3/31/25 L1: UNSIGNED BINARY, BINARY, C BASICS

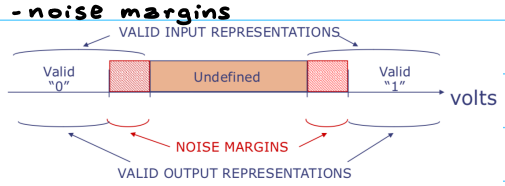
SCHEDULE:
M: lecture
T: rec, due @ night
W: LAB (DON'T BE LATE)
S: online ex, lab/post lab checkoffs due
FINAL: May 7, 7:30-9:30

WHAT IS INFO?
- anything providing answer to question of some kind
- BIT: one Y/N amount of info (represented as 1/0)
  - more info = more bits
  - 8 bits? $2^n = m \rightarrow 2^8 = 256$
- BYTE: grouping of 8 bits. 00000000 through 11111111
  Ex: 2341 options? $\log_2(2341) = 11.19$ bits

ANALOG VS. DIGITAL:

ANALOG: many voltage value
- Ex: 0→10V, 0.01 step - single V can be $\log_2(1000) \approx 9.96$ bits of info
- susceptible to noise (ex: 2.03 could be 1.98 + 0.03V)

DIGITAL: limited voltage values
- Ex: 0V or 10V - $\log_2(2) = 1$ bit of info
- can deal w/ noise (above certain amt = 10V, below = 0V)
- tighter constraints
- noise margins



EX: paul revere
- full - by land
- half - by sea
- off - not come

- land
- sea ← easier to know w/ fog!
- not come

SCALING: encoding + ints
- each bit assigned a weight

$$V = \sum_{i=0}^{N-1} 2^i b_i$$

- smallest: 0
- largest: $2^N - 1$

BINARY: ADD & SUBTRACT
- just like base 10

Base 2

```
 111  ← carry
 1110
+ 111
10101
```

```
-1-1-1
 1110
- 111
 0111
```

-1-1
1110
111
111

MOD:

OVERFLOW:
- computers have fixed # bits, but may want to do more rigerous calculations

```
 111
 1110
+0111
10101
```

Overflow, actual result is $21 \bmod 2^4 = 5$

↳ only returns this - can calculate from here

Ex: encode 21 using 6 bits

START: 21
→ does 2^5 (32) fit 21? x → 0
2^4 (16) → ✓ → 21 - 16 = 5
2^3 (8) fit 5? → x → 0
2^2 (4) fit 5? → ✓ → 5 - 4 = 1
2^1 fit 1? x
2^0 ? ✓

ANSWER: 0b010101 (binary)

0 16 0 4 0 1

C: THE LANGUAGE

- more unsafe
- older & more low-level
- K&R C book

PYTHON	C ← "closer to the metal"
· interpreted (slower)	· compiler
· no type declaration	· faster execution (typically)
· no pointers	· type declaration
· auto memory manag	· pointers
	· manual mem. manage.
	- have to tell explicitly

CODE:

```
//comments are with "//"
//This is our "entry point" function called app_main...start here
void app_main(){ //gotta use curly braces (and ; at end of most lines!)
  int x; //declare a variable of type int without defining it
  x = 11; //assign 11 to
  int y = 15; //declare and defie a variable of type int
  int z = x+y;
  printf("the value of x is %d\n", x); //print using C-string format
  printf("the value of y is %d\n", y);
  printf("the value of z is %d\n", z);
  // what did this "make" in memory?
}
```

At the end of our code, our computer memory looked like this:

Address:	Value:	
0x3fc93f48:	0x00000000	
0x3fc93f4c:	0x420000e4	
0x3fc93f50:	0x00000000	
0x3fc93f54:	0x0000001a	← z → 1a = 16+10 = 26
0x3fc93f58:	0x0000000f	← y (f=15)
0x3fc93f5c:	0x0000000b	← var x in memory (11)
0x3fc93f60:	0x00000000	
0x3fc93f64:	0x00000000	
0x3fc93f68:	0x00000000	
0x3fc93f6c:	0x4201652c	

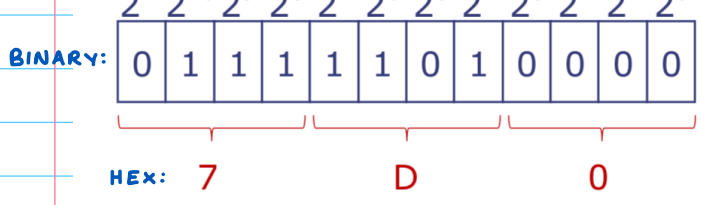
SYNTAX:

- ; @ end of all lines
- spacing doesn't matter
- { delineates code chunks

MEMORY: large chunk of electrical storage

- stored digitally (0 & 1s)
- low to high
- 0b @ start to indicate binary (base 2)
- **HEXADECIMAL:** 0x, 4 binary → 1 hex, base 16

git.



0b011111010000 = 0x7D0

*Preface hexadecimal numbers with **0x***

- underlying bits are still the same!

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - A
0011 - 3	1011 - B
0100 - 4	1100 - C
0101 - 5	1101 - D
0110 - 6	1110 - E
0111 - 7	1111 - F

ddress: Value:

"0"	Byte 3	Byte 2	Byte 1	Byte 0
"4"	Byte 7	Byte 6	Byte 5	Byte 4
"8"	Byte 11	Byte 10	Byte 9	Byte 8
"12"	Byte 15	Byte 14	Byte 13	Byte 12
"16"	Byte 19	Byte 18	Byte 17	Byte 16
"..."	*ad nauseum...*			

ORGANIZATION:

- smallest accessible unit of mem = byte (8 bits)
- our system: 32-bit system = 4 byte words
- 4 byte words w/ addresses
  - each word → 32 bits data
  - each addy
- chars only require 1 byte in memory
- float: 4 bytes
- int: 4 bytes

} TYPES

PRINT:

- specify width: printf("%3d, %6d\n", a, b); ← 3 cols & 6 cols of space
- %f prints float, %d = decimal, %c = char

```
int app_main ( ){
  printf ( "%5d\n", 123 ); /* Prints " 123" */
  printf ( "%*d\n", 5, 123 ); /* Prints " 123" */
  printf ( "%+05d\n", 123 ); /* Prints "+0123" */
  printf ( "%x\n", 123); /* Prints "7b" */
  printf ( "%#x\n", 123); /* Prints "0x7b" */
  printf ( "%#X\n", 123); /* Prints "0X7B" */
  printf ( "%-10.2f\n", 12.3 ); /* Prints "12.30" */
  printf ( "%10.2f\n", 12.3); /* Prints " 12.30" */
  printf ( "%lu\n", 123); /* Prints "123" */
  printf ( "%s\n", "Testing"); /* Prints "Testing" */
  printf ( "%c\n", 'A'); /* Prints "A" */
}
```

- careful not to represent float as int!

IMPORTS:

#include <stdint.h> ← more defs

#include <stdio.h> ← printf

C

- loops
- conditionals
- logical ops
  - → &&, ||, !
  - → any nonzero value is considered T (1)
  - → ^ = XOR
  - → ~ = NOT
- bitwise ops
  - shift right by i = divide by 2^i
  - shift left by i = multiply by 2^i

C:

```
if (a < b) {
    statement(s);
} else if (a == b) {
    statement(s);
} else {
    statement(s);
}
```

```
while (n > 1) { statement(s); }

for ( i = 1; i < 10; i++) { statement(s);
}
```

```
x << y; //shift x left by y bits
x >> y; //shift x right by y bits
x & y;  //bitwise and x with y
x | y;  //bitwise or x with y
~x;     //bitwise invert x
x ^ y;  //bitwise xor x with y
```

Ex: x = 0b00000101 shift by 3 (x<<3) left

```
000101   5
001010   10
010100   20
101000   40
```

Ex:

x bitwise AND y x & y	x bitwise OR y x | y	x bitwise XOR y x ^ y	bitwise NOT x ~x
00101	00101	00101	00101
00110	00110	00110	
00100	00111	00011	11010

```
int summ(int a, int b) {
    return a + b;
}
int x, y, z;
x = 1;
y = 2;
z = summ(x, y);
```

· functions

· pointer → data type that "points" to other data
  - stores reference mem. addy
  - can recall if needed (dereference)
  - dangerous but freeing

Ex:

```
void app_main(){
  int x = 5; //make int called x
  int * ptr_to_x; //make int pointer called ptr_to_x
  ptr_to_x = &x; //give ptr_to_x the address of x
  printf("x is %d\n" , x);
  printf("ptr_to_x's value is %d\n" ,ptr_to_x);
  printf("The value ptr_to_x points at is %d\n", *ptr_to_x);
}
```

- This Prints:

```
x is 5
ptr_to_x's value is 1070153564
The value ptr_to_x points at is 5
```

The value of "x"

The value of "ptr_to_x" Which is the address of x

The value contained in the memory spot that ptr_x points to (which should be x)

Address:	Value:
0x3fc93f58:	0x42016554
0x3fc93f5c:	0x00000005 ← x
0x3fc93f60:	0x3fc93f5c ← ptr_to_x
0x3fc93f64:	0x00000000
0x3fc93f68:	0x00000000
0x3fc93f6c:	0x42016554

*** in C:**

· multiply `int q = a * b;`

· declare pointer type `int * z;` ← declare int pointer named z

· dereferencing `* z = a;` ← set val of z pts to int a
`b = * z;` ← set int b to val z pts to

& in C:

· and (logic) `int q = a & b;`

· reference to / addy `int * z = & a;` ← set pointer z's val to a's addy

* IMPORTANT: * and & are opposites - can cancel out

```
int x = 5; //create x and set to 5
x = 6; //set x to 6
*(&x) = 6; //same thing as line above
```

&x == "the address of x"

**(...) == "the value at address ..."*

PURPOSE OF POINTERS:

1) point to a variable.

- can alter outside world
  - Can allow you to modify variables in place. For example, this function has no return type, but does have an "output" in the form of the pointer z

```
void add_and_assign(int x, int y, int * z){
  //the value of z is a reference (address)
  //the value at that address (*z) is now to be x+y:
  *z = x+y;
}
```

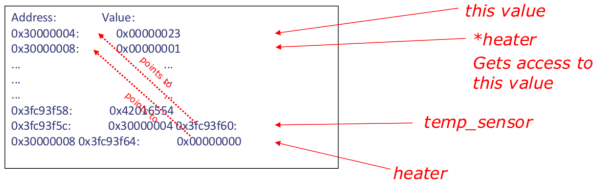
```
void app_main(){
  int a = 1;
  int b = 2;
  int c = 0; //starts at 0
  add_and_assign(a,b,&c);//give int a, b, and ref to c
  //this function call will change variable c!
  printf("Value of c: %d\n",c); //will print 3
}
```

2) refer to memory regions that aren't associated w/ variables

- memory-mapped-input-output
- set/get vals from software → hardware, vv.

MMIO Example...

```
void app_main(){
  int * temp_sensor = 0x30000004; //set pointer to a known address value
  int * heater = 0x30000008; //set pointer to a known address value
  //The two addresses above come from datasheet of processor!
  while(1){//run forever
    //check temperature...
    if (*temp_sensor < 60){ //get value...less than 60?
      *heater = 1; //set value to 1 (let it warm)
    }else{
      *heater = 0; //set value to 0 (let it cool)
    }
  }
}
```



3) point to a group/region of data/variables

- can use pointer for data group & make faster

```
// in C:
int a[6] = {3, 9, 6, 1, 7, 2};
int x;
x = a[1];

int sum(int *a, int len) {
  int s = 0;
  for (int i=0; i<len; i++){
    s += a[i];
  }
  return s;
}
sum(a, 6);
```

pointer to start of array

gth of len array (need to tell)

ARRAYS:

- size of array must be known @ creation
- type of elements must all be same

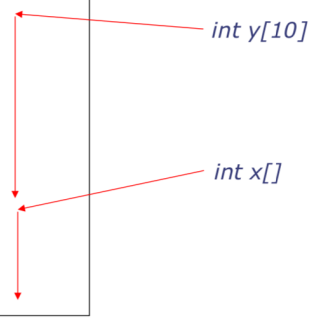
Ex:

```
void app_main(){
 int x[] = {1,2,3,4,5};
 //10 long array of ints accessible through ptr y
 //only four vals specified (rest are unknown)
 int y[10] = {10,11,12,13};
 printf("%d\n", x[1]); //prints 2
 printf("%d\n", *x); //prints 1 same as x[0] or *(x+0)
 printf("%d\n", x); //prints 1070153528 (0x3fc93f38 in hex)
 printf("%d\n", x+1); //prints 1070153532 (0x3fc93f3c in hex) ← 2's memory
 printf("%d\n", *(x+2)); //prints 3...same as x[2]
 printf("%d\n", *(x+5)); //prints undefined...same as x[5]...no idea what's in that memory
 printf("%d\n", *(y+10)); //prints 1...because you bled into where x is and x[0] is 1
 //wait does that mean I could do this???
 y[12]=15; //oh god please throw an error
 printf("%d\n", x[2]); //prints 15...oh no
}
```

changes 3rd element of x!

memory after code:

Address:	Value:
0x3fc93f0c:	0x420001ec
0x3fc93f10:	0x0000000a
0x3fc93f14:	0x0000000b
0x3fc93f18:	0x0000000c
0x3fc93f1c:	0x0000000d
0x3fc93f20:	0x00000000
0x3fc93f24:	0x00000000
0x3fc93f28:	0x00000000
0x3fc93f2c:	0x00000000
0x3fc93f30:	0x00000000
0x3fc93f34:	0x00000000
0x3fc93f38:	0x00000001
0x3fc93f3c:	0x00000002
0x3fc93f40:	0x00000003
0x3fc93f44:	0x00000004
0x3fc93f48:	0x00000005



EXERCISES:

Ex: 0b0110 + 0b0010 = 0b1000

Ex: 0b01100 − 0b00011 = 0b01001 (borrowing 2; ALWAYS borrow the base! ex: 2 in binary, 6 in hex)

Ex: 0b100 * 0b011 = 0b1100 (4 × 3 = 12)

Ex: largest # bits to represent the result of multiplying two 5 bit #s together → 10 BUT WHY?

