ECE 471 – Embedded Systems
Lecture 12

Vince Weaver

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

vincent.weaver@maine.edu

29 September 2025

Announcements

- HW#4 was posted
- If you weren't here Friday and need the connector wires, be sure to grab some
- Also if you need any parts (LED, breadboard) let me know

Q from last time – Initializing a Struct to 0

- Using `memset()` definitely safe
- Using ={0} will work but is a bit of a hack
  (and I worry a bit about padding, which can be an issue when talking to the kernel like we are here)
  also possibly breaks with unions in C23
- Using ={} which is a brand new C23 way of doing it.

(Review) 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

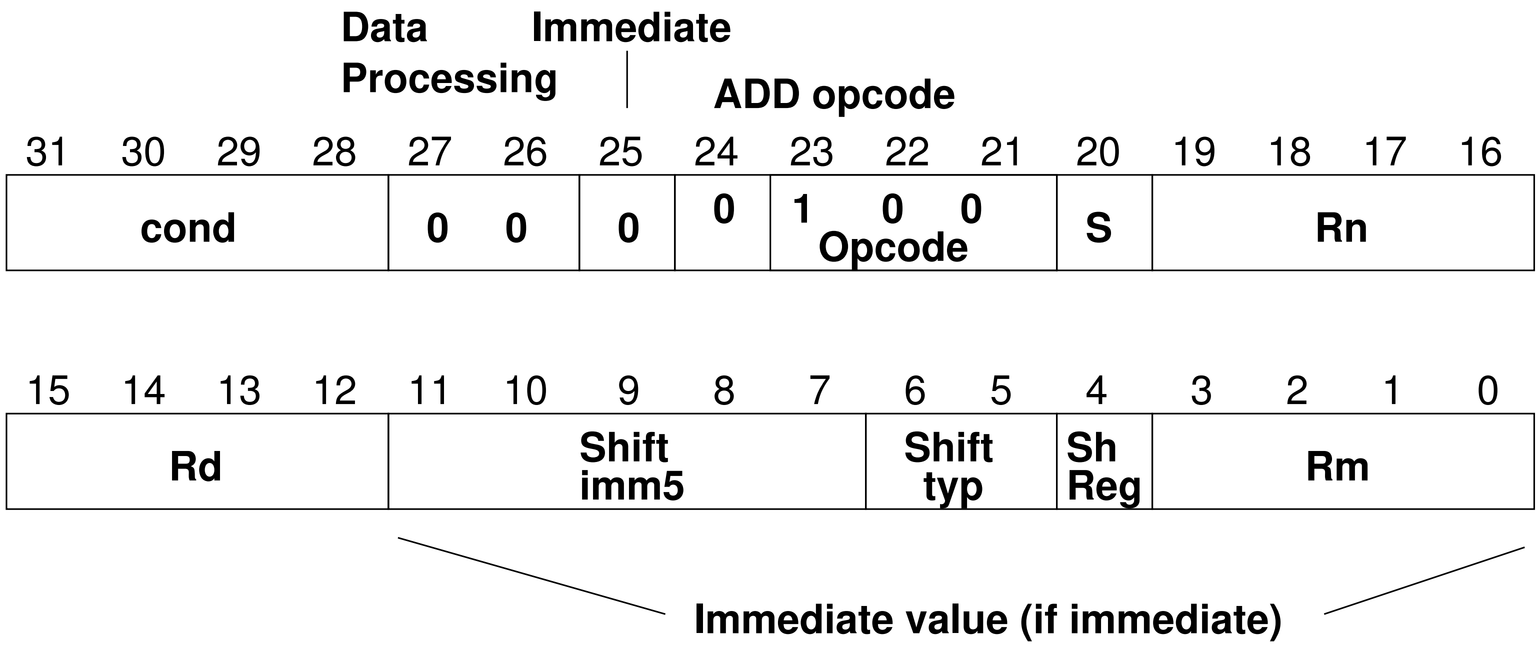
- Creates executable files from object files
- resolves addresses of symbols.
- Links to symbols in libraries.
- Examples: ld, gold

Converting Assembly to Machine Language

Thankfully the assembler does this for you.

