Review

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

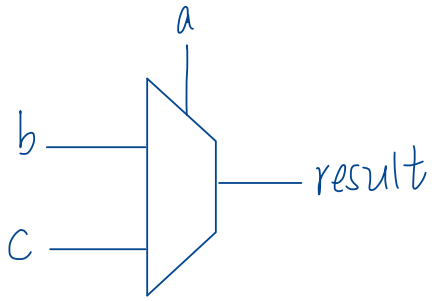
- Final exam: 7-10pm April 30, Gilmer 301 (different room!)
  - Cumulative, see practice tests
  - Exam conflict form in email
- Remember to fill out course evaluations

Multiplexer (mux)

ternary operator

x=a ? b : c

A multiplexer (mux) is commonly drawn as a trapezoid in circuit diagrams.




```
if(a){
    b
}else{
    c
}
```

a	b	c	result
0			c (depends on c)
1			b (depends on b)

a, b, c could be anything, they could be strings, numbers, functions.....

a.b.c are all one bit.

a	b	c	result
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

when the result is true ?

(!a& !b& c) | (!a& b&c) |

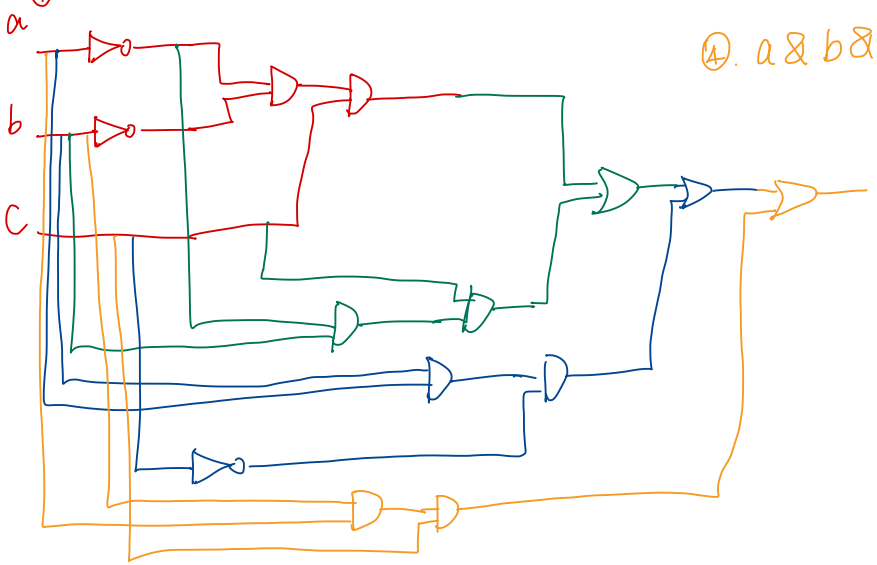
(a& b& !c) | (a& b &c)

① !a& !b& c.

② .!a& b& c.

③. a& b& !c.

④. a& b& c



11. [15 points] In the space below, draw a 2-bit decrement circuit: that is, a set of logic gates with 2 input wires (x_0 and x_1) and 2 output wires (z_0 and z_1) such that the output is numerically 1 less than the input ($z = x - 1$). Construct the circuit using only wires and gates from the set {and, or, not, xor}. Clearly label your input and output wires. *Hint: writing out a truth table can help design and/or check your circuit.*

Bases

We will discuss a few in this class

- Base-10 (decimal) - talking to humans
- Base-8 (octal) - shows up occasionally
- Base-2 (binary) - most important! (we've been discussing 2 things!)
- Base-16 (hexadecimal) - nice grouping of bits

Exercise

Turn 1101011110010_2 into base-10:

1 1 0 1 0 1 1 1 1 0 0 1 0

$2^{12}+2^{11}+2^9+2^7+2^6+2^5+2^4+2^1$

$=4096+2048+512+128+64+32+16+2$

$=6898$

Exercise

Turn 342_{10} into binary:

512	2^8 256	2^7 128	2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1
	1	0	1	0	1	0	1	1	0

342 - 256 = 86

86 - 64 = 22

22 - 16 = 6

6 - 4 = 2

2 - 2 = 0

Exercise

Turn 1101011110010_2 into **hexadecimal**:

1101 1110 0010

0001 1010 1111 0010

1 A F 2

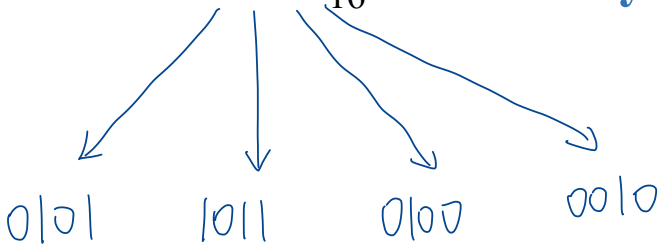
$1AF2_{16}$

//Turn 1101011110010_2 into **8-bit hexadecimal**:

How to extend it ?

Exercise

Turn $5b42_{16}$ into **binary** :



$5b42_{16} = 0101\ 1011\ 0100\ 0010_2$

Exercise

Turn $5b42_{16}$ into **decimal** :

$$\begin{array}{cccc} 5 & b & 4 & 2 \\ 16^3 & 16^2 & 16^1 & 16^0 \end{array}$$

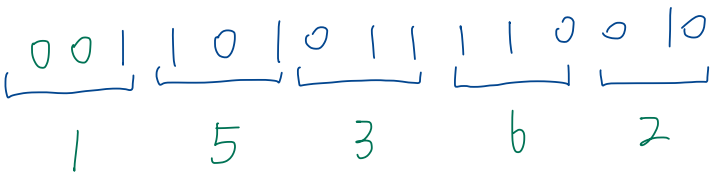
$$5\times16^3 + 11\times16^2 + 4\times16^1 + 2\times16^0$$

$$= 20480 + 2816 + 64 + 2$$

$$= 23362$$

Exercise

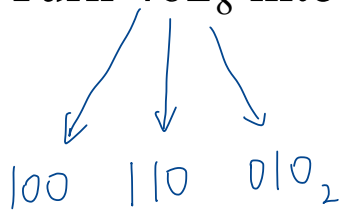
Turn 1101011110010_2 into **octal** :



So, $1101011110010_2 = 15362$

Exercise

Turn 462_8 into **binary** :



Binary Addition

01101011 + 01100101

```
  0 1 1 0 1 0 1 1   (107)
+ 0 1 1 0 0 1 0 1   (101)
-----------------
  1 1 0 1 0 0 0 0   (208)
```

11101011 + 11100101

```
  1 1 1 0 1 0 1 1   (235)
+ 1 1 1 0 0 1 0 1   (229)
-----------------
1 1 1 0 1 0 0 0 0   (208)
```

The correct answer should be 464.
But I only have 8 bits for my result (read as 208)

range for 8 bits: (0 ~ 255)

$2^8 - 1 = 256 - 1 = 255$

Binary Subtraction

01111011 - 01100101

```
  0 1 1 1 1 0 1 1   (123)
- 0 1 1 0 0 1 0 1   (101)
-----------------
  0 0 0 1 0 1 1 0   (22)
```

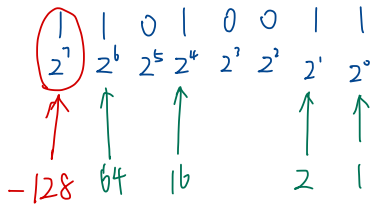
Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

11010011

What is its value in decimal?

Method 1:



$-128 + 64 + 16 + 2 + 1 = -45$

Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

11010011

What is its value in decimal?

1 1 0 1 0 0 1 1
128 64 32 16 8 4 2 1

1. Flip all bits
2. Add 1

00101100
32 16 8 4 2

\+ 1

00101101

32+8+4+1 = 45 ⇒ −45

−128 +64+16+2+1 = −45

Example: Bitwise AND

```
  11001010
& 01111100
----------
  01001000
```

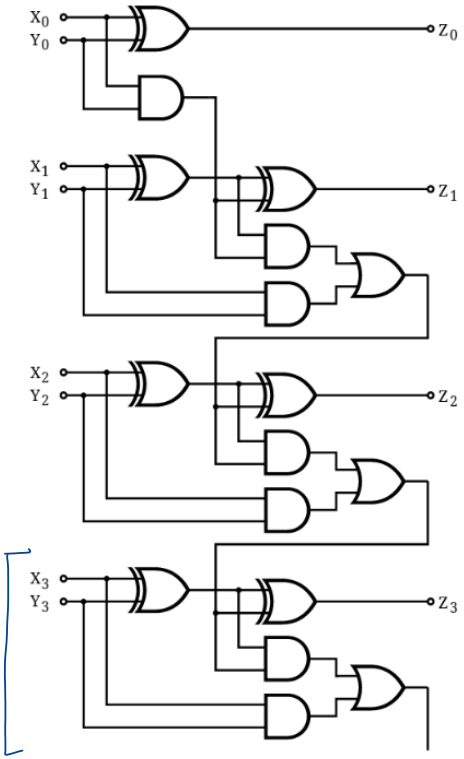
Example: Bitwise OR

```
  11001010
| 01111100
----------
  11111110
```

Example: Bitwise XOR

```
  11001010
^ 01111100
----------
  10110110
```

Ripple-Carry Adder



$X_3\ X_2\ X_1\ X_0$
$+\ Y_3\ Y_2\ Y_1\ Y_0$
$Z_3\ Z_2\ Z_1\ Z_0$

repeat if more bits

verify:

unsigned. 1 1 1 1 (15)
+ 1 1 1 1 (15)
drop ← (1) 1 1 1 0 (14) ← overflow!

signed? 1 1 1 1 (-1)
+ 1 1 1 1 (-1)
1 1 1 0 (-2)
works!

Operations (on Integers)

Logical not: $!x$

- $!0 = 1$ and $!x = 0, \forall x \neq 0$
- Useful in C, no booleans
- Some languages name this one differently

0 means false
Any non-zero value means true.

C does not have a built-in Boolean type in older standards.
Logical expressions return integers (0 or 1)

Example:
```
if(!ptr) {
    // ptr is NULL
}
```

Python: not
Java/C/C++: !
Bash: !

