ECE 531 – Advanced Operating Systems
Lecture 3

Vince Weaver

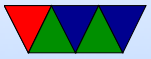
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

8 September 2025

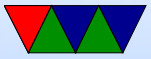
Announcements

- HW#1 was posted, due on Friday.
- Will hand out Pis later this week



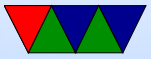
Operating Systems Types

- Monolithic kernel – everything in one big address space. Something goes wrong, lose it all. Faster
- Microkernel – separate parts that communicate by message passing. can restart independently. Slower.
- Microkernels were supposed to take over the world. Didn't happen. (GNU Hurd?)
- Famous Torvalds (Linux) vs Tannenbaum (Minix) flamewar



Linux on the Pi

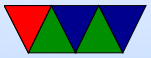
- Mainline kernel, bcm2835/bcm2836 tree
  Missing some features
- Raspberry-pi foundation bcm2708/bcm2709 tree
  More complete, not upstream
- Why everything not upstream? Common problem, especially on ARM. Getting upstream is hard, high standards. Takes patience and time, small one-off ARM boards do not have the resources for the process.



Compiler Followup Last Time

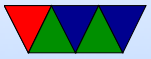
- Not really needed for this course, but interesting, and a bit sad UMaine doesn't offer a course
- Compilers often broken into two parts
- Front End
  - Parses the code and creates something called Intermediate Representation (IR)
  - Lexer/Parser, can use tools like lex/yacc (or on Linux, flex/bison)

    Lexer scans input, converts to tokens



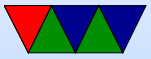
Parser processes the tokens

- Can also write own EBNF parser. As a last resort brute-force with C string parsing (but that's a pain)
- Recurses through code, so you have a for statement, it will get that, then the first expression it will call off and if that statement includes other statements it can nest
- This is where things get tricky. In C can have arrays inside of structs inside of arrays, etc, and infinitely nesting loops and if/else statements
- Nice thing about this, you can have different front-



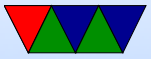
ends, C, C++, Java, FORTRAN, etc

- Back End
  - Take IR and converts it to assembly language
  - Can have multiple backends, so can take IR to x86, or ARM, or RiscV, etc. Also how cross-compilers work.
  - One tricky part is "register allocator". Most architectures have a limited number of registers so need to allocate variables to these in an optimal way
    Need to find out when register no-longer used so can re-use it.
    Also issues like various paths though if/else statements



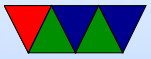
but need to make sure the final value for a variable is correct

- Optimizing Compiler
  - Before running the back-end can run multiple optimizing passes
  - Things like constant folding (2*2 = 4) or things like (x*2 to a left shift) or hoisting, unrolling, etc
  - This is where "undefined behavior" can be an issue as compiler authors love exploiting that in order to make optimizations better



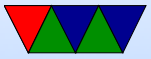
Brief history on C

- 1970-1972, Dennis Ritchie
  Systems language for UNIX, moving from PDP-7 to PDP-11
- Simplified version of BCPL ("B"), updated version called "C"
- K&R C book 1978. ANSI C 1989, C89/C90/C99/C11/C17/C23



How Small can a C compiler be?

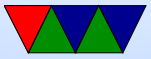
- The original PDP-11 wasn't exactly resource heavy
- Fabrice Bellard OTCC for the IOCCC, tiny C compiler fitting in 2k or so of source code



Compiling – how does it work?

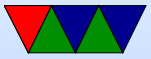
Traditionally this is how it works on gcc, others may vary.

- **compiler** takes C-code (.c), makes assembly language (.s)
- **assembler** takes assembly (.s), makes object file (.o or .obj) machine language
- **linker** takes object file, resolves addresses, arranges output based on *linker script*, creates executable
- Who wrote the first compiler? Assembler? Machine language?



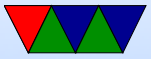
Tools

- compiler: we use gcc, others exist (intel, microsoft, llvm/clang)
- assembler: GNU Assembler as (others: tasm, nasm, masm, etc.)
- linker: ld



Converting C to assembly

- You can use `gcc -S` to have it dump out the assembly it makes
- The whole process is fairly complex, you can take whole classes on it.