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

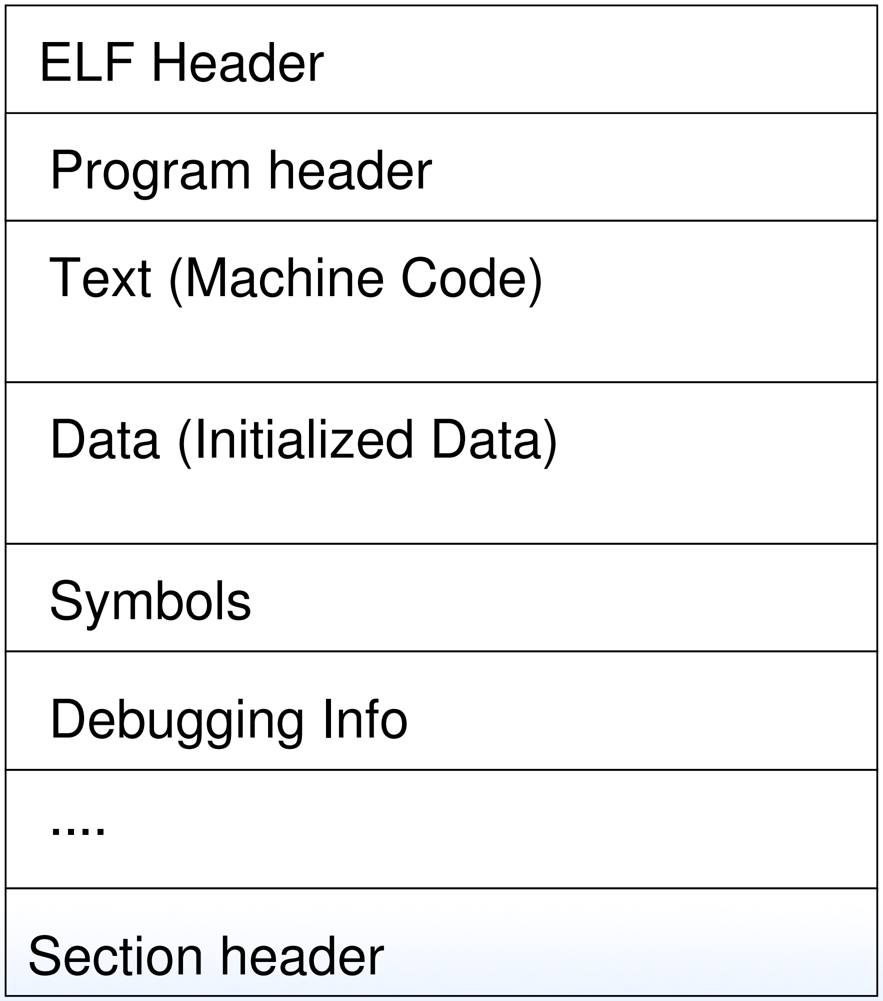
```
ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```


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
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

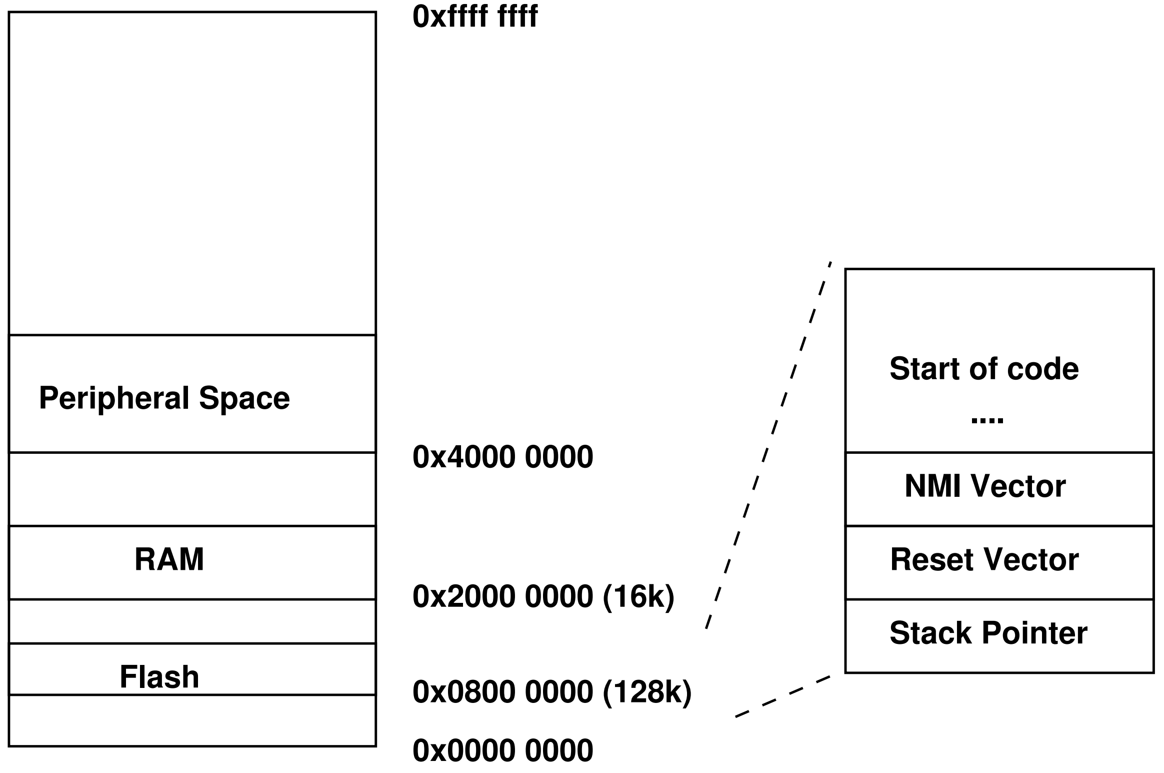
- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

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

Where do Programs Get Loaded?

- When you run your programs, where do they get loaded in memory?
- It's more obvious in low-end embedded systems with limited resources
- You (or your tools) specify exactly where in physical memory things go and where they run

STM32L-Discovery Physical Memory Layout

