ECE 471 – Embedded Systems
Lecture 6

Vince Weaver

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

vincent.weaver@maine.edu

17 September 2018

Announcements

- HW#2 was posted, it is due Friday

Homework #1 Review

- Characteristics of embedded system
  - embedded inside, resource constrained, dedicated purpose, real-time
  - Toothbrush is actual specs I came across
  - Real-Time Confusion: we will discuss this more in future.

    Toothbrush: Just turning off the motor, and it takes an extra 1/2s is not really considered a real time thing. No one dies, no hardware destroyed, just mild annoyance

if noticed at all. Now if somehow it had to keep the waveform to H-bridge exact within 1ms or the motor would overheat and catch on fire, that could be a real-time issue.

Microwave: having a clock doesn't make it real time. Hopefully the door control has a physical interlock, but you never know. Usually when cooking food second granularity and some jitter not matter much.

○ Limited Hardware

bitness of processor: while 8 or 16 bit probably embedded these days, 32 vs 64 bit not necessarily

a sure sign.

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.

○ Operating system?
Can have an OS and still be considered embedded.
○ Be strong in your convictions!
• ASIC

- ○ cost/power. Depends a lot on numbers made, process, and how well designed it is.
- ○ Extra hardware overhead? ASIC mostly just flip flops and gates. SoC internally a lot more, but these days not much else is needed.
- ○ More secure? Can you reverse engineer an ASIC?
- ARM1176JZF-S: Java, TrustZone, Vector Floating, Synthesizable Jazelle $=$ Java acceleration
  This was in the class notes (which I post), and in ARMv6 documentation.

How Executables are Made

- 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.
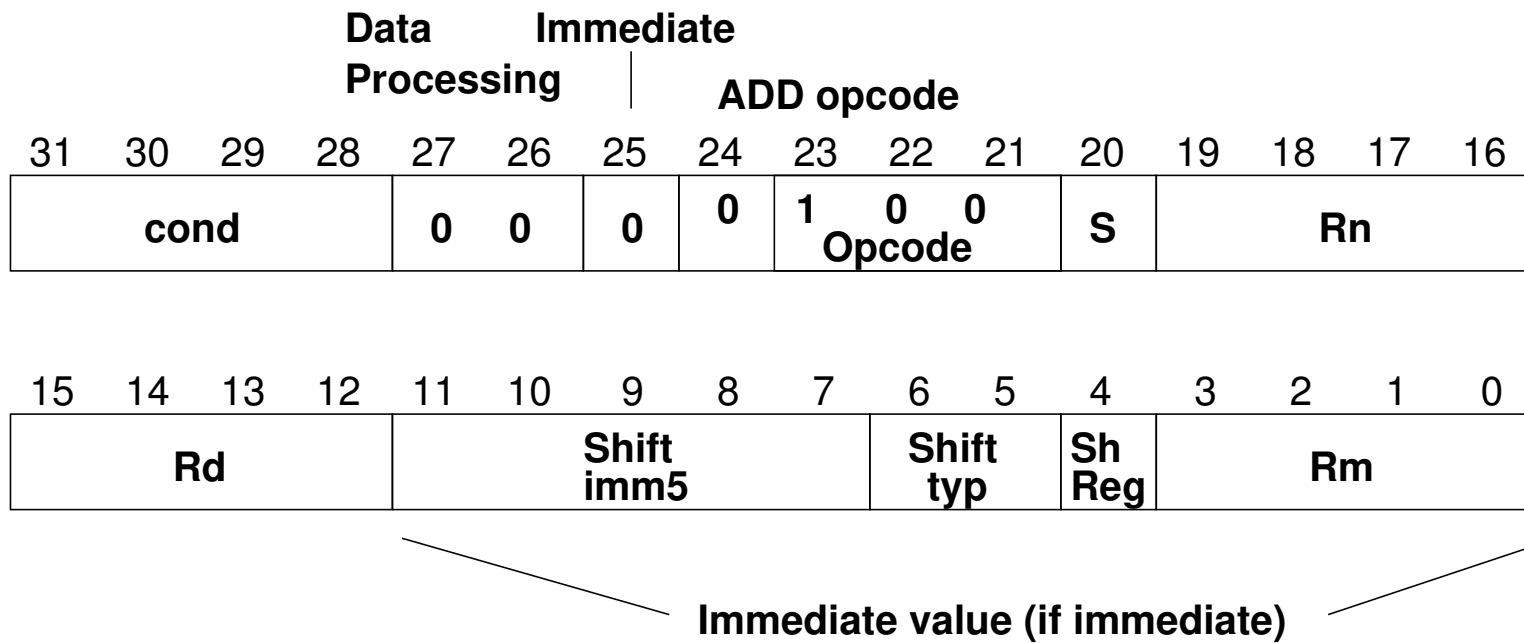
Converting Assembly to Machine Language

Thankfully the assembler does this for you.

