# ECE 471 – Embedded Systems
# Lecture 5

Vince Weaver

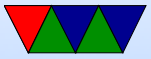
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

13 September 2016

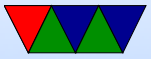
# Announcements

- HW#2 is due Thursday
- It is going OK?



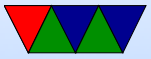
# Homework #1 Review

- Characteristics of embedded system: embedded inside, resource constrained, dedicated purpose, real-time
- Cost is an interesting one. Something like a desktop might be optimized for cost extremely, while a one-off embedded system might not, and in fact might be over-engineered (like a spaceprobe) because has to operate in tough conditions.
- Interesting that some people didn't think iPhone is optimized for battery life



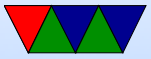
- Operating system? Can have an OS and still be considered embedded.
- Be strong in your convictions!
- Embedded inside of? Not simply the SoC is inside, as that's always true. Usually means the device is inside of something bigger, and controlling part of it. Complicated. For example, Sandybridge chip in laptop is not considered embedded, but the ARM processors in the hard drive are.

  Another example, if put a desktop inside an enclosure, attach a money dispenser, and make an ATM, is it an



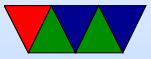
embedded system?

- Interesting question, is a Pi an embedded system? Itself, probably not. If you hooked it up to something (controlled the lights, robot, etc) then probably.
- Make sure realize difference between the embedded system (the whole device) and the processor (SoC or microcontroller)
- Phone has real time constraints? Know I didn't explain real-time well, later. Can meet deadlines. Yes, but on which processor?
- iPhone – most said was *not* embedded system



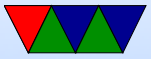
iPhone7

- Microwave – all said was embedded system
- ARM1176JZF-S: Java, TrustZone, Vector Floating, Synthesizable Jazelle = Java acceleration
  This was in the class notes (which I post), and in ARMv6 documentation.



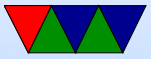
# How Code Works

- Compiler generates ASM (Cross-compiler)
- Assembler generates machine language objects
- Linker creates Executable (out of objects)



# Tools

- compiler: takes code, usually (but not always) generates assembly
- assembler: GNU Assembler as (others: tasm, nasm, masm, etc.)
  creates object files
- linker: ld
  creates executable files. resolves addresses of symbols. shared libraries.

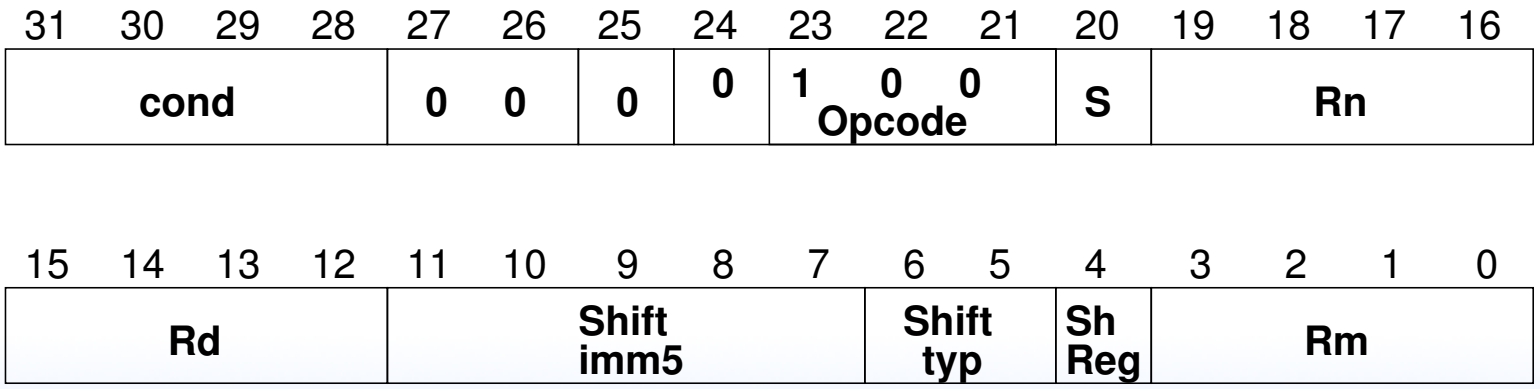


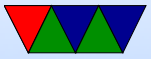
# Converting Assembly to Machine Language

Thankfully the assembler does this for you.

