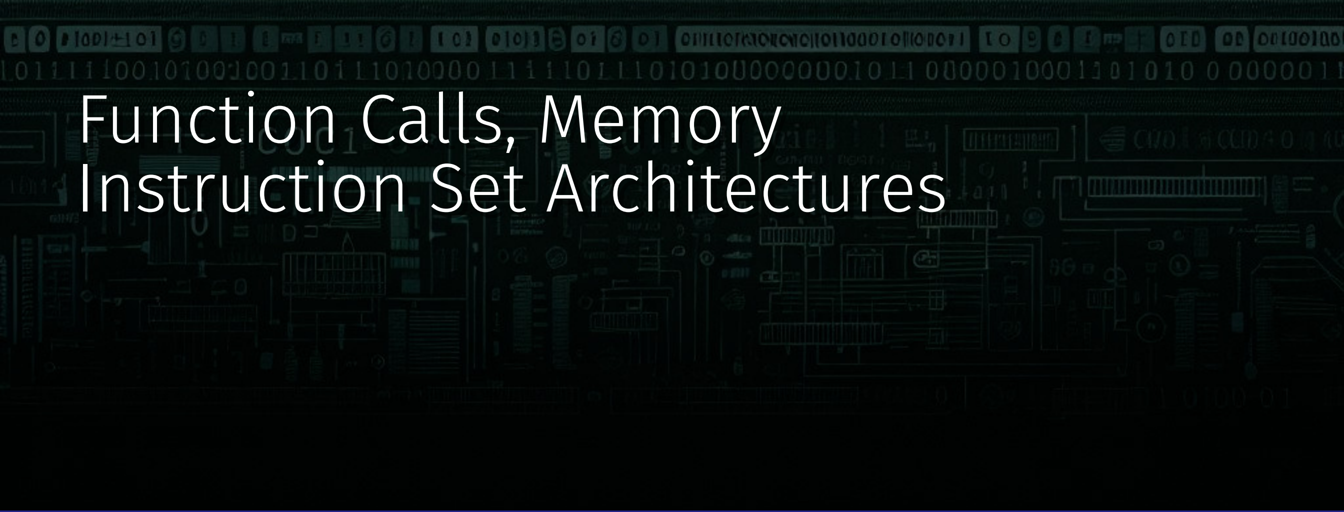


Function Calls, Memory
Instruction Set Architectures

CS 2130: Computer Systems and Organization 1
February 16, 2026

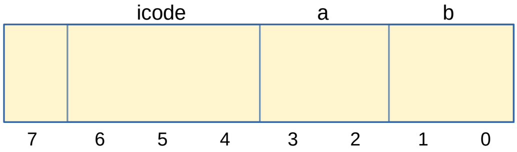
Announcements

- Homework 3 due next Monday on Gradescope
- Midterm 1 Friday in class
  - Written, closed notes
  - If you have SDAC, please schedule ASAP

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



Instructions

icode	b	meaning
0		`rA = rB`
1		`rA &= rB`
2		`rA += rB`
3	0	`rA = ~rA`
	1	`rA = !rA`
	2	`rA = -rA`
	3	`rA = pc`
4		`rA` = read from memory at address `rB`
5		write `rA` to memory at address `rB`
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 `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

while(i<5)

≡

next byte in program

R[A] = M[pc+1]

&=

+=

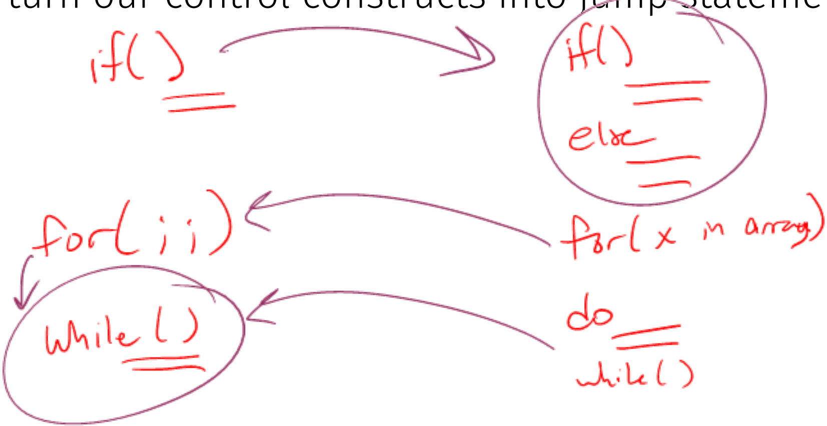
R[A] = M[M[pc+1]]

