code and other required data for the systems

Memory Architecture: Issues

- Types of memory
  - volatile vs. non-volatile, SRAM vs. DRAM
- Memory maps
  - Harvard architecture
  - Memory-mapped I/O
- Memory organization
  - statically allocated
  - stacks
  - heaps (allocation, fragmentation, garbage collection)
- The memory model of C
- Memory hierarchies
  - scratchpads, caches, virtual memory
- Memory protection
  - segmented spaces

These issues loom larger in embedded systems than in general-purpose computing.

Volatile Memory

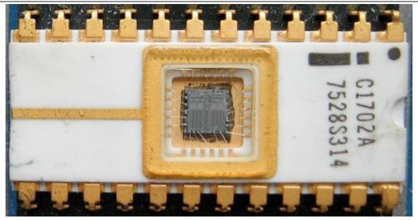
Loses contents when power is off.

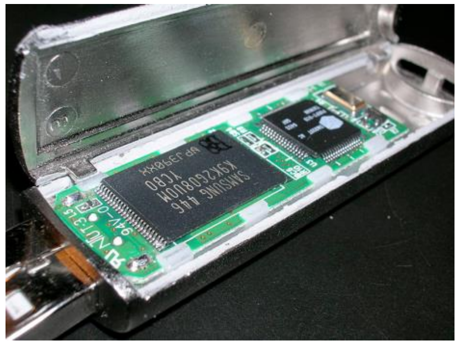
- SRAM: static random-access memory
  - Fast, deterministic access time
  - But more power hungry and less dense than DRAM
  - Used for caches, scratchpads, and small embedded memories
- DRAM: dynamic random-access memory
  - Slower than SRAM
  - Access time depends on the sequence of addresses
  - Denser than SRAM (higher capacity)
  - Requires periodic refresh (typically every 64msec)
  - Typically used for main memory
- Boot loader
  - On power up, transfers data from non-volatile to volatile memory.

Non-Volatile Memory

Preserves contents when power is off

- EPROM: erasable programmable read only memory
  - Invented by Dov Frohman of Intel in 1971
  - Erase by exposing the chip to strong UV light
- EEPROM: electrically erasable programmable read-only memory
  - Invented by George Perlegos at Intel in 1978
- Flash memory
  - Invented by Dr. Fujio Masuoka at Toshiba around 1980
  - Erased a “block” at a time
  - Limited number of program/erase cycles (~ 100,000)
  - Controllers can get quite complex
- Disk drives
  - Not as well suited for embedded systems





Images from the Wikimedia Commons

Flash memory

Flash memories have reasonably fast read times (but not as fast as SRAM and DRAM).

The write times are much longer than the read times.

There are two types of flash memories, known as NOR and NAND flash.

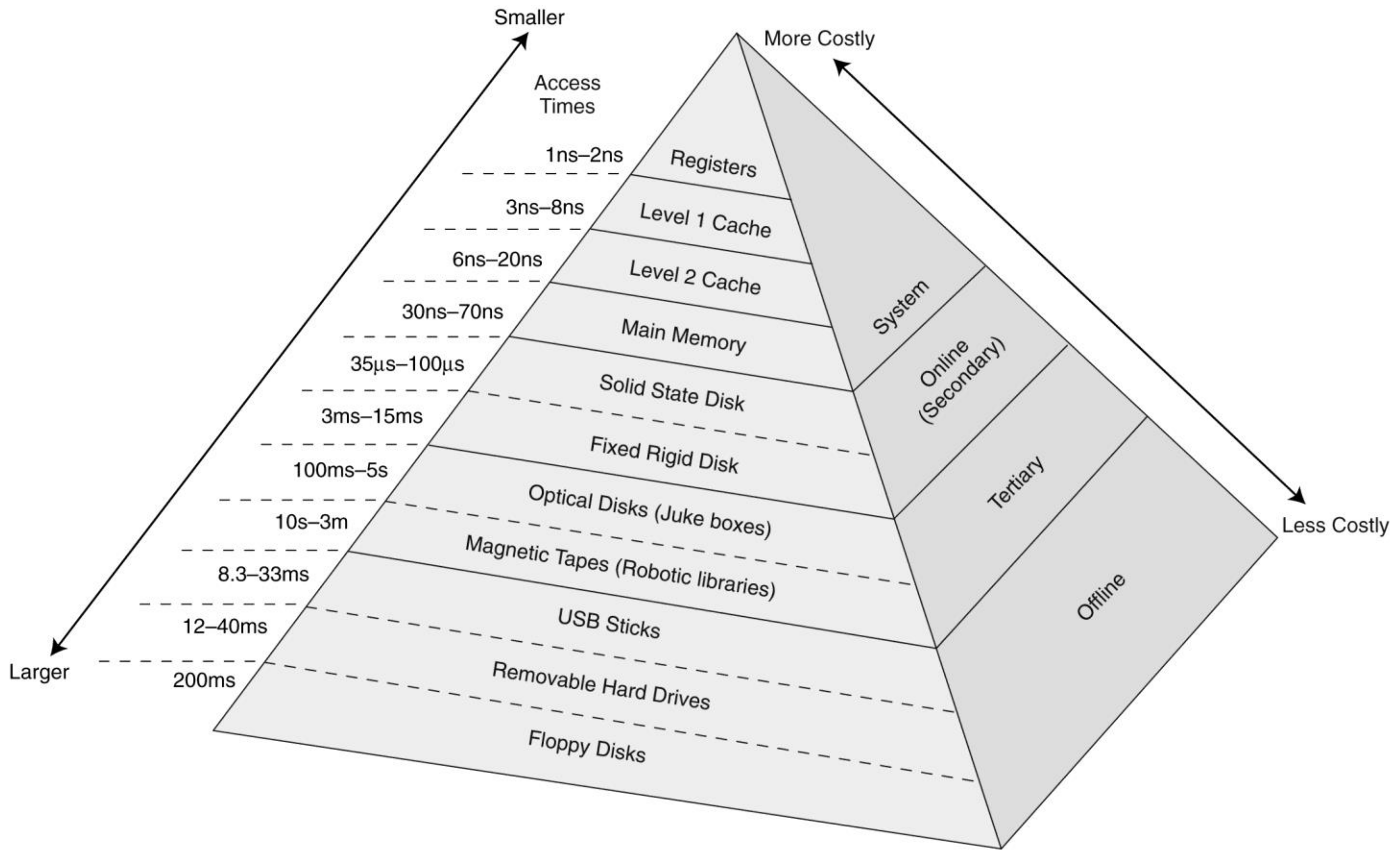
- NOR flash has longer erase and write times, but it can be accessed like a RAM.
- NAND flash is less expensive and has faster erase and write times, but data must be read a block at a time,

  where a block is hundreds to thousands of bits. (Like as secondary storage, such as HHD)

Memory Hierarchy in PC

- Many processors use a memory hierarchy to provide the best performance at the lowest cost
- It combines different memory technologies to increase the overall memory capacity while optimizing cost, latency, and energy consumption
- Generally speaking, faster memory is more expensive than slower memory.
- Small, fast storage elements are kept in the CPU, larger, slower main memory is accessed through the data bus.
- Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.

Memory Hierarchy

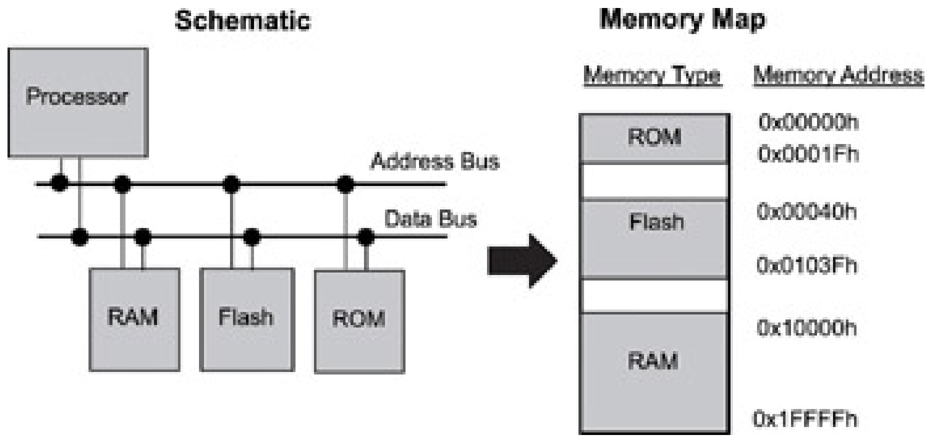


Memory Hierarchy in PC

- We are most interested in the memory hierarchy that involves registers, cache, main memory, and virtual memory.
- Registers are storage locations available on the processor itself.
- Virtual memory is typically implemented using a hard drive; it extends the address space from RAM to the hard drive.
- Virtual memory provides more space: Cache memory provides speed.

Memory Maps

- A memory map for a processor defines how addresses get mapped to hardware.
- The total size of the address space is constrained by the address width of the processor.
- For instant, a 32-bit processor can address 2^{32} locations, or 4 gigabytes (GB), assuming each address refers to one byte.
- The address width typically matches the word width, except for 8-bit processors, where the address width is typically higher (often 16 bits).



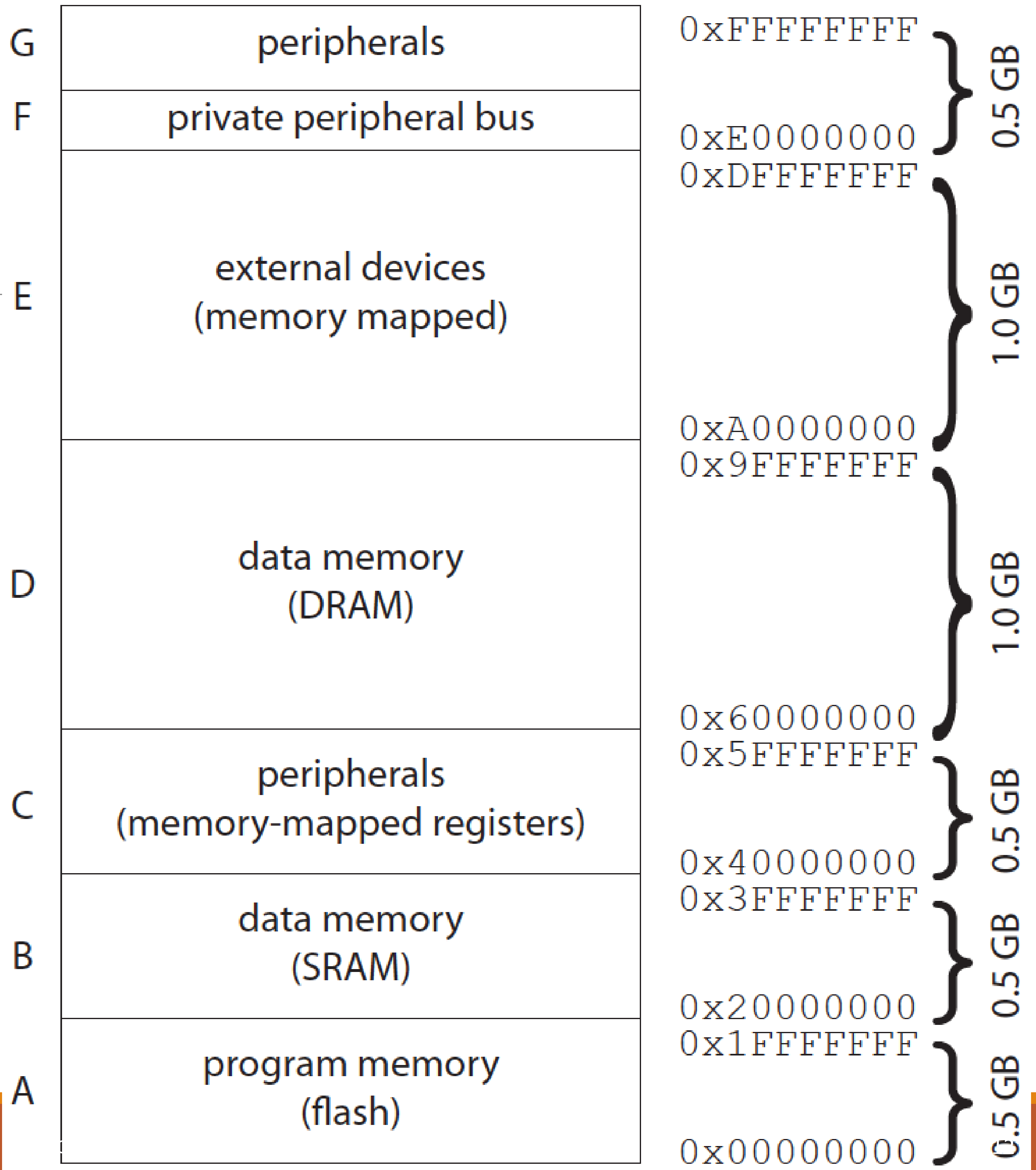
Source: http://www.embeddedlinux.org.cn/

Memory Map of an ARM Cortex^{TM} - M3 architecture

Defines the mapping of addresses to physical memory.