Ex: 0b111 (7) * 0b100 (4) = 0b011100 (28)

Ex: 0b0011100000 ^ 0b1010101010 = 0b1001101010

Ex: 0b0001010 << 2 = 0b1101000

Ex: 0b1110010 >> 2 = 0b11100

does NOT wrap around.

ends become 0 when bit-shifting

SIGNED → + & − #s

UNSIGNED → + & 0 #s

Ex: 0b0100 && 0b0011 = 1 b/c # = 1 always? 0 = 0?

Ex: !0b10101010 = 1

Ex: (0b0010 − 0b0001 − 0b0001) && 0b0101 = 0! (this = 0!)

& evaluates each digit.

&& evaluates numbers (nonzero)

[4/1/25] - recitation 1

BIT:
- fundamental unit
- 0 or 1

BYTE:
- 8 bits
- $2^8 = 256$ configurations

WORD:
- 2^{32}
- 8 bytes

UNSIGNED (non-neg) BINARY

uint8_t = unsigned int – 8 bits type

each can represent up to $2^N - 1$ numbers $[0, 2^N - 1]$

in binary → prefix 0b

methods

REMAINDER:

	div	remain	
Ex: 13//2	= 6	1	← least significant
6// 2	= 3	0	
3 // 2	= 1	1	
1 // 2	= 0	1	← most significant (left)

→ 13 = 0b1101

POWER OF 2:

Ex: 13 = 8 + 4 + 1

2^3 2^2 2^1 → 1101

BINARY TO DECIMAL:

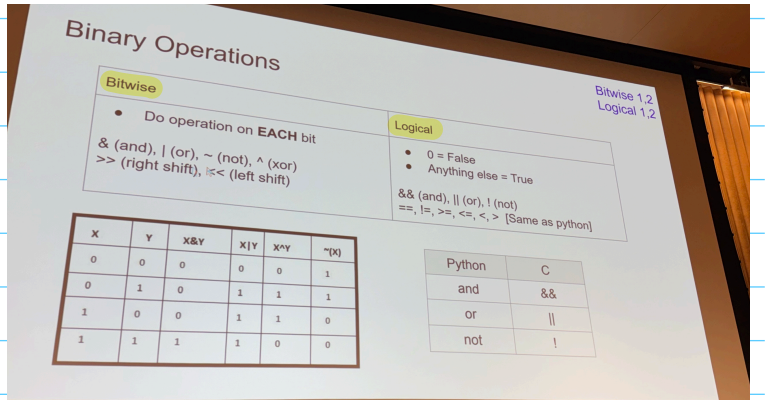
Ex: 0b1001 = $2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 2^0 = 9$

OPERATIONS:

Ex:
```
  11
  0b111
+ 0b001
 1000
```
for 3 bits, result would be 000

Ex:
```
  0b1010
  0b1100
  10110
```

Ex:
```
  *0b1010
   0b1111
    1010
   1010
  1010
 1010
 10010110
```
10010110



BITWISE

Ex: (((0b1001 | 0b1100) & 0b0011) << 3

0b1101 & 0b0011

0b0001 <<3 = 1000

Ex: ((~0b0000_1111) ^ 0b1100_1100) >> 4

0b1111_0000 ^ 0b1100_1100 = 0b0011_1100 >> 4 = 0000_0011

Ex: (0b100 & (0b 000 <= 0b111)) == 0b001

1 (T)

1 (T) - both #s

1 == 1 ✓ TRUE → (1)

Ex: !0b101 || ((0b100 - 0b011 - 0b001) && 0b001)

0b000

0

0

HEXADECIMAL:

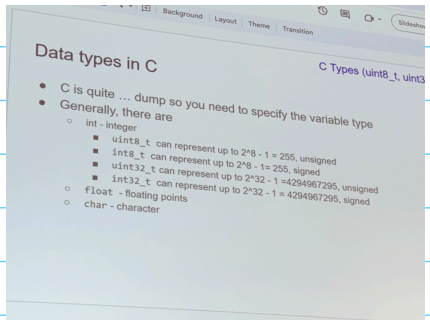
- base 16
- prefix 0x
- 1 digit of hex = 4 digits bin.

review hex

EX: hex of 0b00011100 → 1 C

EX: 0x20 → 2×16 = 32

DATA TYPES:



DIVISION:

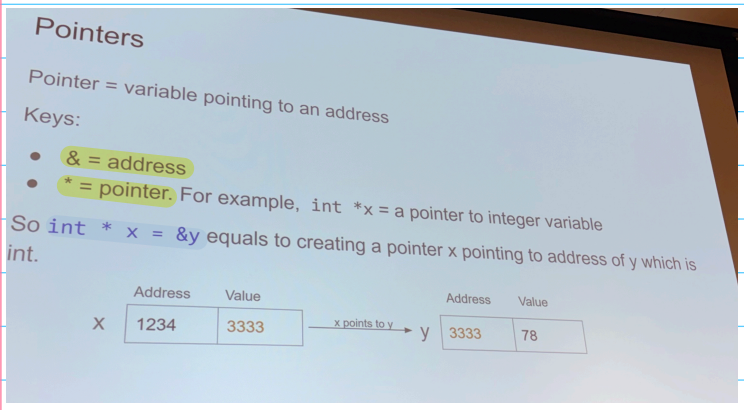
- integer division: int/int
- floating point division: if one or both its operands is float

EX: float x = 9/2 → x = 4.0 (9/2 → 4)

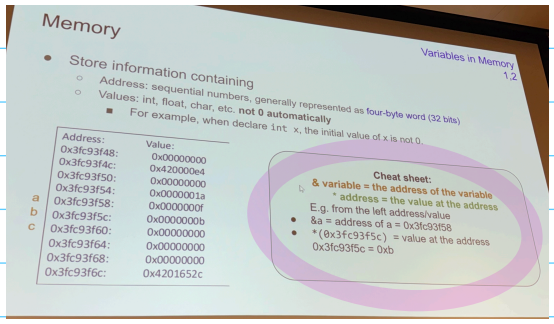
POINTERS:

& = addy

* = pointer



MEMORY:

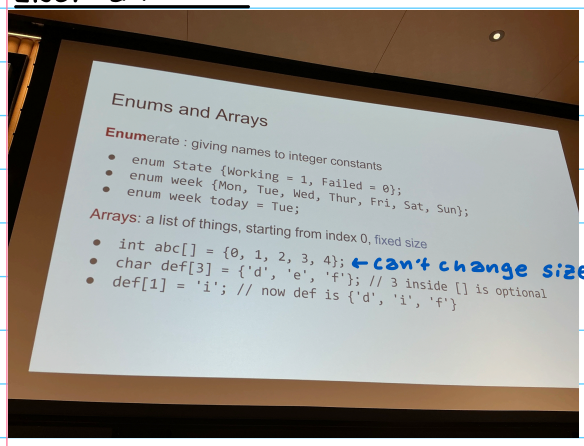


HELP EX:

```c
#include<stdio.h>
int main(void){
  int v = 16;
  int *x = &v;  val of x = addy of v
  x = &v;
  *x = ****x;   val of x = v
  printf("value at %u is %d\n",(int)x, *x);
}
```

diff.

ENUM & ARRAYS:



Exercises - Hex

Ex: 0b0110 - 0100

6 4

Ex:
```
  1
  0x2A
+ 0x49
------
  0x73
```

* review ops (add → 2 carries
sub → base #)

Ex: 8 bit result of 0x2A << 2

8 bits 32 / 4 = 10 = 42

binary: 0b00101010

<< 2 → 0b10101000

hex: A8

Ex: 12-bit result 0x26 << 4

32 / 11 = 43

0b000000101011

<<4 → 000010101100

2 B 0

4/7/25 POINTERS, ARRAYS, CHARS, INTS, FLOATS... DATA REPRESENTATION

DIGITAL SYSTEMS:

- represent using binary, on/off, 1/0
- each 0 or 1 value = bit
- n bits - encode 2^n unique things
  - → 1 bit = 2 (0 or 1)
  - → 2 bits = $2^2 = 4$
  - → 8 bits = $2^8 = 256$

HOW MANY BITS NEEDED?

- t possibilities = ceil($\log_2(t)$) bits
- light on/off → 1 bit → $\log_2 2 = 1$
- days of week → ceil($\log_2 7$) → 3
- states in US → ceil($\log_2 50$) → 6

CONTEX MATTERS: 8-bit value, 2^8 unique things, ex: 0b10100011 can mean...

- 163 unsigned
- -93 signed
- memory location

etc.

COMPUTER ORG:

- byte = 8 bits
- 32-bit system = four-byte word

COMPUTER MEMORY:

- 4 byte words w/ addresses
- each word contains 32 bits of data
- express addresses as data in hexadecimal
- consecutive words have addresses that are 4 bytes apart
- 32-bit system can have 2^{32} bytes of data, or 2^{30} words
- variable store & pointers

POINTERS:

- pointers must have size (32 bits in size)
- data types have diff. amounts of bytes
- pointers must have type

POINTER SYNTAX:

```
int x;
int *ptr_x = &x;        & gets addy of x
int r = *ptr_x + *ptr_y  * creates pointer of type int
```

*dereference when used w/ op.

EX:

- What will this output?

```
int main(void){
  int x = -16; //make signed int -16
  int* ptr_x = &x; //point pointer ptr_x to address of x
  int y = -32; //make a int of -32
  int * ptr_y = &y; //point pointer ptr_y to address of y
  int result = *ptr_x + *ptr_y; //add thing ptr_x points
                                //to thing ptr_y points
  printf("%d + %d is %d\n", x,*ptr_y,result);
}
```

-16 -32 ↖ pointer to y points to y

Output:

```
-16 + -32 is -48
```

Ex:

- What will this output?

```
int main(void){
  int x = -16; //make signed int -16
  int* ptr_x = &x; //point pointer ptr_x to address of x
  float y = -32; //make a float of -32
  int* ptr_y = &y; //point pointer ptr_y to address of y
  int result = *ptr_x + *ptr_y; //add thing ptr_x points
                                //to thing ptr_y points
  printf("%d + %f is %d\n", x,y,result);
}
```

→ made a float but an int pointer to it

→ interpreted float as an int → CANT!

→ diff. from casting — int & floats are incompatible.

Output:

```
-16 + -32.000000 is -1040187408
```

Interpreting the contents at &y as an int is a problem. y is a float, but ptr_y is an int pointer. When we dereference, we're interpreting the bits as if they encode an integer

Ex:

- What will this output?

```
int main(void){
  char c = 'J';
  char* ptr_c = &c; //point ptr_c to c's address
  int* ptr_d = &c; //point ptr_d to c's address
  printf("ptr_c points to %c. ptr_d points to %d\n", *ptr_c, *ptr_d);
}
```

→ interpreting char as int

→ interpreting right byte but wrong neighb.

Output:

```
ptr_c points to J. ptr_d points to 525944906
```

Interpreting the contents at &c as an int was a problem. A char is not an int...their encodings and sizes are different! See that in a little bit!

Ex:

- What will this output?

```
int main(void){
  int8_t e = 22; //8 bit signed int e
  int8_t* ptr_e = &e; //point to it with correct pointer type
  int* ptr_f; //make a 32 bit signed pointer
  ptr_f = &e; //point a (32 bit) signed int to e
  printf("ptr_e points to %d. ptr_f points to %d\n", *ptr_e, *ptr_f);
}
```

→ both of type int but diff. int sizes

Output:

```
ptr_e points to 22. ptr_f points to 923355158
```

Interpreting the contents at &e as a 32-bit int was a problem. Both pointers are of integer type but different sizes of integers! (more on that in a little bit!!!)

