Review

CS 2130: Computer Systems and Organization 1

November 5, 2025

Announcements

- Midterm 2 Friday (November 7, 2025) in class
  - Written, closed notes
  - If you have SDAC, please schedule ASAP
- No Quiz this Friday!
- No Homework this week!

The Stack

Stack - a last-in-first-out (LIFO) data structure

- *The* solution for solving this problem

`rsp` - Special register - the *stack pointer*

- Points to a special location in memory
- Two operations most ISAs support:
  - `push` - put a new value on the stack
  - `pop` - return the top value off the stack

The Stack: Push and Pop

`push r0`

- Put a value onto the "top" of the stack
  ```
  rsp -= 1
  M[rsp] = r0
  ```

`pop r2`

- Read value from "top", save to register
  ```
  r2 = M[rsp]
  rsp += 1
  ```

Patents and Copyright

Copyright

- "Everyone is a copyright owner. Once you create an original work and fix it, like taking a photograph, writing a poem or blog, or recording a new song, you are the author and the owner."
  from https://www.copyright.gov/what-is-copyright/

Patents and Copyright

Patent

- “Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title.”

  from 35 U.S.C. 101

Patents

In software and hardware, patents become messy

- Code is a description of a process we want the computer to do
- Do not have to implement the process to patent it

Question: Should we patent something like our ISA?
What is the current state of the art?

Common Approaches to Software

How can we get value from what we create?

- Copyright - distribute closed source software
- License Agreements (in contract law)
- Always innovate

Backdoors

Backdoor: secret way in to do new *unexpected* things

- Get around the normal barriers of behavior
- Ex: a way in to allow me to take complete control of your computer

Backdoors

Exploit - a way to use a vulnerability or backdoor that has been created

- Our exploit today: a **malicious payload**
  - A passcode and program
  - If it ever gets in memory, run my program regardless of what you want to do

Our Hardware Backdoor

Our backdoor will have 2 components

- Passcode: need to recognize when we see the passcode
- Program: do something bad when I see the passcode

Our Hardware Backdoor

Our Hardware Backdoor

Will you notice this on your chip?

- Modern chips have **billions** of transistors
- We're talking adding a few hundred transistors
- *Maybe with a microscope? But you'd need to know where to look!*

Our Hardware Backdoor

Have you heard about something like this before?

- Sounds like something from the movies
- People claim this might be happening
- To the best of my knowledge, no one has ever *admitted* to falling in this trap

Ethics, Business, Tech

Are there reasons to do this? Not to do this?

- No technical reason not to, it's easy to do!
- Ethical implications
- Business implications (lawsuits, PR, etc)

Can we make a system where one bad actor can't break it?

- Code reviews, double checks, verification systems, automated verification systems, …

Why?

Why does this work?

- **It's all bytes!**
- Everything we store in computers are bytes
- We store code and data in the same place: memory

It’s all bytes

Memory, Code, Data… It’s all bytes!

- **Enumerate** - pick the meaning for each possible byte
- **Adjacency** - store bigger values together (sequentially)
- **Pointers** - a value treated as address of thing we are interested in

Enumerate

Enumerate - pick the meaning for each possible byte

What is 8-bit `0x54`?

Unsigned integer	eighty-four
Signed integer	positive eighty-four
Floating point w/ 4-bit exponent	twelve
ASCII	capital letter T: `T`
Bitvector sets	The set $\{2, 3, 5\}$
Our example ISA	Flip all bits of value in r1

Adjacency

Adjacency - store bigger values together (sequentially)

- An array: build bigger values out of many copies of the same type of small values
  - Store them next to each other in memory
  - Arithmetic to find any given value based on index

Adjacency

Adjacency - store bigger values together (sequentially)

- Records, structures, classes
  - Classes have fields! Store them adjacently
  - Know how to access (add offsets from base address)
  - If you tell me where object is, I can find fields

Pointers

Pointers - a value treated as address of thing we are interested in

- A value that really points to another value
- Easy to describe, hard to use properly
- *We'll be talking about these a lot in this class!*

Pointers

Pointers - a value treated as address of thing we are interested in

- Give us strange new powers (represent more complicated things), e.g.,
  - Variable-sized lists
  - Values that we don’t know their type without looking
  - Dictionaries, maps

64-bit Machines

64-bit machine: The **registers** are 64-bits

- i.e., `r0`, but also `PC`

Important to have large values. Why?

