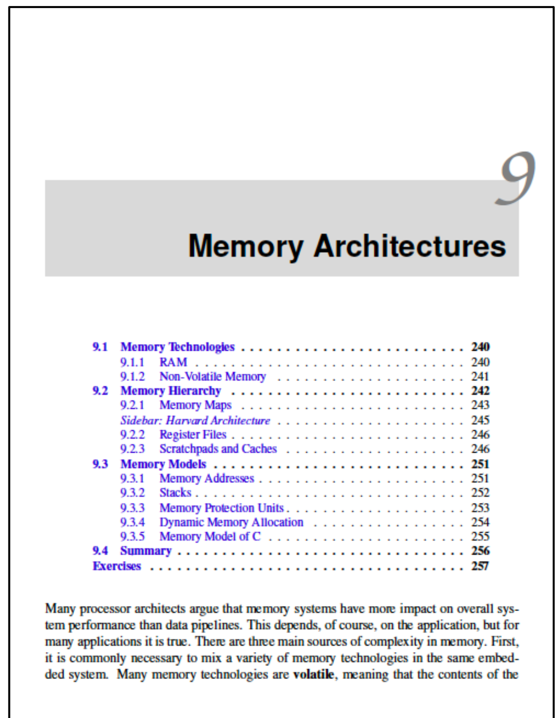
Lecture 6 : Memory Architectures

Slides were originally developed by Profs. Edward Lee and Sanjit Seshia, and subsequently updated by Profs. Gavin Buskes and Iman Shames.

Outline

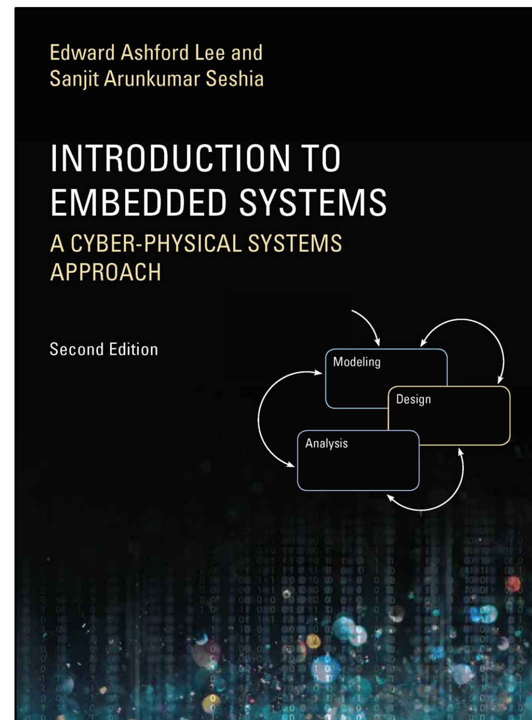
- Types of memory
- Memory maps
- Memory organisation
- Memory model of C
- Memory hierarchies
- Memory protection



9

Memory Architectures

Many processor architects argue that memory systems have more impact on overall system performance than data pipelines. This depends, of course, on the application, but for many applications it is true. There are three main sources of complexity in memory. First, it is commonly necessary to mix a variety of memory technologies in the same embedded system. Many memory technologies are **volatile**, meaning that the contents of the



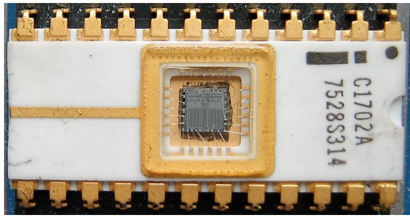
Role of Memory in Embedded Systems

- 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

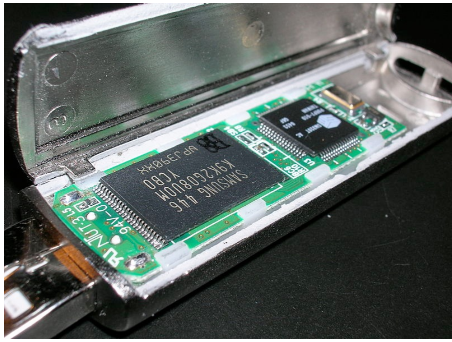
Non-Volatile Memory

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

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

Example:

Die of a STM32F103VGT6 ARM Cortex-M3 microcontroller with 1 megabyte flash memory by STMicroelectronics.

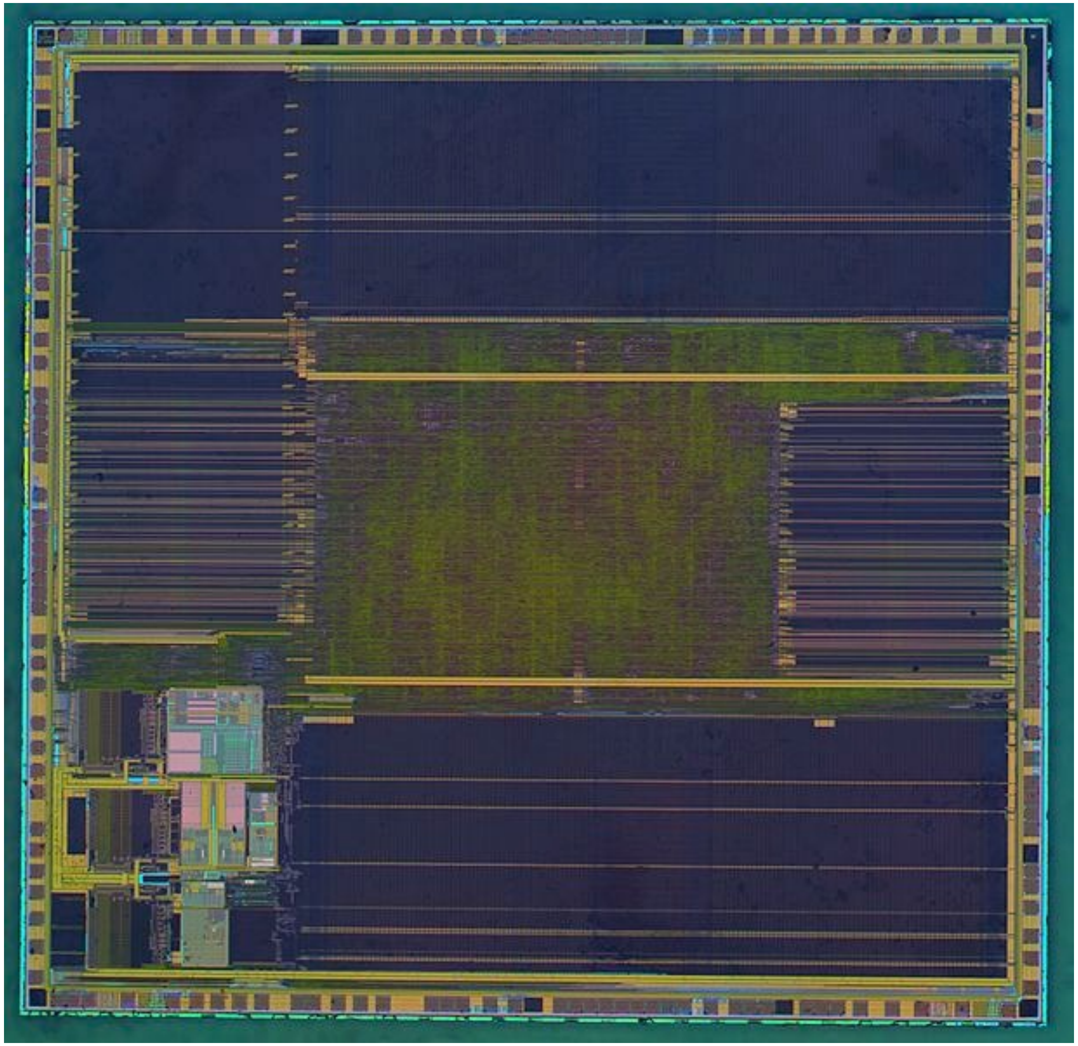


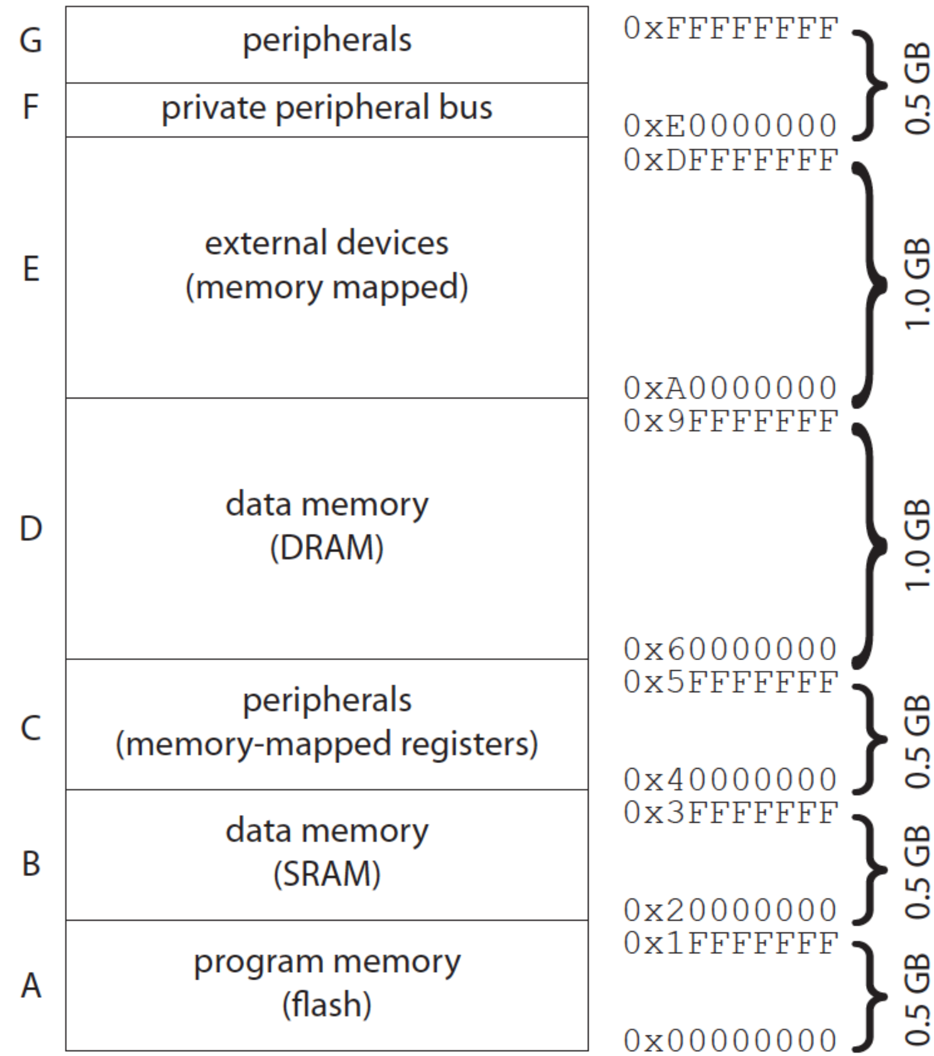
Image from the Wikimedia Commons

Memory Map in ARM Cortex-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!



