Midterm 2 Review

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Instructions Set Architecture

We've designed an 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

Intel and AMD's CPU both using x86_64 ISA. So they can run the same program.

Instructions Set Architecture

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

Lots of flexibility and freedom to build things that would be faster, like hyperthreading. I don't worry about on the software side.

Just make sure the code can be compiled to ISA. I can run it on hardware.

- 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

It's easy to handy

Backwards compatibility

0
reserved

- Include flexibility to add additional instructions later — I can add more things.
- Original instructions will still work — My new machine still can run the old code. Old instructions still there.
- 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

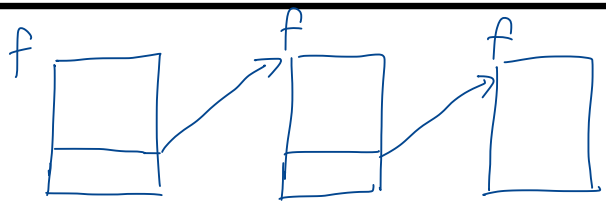
My new machine is much faster.

There may be some issues, but that is another story

One manufacturer that has enough following to be able to make these changes and still come out ahead.

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
```

sums up the numbers between one and x.

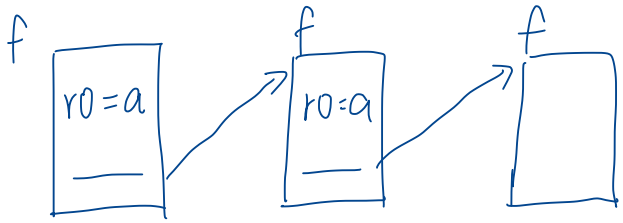
Recursion

- The formal study of a function that calls itself (while it computing the solution or the result for f, it actually calls itself in a smaller set),

Storing Variables in Memory

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

Where do we store a?



Save it in a register R0 ? No !

override this value over and over

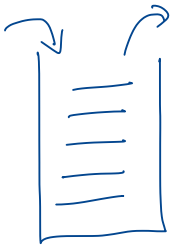
save it to memory ? No!

We need something that will help us to organize memory so that we keep track of these variables.

The Stack

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

- The solution for solving this problem



stack of plates

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

have the address.
(the index in memory of a certain point).

The Stack: Push and Pop

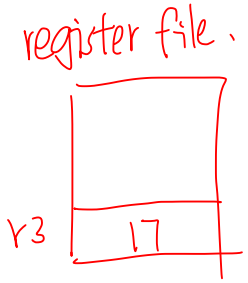
push r0

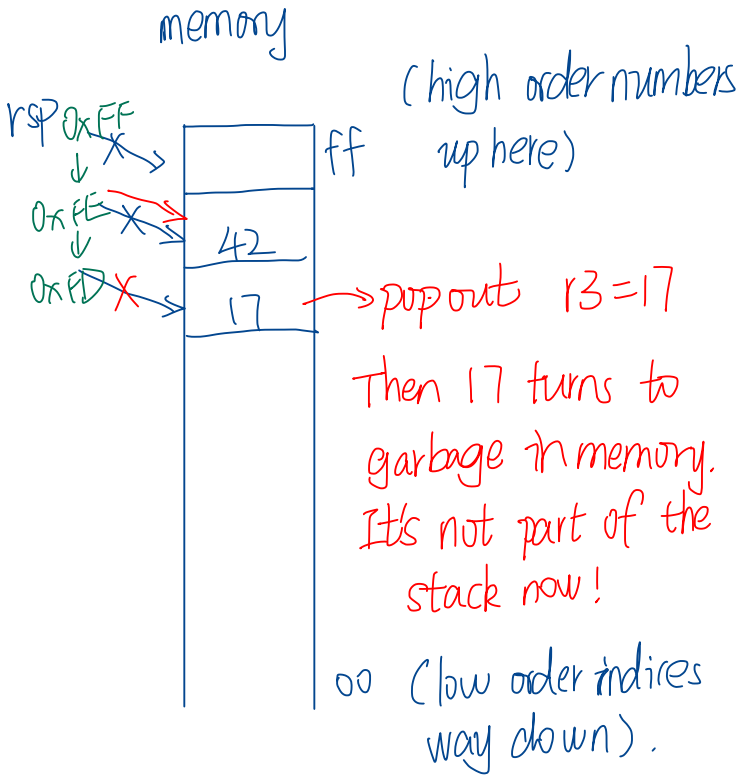
- Put a value onto the "top" of the stack
  - rsp -= 1
  - M[rsp] = r0

push 42
push 17
pop r3

pop r2

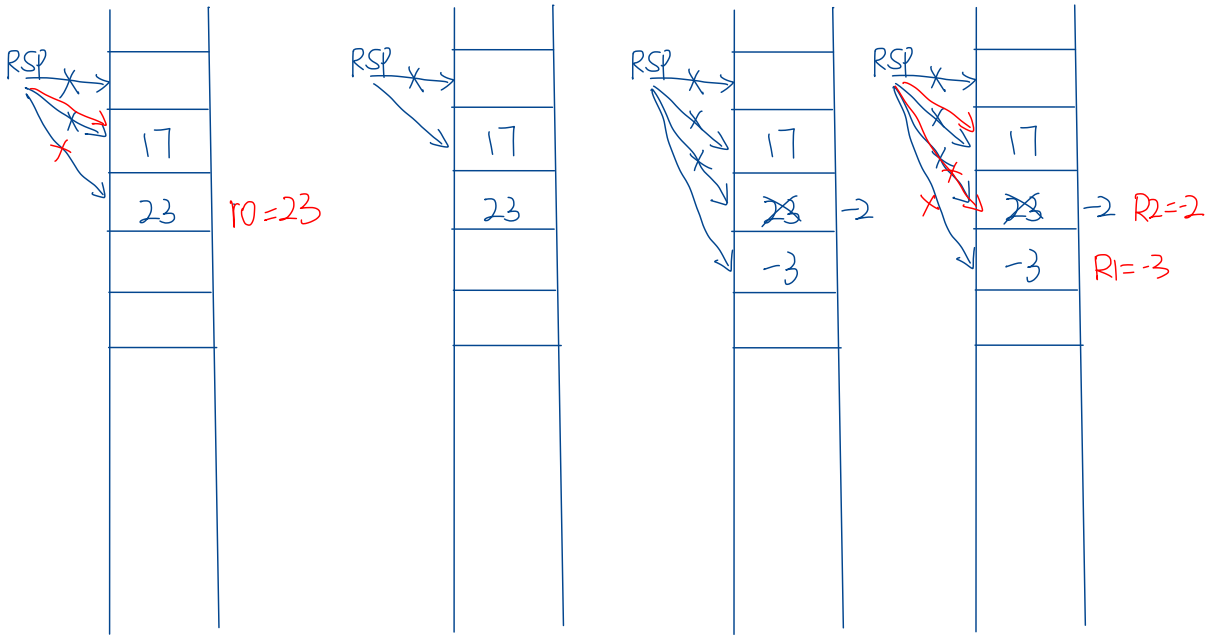
- Read value from "top", save to register
  - r2 = M[rsp]
  - rsp += 1



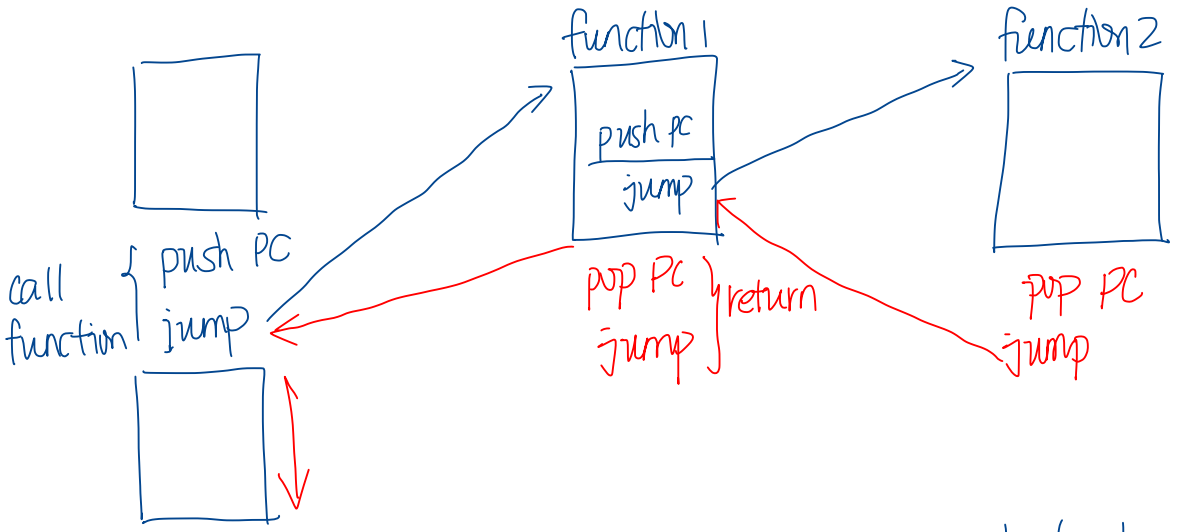


The Stack: Push and Pop

push(17)
push(23)
x = pop (R0)
push(-2)
push(-3)
y = pop (R1)
z = pop (R2)



Function Calls



What about parameters ?
return value?

calling conventions
- r2, r3 have operands
- r0 has return value.

Backdoors

Backdoor: secret way in to do new unexpected things

→ allow an attacker/developer to take complete control of your system — often without you knowing it.

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