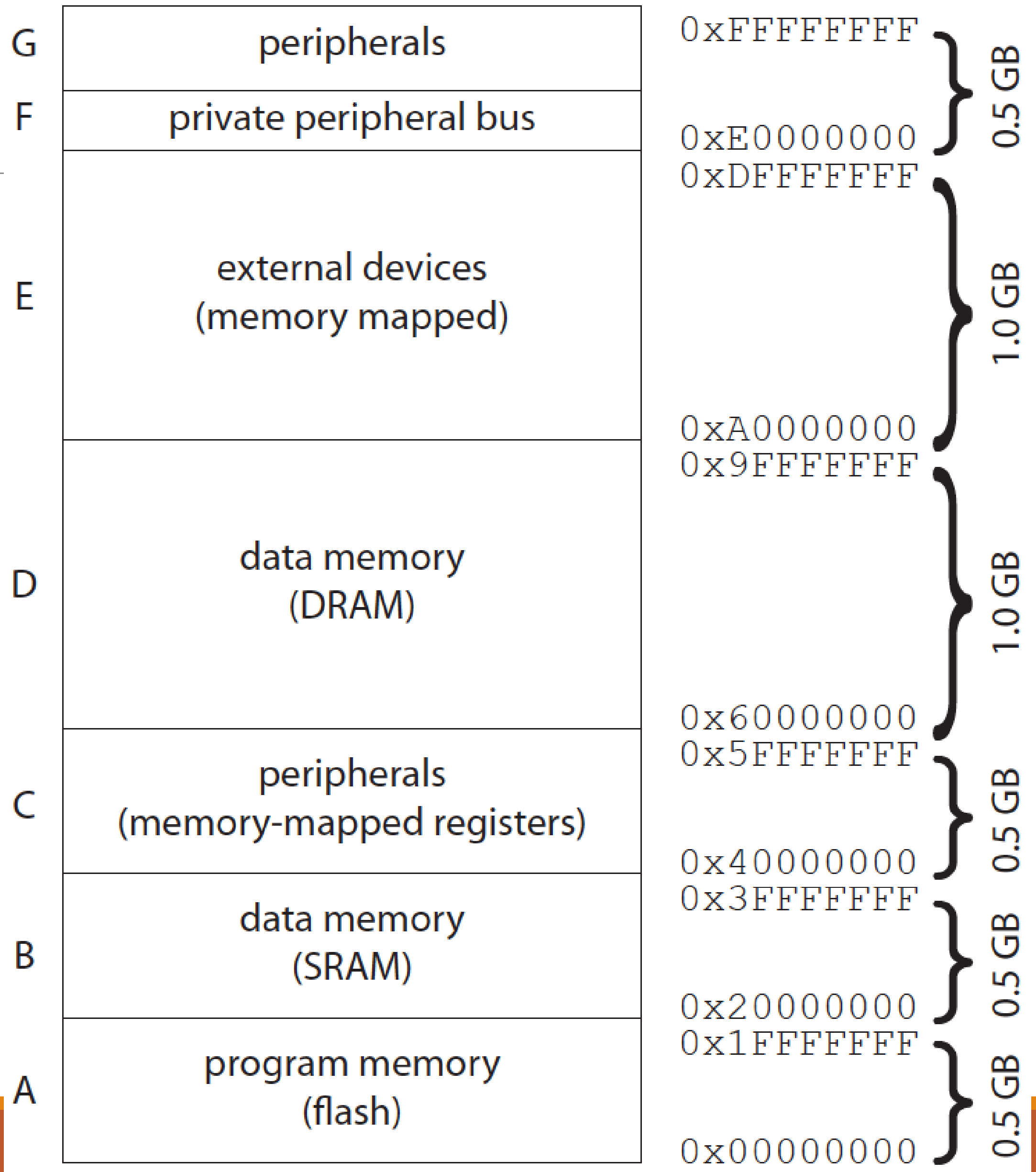
Why do this?

Note that this does not define how much physical memory there is!



Memory Map of an ARM Cortex™ - M3 architecture

- For example, a processor has 256 KB of on-chip flash memory (significantly below 0.5 GB that the architecture allows).
- This memory is mapped to addresses `0x00000000` through `0x0003FFFF`.
- The remaining addresses that the architecture allows for program memory, which are `0x00040000` through `0x1FFFFFFF`, are "reserved addresses."
- They should not be used by a compiler targeting this particular device.



SOURCE: LEE, BERKELEY

Memory-mapped Registers

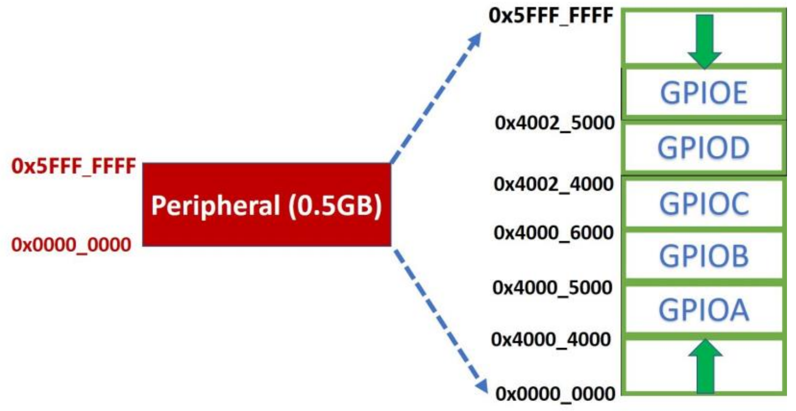
A on-chip peripherals including `timers`, `ADCs`, `GPIO`, `UARTs`, and other I/O devices are accessed by the processor using some of the memory addresses in the range from `0x40000000` to `0x5FFFFFFF`.

Each of these devices occupies a few of the memory addresses by providing memory-mapped registers.

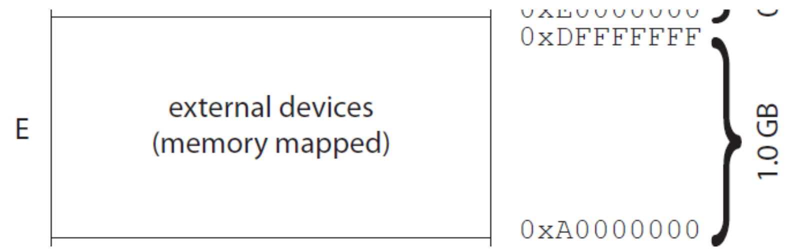
The processor may write to some of these registers to configure and/or control the peripheral, or to provide data to be produced on an output.

Some of the registers may be read to retrieve input data obtained by the peripheral.

Use of pointers to access Peripheral Registers



https://microcontrollerslab.com/accessing-memory-mapped-io-microcontrollers-pointer/



Another Example: AVR

The AVR is an 8-bit single chip microcontroller first developed by Atmel in 1997. The AVR was one of the first microcontroller families to use on-chip flash memory for program storage. It has a modified Harvard architecture.[1]

AVR was conceived by two students at the Norwegian Institute of Technology (NTH) Alf-Egil Bogen and Vegard Wollan, who approached Atmel in Silicon Valley to produce it.

[1] A Harvard architecture uses separate memory spaces for program and data. It originated with the Harvard Mark I relay-based computer (used during World War II), which stored the program on punched tape (24 bits wide) and the data in electro-mechanical counters.

A Use of AVR: Arduino

Arduino is a family of open-source hardware boards built around either 8-bit AVR processors or 32-bit ARM processors.

Example:
Atmel AVR
Atmega328
28-pin DIP on an
Arduino Duemilanove
board

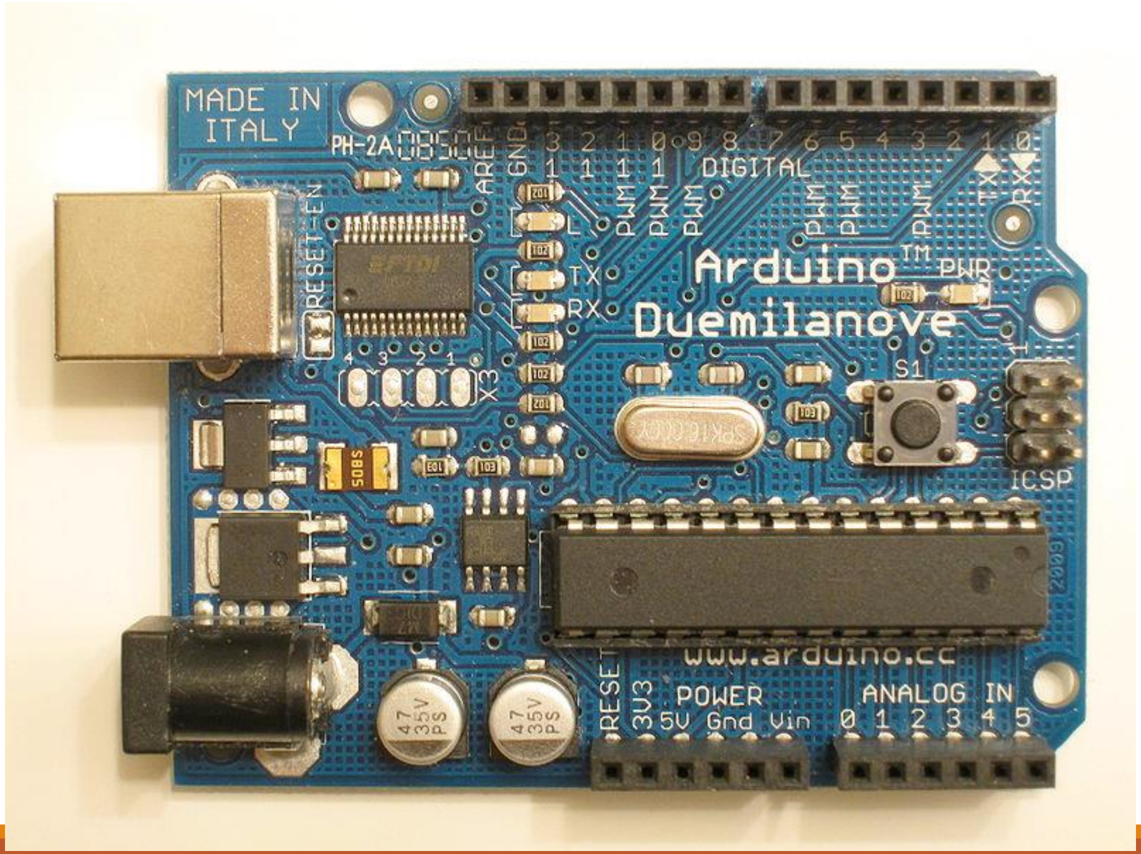
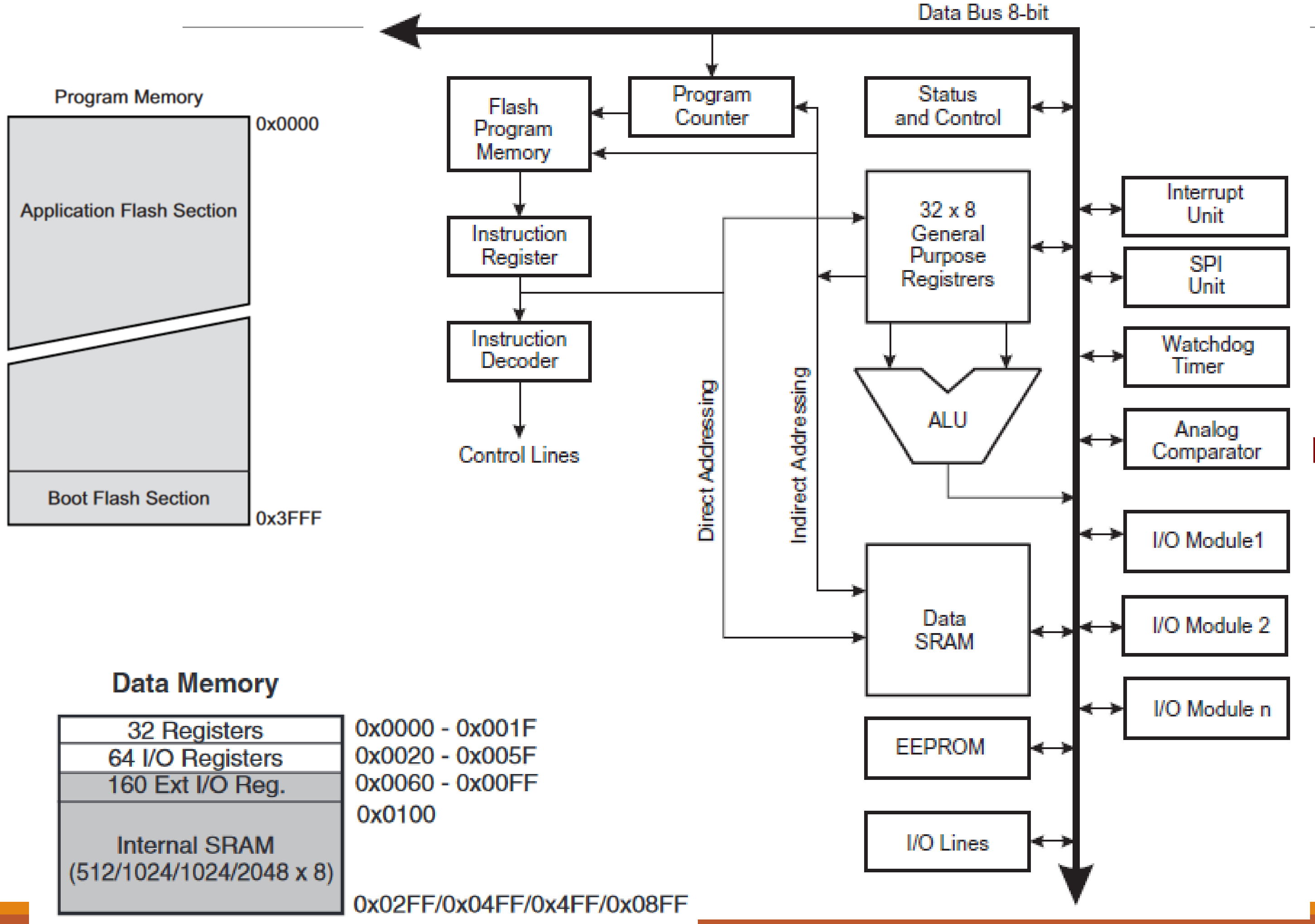


Image from Wikimedia Commons

Atmel ATmega 328P:

An 8-bit microcontroller with 16-bit addresses

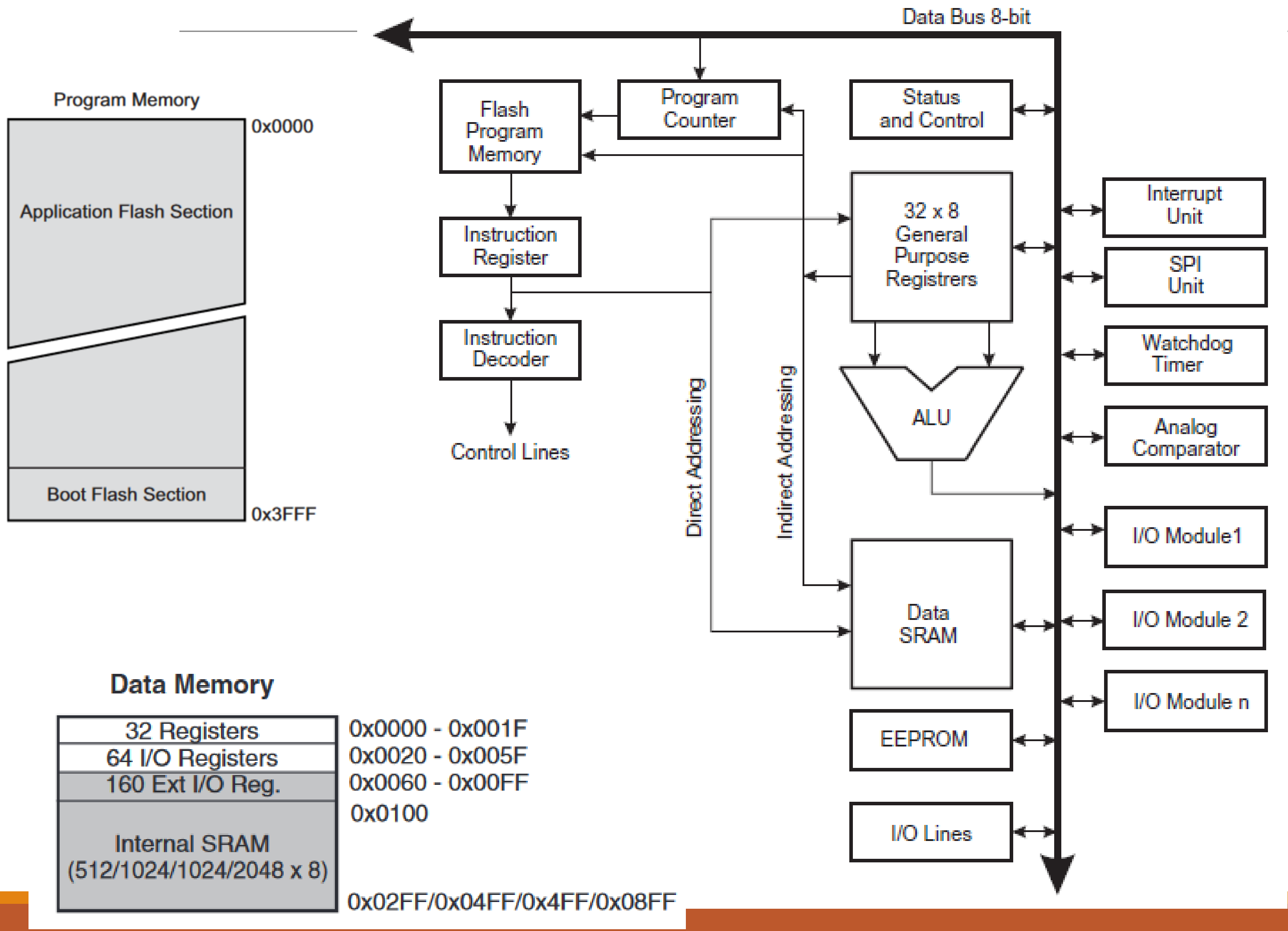


Why is it called an 8-bit microcontroller?

http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

Atmel ATmega 328P:

An 8-bit microcontroller with 16-bit addresses



Additional I/O on the command module:

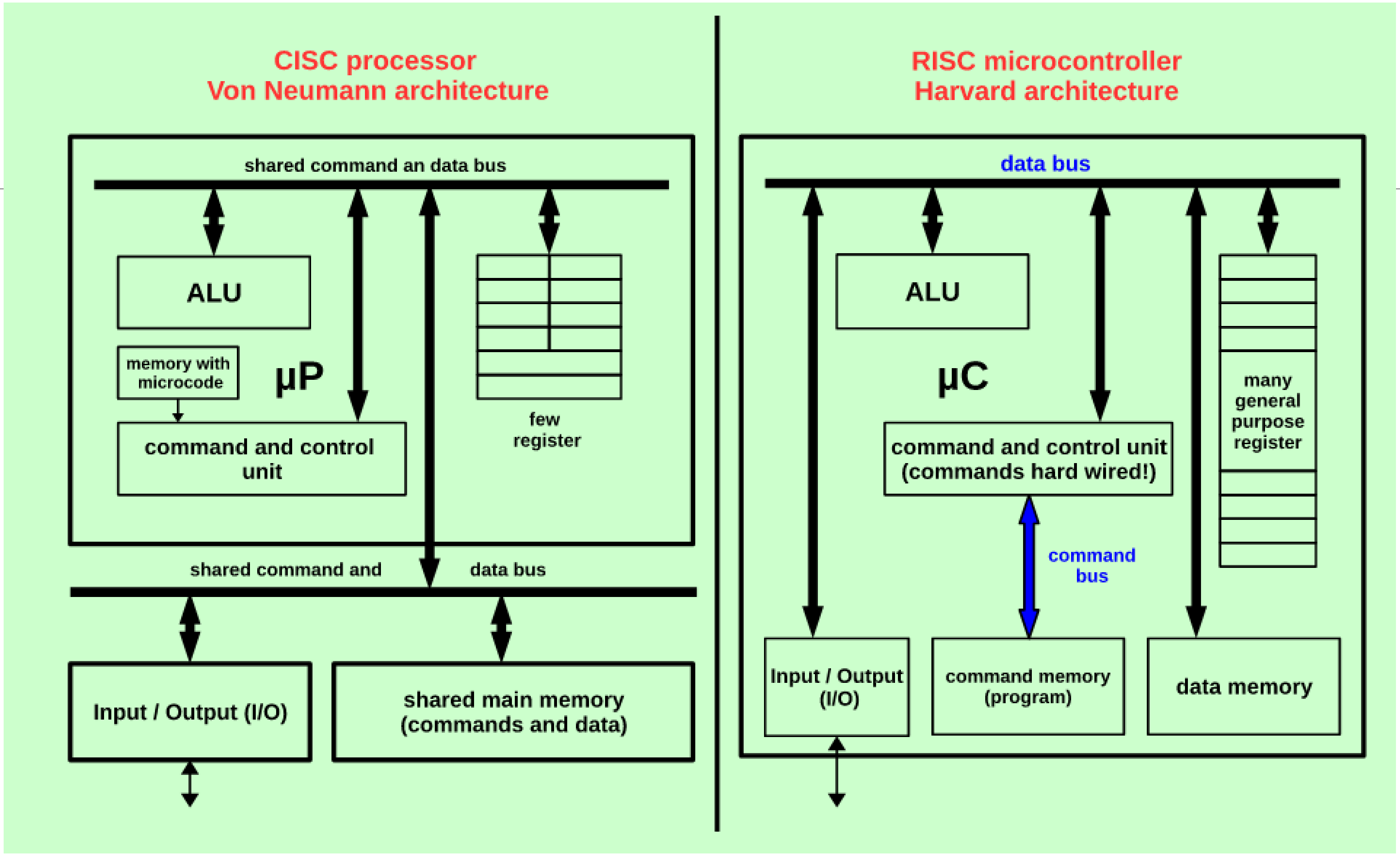
- Two 8-bit timer/counters
- One 16-bit timer/counter
- 6 PWM channels
- 8-channel, 10-bit ADC
- One serial UART
- 2-wire serial interface

AVR Memory Summary

What is EEPROM

Two Types of Memory Architecture

- Notice that the previous architecture separates addresses used for program memory (flash program memory) from those used for data memory (Data SRAM).
- This (typical) pattern allows these memories to be accessed via separate buses, permitting instructions and data to be fetched simultaneously. This effectively doubles the memory bandwidth.
- Such a separation of program memory from data memory is known as a **Harvard architecture**.
- Another type, the classical **von Neumann architecture**, which stores program and data in the same memory.


CISC processor
Von Neumann architecture
shared command an data bus
ALU
memory with microcode
µP
command and control unit
few register
shared command and data bus
Input / Output (I/O)
shared main memory (commands and data)
RISC microcontroller
Harvard architecture
data bus
ALU
µC
command and control unit (commands hard wired!)
many general purpose register
command bus
Input / Output (I/O)
command memory (program)
data memory

Register Files

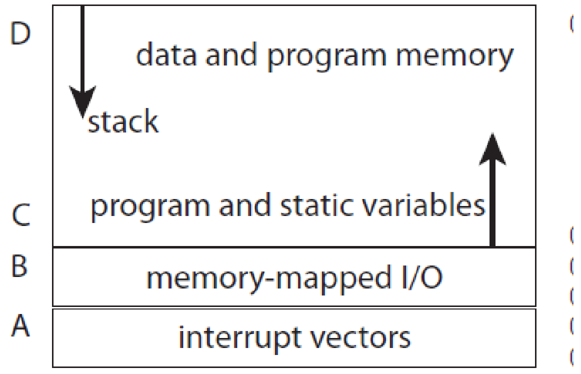
- The most tightly integrated memory in a processor is the register file.
- Each register in the file stores a **word.** (one byte on an 8-bit architecture, four bytes on a 32-bit architecture, and eight bytes on a 64-bit architecture)
- The register file may be implemented directly using flip flops in the processor circuitry, or the registers may be collected into a single memory bank (like SRAM).
- The number of registers in a processor is usually small.
- An instruction set architecture (ISA) typically provides instructions that can access one, two, or three registers.
- Complexity of architecture $\propto$ # of registers

Memory Addresses

- At a minimum, a memory model defines a range of memory addresses accessible to the program.
- In a **32-bit architecture**, memory addresses are 32-bit unsigned integers, capable of representing addresses 0 to $2^{32} – 1$.
- Each address refers to a byte (eight bits) in memory.
- The C `char` data type references a byte. (one memory location)
- The C `int` data type references a sequence of at least two bytes (for 16-bit machine) and four bytes (for 32-bit/64 bits). (two/four memory locations)
- The `double` data type in C refers to a sequence of eight bytes encoded according to the IEEE floating point standard (IEEE 754).

Memory Organization for Programs

- Statically-allocated memory
  - Compiler chooses the address at which to store a variable.
- Stack
  - Dynamically allocated memory with a Last-in, First-out (LIFO) strategy
  - A stack pointer (typically a register) contains the memory address
  - of the top of the stack.
  - Stores temporary variables created by a function/procedure.
  - Variables are declared, stored and initialized during runtime.
- Heap
  - Dynamically allocated memory.
  - Stores global variables (By default, all global variable are stored in heap memory space.)
  - tightly managed by the CPU (not programmer).



Statically-Allocated Memory in C

```
char x;
int main(void) {
    x = 0x20;
    …
}
```

Compiler chooses what address to use for x, and the variable is accessible across procedures. The variable’s lifetime is the total duration of the program execution.

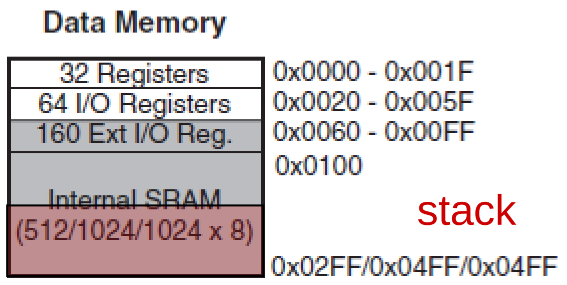
Statically-Allocated Memory with Limited Scope

```
void foo(void) {
    static char x;
    x = 0x20;
    …
}
```

Compiler chooses what address to use for x, but the variable is meant to be accessible only in foo(). The variable's lifetime is the total duration of the program execution (values persist across calls to foo()).

Variables on the Stack (“automatic variables”)

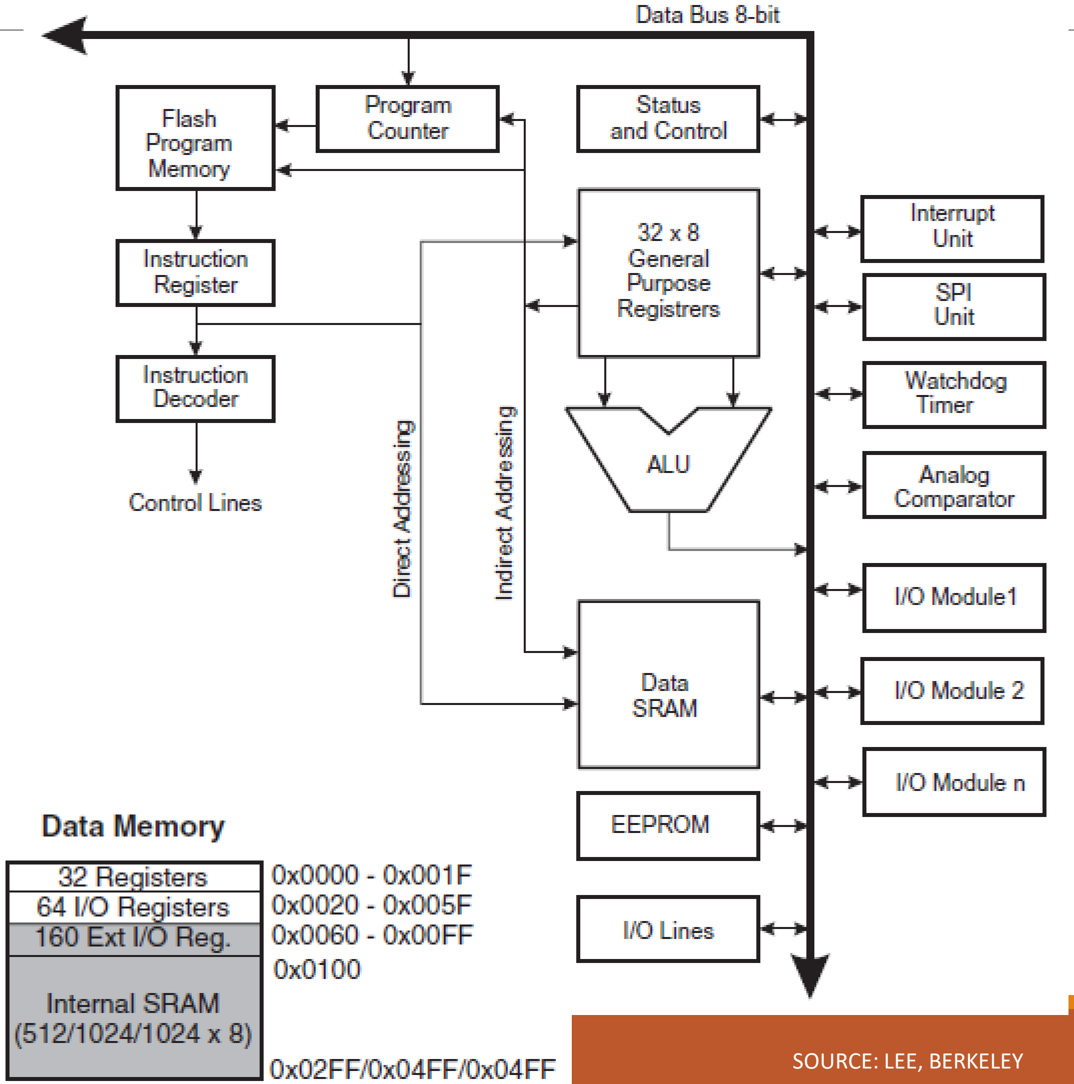
```
void foo(void) {
     char x;
     x = 0x20;
     …
}
```



As nested procedures get called, the stack pointer moves to lower memory addresses. When these procedures, return, the pointer moves up.