Operations (on Integers)

Left shift: $x << y$ - move bits to the left

- Effectively multiply by powers of 2

Right shift: $x >> y$ - move bits to the right

- Effectively divide by powers of 2
- Signed (extend sign bit) vs unsigned (extend 0)

Floating Point Example

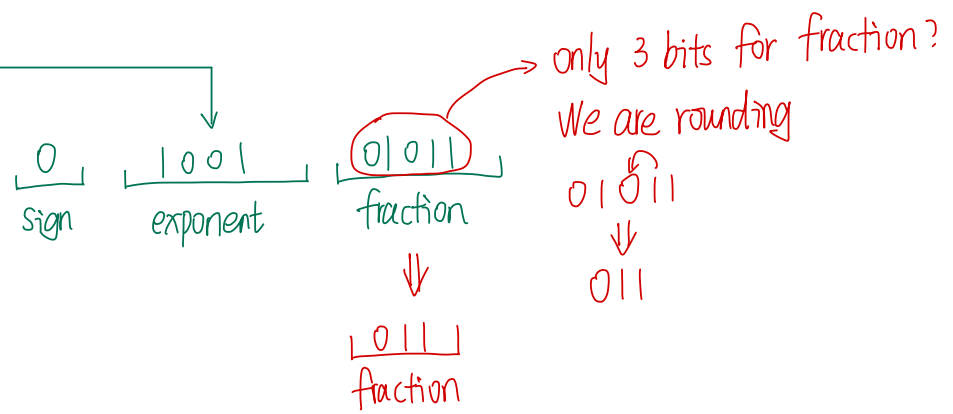
1 bit: sign
4 bits: exponent
3 bits: fraction

101.011_2

1.01011×2^2

2's complement for 2: 0010

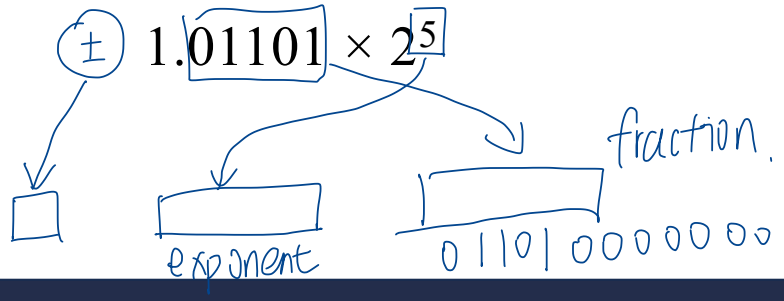
add the bias: 0010
+0111
1001



Floating Point in Binary

How do we store them?

- Originally many different systems
- IEEE standardized system (IEEE 754 and IEEE 854)
- Agreed-upon order, format, and number of bits for each



Floating Point Example

$\underbrace{101}_{5}.\underbrace{011}_{}_2$

couple of ways to think about what comes after the binary point.

① in fractions

101.011

$2^2\ 2^1\ 2^0\ \ 2^{-1}\ 2^{-2}\ 2^{-3}$

$4+1=5 \qquad 2^{-2}+2^{-3}=\frac{1}{4}+\frac{1}{8}=\frac{3}{8}$

so: $101.011 = 5\frac{3}{8}$

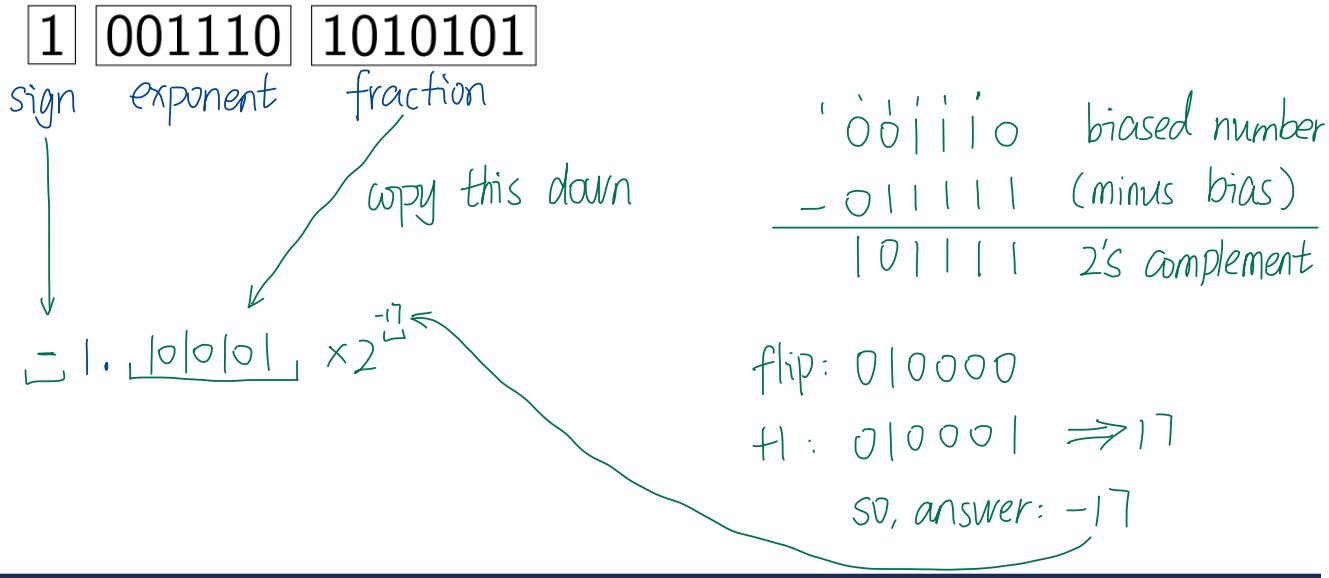
② positionally

3 positions: up to $2^3=8$ values (positions)

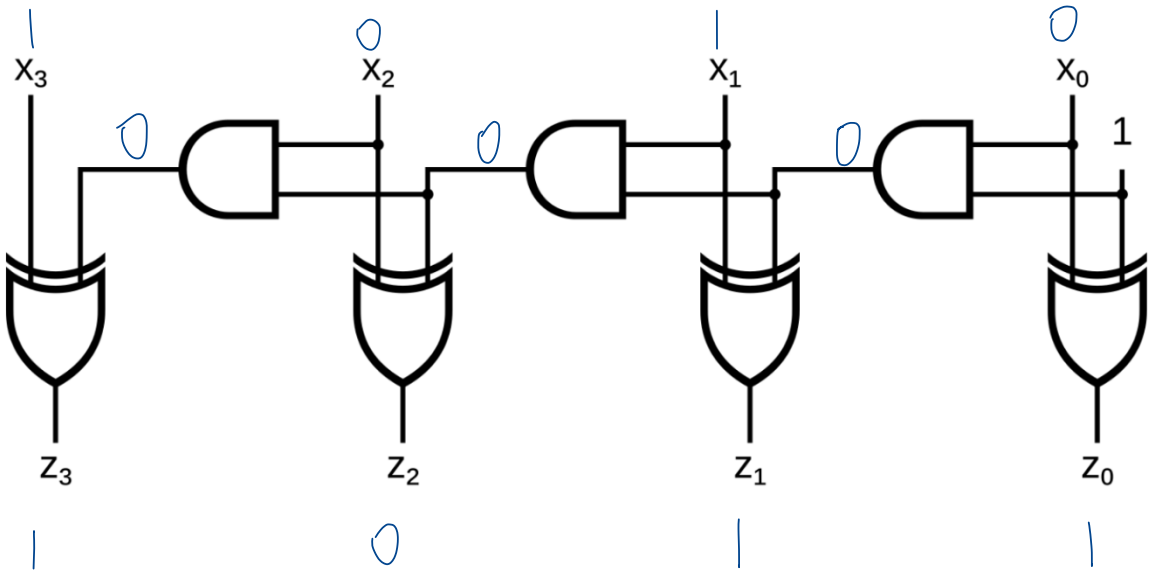
011 is 3, so 3/8 (three · eight)

Floating Point Example

What does the following encode? 14 bits.

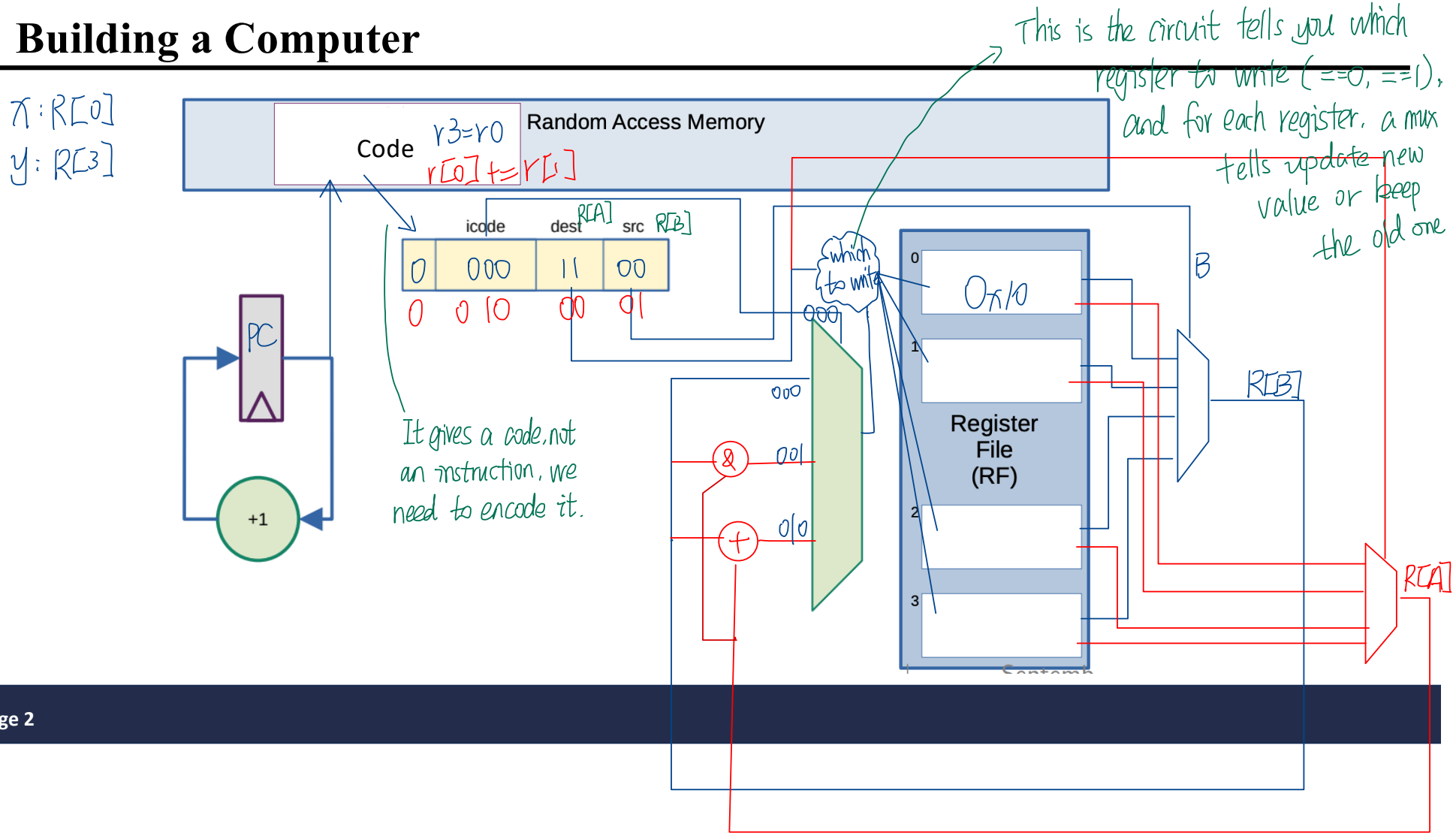


What does this circuit do?



