

Stacks and Functions, Backdoors

CS 2130: Computer Systems and Organization 1

February 27, 2023

Announcements

- Homework 4 due **Friday** at 11pm on Gradescope
- Lab tomorrow (git and assembly)
- Exam 1 grades coming soon

Our Instruction Set Architecture

icode	b	meaning
0		$rA = rB$
1		$rA += rB$
2		$rA \&= rB$
3		$rA =$ read from memory at address rB
4		write rA to memory at address rB
5	0	$rA = \sim rA$
	1	$rA = -rA$
	2	$rA = !rA$
	3	$rA = pc$
6	0	$rA =$ read from memory at $pc + 1$
	1	$rA +=$ read from memory at $pc + 1$
	2	$rA \&=$ read from memory at $pc + 1$
	3	$rA =$ read from memory at the address stored at $pc + 1$ For icode 6, increase pc by 2 at end of instruction
7		Compare rA as 8-bit 2's-complement to θ if $rA \leq \theta$ set $pc = rB$ else increment pc as normal

What about real ISAs?

Our Instruction Set Architecture

What about our ISA?

- Enough instructions to compute what we need
- As is, lot of things that are painful to do
 - This was on purpose! So we can see limitations of ISAs early
- Add any number of new instructions using the reserved bit (7)
- Missing something important: *Help to put variables in memory*

Storing Variables in Memory

So far... we/compiler chose location for variable

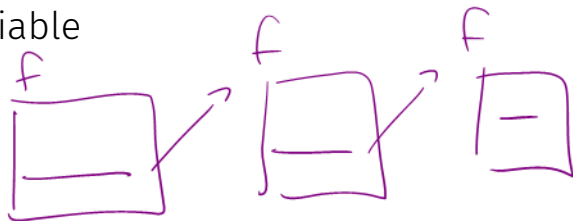
Consider the following example:

f(x):

a = x

if (x <= 0) return 0

else return f(x-1) + a



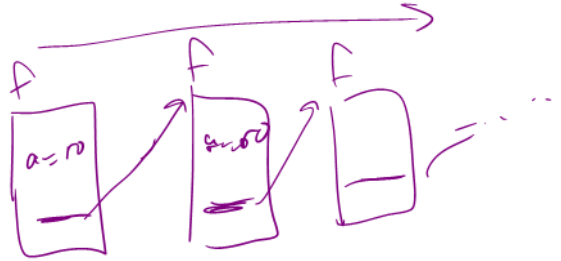
Recursion

- The formal study of a function that calls itself

Storing Variables in Memory

```
f(x):  
  a = x  
  if (x <= 0) return 0  
  else return f(x-1) + a
```

Where do we store a?



~~0~~
~~m[x-1]~~

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

$rsp -= 1$

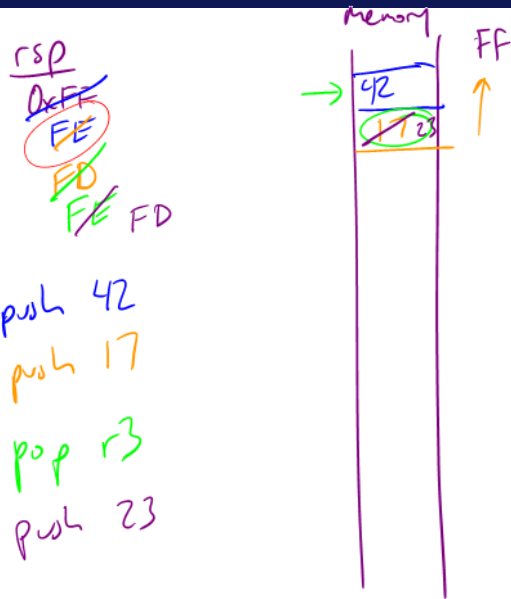
$M[rsp] = r0$

pop r2

- Read value from "top", save to register

$r2 = M[rsp]$

$rsp += 1$



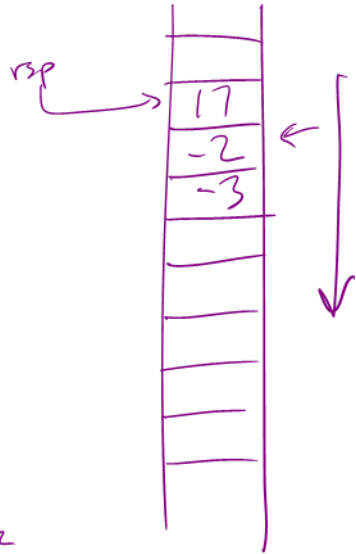
The Stack: Push and Pop

push(17)
push(23)
x = pop r0
push(-2) ←
push(-3)