Ex:

- What will this output?

```
int main(void){
  int8_t e = 22; //8 bit signed int e
  int8_t* ptr_e = &e; //point to it with correct pointer type
  int* ptr_f; //make a 32 bit signed pointer
  ptr_f = &e; //point a (32 bit) signed int to e
  printf("ptr_e points to %d. ptr_f points to %d\n", *ptr_e, *ptr_f);
}
```

→ printing the address, not val.

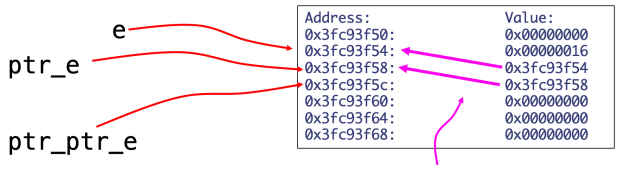
Output:

```
ptr_e points to 22. ptr_f points to 923355158
```

Interpreting the contents at &e as a 32-bit int was a problem. Both pointers are of integer type but different sizes of integers! (more on that in a little bit!!!)

· pointers live in memory & can be pointed to (pointer pointer)

```
int main(void){
  int e = 22; //32 bit signed int e
  int* ptr_e = &e; //point to it with correct pointer type
  int** ptr_ptr_e= &ptr_e; //make a pointer to a pointer
}
```



- Consider this:

```
int main(void){
  int e = 22; //32 bit signed int e
  int* ptr_e = &e; //point to it with correct pointer type
  int** ptr_ptr_e= &ptr_e; //make a pointer to a pointer
  **ptr_ptr_e = 15;
  printf("The value of e is %d \n", e);
}
```

Output: `The value of e is 15`

ptr_ptr_e points to ptr_e, so we need to first derference to get to ptr_e's value, which is itself a reference (address) to e, so we need to then dereference a second time!

· can manipulate data in place

```
int f1(int x, int y){
  return x+y;
}
void app_main(){
  int a = 5;
  int b = 9;
  int c;
  c = f1(a,b);
}
```

"Take in x and y, give me back their sum. I will then copy it into c."

```
void f2(int x, int y, int *z){
  *z = x+y;
} val of c = x+y      val of c
void app_main(){
  int a = 5;
  int b = 9;
  int c;
  f2(a,b, &c);
}          addy of c
```

"Take in x and y, store their sum where z points. Just take care of it. I trust you."

These effectively do the same thing, one modifies "in place" and one modifies externally

· pointers are valuable (refer to large memory structures)

ARRAYS:

- in C, arrays = continuous chunks of memory
  - fixed size & memory (define length @ start)

1) allocates memory for array

2) creates pointer by which you access that spot in memory

- Consider this code...

```
int y[10] = {1,2,3,4,5,6,7,8,9,10};
*(y+3) = 123;     →123
printf("%d\n",y[3]); //what does this print?
```

- Dereferencing with an offset is identical to indexing! In fact:
  - **y[3] === *(y+3)//SAME THING**
- They are equivalent...and goes both ways:

```
int* d; pointer    123
int y[10] = {1,2,3,4,5,6,7,8,9,10};
*(y+3) = 123; change 3rd el
d = y+3;
printf("%d\n",*(y+3)); 123
printf("%d\n",d[1]); //what does this print?
                     →5  d pointing to 1th el → 1 more
*(d-1)=3
```

INDEXING:

- no boundary checks!
- watch limits!
- can print into unknown parts of mem.

ARRAY = FIXED pointer (cant reassign)

```
int a = 11;
int* x = &a;
int y[10] = {1,2,3,4,5,6,7,8,9,10};
//x and y are the same type! int pointers! (sort of)
y = &a; //not allowed will throw compile error
printf("%d\n",*y);
```

ARRAYS INTO FUNCTIONS:

- all C function sees is array's pointer.
- Hand length/metadata about array in along with array

→no way to know len of array in a func.

```
void double_every_entry_better(int* ptr, int length){
  for (int i = 0; i<length; i++){
  *(ptr+i) = *(ptr+i)*2;
  }
}
void app_main_better(){
  int LEN = 10;
  int y[LEN] = {1,2,3,4,5,6,7,8,9,10};
  double_every_entry_better(y,LEN);
}
```

metadata

special char mark end (null)

→ string/char array: char x[] = {'C', 'a', 't', 's', '\0'} longer

= "Cats" ← easier

ASCII: 8 bits, 256 possibilities (only 128 used)

- 0b01000001 = 'A'
- 0b01100001 = 'a'
- 0b0 ____ = NULL

UNICODE: more modern

- up to 4 bytes (32 bits) vs. 1 byte / 8 bits ASCII
- represent mil mill possible chars - we've used 155000 so far

8 bits = 1 byte

- based in ascii
- computers expand in 8... (16, 32, 64)

BOOLEANS:

- 0 = false
- everything else = true

] unsigned int

- one bit

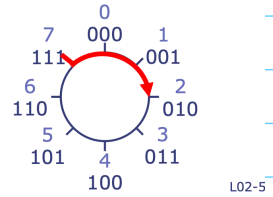
UNSIGNED INTS (uint):

- uint8_t → 1 byte, 8 bits = 2^8 vals
- uint16_t → 2 bytes, 16 bits = 2^{16} vals

INHERENT MODULARITY:

- using fixed # bits may result in overflow (results out of range)
- common approach: ignore extra bit.
- wrap around
- potential cause of bugs, can be good for cryptography

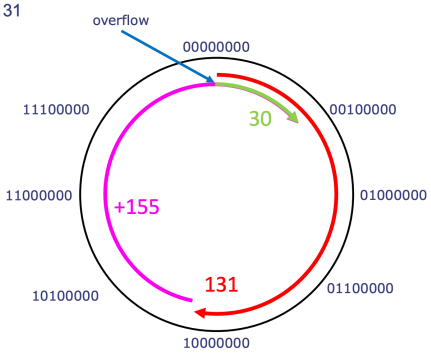
Example: (7 + 3) mod 2^3 ?



- What happens if you add 131 to 155 with 8 bit uint8_t?

```
  10000011   (131)
+ 10011011   (155)
  100011110  (286)
  00011110   (30)
```

overflow



NEGATIVES: solutions...

SIGN BIT:

- if msb is 1, interpret as neg. sign

0b00010001 = +(16+1) == 17
0b10010001 = -(16+1) == -17

→ however, signed 0 (+0 & -0) = BIG PROBLEM - double 0

"ONE'S COMPLEMENT":

- if msb is 0, interpret as unsigned val
- if msb is 1, neg.

-A = ~A

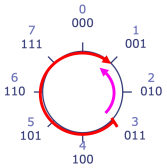
0b00010001 == +(16+1)
0b11101110 == bitflip of 17 == -17

] PROBLEMS:
- signed 0
- add/sub has carry/wrap around

INHERENT MOD:

- add 1 & move CW

- If I start at "3" aka 0b011, what could I add to get to 1?
- To go back 2, I can add:
  - $2^3 - 2 = 6$
- (3+6)%8 = 1.
- Or: "-010" = 110
- -2 = 8 - 2

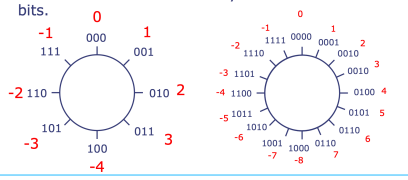


- The negative of a number A can be expressed as:
  $-A = 2^3 - A$ $\quad -A = 8 - A$
- Or written a different way:
  $-A = 0b001 + 0b111 - A$
- 0b111 *minus* any 3 bit value will be the same as the bitflip aka bitwise inversion of that value ($\sim A$)
  $-A = 0b001 + (0b111 - A)$
- So the negative of any value must be:

$$-A = 1 + \sim A$$

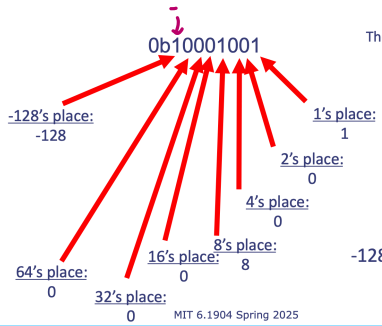
2'S COMPLEMENT: $-A = 1 + \sim A$

- If we make 0b100 into -4, the system of numbers becomes consistent and easily extensible to more bits.



- pos → neg range, not inherently -
- negative weighted bits
- msb → -2^{N-1}
- most pos # = $+2^{N-1}-1$
- if all bits = 1 → 0b111...11 → -1
- no double zero

- If you want to interpret it as a 2's complement



The 0b means binary

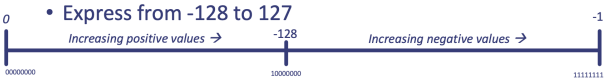
Ex: 0b01001001 = +89!

-128+8+1 = **-119**

/7/25 MIT 6.1904 Spring 2025

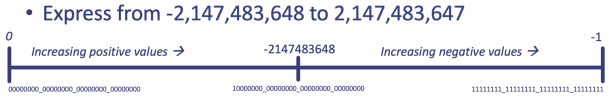
SIGNED INT (ints):

- int8_t: 1 byte (8 bits): 2^8 values: 256 numbers to rep
  - Express from -128 to 127



- int16_t: 2 bytes (16 bits): 2^{16} values: 65,536 numbers
  - Express from -32,768 to 32,767

- int32_t: 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums
  - Express from -2,147,483,648 to 2,147,483,647



int32_t == "int"

4/7/25 MIT 6.1904 Spring 2025 L02-67

EXTENDING DIGITS:

- need to sign extend

Ex: 8 bit -64 = 0b11000000

16 bit -64 = 0b1111111111000000

SHIFT LEFT: FILL w/ 0s

Ex: +64 = 0b01000000

-64 = 0b11000000

SHIFT RIGHT:

- more sticky
- type dependent
- signed: fill w/ 1s
- unsigned: fill w/ 0s

* LOGICAL = unsigned
* ARITHMETIC = signed!

NON-INT #S: fixed pt representation

- radix marker "." @ fixed pt in binary
- precision & range limited
- more bits on right of . = fewer on left
- many things do not exist in linear space

- Consider two signed numbers: +32 and -32 as 8 bit signed values:

+32 = 0b00100000

-32 = 0b11100000

- If we left shift (<<) what do we get?

+64 = 0b01000000

-64 = 0b11000000

New digits that show up are zeros/same

- Consider two signed numbers: +32 and -32 as 8 bit signed values:

+32 = 0b00100000

-32 = 0b11100000

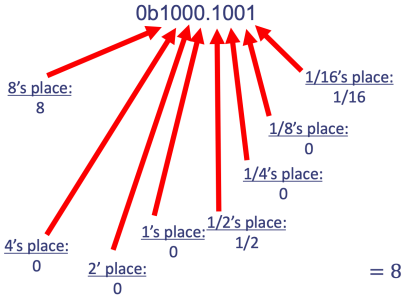
- If we right-shift (>>) what do we get?

+16 = 0b00010000

-16 = 0b11110000

New digits that show up are different

- Pretend there's a radix marker (".") at a fixed location in your binary…then reinterpret meaning of bits.



$= 8\frac{9}{16}$

EXPONENTIAL FORMAT:

- If you needed to encode something that is best represented by orders of magnitude?
- Have an 8-bit type called exp8_t where the 8 bits represent a unsigned 8 bit number **exp** which is used in the following encoding formula*:

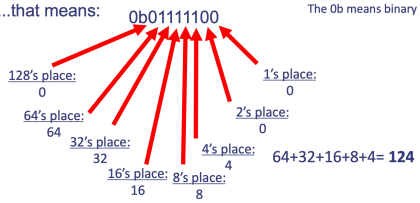
$$\text{Value} = 2^{\text{exp}-127}$$

Base-two exponentials are easy to calculate in binary!

- Represent from:
  - Up to: $2^{255-127} = 2^{128} \approx 3.403\times10^{38}$
  - Down to: $2^{0-127} = 2^{-127} \approx 5.877\times10^{-39}$

made this up but I'm going somewhere with this so bear with me...

