ECE 2049 Lecture 4

0

- LAB 0
  - Report due Friday by 11:59PM EDT
  - Signoff due by end of office hours (or ask for an appointment)
  - If you are having any parts issues, or you are concerned about the deadline, please talk to me!

- LAB 1
  - Starts today (still okay if you are finishing Lab 0)
  - Short prelab assignment due by Monday (6/7) – see lab for details
  - Intro video posted ← HIGHLY RECOMMENDED!

- HW2 – Due Tuesday (6/8) by 2PM EDT

- Office Hours
  - Today: 2-4PM, 5-7PM (Nick)
  - Friday: 11AM-1PM (Chintan)

Module 4. MSP430 Architecture & Intro to Digital I/O

Topics

- Getting to know the MSP430 Hardware
- Start of Digital I/O

Last Time

- Memory Organization
- Floating point format

Getting to know the MSP430 Hardware

In a programming course, typically you focus on just the code:

- Learn a high level programming language and some algorithms
- Use a "computer" from a high level

A typical program does three things:

1. READ IN SOME DATA
2. MANIPULATE DATA SOMEHOW
3. OUTPUT DATA

A few common points for all kinds of software:

- Use constructs like loops, conditional, algorithms like search, sort, data structures
- Software: write logic and syntax correctly and it will just work
- Library functions for I/O

... but what's inside the "computer"? How does it work? When you are writing in a high level programming language, do you care?

In contrast, developing software for embedded **requires** much more in depth knowledge about the microprocessor that is the target of your program. For instance, it's important to know:

"CLOSER" TO HARDWARE.

A general software hierarchy

We can think of the software components in a system and the way they interact with the hardware as a hierarchy or *software stack*:

A general computing device (PC) might have a software hierarchy like this:

Applications *Browser, Discord, MATLAB, Skyrim*
Operating System **(Includes user interface)** *Windows, Linux, Mac OS*
System Software: Interface to hardware *Drivers*
Hardware (CPU, Memory, Peripherals)

Provides interface + standard libraries.

Hardware Abstraction Layer (HAL)
Common way to work w/ different hardware types.

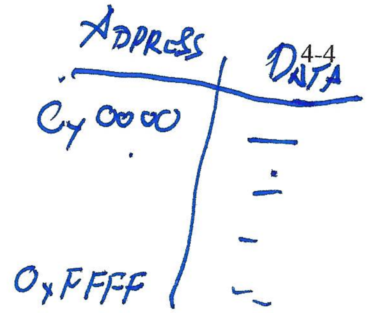
On an embedded system, this stack gets "squashed":

Application *(singular)*
System software / OS *(thin)*
Hardware (CPU, Memory, Peripherals)

– Smaller software libraries, but much tighter integration with hardware.