operating system has security checks: passwords, permissions, firewalls ⇒ all meant to control "who can do what"

Someone installs a backdoor ⇒ sneak past those protections and gain control without going through the normal process.

Backdoors

maybe a program ? scripts ?

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

backdoor : hidden entrance

exploit : how you find it, open it and use it to get inside.

Our Hardware Backdoor

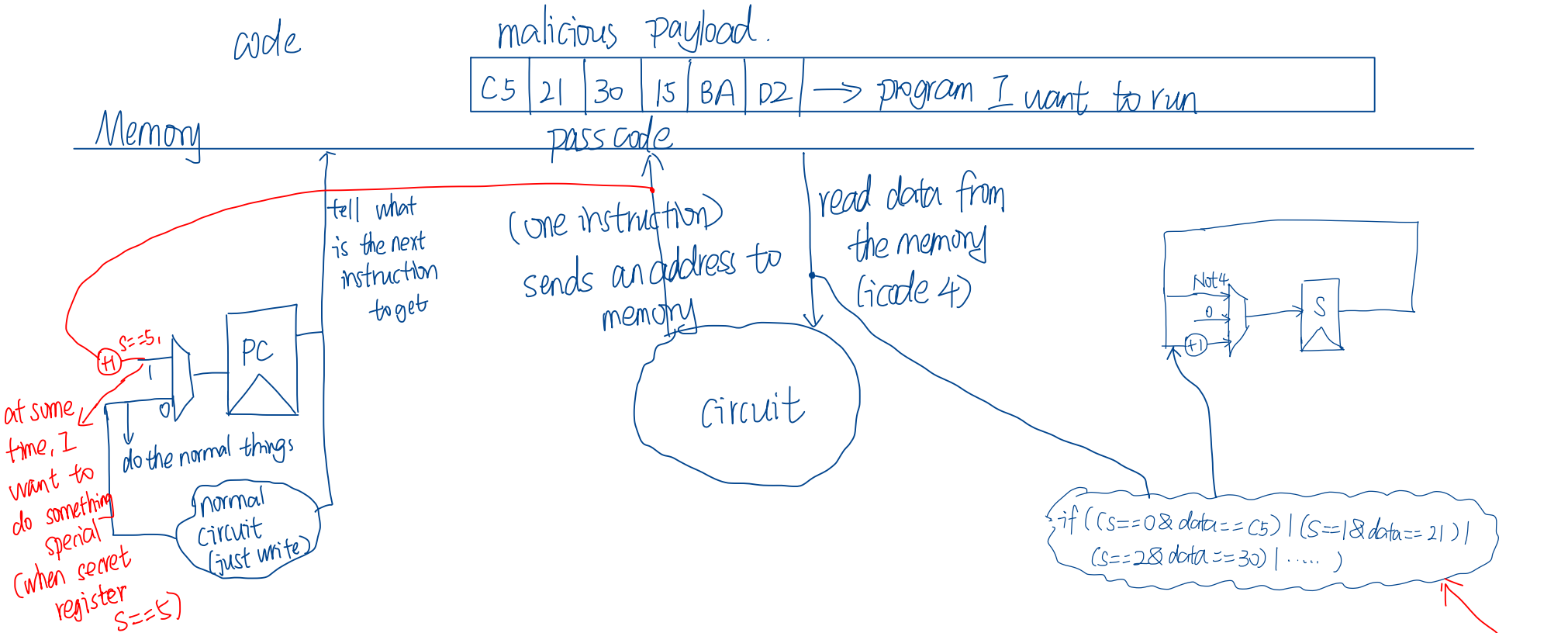
Our backdoor will have 2 components

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

The hardware needs to recognize it when it appears.
(Like secret knock on a door. if you hear that pattern, you know someone special wants in.

Once the hardware recognizes the passcode, it executes some hidden or harmful behavior.

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!

If I had a microscope and I knew exactly where to look, I probably can find it.
Or, maybe, I'm very, very lucky.

But - Most exploits are going to be found after somebody tries to use them.

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

Ordering Values

0x|00|AB|CD|EF

Little-endian

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

EF CD AB 00
0x600 0x601 0x602 0x603

Big-endian

- Store the high order part/byte first

00 AB CD EF
0x600 0x601 0x602 0x603

Why we want to talk about 2 ways?
Because people decided to do different things.
We write 00ABCDEF, but we calculate from F to 0,

Maybe that's the reason to see EF first?

Example

array of 2 numbers, each number should use 2 bytes.

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

	address	little endian	big endian
0x1234	0xF00	34	12
	0xF01	12	34
0x5678	0xF02	78	56
	0xF03	56	78

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

People didn't use the same thing.

In fact, your computers are probably doing different things now.

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!

- ISA is like the grammar and vocabulary of a language.
- Assembly code is a sentence written in that language.

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!

(.text .data .byte)

(extra information)

It's going to replace them with the actual addresses when it builds the binary that we're going to run.

Assembly Languages

There are many assembly languages

Each CPU family has its own unique set of machine instructions – therefore it's own assembly language.

- But, they're backed by hardware!
- Two big ones these days: x86-64 and ARM
  - most computers
  - M1/M2 chips on MAC and cellphones
  – You likely have machines that use one of these
- Others: RISC-V, MIPS, ... PowerPC

We will focus on **x86-64**

x86-64

x86-64 has a weird and long history

- Expansion of the 8086 series (Intel)

8 bits 16 bits 32 bits
8008 – 8086, 8286, 8386, 8486, x86

- AMD expanded it with AMD64 A 64-bit that was backward compatible with X86.
