ECE 471 – Embedded Systems
Lecture 12

Vince Weaver

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

vincent.weaver@maine.edu

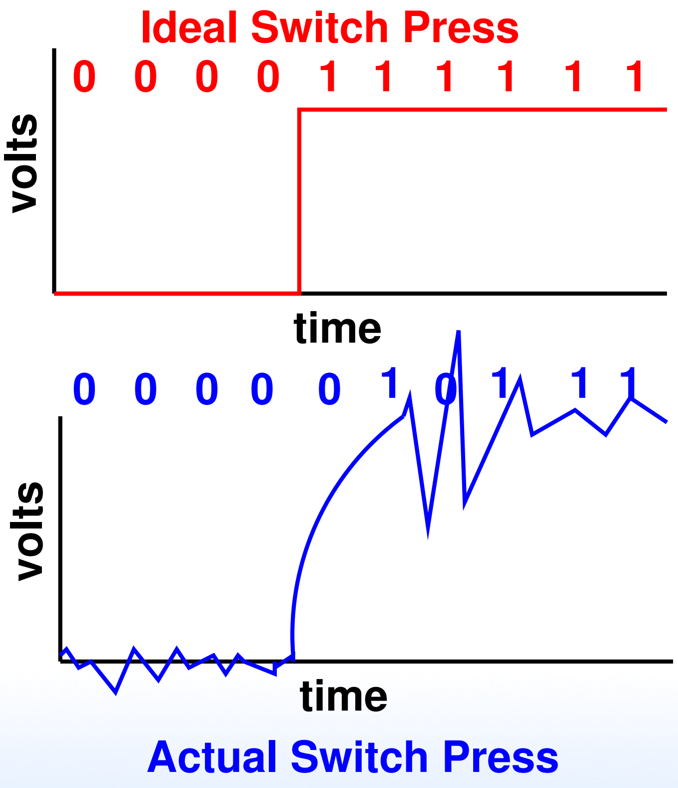
26 September 2022

Announcements

- HW#4 was posted
- If you need any parts (LED, breadboard) let me know

Debouncing (from last time)

- Noisy switches, have to debounce



Debouncing!

- Can you fix in hardware?
  - Capacitors (not for homework, will be grading software)
  - Built-in debounce (shift-registers?) like on STM32L?
- Can you fix in software? Algorithms
  - Wait until you get X consecutive values before changing
  - Get new value, wait short time and check again
  - These all have tradeoffs and can get caught by different

patterns of noise

Bypassing Linux for speed

`http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/`

Trying to generate fastest GPIO square wave.

shell	gpio util	40Hz
shell	sysfs	2.8kHz
Python	WiringPi	28kHz
Python	RPi.GPIO	70kHz
C	sysfs (vmw)	400kHz
C	WiringPi	4.6MHz
C	libbcm2835	5.4MHz
C	Rpi Foundation “Native”	22MHz

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
- Compiler can have front-end which generates intermediate language, which is then optimized, and back-end generates assembly
- Can be quite complex
- Examples: gcc, clang
- What language is a compiler written in? Who wrote the first one?

Tools – Assembler

- Takes assembly language and generates machine language
- creates object files
- Relatively easy to write
- Examples: GNU Assembler (gas), tasm, nasm, masm, etc.