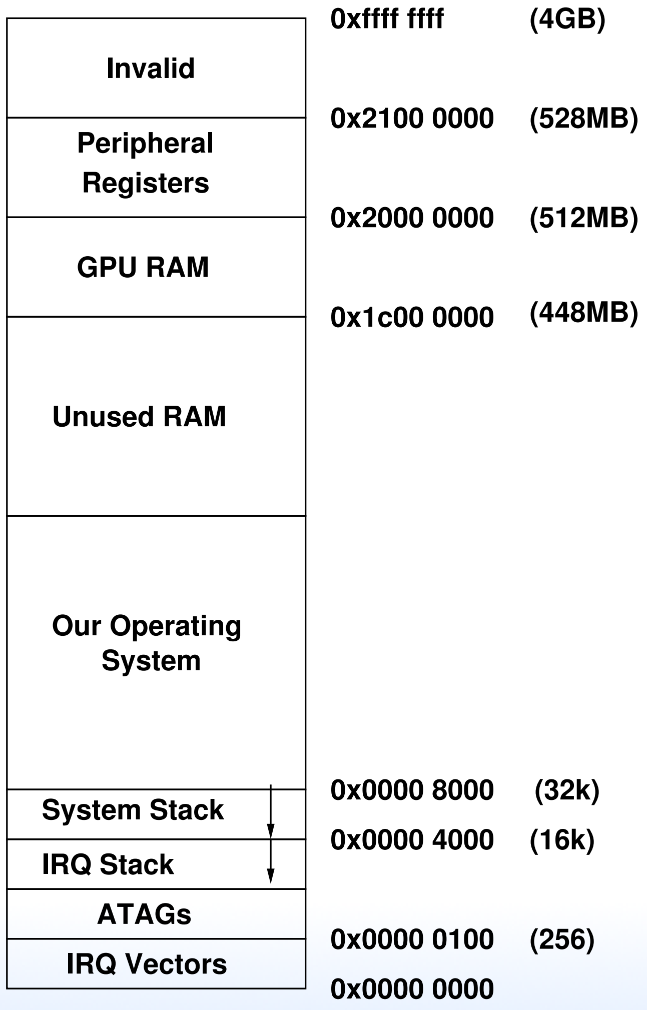


Where do Programs Get Loaded on Pi?

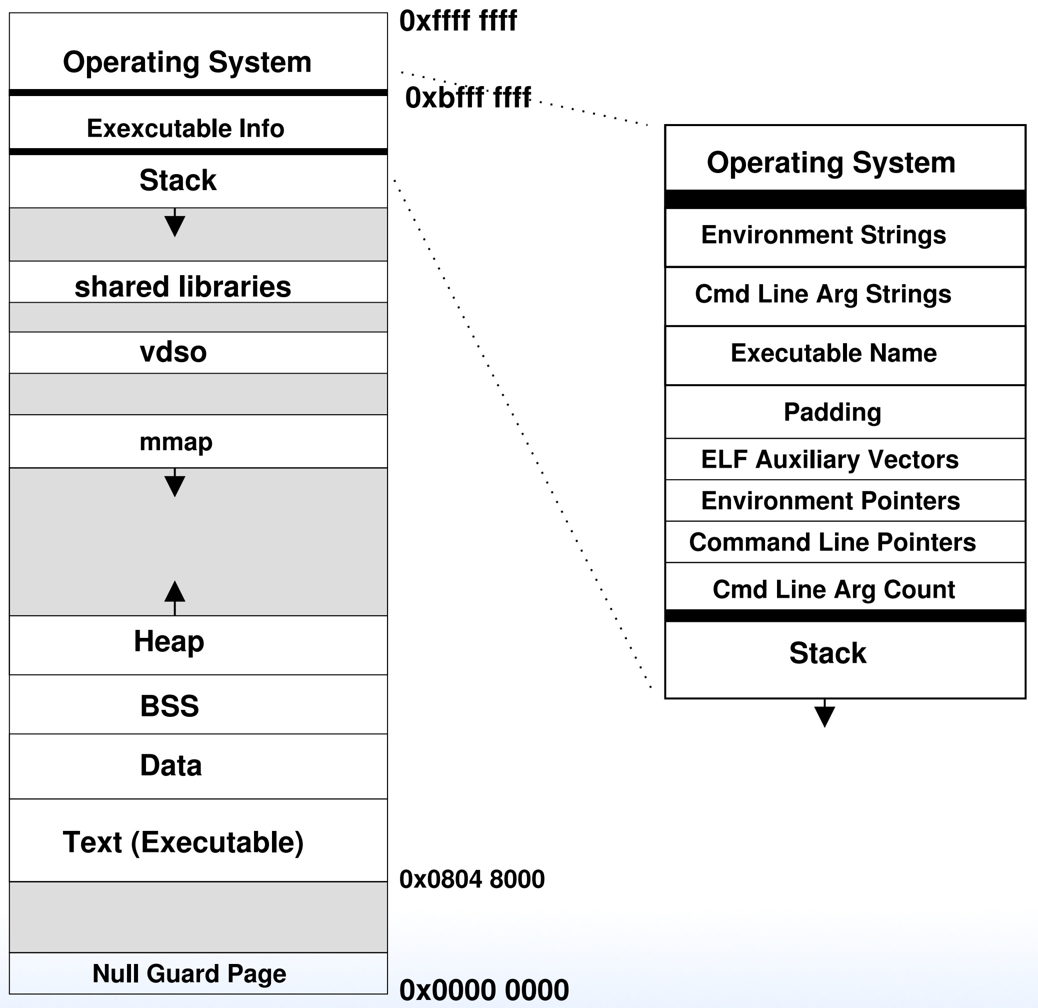
- You can also run a Pi bare-metal (see ECE531) but in this class we are going to run on top of an Operating System
- With an OS you might want to run more than one application at a time
- What happens if they are compiled to run at the same memory offset?
- You can use PIC (position independent code, which uses relative addresses) but that can be a pain

- With an OS instead you can use virtual memory

Raspberry Pi (32bit) Physical Layout



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.
- mmap() / shared libraries also go in there

Other Linux memory 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.