When the procedure is called, x is assigned an address on the stack (by decrementing the stack pointer). When the procedure returns, the memory is freed (by incrementing the stack pointer). The variable persists only for the duration of the call to foo().

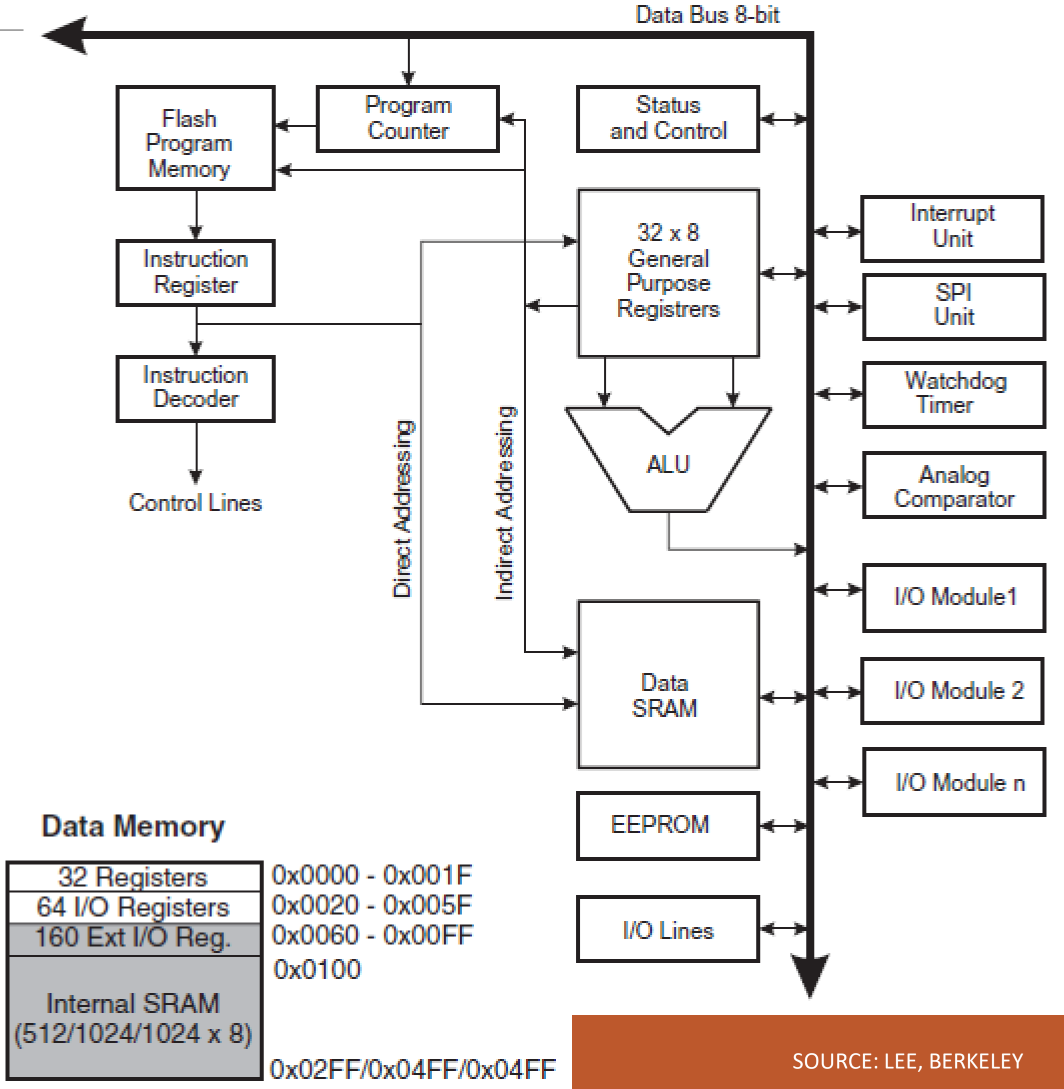
Question 1



What is meant by the following C code:

```
char x;
void foo(void) {
  x = 0x20;
  …
}
```

Answer 1

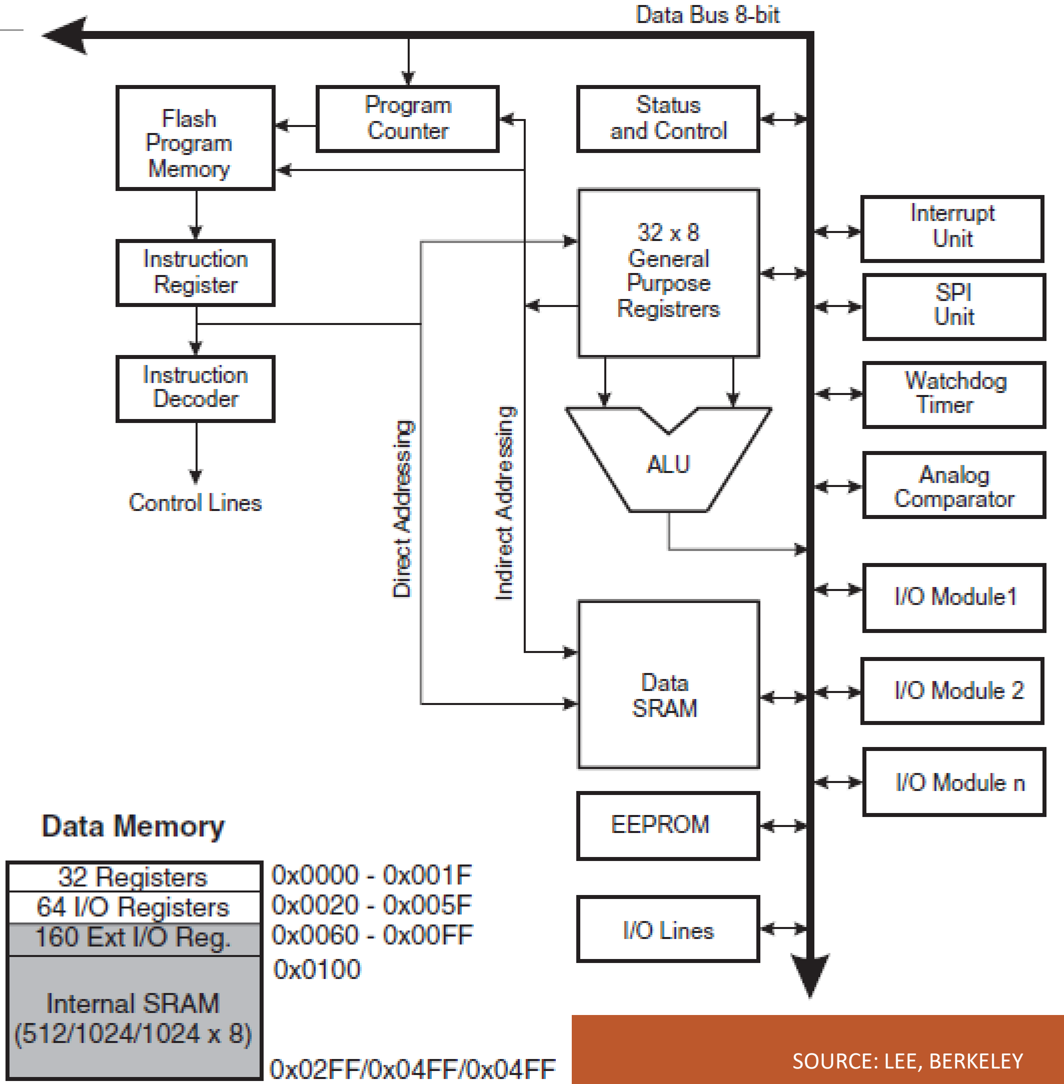


What is meant by the following C code:

```
char x;
void foo(void) {
  x = 0x20;
  …
}
```

An 8-bit quantity (hex 0x20) is stored at an address in statically allocated memory in internal RAM determined by the compiler.

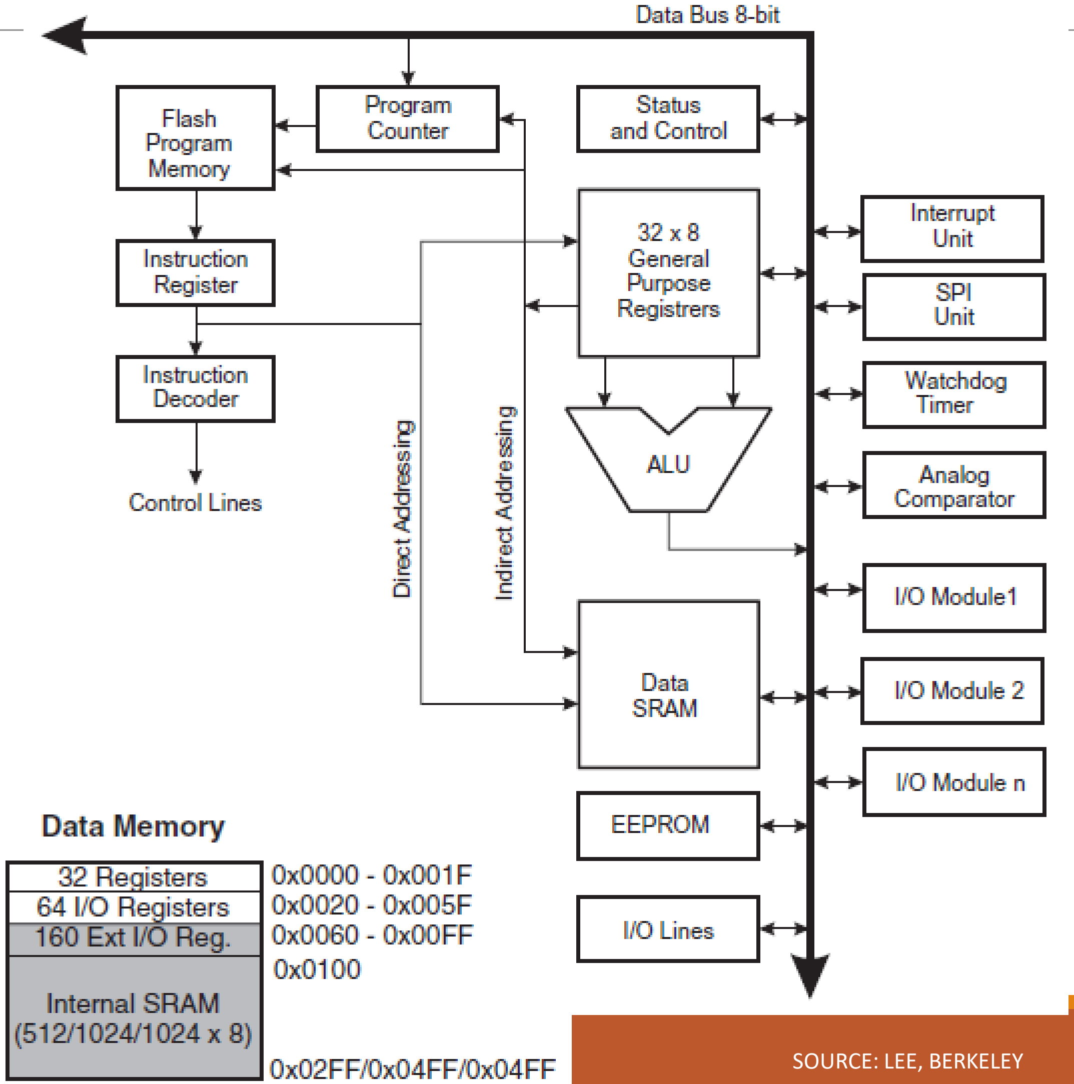
Question 2



What is meant by the following C code:

```
char *x;
void foo(void) {
  x = 0x20;
  …
}
```

Answer 2

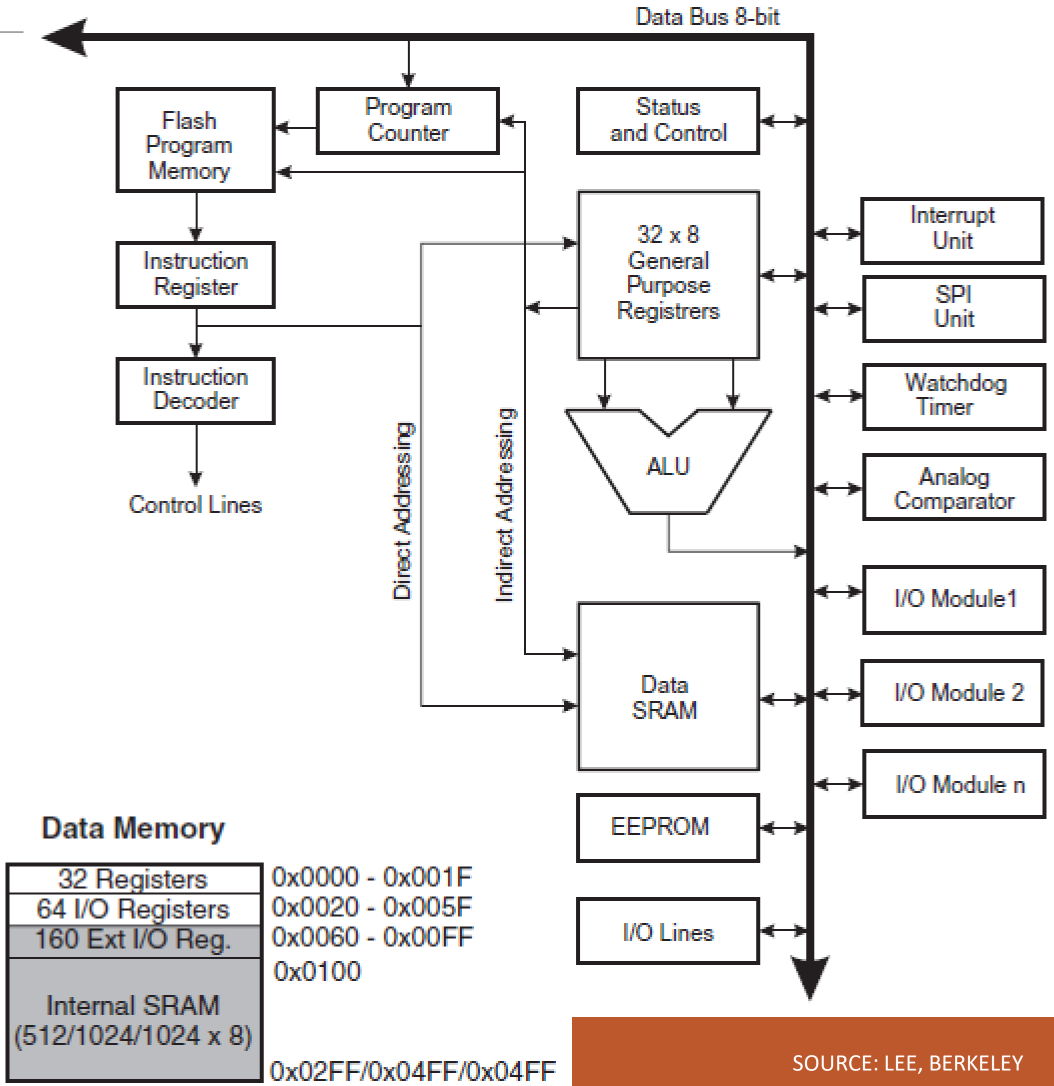


What is meant by the following C code:

```
char *x;
void foo(void) {
   x = 0x20;
   …
}
```

An 16-bit quantity (hex 0x0020) is stored at an address in statically allocated memory in internal RAM determined by the compiler.