ARM32 ADD instruction – `0xe0803080` == `add r3, r0, r0, lsl #1`

`ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}`

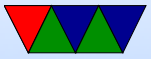


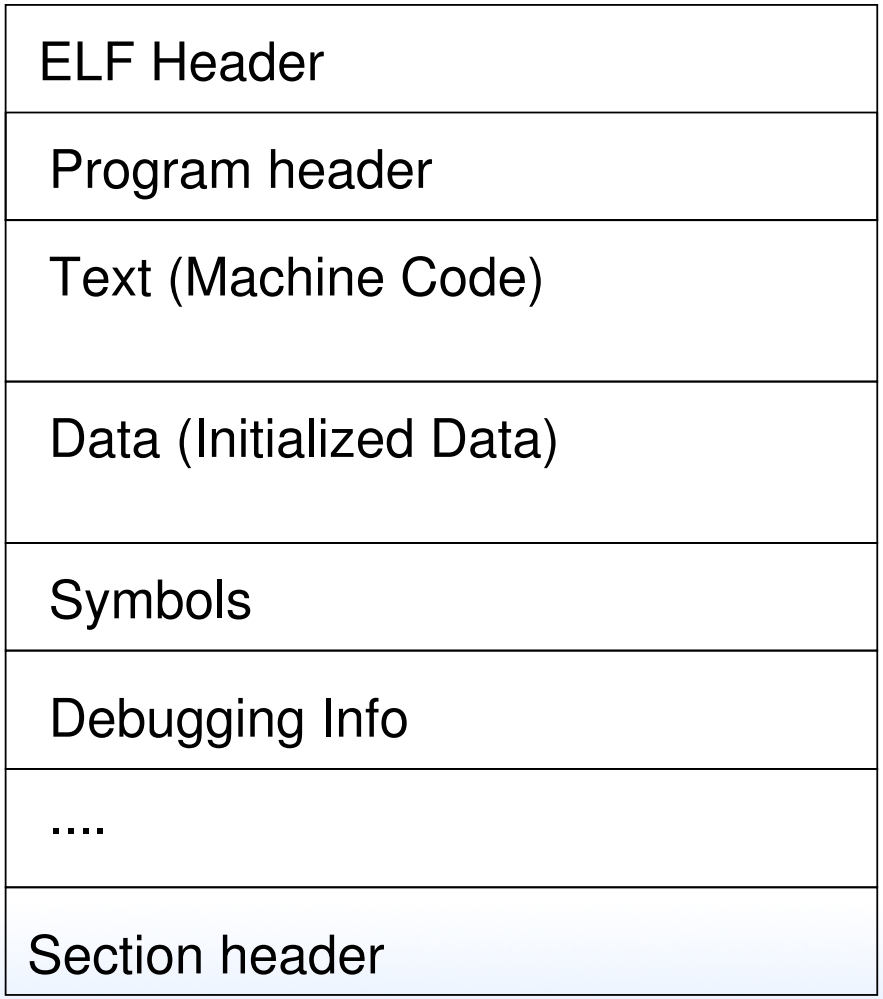
# Executable Format

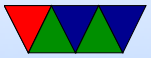
- ELF (Executable and Linkable Format, Extensible Linking Format)
  Default for Linux and some other similar OSes
  header, then header table describing chunks and where they go

- Other executable formats: a.out, COFF, binary blob



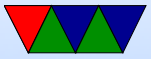
# ELF Layout


ELF Header
Program header
Text (Machine Code)
Data (Initialized Data)
Symbols
Debugging Info
....
Section header



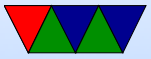
# ELF Description

- ELF Header includes a "magic number" saying it's 0x7f,ELF, architecture type, OS type, etc. Also location of program header and section header and entry point.
- Program Header, used for execution:
  has info telling the OS what parts to load, how, and where (address, permission, size, alignment)
- Program Data follows, describes data actually loaded into memory: machine code, initialized data