Tools – Linker

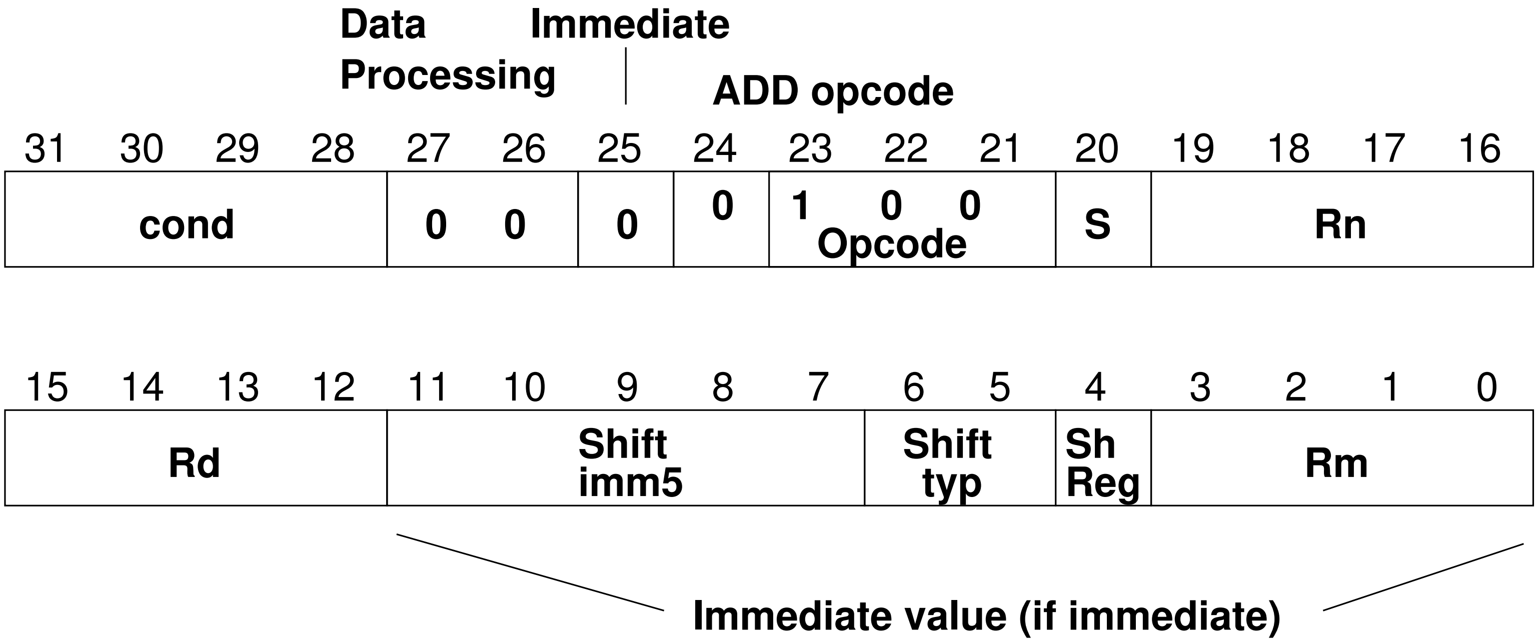
- Creates executable files from object files
- resolves addresses of symbols.
- Links to symbols in libraries.
- Examples: ld, gold

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**	**S**	**Rn**			
								Opcode							