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 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

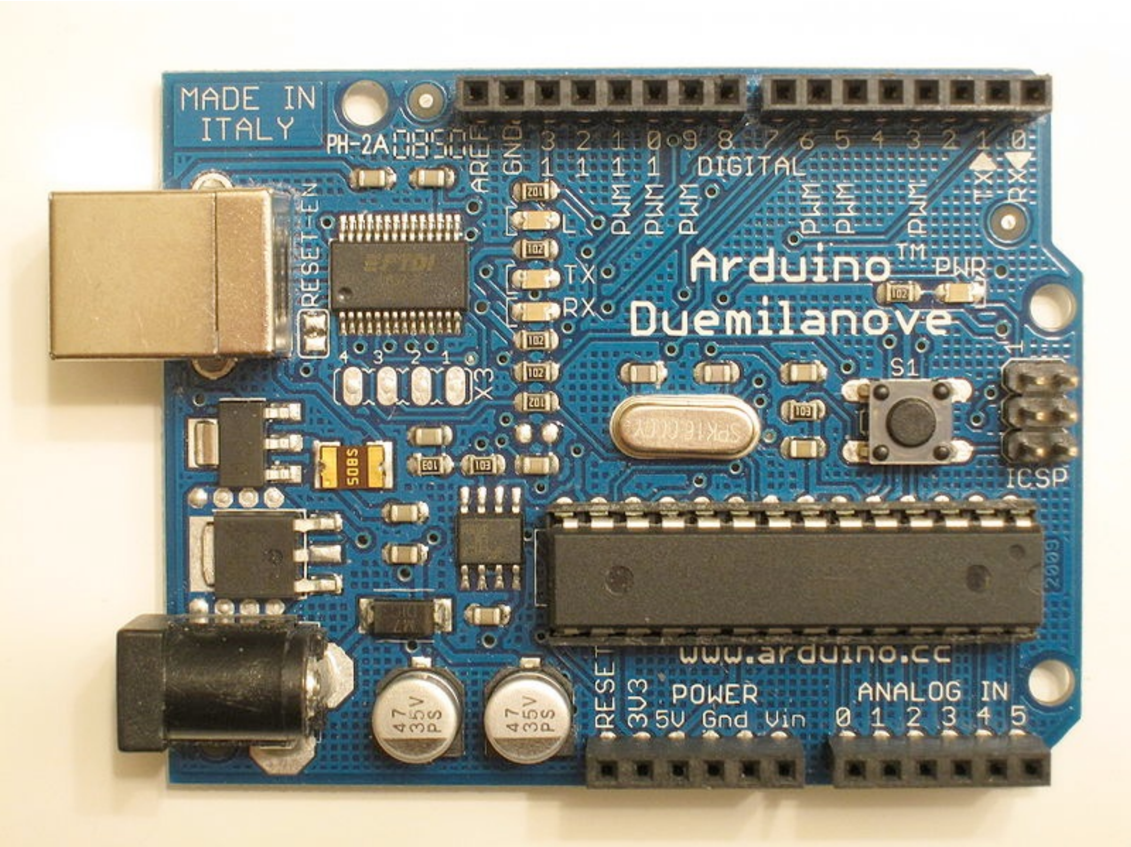
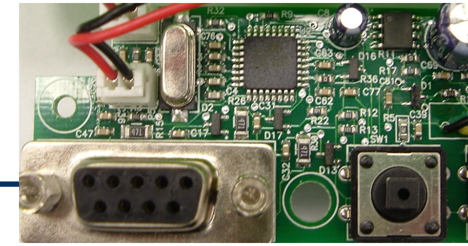
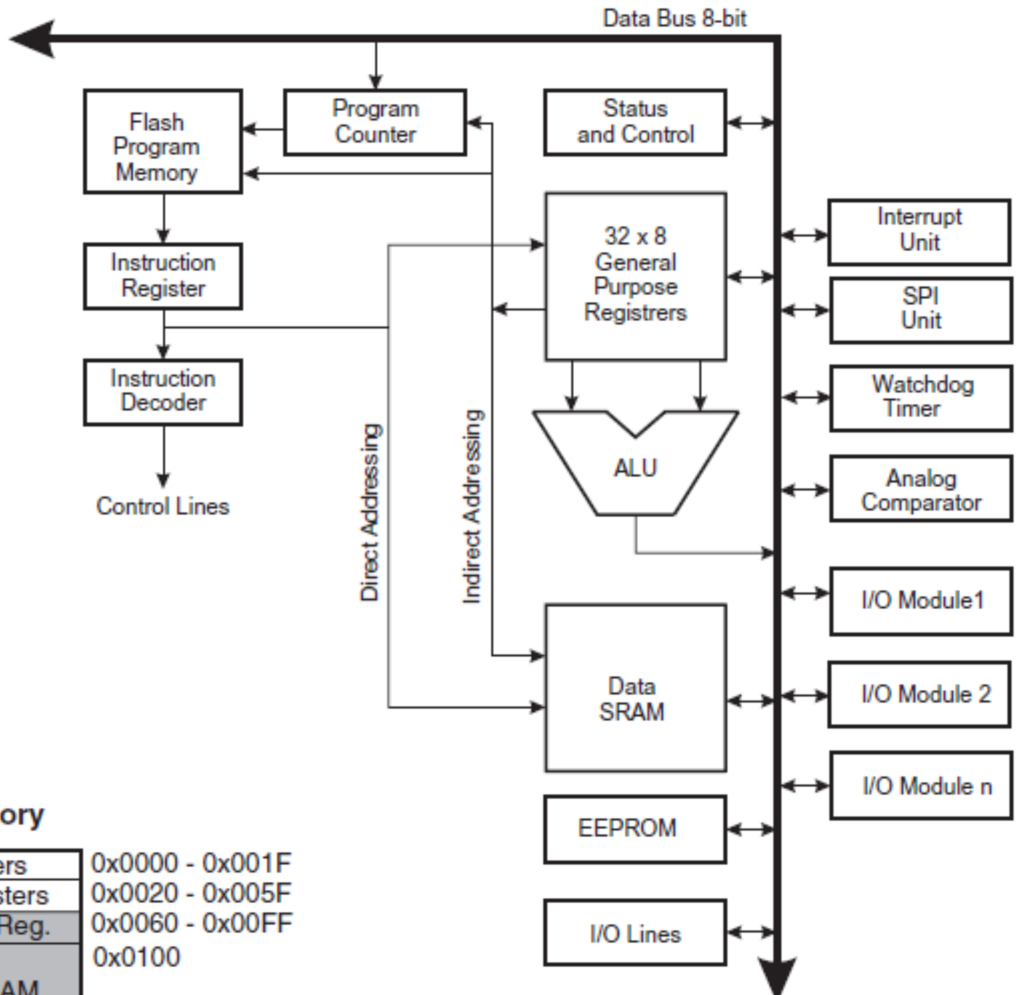


Image from the Wikimedia Commons

ATMega168 Memory Architecture

An 8-bit microcontroller with 16-bit addresses





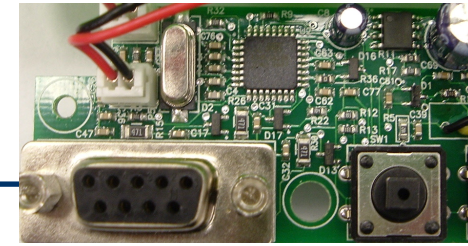
Why is it called an 8-bit microcontroller?

Data Memory

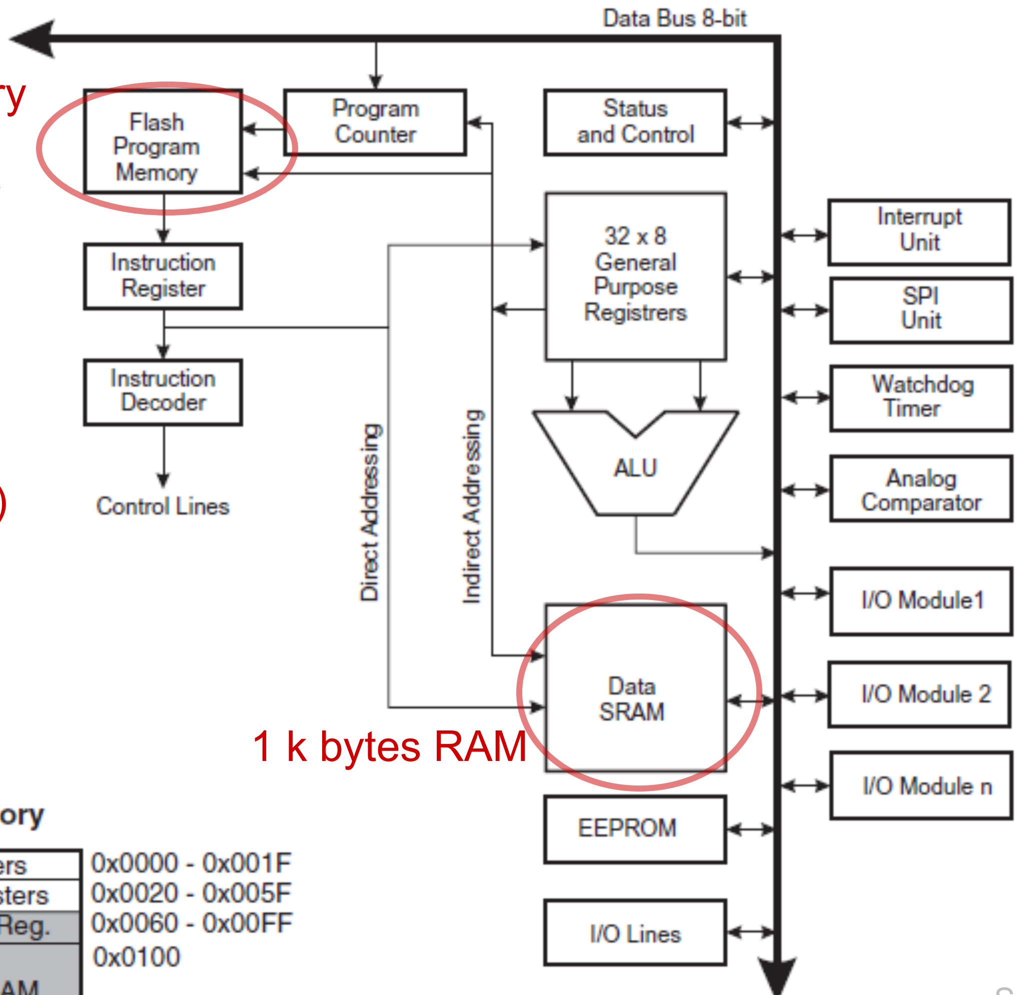
32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Reg.	0x0060 - 0x00FF
Internal SRAM (512/1024/1024 x 8)	0x0100
	0x02FF/0x04FF/0x04FF

ATMega168 Memory Architecture

An 8-bit microcontroller with 16-bit addresses



16K bytes flash memory (14,336 available for the user program. Includes interrupt vectors and boot loader.)



1 k bytes RAM

The “8-bit data” is why this is called an “8-bit microcontroller.”

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

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

Source: ATmega168 Reference Manual

Memory Organisation 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
- Heap
  - Dynamically allocated memory

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.