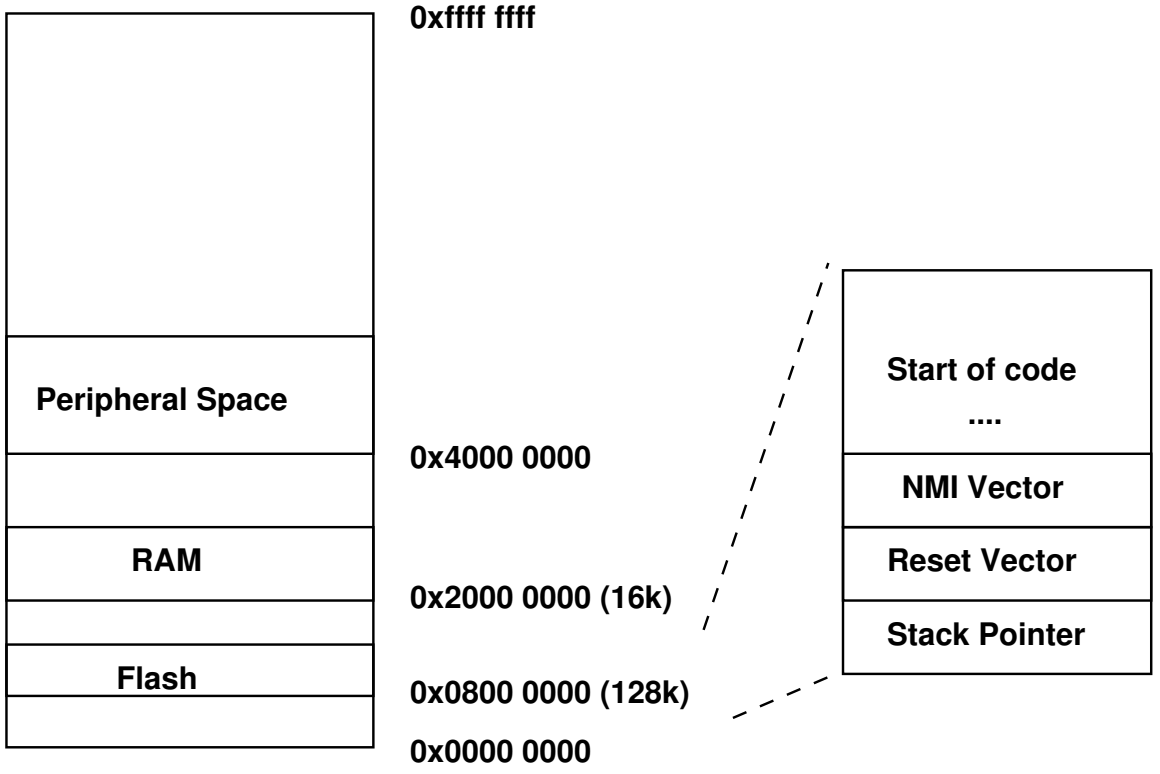


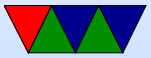
- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

- Section Header, used when linking:
  has info on the additional segments in code that aren't loaded into memory, such as debugging, symbols, etc.

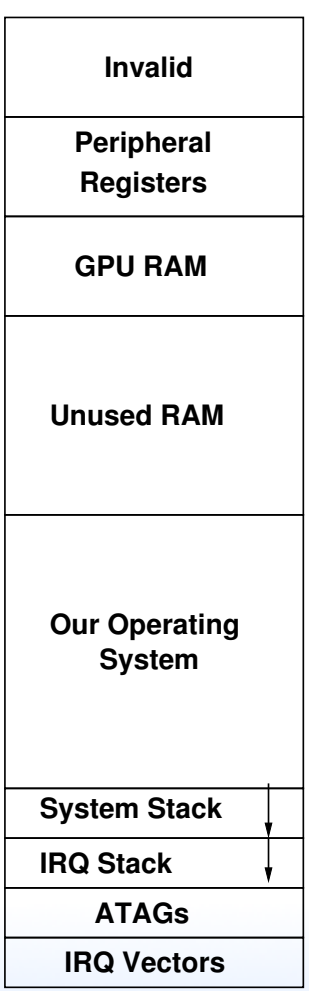


# STM32L-Discovery Physical Memory Layout



# Raspberry Pi Layout




0xffff ffff (4GB)

0x2100 0000 (528MB)

0x2000 0000 (512MB)

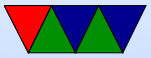
0x1c00 0000 (448MB)

0x0000 8000 (32k)