It adds 1 !

Building a Computer



Encoding 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

in decimal

Example 1: r1 += 19

19 in hexdecimal: 0x13

hex: 0b13

Encoding Instructions

idea: ①. I have a value in memory at address hex 82

②. I want to add whatever in R3 to that value.

Example 2: M[0x82] += r3

Read memory at address 0x82, add r3, write back to memory at same address

One point: No instructions allow us to pass an immediate value as the address.
So let's save ourselves some time: just put 82 in a register.
Then we use icode 4 to read it out, icode 5 to write it back.

r2 = 0x82 — 0 110 10 00 — 82

r1 = M[r2] — 0 100 01 10 — 4 6

r1 += r3 — 0 010 01 11 — 2 7

M[r2] = r1 — 0 101 01 10 — 5 6

our machine reads 0 and 1.

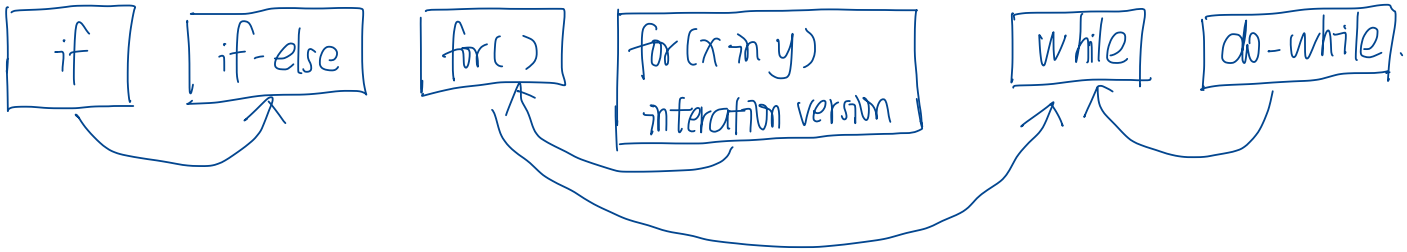
But, for us → easier to read → pairs of hex

68 82 46 27 56

One interesting finding: first hex is always our icode!

Our code to this machine code

How do we turn our control constructs into jump statements?



how to convert for to while:

```
for (int i=0; i<25; i++){
    =
    =
}
```

⇒

```
int i=0
while(i<25) {
    =
    =
    i++; }
```

In C/Java, we have if/else. But from the CPU perspective, there is no such thing as if/else — only sequential execution and jumps. Think of the CPU as walking forward on a road. Default behavior: keep moving forward. Only when we do NOT want to continue forward, we need to jump somewhere else. So the compiler naturally ask "When should I jump away?" instead of "When should I continue?"

if/else to jump

```
if ( D ) {
    A
} else {
    B
}
C.
```

if condition D is true, I will do A, skip B, continue to do C

For human: if D is true → A, otherwise → B. But for machine: 1. The CPU will continue downward by default. 2. If D is false, where should we go?

if(!D), jump to B

A

jump to C (unconditional jump)

B

C

2 situations:
① if D is true: don't jump, continue A, then C.
② if D is false: jump to B, then C.

jumps happens at 2 places where?

it looks like we're "thinking backwards", but actually we are just following the CPU's natural rule: continue unless force to jump.

Why do compilers generate code this way?

① "Fewer jumps" ⇒ faster execution

② "Sequential flow" ⇒ better branch prediction

③ "This pattern appears in all architectures".

while to jump

while (C) {

A

}

B

How do I know where to jump?
① hard code if you know the address
② icode 3-3

if (!C), jump to B

A

jump to A? No! Infinate loop!

jump to if, do the condition check!

B

each loop iteration has 2 jump checks.

if (!C), jump to B

A

if (C), jump to A.

B

each loop iteration only has 1 jump check.

It's not the same loop

It's a do-while loop!

Encoding 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

Example 3: if r0 < 9 jump to 0x42

I don't have an instruction say r0<9.
I need "r0<=0" for icode 7, what should I do?

r0<9 $\Leftrightarrow$ r0<=8 $\Leftrightarrow$ (r0-8)<=0
$\Leftrightarrow$ r0+=-8 (0xF8)
r0<=0

r1 = 0x42 — 0 110 01 00 — 42
6 4 42

r0 += F8 — 0 110 00 10 — F8
6 2 F8

if r0<=0, PC=r1 — 0 111 00 01
(R0) (R1)
7 1

b442b2F871

register : ir : current instruction.
PC : address of the next instruction

Encoding 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

Example 4: 0x17 * 3

0x17 * 3 ⇒ 0x17+0x17+0x17 ⇒

for(i=0;i<3;i++) x+=0x17;

```
x=0;
i=0;
while(i<3){
  x+=0x17;
  i+=1;
}
```

```
x=0;
i=~0;-2
//HERE
x+=0x17
i+=1;
if(i<3)jump HERE
if(i<=2)
if(i-2<=0)
if (i<=0) jump HERE
```

icode 3-3: rA=PC (Save where to back)

0011 | 10 | 11
3 | r2 | 3

what values we need?

x → r0

i → r1

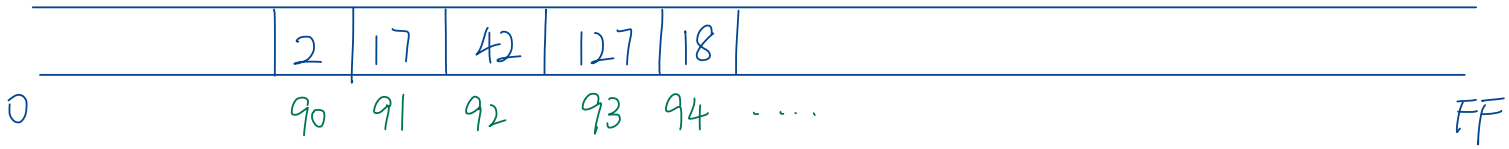
HERE → r2

y=0x17 → immediate

Arrays

arr = {2, 17, 42, 127, 18} @0x90

I assume all these are one byte so that each will fit in one of these slots.



arr[3] = 0x90 + 3 = 0x93

3 → index

0x90 → position of where our array start.

Instructions Set Architecture

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

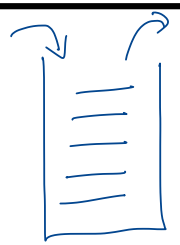
Lots of flexibility and freedom to build things that would be faster, like hyperthreading. I don't worry about on the software side.

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

The Stack

(One solution won out.)



stack of plates

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

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

The Stack: Push and Pop

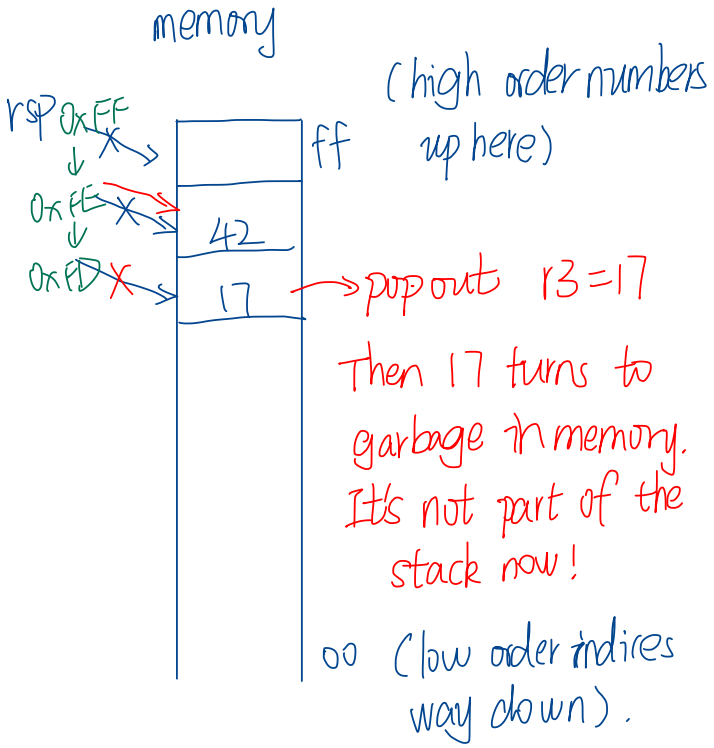
push r0

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

pop r2

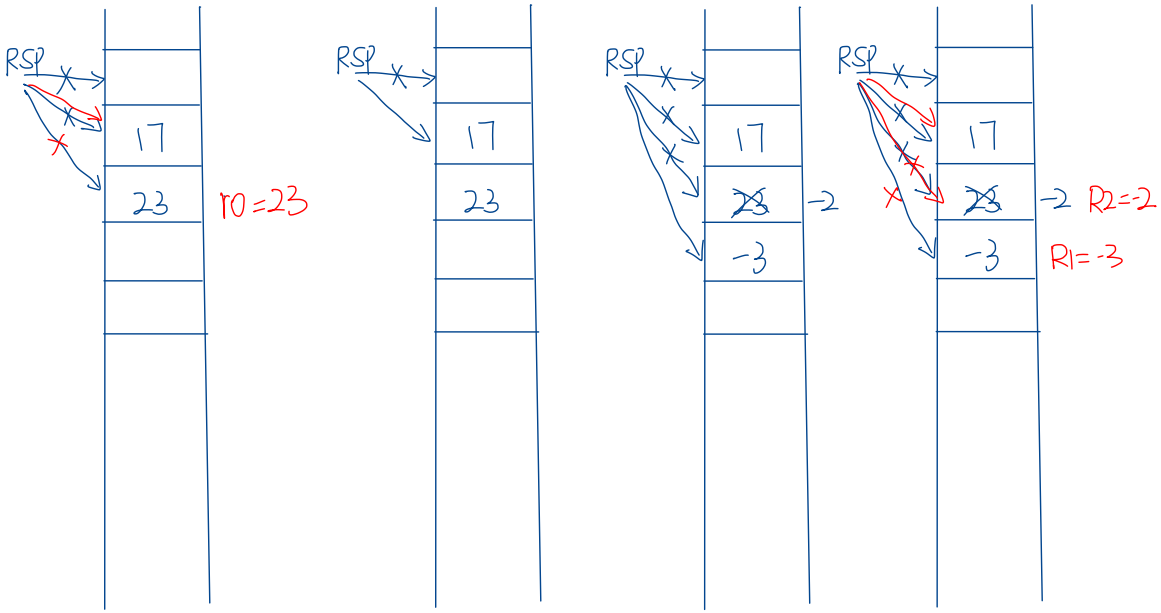
- Read value from "top", save to register
  - r2 = M[rsp]
  - rsp += 1