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`

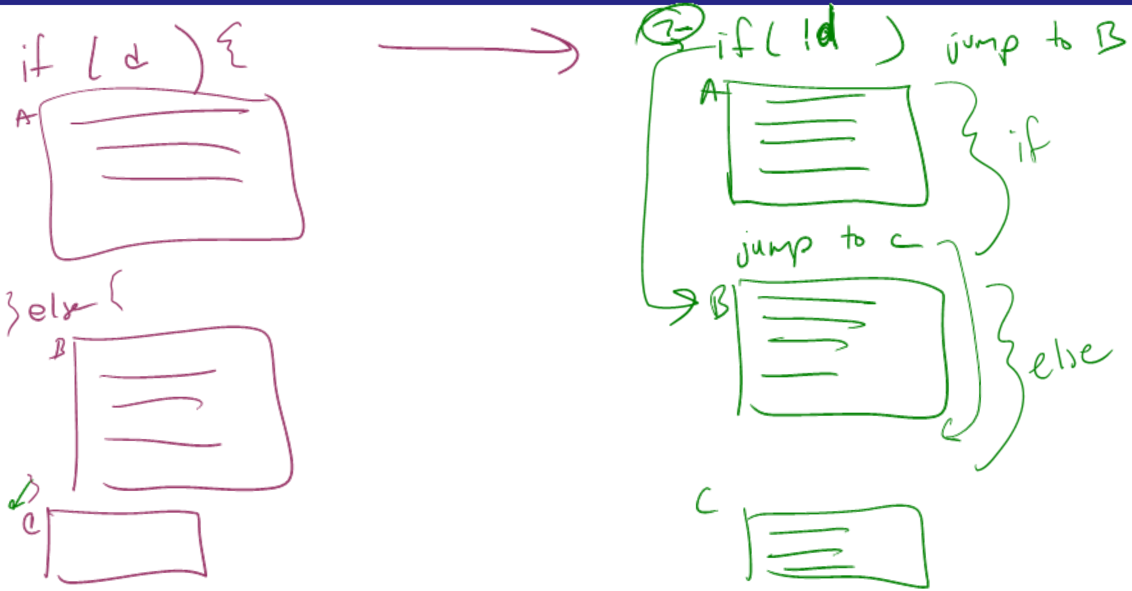
Our code to this machine code

How do we turn our control constructs into jump statements?

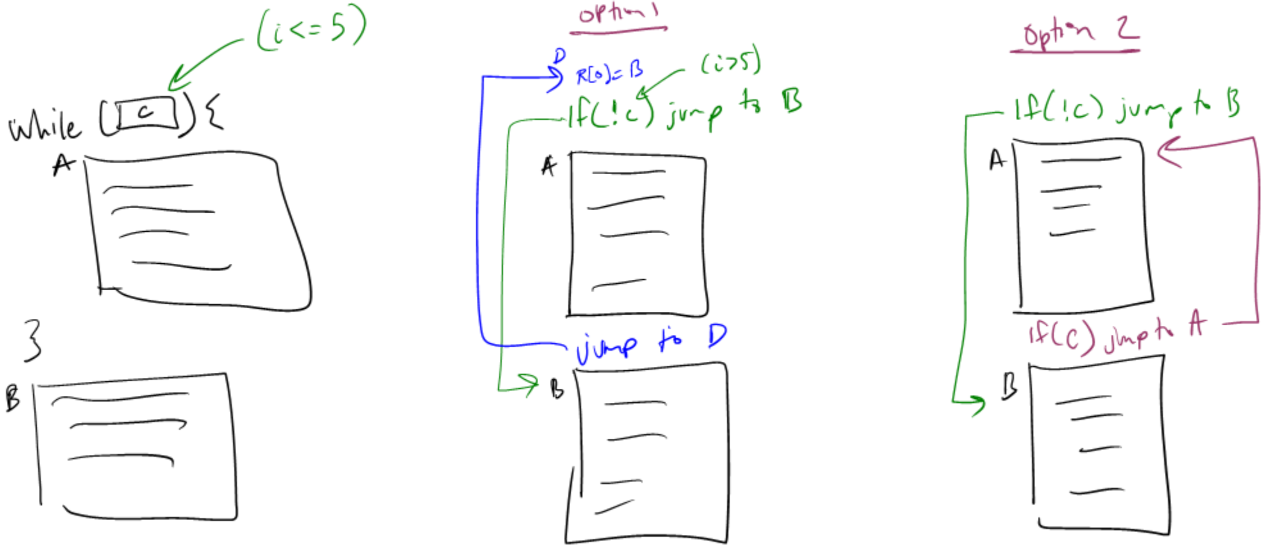


```
for(i=0; i<5; i++){

}
    ↓
i=0;
while( i<5){

    i++;
}
```

