

Week 1



Question 9.

int x; // variable at address 1000 with initial value 0.
int *p; // variable at address 2000 with initial value 0.

statement	x value	p value	x address	p address
initial. -	0	0	1000	2000
a. p = &x;	0	1000	1000	2000
b. x = 5;	5	1000	1000	2000
c. *p = 3;	3	1000	1000	2000
d. x = (int) p;	1000	1000	1000	2000
e. x = (int) &p;	2000	1000	1000	2000
f. p = NULL;	2000	NULL	1000	2000
g. *p = 1;		fails	because	p is NULL.

Question 6.

when to use * and/or malloc for structs?

struct node a; use •

- declared on the stack
- don't need malloc.

- only use within a function.

struct node *b; use →

- declared on the heap
- need malloc

good for passing between functions

a pointer is a variable that points to the address of a variable.

int *var;

int var;

int * var

struct

what does sizeof do?

sizeof gets the # of bytes (of variable eg 'a' or type eg 'int')

what does malloc do?

allocates memory of given size @ the address or CHS.

c = malloc (8) allocates 8 bytes @ c's address
↑ variable to allocate for ↑ # of bytes

492010

← :

$$f(5) = f(4) + f(3)$$
$$\begin{array}{ccccc} & / & \backslash & / & \backslash \\ f(3) & & f(2) & f(1) & f(0) \end{array}$$

r(0)

↓

r(1)

↓

r(2)

↓

3

4

↓

10

Week 2

1. When should the types in `stdint.h` be used?

⇒ what is the type `uint8_t`? ^{unsigned}
`00000000` ^{8 bits}
 how is it different to `int8_t`? ^(positive & negative)
 → not unsigned

2. How are the bases [decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2)] denoted in C?

hexadecimal: starts with `0x`

octal: starts with `0`

decimal: everything else

binary: ⇒ `0b` → not standard → pls don't use it -

	decimal	binary	octal ^{= 8 digits}	hexadecimal
a.	1	0 0 0 0 0 0 0 1	0 0 1	0 1
b.	8	→ 0 0 1 0 0 0 oct: $2^3 + 2^2 + 2^0$ hex: 0 0 1 0 0 0 $2^3 + 2^2 + 2^0 = 8$	0 1 0	0 8
c.	10	0 0 1 0 1 0 $2^3 + 2^2 + 2^0$	0 1 2	0 A
d.	15	0 0 0 1 1 1 1	0 1 7	0 F
e.	16			
f.	100			
g.	127			
h.	200			

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0 0 0 0	0	8	1 0 0 0	8
1	0 0 0 1	1	9	1 0 0 1	9
2	0 0 1 0	2	10	1 0 1 0	A
3	0 0 1 1	3	11	1 0 1 1	B
4	0 1 0 0	4	12	1 1 0 0	C
5	0 1 0 1	5	13	1 1 0 1	D
6	0 1 1 0	6	14	1 1 1 0	E
7	0 1 1 1	7	15	1 1 1 1	F

convert 8 to binary

8			
(÷2)	4	r	0
	2	r	0
	1	r	0
	0	r	1

↑
binary digit

$$\Rightarrow 8_{10} = 1000_2$$

↑
base
10
(decimal)
↑
base
2
(binary)

(÷2)	10		
	5	r	0
	2	r	1
	1	r	0
	0	r	1

↑
1010₂

(÷2)	15		
	7	r	1
	3	r	1
	1	r	1
	0	r	1

$$\begin{array}{r} 01010 \\ \& 11100 \\ \hline 01000 \end{array}$$

wherever we want to set a bit to 1 in original value

(now)

turned to 0

original & mask

A	B	^	& ~
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	0

bad for setting to 0

same as original when B=0

set to 0 when B=1

3. Bitwise Operations

True = 1
False = 0

!101010 → True = 00000
~101010 = 010101
not same as!