- 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`

(typically we will not see more than 2)

lea loading the addresses

In most of the cases, we are doing something using the value. Except for

AT&T x86-84 Assembly

`mylabelname:` end with a colon

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

`.something something` start with a dot

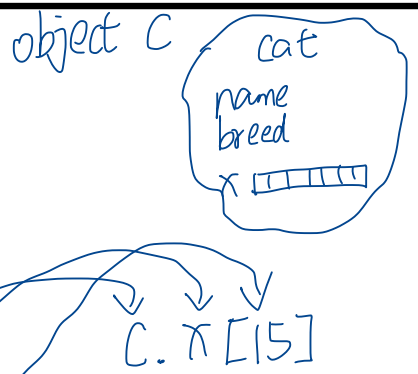
- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

`// we can have comments!`

Addressing Memory

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

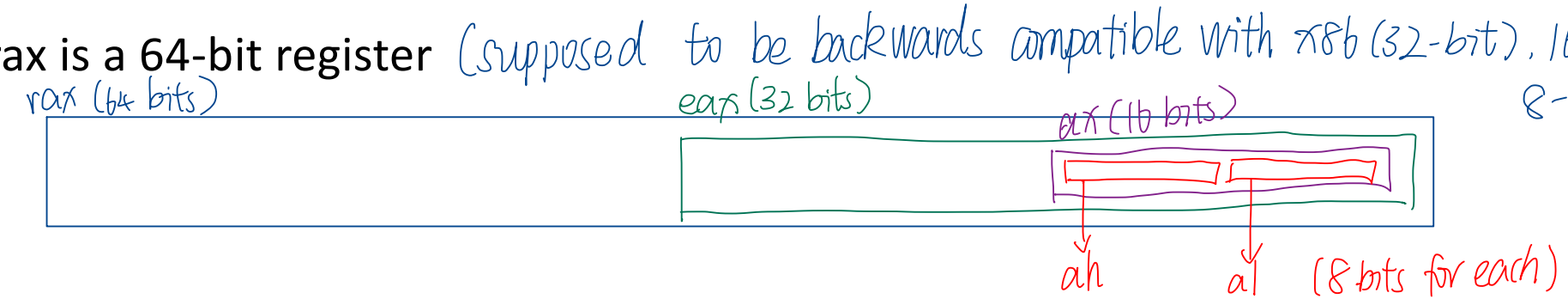
- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `2130 + %rax + (%rsp * 8)`
- Common usage from this example:
  - `rax` - address of an object in memory
  - 2130 - offset of an array into the object
  - `rsp` - index into the array
  - 8 - size of the values in the array (used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



If I don't have all the pieces, it will calculate what it can.

Registers

rax is a 64-bit register (supposed to be backwards compatible with x86 (32-bit), 16-bit, 8-bit)



If I look at 32-bit version, it will just zero out the top 32 bits.

We'll see this with all our registers, in slightly different way.

(check the reading)

Instructions (short acronyms for what we want to do, like mov, add, and, or, xor, neg)

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move (similar for addq, subq)
  - `q` = quad word
- `movl` - 32-bit move
  - `l` = long

The instruction followed by how wide of the thing we want to do.

- 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

We cannot do memory to memory calculations.

Other Instructions

Other instructions work the same way

- `addq %rax, %rcx` — rcx += rax
  - src, dest
- `subq (%rbx), %rax` — rax -= M[rbx]
  - going to memory and get the value
- xor, and, and others work the same way!
- Assembly has virtually no 3-argument instructions
  - All will be modifying something (i.e., +=, &=, ...)
    - modify one of the inputs directly, doesn't have a seperate output.

Load Effective Address

I'm not going to the memory, "lea" is a special instruction that calculates the memory address and store the memory address itself in a register.

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

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

%rax = %rcx + 4.

Jumps

`jmp foo`

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

Conditional jumps

- `jl`, `jle`, `je`, `jne`, `jg`, `jge`, `ja`, `jb`, `js`, `jo`
  - `jl` `<`, `jle` `<=`, `je` `==`, `jne` `!=`, `jg` `>`, `jge` `>=`
  - `ja` above, `jb` below (unsigned)
  - `js` If the assigned bit is set.
  - `jo` If there's a overflow

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

Jumps

We jump based on the result of some special registers called condition codes.

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
```