- So if you had an exp8_t with 0b01111100 as its value...that means: 0b01111100 (The 0b means binary)



64+32+16+8+4= **124**

- So actual = $2^{124-127} = 2^{-3} = 0.125$

· large range of #s, but limited precision

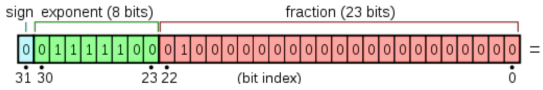
IEEE 754

· float: 4 bytes (32 bits), 2^{32} vals

· wide range & good precision

· decimal pt is floating not fixed

format:



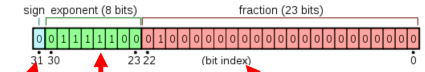
= 0.15625

$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

· exp up to $2^{255-127} = 2^{128}$

down to $2^{0-127} = 2^{-127}$

$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$



Sign bit: +1

Exponent*: 1/8 (0.125)

Mantissa: $1 + 2^{-1} \cdot 0 + 2^{-2} \cdot 1 + 0 \ldots = 1.25$

*already did earlier

ALL TOGETHER:

$$= +1 \cdot \frac{1}{8} \cdot \frac{5}{4} = \frac{5}{32} = 0.15625$$

in class: just care ab REPRESENTATION

· 32 bit float does NOT let us represent more #s. 2^{32} unique vals only

· compromise → sacrifice precision for range

· can have massive "gaps"

· approximation based

- 32 Bit floats represent 2^{32} ($4.3\text{x}10^9$) values between $-3.4\text{x}10^{38}$ to $+3.4\text{x}10^{38}$
- There is NO WAY to represent every single number between these to limits with only 32 bits.
- There is NO WAY to represent *even* every integer between those limits

```
#include<stdio.h>
int main(void){
  float x = 123456789;
  float y = 123456788;
  printf("The value of x is %f\n",x);
  printf("The value of y is %f\n",y);
}
```

Output:

```
The value of x is 123456792.000000
The value of y is 123456784.000000
```

→ rounded to nearest representable float

- Exponential representation means:
  - Merging (+/-, etc...) two numbers *of vastly different scales* can result in the smaller one getting consumed by the larger one and lost

```
int main(void){
  float x = 1.35e9; //create float that is "1.35 billion"
  float y = 23.0; //create float that is "23.0"
  float sum = x+y; //sum them
  float sum_wo_x = sum - x; //remove x again
  printf("x is %f\ny is %f\nx+y is %f\nx+y-x is %f\n",x,y,sum,sum_wo_x);
}
```

Output:

```
x is 1350000000.000000
y is 23.000000
x+y is 1350000000.000000
x+y-x is 0.000000
```

FLOATS (cont.)

- has a double zero
  → not necess. issue (beneficial for limits)

EXERCISES:

Ex: -5 in 8-bit 2's complement: 0b11111011

Ex: negate 4-bit 2's comp. 0b1110 to make it +2's comp. #

-A = 1 + ~A = 1 + 0001 = 0b0010

Ex: negate 5-bits 2's comp 0b01000 to make neg 2's comp #

-A = 1 + 0b10111 = 11000

Ex:
```
 0b0111
+0b0011
--------
 0b1010
```

Ex:
```
 0b11110
-0b00111
--------
  10111
```

Ex: 0b10000001 > 0b01111101 → 0 (2's comp. not.)

Ex: decimal equivalent of 2's comp. hex 0xab

to binary: 1010_1011

invert: 0101_0100 +1 = 0101_0101 → -85 (64 16 4 1; apply neg. sign)

Ex: 0x3c << 2 in 8 bits?

0011_1100 << 2 = 1111_0000 (F 0)

Ex: 0x8e >>3

1000_1110 >>3 = 1111_0001 (F 1)

Ex: 2's comp. hex of decimal # -1

1 in binary: 0b0001

~A+1 = 0b1111 = F

Ex: 2's comp. 8 bit result of 0x8e >>3

binary: 1000_1110

arithm. Shift: 1111_0001 = F1

Ex: 0xC1080000 < 0xC3980000 → 0

- binary, invert bits & add 1

Ex: What is the decimal equivalent of 32-bit floating point number 0b1011_1111_0100_0000_0000_0000_0000_0000?

- exp = $2^{126-127}$ = 2^{-1} (2+4+8+16+32+64 = 126) × frac = $(1+2^{-1}) = (1+\frac{1}{2})$

$-2^{-1} \times (1+2^{-1}) = -\frac{1}{2} - \frac{1}{2^2} = -0.5 - 0.25 = -0.75$

Ex: 0x43A80000 (10)

0100_0011_1010_1000_0000_0000_0000_0000

+ $2^{135-127} = 2^8 \cdot (1+2^{-2}+2^{-4})$

256 · (1 + 1/4 + 1/16) = 336
1.3125

Ex: -25.0 in floating 32-bit hex

Sign = 1 (-)

25 = 16+8+1 = 0b11001 → >>4

0 = 0

0b1.1001 × 2^4

exp-127 = 4 → exp = 131

131 = 128 + 2 + 1 = 0b10000011

1_10000011_1001_000...

C 1 C 8 0000 → C1C80000

Ex: 0.21875 in 32-bit f.p. hex

Sign = 0

0 = 2^0 = 0b0 → << -3

0.21875 = 2^{-4} 0.125 + 2^{-3} 0.625 + 2^{-5} 0.03125

= .00111 0... ×2^{-3}

110....

exp-127 = -3 → exp = 124 → 0b01111100 × 2^{-3}

0_01111100_110 0000 0_000...

sign exp mantis.

3 E 6 0 0000

HELP!

Write the number 0.21875 in 32-bit floating point representation. Write your answer in hexadecimal:

3E600000

Submit View Answer 100.00%

Ex: Signed 0b000011 = 3 0b1010 ← invert = 0b0101 + 1 = 0b0110 = 6

-6+3 = -3 → 0b11101

Ex:

```
a = 0b1100;
b =  0b100;
signed result = a == b;
result = ?
```

○ 0

◉ 1

Ex: a = 0b11011 = 0b00101 = -5 a-b = 2 → 0b00010

b = 0b1001 = 0b0111 = -7

Ex: a = 0b01011 = 11 a+b = 17 0b10001

b = 0b0110 = 6

Ex: 2+3<<4 / 2 == 7^5*2 | 14 mult/div

2+3<< 2 == 7^10 |4 add

0101

5<< 2 == 7^10 | 4 bitshift

0100 == 0111 ^ 1010 | 0100 ==

0000 ^1010 | 0 100 ^

1010 | 0100 = 1110 = 14

Ex:

```
int* x = (int*) 0;
x=x+3;
```

What is the value of x? 12

View Explanation 100.00%

each int takes 4 bytes in memory

chars take 1

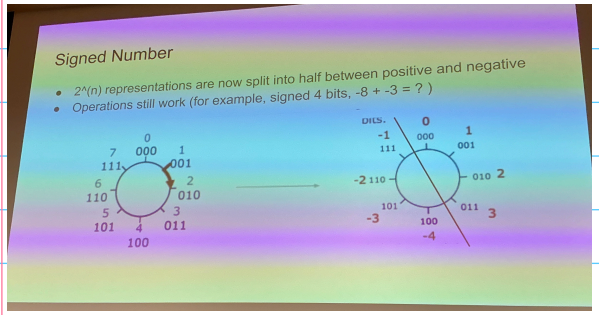
so x=x+3 = x + 3(4) = 12

Ex: int [10] vals = {0, 3, 6, 9, 12, 15, 18, 21, 24, 27}

*(&vals[7] -2) = move 2 elements back from vals[7] = 15

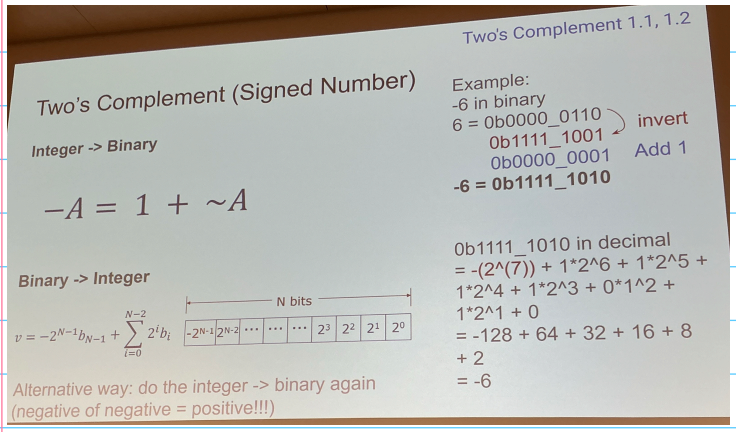
4/8/25 - RECITATION 2: signed #s, floats, pointers

SIGNED #:



INT → BINARY $-A = 1 + \sim A$

BINARY → INT $v = -2^{N-1} b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i$



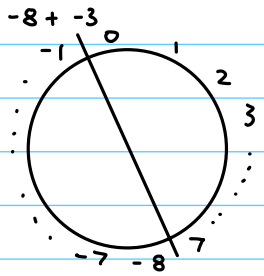
EX: 1111_1010

~ invert: 0000_0101

+1: 0000_0110

#: = 6

EX: -8 + -3



4 bits signed

-8: 0b1000 3: 0b0011

-3: ~ & +1: 0b1101

add
0b1000
0b1101
1 0101
5

EX: Smallest/largest # represented using 2's complement?

6 5 4 3 2 1 0

smallest = largest ⊖ value = $0b1000000 = -2^6 = -64$

largest = largest ⊕ value = $0b0111111 = 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 63$

LOGICAL & ARITHMETIC SHIFT:

<< always fill with 0s

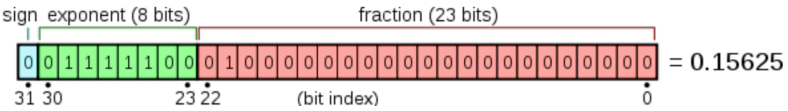
>>

unsigned = patch with 0s (logical)

signed = patch with 1s if MSB=1, 0s other (arithmetic)

FLOATS:

2) Floating point representation (Wikipedia)



$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

Floating point numbers are represented using 32 bits of binary. The most significant bit represents the sign (1 is negative, 0 is positive), the next 8 bits represent the exponent +127, and the last 23 bits represent the mantissa. The mantissa encodes the fractional part of the number. The actual value of the number is $sign * 2^{exponent-127} * 1.fraction$.

2.1) Floating point to decimal

To convert from floating point to decimal:

- Convert hex representation to sign_normalizedExponent_mantissa representation.
- If sign is 1, then sign is negative otherwise sign is positive.
- Find the actual exponent by subtracting 127 from the normalized exponent.
- Express numeric portion as 1.mantissa.
- If the actual exponent is positive, shift this number to the left by exponent bits. If the exponent is negative, shift this number to the right by exponent bits.
- Convert the resulting value to decimal treating each position to the left of the decimal as a positive power of 2 starting at 0, and each position to right of the decimal point as a negative power of 2.

Ex: decimal equivalent of 32-bit floating point number 0x40840000 ?

convert to binary: 0100_0000_1000_0100_0000_0000_0000_0000

sign | exponent | mantissa

exponent: $2^7+1 = 129$

$2^{-1}\ 2^{-2} \ldots$

$= (-1)^0 * 2^{129-127} * (1 + [0.2^{-1}] + [0.2^{-2}] \ldots + [2^{-5}])$

$= 1 * 2^2 * (1 + \frac{1}{25})$

$= 4 + \frac{1}{2^3} = \boxed{4.125}$

2.2) Decimal to Floating point

To convert from decimal floating point:

- Convert original number to base 2
- Shift number to the right or left so that the shifted number is of the form 1.mantissa. In other words, the decimal is just to the right of the leftmost 1. A shift to the left represents a negative exponent whose magnitude is the number of bits shifted, while a shift to the right represents a positive exponent. For example, $0b100.0 = 1.0 * 2^2$, whereas $0.01 = 1.0 * 2^{-2}$.
- Find the normalized exponent by adding 127 to the actual exponent.
- Your floating point number is sign_normalizedExponent_mantissa
- Convert the 32-bit binary representation to hexadecimal.