push 42
push 17
pop r3

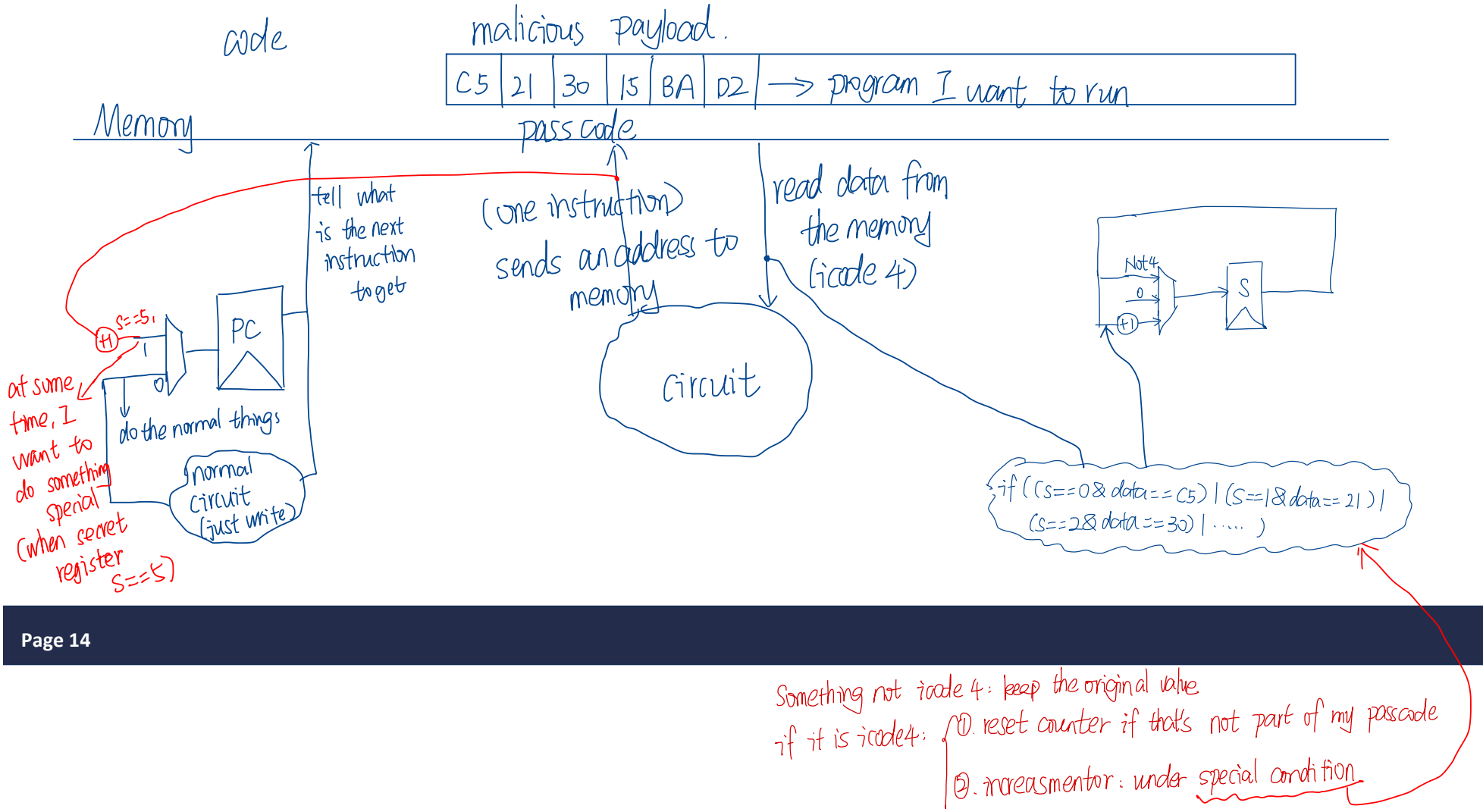


The Stack: Push and Pop

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



Our Hardware Backdoor



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

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

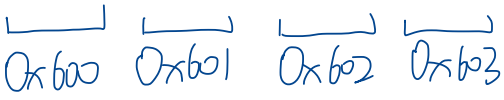
You've seen all 3 of these already.

Rules

0x | 00 | AB | CD | EF (4 bytes).

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



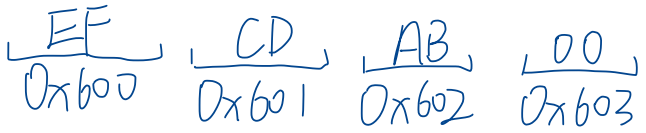
starting from an address and going up

Ordering Values

0x|00|AB|CD|EF

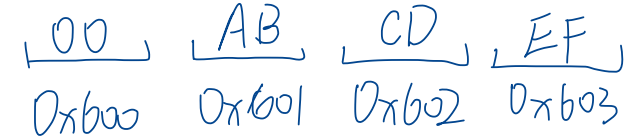
Little-endian

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



Big-endian

- Store the high order part/byte first



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 for processors 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

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 (extra information)
  - 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)

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

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

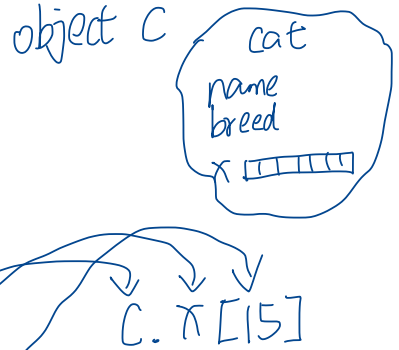
- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

```
// we can have comments!
```

Addressing Memory

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

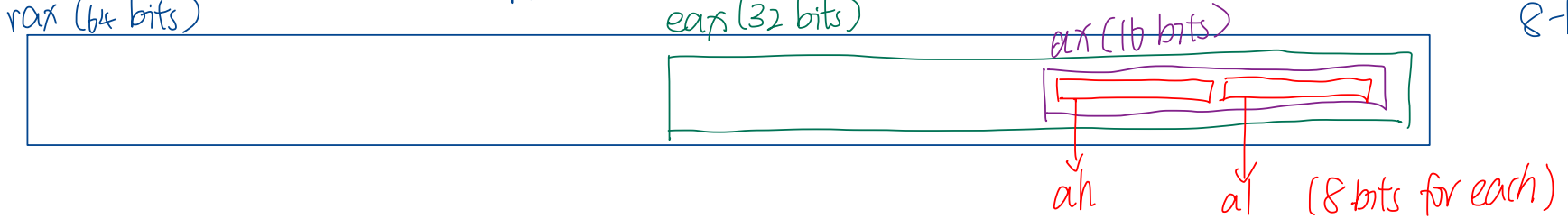
- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `2130 + %rax + (%rsp * 8)`
- Common usage from this example:
  - `rax` - address of an object in memory
  - 2130 - offset of an array into the object
  - `rsp` - index into the array
  - 8 - size of the values in the array (used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

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

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
  - `ja`, `jb`: unsigned
  - `jo`: If there's a overflow
  - `js`: If the signed bit is set.

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.

→ You can do something like move things around.