They don't have to be back to back.

jump will be based on the most recent thing that set the condition code.

→ You can do something like move things around.

  - 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!

test could be used to check if a register has 0 in it.

```
testq %rax, %rax
je zero_case    // if rax==0
jne nonzero_case // if rax!=0
```

Example: Loops

```
while (i < 10)
    i += 1
```

```
top: ← lable
    // check !condition, jump out
    if(i>=10) goto end
        i+=1;
    // jump back to condition
    go to top;
end : ← lable
```

```
main: ——————→ lable
    movq $0, %rax    // we set rax=0 for int i=0;
loop: cmp $10, %rax  // rax-10 = ?  { if rax<10, we got negtive, then do loop body.
                                     { if rax >=10, we got positive or 0, then jump out the loop.
    jge after
    addq $1, %rax
    jmp loop
after: retq  // return with a "q" because we're working with a 64 bit thing. (pop a 8-byte address and jump back to caller)
```

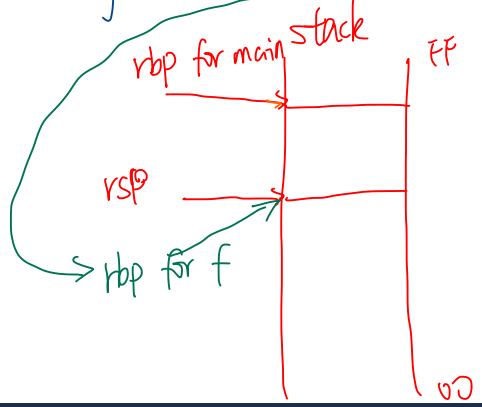
Example: Functions

```
f(x,y):
    ...
    ...
    return 4

...
z = f(2,5)
```

```
int f(int x, int y){
    return 4;
}

int main(){
    int z = f(2,5);
}
```




```
.global f
f:
    pushq %rbp          //store old base pointer
    movq %rsp, %rbp     //create a new stack for f
    movl $4, %eax       //put return value 4 to %eax
    popq %rbp           //pop old base pointer (for main)
    retq                //return.
.global main
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $2, %edi       //put parameter 2 to %edi
    movl $5, %esi       //put 5 to %esi
    callq f
    movl %eax, -4(%rbp) //store return value to local variable z.
```

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)

The function I'm running currently and the function that I call are both sharing the same registers.

`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

More conventions, check readings.

Calling Conventions: Registers

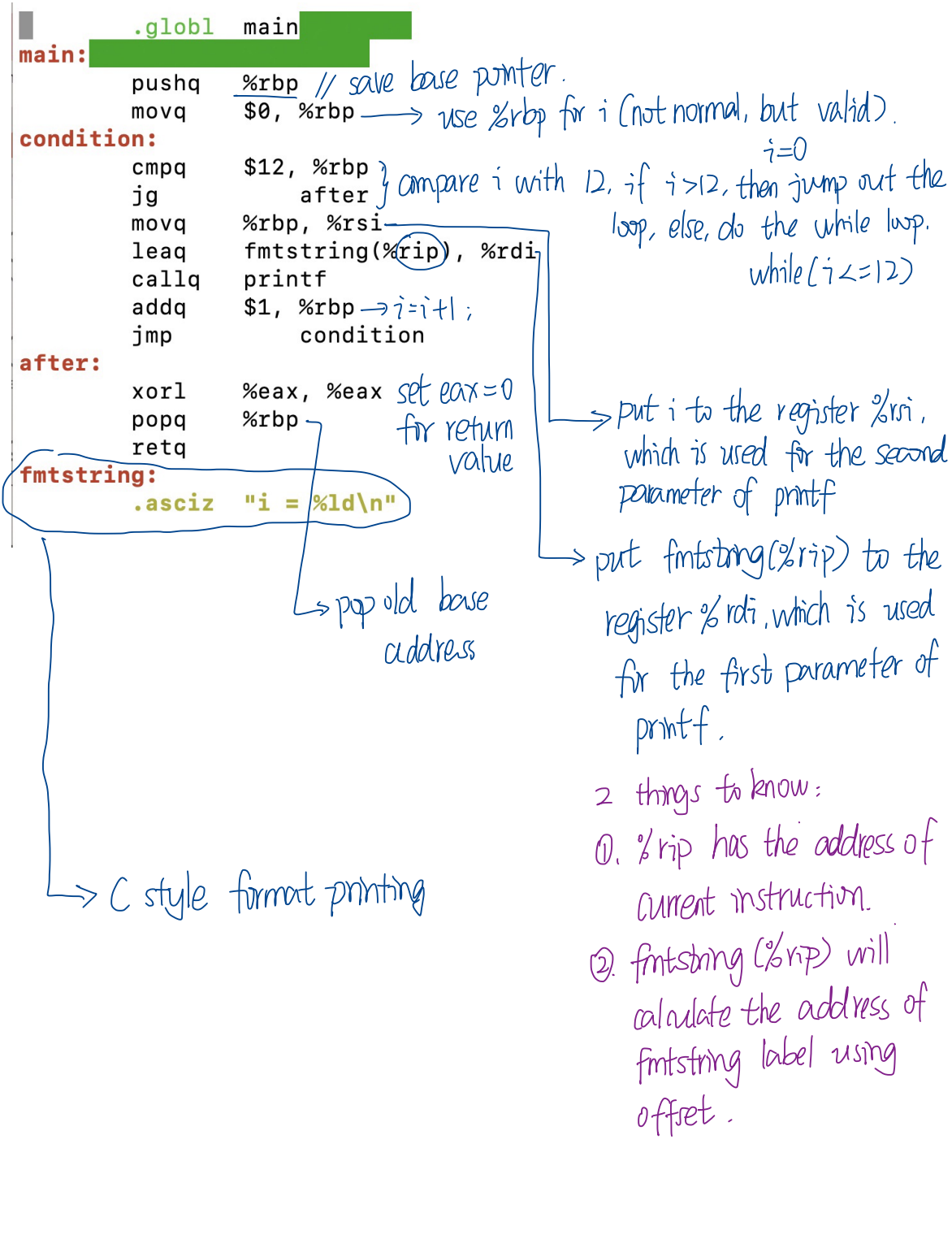
Why? Caller and callee share the same registers.