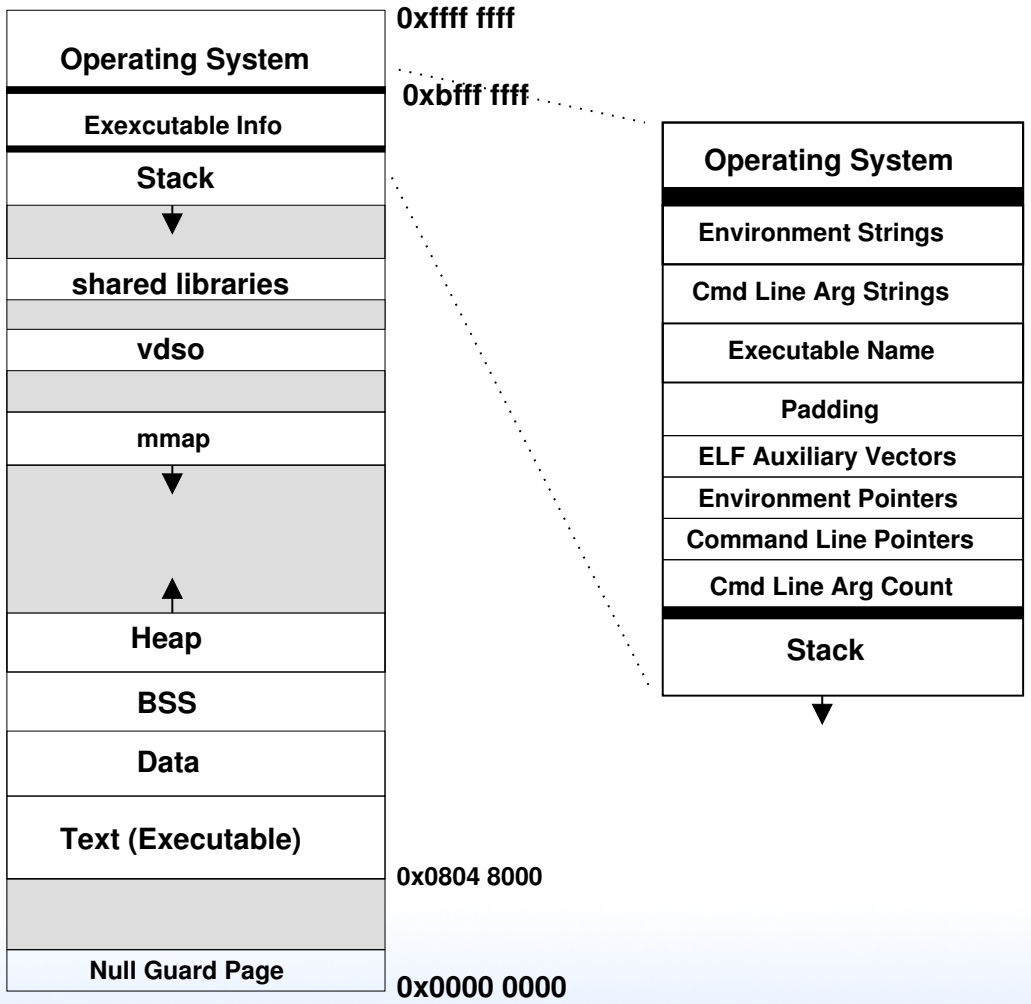
0x0000 4000 (16k)

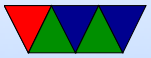
0x0000 0100 (256)

0x0000 0000



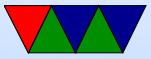
# Linux Virtual Memory Map



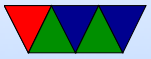
# Program Memory Layout on Linux

- Text: the program's raw machine code
- Data: Initialized data
- BSS: uninitialized data; on Linux this is all set to 0.
- Heap: dynamic memory. `malloc()` (`brk()` syscall) and C++ `new()`. Grows up.
- Stack: LIFO memory structure. Grows down.



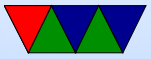
# Program Layout

- Kernel: is mapped into top of address space, for performance reasons
- Command Line arguments, Environment, AUX vectors, etc., available above stack
- For security reasons "ASLR" (Address Space Layout Randomization) is often enabled. From run to run the exact addresses of all the sections is randomized, to make it harder for hackers to compromise your system.



# Loader

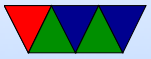
- /lib/ld-linux.so.2
- loads the executable (handles linking in libraries, etc)



# Static vs Dynamic Libraries

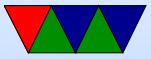
- Static: includes all code in one binary.
  Large binaries, need to recompile to update library code, self-contained, don't have to worry about incompatible updates

- Dynamic: library routines linked at load time.
  Smaller binaries, share code across system, automatically links against newer/bugfixes when system library updated

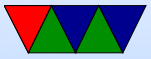


# How a Program is Loaded

- Kernel Boots
- `init` started
- `init` calls `fork()` – makes an exact copy of itself
- child calls `exec()` – replaces itself with executable from disk
- Kernel checks if valid ELF. Passes to loader



- Loader loads it. Clears out BSS. Sets up stack. Jumps to entry address (specified by executable)
- Program runs until complete.
- Parent process returned to if waiting. Otherwise, init.



# Assembly Language: What's it good for?

- Understanding your computer at a low-level
- Shown when using a debugger
- It's the eventual target of compilers
- Operating system writers (some things not expressible in C)
- Embedded systems (code density)
- Research. Computer Architecture. Emulators/Simulators.
- Video games (or other perf critical routines, glibc, kernel, etc.)