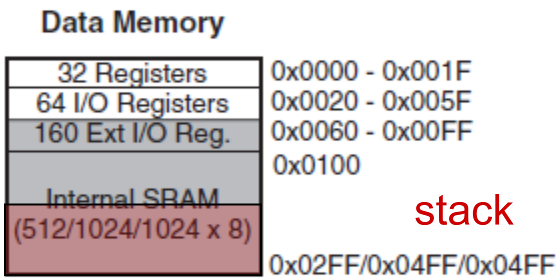
Statically-Allocated Memory with Limited Scope

```
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")

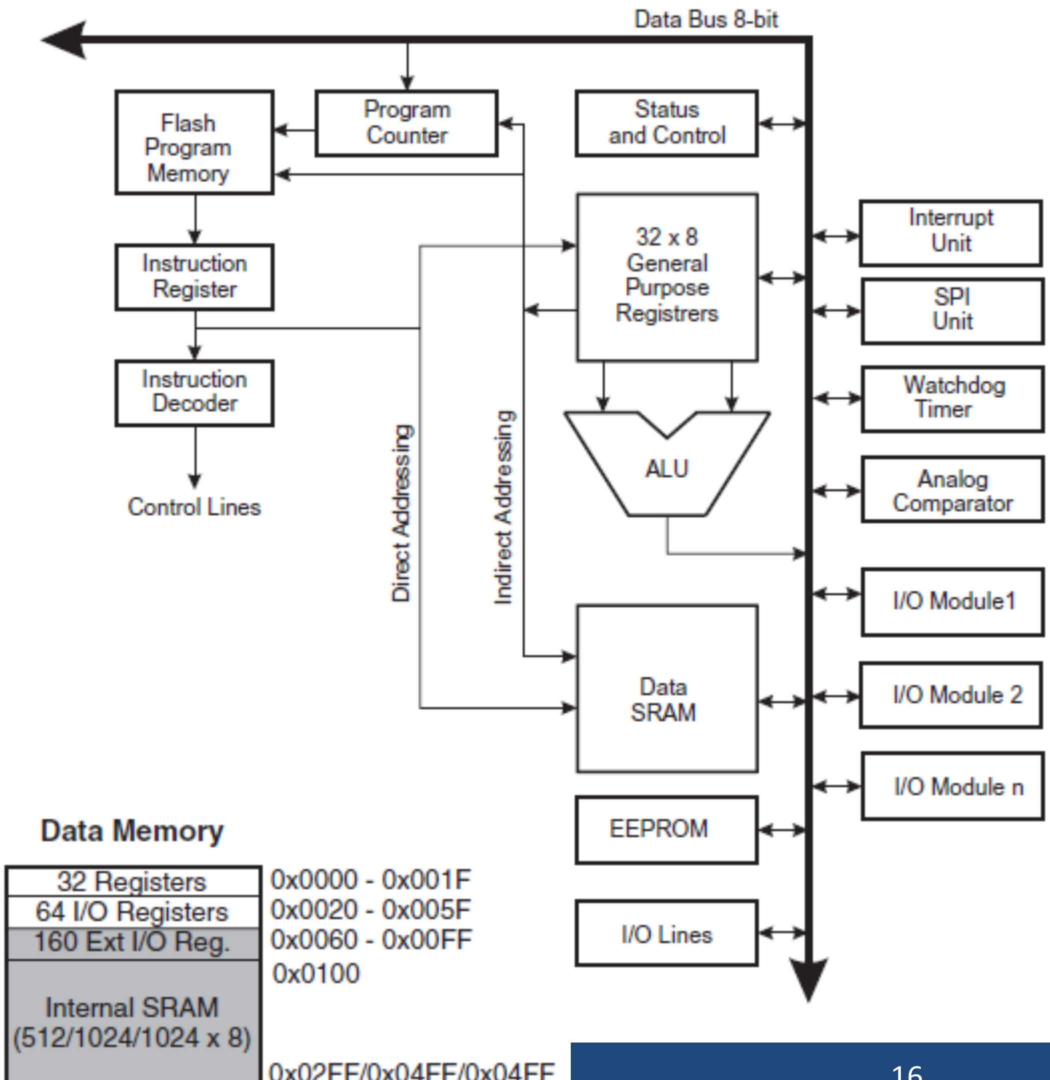
```
void foo(void) {
   char x;
   x = 0x20;
   …
}
```



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().

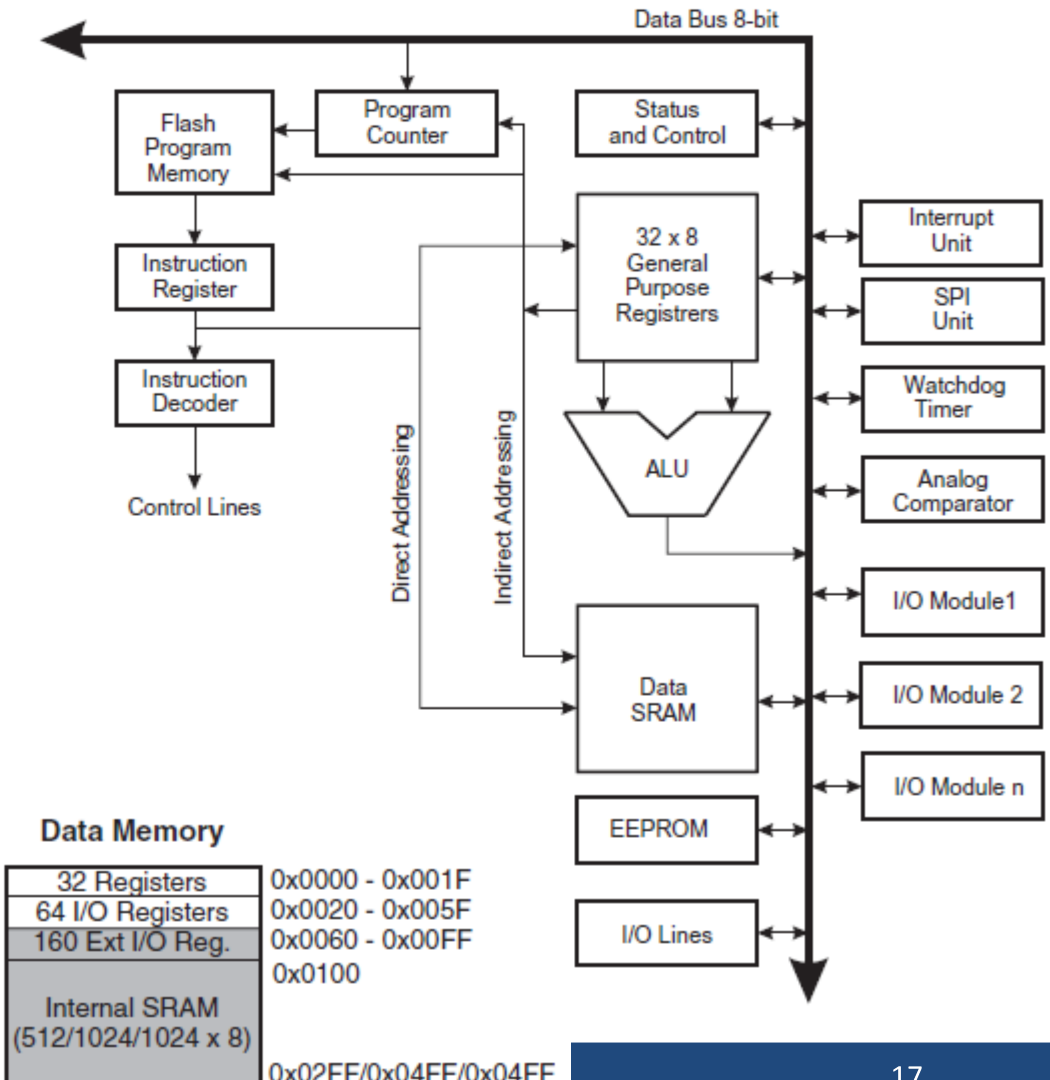
Question 1



What is meant by the following C code:

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

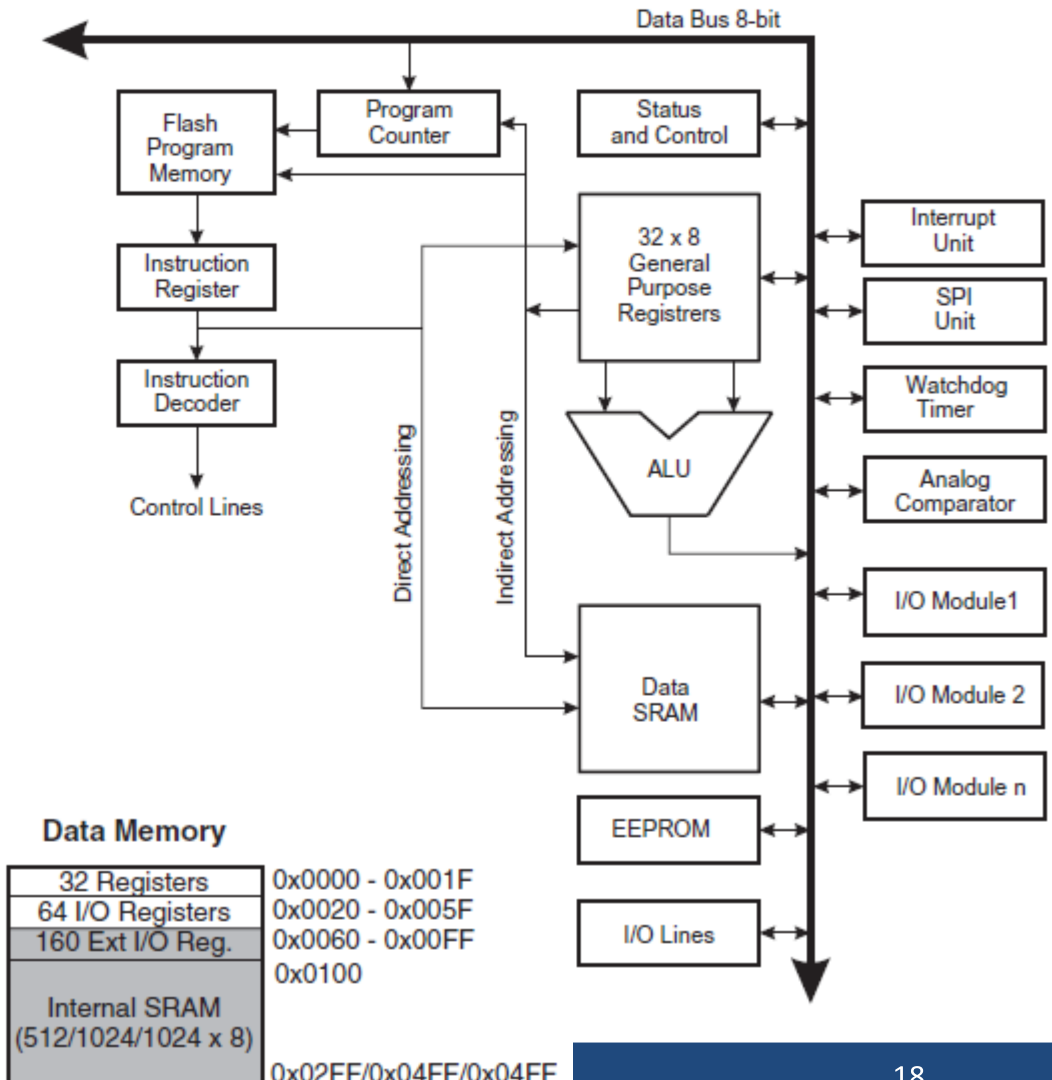
Answer 1



What is meant by the following C code:

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

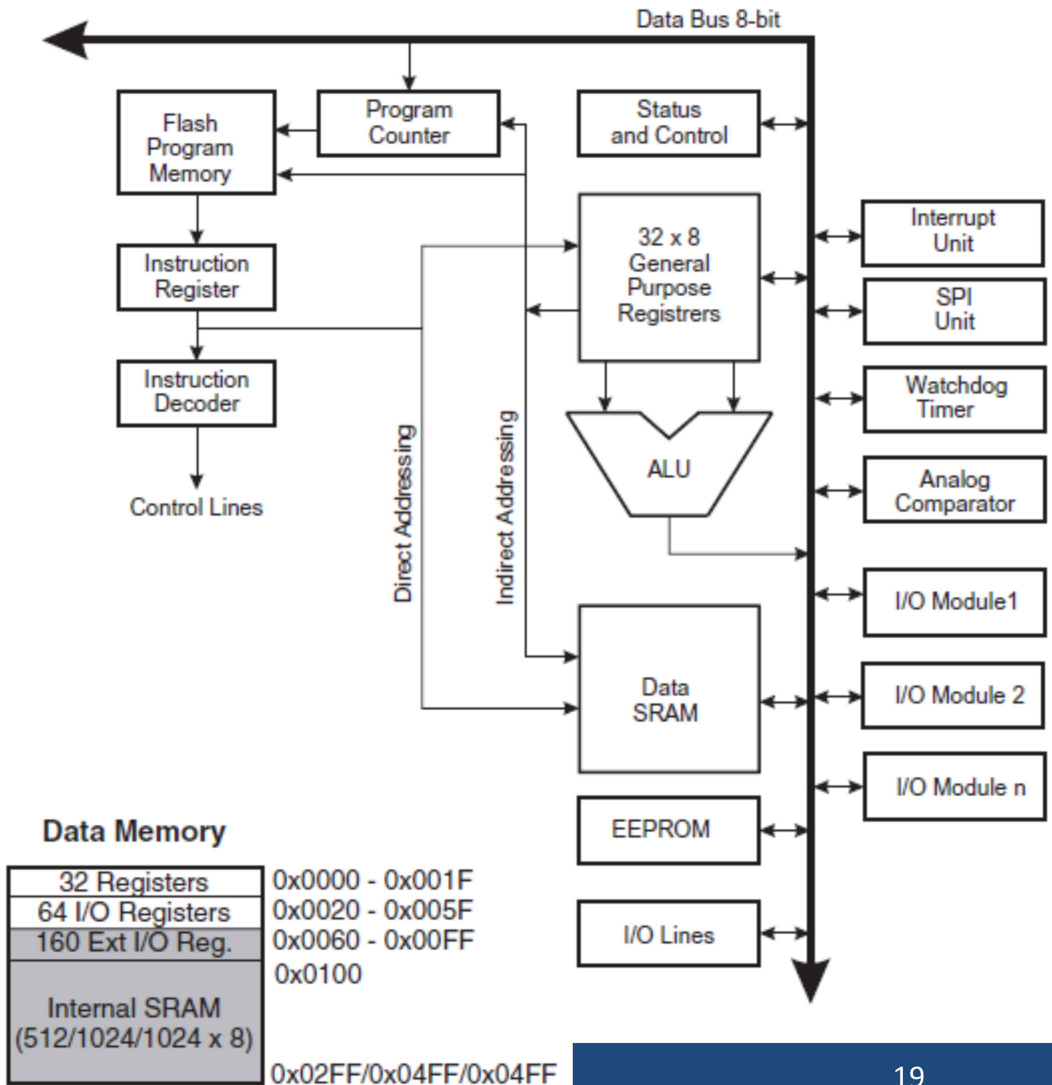
Question 2



What is meant by the following C code:

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

