

# Function Calls, Memory Instruction Set Architectures

---

## CS 2130: Computer Systems and Organization 1

**Xinyao Yi** Ph.D.  
Assistant Professor

## Announcements

---

- Homework 3 due Wednesday at 11:59pm on Gradescope
- Midterm 1 Friday (October 3, 2025) in class
  - Written, closed notes
  - If you have SDAC, please schedule ASAP

## Quiz Questions – Quiz 1

---

### Q5.3 XOR

1 Point

Suppose we then shift it back and xor it with the original, like

```
((0xCA >> 3) << 3) ^ 0xCA.
```

The result is:

- ☒ the same for both signed and unsigned integers
- ☐ larger for signed than unsigned integers
- ☐ larger for unsigned than signed integers
- ☐ there is no way to know

## Quiz Questions – Quiz 3

---

### Q8 Counter

1 Point

To build a 4-bit counter circuit, we could directly connect the output of the increment circuit back to the input.

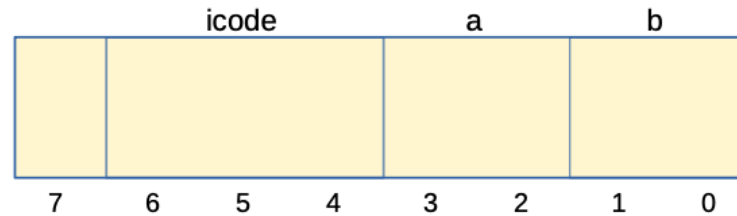
- ☐ True
- ☒ False

## Encoding Instructions

---

### Encoding of Instructions

- 3-bit icode (which operation to perform)
  - Numeric mapping from icode to operation
- Which registers to use (2 bits each)
- Reserved bit for future expansion



## 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

## Memory

---

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:  $x += y$



## 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

---

## 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

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

## Instructions 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

## Instructions 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



## Instructions 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)

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

## Instructions 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 Instructions 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 Instructions 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 Instructions 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  $(x \leq 0)$  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  $(x \leq 0)$  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
  - $\text{rsp} -= 1$
  - $\text{M}[\text{rsp}] = \text{r0}$

pop r2

- Read value from “top”, save to register
  - $\text{r2} = \text{M}[\text{rsp}]$
  - $\text{rsp} += 1$

## The Stack: Push and Pop

---

## The Stack: Push and Pop

---

**What about real ISAs?**