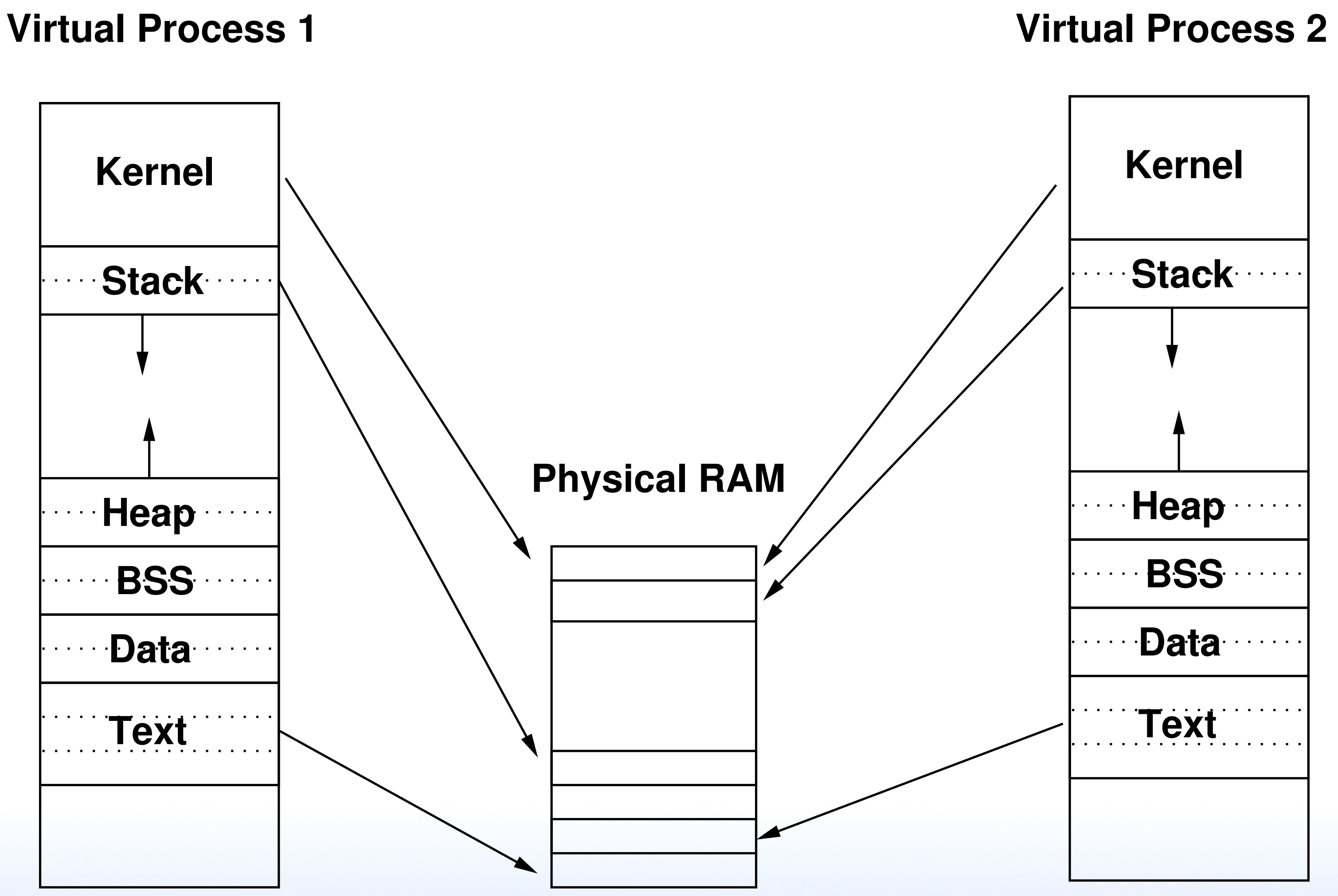
Brief overview of Virtual Memory

- Each program gets a flat 4GB (on 32-bit) view of memory
- CPU and Operating system work together to provide this illusion
- Program sees 4GB even if it doesn't have that much RAM (can make "virtual" memory out of disk)

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 “page table” to see which physical address your virtual address maps to
- This is slow. How to improve slow memory in CPU? Cache!
- TLB caches pagetable translations
- As long as you don’t run out of TLB entries this is fast.

Virtual / Physical Mapping



What if load/store address not in TLB?

- Walk the "page-tables" which are memory structures that describe currently known pages belonging to a process
- What if not in the page-tables? **PAGE FAULT**
  - ask OS if the address is valid, if so update page tables
  - If in text/data, load from disk
  - If in bss/heap alloc page of 0s
  - If near stack, autogrow it
  - If not valid, segfault

Benefits of Virtual Memory

- Disk swapping/paging: can use disk as (slow) “virtual” memory and move pages to and from disk into memory to give the illusion you have more
- 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.

- Flat memory space, all processes can start at same memory location without having to recompile
- Security: different processes can't see others memory (or over-write it)