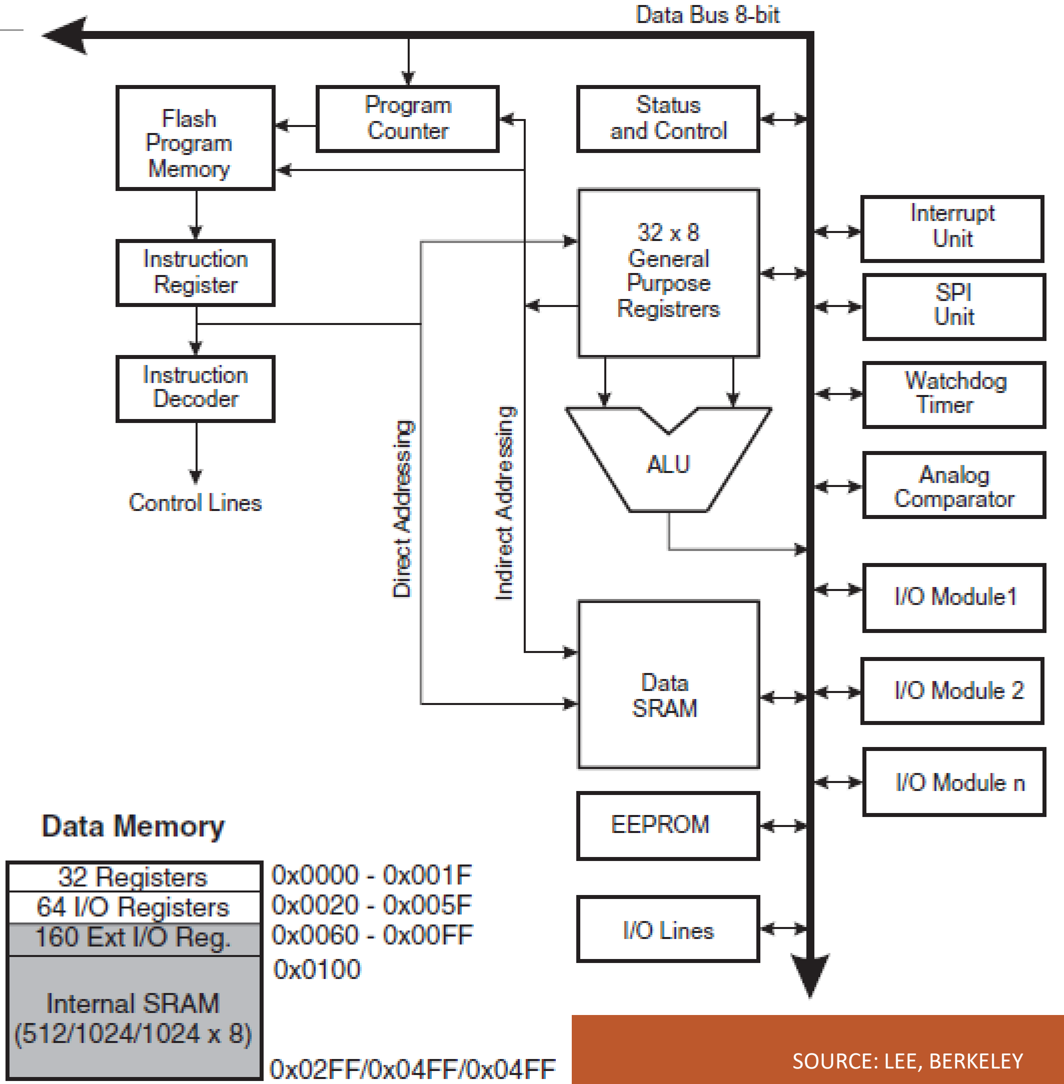
Question 3



What is meant by the following C code:

```
char *x, y;
void foo(void) {
  x = 0x20;
  y = *x;
  …
}
```

Answer 3

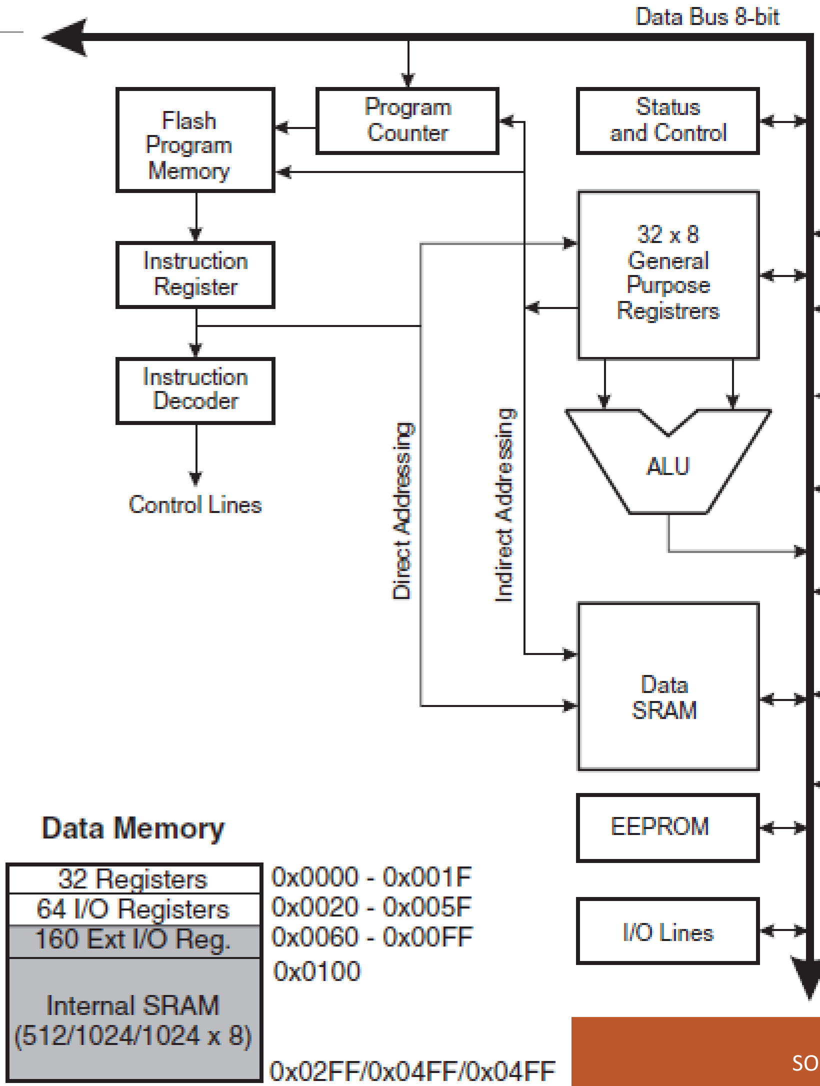


What is meant by the following C code:

```
char *x, y;
void foo(void) {
   x = 0x20;
   y = *x;
   …
}
```

The 8-bit quantity in the I/O register at location 0x20 is loaded into y, which is at a location in internal SRAM determined by the compiler.

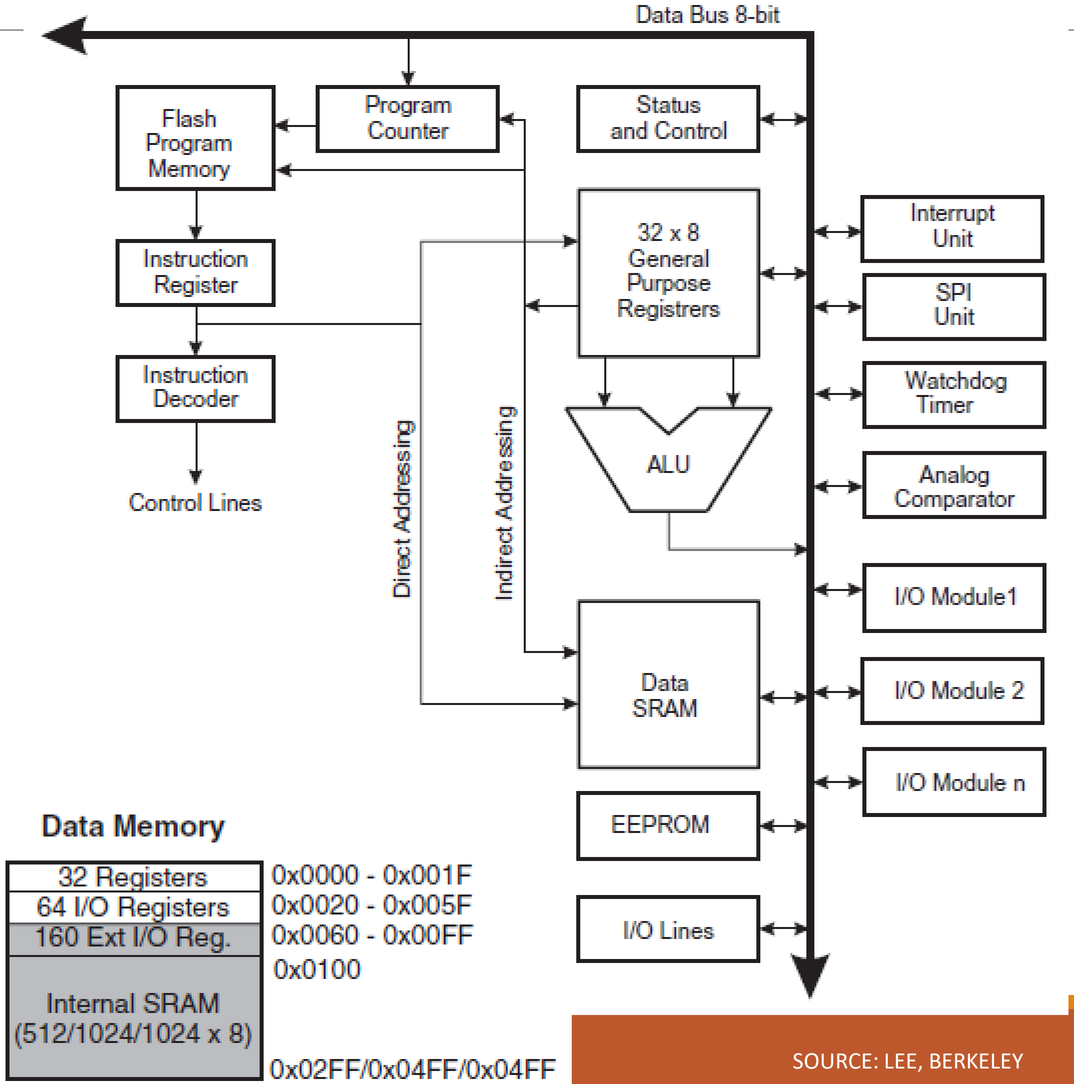
Question 4



```
char foo() {
  char *x, y;
  x = 0x20;
  y = *x;
  return y;
}
char z;
int main(void) {
  z = foo();
  …
}
```

Where are x, y, z in memory?

Answer 4

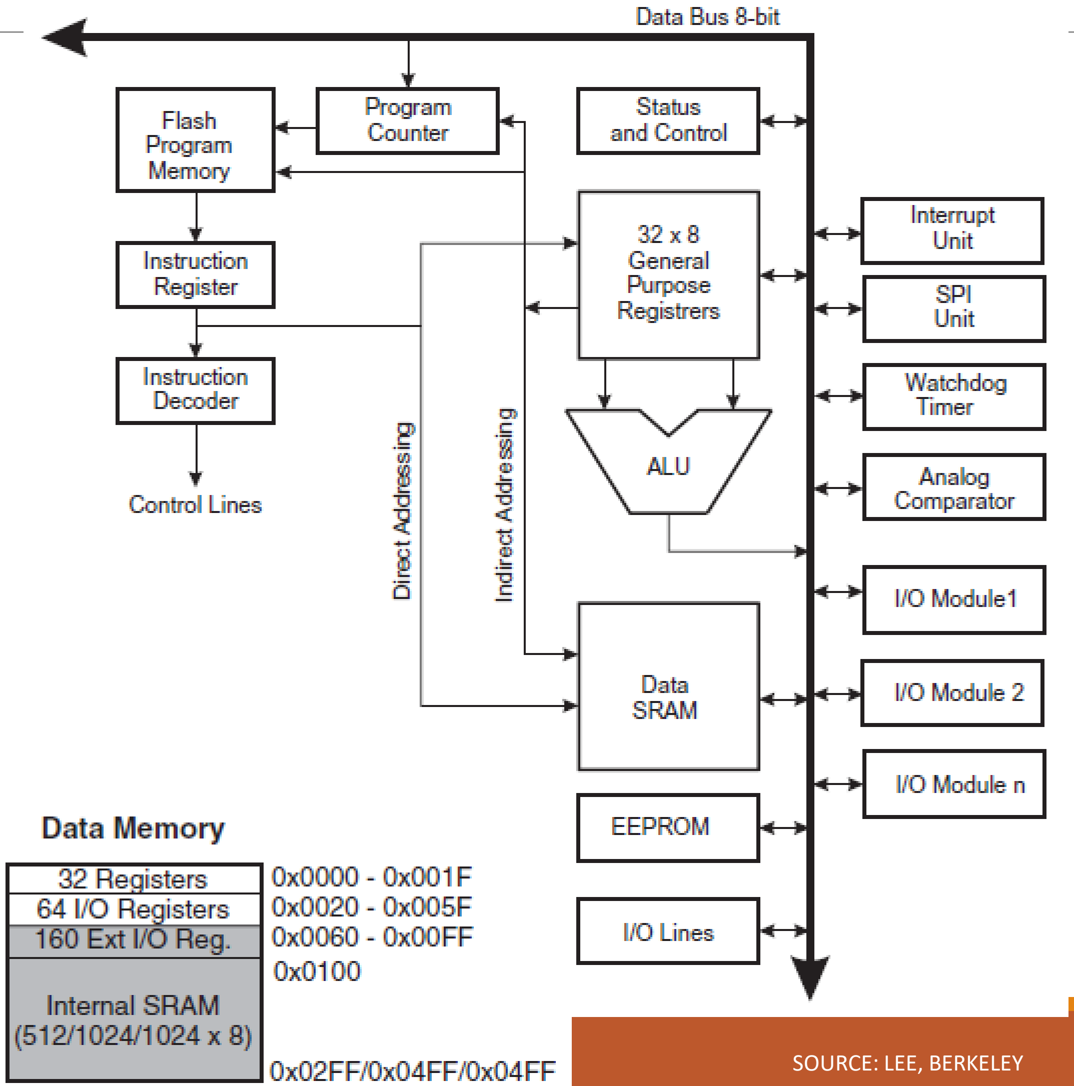


32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Reg.	0x0060 - 0x00FF
Internal SRAM (512/1024/1024 x 8)	0x0100
	0x02FF/0x04FF/0x04FF

```
char foo() {
  char *x, y;
  x = 0x20;
  y = *x;
  return y;
}
char z;
int main(void) {
  z = foo();
  …
}
```

x occupies 2 bytes on the stack, y occupies 1 byte on the stack, and z occupies 1 byte in static memory.