Ex: -5.75 in 32-bit floating point representation

sign = 1 (negative)

$5 = 4 + 1 = 2^2 + 1 = 0b101$

$0.75 = 0.5 + 0.25 = 2^{-1} + 2^{-2}$

→ 0b101.11000

→ 2 (sci. notation)

0b1.011100 $\times 2^2$ (shift left 2)

mantissa

exp - 127 = 2 → exp = 129

$129 = 128 + 1 = 1000001$

1_10000001_01110000000000000000000

23 bits

C 0 B 8 0 0 0

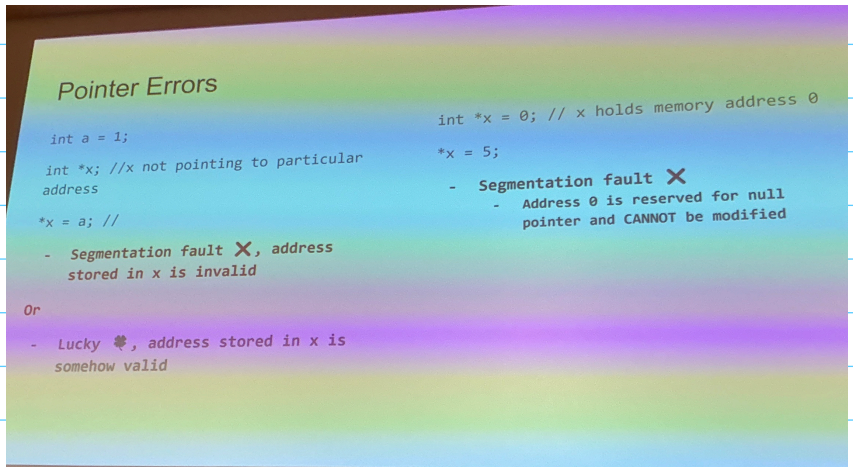
Ex: $0.111 \rightarrow 1.11 \times 2^{-1}$

POINTERS:

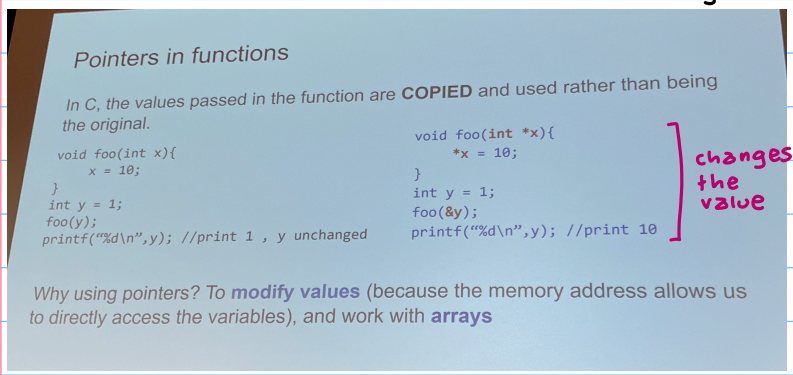
- 32 bit memory addy
- diff. data types = diff. # of bytes
- need to specify type of pointer so it knows how many bytes to read & interpret

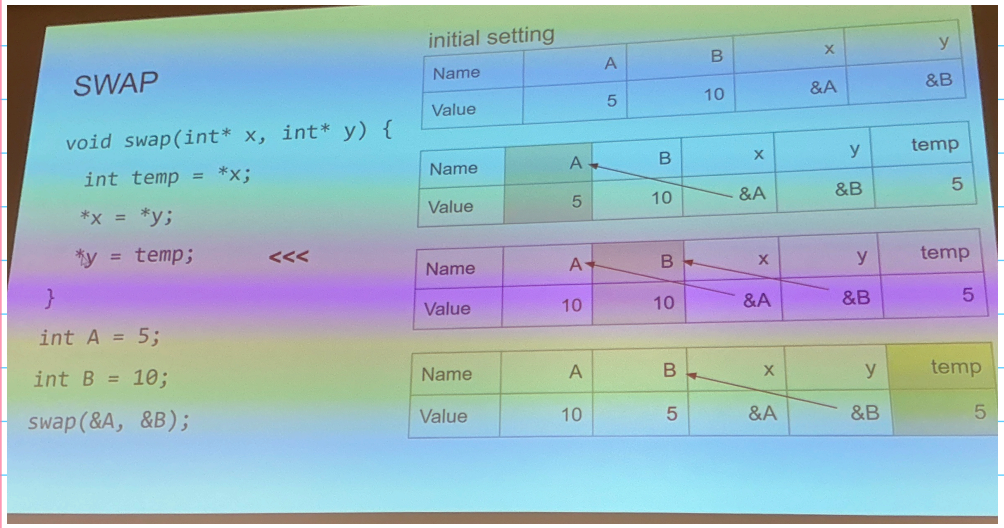
&x = addy of x

*y = value of memory addy of y = dereference y



- values passed in function as COPIES not original





ARRAYS:

- continuous chunks of memory
- must declare type/size
- stores first addy in section of memory

Ex: `int y[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`

```
printf("%d \n", y[3]);
        *(y+3)
        (y+3)[0]
```

all print 4!

`(y+3)[2]` → prints 6

STRING

- array of chars (constant pointer)
- use ascii encoding to represent chars
- use `'\0'` null char to indicate end of string

Ex: `printf("%c \n", 65)` → 'A'

`%d \n` 'C' → 67

String = " " double quotes

char = ' ' single quotes

deref: `s[2]='a'`

`*(s+12)='?'`

`printf("%s", s+2)` ← prints until end when it sees \0 null

output: "llo world"

Ex:

3) Square

Write a function that takes a pointer to an int as an input and returns the square of the value it points to.

```
int square(int* x) {
    //your code
    return ****x;  value @ x  # value @ x
}
```

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

Ex:

1) String index

Write a function that recieves a string and a character as inputs and it returns the first index at which that character is four in the string or -1 if its not found. Again, write this using array indeces and then rewrite it by updating the pointers and dereferencing them to access the string's characters.

```
int string_index(char *s, char c) {
    // don't have the len -- know to stop when reach \0 (value = 0)
    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] == c) return i; // if at index i, char == char
    }
    return -1; // if failed

    // pointer version
    int i = 0;
    while (*s != 0) { // loop until s hits end of the string
        if(*s == c) return i; // dereference s, check equal to c
        s++; // move s to next element
        i++;
    }
    return -1;
}
```

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results

5) Pointers to pointers

EX:

Show/Hide Line Numbers

```c
int x = 60004, y = 314;
int *ptr, *ptr2;
ptr = &x; // ptr points to x, so *ptr == 60004
int **ptr_to_ptr;
ptr_to_ptr = &ptr; // ptr_to_ptr points to ptr, so *ptr_to_ptr == ptr and **ptr_to_ptr == 60004
x = 0; // we still have **ptr_to_ptr == *ptr == x, just now they're all 0
*ptr = 1; // we still have **ptr_to_ptr == *ptr == x, just now they're all 1
ptr2 = &y;
*ptr_to_ptr = ptr2; // This is tricky! Draw a diagram with what everything points to and what value everything has.
// Answer: We still have *ptr_to_ptr == ptr1. Since we assigned the new value to ptr2, now ptr1 == ptr2. In particular this means **ptr_to_ptr == *ptr == *ptr2 == y == 314
y = 77; // Changing the value of y changes all of the above equalities!
// But what is x? Nothing points to x anymore, so we still have x == 1
x = 2; // This changes x, but none of our pointers are related to x anymore so nothing else changes
```

[4/14/25] - C STRINGS

Will inherit what's in memory beforehand

```c
int x[] = {1,2,3,4}; //make four long int array call it x
int y[10]; //create ten long int array, call it y (But don't declare values)
int z[5] = {1}; //create array 5 long called z, 0th element is 1, rest are 0
```

ARRAYS:

- continuous portions of memory accessible via pointer
- char array syntax more flexible
- char arrays always null-terminated

```c
char a[10] = {'t','h','e',' ','c','a','t','.'}; //10 char array with two nulls
                                                 //at end
char b[] = "the cat."; //9 long array with "the cat." followed by null
```

```c
void app_main(){
  int x[] = {1,2,3,4,5}; //compiler figures out array length large enough for items listed
  //10 long array of ints accessible through ptr y
  //only five vals specified (rest are unknown)
  int y[10] = {10,11,12,13};
  printf("%d\n", x[1]); //prints 2
  printf("%d\n", *x); //prints 1 same as x[0] or *(x+0)
  printf("%d\n", x); //prints 1070153528 (3fc93f38 in hex)
  printf("%d\n", x+1); //prints 1070153532 (3fc93f3c in hex)
  printf("%d\n", *(x+2)); //prints 3...same as x[2]
  printf("%d\n", *(x+5)); //prints undefined...same as x[5]…no idea what's in that memory
  printf("%d\n", *(y+10)); //prints 1...because you bled into where x is and x[0] is 1
}
```

- What does the following print?

hexadecimal

```c
printf("%x\n", *(y+2)); 0x0C
```

12 in hex

```c
void app_main(){
  int x[] = {1,2,3,4,5}; //compiler figures out array length large enough for items listed
  //10 long array of ints accessible through ptr y
  //only five vals specified (rest are unknown)
  int y[10] = {10,11,12,13};
  printf("%d\n", x[1]); //prints 2
  printf("%d\n", *x); //prints 1 same as x[0] or *(x+0)
  printf("%d\n", x); //prints 1070153528 (0x3fc93f38 in hex)
  printf("%d\n", x+1); //prints 1070153532 (0x3fc93f3c in hex)
  printf("%d\n", *(x+2)); //prints 3...same as x[2]
  printf("%d\n", *(x+5)); //prints undefined...same as x[5]…no idea what's in that memory
  printf("%d\n", *(y+10)); //prints 1...because you bled into where x is and x[0] is 1
}
```

each int occupies 32 bits

Address:	Value:
0x3fc93f0c:	0x420001ec
0x3fc93f10:	0x0000000a
0x3fc93f14:	0x0000000b
0x3fc93f18:	0x0000000c
0x3fc93f1c:	0x0000000d
0x3fc93f20:	0x00000000
0x3fc93f24:	0x00000000
0x3fc93f28:	0x00000000
0x3fc93f2c:	0x00000000
0x3fc93f30:	0x00000000
0x3fc93f34:	0x00000000
0x3fc93f38:	0x00000001
0x3fc93f3c:	0x00000002
0x3fc93f40:	0x00000003
0x3fc93f44:	0x00000004
0x3fc93f48:	0x00000005

"int y[10]" — Stack into memory

"int x[]"

- but, computer doesn't see whole memory - just glimpses

sizeof: provides size of data object (in BYTES)

```c
char a[10] = {'t','h','e',' ','c','a','t','.'}; //10 char array with two nulls
char b[] = "the cat"; //8 long array with "the cat" followed by null
int x;
char y;
int z[15];
printf("%d\n",sizeof(a)); //prints 10
printf("%d\n",sizeof(b)); //prints 8
printf("%d\n",sizeof(x)); //prints 4
printf("%d\n",sizeof(y)); //prints 1
printf("%d\n",sizeof(z)); //prints 60 ← 4·15 (each int = 4 bytes)
printf("%d\n",sizeof(z)/sizeof(int)); //prints 15
```

will get you array

- works for char b/c each char is only 1 byte

in C, all function args are passed by value!

- when array is handed in as arg to function in C, can only see pointer, NOT the array

```c
int sum_array(int* ai){
  int sum = 0;
  for (int i = 0; i < sizeof(ai); i++){
    sum+= ai[i];
  }
  return sum;
}

int main(void){
  int aa[10] = {2,4,6,8,10,12,14,16,18,20};
  int c = sum_array(aa);
  //value of c? =2
}
```

- **sizeof** provides the number of bytes that a *thing* is.
- On previous slide, the *"thing"* is a pointer inside a function and that is:
  - only four bytes (on 32-bit system)
  - or eight bytes (on a 64-bit system)
- There is NO WAY to inherently know the size of an array that is passed in as an argument in C.
- You MUST:
  - Pass along variable informing of size of array
    - (general solution)
  - Establish some sort of rule-set for the size of the object
    - (C-strings)

```c
int aa[10] = {2,4,6,8,10,12,14,16,18,20};

int sum_array(int* ai){
  int sum = 0;
  printf("size of ai: %d\n",sizeof(ai)); 4
  printf("size of aa: %d\n",sizeof(aa)); 40
  for (int i = 0; i < sizeof(ai); i++){
    sum+= ai[i];
  }
  return sum;
}

int main(void){
  int c = sum_array(aa);
  printf("%d\n",c);
}
```

What is printed here?

more acceptable:

```c
int sum_array(int* ai, int arr_len){
  int sum = 0;
  printf("size of ai: %d\n",sizeof(ai));
  for (int i = 0; i < arr_len; i++){
    sum+= ai[i];
  }
  return sum;
}
int main(void){
  int aa[10] = {2,4,6,8,10,12,14,16,18,20};
  int bb[5] = {19,20,21,4,1};
  int c = sum_array(aa,sizeof(aa)/sizeof(int));
  int d = sum_array(bb,sizeof(bb)/sizeof(int));
  printf("%d\n",c);
  printf("%d\n",d);
}
```

- sizeof is NOT a function, it's an operator
- interpreted @ compile time not run time
- before code runs, evals where used & replaced w/ #

there's no way to inherently know size of array

SOLUTIONS:
- pass along var informing size of array (general soln)
- establish some sort of rule-set for size of obj (C-str)

C-STRING
- null-terminated char array
- all elements are ASCII
- c-string array points to where 1st null occurs

String.h
- library
- determine attributes, manipulate, compare
- **Strlen**: determine len of C-string
  `int strlen(const char *str)`
  - takes in pointer to C-string `const char *`
  - returns length (# of non-null chars up until null)
- **Strcpy**: copy a c-string
  - copy to destination pointed by src (transfer data)
  - can be dangerous (overwrite)
  - prez lives after buffer in memory
    So when we call jstrcopy, copies over it
- **Strcat**: concat. a c-string
  - adds onto destination pted to by src