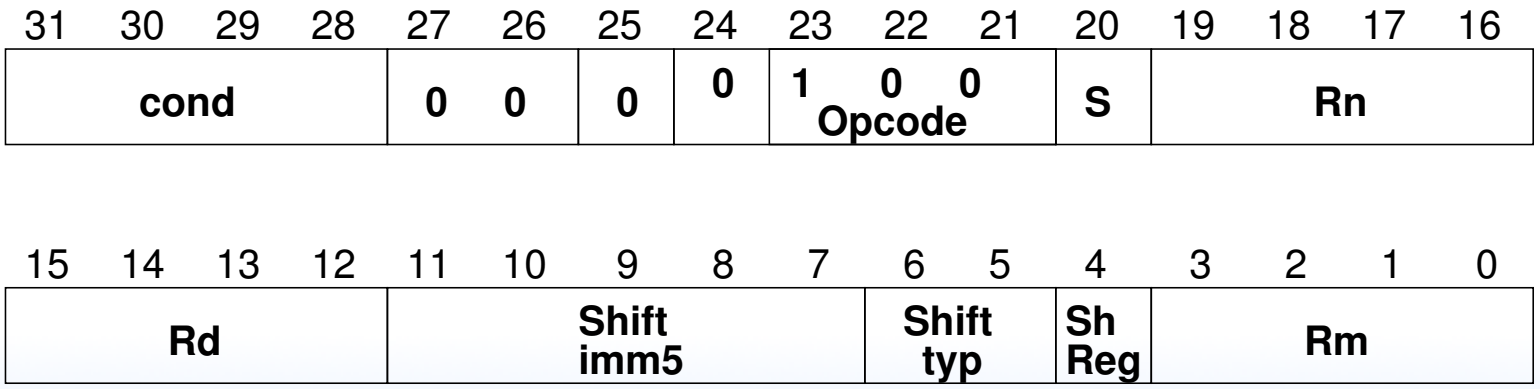


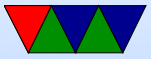
Converting Assembly to Machine Language

Thankfully the assembler does this for you.

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

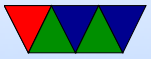
`ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}`



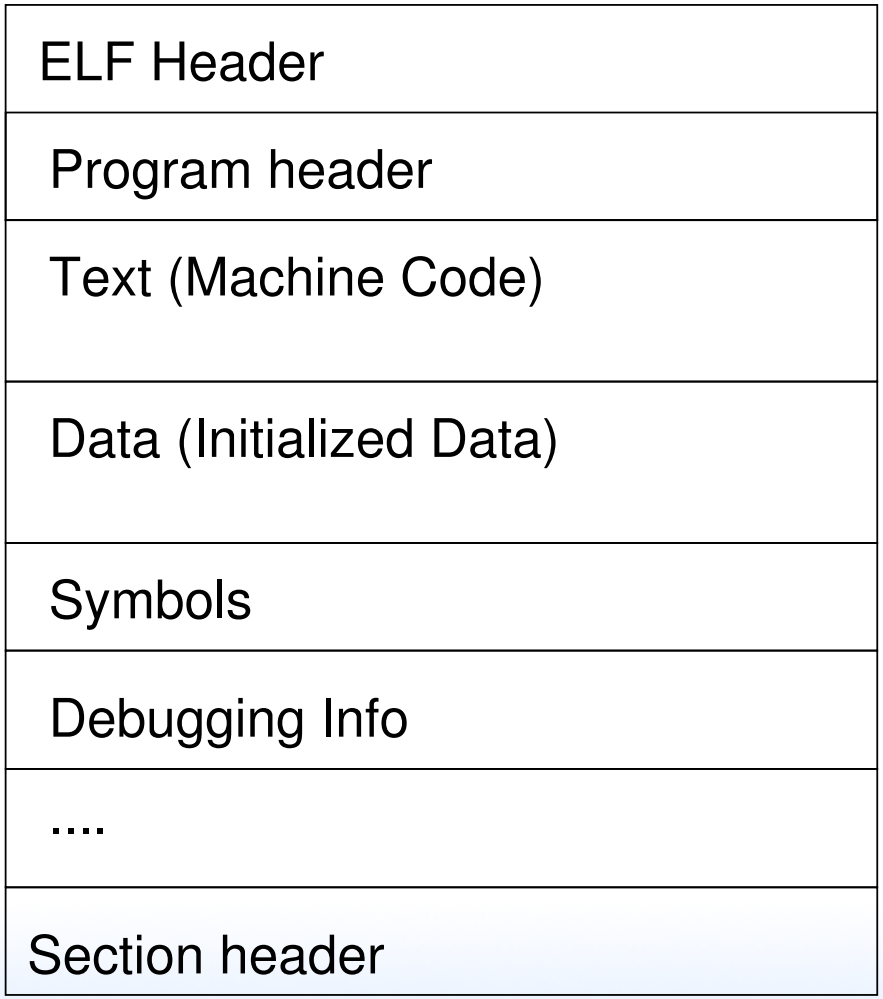


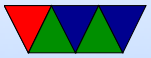
Executable Format

- 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
- B-flat format (used in this class)



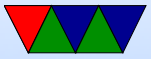
ELF Layout


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



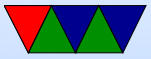
ELF Description

- 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



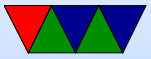
- Other data: things like symbol names, debugging info (DWARF), etc.
  DWARF backronym = "Debugging with Attributed Record Formats"

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



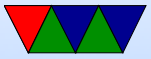
Use of ELF format

- Userspace programs will be built as ELF executables
- Kernel images often are too, though either the bootloader has to be aware of this or you might have to strip the header off before using it
- vmlinux on Linux



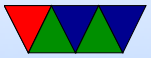
Cross-compiling

- Building for a different architecture
- Why? ARM machines often slow
- Why not? Source tree has to be handle this. Makefile. etc. Usually easier to compile natively
- Linux kernel tends to cross compile OK.



Booting a System

- Why is it called booting?
- Most likely source is the idea of “Pulling oneself up by ones bootstraps”, i.e., getting somewhere by starting with nothing



Simple Booting

- Simplest systems have code in ROM.
  The CPU initializes, points the Program Counter to a known location, and starts executing.

- The STM32L boards in ECE271 do something similar; code is in flash, reset vector (at offset 0) points at code to start. press reset, runs reset vector, up to you to do everything else.