Cyber-Physical Systems

Memory Architecture

ICEN 553/453– Fall 2018

Prof. Dola Saha

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 Issues in Embedded System

- 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)

Memory Architecture Issues in Embedded System

- The memory model of C
- Memory hierarchies
  - scratchpads, caches, virtual memory
- Memory protection
  - segmented spaces

Non-volatile Memory

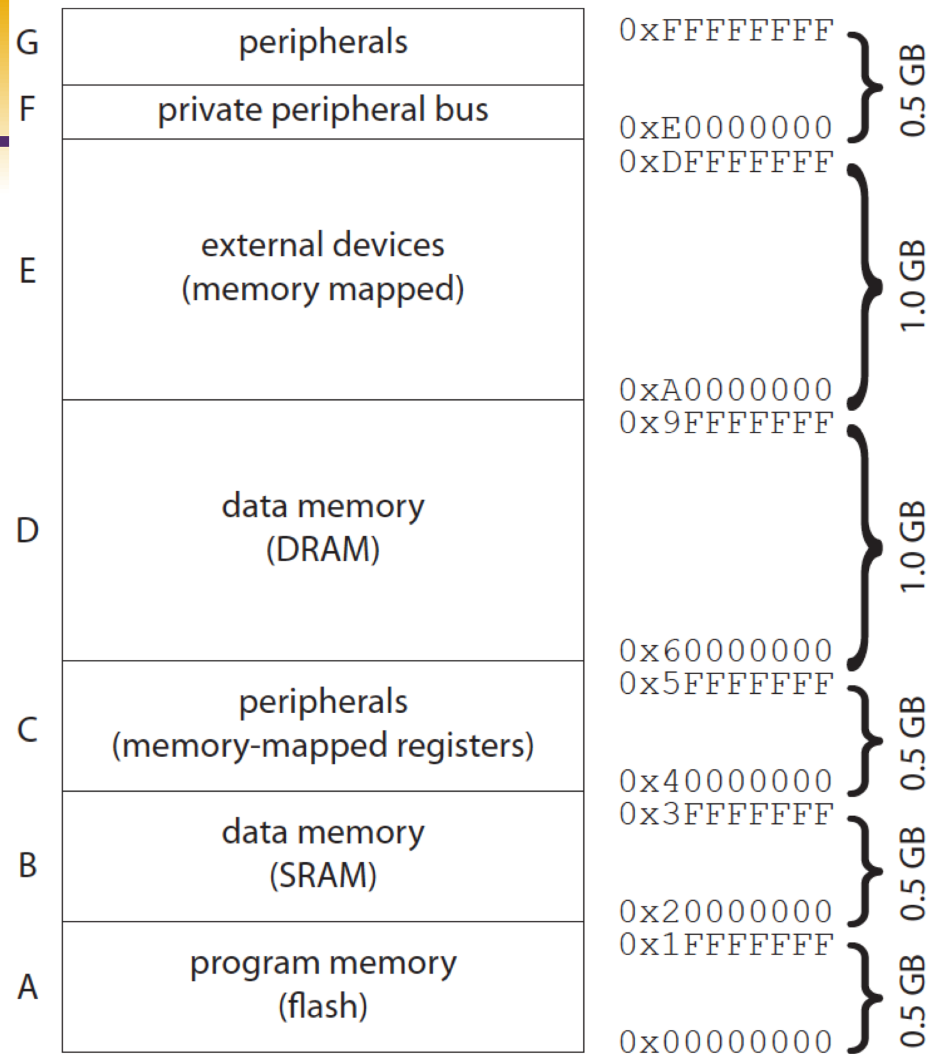
- preserves the content when power is off
  - **EPROM**: erasable programmable read only memory
    - Erase by exposing the chip to strong UV light
  - **EEPROM**: electrically erasable programmable read-only memory
  - **Flash memory**
    - Erased a "block" at a time, Limited number of program/erase cycles
    - Controllers can get quite complex
  - **Disk drives**
    - Not as well suited for embedded systems

Volatile Memory

- 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 Memory Map

- ARM Cortex M3
- Defines the mapping of addresses to physical memory.
- Why do this?