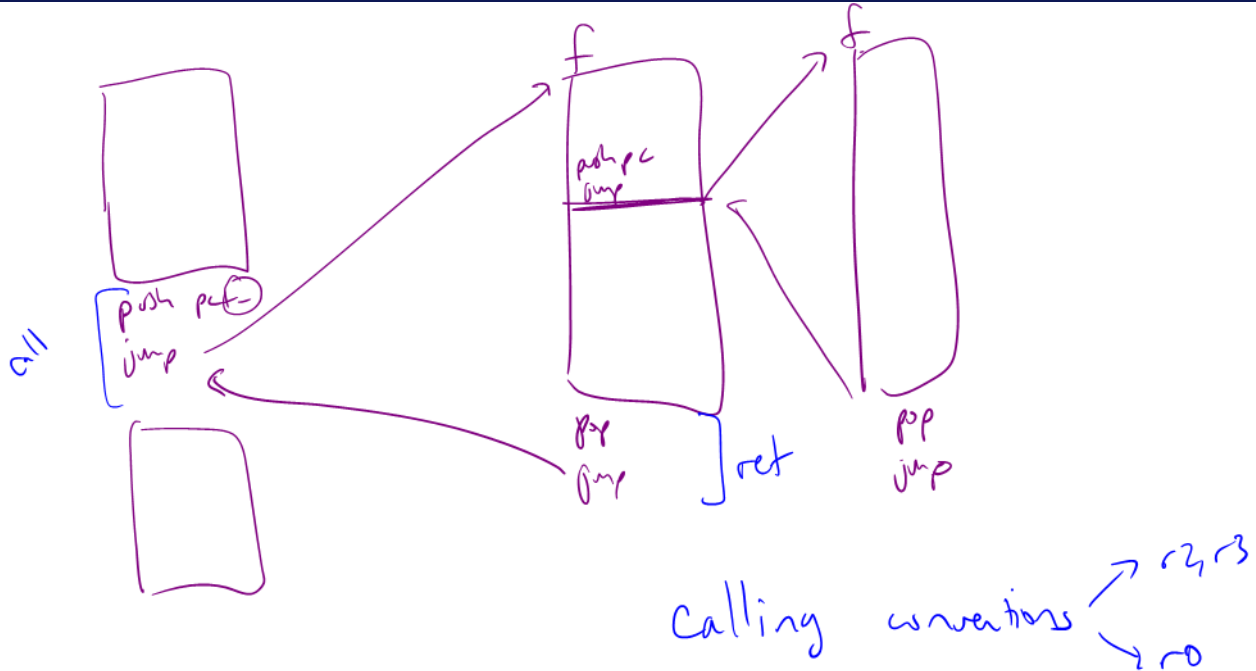
y = pop r1
z = pop r2

M[rsp]
rsp+1

<u>r0/x</u>	<u>r1/y</u>	<u>r2/z</u>
23	-3	-2



Function Calls



A short aside...

Time to take over the world!