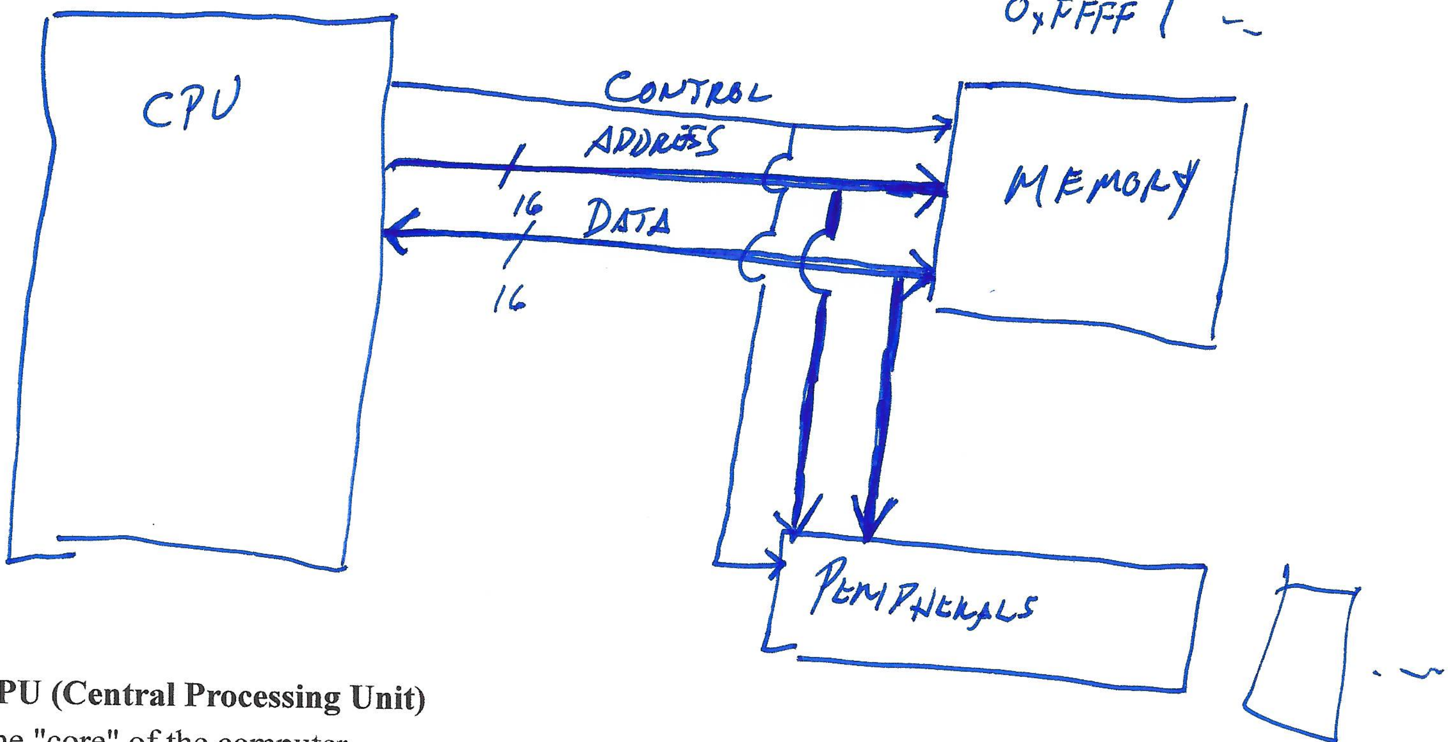
- Application is closely integrated with the hardware layer
- Little or no operating system–usually only runs one task or a set of tasks
- Often little or no "wrapping" of functionality
- On larger systems, you may use a Real Time Operating System (RTOS) that provides some basic support for multitasking…

→ ECE 3849



A general microprocessor hardware architecture

In general, any microprocessor system has the following components:



CPU (Central Processing Unit)

The "core" of the computer

- EXECUTES YOUR PROGRAM
- PROVIDES TIMING

Memory

Stores information

- CONTROLLED BY CPU
- TWO TYPES OF MEMORY
  - VOLATILE MEMORY (WORKING MEMORY)
  - NON-VOLATILE MEMORY (LONG-TERM STORAGE)

Peripherals

- BUTTONS, LEDS, TIMERS, ADC, ...
- EVERYTHING THAT'S NOT MEMORY

How does the CPU work?

The CPU executes *machine code*, which are low-level instructions directly run by the hardware. Machine code is a binary format seen by the CPU.

- Instructions perform very specific tasks
- Instruction set (ISA) is different for every CPU type (MSP430, ARM, x86, …)
- **Compiler is responsible for figuring out how to build all programs using these instructions!**

main.c

```c
while (1)     // Forever loop
{
    // Read buttons S1-S4
    ret_val = readButtons();

    setLeds(~ret_val);
    if (~ret_val & 0x01) {
        BuzzerOn();
    }
    if (~ret_val & 0x08) {
        BuzzerOff();
    }

    // Check if any keys have been pressed on the 3x4 keypad
    currKey = getKey();
    if ((currKey >= '0') && (currKey <= '9')) {
        setLeds(currKey - 0x30);
    }
    else if (currKey == '*') {
        BuzzerOn();
    }
    else if (currKey == '#') {
        BuzzerOff();
    }

    if (currKey)
    {
```