A	B	A B OR	A & B AND &	A ^ B XOR ^	A	NOT ~
0	0	0	0	0	0	1
0	1	1	0	1	1	0
1	0	1	0	1		
1	1	1	1	0		

a) $0x5555 | 0xAAAA = 0xFFFF$ g) $0x5555 \& (0xAAAA \ll 1)$

$$\begin{array}{cccc} 0101 & 0101 & 0101 & 0101 \\ | & 1010 & 1010 & 1010 & 1010 \\ \hline 1111 & 1111 & 1111 & 1111 \end{array}$$

$$\begin{array}{cccc} 1010 & 1010 & 1010 & 1010 \ll 1 \\ = & 0101 & 0101 & 0101 & 0100 \end{array}$$

b) $0x5555 \& 0xAAAA = 0x0000$

$$\begin{array}{cccc} 0101 & 0101 & 0101 & 0101 \\ \& & 1010 & 1010 & 1010 & 1010 \\ \hline & 0000 & 0000 & 0000 \end{array}$$

$$\begin{array}{cccc} \Rightarrow & 0101 & 0101 & 0101 & 0101 \\ \& & 0101 & 0101 & 0101 & 0100 \\ \hline & 0101 & 0101 & 0101 & 0100 = 0x5554 \end{array}$$

c) $0x5555 \wedge 0xAAAA = 0xFFFF$

$$\begin{array}{cccc} 0101 & 0101 & 0101 & 0101 \\ \wedge & 1010 & 1010 & 1010 & 1010 \\ \hline & 1111 & 1111 & 1111 \end{array}$$

h) $0xAAAA | 0x0001$

$$\begin{array}{cccc} 1010 & 1010 & 1010 & 1010 \\ | & 0000 & 0000 & 0001 \\ \hline 1010 & 1010 & 1010 & 1011 \end{array}$$

d) $0x5555 \& \sim 0xAAAA$

$$\begin{array}{cccc} 0101 & 0101 & 0101 & 0101 \\ \& & 0101 & 0101 \\ \hline & 0101 & 0101 & 0101 = 0x5555 \end{array}$$

i) $0xAAAA \& \sim 0x0001$

$$\begin{array}{cccc} 1010 & 1010 & 1010 & 1010 \\ \& & 1111 & 1111 & 1111 & 1110 \\ \hline 1010 & 1010 & 1010 & 1110 \end{array}$$

e) $0x0001 \ll 6$

$$\begin{array}{cccc} 0000 & 0000 & 0000 & 0001 \\ \ll 6 & 0000 & 0000 & 0100 & 0000 \end{array}$$

f) $0x5555 \gg 4$

$$\begin{array}{cccc} 0101 & 0101 & 0101 & 0101 \\ \gg 4 & 0000 & 0101 & 0101 & 0101 \end{array}$$

Given a variable X...

Copy / extract the value of a specific bit: $X \& \text{mask}$

Set a specific bit to 1: $X | \text{mask}$

Set a specific bit to 0: $X \& \sim \text{mask}$

to set a specific bit(s) to 1.

original value \Rightarrow 0001 0101
 0011 0000 \Rightarrow mask
 try and & try or
 use to set digits to 1.
 0011 0101
 set to 1 like we wanted
 still same
 0 \Rightarrow not what we want

try copying specific bits from a value:

don't copy \Rightarrow 0101 0101 \Rightarrow original
 0110 0000 \Rightarrow mask
 & 0100 0000
 some as the original

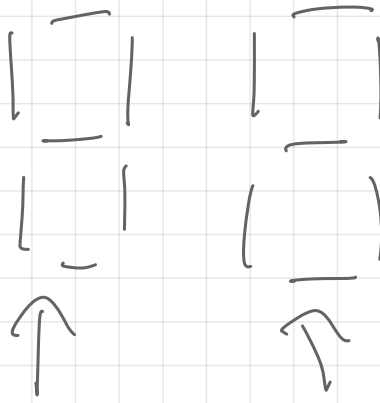
set specific bits to 0.