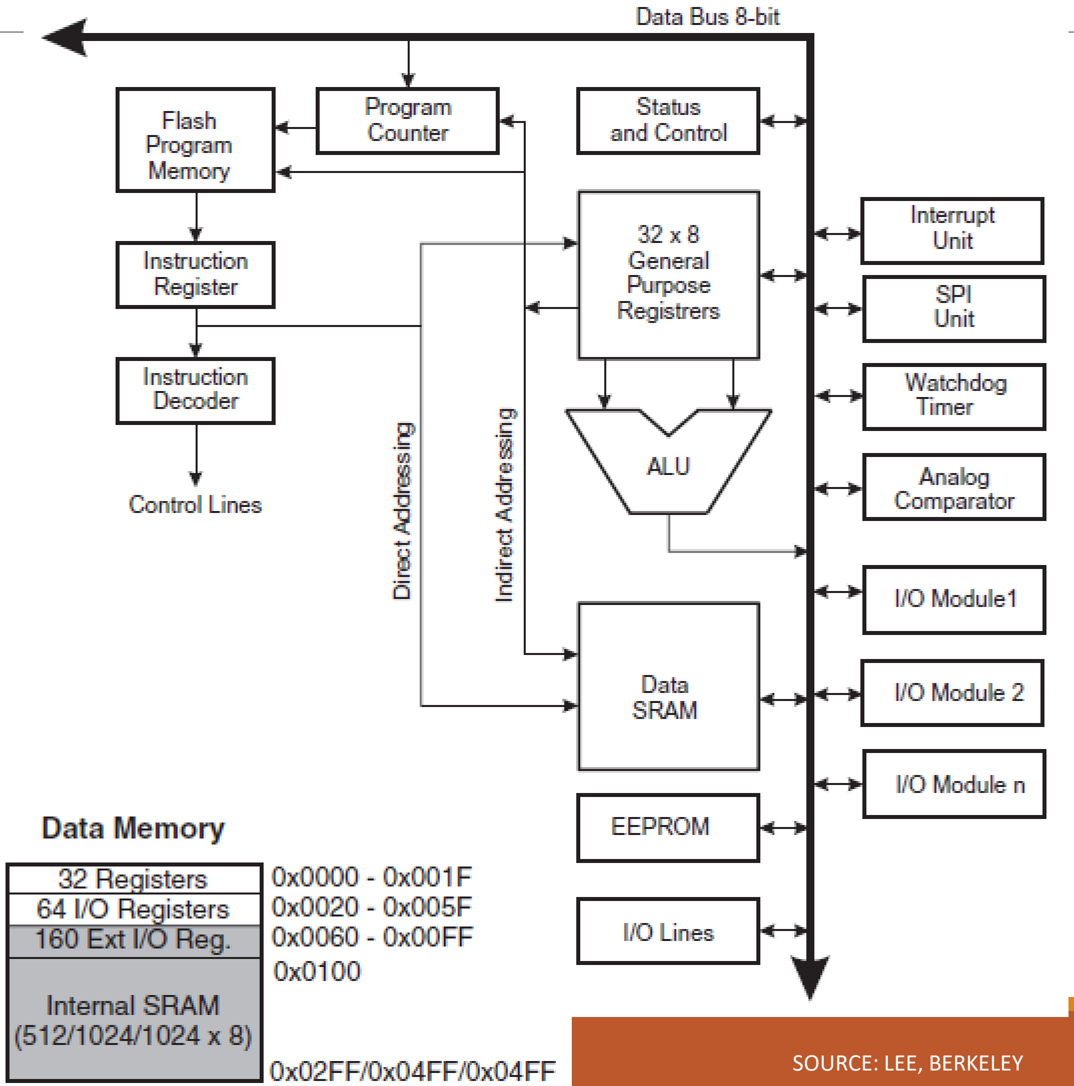
Question 5



What is meant by the following C code:

```
void foo(void) {
   char *x, y;
   x = &y;
   *x = 0x20;
   …
}
```

Answer 5

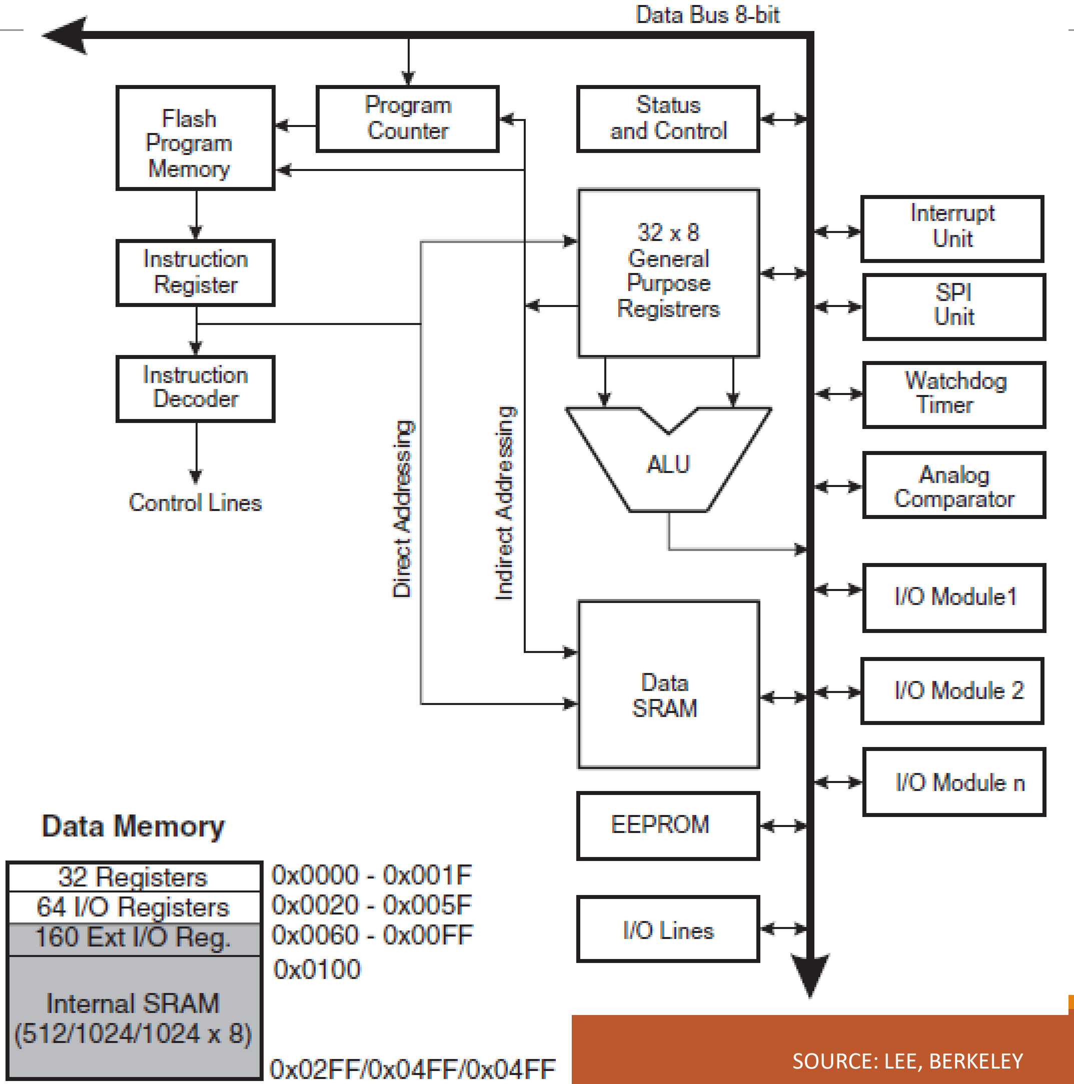


What is meant by the following C code:

```
void foo(void) {
  char *x, y;
  x = &y;
  *x = 0x20;
  …
}
```

16 bits for x and 8 bits for y are allocated on the stack, then x is loaded with the address of y, and then y is loaded with the 8-bit quantity 0x20.

Question 6



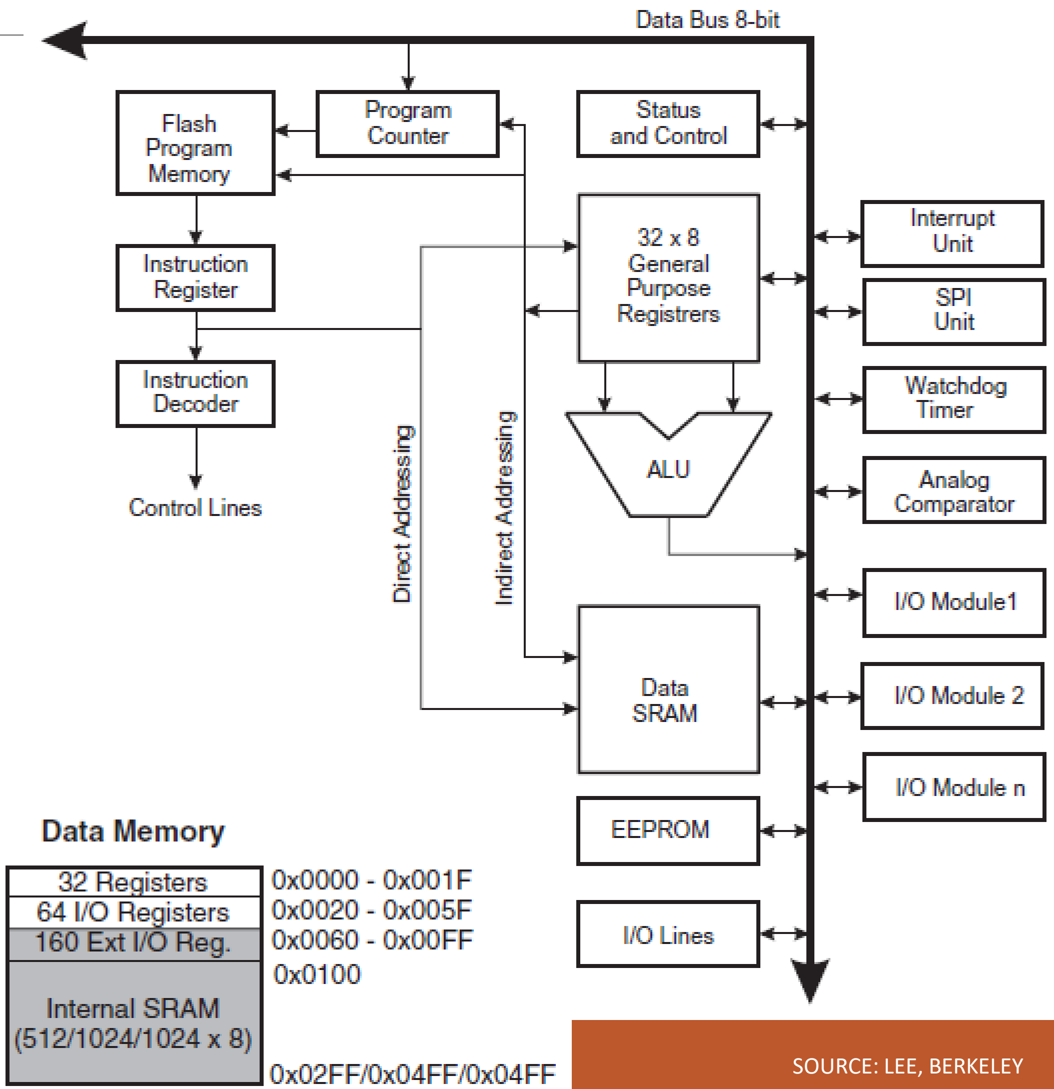
Data Memory

32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Reg.	0x0060 - 0x00FF
Internal SRAM (512/1024/1024 x 8)	0x0100
	0x02FF/0x04FF/0x04FF

What goes into z in the following program:

```
char foo() {
    char y;
    uint16_t x;
    x = 0x20;
    y = *x;
    return y;
}
char z;
int main(void) {
    z = foo();
    …
}
```

Answer 6



What goes into z in the following program:

```
char foo() {
      char y;
      uint16_t x;
      x = 0x20;
      y = *x;
      return y;
}
char z;
int main(void) {
      z = foo();
      …
}
```

z is loaded with the 8-bit quantity in the I/O register at location 0x20.

Quiz: Find the flaw in this program
(begin by thinking about where each variable is allocated)

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

Solution: Find the flaw in this program

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

statically allocated: compiler assigns a memory location.

arguments on the stack

automatic variables on the stack

program counter, argument 10, and z go on the stack (and possibly more, depending on the compiler).

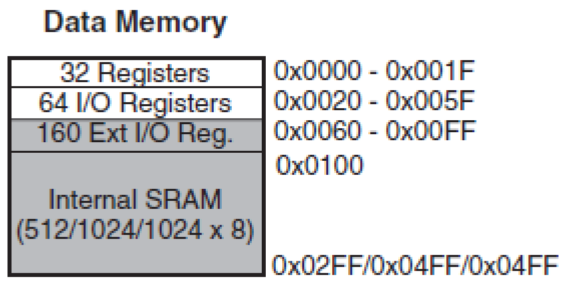
The procedure foo() returns a pointer to a variable on the stack. What if another procedure call (or interrupt) occurs before the returned pointer is de-referenced?