Disassembly

```
00562c:  134F            CALLA   R15
00562e:  40F1 0020 0010  MOV.B   #0x0020,0x0010(SP)
005634:  40F1 0020 0012  MOV.B   #0x0020,0x0012(SP)
00563a:  13B0 6646       CALLA   #readButtons
00563e:  4CC1 000C       MOV.B   R12,0x000c(SP)
005642:  415C 000C       MOV.B   0x000c(SP),R12
005646:  E37C            INV.B   R12
005648:  13B0 6834       CALLA   #setLeds
00564c:  415F 000C       MOV.B   0x000c(SP),R15
005650:  E33F            INV.W   R15
005652:  B31F            BIT.W   #1,R15
005654:  2402            JEQ     (C$DW$L$main$5$E)
005656:  13B0 6932       CALLA   #BuzzerOn
00565a:  415F 000C       MOV.B   0x000c(SP),R15
00565e:  E33F            INV.W   R15
005660:  B23F            BIT.W   #8,R15
005662:  2402            JEQ     (C$DW$L$main$7$E)
005664:  13B0 6A76       CALLA   #BuzzerOff
005668:  13B0 610E       CALLA   #getKey
00566c:  4CC1 000D       MOV.B   R12,0x000d(SP)
005670:  90F1 0030 000D  CMP.B   #0x0030,0x000d(SP)
005676:  280B            JLO     (C$DW$L$main$10$E)
005678:  90F1 003A 000D  CMP.B   #0x003a,0x000d(SP)
00567e:  2C07            JHS     (C$DW$L$main$10$E)
005680:  415C 000D       MOV.B   0x000d(SP),R12
005684:  807C 0030       SUB.B   #0x0030,R12
005688:  13B0 6834       CALLA   #setLeds
00568c:  3C0D            JMP     (C$DW$L$main$14$E)
```

We will never write in assembly in this class. However, it is important that you understand that these instructions exist!

CPU instructions operate on…

- **Internal Registers**: 16 general purpose registers (R0-R15)
  - Storage locations inside the CPU used for recent instructions
    All registers are 16-bits wide (except R0 and R1, which are 20 bits)
  - Can be accessed very quickly (one clock cycle)
  - Some registers control program execution
    (R0 = Program counter, R1 = Stack pointer, R2 = Status register)
- **Memory**: Instructions read from and write to memory
  - Load and store data from the outside world using the memory bus!

What goes in memory?

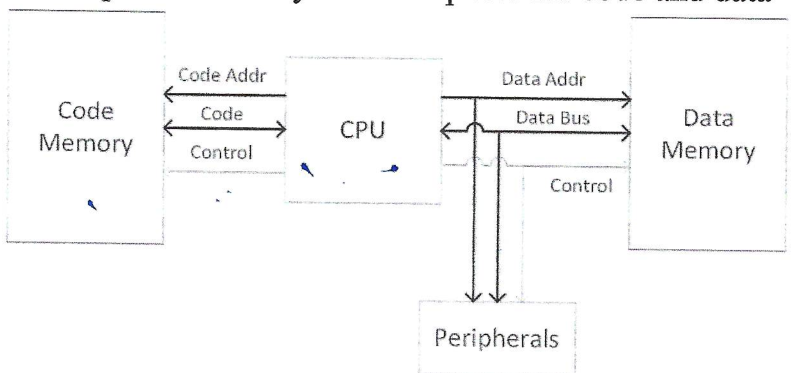
Remember that memory doesn't just store your variables–it stores the program's code as well!

- The CPU needs to load both code **and** data from memory

There are two generic types of memory architectures used by microprocessors and microcontroller systems:

- **Von Neumann** Architecture (~1952)
- **Harvard** Architecture (~1944)

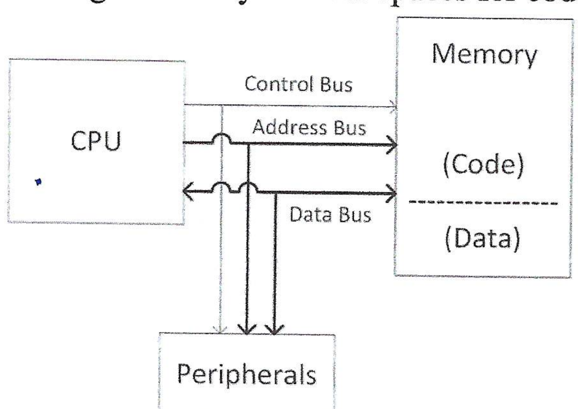
Harvard Architecture: Separate memory address spaces for code and data



Benefits: Instruction fetch and data read happen in parallel
Drawbacks: Separate instruction and data buses
In this form, the Harvard architecture is used today by highly-pipelined systems like DSP chips.

Von Neumann Architecture: Single memory address spaces for code and data

↑ MSP430



Benefits: Single address and data buses (simpler to interface)
Drawbacks: Implicit bottleneck since we have the same pipeline for code and data

ON MODERN SYSTEMS: USE A HYBRID FORM OF THE TWO

"MODIFIED HARVARD ARCH."