0101 1010 \Rightarrow original
 1001 1111 \Rightarrow mask
 & 0001 1010
 0101 1010 \Rightarrow original
 0110 0000 \Rightarrow mask
 $\sim \& (\sim \text{mask} =$
 1001 1111
 & 0001 1010

BCD



digits
are
separately
converted to binary

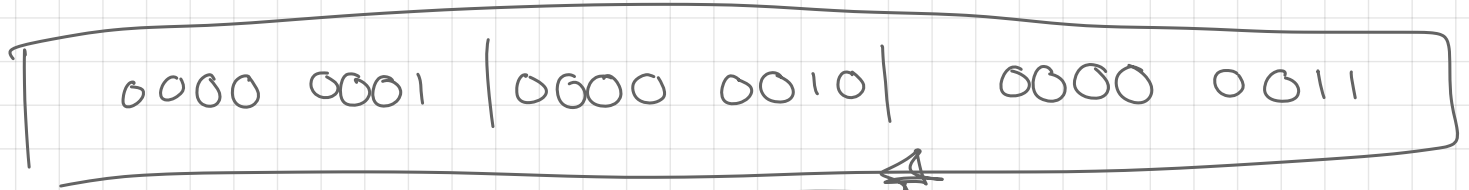


$$1_{10} : 1_2 \Rightarrow 0000 \ 0001_2$$

$$2_{10} : 10_2 \Rightarrow 0000 \ 0010_2$$

$$3_{10} : 11_2 \Rightarrow 0000 \ 0011_2$$

(8bits
 \Rightarrow uncompressed
BCD



vs normal binary



compressed BCD

\Rightarrow 4 bits instead of 8.

$$258 = \underbrace{0000 \ 0001}_1 \mid \underbrace{0000 \ 0010}_2$$

uint8_t a = 0x05 (0000 0101₂)
uint8_t b = 0xA (0000 1010₂)

0000 0101
0000 1010

(or) | 0000 1111

(and) & 0000 0000

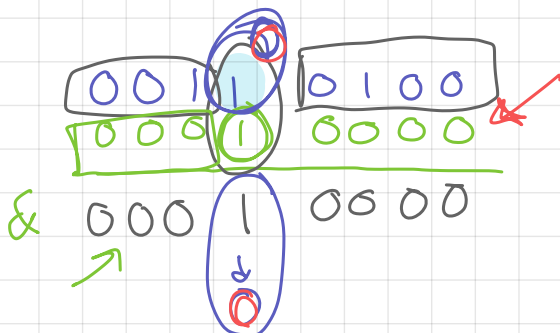
(xor) ^ 0000 1111



⇒ 1111 1100

0011 1111 >> 2

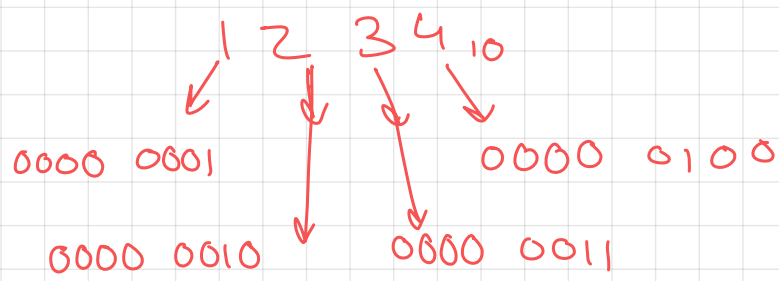
uint ...
(unsigned int) ...



0011 0100
0001 0100
| 0011 0100

0011 0100
0001 0000
~ 1111 0111
& 0011 0100

mask = 1 << i;
0000 0001



1234

0000 0001 | 0000 0010 | 0000 0011 | 0000 0100

258's

0000 0001 | 0000 0010

↓ ↓

$1 \times 10^1 + 2 \times 10^0$

Week 3

!! weekly tests start this week !!

Negative Values

How are signed values represented?