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

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

**Data
Processing**

Immediate

ADD opcode

31 30 29 28	27 26	25	24	23 22 21	20	19 18 17 16
cond	**0 0**	**0**	**0**	**1 0 0** **Opcode**	**S**	**Rn**

15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
Rd	**Shift** **imm5**	**Shift** **typ**	**Sh** **Reg**	**Rm**

Immediate value (if immediate)

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

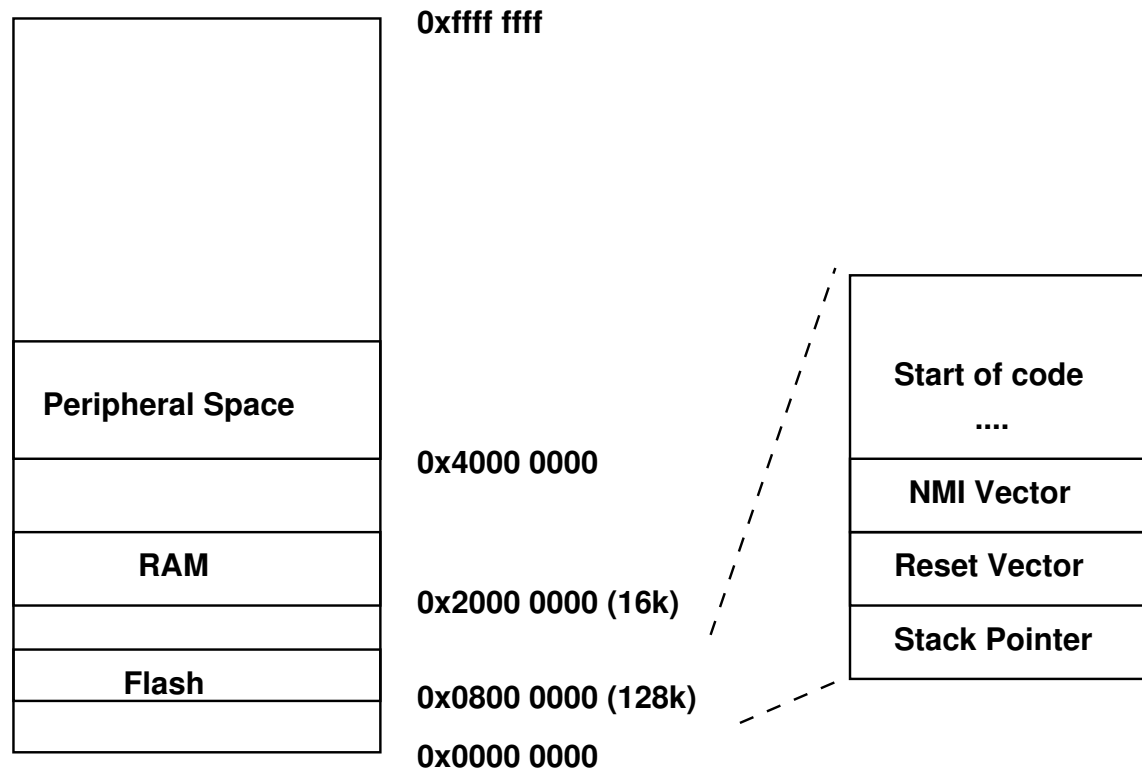| |
|---|
| ELF Header |
| Program header |
| Text (Machine Code) |
| Data (Initialized Data) |
| Symbols |
| Debugging Info |
| .... |
| Section header |