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

  - 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)
```

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)

`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

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

Why? Caller and callee share the same registers.

Calling conventions - recommendations for making function calls

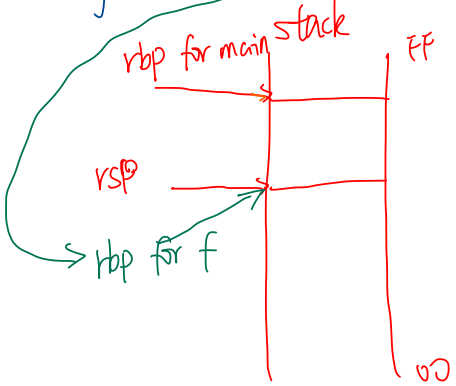
- Where to put arguments/parameters for the function call?
- Where to put return value? in `rax` before calling `retq`
- What happens to values in the registers?
  - **Callee-save** - The function should ensure the values in these registers are unchanged when the function returns
    - `rbx`, `rsp`,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

① push the old values before calling ② pop the values before returning.

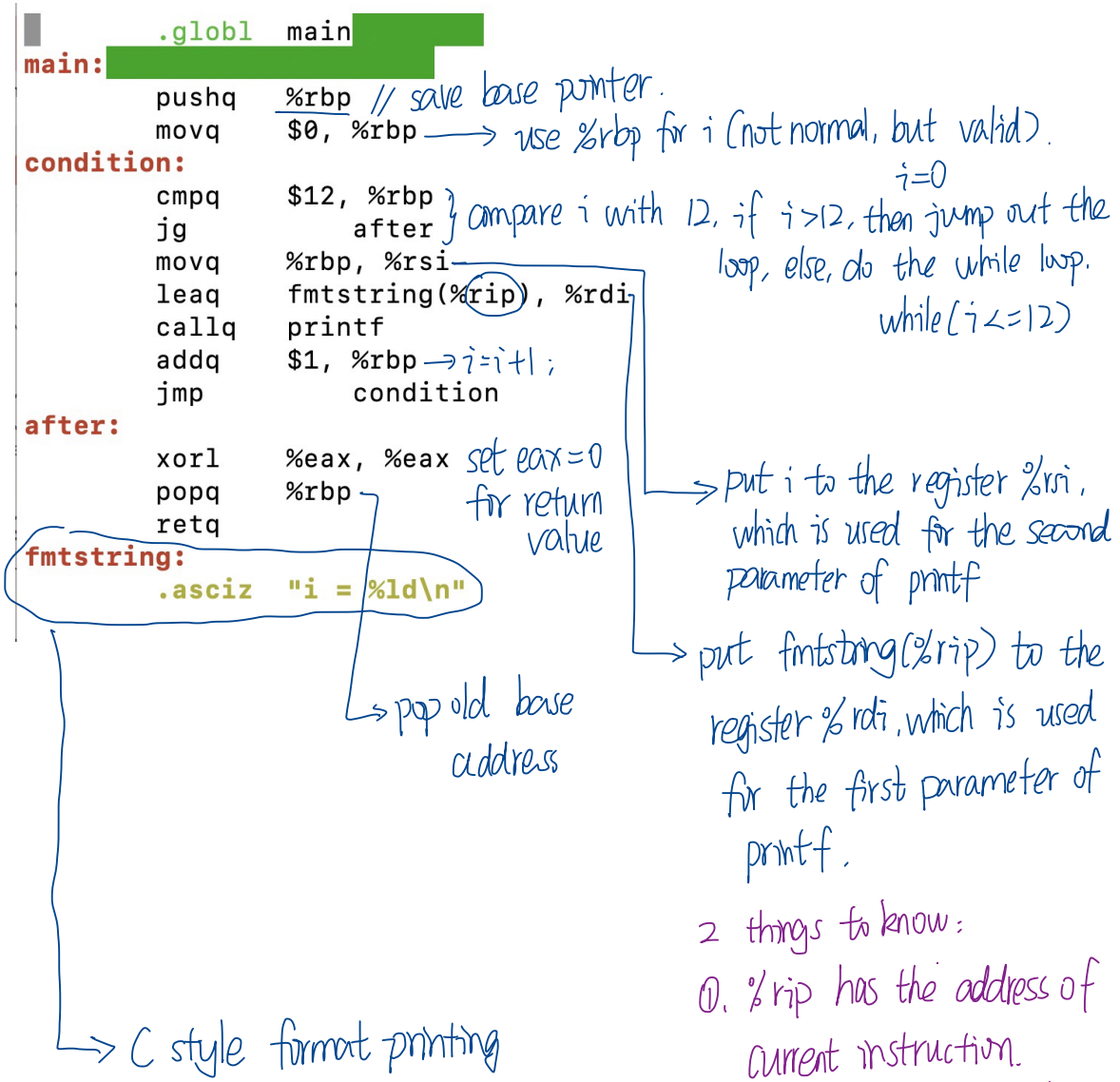
Example: Functions

```
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.
```


.globl main
main:
pushq %rbp // save base pointer.
movq $0, %rbp → use %rbp for i (not normal, but valid).
condition:
cmpq $12, %rbp } compare i with 12, if i>12, then jump out the
jg after loop, else, do the while loop.
movq %rbp, %rsi
leaq fmtstring(%rip), %rdi
callq printf
addq $1, %rbp → i=i+1;
jmp condition
after:
xorl %eax, %eax set eax=0 for return value
popq %rbp
retq
fmtstring:
.asciz "i = %ld\n"
i=0
while(i<=12)
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.

Most Common Instructions

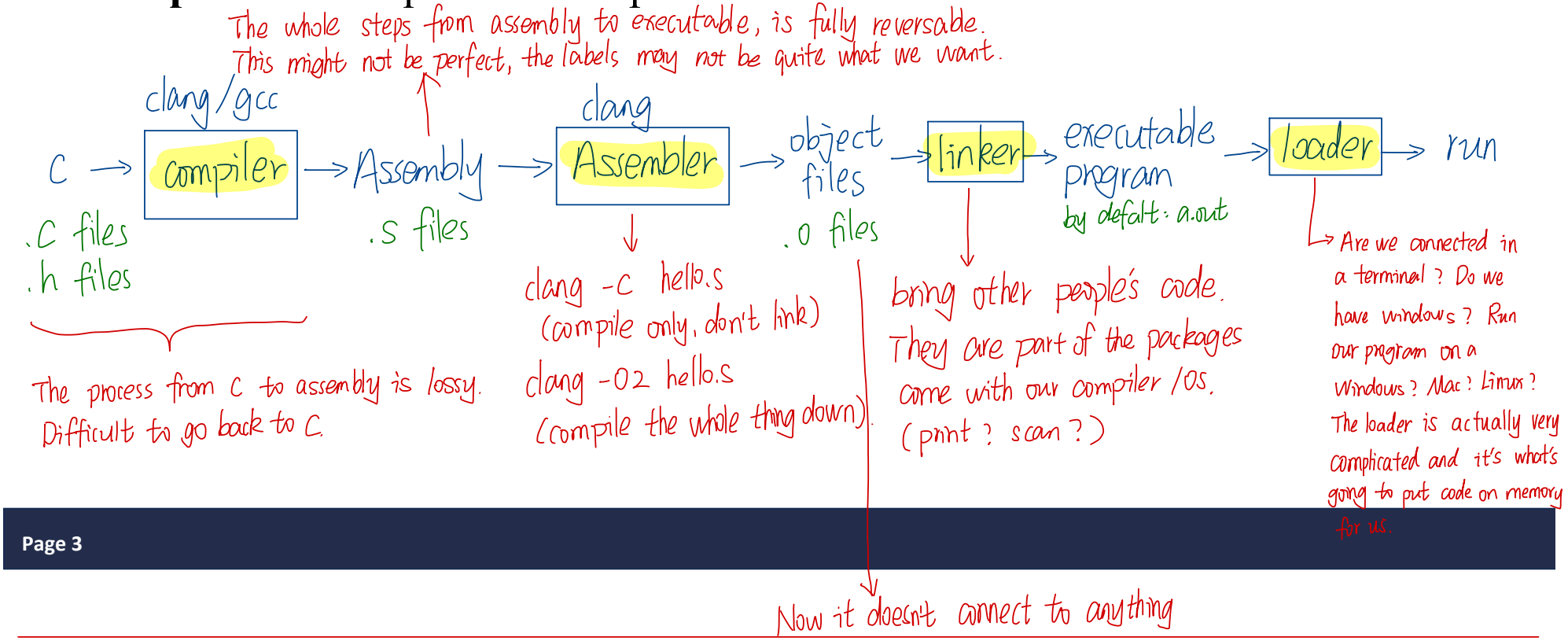
- `mov` - =
- `lea` - load effective address
- `call` - push PC and jump to address
- `add` - +=
- `cmp` - set flags as if performing subtract
- `jmp` - unconditional jump
- `test` - set flags as if performing &
- `je` - jump iff flags indicate == 0
- `pop` - pop value from stack
- `push` - push value onto stack
- `ret` - pop PC from the stack

Patents and Copyright

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



ls /usr/lib : lots of compiled things ready to be used.

ls /usr/lib64 | wc -l : show how many files.

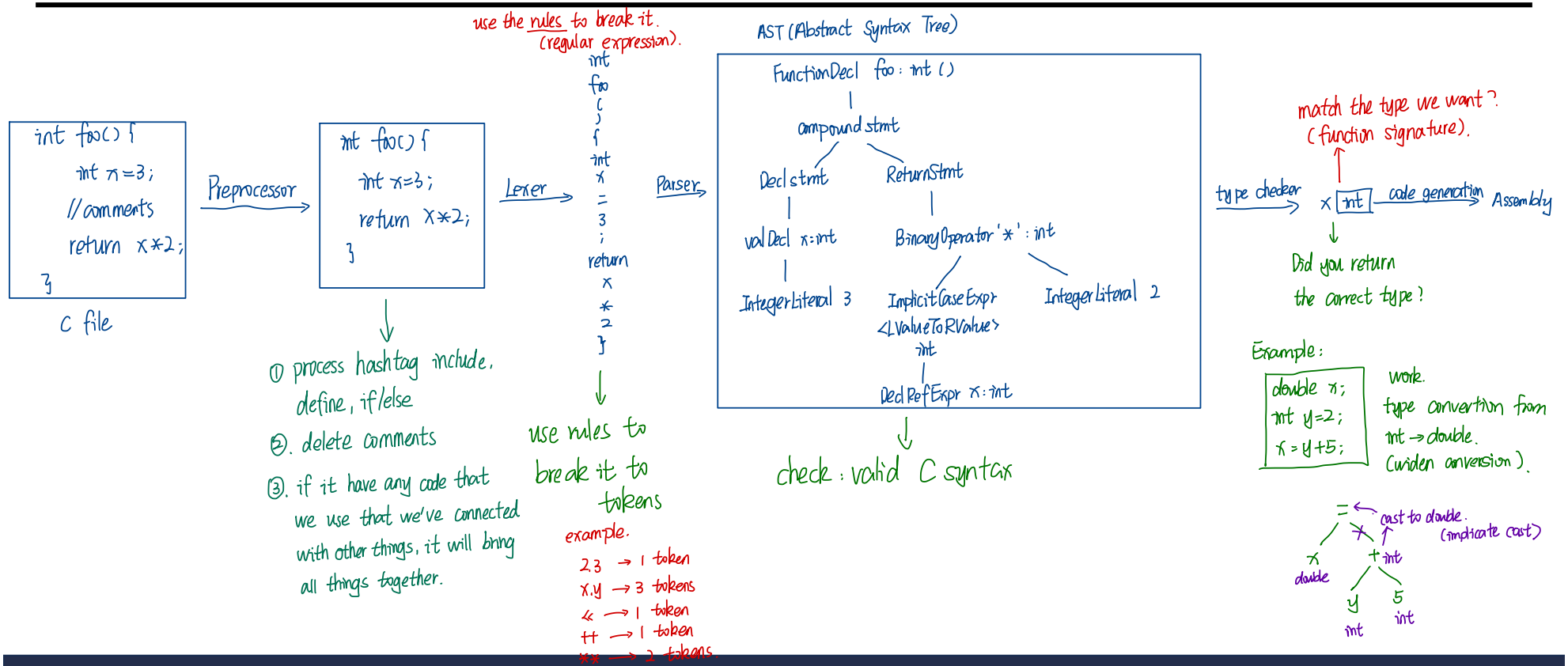
Compiling C to Assembly

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

has a lot of things like hashtags

Compiling C to Assembly



```c
int main() {
    return 0;
}
```

clang -O0 -c hello.c -Wno-implicit-function-declaration -o hello.o

-c → compile only, no link.

-O0 → No optimization

-Wno-implicit-function-declaration → I don't want the warning. (for example, I call a function but didn't declare it.)

llvm-objdump -d hello.o

-d → disassemble.