Raspberry Pi

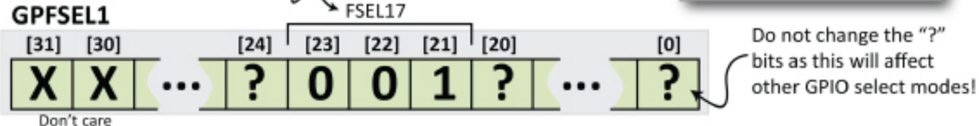
- Memory Map

Register	Offset	MSB			32-bits for each register							LSB
Offset is from the virtual address 0x3F000000 on the RPi 2/3, and 0x20000000 on all other RPi models.												
GPFSELn	The Function Select mode for each GPIO (3 bits for each GPIO and 54 GPIOs in total)											Read/Write
Bits		[31-30]	[29-27]	[26-24]	[23-21]	[20-18]	[17-15]	[14-12]	[11-9]	[8-6]	[5-3]	[2-0]
GPFSEL0	0000	X	FSEL9	FSEL8	FSEL7	FSEL6	FSEL5	FSEL4	FSEL3	FSEL2	FSEL1	FSEL0
GPFSEL1	0004	X	FSEL19	FSEL18	FSEL17	FSEL16	FSEL15	FSEL14	FSEL13	FSEL12	FSEL11	FSEL10
GPFSEL2	0008	X	FSEL29	FSEL28	FSEL27	FSEL26	FSEL25	FSEL24	FSEL23	FSEL22	FSEL21	FSEL20
GPFSEL3	000C	X	FSEL39	FSEL38	FSEL37	FSEL36	FSEL35	FSEL34	FSEL33	FSEL32	FSEL31	FSEL30
GPFSEL4	0010	X	FSEL49	FSEL48	FSEL47	FSEL46	FSEL45	FSEL54	FSEL43	FSEL42	FSEL41	FSEL40
GPFSEL5	0014	[32-12] X							FSEL53	FSEL52	FSEL51	FSEL50
GPSETn	The Output Set register - use this to set a GPIO high (1 bit for each of the 54 GPIOs)											Read/Write
GPSET0	001C	[31-0] mapped to GPIO31 to GPIO0 (1 = set GPIO)(0 = no effect)										
GPSET1	0020	[31-22] X			[21-0] mapped to GPIO53 to GPIO32 (1 = set GPIO)(0 = no effect)							
GPCLRn	The Output Clear register - use this to set a GPIO low (1 bit for each of the 54 GPIOs)											Read/Write
GPCLR0	0028	[31-0] mapped to GPIO31 to GPIO0 (1 = clear GPIO)(0 = no effect)										
GPCLR1	002C	[31-22] X			[21-0] mapped to GPIO53 to GPIO32 (1 = clear GPIO)(0 = no effect)							
GPLVLn	The Level Read register - use this to read the value of a GPIO (1 bit for each of the 54 GPIOs)											Read Only
GPLVL0	0034	[31-0] mapped to GPIO31 to GPIO0 (1 = level is high)(0 = level is low)										
GPLVL1	0038	[31-22] X			[21-0] mapped to GPIO53 to GPIO32 (1 = level is high)(0 = level is low)							
GPPUD	The Pull-up/Pull-down Enable register - use this to define the configuration											Read/Write
GPPUD	0094	[31-2] X										[1-0]
GPPUDCLKn	The Pull-up/Pull-down Enable Clock register - use this to apply GPPUD to a particular GPIO											Read/Write
GPPUDCLK0	0098	[31-0] mapped to GPIO31 to GPIO0 (1 = assert clock)(0 = no effect)										
GPPUDCLK1	009C	[31-22] X			[21-0] mapped to GPIO53 to GPIO32 (1 = assert clock)(0 = no effect)							

Function	GPFSELn
Bits	**Meaning**
000	input
001	output
100	ALT0
101	ALT1
110	ALT2
111	ALT3
011	ALT4
010	ALT5

Example: Set GPIO17 to be an output and set it high.
Solution: Write bits 001 to FSEL17, which is bits 21, 22, and 23 of the GPFSEL1 register to set the pin up as an output.
Then write a 1 to bit 17 of the GPSET0 register to set the output high.