Dynamically-Allocated Memory
The Heap

An operating system typically offers a way to dynamically allocate memory on a "heap".

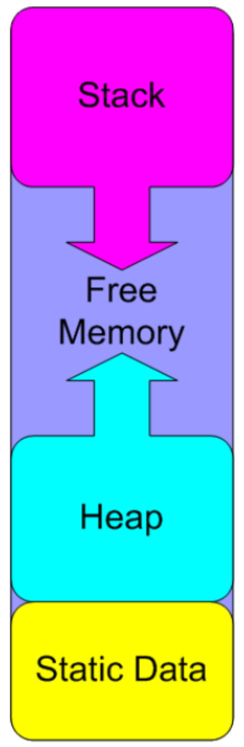
Memory management (malloc() and free()) can lead to many problems with embedded systems:

- Memory leaks (allocated memory is never freed)
- Memory fragmentation (allocatable pieces get smaller)

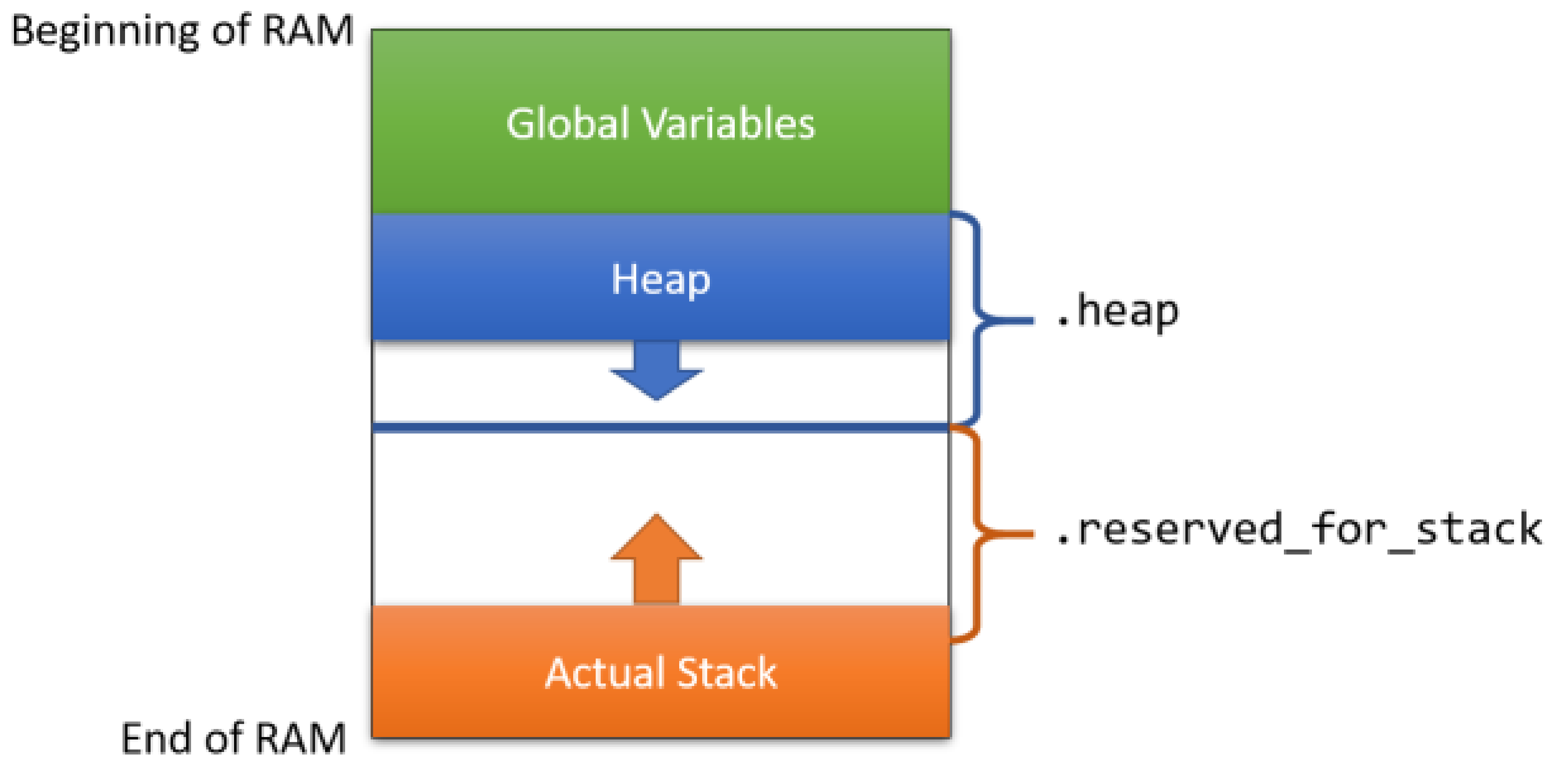


Automatic techniques ("garbage collection") often require stopping everything and reorganizing the allocated memory. This is deadly for real-time programs.

Stack vs Heap



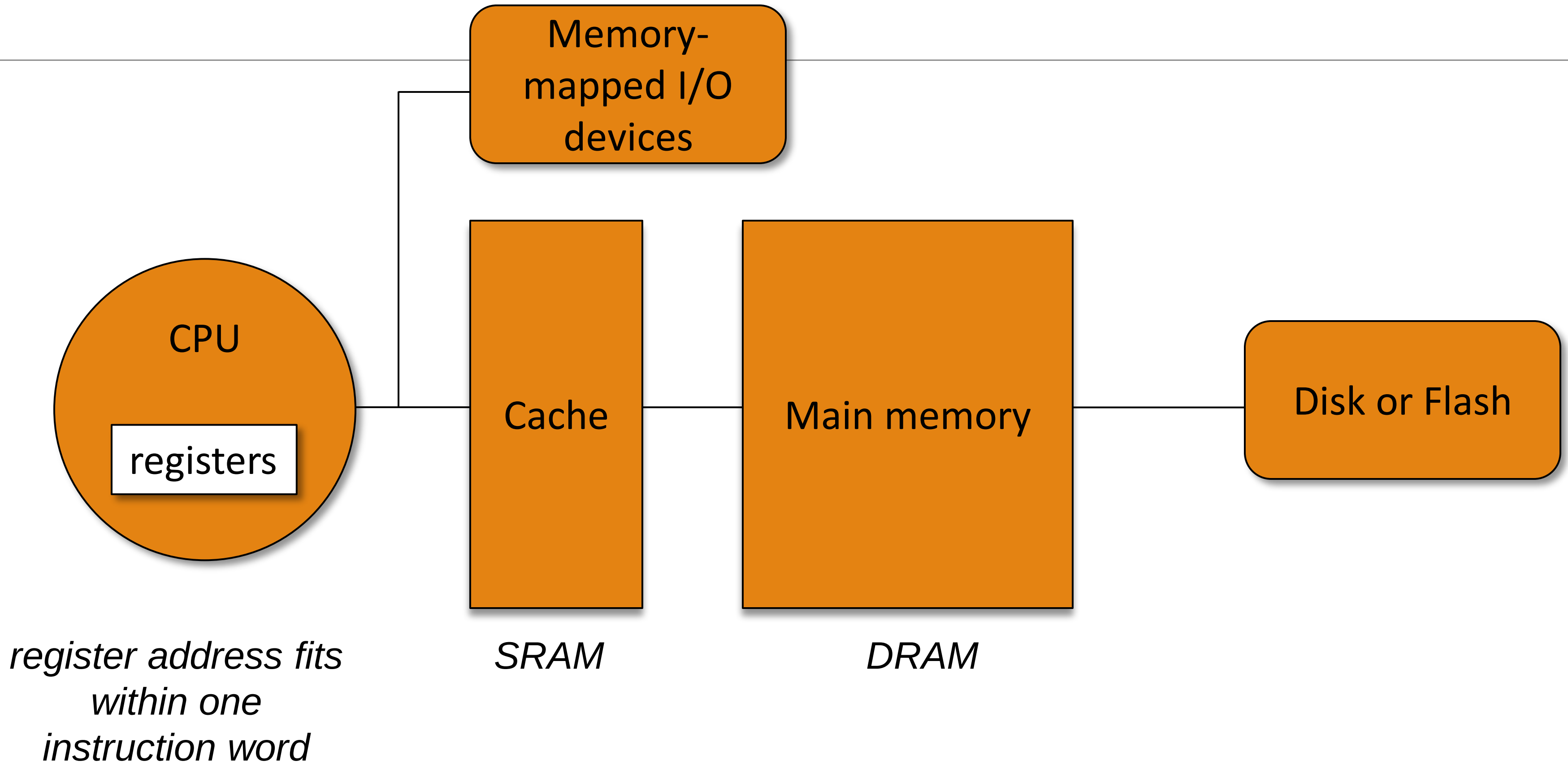
Normal SRAM Operation



Memory Hierarchies

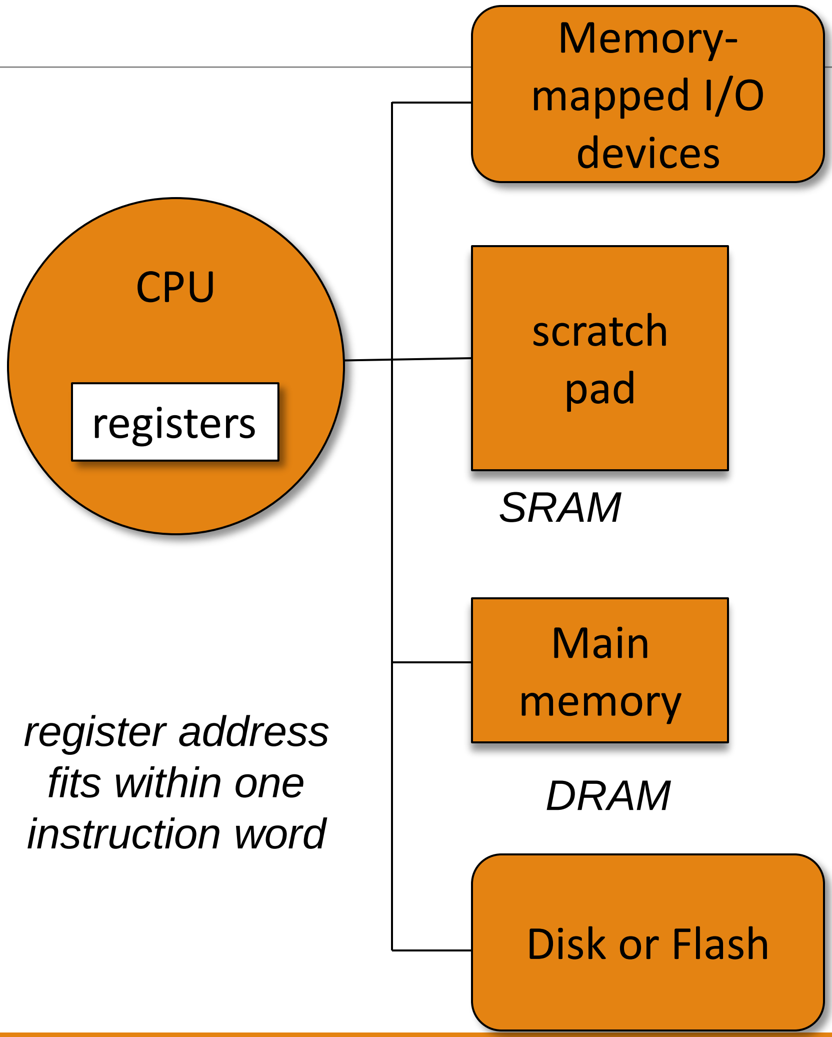
- Memory hierarchy
  - Cache:
    - A subset of memory addresses is mapped to SRAM
    - Accessing an address not in SRAM results in *cache miss*
    - A miss is handled by copying contents of DRAM to SRAM
  - Scratchpad:
    - SRAM and DRAM occupy disjoint regions of memory space
    - Software manages what is stored where
- Segmentation
    - Logical addresses are mapped to a subset of physical addresses
    - Permissions regulate which tasks can access which memory

Memory Hierarchy


Memory-mapped I/O devices
CPU
registers
Cache
Main memory
Disk or Flash
register address fits within one instruction word
SRAM
DRAM

Here, the cache or scratchpad, main memory, and disk or flash share the same address space.

Memory Hierarchy

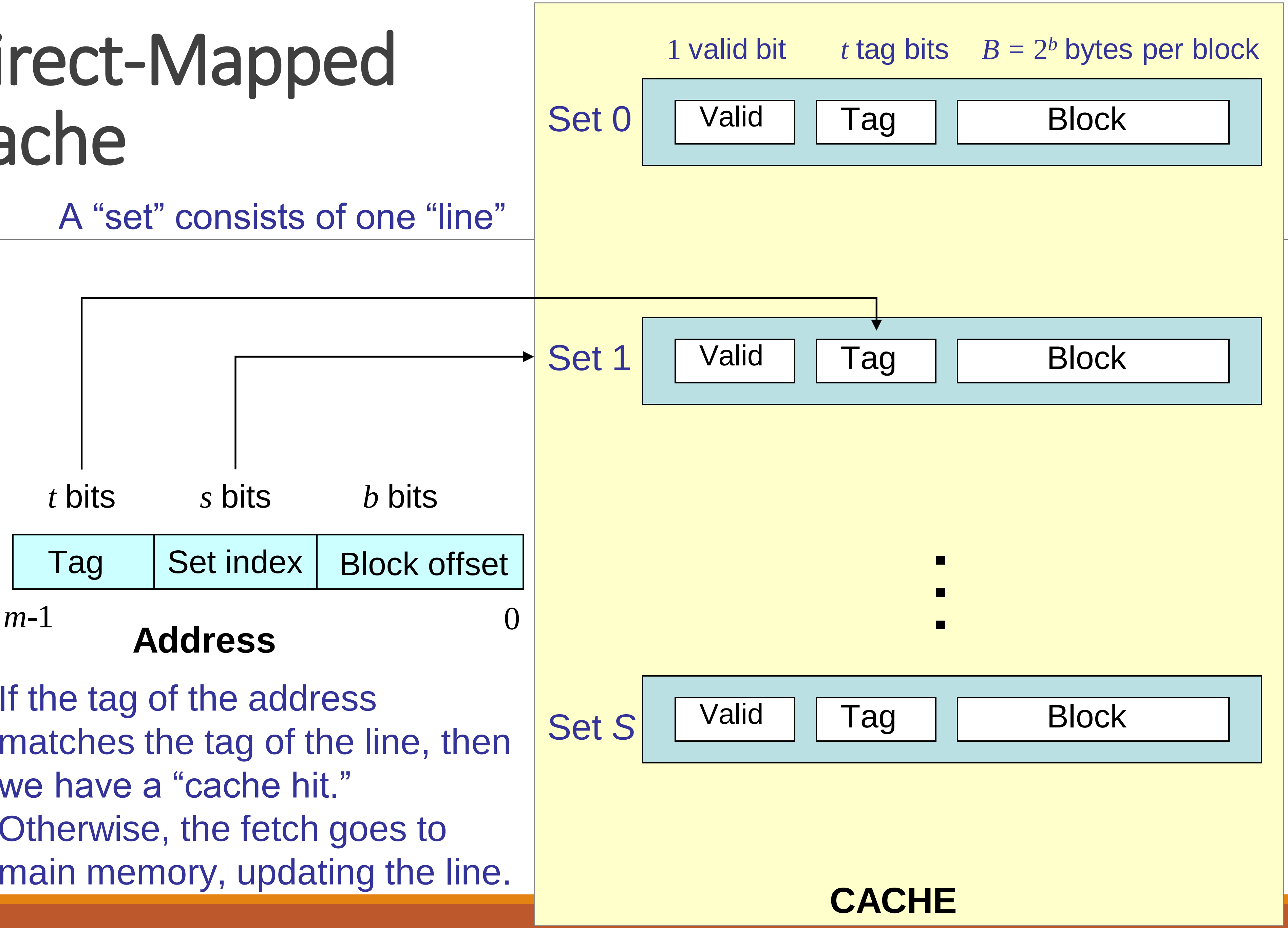

Memory-mapped I/O devices
CPU
registers
scratch pad
SRAM
Main memory
DRAM
register address fits within one instruction word
Disk or Flash

Here, each distinct piece of memory hardware has its own segment of the address space.