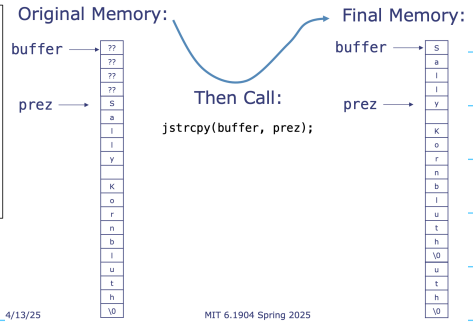
```c
int jstrlen(const char *str) {
  char *searcher = str;
  while (*searcher != '\0') {
    searcher++;
  }
  return searcher-str;
}
```

```c
char * jstrcpy (char *destination, const char *source) {
  uint_32 spot = 8;
  while (*(source+spot) != NULL) {
    destination[spot] = *(source+spot);
    spot += 1; ← write code before switch ind.
  }
  destination[spot] = 0;
  return destination;
}
```

```c
char *jstrcat (char *destination, const char * source) {
  int null_loc = jstrlen(destination);
  return jstrcpy(destination + null_loc, source);
}
```

IN MEMORY:

```c
int main(void){
  char prez[] = "Sally Kornbluth";
  char buffer[4];
  int len = jstrlen(prez);
  printf("length of string: %d\n", len);
  jstrcpy(buffer, prez);
  printf("Original String: \"%s\"\n",prez);
  printf("Copied String: \"%s\"\n",buffer);
}
```




```
length of string: 15
Original String: "y Kornbluth"
Copied String: "Sally Kornbluth"
```


- all bytes have addresses & accessible

restrictions: →

- **strcpy**: copy a c-string
- **strncpy**: copy n characters from a c-string
- **strlcpy**: copy n-1 characters from a c-string and makes sure to NULL terminate

- **strcat**: concatenate a c-string
- **strncat**: concatenate n characters from a c-string
- **strlcat**: concatenate n-1 characters from a c-string and makes sure to NULL terminate

STRING.H COMPARISONS:

strcmp compares first n chars

strncmp compares entire c-string

OUTPUTS:
<0 first unmatching char is lower in str1 than str2
0 equal
>0

string.h Search Functions

- **strchr**: Find first instance of character in c-string
- **strrchr**: Find last instance of character in c-string
- **strstr**: Find location of sub c-string inside c-string
- **strtok**: Tokenize c-string into sub c-strings. (exercise this week!)

```c
char * strchr ( const char * haystack, char needle );
```

- Looks for first instance of **needle** in c-string **haystack** and returns a pointer to it
- If it isn't found, it returns a **NULL** Pointer!
- Example:
  - Determine if there is an 'l' in "Sally Kornbluth"
  - How could we determine where this first occurrence is?

```c
char *jstrchar(const char *haystack, char needle) {
  uint_32 ind=0;
  while (haystack[ind] != NULL) {
    if (*(haystack+ind)==needle) {
      return haystack+ind;
    }
    ind++;
  }
```

```c
char * strrchr ( const char * haystack, char needle );
```

- Looks for last instance of **needle** in c-string **haystack** and returns a pointer to it
- How must this function's search *differ* from regular strchr?

sprintf:
- not part of string.h lib
- similar to printf but writes to a string you provide, not the terminal
- function returns # chars written

```c
int i = some_function(19,86); //return 2012
char temp[10];
sprintf(temp,"%d",i);
//temp now holds the c-string: "2012\0"
//(NULL is auto-added in sprintf)
```

Ex:

Generate C-string that has every number in it from 0 to 1000!

- One function that repeatedly uses `strcat`:
- One function that uses `sprintf` in conjunction with pointer arithmetic directly!:

```c
//version one using strcat:
char totes[100000];//big enough
void build_num_string_1(){
  totes[0] = 0;//null first value
  for (int i =0; i<1000; i++){
    char temp[10];
    sprintf(temp,"%d ",i);
    strcat(totes,temp);
  }
  printf("%s\n",totes);
}
```

```c
//version 2 using sprintf and
//pointer arithmetic:
char totes[100000];//big enough
void build_num_string_2(){
  totes[0] = 0;//null first value
  int tally = 0;
  for (int i =0; i<1000; i++){
    tally += sprintf(totes+tally,"%d ",i);
  }
  printf("%s\n",totes);
}
```

- strcat has to go through str & search for null repeatedly
  - slow, takes longer
  - longer run times for longer strs

Structs:

- can have member vars, but do not have member methods/funcs
- define, access w/ dot op (ex: thing.x to access thing's x op)

```c
struct Subject{
  int dept;
  int units;
  char name[100];
  int num_students;
};
```

```c
#include<stdio.h>
#include<string.h>
struct Subject{
  int dept;
  int units;
  char name[100];
  int num_students;
};

int main(void){
  struct Subject our_course;
  our_course.dept = 6;
  our_course.units = 6;
  strcpy(our_course.name, "6.1904: Intro to C and Assembly");
  our_course.num_students = 255;
  printf("%s has %d students and is in Course %d and is %d credits.\n",
         our_course.name, our_course.num_students,
         our_course.dept, our_course.units);
}
```

Structs with functions: modular package code

```c
//function takes in struct Subject, returns a struct Subject
struct Subject make_subject2(struct Subject s){
  s.dept = 8;
  s.units = 12;
  strcpy(s.name, "8.02: Physics 2");
  s.num_students = 500;
  return s;
}
```

- args always passed by value

```c
our_course = make_subject2(our_course); //call function on our_course return/overwrit
printf("%s has %d students and is in Course %d and is %d credits.\n",
                our_course.name, our_course.num_students,
                our_course.dept, our_course.units);
```

- lots of time to make copies - create pointers instead
  - lots of mem.

Struct Pointer:

- Just like everything else, you can have a pointer to a struct
- The pointer allows us to reference things in memory (round-about way of passing by reference) and modify it directly!

```c
void make_subject3(struct Subject* s);
```

```c
make_subject3(&our_course); //call function on pointer to our_course
printf("%s has %d students and is in Course %d and is %d credits.\n",
                our_course.name, our_course.num_students,
                our_course.dept, our_course.units);
```

- What would/should this print?

Access members: `(*s).units = 12;` () b/c * is lower priority than .

`s -> units = 12;` same

combo of * . (deref. & member)

C OPERATOR HIERARCHY:

Appendix 3: C Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	

- So putting it all together:

```
void make_subject3(struct Subject* s){
  s->dept = 8; //or (*s).dept
  s->units = 12; //or (*s).units
  strcpy((*s).name, "8.02: Physics 2"); //or s->name
  (*s).num_students = 500; //or s->num_students
}
```

```
make_subject3(&our_course); //call function on pointer to our_course
printf("%s has %d students and is in Course %d and is %d credits.\n",
                our_course.name, our_course.num_students,
                our_course.dept, our_course.units);
```

How Big is Struct:

- at least as big as sum of component parts

Ex:

```
struct Subject{
  int dept;
  int units;
  char name[100];
  int num_students;
};
```

4+4+100+4 = 112 bytes

Struct notes:

- no concept of public/private
- cannot have struct with member struct
- many ways to initialize:

```
//declare but don't initialize:
struct Subject our_course;
//declare and initialize:
struct Subject other_course ={9, 12, "9.01: Intro Brain Stuff", 100};
//declare and initialize (named fields):
struct Subject other_other_course ={.dept=1, .units=12,
                                    .name="1.01: Intro Prob/Inference",
                                    .num_students=100};
```

NULL:

- C says "NULL" is none

EXERCISES - WK 3:

→ char digit = '7'; ← 55 in dec

int result = digit + 0; = 55 + 0 = 55

→ hello! = 104 + 101 + 108 + 108 + 111 + 33 = 565

[4/15/25] - RECITATION

STRUCT:

· Similar to class in Python but no function, only vars

· use . to access members

```
struct Subject {
};
```

returns size (in bytes) of variables

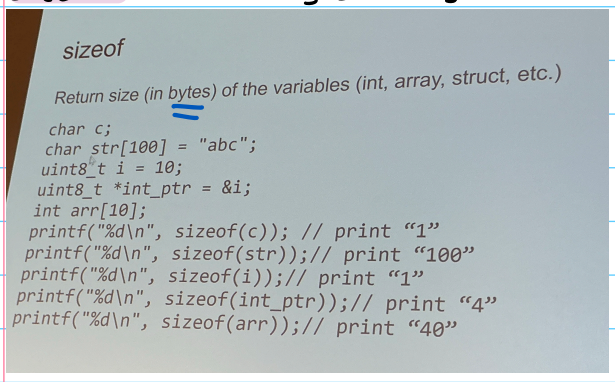
STRUCT POINTERS:

· avoid copying struct (large; directly modify value if needed)

· dereference pointer with * before access with .

(*struct_ptr).member or struct_ptr -> member

sizeof: finds how big something is



int = 4 bytes

char = 1 byte

string.h functions:

· strlen returns length of string (NOT same as sizeof)

· strcpy puts src string into a destination until '\0'

- can overflow if sizeof(dest) < sizeof(src) → might touch other memory

strncpy copy up to n chars (more control)

strlcpy copy up to n-1 character + '\0'

strcat concatenates & overwrites first '\0' of dest with 1st char of src

· can still overflow if len(dest) + len(src) > sizeof(dest)

strncat up to nth char

strncat up to n-1 char + '\0'

strcmp compares str1 & str2

=0 → same

>0 → first diff char: str1's char ascii > str2 char ascii

<0 → 1st str's first diff. char ascii < 2nd

do: strcmp(s1, s2) == 0 don't: s1 == s2

strncmp compares up to n chars

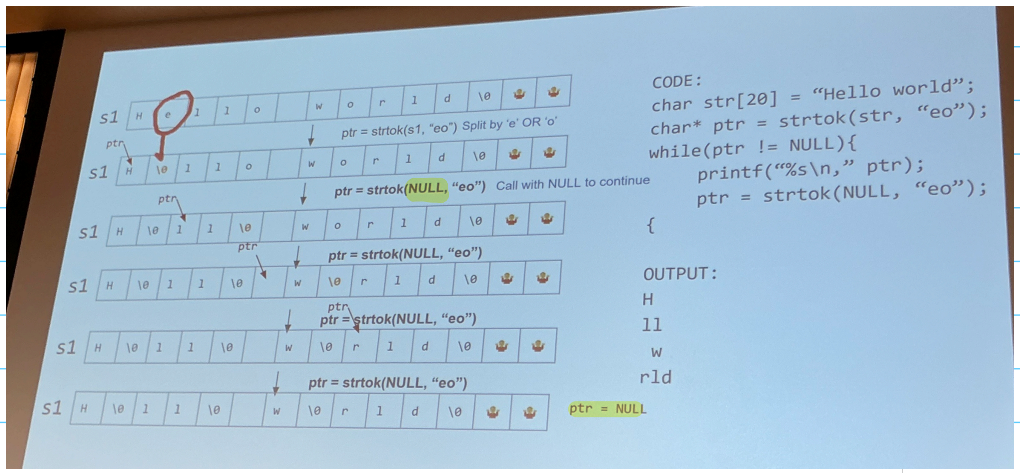
strchr finds 1st occurence of character in string

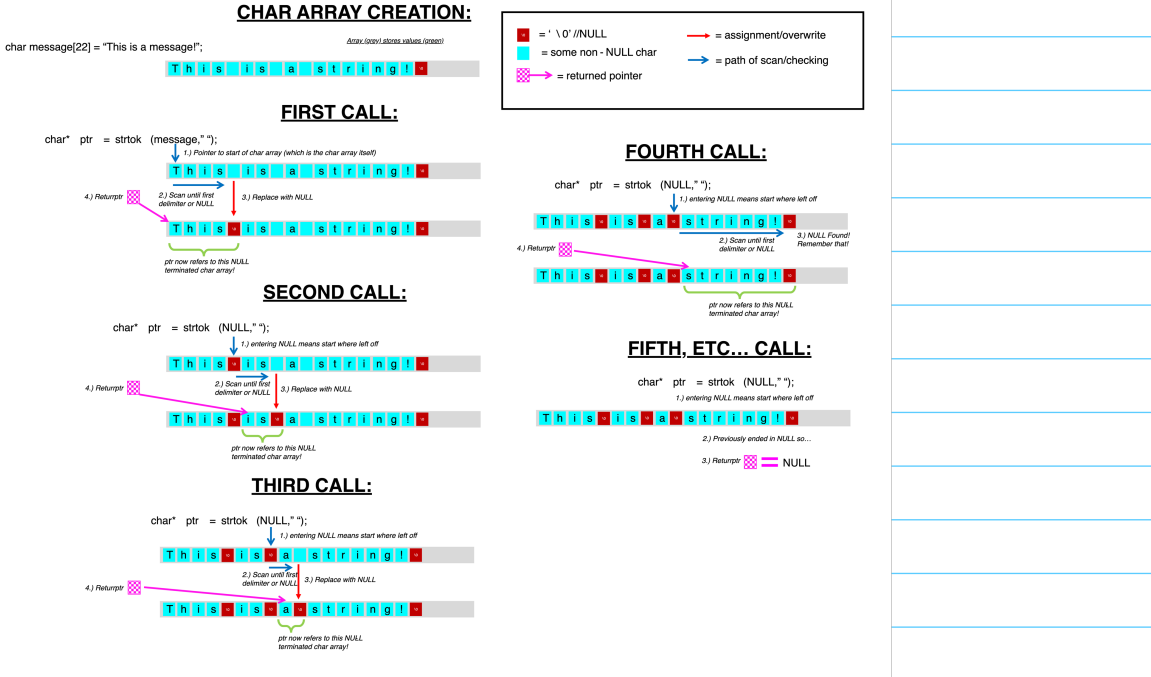
- returns pointer to 1st occurence of char (null if not found)
- **strrchr** finds last instance of char
- **strstr** finds sub C-strings inside

REVIEW

strtok splits str into tokens by delimiters

- split chars by something
- in subsequent call → str replaced by null.