Function	PUD
Bits	**Meaning**
00	no pull-up/down
01	pull-down
10	pull-up
11	X



UNIVERSITY AT ALBANY
State University of New York

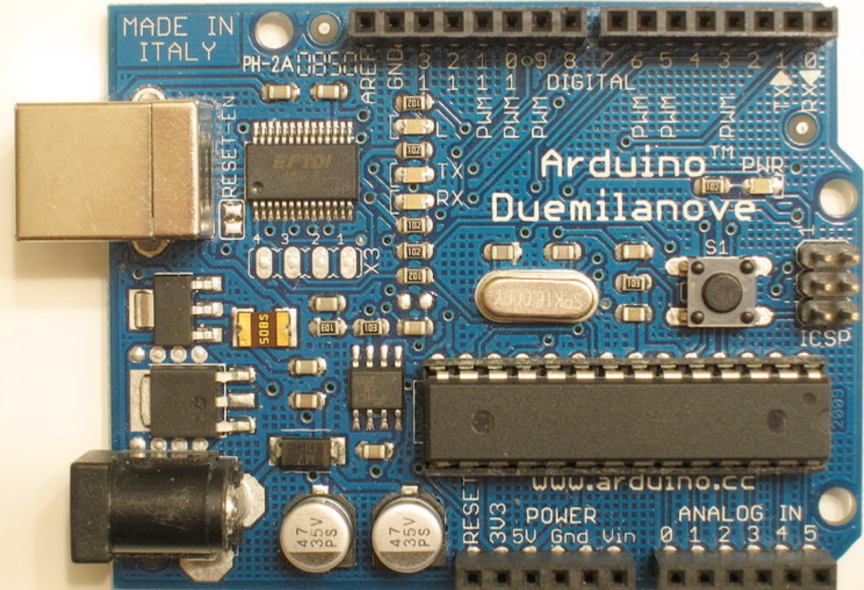
AVR

- 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.

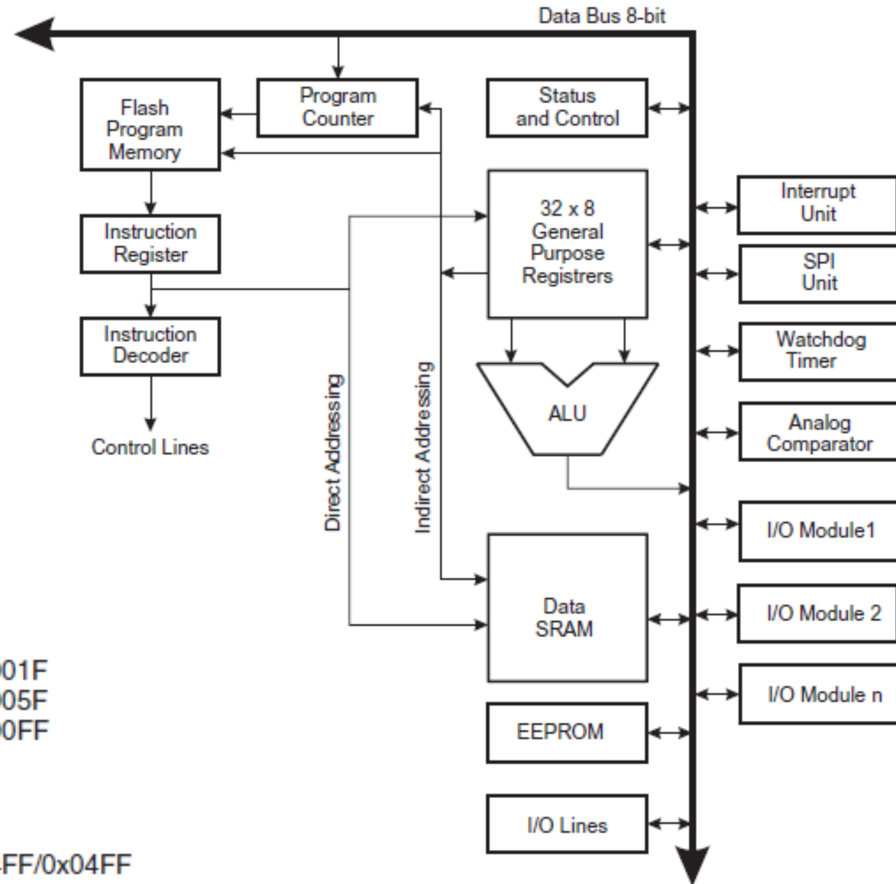
A Use of AVR: Arduino

- 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



ATMega 168: An 8-bit microcontroller with 16-bit addresses



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

AVR microcontroller architecture used in iRobot command module.

Why is it called an 8-bit microcontroller?

Questions?

1. What is the difference between an 8-bit microcontroller and a 32-bit microcontroller?

2. Why use volatile memory? Why not always use non-volatile memory?

Memory Organization

- 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

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

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).

What is meant by the following C code?

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

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

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

Dynamically-Allocated Memory

- 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.

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