X X X X X X X X X X X X X X

1 = negative 0 = positive if -ve represent using 2's complement

* for positive values:
use normal bin representation
signed bit is 0

* for negative values:
set signed bit to 1
use 2's complement.

to determine a number's representation in 2's complement:

- get normal bin. representation
- invert all bits
- add 1

Eg. 5_{10} :

0000 0000 0000 0101

vs -5_{10} :

invert: 1111 1111 1111 1010
add 1: 1111 1111 1111 1011

100_{10} : invert

0000 0000 0110 0100

vs -100_{10} :

cheat 1111 1111 1001 1100
invert 1111 1111 1001 1011
+1 1111 1111 1001 1100

$11_{10} = 1011_2$

2's comp = 0100 + 1

$-11_{10} = 0101$

cheaty method

$11_{10} = 1011_2$

invert everything to left of the first 1
 $= 0101$

$10_{10} = 1010_2$
 $2'sc = 0110$

$10_{10} = 1010$
 $= 0101$
 $+ 1$
 $0110 = -10_{10}$

convert 16-bit representation to the corresponding decimal value:

• $0x0013 = 0000 0000 0001 0011 = +19$
= positive

• $0xffff = 1111 1111 1111 1111 = -1$
= negative

\therefore need to use 2's complement
 $= 0000 0000 0000 0001 = 1$

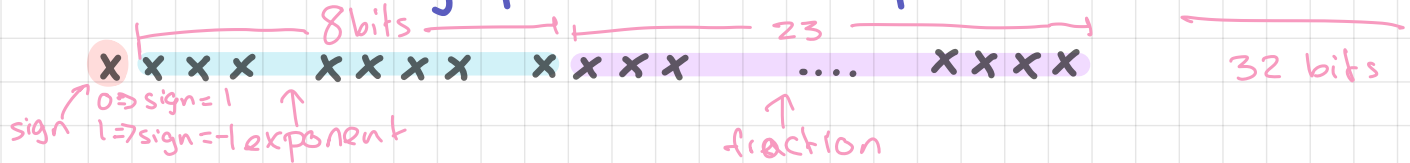
2's complement
undoes itself

ie: $2'sc(2'sc(a)) = a$

ie $-100 = 1111 1111 1001 1100$
normal 0000 0000 0110 0100
method 0000 0000 0110 0100

Floating Point Values

How are floating point values represented? (IEEE 754)



from this we can calculate the value with the formula:

$$\text{sign} \times (1 + \text{frac}) \times 2^{\text{exp} - 127}$$

note: we don't use two's complement here.

Eg: Convert the following to decimal numbers

f) 0 10000000 | 0110...0

$$\begin{array}{c} 10^{-1} \ 10^{-2} \ 10^{-3} \ 10^{-4} \\ 0. \ 1 \ 2 \ 3 \ 4 \\ 0.1 = 1 \times 10^{-1} \\ 0.02 = 2 \times 10^{-2} \\ \dots \end{array}$$

sign: positive $\therefore = 1$

exp: $2^7 = 128$

frac: $2^{-2} + 2^{-3} = 0.375$

value = $1 \times (1 + 0.375) \times 2^{128 - 127} = 2.75$

c) 0 0111111 | 100...0

sign: $= 1$ (+ve)

exp: $= 127$

frac: $= 2^{-1} = 0.5$

value = $1 \times (1 + 0.5) \times 2^{127 - 127} = 1.5$

don't follow formula

a) 0 00000000 | 000...0

sign: 1

exp: 0

frac: 0

value = $1 \times (1 + 0) \times 2^{0 - 127} = 2^{-127} \approx 0$

'denormal' $X \neq Y$ where $X = 0$

b) 1 00000000 | 000...0

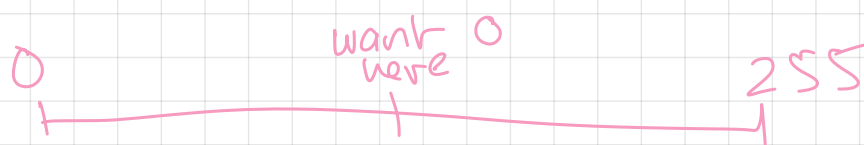
sign: -1

exp: 0

frac: 0

value = $-1 \times (1 + 0) \times 2^{0 - 127} = -2^{-127} = -0$

$$255 = 1111 \ 1111$$



Eg 2: Convert the following to IEEE 754 encoded bit strings:

a) $2.5 \div 2^1 = 1.25 \Rightarrow 2.5 = 1.25 \times 2^1$
 sign: +ve $\therefore 0$
 exp: $\text{exp} - 127 = 1 \Rightarrow \text{exp} = 1 + 127 = 128$
 frac: $010 \dots 0$
 bits = $0 \ 1000 \ 0000 \ 010 \dots 0$