Calling conventions - recommendations for making function calls

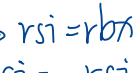
- Where to put arguments/parameters for the function call?
- Where to put return value? in `rax` before calling `retq`
- What happens to values in the registers?
  - **Callee-save** - The function should ensure the values in these registers are unchanged when the function returns (① push the old values before calling ② pop the values before returning.)
    - `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`
  - **Caller-save** - Before making a function call, save the value, since the function may change it

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


.globl main
main:
pushq %rbp // save base pointer.
movq $0, %rbp → use %rbp for i (not normal, but valid).
i=0
condition:
cmpq $12, %rbp } compare i with 12, if i>12, then jump out the
jg after loop, else, do the while loop.
while(i<=12)
movq %rbp, %rsi
leaq fmtstring(%rip), %rdi
callq printf
addq $1, %rbp → i=i+1;
jmp condition
after:
xorl %eax, %eax set eax=0
popq %rbp for return
retq value
fmtstring:
.asciz "i = %ld\n"
put i to the register %rsi, which is used for the second parameter of printf
put fmtstring(%rip) to the register %rdi, which is used for the first parameter of printf.
pop old base address
C style format printing
2 things to know:
① %rip has the address of current instruction.
② fmtstring(%rip) will calculate the address of fmtstring label using offset.

```
        .globl  main
main:
        pushq   %rbp
    movq    $0x42, %rax
    movq    $0x15, %rbx
    movq    %rbx, %rsi
    negq    %rsi
    addq    %rax, %rsi
        leaq    fmtstring(%rip), %rdi
        callq   printf
    xorq    %rax, %rax
        popq    %rbp
        retq
fmtstring:
        .asciz  "%X\n"
```



Debugger

objdump -D substract | less → show by pages.
(tool name → disassemble all sections.)

// disassemble all sections from binary to assembly

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!**

lldb substrat // run the executable file "substract" in debugging mode.

b main // add a breakpoint at the beginning of main function

r / run // run the code

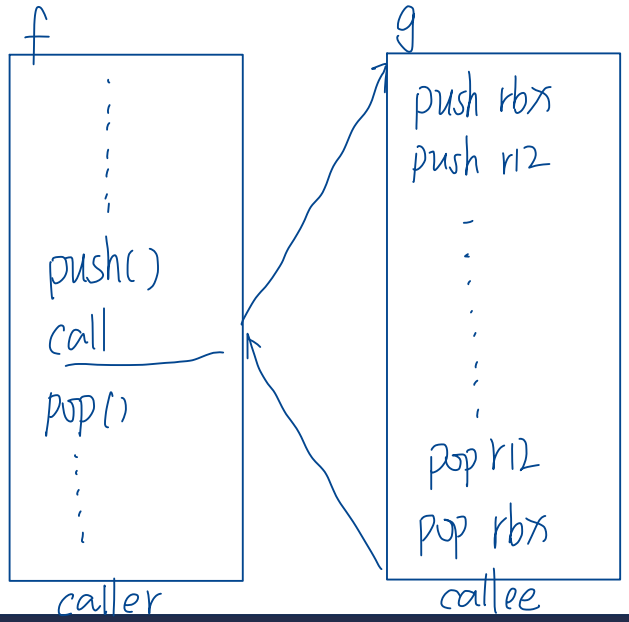
n // run the next instruction

disassemble // disassemble the binary code to assembly code (work for the current frame-main).

re read // read the registers.

The Stack

pushq %rax
popq %rdx




```
.globl  main
main:
    pushq   %rbp

    # Set some example values in registers
    movq    $0x42, %rax
    movq    $0x15, %rbx
    movl    $4, %esi
    # Push 64-bit rax, then 16-bit si
    pushq   %rax
    pushw   %si

    # Pop -- oops!
    popq    %rdi
    popw    %si

    # Return 0 (all is well)
    xorq    %rax, %rax
    popq    %rbp
    retq
```

32bits 0x00000004

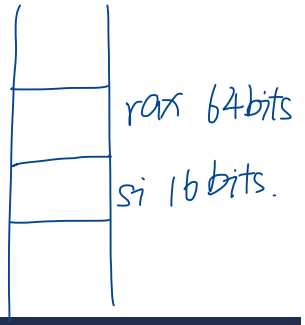
64 bits

16 bits (only use the lower 16 bits of %esi)

64 bits

16 bits

since the first pop moved the stack pointer by 8, so the data got is totally wrong.



Key Lesson: Mixing operand sizes on the stack breaks alignment and leads to corrupted data.