- Most important: PC and memory addresses
- How much memory could our 8-bit machine access? 256 bytes
- Late 70s - 16 bits: 65,536 bytes
- 80s - 32 bits: $\approx$ 4 billion bytes
- Today's processors - 64 bits: 2^{64} addresses

Aside: Powers of Two

Powers of Two

Value	base-10	Short form	Pronounced
2^{10}	1024	Ki	Kilo
2^{20}	1,048,576	Mi	Mega
2^{30}	1,073,741,824	Gi	Giga
2^{40}	1,099,511,627,776	Ti	Tera
2^{50}	1,125,899,906,842,624	Pi	Peta
2^{60}	1,152,921,504,606,846,976	Ei	Exa

Example: 2^{27} bytes $= 2^7 \times 2^{20}$ bytes $= 2^7$ MiB $= 128$ MiB

64-bit Machines

How much can we address with 64-bits?

- 16 EiB (2^{64} addresses = $2^4 \times 2^{60}$)
- But I only have 8 GiB of RAM

A Challenge

There is a disconnect:

- Registers: 64-bits values
- Memory: 8-bit values (i.e., **1 byte** values)
  - Each address addresses an 8-bit value in memory
  - Each address points to a 1-byte slot in memory
- How do we store a 64-bit value in an 8-bit spot?

Rules

Rules to break "big values" into bytes (memory)

1. Break it into bytes
2. Store them adjacently
3. Address of the overall value = smallest address of its bytes
4. Order the bytes
    - If parts are ordered (i.e., array), first goes in smallest address
    - Else, hardware implementation gets to pick (!!)
        - Little-endian
        - Big-endian

Ordering Values

Little-endian

- Store the low order part/byte first
- Most hardware today is little-endian

Big-endian

- Store the high order part/byte first

Example

Store `[0x1234, 0x5678]` at address `0xF00`

Endianness

Why do we study endianness?

- It is **everywhere**
- It is a source of weird bugs
- Ex: It's likely your computer uses:
  - Little-endian from CPU to memory
  - Big-endian from CPU to network
  - File formats are roughly half and half

Assembly

General principle of all **assembly languages**

- Code (text, not binary!)
- 1 line of code = 1 machine instruction
- One-to-one reversible mapping between binary and assembly
  - We do not need to remember binary encodings!
  - A program will turn text to binary for us!

Assembly

Features of assembly

- Automatic addresses - use **labels** to keep track of addresses
  - Assembler will remember location of labels and use where appropriate
  - Labels will not exist in machine code
- Metadata - data about data
  - Data that helps turn assembly into code the machine can use
- As complicated as machine instructions
  - There are a lot of instructions, and it is one-to-one!

Assembly Languages

There are many assembly languages

- But, they're backed by hardware!
- Two big ones these days: x86-64 and ARM
  - You likely have machines that use one of these
- Others: RISC-V, MIPS, ...