first express the number in the form:
 $(1 + \text{frac}) \times 2^n$

b) $0.375 \times 2^2 = 1.5 \Rightarrow 0.375 = \frac{1.5}{2^2} = (1 + 0.5) \times 2^{-2}$
 sign: 0
 exp: $\text{exp} - 127 = -2 \Rightarrow \text{exp} = -2 + 127 = 125_{10} = 01111101_2$
 frac: $100 \dots 0$
 bits = $0 \ 0111 \ 1101 \ 100 \dots 0$

how to convert fract. to bin?

$(\times 2) 0.25$
 \downarrow
 0.5
 \downarrow
 1.0

c) $27.0 \div 2^4 = 1.6875 \Rightarrow 27.0 = (1 + 0.6875) \times 2^4$
 sign: 0
 exp: $4 = \text{exp} - 127 = 13_{10} = 1000 \ 0011_2$
 frac: $0.6875_{10} = 101100 \dots 0$
 bits = $0 \ 1000 \ 0011 \ 1011 \ 0 \dots 0$

0.5
 \downarrow
 1.0

0.6875
 \downarrow
 1.375
 \downarrow
 0.75
 \downarrow
 1.5
 \downarrow
 1.0

d) 100
 sign:
 exp:
 frac:
 bits =

steps:

- start w/ fraction component.
- multiply by 2 until either we get 0 or run out of space for digits
- take the digit on the LHS as the next bin. digit.
- take the value on RHS and repeat process

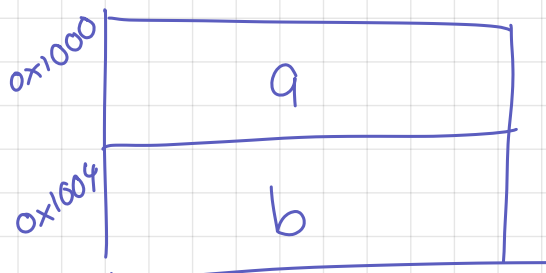
Structs vs Unions

struct {
 assume 4 bytes
 int a;
 float b;
} x1;

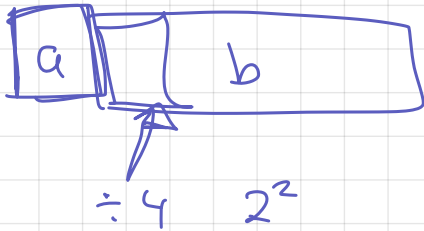
$\&x1 = 0x1000$
 $\&(x1.a) = 0x1000$
 $\&(x1.b) = 0x1004$

union {
 int a;
 float b;
} x2;

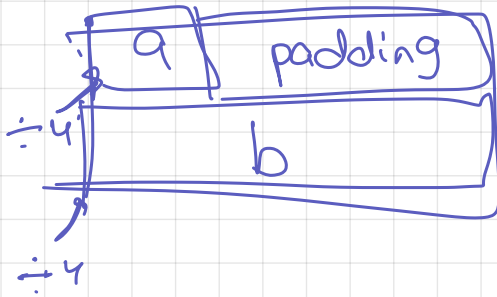
$\&x2 = 0x2000$
 $\&(x2.a) = 0x2000$
 $\&(x2.b) = 0x200$