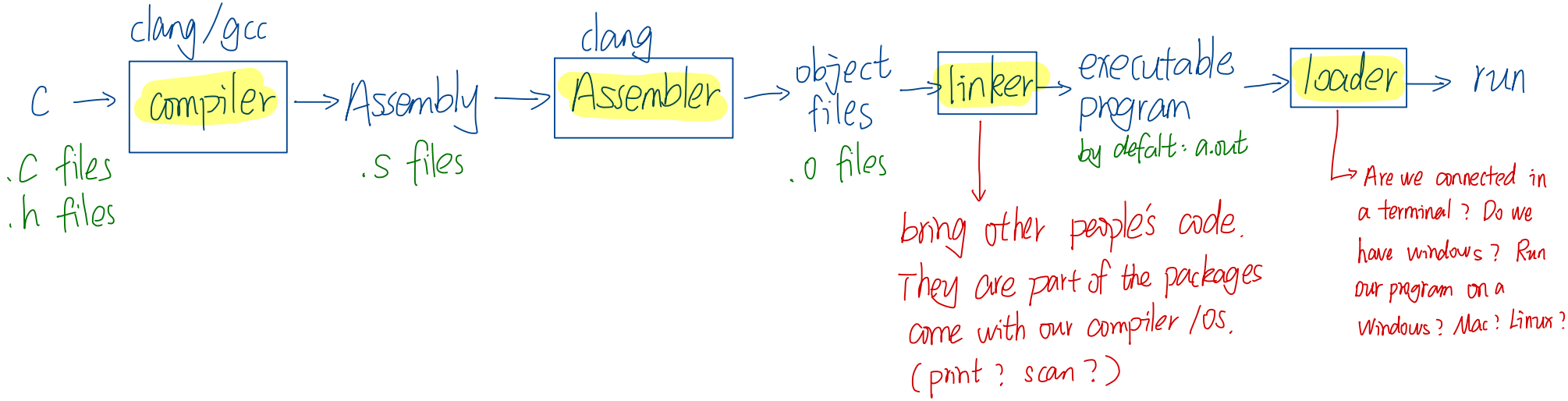
Category	**Patent**	**Copyright**	**License Agreement**
Protected Object	Technological inventions and innovations	Creative expressions and forms (e.g., code, text, images)	Usage rights of protected content
Source of Right	Granted by government agencies after examination	Automatically arises upon creation	Established through contractual agreement
Automatic Acquisition	❌ No	✅ Yes	❌ No (requires signing)
Duration	Typically 20 years	Author's lifetime + a number of years (e.g., 70 years in the U.S.)	Determined by contract terms
Nature of Right	Exclusive right to make, use, or sell an invention	Prevents copying or unauthorized reproduction	Defines how others can use or distribute the work
Examples	A new GPU scheduling algorithm, hardware design	Source code, research paper, textbook	MIT License, GPL, Apache 2.0, commercial EULA
Key Focus	Innovation and technical exclusivity	Expression and originality	Permission and terms of use

Compilation Pipeline

We want to bridge the gap between the assembly and C.

Turning our code into something that runs

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



Notes:

① The whole steps from assembly to executable, is fully reversable. This might not be perfect, the labels may not be quite what we want.

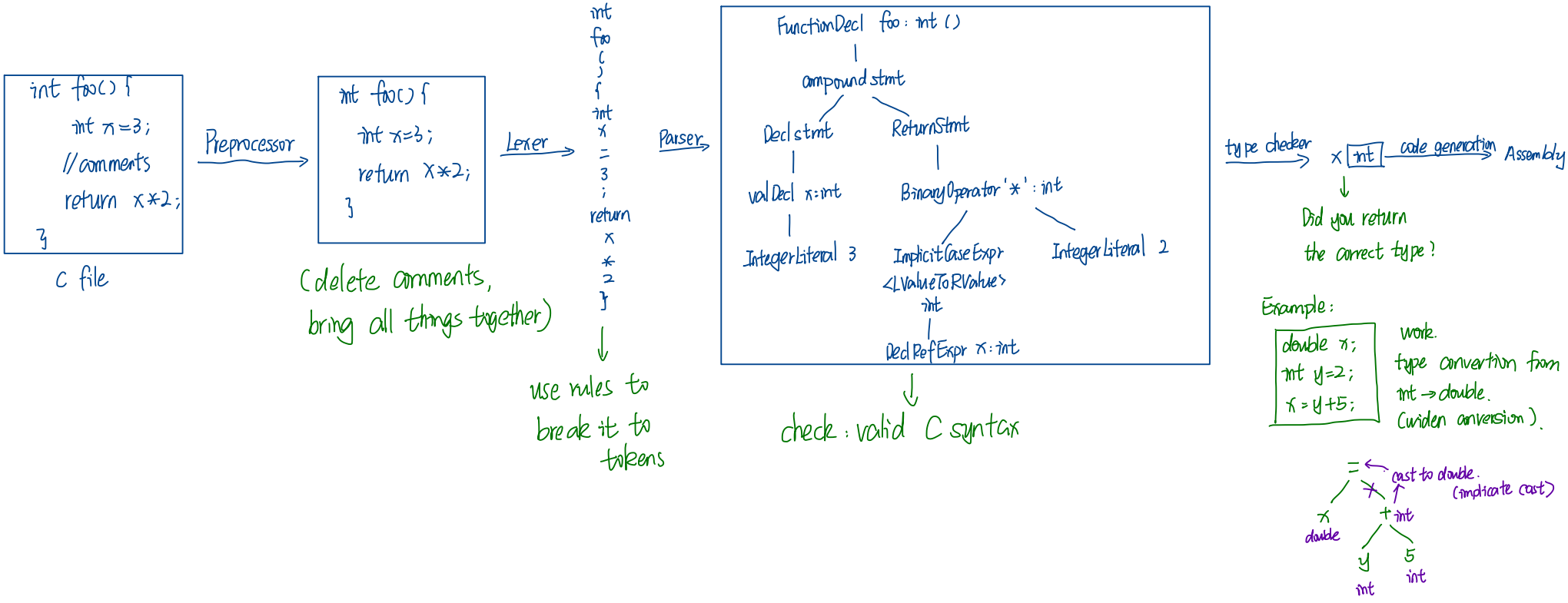
② The process from C to assembly is lossy. Difficult to go back to C.

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

Compiling C to Assembly



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 (Example: Segmentation fault).