We will focus on **x86-64**

x86-64

x86-64 has a weird and long history

- Expansion of the 8086 series (Intel)
  - 8086, 8286, 8386, 8486, x86
- AMD expanded it with AMD64
- Intel decide to use same build, but called it x86-64
- Backwards compatible with the 8086 series

x86-64

Two dialects - two ways to write the same thing

- Intel - likely using with Windows
  `mov QWORD PTR [rdx+0x227],rax`
- AT&T - likely using with anything else
  `movq %rax,0x227(%rdx)`

We will use AT&T dialect

AT&T x86-84 Assembly

`instruction source, destination`

- Instruction followed by `0` or more operands (arguments)
- 4 types of operands:
  - Number (immediate value): `$0x123`
  - Register: `%rax`
  - Address of memory: `(%rax)` or 24 or `labelname`
  - Value at an address in memory: `(%rax)` or 24 or `labelname`

AT&T x86-84 Assembly

`mylabelname:`

- Label - remember the address of next thing to use later

`.something something`

- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

`// we can have comments!`

Addressing Memory

`2130(%rax, %rsp, 8)`

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `2130 + %rax + (%rsp * 8)`
- Common usage from this example:
  - `rax` - address of an object in memory
  - `2130` - offset of an array into the object
  - `rsp` - index into the array
  - `8` - size of the values in the array
- Don't need all parts: `(%rax)` or `(%rax, 4)` or `4(%rax)`
- This is all one operand (one memory address)

Registers

`rax` is a 64-bit register

Instructions

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move
  - `q` = quad word
- `movl` - 32-bit move
  - `l` = long
- There are encodings for shorter things, but we will mostly see 32- and 64-bit

More powerful than our ISA

Instructions can move/operate between memory and register

- `movq %rax, %rcx` - register to register
  - Remember our icode 0
- `movq (%rax), %rcx` - memory to register
  - Remember our icode 3
- `movq %rax, (%rcx)` - register to memory
  - Remember our icode 4
- `movq $21, %rax` - Immediate to register
  - Remember our icode 6 (b=0)

Note: at most one memory address per instruction

Other Instructions

Other instructions work the same way

- `addq %rax, %rcx` — rcx += rax
- `subq (%rbx), %rax` — rax -= M[rbx]
- xor, and, and others work the same way!
- Assembly has virtually no 3-argument instructions
  - All will be modifying something (i.e., `+=`, `&=`, …)

Load Effective Address

Load effective address: `leaq 4(%rcx), %rax`

- Performs memory address calculation
- Stores address, not value at the address in memory

Jumps

`jmp foo`

- Unconditional jump to `foo`
- `foo` is a label or memory address
- Need `jmp*` to use register value

Conditional jumps

- `jl`, `jle`, `je`, `jne`, `jg`, `jge`, `ja`, `jb`, `js`, `jo`

Unlike our Toy ISA, these do not compare given register to 0

Jumps

Condition codes - 4 1-bit registers set by every math operation, `cmp`, and `test`

- Result for the operation compared to 0 (if no overflow)
- Example:

```
addq $-5, %rax
// ...code that doesn't set condition codes...
je foo
```

  - Sets condition codes from doing math (subtract 5 from rax)
  - Tells whether result was positive, negative, 0, if there was overflow, ...
  - Then jump if the result of operation should have been $= 0$

Jumps: compare...

`cmpq %rax, %rdx`

- Compare checks result of `-=` and sets condition codes
- How `rdx - rax` compares with 0
- Be aware of ordering!
  - if `rax` is bigger, sets < flag
  - if `rdx` is bigger, sets > flag

Jumps: ... and test

```
testq %rax, %rdx
```

- Sets the condition codes based on `rdx & rax`
- Less common

Neither save their result, just set condition codes!

Function Calls: Calling Conventions

`callq myfun`

- Push return address, then jump to myfun
- Convention: Store arguments in registers and stack before call
  - First 6 arguments (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - If more arguments, pushed onto stack (last to first)

`retq`

- Pop return address from stack and jump back
- Convention: store return value in `rax` before calling `retq`

This is similar to our Toy ISA's function calls in homework 4

Calling Conventions: Registers

Calling conventions - recommendations for making function calls

- Where to put arguments/parameters for the function call?
- Where to put return value? in `rax` before calling `retq`
- What happens to values in the registers?
  - **Callee-save** - The function should ensure the values in these registers are unchanged when the function returns
    - `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`
  - **Caller-save** - Before making a function call, save the value, since the function may change it

Debugger

Debugger - step through code!