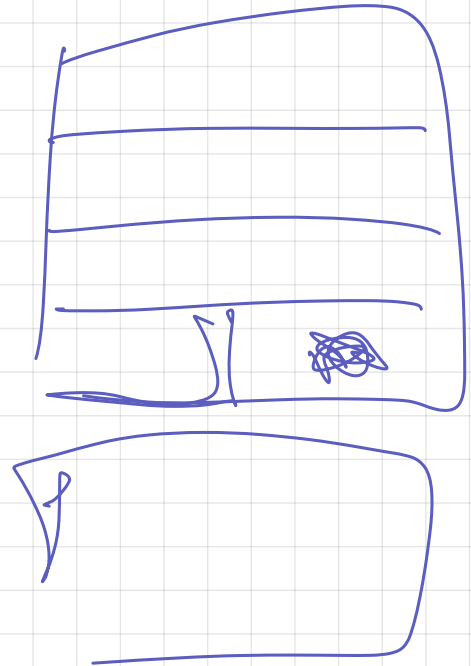
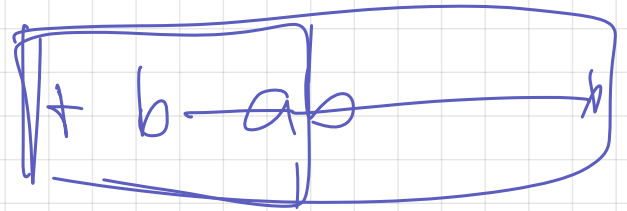
$x2.a = 10$
 $x2.b = 1(1 + 2^{-10} + 2^{-22}) \times 2^{0-127}$
 $= 1(1. \dots) 2^{-127}$



what actually happens



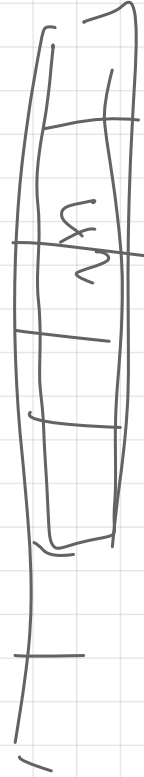
$$4 + 4 = 8$$



Week 5

3. $0_{\text{x}} 10000020$

4 bytes			
20	21	22	23
20	—	4 2	—
24	?	?	?
28	a	b	c d
32	^{2C} e	²⁰ 10	^{2E} X (align) ^{2F} X
30	0000 00	← 1 →	← 2 →
34	0100	←	1 →
38		←	2 →
3C		←	3 →
40		←	4 →
...	??		
...			
...			



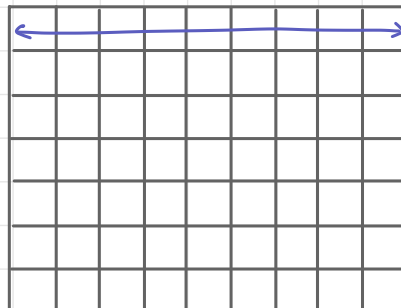
Arrays

eg array of ints
[0] [1] [2] . . .



calculate index address:

(1d) $\text{address} = \text{starting address of array} + \text{offset}$

$$\text{offset} = \text{index} \times \text{size of elements} \checkmark$$
$$\text{Id array-address} = \text{label} + \text{index} \times \text{sizeof}(\text{element})$$
$$2d \text{ array-address} = \text{label} + \text{row} \times \text{NUMCOLS} \times \text{sizeof}(\text{element}) + \text{index} \times \text{sizeof}(\text{element})$$


(week 7 content) from wk 5 !!

Functions

main
caller
calls another
function

main
callee
is being
called

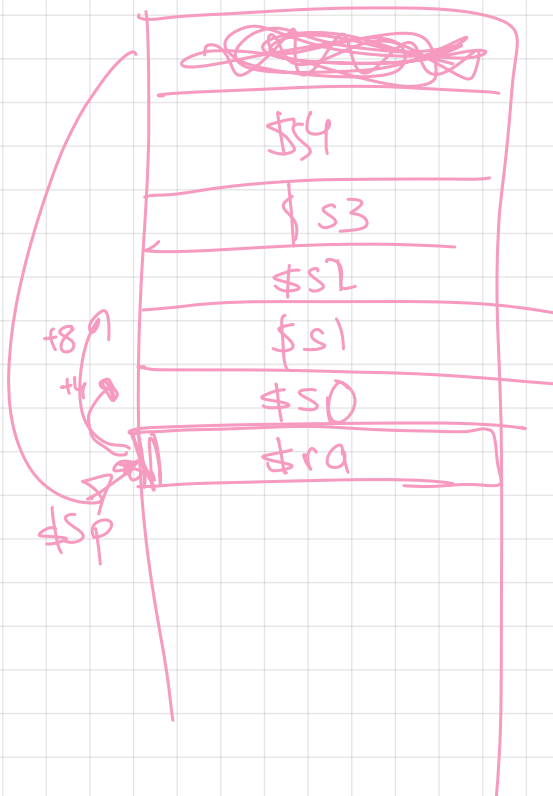
to "call another fn"
→ use jal address
→ changes \$ra to address of next instr...