Simple C Example

(we demonstrated an example in portal. check recording).

```
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

why "xorl %eax, %ecx" ?

The compiler is choosing to use this as an easy way – a short way – to zero out the return register. It's only two bytes to clear that register with zero.

Data Types in C

Integer data types

Data type	Size (bits)	Size (bytes).
char	8	1
short	16	2
int	32	4
long	64	8
long long	64	8

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

by default, short, int, long, long long are signed.

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

The size of a pointer? How many bytes? — 64 bits (8 bytes) (Registers are 64 bits because we had 64-bit memory addresses)

If I want to have a pointer to an int: `int *y;`

→ the star references the fact that this is not an integer, it's a pointer.

`int x=5;` (4 bytes)

`int *y;` (8 bytes)

`y=x;` → doesn't work

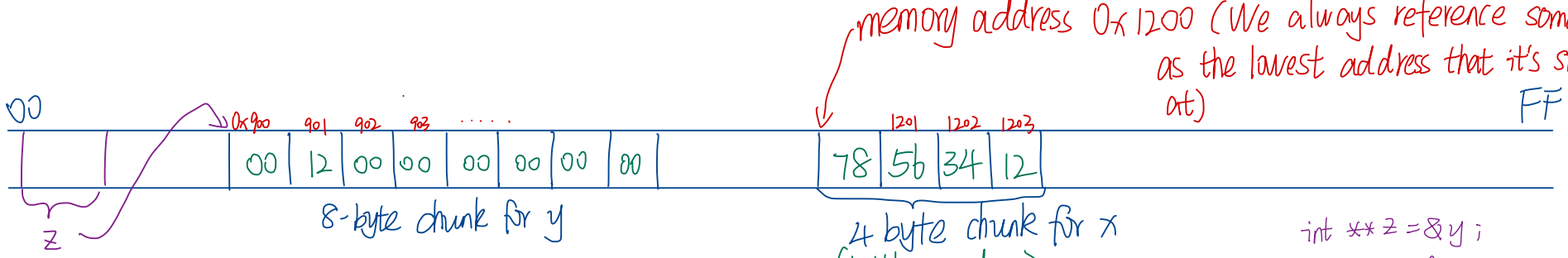
`y=&x;` It works!

`*y=25`, will change x to 25.

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



int x = 0x12345678

int *y;

y = &x, (I want y point to x, which means store the address of x in y).

int **z = &y;

*z : dereference z

**z: dereference *z

**z = 0x00000000;
(set x to 0x00000000)

*z = 25; (Warning: you're trying to assign an integer to a pointer)
(But later, when you use it as an address, may seg Fault).

y = 0xABCDEF01; (I'm changing the pointer) → I may don't want to do this. Sometimes seg fault?

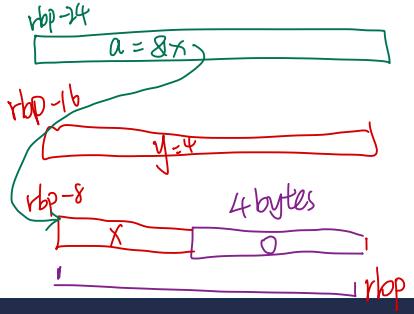
*y = 25; (When I use an asterisk, it will say is follow the pointer on y, follow the address to the place in memory that y points to and change that value)

Example

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

Example

```
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
```

(compiler knows we are working on a 64-bit machine. I'll probably try to line everything to 8 bytes if I can. to make things faster for you)

loading the addr of x into rcx

y is treated as the same way.

&x

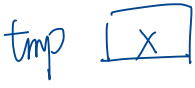
signed extend I to q, long to quad

b going from long to int.

move the value of y into rcx, but I will just read out the value of ecx.

Swap Example:

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```



Arrays

(We pick a spot in memory, and the next so many spots are our array)

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

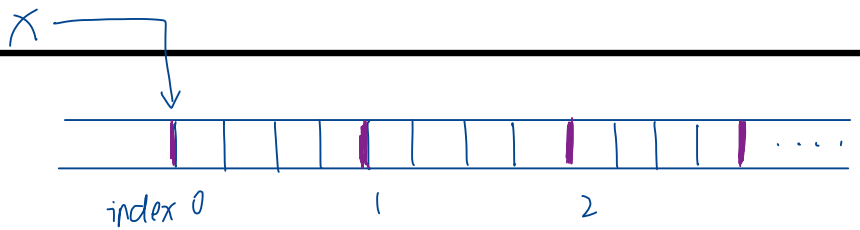
- Declare as you would use: `int myarr[100];` (100 integers in my array)
- `sizeof(myarr)` = 400 — 100 4-byte integers (How big is my array in bytes?)
- `myarr` treated as pointer to first element
- Can declare array literals:

```
int y[5] = {1, 1, 2, 3, 5}
```

(optional)

When I create an array, it actually create a pointer to the first thing in that list.

Pointers and Arrays



(dereference x will be the first element of array x)

`*x` and `x[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: `x[n]` and `*(x+n)` → pointer arithmetic
  - If `x` is an `int *`, then `x+1` points to **next int** in memory
  - Adding 1 to pointer adds `sizeof()` the type we're pointing to

Not skip 5 bytes in memory

I know this pointer is a pointer to an integer, so I plus 5 * sizeof(int)

Pointers and Arrays

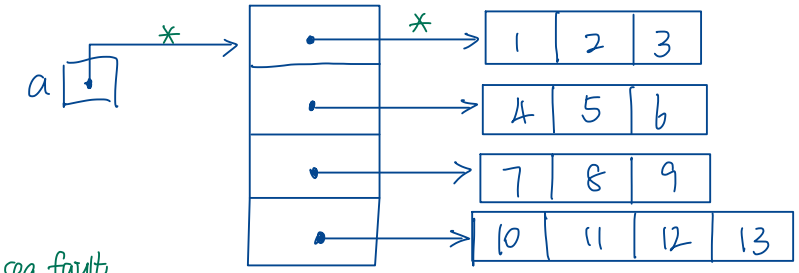
one pointer or an array of pointer?

How do I know? sizeof( )

one integer or an array of integer?

Consider: **int **a** (a pointer to a pointer to an integer)

I suppose "a" point to the first thing of an array. This array should be an array of pointers
And I suppose the pointers in this pointer array point to an array of integers.



They are just pointers to a place in memory.
It's doesn't matter how big the arrays are