This requires more careful software design, but gives more direct control over timing.

Direct-Mapped Cache

A "set" consists of one "line"

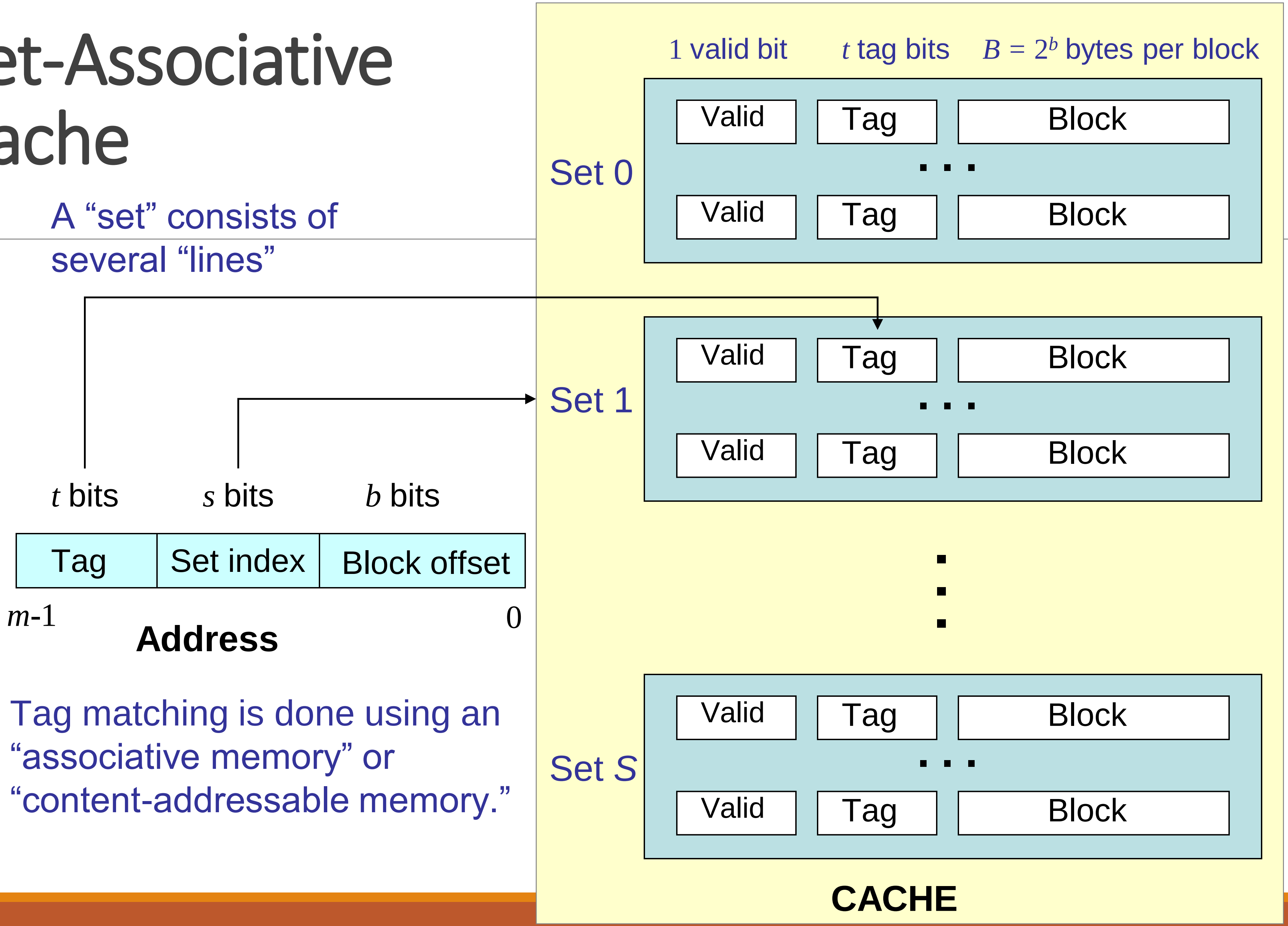


If the tag of the address matches the tag of the line, then we have a "cache hit." Otherwise, the fetch goes to main memory, updating the line.

SOURCE: LEE, BERKELEY

Set-Associative Cache

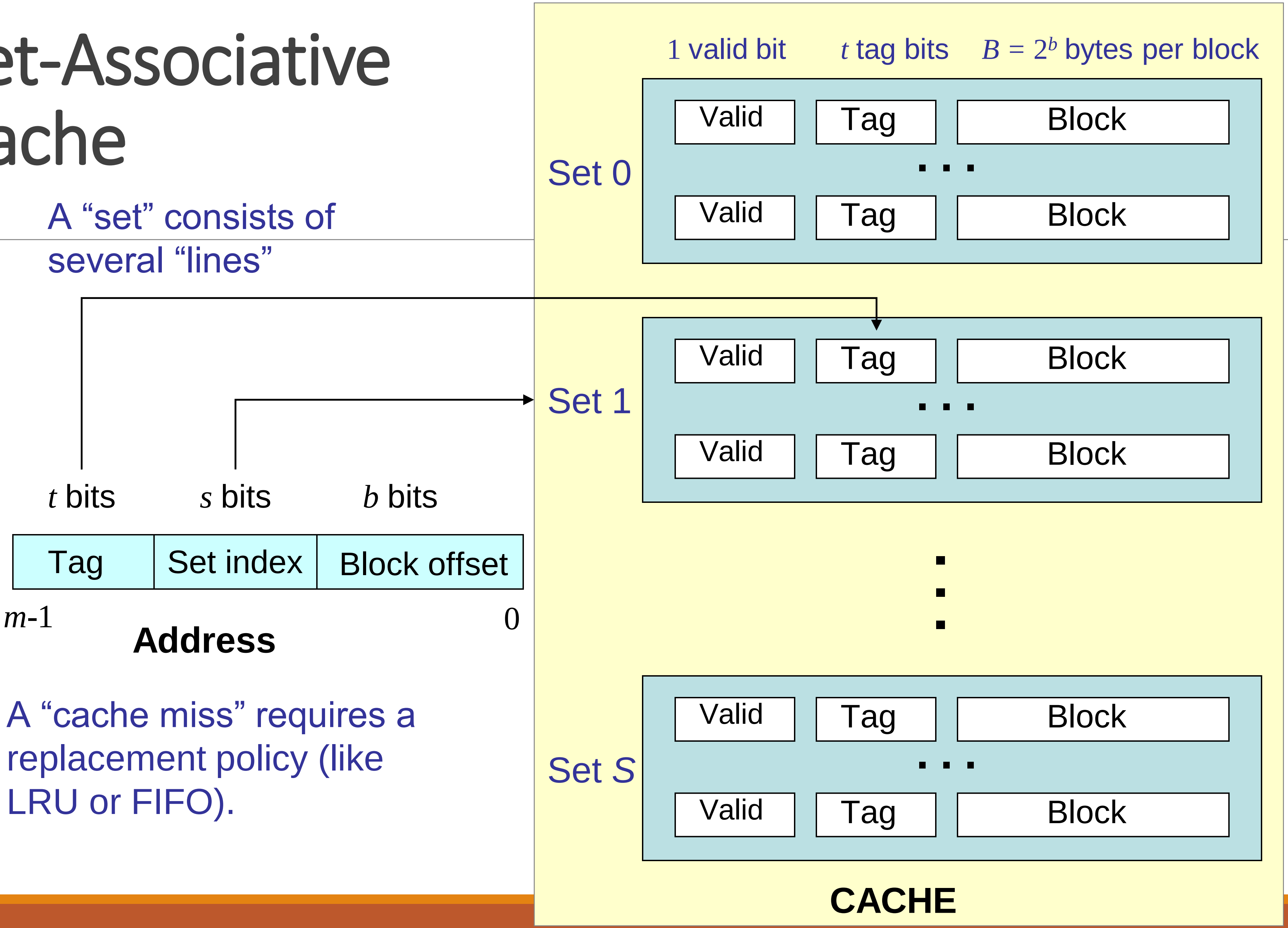
A “set” consists of several “lines”



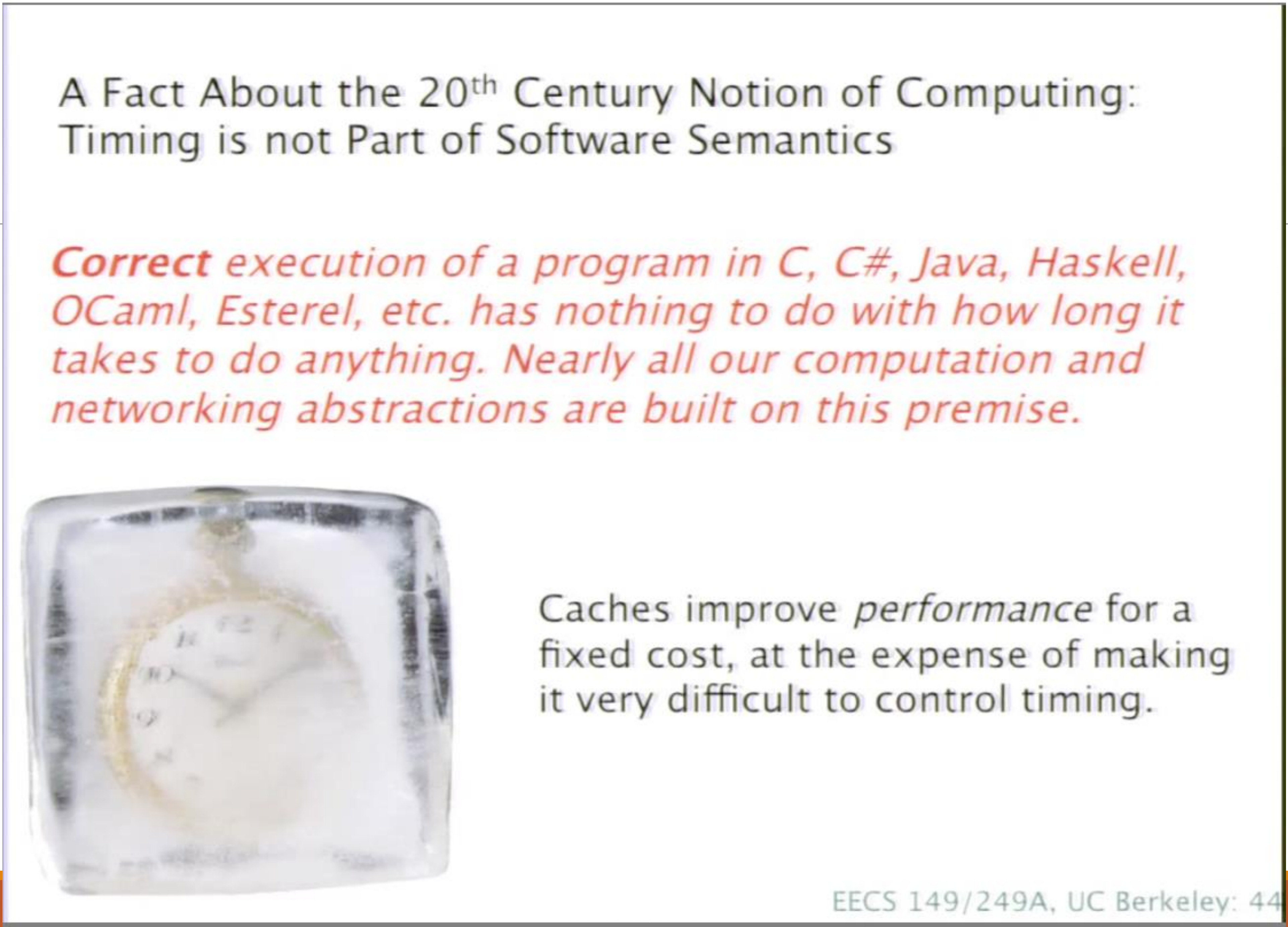
Tag matching is done using an “associative memory” or “content-addressable memory.”

Set-Associative Cache

A “set” consists of several “lines”



A “cache miss” requires a replacement policy (like LRU or FIFO).



A Fact About the 20th Century Notion of Computing: Timing is not Part of Software Semantics

***Correct** execution of a program in C, C#, Java, Haskell, OCaml, Esterel, etc. has nothing to do with how long it takes to do anything. Nearly all our computation and networking abstractions are built on this premise.*

Caches improve *performance* for a fixed cost, at the expense of making it very difficult to control timing.

Conclusion

Understanding memory architectures is essential to programming embedded systems.

Reference

- Lee, Edward & Seshia, Sanjit. (2011). Introduction to Embedded Systems - A Cyber-Physical Systems Approach.
- Lecture Note Slides from EECS 149/249A: Introduction to Embedded Systems (UC Berkeley) by Prof. Prabal Dutta and Sanjit A. Seshia
- http://www.weigu.lu/tutorials/microcontroller/04_memory/index.html