- You will be using this for lab 7
- Experience seeing results of these instructions step-by-step
- **Please read the x86-64 summary reading before lab!**

Most Common Instructions

- `mov` - =
- `lea` - load effective address
- `call` - push PC and jump to address
- `add` - +=
- `cmp` - set flags as if performing subtract
- `jmp` - unconditional jump
- `test` - set flags as if performing &
- `je` - jump iff flags indicate == 0
- `pop` - pop value from stack
- `push` - push value onto stack
- `ret` - pop PC from the stack

Compilation Pipeline

Turning our code into something that runs

- **Pipeline** - a sequence of steps in which each builds off the last

C

C is a thin wrapper around assembly

- This is by design!
- Invented to write an operating system
  - Can write inline assembly in C
- Many other languages decided to look like C

Compilation Pipeline

Earlier, we saw:

- C files (`.c`) compiled to assembly (`.s`)
- Assembly (`.s`) assembled into object files (`.o`)
- Object files (`.o`) linked into a program / executable

Compiling C to Assembly

Multiple stages to compile C to assembly

- Preprocess - produces C
  - C is actually implemented as 2 languages:
    C preprocessor language, C language
  - Removes comments, handles preprocessor directives (`#`)
  - `#include`, `#define`, `#if`, `#else`, ...
- Lex - breaks input into individual tokens
- Parse - assembles tokens into intended meaning (parse tree)
- Type check - ensures types match, adds casting as needed
- Code generation - creates assembly from parse tree

Errors

Compile-time errors

- Errors we can catch during compilation (this process)
- **Before** running our program

Runtime errors

- Errors that occur when running our programs

Simple C Example

```
int main() {
    return 0;
}
```

The `main` function

- Start running the `main()` function
- `main` must return an integer - **exit code**
  - 0 = everything went okay
  - Anything else = something went wrong
- There *should* be arguments to main

Data Types in C

Integer data types

Data type	Size
char	
short	
int	
long	
long long	

Each has 2 versions: *signed* and *unsigned*

Data Types in C

Floating point

- float
- double

Data Types in C

Pointers - how C uses addresses!

- Hold the address of a position in memory
- Need to know the kind of information stored at that location

Example

```c
int main() {
    int x = 3;
    long y = 4;
    int *a = &x;
    long *b = &y;
    long z = *a;
    int w = *b;
    return 0;
}
```

```
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   31 c0                   xor    %eax,%eax
   6:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   d:   c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  14:   48 c7 45 f0 04 00 00    movq   $0x4,-0x10(%rbp)
  1b:   00
  1c:   48 8d 4d f8             lea    -0x8(%rbp),%rcx
  20:   48 89 4d e8             mov    %rcx,-0x18(%rbp)
  24:   48 8d 4d f0             lea    -0x10(%rbp),%rcx
  28:   48 89 4d e0             mov    %rcx,-0x20(%rbp)
  2c:   48 8b 4d e8             mov    -0x18(%rbp),%rcx
  30:   48 63 09                movslq (%rcx),%rcx
  33:   48 89 4d d8             mov    %rcx,-0x28(%rbp)
  37:   48 8b 4d e0             mov    -0x20(%rbp),%rcx
  3b:   48 8b 09                mov    (%rcx),%rcx
  3e:   89 4d d4                mov    %ecx,-0x2c(%rbp)
  41:   5d                      pop    %rbp
  42:   c3                      retq
```

Arrays

Array: 0 or more values of same type stored contiguously in memory

- Declare as you would use: `int myarr[100];`
- `sizeof(myarr)` = 400 — 100 4-byte integers
- `myarr` treated as pointer to first element
- Can declare array literals:
  `int y[5] = {1, 1, 2, 3, 5}`