```
**a ⇒ 1
a[0][0] ⇒ 1
*(a[2]) ⇒ 7
(*a)[2] ⇒ 3
a[1][3] ⇒ Something weird here. In Java, just seg fault.
a[1][0] ⇒ 4
a[0][1] ⇒ 1
*a[1] ⇒ *(a[1]) ⇒ 4
```

But in C, I can just give you the value at that place.
① Maybe seg fault if you read too far
② Maybe some number there?

I will know something wrong when I use this value later. And if I'm writing sth, I may overwrite sth I don't know.

Pointers

sizeof(char *) == sizeof(long double *)

- All pointers are the same size: address size in underlying ISA
- Two special int 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`

sizeof() returns a value of type size_t

→ These represent integers with the same size as a pointer.

Pointers

Consider the following code:

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

We suppose x lives at address 1000

y point to x, so y == 1000

pointer arithmetic, y points to an integer, so y+2 means go forward 2 ints in memory. which is 4 bytes

since y was 1000. so z will be 1000+8 = 1008

cast the address to long. So we treat them like plain integers. 1008-1000 = 8

Why is w = 8?

Other Types and Values

if I write numbers, they are implicitly cast to integers, which is 32 bits.

If I want to write something longer, I need to add some suffixes.

- Literal values - integer literals are implicitly cast
  - `unsigned long very_big = 9223372036854775808uL`
  - `u` for unsigned, `L` for long, `LL` for long long (capitalization doesn't matter, but you'll usually see "u" lowercase and "L" uppercase.
- `enum` - named integer constants (in ascending order)
  - `enum { a, b, c, d=100, e };` (when I dealing with choices the user has to make, the easiest way to do is using integers, for easier reading. I give them some names)
    0 1 2 101
    I can change where I'm at using "="
    `int foo = e;` It's a type!
- `void` - a byte with no meaning or "nothing" → I can't do equals or any operations on it.
  - Pointers: `void *p` (I don't care what's there)
  - Return values: `void myfunction();` (I don't care what is in register rax).

```
float b = 123.4;
void *p = &b;
float c = *(float*)p;
```

- 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)

casting pointers does some weird things. we'll talk about that later.

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;
```

the order of this list is important. It tells the compiler how to build this thing.

4+2+1=7

may be one byte of padding?
I don't know.

↳ declare a variable x of type struct foo.

I can set directly the fields.

When you create a struct in C, the compiler decides where to place each field in memory. It tries to align each field on an address that's a multiple of it's natural size, to make the CPU access faster.

A struct in a struct?

Yes, just set a struct as one field of another struct.

Structure Literals

We can use struct literals just like we saw with the array 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 };
```

"a" is a part of the type, not a part of the name.

→ must give values to fields in order.

→ you can use equal notation to give them values in other orders.

`struct a z;` ⟹ If I don't give any values, just allocate the memory and whatever is in memory is whats going to be there. I don't know what they are.

typedef

`typedef` - give new names to any type!

typedef unsigned long size_t;
typedef unsigned long address_t;

- 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;
```

Integer becomes another name for int.

dpp ptr; (Using typedef can make complex pointer types look simpler)

- Used with *anonymous structs*:

```
typedef struct { int x; double y; } foo;
foo z = { 42, 17.4 };
```

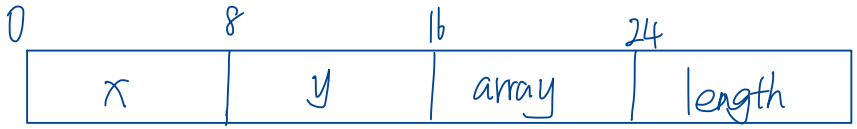
struct { ... } defines an anonymous struct (no struct name given).
The typedef immediately gives it the alias foo.
You can now create variables like foo z; instead of writing struct something z;.

Struct Example

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

remember we need this ";" at the end!

Assuming I have no padding because all of these are 64 bits.



foo a;

a.array ⇒ "a+16" to get me to the array.

Struct Example

Sum2 gets a pointer, it must dereference to access fields.

```
long sum2(foo *arg) {
    long ans = arg->x;
    for(long i = 0; i < arg->length; i += 1)
        ans += arg->y * arg->array[i];
    return ans;
}
```

```
typedef struct {
    long x;
    long y;
    long *array;
    long length;
} foo;
```

```
sum2:
    movq    (%rdi), %rax        ans = arg->x
    movq    24(%rdi), %r8       r8 = arg->length
    testq   %r8, %r8            performs bitwise AND.
    jle     .LBB1_3             if negative, jump to .LBB1_3
    movq    8(%rdi), %rdx       rdx = arg->y
    movq    16(%rdi), %rsi      rsi = arg->array
    xorl    %edi, %edi          i=0
.LBB1_2:
    movq    (%rsi,%rdi,8), %rcx rcx = array[i]
    imulq   %rdx, %rcx          rcx = y*array[i]
    addq    %rcx, %rax          ans += .....
    incq    %rdi                i++
    cmpq    %rdi, %r8           compare i<length
    jne     .LBB1_2
.LBB1_3:
    retq
```

integer multiply (signed)

increment

the first parameter (arg) is passed in register %rdi

Struct Example

sum1 gets a copy of the whole struct (foo arg), it accesses fields directly on the stack.

```
long sum1(foo arg) {
    long ans = arg.x;
    for(long i = 0; i < arg.length; i += 1)
        ans += arg.y * arg.array[i];
    return ans;
}
```

```
sum1:
    movq    8(%rsp), %rax   ans = arg.x
    movq    32(%rsp), %r8   r8 = arg.length
    testq   %r8, %r8
    jle     LBB0_3
    movq    16(%rsp), %rdx  rdx = arg.y
    movq    24(%rsp), %rsi  rsi = arg.array
    xorl    %edi, %edi      i = 0
.LBB0_2:
    movq    (%rsi,%rdi,8), %rcx
    imulq   %rdx, %rcx
    addq    %rcx, %rax
    incq    %rdi
    cmpq    %rdi, %r8
    jne     .LBB0_2
.LBB0_3:
    retq
```

Why does the compiler access the struct fields using (%rsp)+offset, instead of something like (%rdx-something)?
Modern compilers often omit it to save a register.
When you don't see %rbp, the compiler just use %rsp+offset instead.