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.
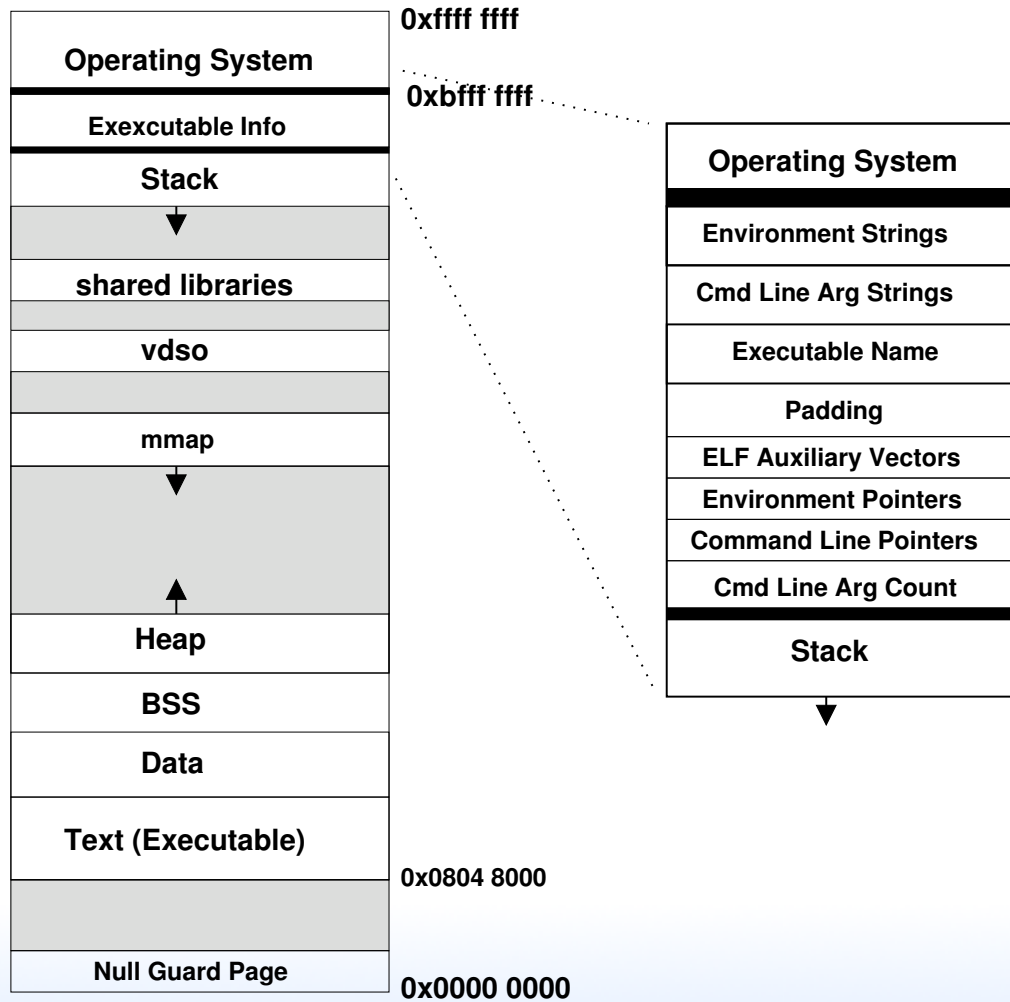
STM32L-Discovery Physical Memory Layout

0xffff ffff

Peripheral Space

0x4000 0000

RAM

0x2000 0000 (16k)

Flash

0x0800 0000 (128k)

0x0000 0000

Start of code
....

NMI Vector

Reset Vector

Stack Pointer

Raspberry Pi Layout

Memory Region	Address	Size
Invalid	0xffff ffff	(4GB)
Peripheral Registers	0x2100 0000	(528MB)
GPU RAM	0x2000 0000	(512MB)
Unused RAM	0x1c00 0000	(448MB)
Our Operating System		
System Stack	0x0000 8000	(32k)
IRQ Stack	0x0000 4000	(16k)
ATAGs		
IRQ Vectors	0x0000 0100	(256)
	0x0000 0000	

Linux Virtual Memory Map

Operating System	0xffff ffff
Exexcutable Info	0xbfff ffff
Stack ▼	
shared libraries	
vdso	
mmap ▼	
▲	
Heap	
BSS	
Data	
Text (Executable)	0x0804 8000
Null Guard Page	0x0000 0000

Operating System
Environment Strings
Cmd Line Arg Strings
Executable Name
Padding
ELF Auxiliary Vectors
Environment Pointers
Command Line Pointers
Cmd Line Arg Count
Stack ▼

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.