OTHER STRING FUNCS:

sprintf

- Similar to printf, but prints into a string not terminal

```
sprintF (str_var, "format %d %c %f %s \n", a, b, c, d);
```

atoi

- converts string to int
- expects char array

```
Ex: int x = atoi("61904");
```

const char*

- constant char array

TYPECASTING

```
Ex: uint8_t hi = (uint8_t) message[i];
```

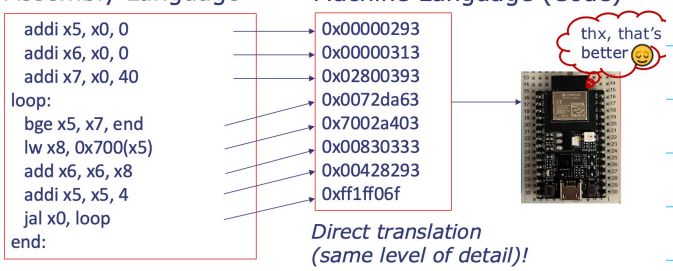
LECTURE 4: RISC-V ASSEMBLY

GENERAL PURPOSE PROCESSOR:

- list of instructions that can execute using its hardware (assembly language)
- assembly lang gets translated into binary (machine code)
  - tells us exact steps that processor carries out

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.



Assembly is for the programmer, machine code is for the processor.

High Level vs Assembly Language

High Level Language	Assembly Language
1. Complex arithmetic and logical operations	1. Primitive arithmetic and logical operations
2. Complex data types and data structures	2. Primitive data structures – bits/bytes
3. Complex control structures - conditional statements, loops and functions	3. Control flow instructions
4. Not suitable for direct implementation in hardware	4. Designed to be directly implementable in hardware
	tedious programming!

ASSEMBLY LANG: sequence of instructions which execute in sequential order unless control flow instruction is executed

- math/logic ops, comparisons, jumps, memory access

INSTRUCTION SET ARCHITECTURE (ISA): contract b/t software & hardware

- functional def. of ops & storage locations
- precise def. of how software can invoke/access them

CISC: (complex instruction set computer) larger instruction set w/ more options/ operation variants

- can have many instructions; more complex; fewer instructions

RISC: (reduced) smaller w/ only bare minimum ops

- basic, smaller set, may need more instructions (but simpler)

Ex:

- We'd like to add two numbers from memory locations 0x7E8 and 0x7EC and store the resulting sum back in 0x7EC.
- In a hypothetical **CISC** architecture you might be able to do this in one instruction:

  ADDMR 0x7EC, 0x7E8

  "add contents stored at memory address 0x7E8 to contents stored at 0x7EC and put the resulting sum into 0x7EC"

- With a **RISC** architecture, this might look like:

```
addi x10, x0, 0x7E8  # x10 <- 0x7E8 + 0 = 0x7E8
lw x11, 0(x10)       # x11 <- Mem[x10]
lw x12, 4(x10)       # x12 <- Mem[x10 + 4]
add x11, x11, x12    # x11 <- x11 + x12
sw x11, 4(x10)       # Mem[x10 + 4] <- x11
```

RISC-V ISA:

- 'I' base int, 'M' x & ÷, 'C' compressed instruct, 'F' & 'D' single/doub-precision FP
- ours: base int 32-bit variant

Registers vs. Memory

- Both store information

Registers	Memory
• Expensive • Physically large • Quick to access • Actually in the computer itself	• Cheap • Very dense/small • Slower to access • Further away/separate from the computer.

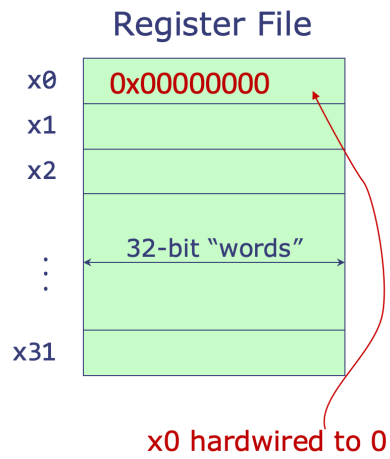
- Conclusion for most computers – *compromise*:
  - Have a small set of registers (maybe twenty or thirty) that you use as much as possible as temporary variables
  - Use memory as rarely as possible!

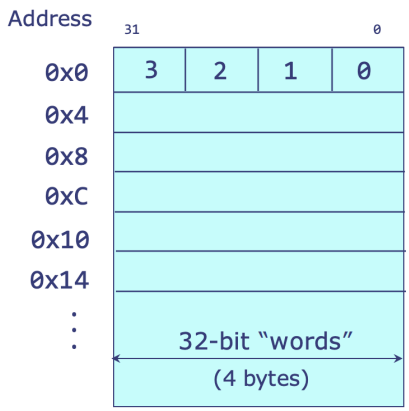
REGISTERS:

- 32 general purpose x0-x31 (x0 = 0)
- each register = 32 bits wide

MEMORY:

- 32 bits wide (1 word)
- addresses = 32 bits





RISC-V INSTRUCTIONS:

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <s reg[rs2]) ? label: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=s reg[rs2]) ? label: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <u reg[rs2]) ? label: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=u reg[rs2]) ? label: pc + 4`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[reg[rs1] + offset]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[reg[rs1] + offset] <= reg[rs2]`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, shamt`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « shamt`
SRLI	`srli rd, rs1, shamt`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u shamt`
SRAI	`srai rd, rs1, shamt`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s shamt`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2][4:0]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2][4:0]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2][4:0]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

computational

control flow

loads/stores

- computational: arithmetic / logical ops on registers
- loads & stores: move data b/t registers & main mem.
- control flow: change execution order of instructions (conditionals, function calls)

Format	Arithmetic	Comparisons	Logical	Shifts
Register-Register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
Register-Immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

COMPUTATIONAL:

REGISTER-REGISTER: oper rd, rs1, rs2

- 2 source operand registers (rs1, rs2) ← can be same
- 1 destination register for result (rd)

Operations (oper):

Arithmetic	Comparisons	Logical	Shifts
add, sub	slt, sltu	and, or, xor	sll, srl, sra

- add x3, x1, x2 # x3 ← x1 + x2
- slt x3, x1, x2 # If x1 < x2 then x3 = 1 else x3 = 0
- and x3, x1, x2 # x3 ← x1 & x2
- sll x3, x1, x2 # x3 ← x1 << x2

REGISTER-IMMEDIATE: oper rd, rs1, constant

↙ indicated w/ i

- 1 source operand comes from register (rs1)
- 1 source operand comes from constant (constant) ← 12 bits (5 for shift)
- 1 destination register (rd)

Arithmetic	Comparisons	Logical	Shifts
addi	slti, sltiu	andi, ori, xori	slli, srli, srai

- addi x3, x1, 4 # x3 ← x1 + 4
- slti x3, x1, 4 # If x1 < 4 then x3 = 1 else x3 = 0
- andi x3, x1, 4 # x3 ← x1 & 4
- slli x3, x1, 4 # x3 ← x1 << 4

- No subi, instead use addi with negative constant.
  - addi x3, x1, -4 # x3 ← x1 - 4

* all values are binary!

NOTE: logical vs. arithmetic right shifts

- Suppose: x1 = **0**0101
  x2 = 00010

`srl` x3, x1, x2 → 00101 / 00010 / 00001

`sra` x3, x1, x2 → **0**0101 / 00010 / 00001

- Suppose: x1 = **1**0101
  x2 = 00010

`srl` x3, x1, x2 → 10101 / 01010 / 00101

`sra` x3, x1, x2 → **1**0101 / 11010 / 11101

Logical right shift: Shift in 0s.
Arithmetic right shift: Shift in value of most significant bit.

Ex:

- What if we wanted to put 0x12345678 into x2?
- No problem!

```
addi x2, x0, 0x12345678 # x2 = 0x12345678 + 0
                        # x2 = 0x12345678
```

- Actually, there is a problem:
  - addi can only encode a 12 bit constant
  - 0x12345678 is more than 12 bits
  - The above code would not compile.

LOAD UPPER IMMEDIATE (LUI): lui rd, luiConstant

- puts 20-bit constant value (luiConstant) in upper 20 bits of a register (rd)
- appends 12 zeros to low end of register
- Supports putting in constants larger than 12 bits into register

```
lui x2, 0x12345     # put 0x12345 into upper 20 bits of x2
                    # x2 = 0x12345000
addi x2, x2, 0x678  # x2 = 0x12345000 + 0x678
                    # x2 = 0x12345678
```

- Let's do a = ((b+3) >> c) - 1

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand
2. Computational instructions can only access registers, not memory.
   - Assume a, b, c are in registers x1, x2, and x3 respectively. Use x4 for temp0, and x5 for temp1.

temp0 = b + 3;	addi x4, x2, 3
temp1 = temp0 >> c;	sra x5, x4, x3
a = temp1 - 1;	addi x1, x5, -1

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
addi x5, x0, 0
addi x6, x0, 0
addi x7, x0, 10

add x6, x6, x8
addi x5, x5, 1
```

LOADS & STORES: only instructions that interact w/ memory

LOAD: lw rd, offset(rs1)

- read 32-bit value stored @ memory addy
- base addy: rs1 stored in register
- offset: offset 12-bit constant
- base + offset must be multiple of 4!
- put 32-bit value in register rd

- lw x1, 0x4(x0) # x1 ← Mem[x0 + 0x4]
- lw x2, 0x8(x0) # x2 ← Mem[x0 + 0x8]

STORE: sw rs2, offset(rs1) taking from register & putting into mem

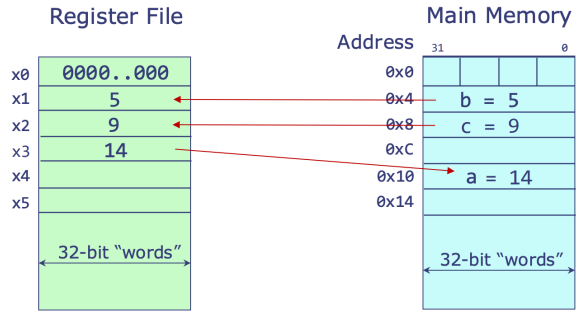
- take 32-bit value in register rs2 and store @ mem addy rs1 + offset
- base addy rs1 always stored in register
- offset offset 12-bit constant
- base+offset must be multiple of 4

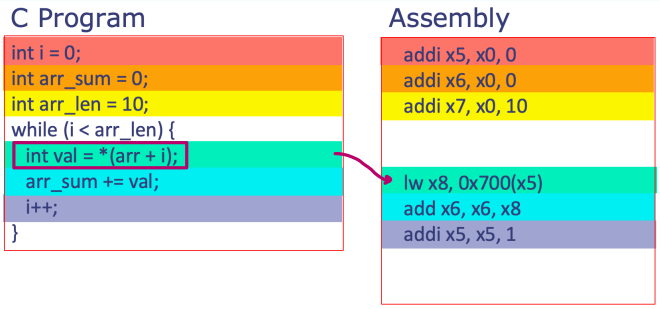
- sw x3, 0x10(x0) # Mem[x0 + 0x10] ← x3

a = b + c

Assume:
- a is at address 0x10
- b is at address 0x4
- c is at address 0x8

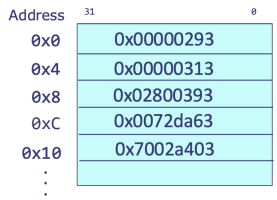
```
lw x1, 0x4(x0)
lw x2, 0x8(x0)
add x3, x1, x2
sw x3, 0x10(x0)
```





CONTROL FLOW:

- assembly lang translated into machine lang & stored in mem. for processor to read as executes
- RISC-V processor has special register called program counter (pc).
  - holds mem. addy of current instruction
  - processor uses this addy to fetch instruction from mem. then decodes & executes it



- Processor fetches 32 bits stored at address 0x8
- Processor interprets those 32 bits as an instruction.
- Processor executes instruction.
  - pc is updated

- assembly always continues to next instruction (pc = pc + 4) unless otherwise told by control flow instruction (pc = pc + offset)

CONDITIONAL BRANCH: branch jump only if condition is met

UNCONDITIONAL JUMP: always jump

allows us to create conditional (if/else statements) & loops (while, for)

LABELS: give convenient names to lines of instructions

- takes no space in memory (no size)