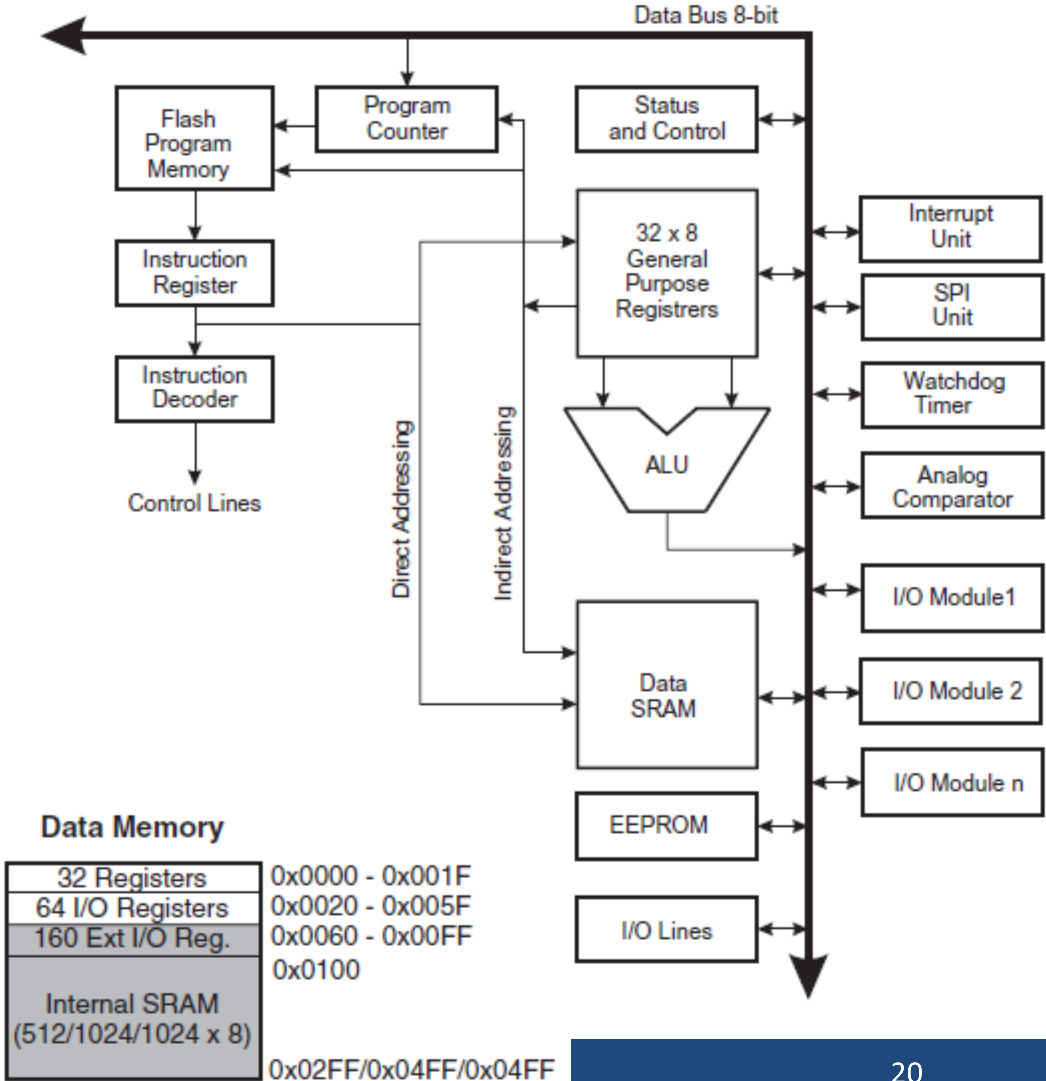
Answer 2



What is meant by the following C code:

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

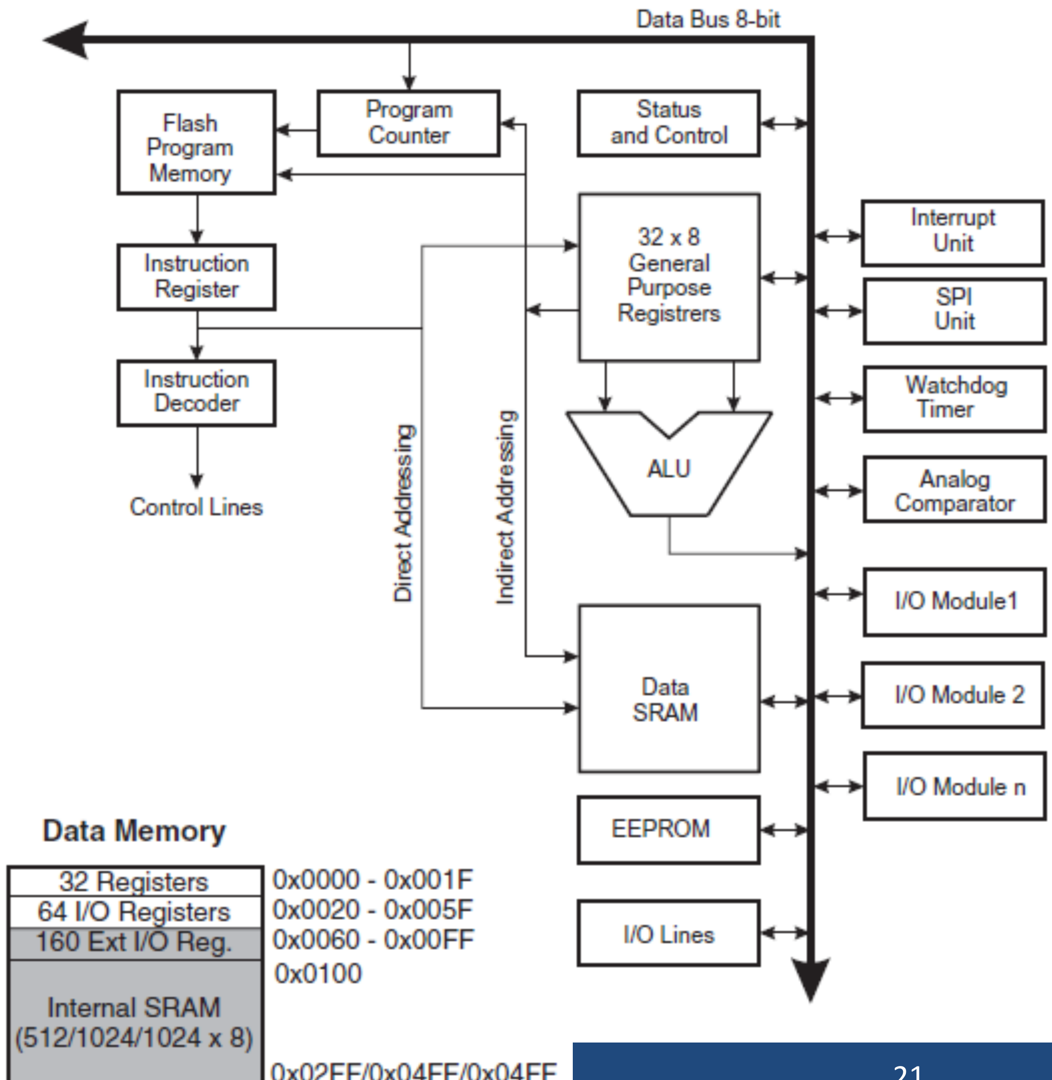
Question 3



What is meant by the following C code:

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

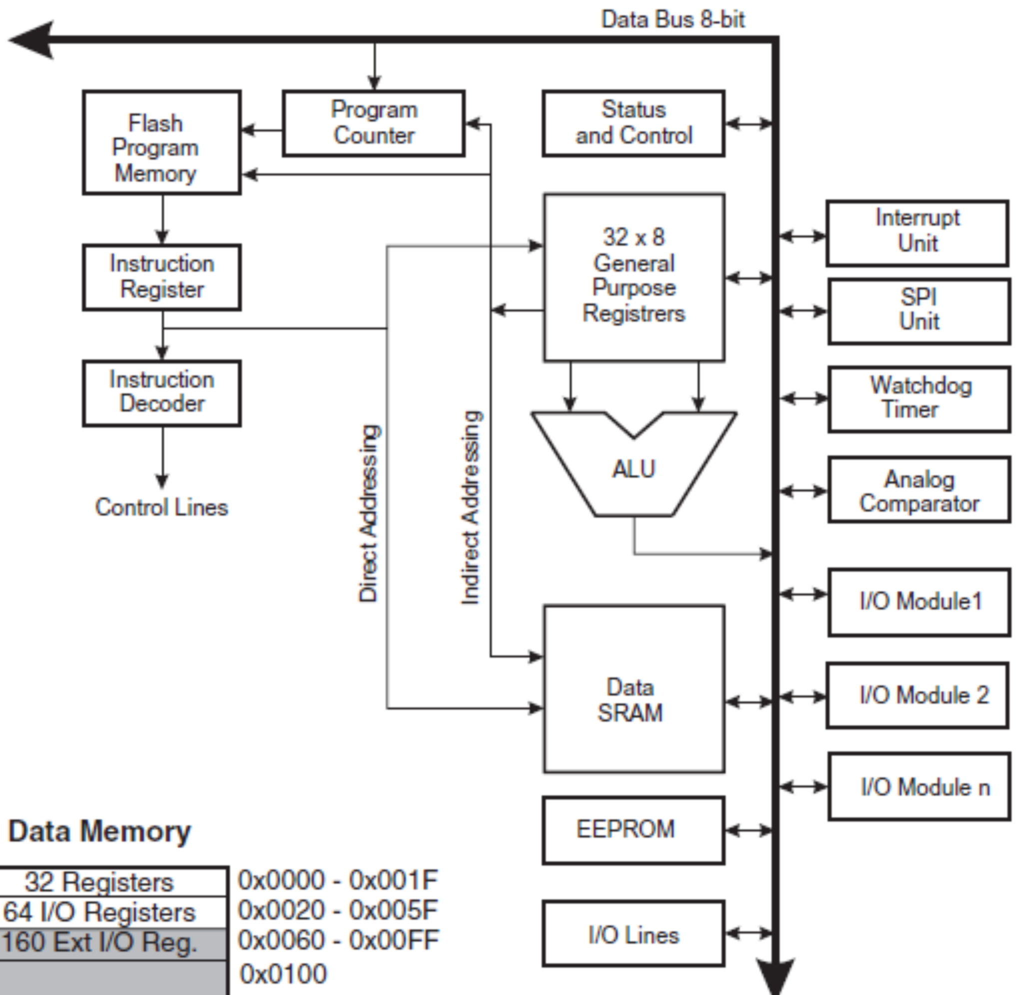
Answer 3



What is meant by the following C code:

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

Question 4



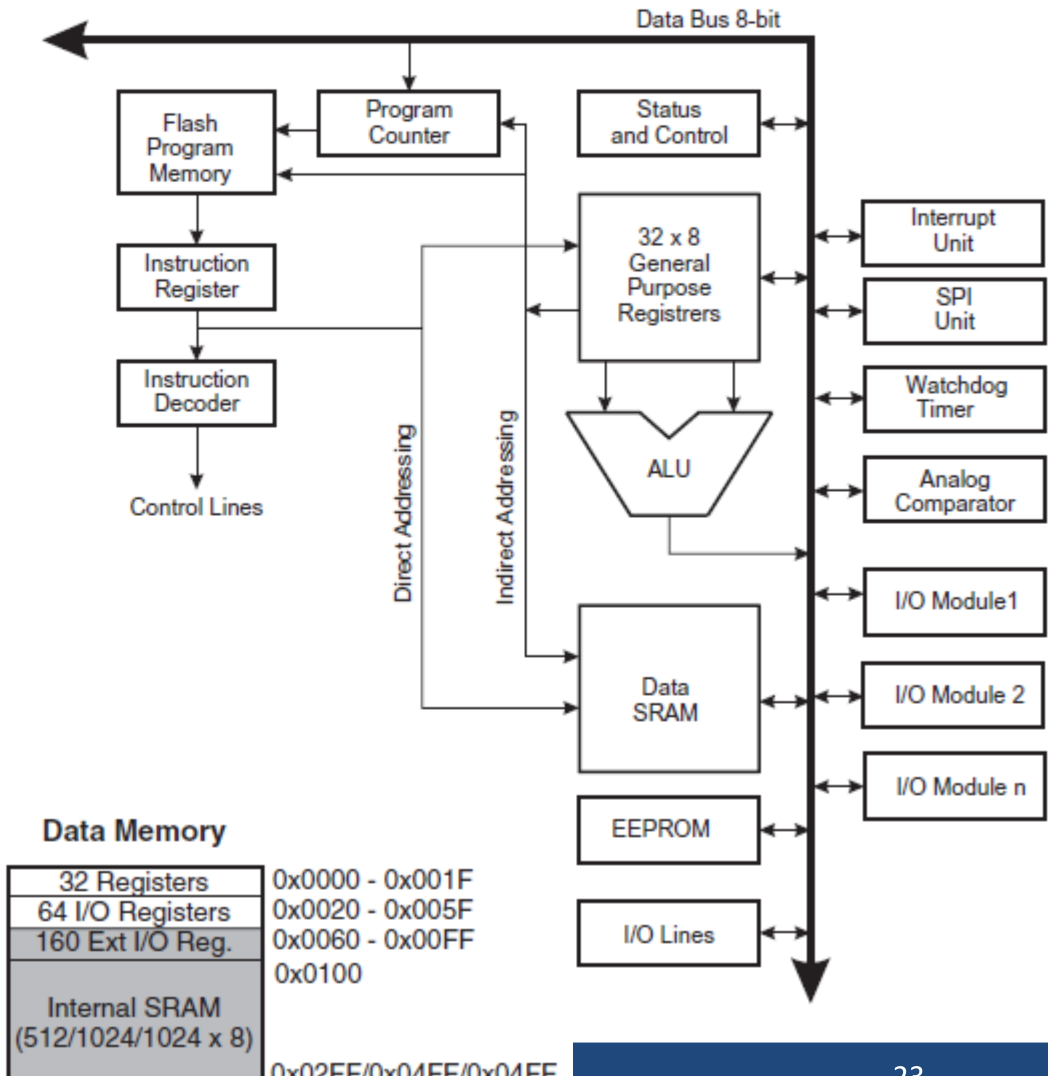
Data Memory

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

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

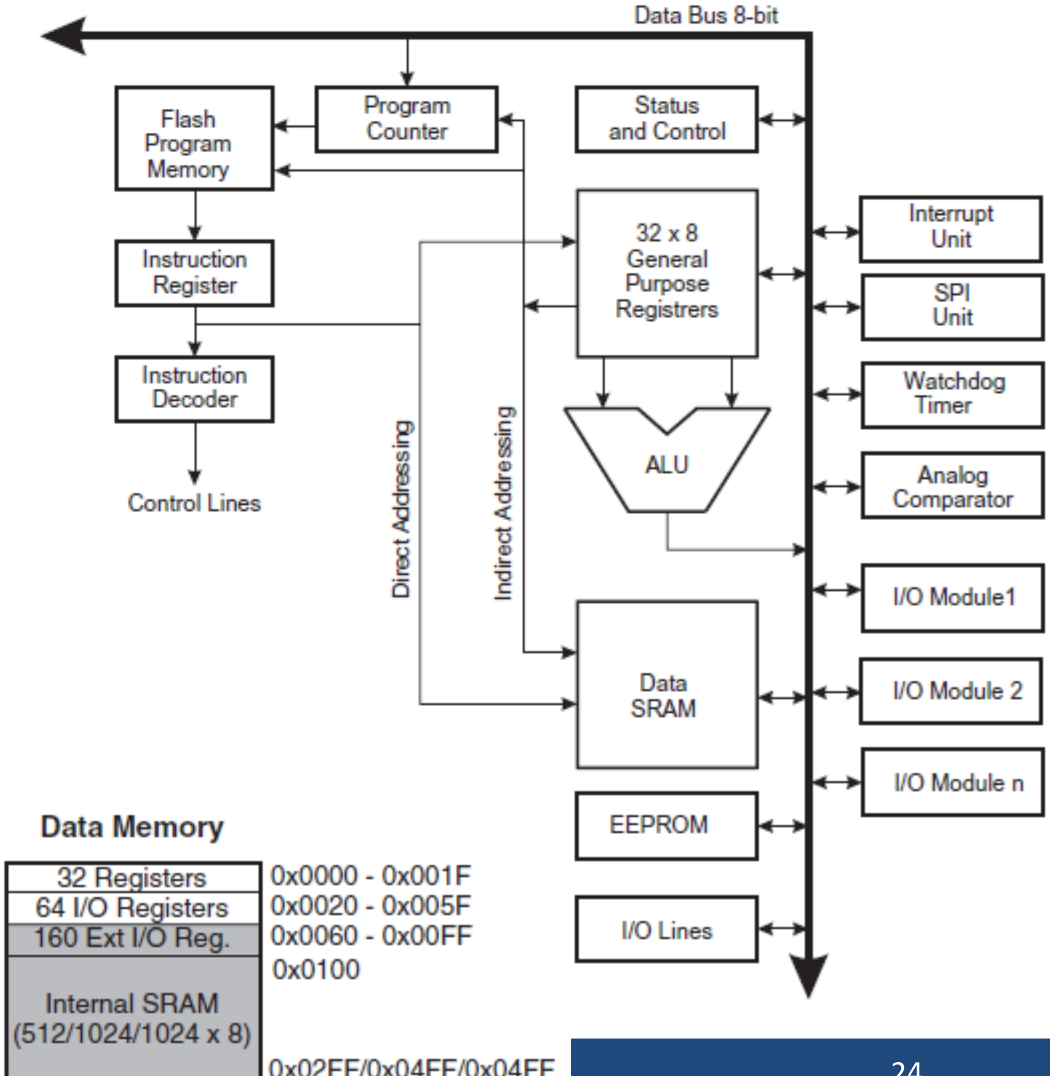
Where are x, y, z in memory?

Answer 4