if/else to jump



while to jump



Encoding Instructions

Example 3: `if r0 < 9 jump to 0x42`

while (x ≥ 9) { → if (x < 9) jump ~~after loop~~ skip loop

}

Instructions

F8 — - — FD FE FF 00
-8 -3 -2 -1

icode	b	meaning
0		`rA = rB`
1		`rA &= rB`
2		`rA += rB`
3	0	`rA = ~rA`
	1	`rA = !rA`
	2	`rA = -rA`
	3	`rA = pc`
4		`rA` = read from memory at address `rB`
5		write `rA` to memory at address `rB`
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 `0`
		if `rA <= 0` set `pc = rB`
		else increment `pc` as normal

if $(r[0] < 9)$ jump to 0x42

$r[0] < 9$

$r[0] \leq 8$

$\boxed{r[0] - 8 \leq 0}$

$R[1] = 0x42$ $\overset{6}{0110}\ \underset{A}{01}\ \underset{B}{00}$ 42

$R[0] += -8$ $\overset{6}{0110}\ \underset{A}{00}\ \underset{B}{10}$ F8

if $(R[\underset{A}{0}] \leq 0)$ jump to $R[\underset{B}{1}]$

$\overset{7}{0111}\ \underset{A}{00}\ \underset{B}{01}$

64 42 62 F8 71

Function Calls

Memory

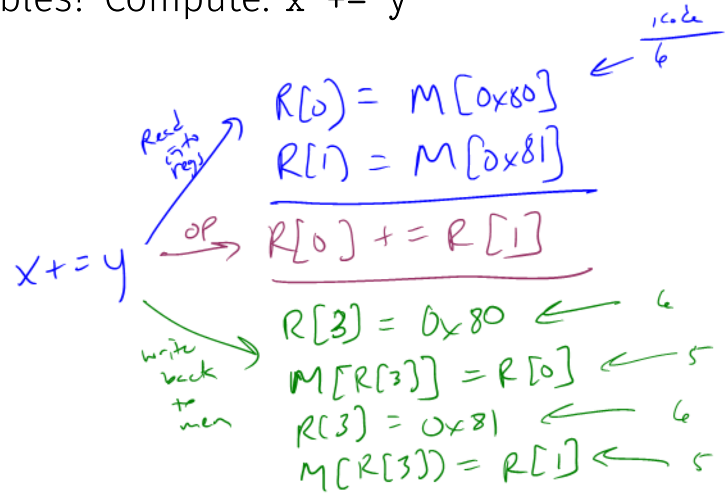
What kinds of things do we put in memory?

- 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`

	M. Addr
x	0x80
y	0x81
z	0x82
w	x83
t	0x84
foo	0x85
	⋮



R[0] = M[0x80] ← 6

R[1] = M[0x81]

R[0] += R[1]

R[3] = 0x80 ← 6

M[R[3]] = R[0] ← 5

R[3] = 0x81 ← 6

M[R[3]] = R[1] ← 5

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 = ~rA`
	1	`rA = !rA`
	2	`rA = -rA`
	3	`rA = pc`
4		`rA` = read from memory at address `rB`
5		write `rA` to memory at address `rB`
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 `0`
		if `rA <= 0` 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

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)

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?