The MSP430 Architecture

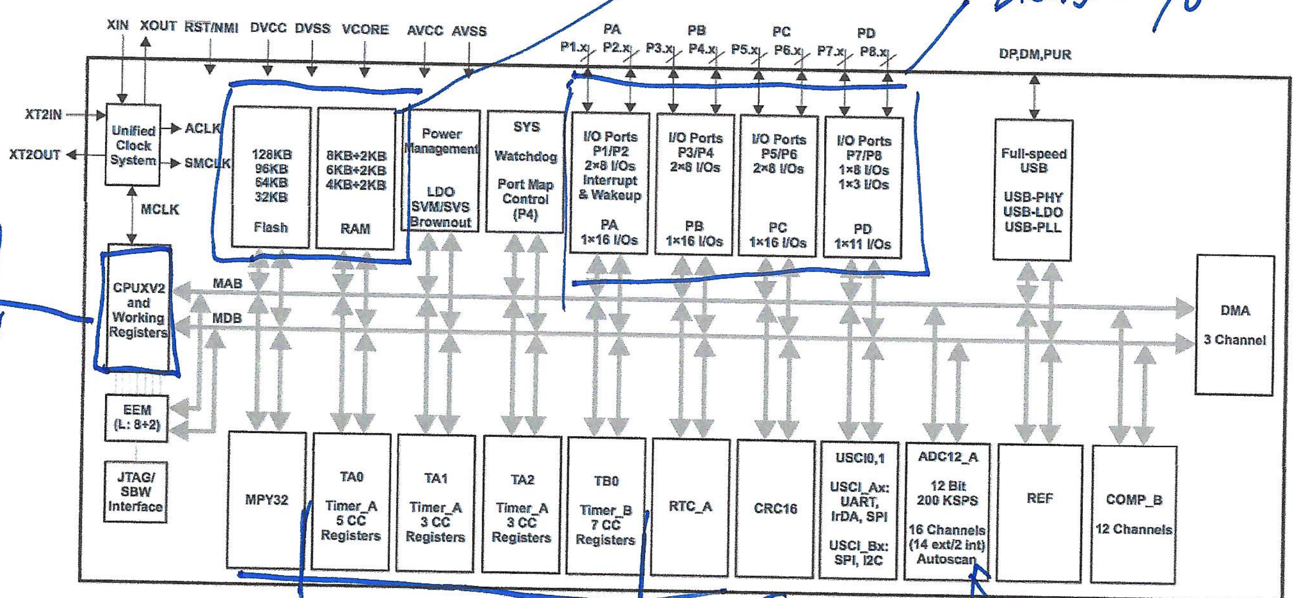
The MSP430 is a family of microcontrollers–there are hundreds of versions of this CPU with various configurations of memory and peripherals!

- You can think of it as a type of *System on a Chip* (SoC)

In our labs, we use the MSP430F5529

- 128KB of flash memory: Used for code storage
- 8 KB of RAM (+ 2KB RAM for USB controller): Used for data storage
- Lots of peripherals
  - 32 bit multiplier
  - Timers, comparator, USB controller
  - Much, much more!

How much more? Here's a block diagram:



Note the lines connecting all of the peripherals: this is the memory bus!

MAB: Memory Address Bus

MDB: Memory Data Bus

MSP430 Memory Organization

Memory: A group of sequential locations where binary data is stored

- On the MSP430, each memory location holds one byte
- Each byte has a unique address which the CPU uses to access it
- Multibyte data is stored in ________ Endian!

→ LITTLE ENDIAN

ADDRESS	DATA
0x0000	—
⋮	
0xFFFF	—

Two types of memory: Volatile and Non-Volatile

RAM (Random Access Memory) (SRAM)

- Our MSP430 has 8KB of RAM + 2KB for USB
- RAM is **volatile,** meaning that it loses its state when the chip is not powered
- Used as data memory
- Accessed via read and write instructions

< 10 ns READ/WRITE

Flash (ROM, READ ONLY MEMORY)

- Used primarily for code memory
- Flash is **non-volatile**, meaning that its state persists even if the chip is not powered
- CPU fetches code from flash automatically
- Accessed via program control, but more difficult than RAM
  - Write time >> Read time
  - Writes must occur in large segments (512 bytes)

→ READ ONLY UNDER NORMAL CONDITIONS.

How are programs stored in memory?