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

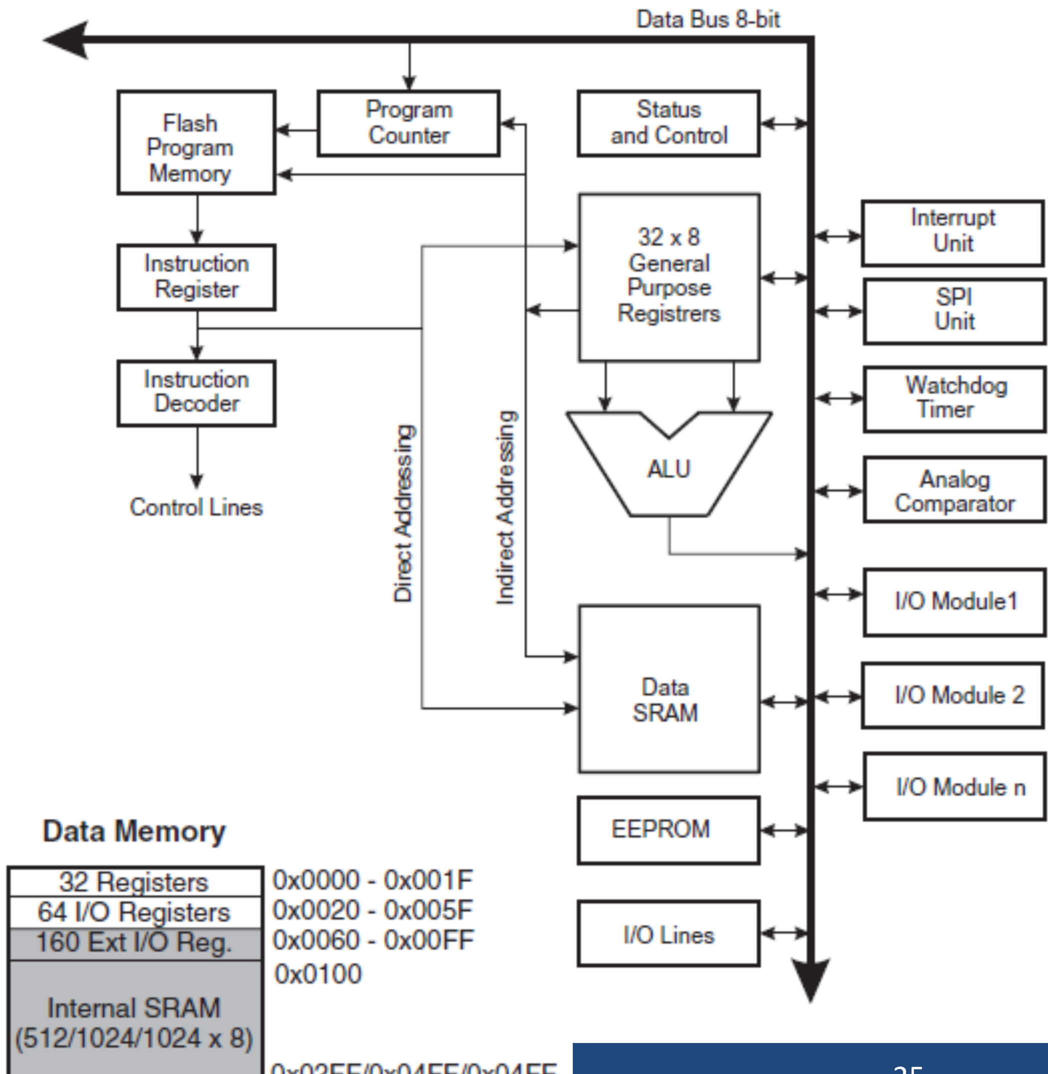
Question 5



What is meant by the following C code:

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

Answer 5

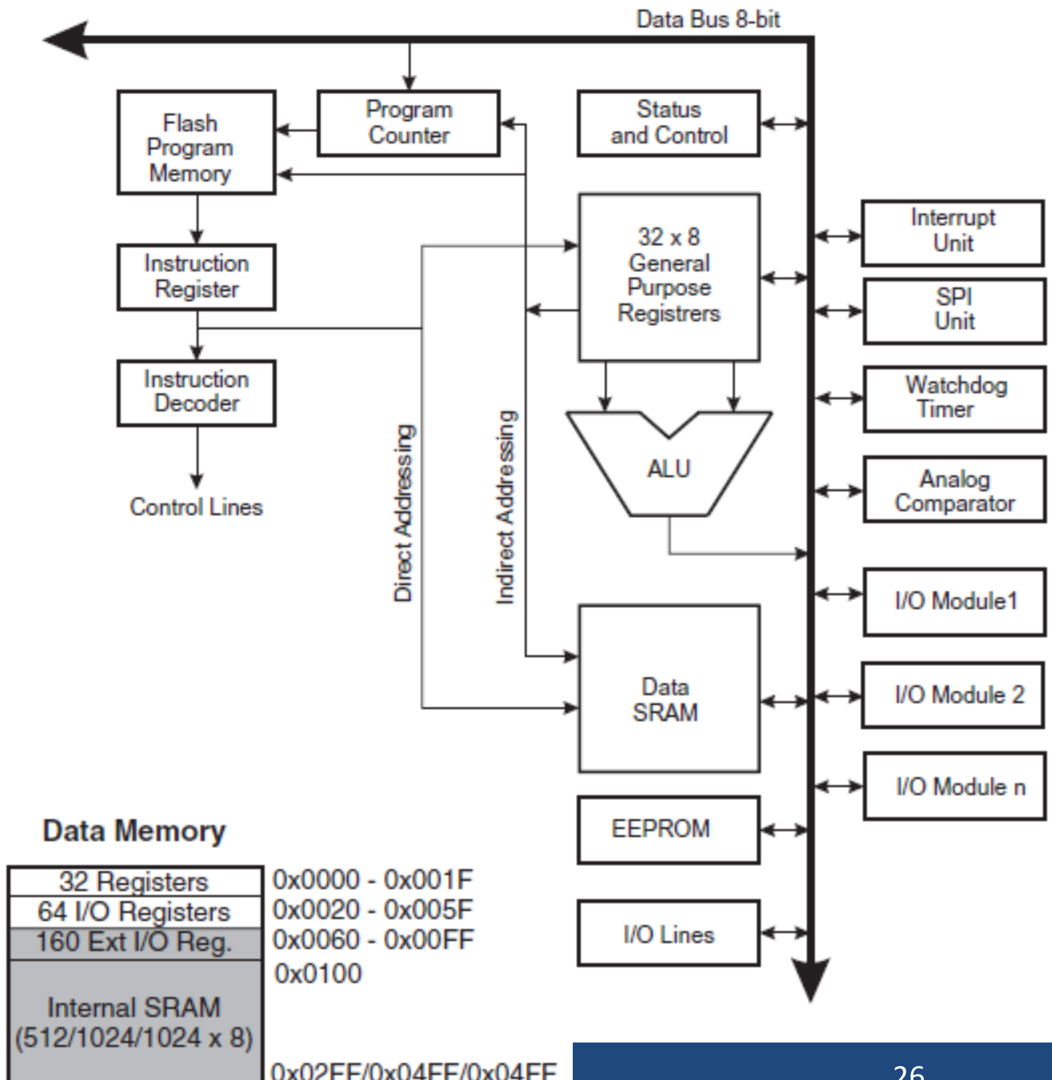


What is meant by the following C code:

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

Question 6

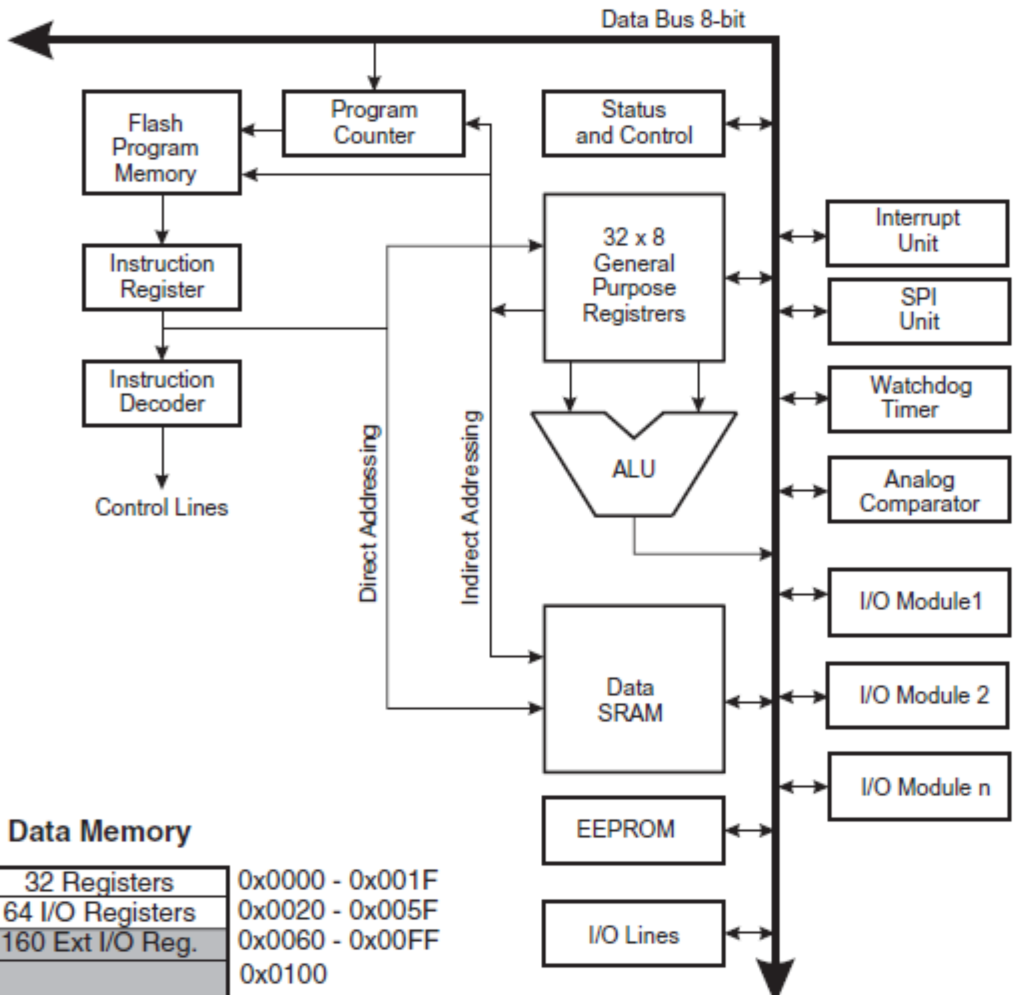
What goes into z in the following program:




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

Answer 6

What goes into z in the following program:



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

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

Quiz: 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);
  ...
}
```