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

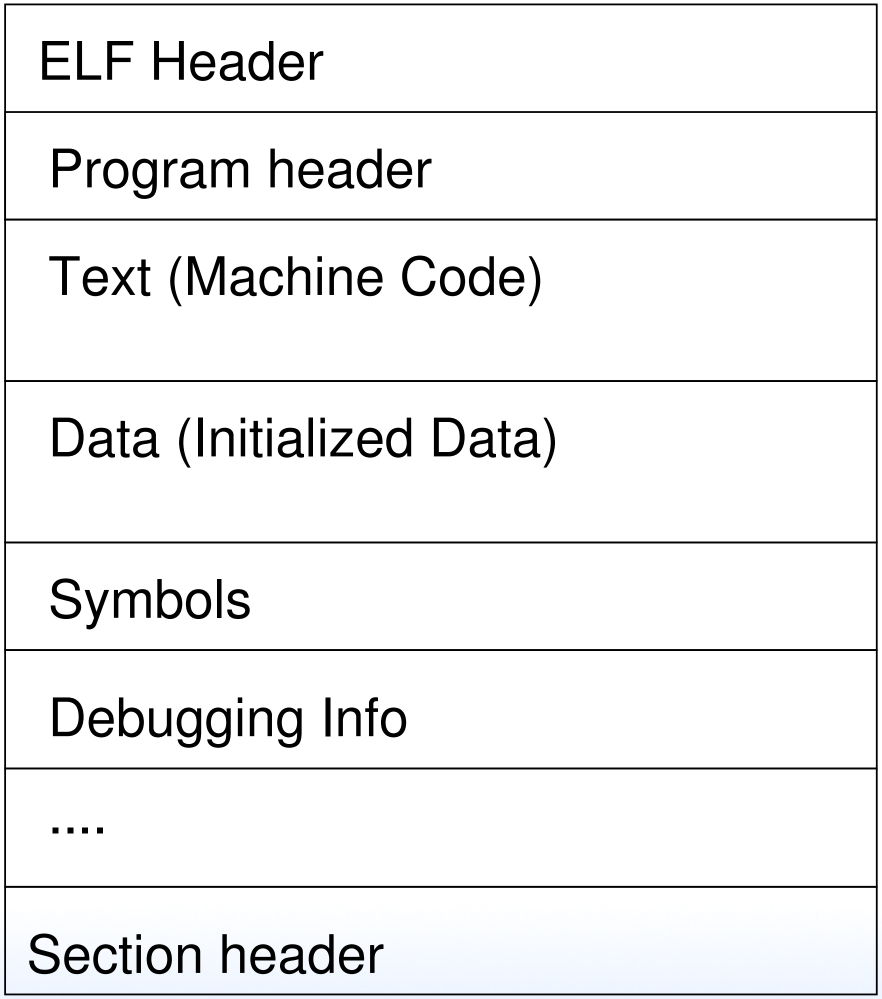
Immediate value (if immediate)