Pointers and Arrays

`*myarr` and `myarr[0]` are equivalent

- Pointer to single value and pointer to first value in array
- Treat array as pointer to the first value (lowest address)
- Indexing into array: `myarr[n]` and `*(myarr+n)`
  - If `myarr` is an `int *`, then `myarr+1` points to **next int** in memory
  - Adding 1 to pointer adds `sizeof()` the type we're pointing to

Pointers and Arrays

Consider: `int **a`

Pointers

- All pointers are the same size: address size in underlying ISA
- Two special integer types (defined using `typedef`)
  - `size_t` - integer the size of a pointer (unsigned)
  - `ssize_t` - integer the size of a pointer (signed)
  - With our compiler and ISA, these are both variants of `long`

Pointers

Consider the following code:

```
int x = 10;
int *y = &x;
int *z = y + 2;
long w = ((long)z) - ((long)y);
```

Why is `w` = 8?

Other Types and Values

- Literal values - integer literals are implicitly cast
  - `unsigned long very_big = 9223372036854775808uL`
  - `u` for unsigned, `L` for long
- `enum` - named integer constants (in ascending order)
  - ```
    enum { a, b, c, d=100, e };
    int foo = e;
    ```
- `void` - a byte with no meaning or ”nothing”
  - Pointers: `void *p`
  - Return values: `void myfunction();`
    ```

Other Types and Values

- Casting - changing type, converting
  - Integer: zero- or sign-extend or truncate to space
  - Int to float: convert to nearby representable value
  - Float to int: truncate remainder (no rounding)

Structures

`struct` - Structures in C

- Act like Java classes, but no methods and all public fields
- Stores fields adjacently in memory (but may have padding)
- Compiler determines padding, use `sizeof()` to get size
- Name of the resulting type includes word `struct`

```
struct foo {
    long a;
    int b;
    short c;
    char d;
};

struct foo x;
x.b = 123;
x.c = 4;
```

Structure Literals

```
struct a {
    int b;
    double c;
};

/* Both of the following initialize b to 0 and c to 1.0 */
struct a x = { 0, 1.0 };
struct a y = { .b = 0, .c = 1.0 };
```

typedef

`typedef` - give new names to any type!

- Fairly common to see several names for same data type to convey intent
- Ex: `unsigned long` may be `size_t` when used in sizes
- Examples:
  ```
  typedef int Integer;
  Integer x = 4;
  typedef double ** dpp;
  ```
- Used with *anonymous structs*:
  ```
  typedef struct { int x; double y; } foo;
  foo z = { 42, 17.4 };
  ```

Calling Functions

The C code

```
long a = f(23, "yes", 34uL);
```

compiles to

```
movl $23, %edi
leaq label_of_yes_string, %rsi
movq $34, %rdx
callq f
# %rax is "long a" here
```

without respect to how `f` was defined. It is the calling convention, not the type declaration of `f`, that controls this.

Calling Functions

But, if the C code has access to the type declaration of `f`, then it might perform some implicit casting first; for example, if we declared

```
long f(double a, const char *b, double c);
```

```
long a = f(23, "yes", 34uL);
```

then the call would be interpreted by C as having implicit casts in it:

```
long a = f((double)23, "yes", (double)34uL);
```

Calling Functions

and the arguments would be passed in floating-point registers, like so:

```
movl $23, %eax
cvtsi2sd %eax, %xmm0             # first floating-point argument

leaq label_of_yes_string, %rdi # first integer/pointer argument

movl $34, %eax
cvtsi2sd %eax, %xmm1             # second floating-point argument

callq f
# %rax is "long a" here
```

Function Declaration

```
int f(int x);
```

- Declaration of the function
- Function header
- Function signature
- Function prototype

We want this in every file that invokes `f()`

Function Definition

```
int f(int x) {
    return 2130 * x;
}
```

- Definition of the function

We only want this in **one** `.c` file

- Do not want 2 definitions
- Which one should the linker choose?