

More Variables, Arrays, and More

CS 2130: Computer Systems and Organization 1

February 20, 2023

Announcements

- Homework 3 due **Wednesday** at 11pm on Gradescope
- Exam 1 Friday in class, review on Wednesday
 - Review session in class on Wednesday (bring questions!)
 - Pen/pencil and paper, we'll provide scratch paper
 - No calculator needed

High-level Instructions

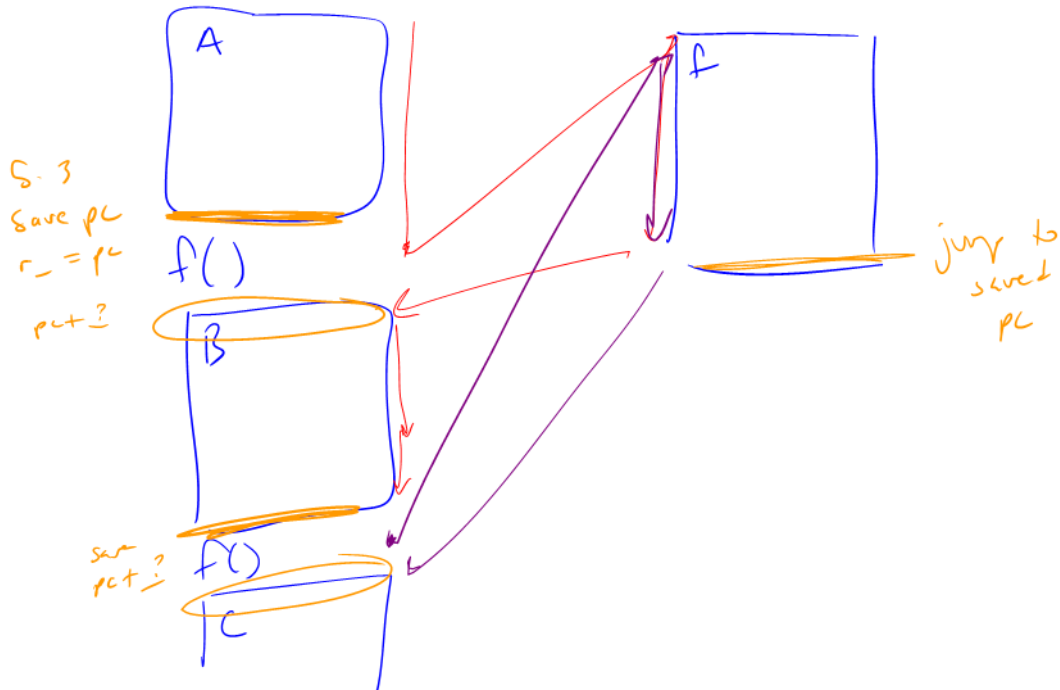
In general, 3 kinds of instructions

- **moves** - move values around without doing “work”
- **math** - broadly doing “work”
- **jumps** - jump to a new place in the code

Jumps

- Moves and math are large portion of our code
- We also need **control constructs**
 - Change what we are going to do next
 - **if, while, for**, functions, ...
- Jumps provide mechanism to perform these control constructs
- We jump by assigning a new value to the program counter PC

Function Calls



What kinds of things do we put in memory?

- Code: binary code like instructions in our example ISA
 - Intel/AMD compatible: x86_64
 - Apple Mx and Ax, ARM: ARM
 - And others!
- Variables: we may have more variables that will fit in registers
- Data Structures: organized data, collection of data
 - Arrays, lists, heaps, stacks, queues, ...

Dealing with Variables and Memory

What if we have many variables? Compute: $x += y$; $x += z$; $x += m[m[r3]]$

$x = 0x80$
 $y = 0x81$
 $z = 0x82$
 $t = 0x83$
 $w = 0x84$
 $f0 = 0x85$

$r3 = 0x80$
 $m[r3] = r0$

load $\begin{cases} r0 = m[0x80] & 6\ 3\ 80 \\ r1 = m[0x81] & 6\ 7\ 81 \end{cases}$

$x += y$

$r0 += r1$

write $\begin{cases} m[0x80] = r0 \\ m[0x81] = r1 \end{cases}$

$r0 = m[0x80]$

$x += z$

Arrays

Array: a sequence of values (collection of variables)

In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

Arrays

Array: a sequence of values (collection of variables)

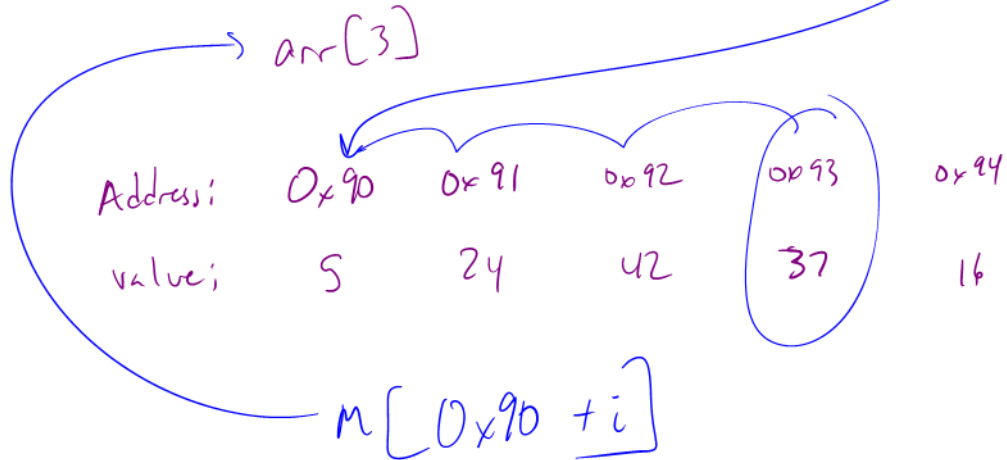
In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

How do we store them in memory?

Arrays

$arr = [5, 24, 42, 37, 16]$ @ $0x90$



Storing Arrays

In memory, store array sequentially

- Pick address to store array
- Subsequent elements stored at following addresses
- Access elements with math

Example: Store array *arr* at **0x90**

- Access *arr*[3] as 0x90 + 3 assuming 1-byte values

arr[3]

What's Missing?

What are we missing?

- Nothing says “this is an array” in memory
- Nothing says how long the array is

Instructions

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

Instruction Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Conceptually, set of instructions that are possible and how they should be encoded
- Results in many *different* machines to implement same ISA
 - Example: How many machines implement our example ISA?
- Common in how we design hardware

Instruction Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Provides an abstraction layer between:
 - Everything computer is really doing (hardware)
 - What programmer using the computer needs to know (software)
- Hardware and Software engineers have freedom of design, if conforming to ISA
- Can change the machine without breaking any programs

CSO: covering many of the times we'll need to think across this barrier

Instruction Set Architecture

Backwards compatibility

- Include flexibility to add additional instructions later
- Original instructions will still work
- Same program can be run on PC from 10+ years ago and new PC today

Most manufacturers choose an ISA and stick with it

- Notable Exception: Apple

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

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)

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?

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

The Stack: Push and Pop

What about real ISAs?