Begin by thinking about where each variable is allocated

Answer: 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);
  ...
}
```

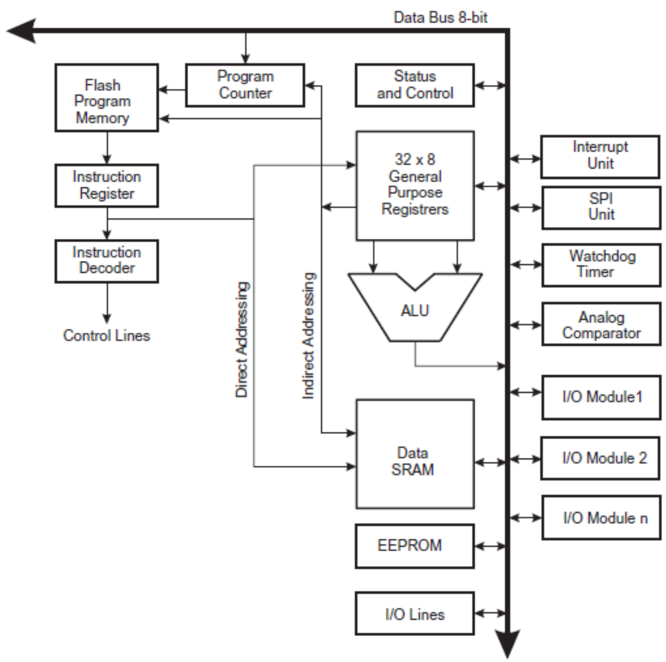
statically allocated: compiler assigns a memory location.

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).

Watch out for Recursion!!
Quiz: What is the Final Value of z?

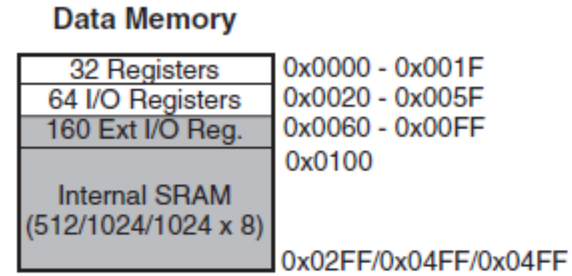


Data Memory

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