```
hello.o:        file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       b: 31 c0                         xorl    %eax, %eax
       d: 5d                            popq    %rbp
       e: c3                            retq
```

This is the disassemble code for main. start from address 0.

This is an unnecessary local variable initialization (a dead store), artificially introduced by compiler under -O0 for simplicity and debuggability.

1. why xorl %eax, %eax ?

① Can we use movl $0, %eax ? Yes, but xor is shorter and faster.

② Why not xorq %rax, %rax ?

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. Performing a 32-bit xor on eax automatically clears the upper 32 bits to zero.

2. For different return types:

Regardless of whether the return type is int, short, long, long long. according to the x86-64 calling convention, return values are passed via the rax register. For floating-point types, such as "float" and "double", use xmm0. (Completely different hardware).

```c
int main() {
    return 0;
}

long foo(){
    return 0;
}
```

```
0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       b: 31 c0                         xorl    %eax, %eax
       d: 5d                            popq    %rbp
       e: c3                            retq
       f: 90                            nop

0000000000000010 <foo>:
      10: 55                            pushq   %rbp
      11: 48 89 e5                      movq    %rsp, %rbp
      14: 31 c0                         xorl    %eax, %eax
      16: 5d                            popq    %rbp
      17: c3                            retq
```

```c
int main() {
    puts("It's Friday!");
    return 0;
}
```

```
hello.o:        file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 48 83 ec 10                   subq    $0x10, %rsp
       8: c7 45 fc 00 00 00 00          movl    $0x0, -0x4(%rbp)
       f: 48 8d 3d 00 00 00 00          leaq    (%rip), %rdi
      # 0x16 <main+0x16>
      16: b0 00                         movb    $0x0, %al
      18: e8 00 00 00 00                callq   0x1d <main+0x1d>
      1d: 31 c0                         xorl    %eax, %eax
      1f: 48 83 c4 10                   addq    $0x10, %rsp
      23: 5d                            popq    %rbp
      24: c3                            retq
```

→ It calls something it doesn't know about. Because I haven't actually told it.

The linker will eventually overwrite and tell it where to actually call "puts".

```
mrq9gz@portal03:~/examples1$ clang hello.c -o hello
hello.c:2:5: error: call to undeclared function 'puts'; ISO C99 and later do not support implicit function declarations [-Wimplicit-function-declaration]
    2 |     puts("It's Friday!");
      |     ^
1 error generated.
```

```c
int puts(const char * string);

int main() {
    puts("It's Friday!");
    return 0;
}
```

→ manually declare puts.

Declaration → let compiler know the signature of the function.

```c
#include <stdio.h>

int main() {
    puts("It's Friday!");
    return 0;
}
```

Also works, and better

if I use "cpp hello.c | less", I can see all things in stdio.h is actually copied to the hello.c file.

cpp: c preprocessor

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.

char is implementation-defined. { char - depends on compiler/platform
signed char
unsigned char

Example of creating a variable: unsigned long long x = 5;

Data Types in C (check "Readings → C Reference" for exponent bits and fraction bits)

Floating point

- float
- double

Data Types in C

Pointers - how C uses addresses!

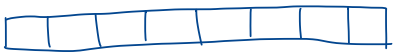
- Hold the address of a position in memory
- Need to know the kind of information stored at that location

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;` x  (4 bytes)

`int *y;` y  (8 bytes)

`y=x;` → doesn't work

`y=&x;` It works!

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

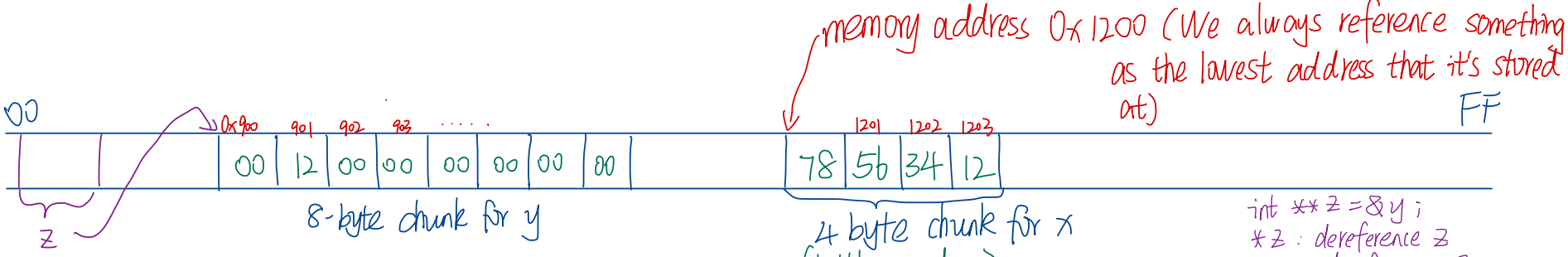
int **z = &y; z is a pointer to a pointer to an int.
z has the address of y.

later, if I say "*z", that means y.
"**z": the value of x.

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).

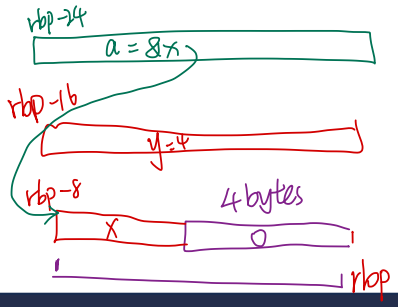
int **z = &y;
*z : dereference z
**z: dereference *z
**z = 0x00000000;
(set x to 0x00000000)
*z = 25; (Warning: you're trying to assign an integer to a pointer)
(But later, when you use it as an address, may Seg Fault).

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;
}
```




```
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

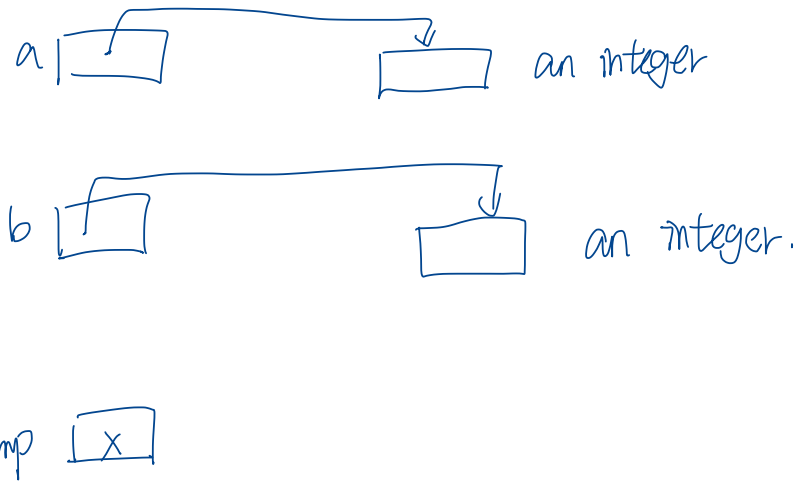
signed extend
1 to 8
long to quad

6 going from long to int. move the value of y into rcx, but I will just read out the value of ecx.

Example

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)

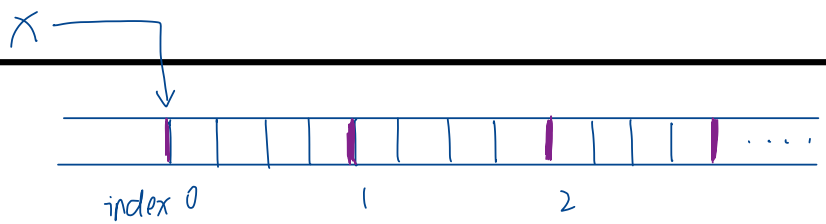
Array: 0 or more values of same type stored contiguously in memory

- 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 (When I create an array, it actually create a pointer to the first thing in that list.)
- Can declare array literals:

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

(optional)

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)

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

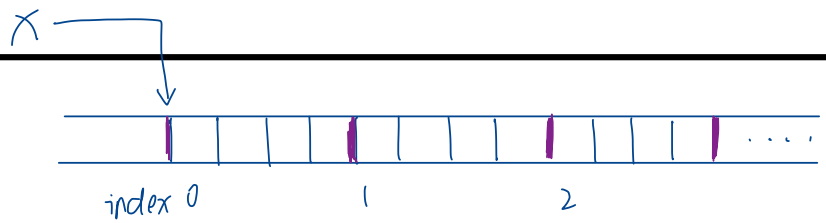
- Declare as you would use: `int myarr[100];` (100 integers in my array)
- `sizeof(myarr)` = 400 — 100 4-byte integers (How big is my array in bytes?)
- `myarr` treated as pointer to first element
- Can declare array literals:

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

(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

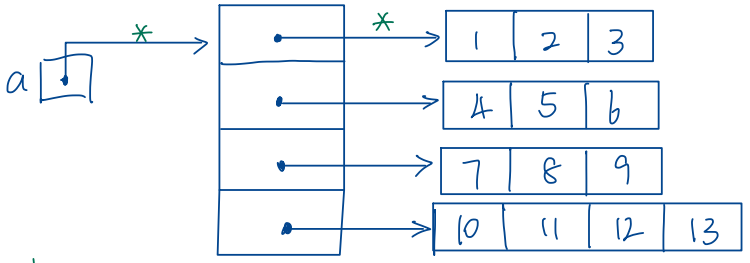
one pointer or an array of pointer?

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 8 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 bnger, I need to add some suffixes.

- Literal values - integer literals are implicitly cast
  - `unsigned long very_big = 92233720368547758O8uL`
  - `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)
    `int foo = e;` It's a type!
    0 1 2 101
    I can change where I'm at using "="
- `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).
- 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)

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

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;
```

long a; 8
int b; 4
short c; 2
char d; 1
4+2+1=7
may be one byte of padding?
I don't know.

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

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 what's 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 };
```

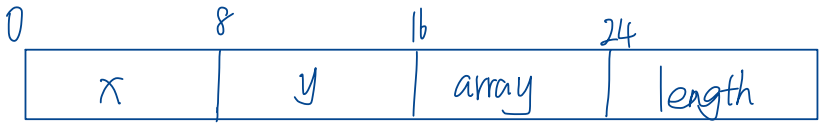
struct { ... } defines an anonymous struct (no struct name given).
The typedef immediately gives it the alias foo.
You can now create variables like foo z; instead of writing struct something z;.

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.

arg->x is same as (*arg).x

```
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.