When a program is compiled, the linker arranges different portions into various memory *segments*, which are stored in different contiguous memory regions. The most important segments are:

SRAM

- The stack (`.stack`): Stores local variables and context information on each function call
- Constant data (`.data`, `.bss`): Stores global variables and other constant data (strings, lookup tables, etc.

FLASH

- Text (`.text`): Compiled code for your program (code you write + libraries)
- Heap: Dynamically allocated memory (avoid using this!)

When compiling, the linker reads a script called a *command file*, which maps each section to a memory device. Usually, most code is stored in flash, while most data goes in RAM, though it may be necessary to adjust these requirements. Why?

Why should we avoid dynamically allocated memory?

MALLOC() → ALLOWS PROGRAM TO ALLOCATE MEMORY AT RUNTIME.

- MEMORY IS LIMITED
- MALLOC IS VERY SLOW.

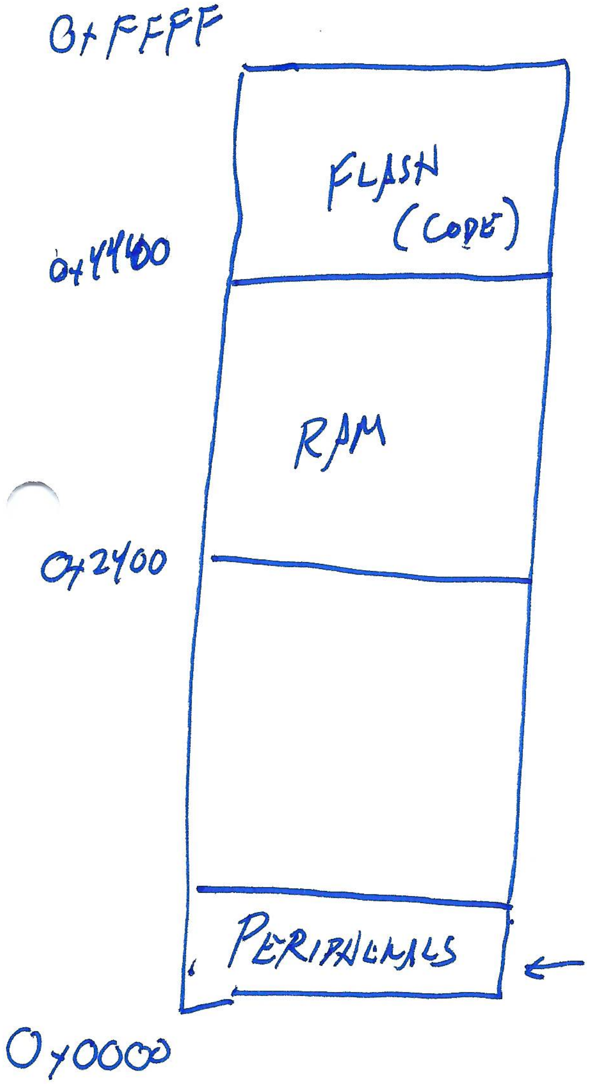
⇒ WE AVOID MALLOC WHEN WRITING EMBEDDED CODE!

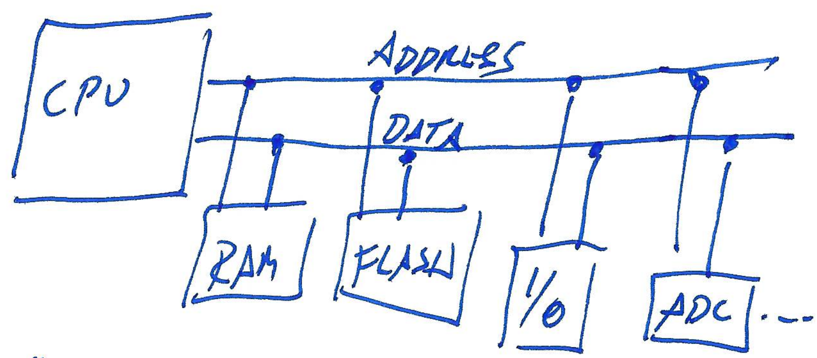
Memory architecture and layout

The MSP430 is a 16-bit microcontroller, meaning that:

- The data bus is 16 bits wide
- Internal CPU registers are 16-bits

Note: MSP430 '5xxx and '6xxx families use a **20 bit address bus** to allow access to at most 1MB of memory.





"MEMORY-MAPPED PERIPHERALS"

I/O + PERIPHERALS ARE ACCESSED IN SAME MANNER AS MEMORY

However, memory isn't just one big block….