to give arguments
use \$a registers
in order!

can assume is
\$s aren't changed
by callee.

can assume \$t
registers are free
to use (don't need
to save them)

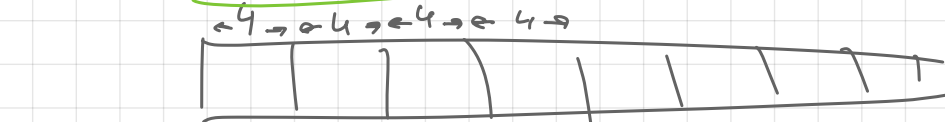
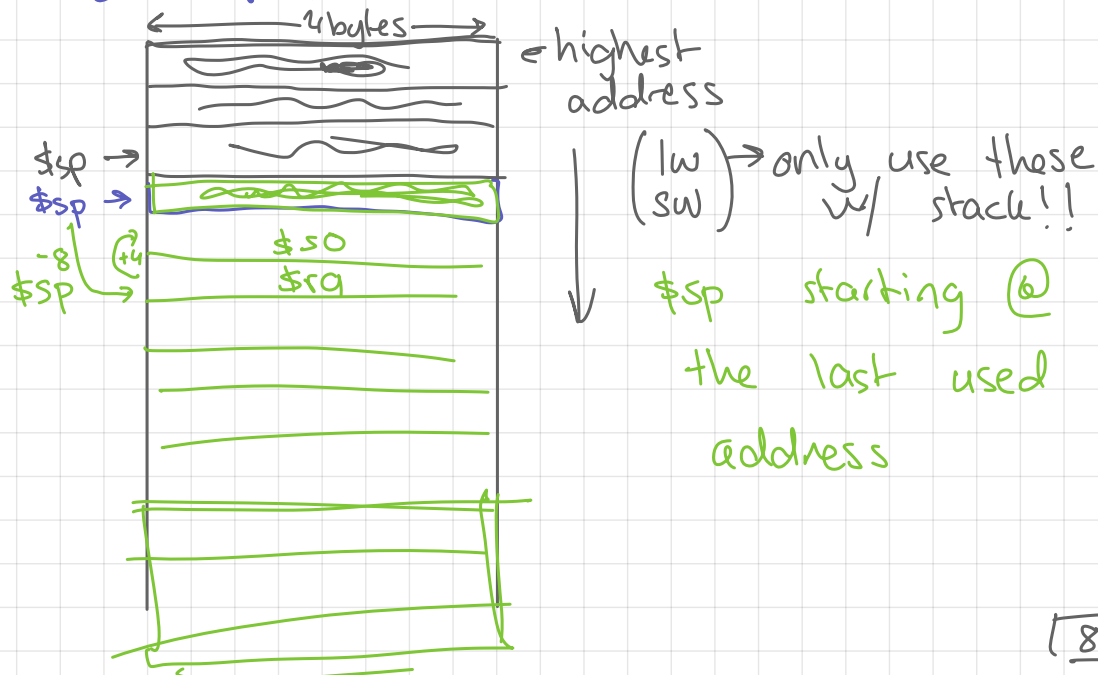
have to save the
\$s registers if
using them.
→ same for \$ra



Actual Week 7

	Rights	Responsibilities
callee function being called	free to use: (change them • \$t as much • \$a as want) arguments: in \$a regs (extras go on stack)	needs to save: • \$s regs /restore • \$ra • \$fp return value in: \$v0 (and v1 if 4 < value ≤ 8 bytes)
caller function doing the calling	\$ra, \$s aren't changed by callee	puts arguments in \$a regs if it ^{really} wants to use \$t before it after (w/ same value) has to save it itself.

The stack



$$4 \times 10 \times 5$$

$$= 40 \times 5$$
$$= 200 \text{ bytes}$$

$$4 * 20 = 80 \text{ bytes}$$