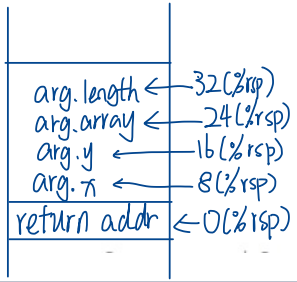
```
long sum1(foo arg) {
    long ans = arg.x;
    for(long i = 0; i < arg.length; i += 1)
        ans += arg.y * arg.array[i];
    return ans;
}
```

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.

```
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:                       array  i  (array+i*8)
    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     .LBB0_2
.LBB0_3:
    retq
```

integer multiply (signed)

increment




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

Switch

```
void describe(int age) {
    switch (age) {
        case 1:
            puts("You're one.");
            break;
        case 2:
            puts("You're two.");
            break;
        case 4:
            puts("You're four.");
        case 5:
            puts("You're four or five.");
            break;
        default:
            puts("You're not 1, 2, 4 or 5!");
    }
}
```

```
    .text
    .globl  describe
describe:
    pushq   %rax
    movl    %edi, %eax
    addl    $-1, %eax
    cmpl    $4, %eax
    ja  Label5
    leaq    Text0, %rdi
    leaq    JumpTable, %rcx
    movslq  (%rcx,%rax,4), %rax
    addq    %rcx, %rax
    jmpq    *%rax
Label2:
    leaq    Text1, %rdi
    jmp Label6
Label5:
    leaq    Text4, %rdi
    jmp Label6
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
```

```
Label6:
    xorl    %eax, %eax
    callq   puts
    popq    %rax
    retq
```

```
    .section    .rodata
JumpTable:
    .long   Label6-JumpTable
    .long   Label2-JumpTable
    .long   Label5-JumpTable
    .long   Label3-JumpTable
    .long   Label4-JumpTable

Text0:
    .asciz  "You're one."
Text1:
    .asciz  "You're two."
Text2:
    .asciz  "You're four."
Text3:
    .asciz  "You're four or five."
Text4:
    .asciz  "You're not 1, 2, 4 or 5!"
```

It looks like a bunch of labels. And it's going to work exactly like we've seen labels. Where I jumped to a label and then I start executing code.

Not like a function: I go there, run things, then return.

Not like if statement: I do a bunch of conditional jumps.

One thing to note. we have to switch on an integer. That integer is going to be the index into our array of labels. (Indexing into an array of labels and picking which one we want to jump to).

```
    .globl  describe
describe:
    pushq   %rax
    movl    %edi, %eax
    addl    $-1, %eax
    cmpl    $4, %eax
    ja  Label5
    leaq    Text0, %rdi
    leaq    JumpTable, %rcx
    movslq  (%rcx,%rax,4), %rax
    addq    %rcx, %rax
    jmpq    *%rax
Label2:
    leaq    Text1, %rdi
    jmp Label6
Label5:
    leaq    Text4, %rdi
    jmp Label6
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
Label4:
    leaq    Text3, %rdi
```

→ first parameter (age).

→ The index of an array starts from 0, so we want to map 1~5 to 0~4

→ if index > 4, then jump to Label 5

→ It's preparing the argument for a later call like puts (Text0).

→ Load the address of the jump table into %rcx.

→ Loads a 32-bit offset from the jump table entry indexed by %rax and sign-extends it to 64 bits. Each entry is 4 bytes long (hence the *4).

→ Adds the base address of the jump table to that offset.

→ indirect jump to the address in %rax.

→ load Text1 and jump to Label 6.

→ load Text4 and jump to Label 6

```
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
Label4:
    leaq    Text3, %rdi
Label6:
    xorl    %eax, %eax
    callq   puts
    popq    %rax
    retq
```

→ empty out eax and outputs.

→ in assembly a long type is 32 bites.

```
    .section    .rodata
JumpTable:
    .long   Label6-JumpTable
    .long   Label2-JumpTable
    .long   Label5-JumpTable
    .long   Label3-JumpTable
    .long   Label4-JumpTable

Text0:
    .asciz  "You're one."
Text1:
    .asciz  "You're two."
Text2:
    .asciz  "You're four."
Text3:
    .asciz  "You're four or five."
Text4:
    .asciz  "You're not 1, 2, 4 or 5!"
```

→ jump table. Why they choose this order?
I don't know but it doesn't matter.

→ read only section

Calling Functions

The C code

```
long a = f(23, "yes", 34uL);
```

follow the calling conventions if I don't know anything else about function f.

compiles to

```
movl $23, %edi
leaq label_of_yes_string, %rsi
movq $34, %rdx
callq f
# %rax is "long a" here
```

put the address of "yes" to rsi.

without respect to how `f` was defined. It is the calling convention, not the type declaration of `f`, that controls this.

Calling Functions

But, if the C code has access to the type declaration of `f`, then it might perform some implicit casting first; for example, if we declared

```
long f(double a, const char *b, double c);
```

```
long a = f(23, "yes", 34uL);
```

We need to make sure that we have the declaration of the function before we call it. So the type checker knows how the function supposed to look

then the call would be interpreted by C as having implicit casts in it:

```
long a = f((double)23, "yes", (double)34uL);
```

Calling Functions

When I call a function with floating pointer numbers, there's a whole set of registers that just yield floating point numbers.

and the arguments would be passed in floating-point registers, like so:

```
movl $23, %eax
cvtsi2sd %eax, %xmm0          # first floating-point argument
leaq label_of_yes_string, %rdi # first integer/pointer argument

movl $34, %eax
cvtsi2sd %eax, %xmm1          # second floating-point argument

callq f
# %rax is "long a" here
```

Not edi anymore. It's a general-purpose integer, not an argument.

the first floating point number register. (follow the calling conventions for floating point numbers).

cvtsi2sd: convert signed integer to signed double.

For different types of the parameters, use different kinds of registers in order.

Macros

```
#define NAME something else
```

- Object-like macro
- Replaces `NAME` in source with `something else`

It does this at the text level.

```
#define NAME(a,b) something b and a
```

- Function-like macro
- Replaces `NAME(X,Y)` with `something Y and X`

whatever is after the comma until the second parentheses is the second thing.

Whatever is the first thing after the opening parentheses until the comma.

text level

Lexical replacement, *not* semantic

Interesting Example

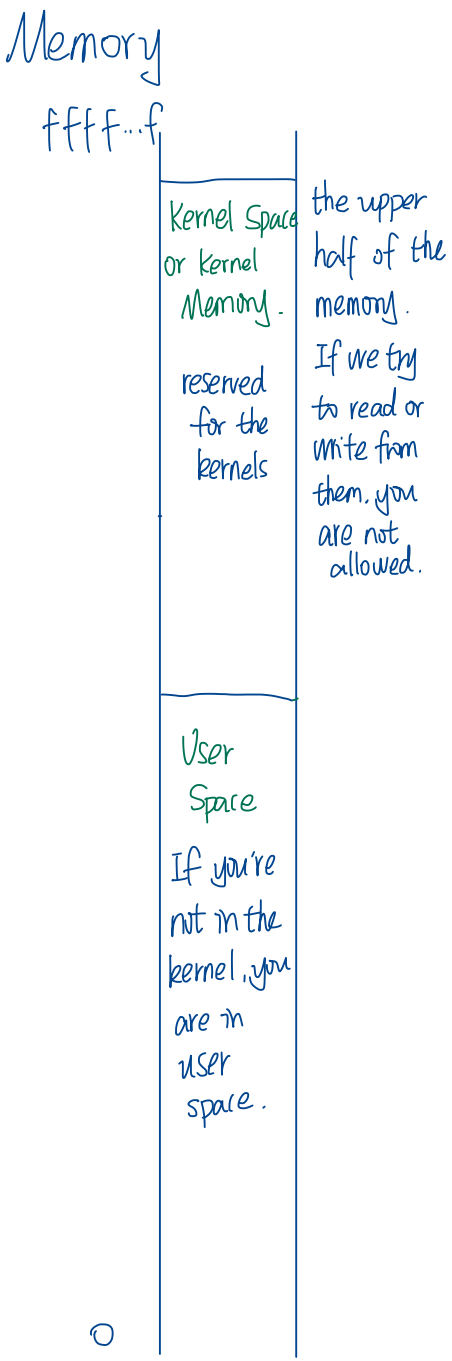
```
#define TIMES2(x)   x * 2          /* bad practice */
#define TIMES2b(x) ((x) * 2)       /* good practice */

int x = ! TIMES2(2 + 3);
```

x = !2+3*2; ⇒ x = 0+6 ⇒ x = 6.
(0 and 6)

```
int y = ! TIMES2b(2 + 3);
```

x = !((2+3)*2);

Be very explicit when you're using the macro define. Wrap them in parentheses and enforce an order of operations.



From assembly:

① Generate a memory address and go to memory.

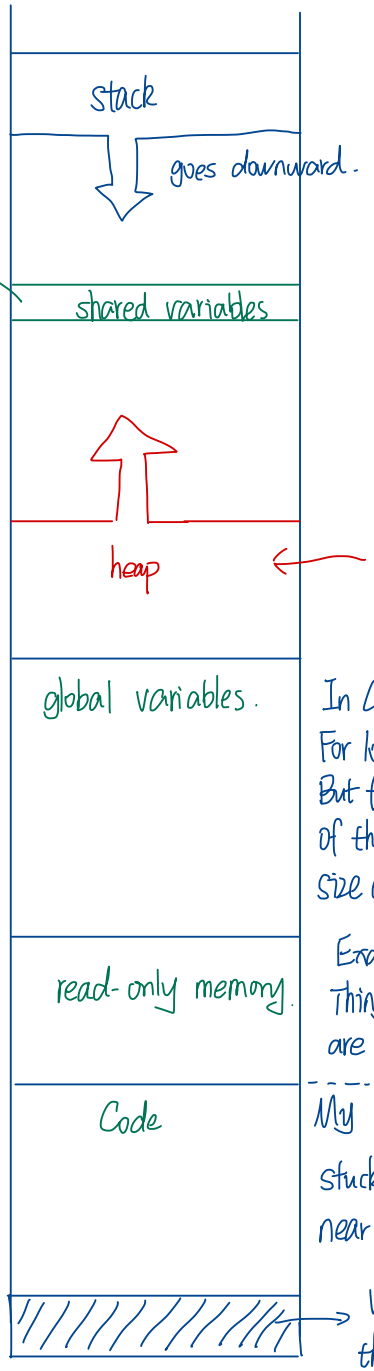
② hardware is going to check if we're actually allowed to read/write from that place in memory. (rely on a piece of software).