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

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

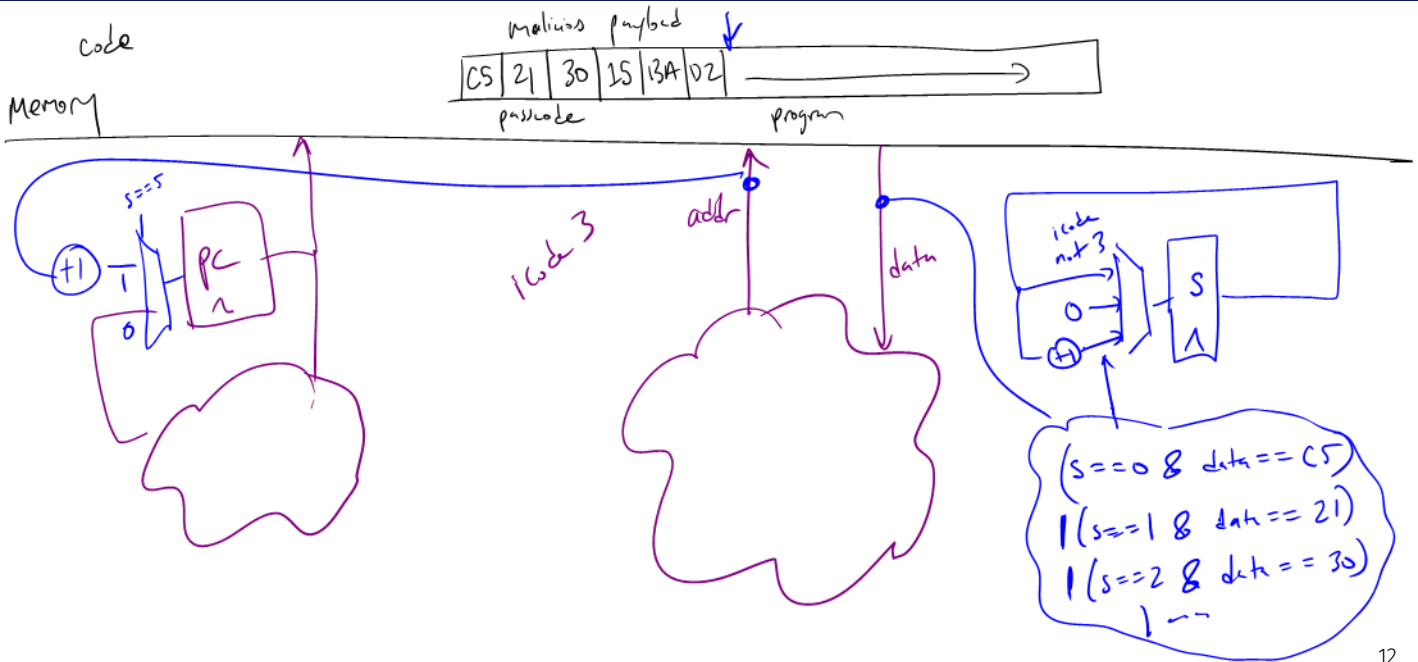
- Our exploit today: a **malicious payload**
 - A passcode and program
 - If it ever gets in memory, run my program regardless of what you want to do

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?

Our Hardware Backdoor

Will you notice this on your chip?

- Modern chips have **billions** of transistors
- We're talking adding a few hundred transistors

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?

Our Hardware Backdoor

Have you heard about something like this before?

- Sounds like something from the movies

Our Hardware Backdoor

Have you heard about something like this before?

- Sounds like something from the movies
- People claim this might be happening

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

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

- No technical reason not to, it's easy to do!

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)

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?

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 does this work?

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 - 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
- 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!*
- 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

Programs Use These!

How do our programs use these?

- Enumerated icodes, numbers
- Adjacenty stored instructions (PC+1)
- Pointers of where to jump/goto (addresses in memory)

Moving On

icode	b	meaning
0		$rA = rB$
1		$rA += rB$
2		$rA \&= rB$
3		$rA =$ read from memory at address rB
4		write rA to memory at address rB
5	0	$rA = \sim rA$
	1	$rA = -rA$
	2	$rA = !rA$
	3	$rA = pc$
6	0	$rA =$ read from memory at $pc + 1$
	1	$rA +=$ read from memory at $pc + 1$
	2	$rA \&=$ read from memory at $pc + 1$
	3	$rA =$ read from memory at the address stored at $pc + 1$ For icode 6, increase pc by 2 at end of instruction
7		Compare rA as 8-bit 2's-complement to θ if $rA \leq \theta$ set $pc = rB$ else increment pc as normal

So far, we've been dealing with an 8-bit machine!

64-bit Machines

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

- i.e., r0, but also PC

Important to have large values. Why?

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?

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

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:

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

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:

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

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:

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

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

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?

64-bit Machines

How much can we address with 64-bits?

- 16 EiB (2^{64} addresses = $2^4 \times 2^{60}$)

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

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 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