```
void foo(uint16_t x) {
 char y;
 y = *x;
 if (x > 0x100) {
      foo(x - 1);
 }
}
char z;
void main(…) {
 z = 0x10;
 foo(0x04FF);
 …
 }
```

Dynamically-Allocated Memory
The Heap



- An operating system typically offers a way to dynamically allocate memory on a “heap”.

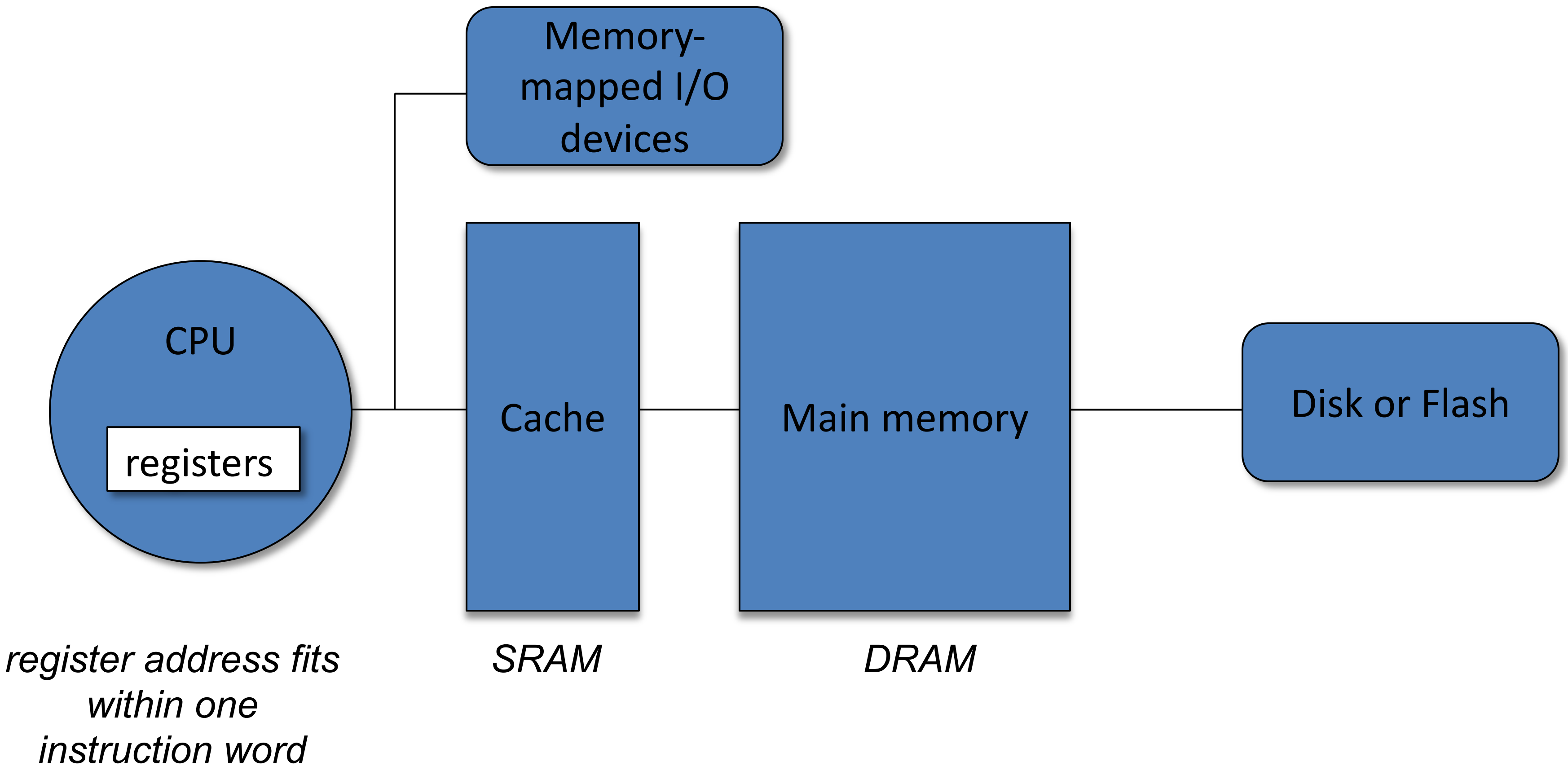
- Memory management (malloc() and free()) can lead to many problems with embedded systems:
  - Memory leaks (allocated memory is never freed)
  - Memory fragmentation (allocatable pieces get smaller)

- Automatic techniques (“garbage collection”) often require stopping everything and reorganising the allocated memory. This is <u>deadly</u> for real-time programs.

Memory Hierarchies

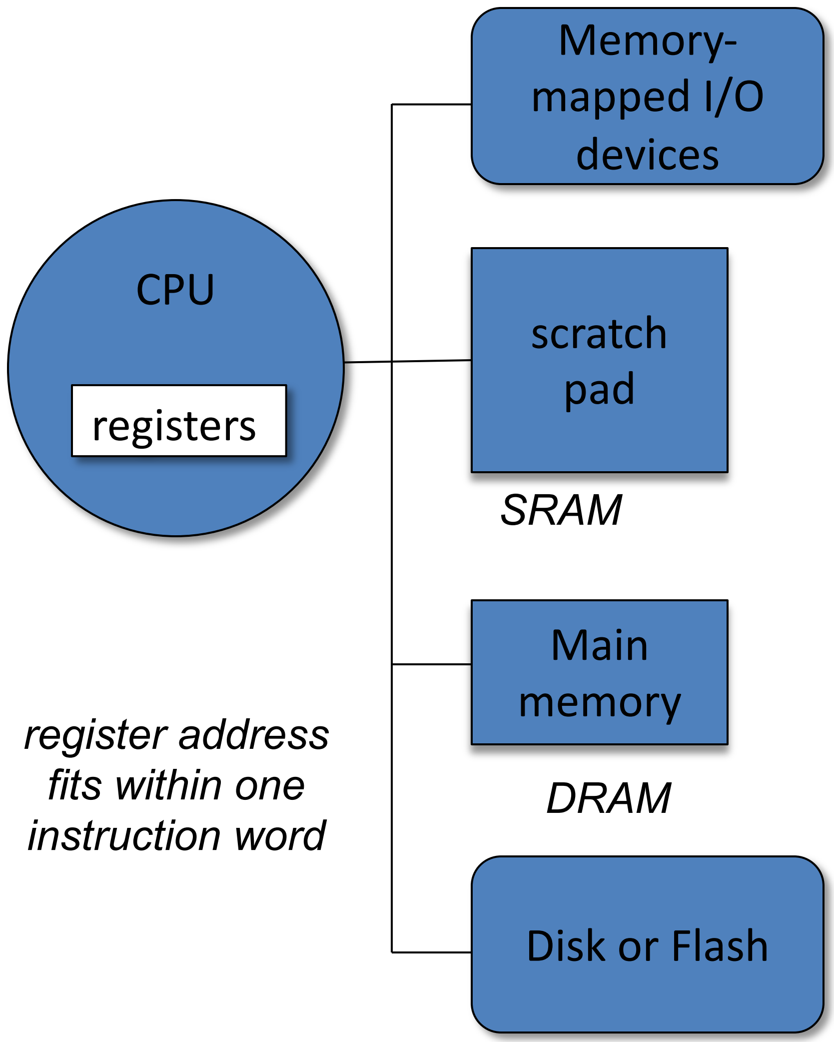
- Memory hierarchy
  - Cache:
    - A subset of memory addresses is mapped to SRAM
    - Accessing an address not in SRAM results in *cache miss*
    - A miss is handled by copying contents of DRAM to SRAM
  - Scratchpad:
    - SRAM and DRAM occupy disjoint regions of memory space
    - Software manages 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



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

Memory Hierarchy

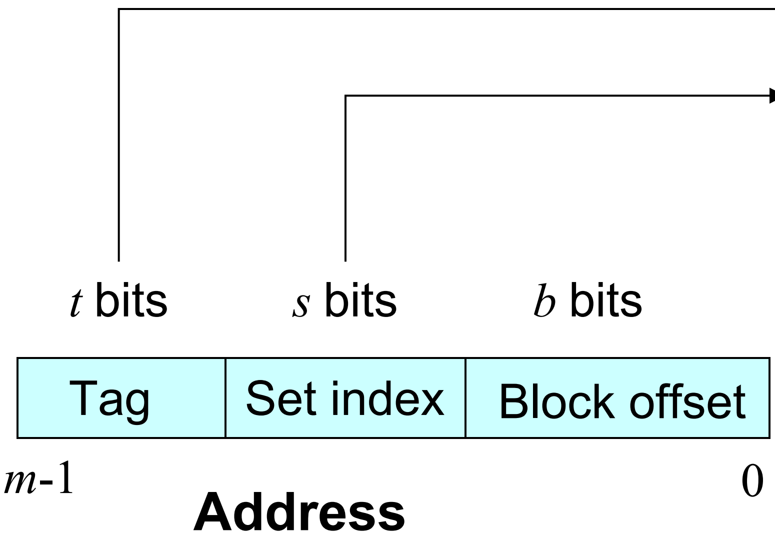


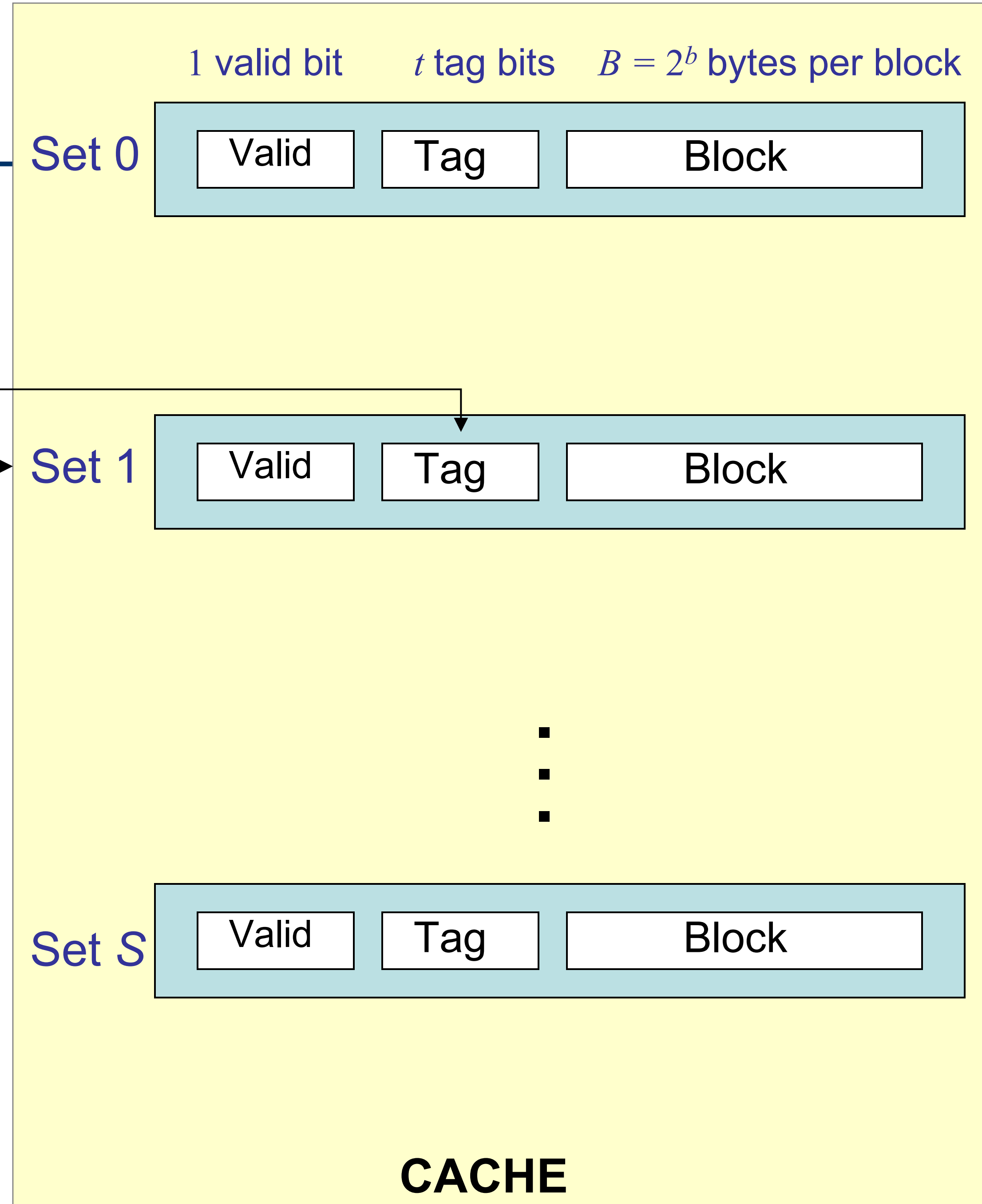
Here, each distinct piece of memory hardware has its own segment of the address space.

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