2 names for that: kernel or OS (Operating system).

↳ help the hardware decide whether these addresses are good or not.

It divides memory up into a couple of different segments. a chunk of memory

It's going to have different properties based on what segment of memory I'm in.



stack frame:

Anytime I call a function, there are a couple of things that get put on the stack: ① parameters
② return address.
③ local variables.

When I return, RSP moves away, memory does not get erased.

One thing missing: pointers, variables that we want to leave when our functions return. (pass to another function?)

In C. I can define variables outside of functions
For local variables in functions: put values in stack.
But for global variables, we put them here. (The size of the variables are set at compile time, and the size cannot be changed.)

Example: string literals
Things are not code, but that were in my code that are really helpful for my program.

My code is going to be stuck in memory somewhere near the bottom.

We put the code near the bottom, but not at the bottom. (Explanation next page).

malloc **man page**

man malloc

`calloc and realloc`

An Interesting Stack Example

```
int *makeArray() {
    int answer[5];
    return answer;
}

void setTo(int *array, int length, int value) {
    for(int i=0; i<length; i+=1)
        array[i] = value;
}

int main(int argc, const char *argv[]) {
    int *a1 = makeArray();
    setTo(a1, 5, -2);
    return 0;
}
```

int * answer = malloc(5 * sizeof(int));

int * answer = (int *) malloc(5 * sizeof(int));
↳ Better! But if you don't have it, it's fine.

because malloc returns a void *. it will automatically cast to int *.

free(a1);

7. [6 points] True/False questions. For each of the statements below, fill in the **T** circle *completely* if you think the statement is True. If you think it's False, fill in the **F** circle *completely*.

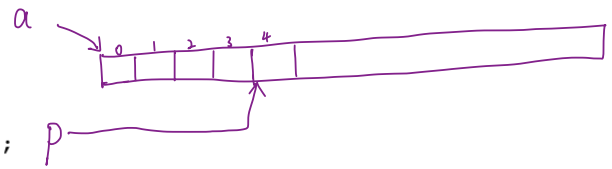
(T) An error will occur in the **type-checking** stage of compilation when compiling our code if we call the function `csoprintf()` which does not exist. linking time

(F) If we accidentally wrote Java instead of C when initializing an array, an error would occur in the **parsing** stage of compilation: `int[100] arr = new arr[100];`

All grammar related things → parsing

For the next two questions, consider the following C code snippet.

```
1  void addressfun() {
2      int a[10];
3      int *p = a + 4;
4      __________ = 42;
5      size_t asize = sizeof(a);
6      return;
7  }
```



8. [6 points] Which of the expressions below could be used in the blank on line 4 to set the 5th element in `a` (i.e., `a[4]`) to 42? *Fill in the circle completely for all that apply.*

- (●) `*(a + 4)`
- (●) `p[0]`
- (●) `*p`
- ( ) `*(a + 16)`
- ( ) `*a + 16`
- ( ) `p[3]`

9. [4 points] What value is stored in `asize` on line 5?

Answer
40

4x10

It XORs together:
① all the numbers that should appear, from 0 to n.
② all the numbers that actually apper in the array.
Since every number except the missing one appears once in both groups, those matching values cancel out under XOR. The only value left at the end is the missing number.

11. [18 points] Write a C function named `missingNumber` that takes two parameters (in order): an array of integers named `nums` given as a pointer, and an integer `n`; it should return an integer.

`nums` points to a valid array of length `n`, which contains `n` distinct integers selected from the range `0` through `n`, inclusive, in random order. Exactly one integer in that range is missing. Your function should find and return the missing integer. If `n = 0`, return `-1`.

Do not allocate another array or modify the contents of `nums`.

For example: given `nums = {3,0,1}` and `n = 3`, return 2; given `nums = {1,4,3,2}` and `n = 4`, return 0; given `nums = {}` and `n = 0`, return `-1`.

method 1:

```
int missingNumber(int *nums, int n){
    for(int x=0; x<=n; x++){
        int found=0;
        for(int i=0; i<n; i++){
            if(nums[i]==x){
                found=1;
            }
        if(!found){ return x; }
        }
    }
    return -1;
}
```

method 2:

```
int missingNumber(int *nums, int n){
    int expected = n*(n+1)/2;
    int actual=0;
    for(int i=0; i<n; i++){
        actual+=nums[i];
    }
    return expected-actual;
}
```

method 3:

```
int missingNumber(int *nums, int n){
    int x=n;
    for(int i=0; i<n; i++){
        x^=i;
        x^=nums[i];
    }
    return x;
}
```

why? $a \wedge a = 0$
$a \wedge 0 = 1$

strcat **example**

```c
#include <string.h>
#include <stdio.h>

int main() {
    char buffer[100];
    const char *s1 = "This is a test";
    const char *s2 = " of string.h";
    buffer[0] = '\0';
    strcat(buffer, s1);
    strcat(buffer, s2);
    puts(buffer);
    return 0;
}
```

Issues:

① May Segmentation fault if s1 or s2 are long. (buffer overflow).

② If I forgot '\0', strcat doesn't work.

" clang problem.c -fstack-protector "

it will show " stack smashing detected " (It gives more error information, some systems do this by default)

```c
#include <stdio.h>

int main() {
    printf("23534624754765☺ç–½8");
    printf("x   x  \n\t");
    printf("\n------\n");
    int x = -2130;
    printf("A number: %d <=- like that\n", x);
    printf("A number: %o <=- like that\n", x);
    printf("A number: %u <=- like that\n", x);
    printf("A number: %x <=- like that\n", x);
    printf("A number: %X <=- like that\n", x);
    printf("%d + %d = %d (%s)\n", 2, 3, 2+3, "yay", 34);
    return 0;
}
```

printf prints exactly the bytes you give it
It does not automatically add a newline (unlike puts)

So we need to add '\n' to print a new line.

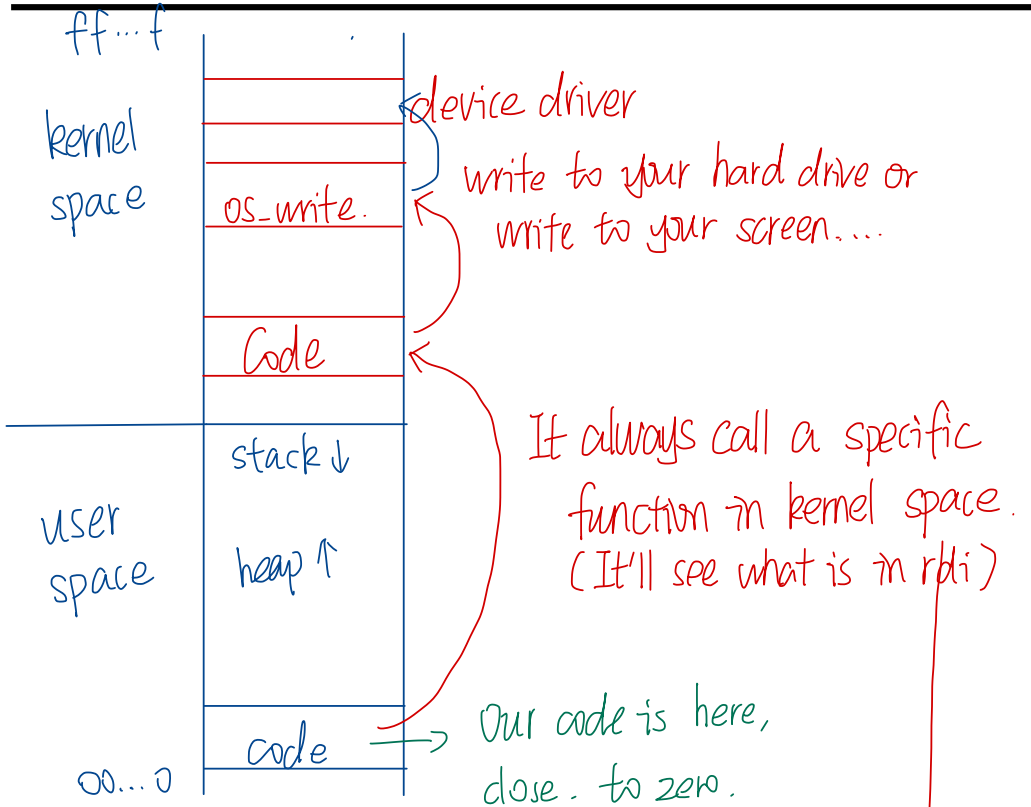
Conversion specifiers: a percentage sign, followed by what type of the thing is.

kernel is allowed to call these special assembly instructions that do things like write to the hard drive, write to the terminal, open/close the files, that kind of thing.

Syscalls

How do I get the kernel to do things?

how do we, as user programs, get the kernel to do things for us?



write.

argument checking

syscall → a special function call.

return value checking

return.

→ It always call the exactly same function

But we are not allowed to read/write in the kernel space. Syscall is allowed to do this. When I run this instruction, it becomes the operation system.

```c
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>

const char *findVowel(const char *word) {
    const char *ans = strpbrk(word, "aeiouAEIOU");
    if (ans != NULL) return ans;
    return word;
}

void showPig(const char *word) {
    const char *vowel = findVowel(word);
    printf("%s", vowel); // ig
    fwrite(word, sizeof(char), vowel - word, stdout);// p
    printf("%s", "ay"); // ay
}

int main(int argc, const char *argv[]) {
    char buffer[1500]; // watch buffer overflow attack!
    int index = 0;
    while (read(0, buffer+index, 1) == 1) {
        if(isalpha(buffer[index])) {
            index += 1;
        } else {
            char keepme = buffer[index];
            if (index > 0) {
                buffer[index] = '\0';
                showPig(buffer);
            }
            fwrite(&keepme, sizeof(char), 1, stdout);
            index = 0;
        }
    }
}
```

Function Pointers

```
void apply(double (*f)(double), double *l, unsigned n) {
  for(int i=0; i<n; i+=1)
    l[i] = f(l[i]);
}
```

`double (*f)(double)` means:

- `*f` – `f` is a pointer
- `(double)` – with a single `double` argument
- `double` – that returns a `double`
- `( )` – to make it parse as `*f` instead of `double *`