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

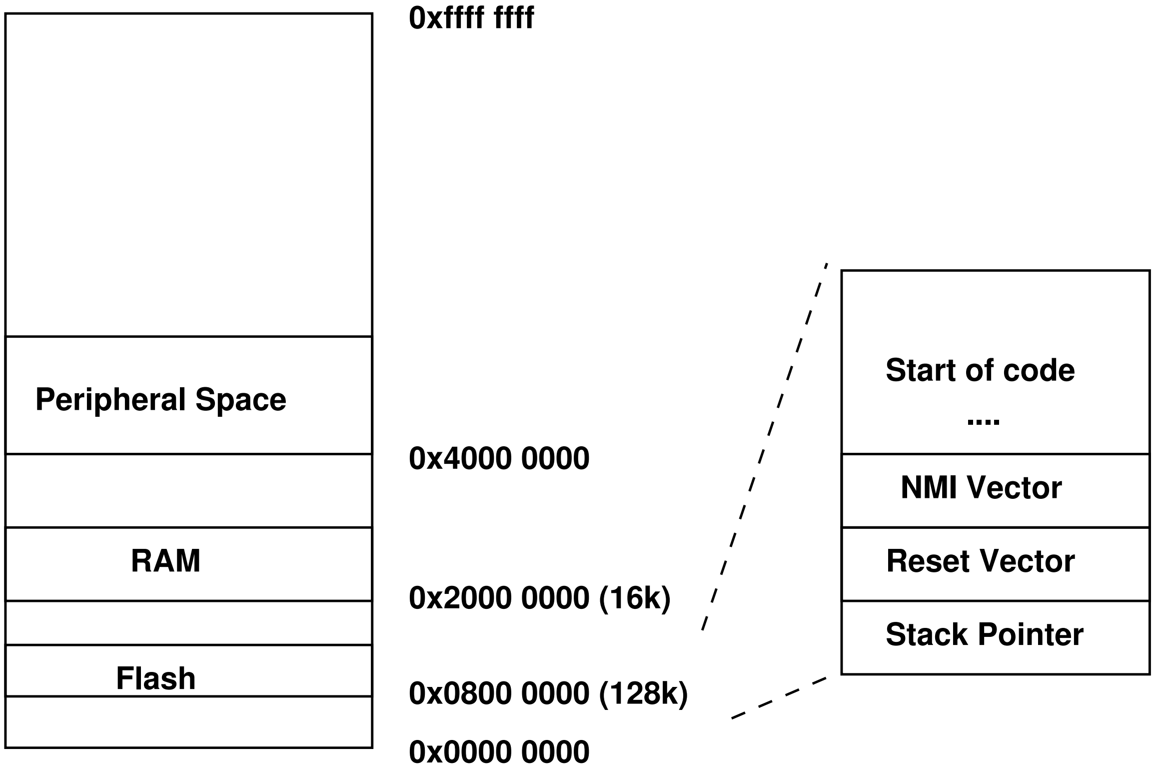
ELF Description

- 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

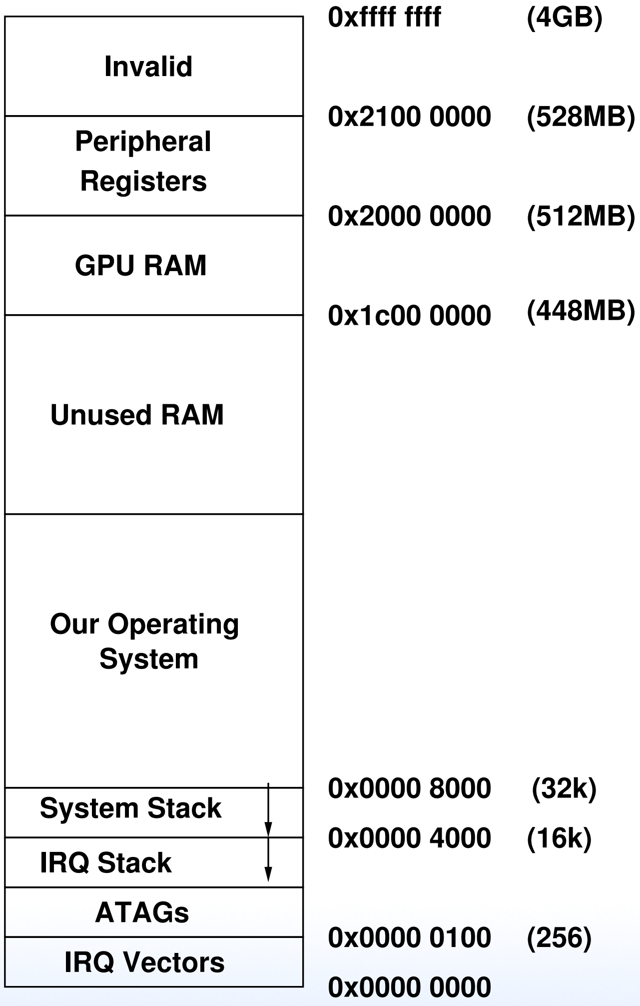
....

NMI Vector

Reset Vector

Stack Pointer

Raspberry Pi Physical Layout



Invalid

Peripheral Registers

GPU RAM

Unused RAM

Our Operating System

System Stack

IRQ Stack

ATAGs

IRQ Vectors

0xffff ffff (4GB)

0x2100 0000 (528MB)

0x2000 0000 (512MB)

0x1c00 0000 (448MB)