Direct-Mapped Cache

A "set" consists of one "line"

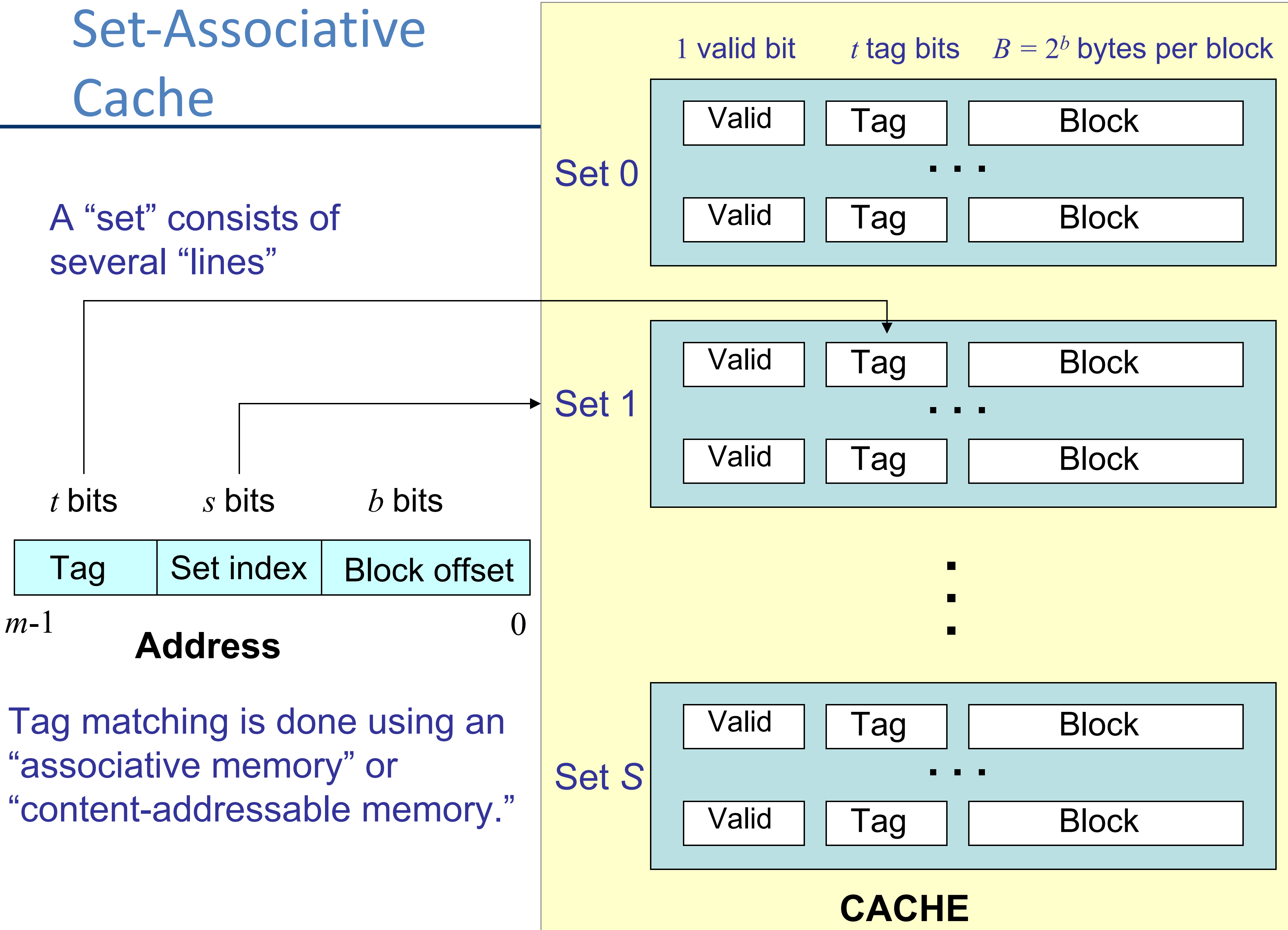




If the tag of the address matches the tag of the line, then we have a "cache hit." Otherwise, the fetch goes to main memory, updating the line.

Set-Associative Cache

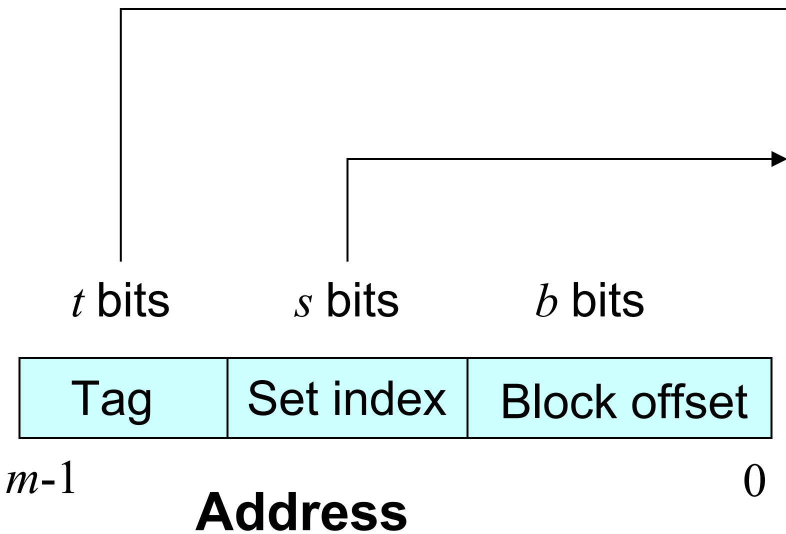
A “set” consists of several “lines”



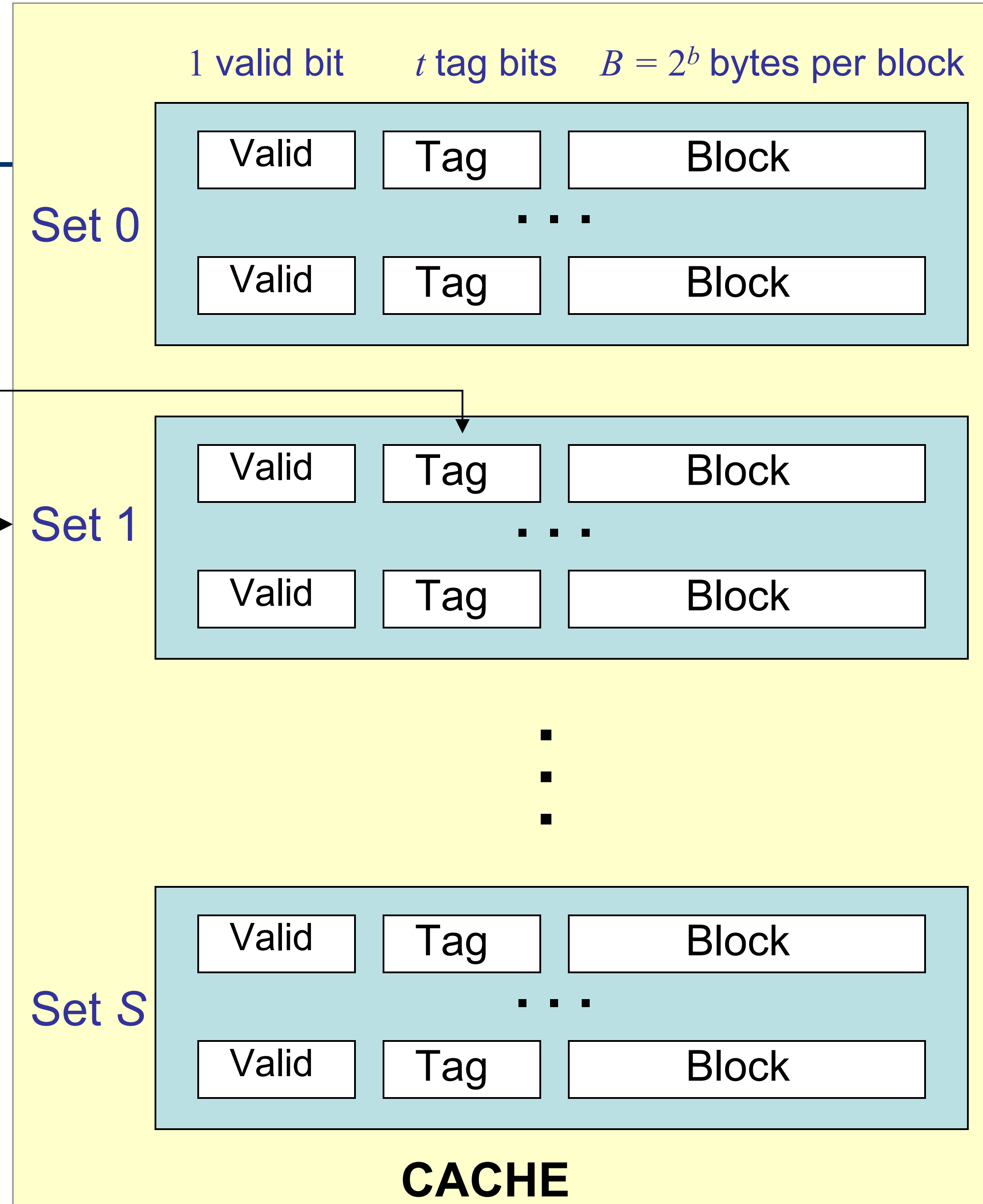
Tag matching is done using an “associative memory” or “content-addressable memory.”

Set-Associative Cache

A “set” consists of several “lines”

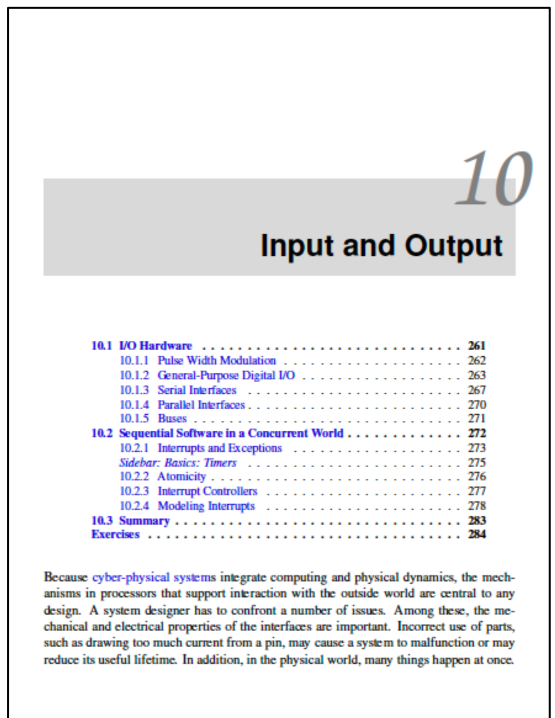


A “cache miss” requires a replacement policy (like LRU or FIFO).



Things to do …

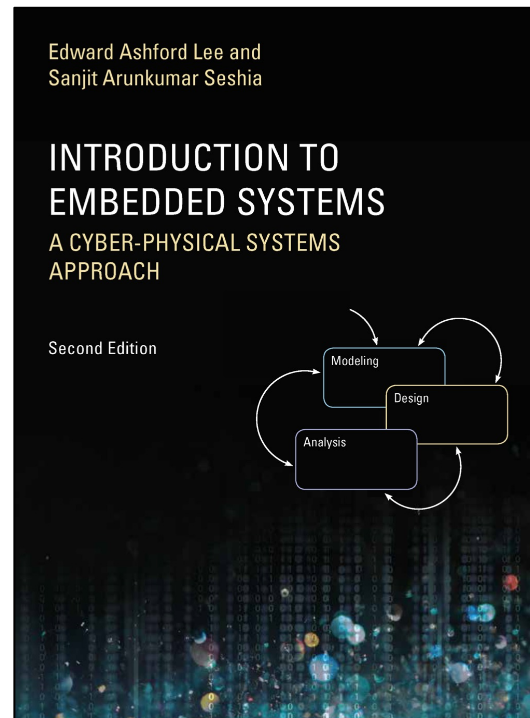
- Download the textbook and read Chapter 10
- Read over Workshop 2 and do the pre-workshop work



10

Input and Output

Because cyber-physical systems integrate computing and physical dynamics, the mechanisms in processors that support interaction with the outside world are central to any design. A system designer has to confront a number of issues. Among these, the mechanical and electrical properties of the interfaces are important. Incorrect use of parts, such as drawing too much current from a pin, may cause a system to malfunction or may reduce its useful lifetime. In addition, in the physical world, many things happen at once.



Next Lecture

- Input/Output