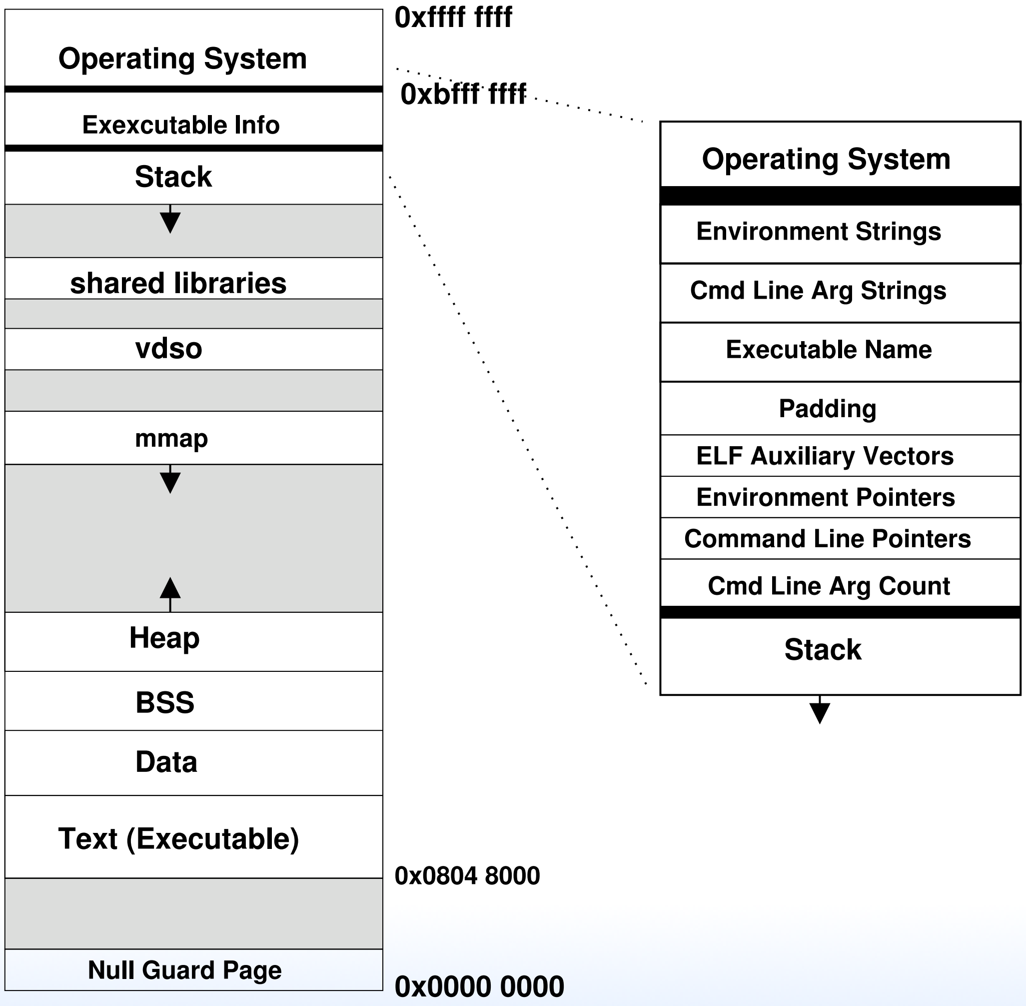
0x0000 8000 (32k)

0x0000 4000 (16k)

0x0000 0100 (256)

0x0000 0000

Linux 32-bit 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 (but security...)
- 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.

Brief overview of Virtual Memory

- Each program gets a flat 4GB (on 32-bit) memory space. The CPU and Operating system work together to provide this, even if not that much RAM is available and even though different processes seem to be using the same addresses.
- Physical vs Virtual Memory
- OS/CPU deal with "pages", usually 4kB chunks of memory.
- Every mem access has to be translated. The operating

system looks in the “page table” to see which physical address your virtual address maps to.

This is slow. That’s where TLB comes in; it caches pagetable translations. As long as you don’t run out of TLB entries this goes fast.

- Demand paging: the OS doesn’t have to load pages into memory until the first time you actually load/store them.
- Context Switch: when you switch to a new program, the TLB is flushed and a different page table is used to provide the new program its own view of memory.