```
some_code: addi x5, x0, 521
  addi x6, x0, 177
some_other_code:
  lui x2, 0x12345
  addi x2, x2, 0x678
```

- some_code refers to addi x5, x0, 521
- some_other_code refers to the instruction immediately below it: lui x2, 0x12345

CONDITIONAL BRANCH: Comp rs1, rs2, label

- first performs comparison to determine if branch taken or not
- if True, branch is taken (jump to label) & updates pc accordingly

Instruction	beq	bne	blt	bge	bltu	bgeu
comp	==	!=	<	≥	<	≥

- beq x1, x2, label # If x1 == x2, jump to # label. Otherwise, go to # next instruction.

Conditional Branch Example

C Code:

```
if (a < b) {
  c = a + 1;
} else {
  c = b + 2;
}
```

Assembly Code:

```
      bge x1, x2, else
      addi x3, x1, 1
      beq x0, x0, end
else: addi x3, x2, 2
 end:
```

Assume:
x1 = a
x2 = b
x3 = c

UNCONDITIONAL BRANCH: jal rd, label

·jump to instruction following label

- pc updates to be addy of instruction

·link stored in register rd

ANOTHER: jalr rd, offset(rs1) (unconditional jump via register+link)

·jump to instruction specified by rs1 + offset

· rs1 is register, offset is constant

·can jump to any 32 bit addy

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
  addi x5, x0, 0
  addi x6, x0, 0
  addi x7, x0, 40
loop:
  bge x5, x7, end
  lw x8, 0x700(x5)
  add x6, x6, x8
  addi x5, x5, 4
  jal x0, loop
end:
```

INSTRUCTION ENCODING:

6 main types:

MIT 6.191 (6.004) ISA Reference Card: Instruction Encodings

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

R-type: Register-Register Instruction Format

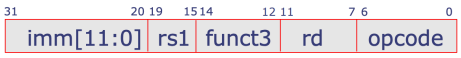
31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode

Some parts are found in the ISA:
- funct7, funct3: encodes the function (add, and, etc.)
- opcode: encodes the instruction type (register-register)

Some parts come from the instruction:
- rs2: source register 2 (5 bits)
- rs1: source register 1 (5 bits)
- rd: destination register (5 bits)

use ISA to determine which register is which!

I-type: Register-Immediate Instruction Format

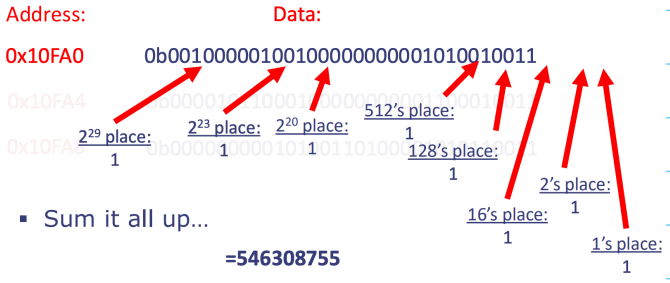


Some parts are found in the ISA:
- funct3: encodes the function (addi, andi, etc.)
- opcode: encodes the instruction type (register-immediate)

Some parts come from the instruction:
- imm[11:0]: immediate (12 bits)
- rs1: source register 1 (5 bits)
- rd: destination register (5 bits)

If we're in a Data Region of Memory

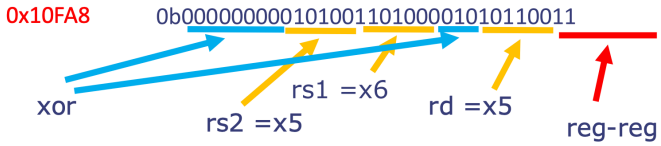
- Maybe the value at 0x10FA0 is an unsigned int:



- Sum it all up…

=546308755

INSTRUCTIONS vs. DATA:



`xor x5, x6, x5`

Bitwise xor x6 with x5 and put back into x5

- So this would get interpreted by the computer as:

Address:	Data:
0x10FA0	0b00100000100100000000001010010011
0x10FA4	0b00000101100001000000000001100010011
0x10FA8	0b00000000010100110100001010110011

```
addi x5, x0, 521
addi x6, x0, 177
xor x5, x6, x5
```

Exercises:

Ex: 0b101 xor 0b011 → 0b110 → 0x06

Ex: 0b101 srli 2 → 0b101 >> 2 → 0b001 = 1 (logical)

★ Ex: lui x4 7 = 0x00007000 ← adds 12 binary 0's (3 hex 0s)

Ex:

```
     addi x1, x0, 7
     addi x2, x0, 2
     blt x1, x2, L1
     addi x2, x2, 4
L1:  addi x2, x2, 5
```

x1→7
x2→2
if 1 < 7 go to L1 ✗
x2 = 2+4 = 6
x2 = 6+5 = 11 → 0x8

Ex: addi x1, x1, -1

imm = -1 → find binary = complement (invert)

1111_1111_1110 +1 → 1111_1111_1111

x1 → 0b00001

variable, table, addi:
1111_1111_1111_00001_000_00001_0010011 ??

Ex: 0x12341234: 0b0000_0000_0111_0001_0101_1000_0110_0011

imm[12] = 0

imm[10:5] = 000000

imm[4:1] = _1000

imm[11] = 0

imm = 00_000000_10000

curr + imm = 0x12341244

Ex: binary → instruction

0b00000000_10111001_10001011_00110011

0 | rs2 = 11 | rs1 = 19 | 000 | rd = 6+16 = 22

Ex: bge x2 x3 label
00010 00011

4 inst after → 4 bits · 4 = 16 → 0000000010000
12 | 4:1 | 0th

0_000000_00011_00010_101_1000_0_1100011
7

Ex: sw x6 8(x2)
rs2 rs1
00010 00110
imm: 12 → 0000000 1 000
8 | 12 | 11:5 | 4:0 | 0
0000001_00110_00010_010_00000_0100011

Ex: 0b01000000_10001000_10000010_00110011 → sub x4, x17, x8
sub | rs2 x8 | rs1 x17 | rd x4 | sub

Ex: 0b00000101_01101001_10110011_10010011 → sltiu x7, x19, 86
imm[11:0] 6+16+64 | xrs1 x19 | 011 sltiu | x7 rd

Ex: 0b10110011_10100101_10101101_10010011 → slti x27, x11, 0xb3a
imm[11:0] | rs1 x11 | 010 SLTI | rd x27
B | 3 | A
0xb3a
signed: 101100111010
u: 010011000101 +1
010011000110
2+4+64+128+1024
$2^{10}+2^7+2^6+6$

[4/22/25] - recitation

ASSEMBLY:

- even lower level than C

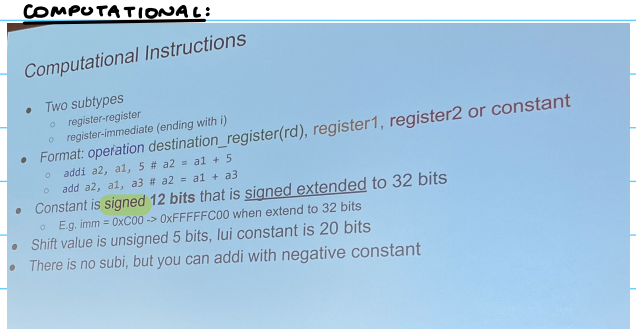
REGISTERS vs. MEMORY

- in C, all vars are in memory
- registers = temp vars that are faster, smaller
- x0 → x31 (0 → 31)
- x0 is always 0 (hardwired)

INSTRUCTION TYPES:

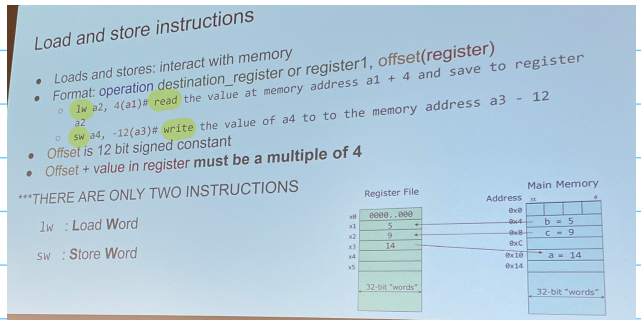
- computational
- loads & stores (move registers ↔ memory)
- control flow (causes PC to jump around)

COMPUTATIONAL:



- LUI = load upper immediate: set upper 20 bits b/c immediate only allows 12 bits

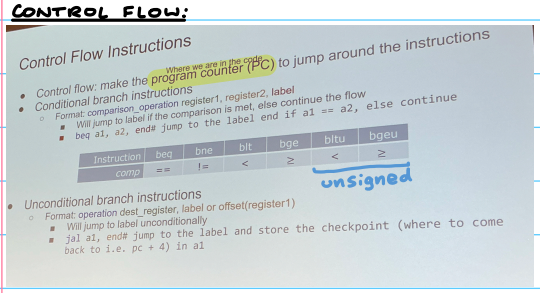
LOAD & STORE:



LABELS:

- takes up no space in memory
- label lines of instructions
- used with control flow instructions

CONTROL FLOW:

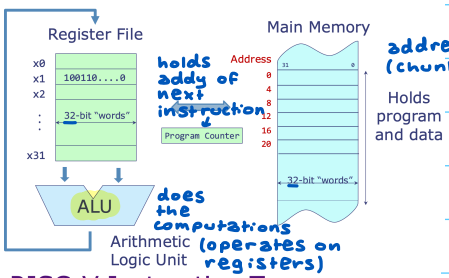


DEBUG:

- csbreak @ lines of interest
- can see registers, memory, etc.

[4/28/25] - COMPILING CODE & IMPLEMENTING PROCEDURES

Components of a RISC-V Processor



RISC-V Instruction Types

- Computational Instructions executed by ALU
  - Register-Register: `oper rd, rs1, rs2`
  - Register-Immediate: `oper rd, rs1, constant (12-bit)` ← only have total of 32-bits, have 12 bits left
  - `lui rd, luiConstant (20-bit)` — can get 32-bit value with these
- Loads and Stores
  - `lw rd, offset(rs1)` — reading val
  - `sw rs2, offset(rs1)` — writing val
  - Memory address = Reg[rs1] + sign_extend(offset)
- Control flow instructions
  - Conditional: `comp rs1, rs2, label` — branch if...
  - Unconditional: `jal rd, label` and `jalr rd, offset(rs1)` — always branch ← sign extended offset
- Pseudoinstructions
  - Shorthand for other instructions

gets sign extented

Ex: 111111111111 = -1

11111_111111111111 = -1 (extend)

PSEUDOINSTRUCTIONS:

Pseudoinstructions

- Not "real", but they make our lives easier. Can use freely!
- They are converted into actual RISC-V instructions when being assembled into machine code.

MIT 6.191 (6.004) ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
`li rd, liConstant`	Load Immediate	reg[rd] <= liConstant
`mv rd, rs1`	Move	reg[rd] <= reg[rs1] + 0
`not rd, rs1` (flip bits)	Logical Not	reg[rd] <= reg[rs1] ^ -1
`neg rd, rs1`	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
`j label`	Jump	pc <= label
`jal label` `call label`	Jump and Link (with ra)	reg[ra] <= pc + 4 pc <= label
`jr rs`	Jump Register	pc <= reg[rs] & ~1
`jalr rs`	Jump and Link Register (with ra)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
`ret`	Return from Subroutine	pc <= reg[ra]
`bgt rs1, rs2, label`	Branch > (Signed)	pc <= (reg[rs1] >$_s$ reg[rs2]) ? label : pc + 4
`ble rs1, rs2, label`	Branch ≤ (Signed)	pc <= (reg[rs1] <=$_s$ reg[rs2]) ? label : pc + 4
`bgtu rs1, rs2, label`	Branch > (Unsigned)	pc <= (reg[rs1] >$_u$ reg[rs2]) ? label : pc + 4
`bleu rs1, rs2, label`	Branch ≤ (Unsigned)	pc <= (reg[rs1] <=$_u$ reg[rs2]) ? label : pc + 4
`beqz rs1, label`	Branch = 0	pc <= (reg[rs1] == 0) ? label : pc + 4
`bnez rs1, label`	Branch ≠ 0	pc <= (reg[rs1] != 0) ? label : pc + 4
`bltz rs1, label`	Branch < 0 (Signed)	pc <= (reg[rs1] <$_s$ 0) ? label : pc + 4
`bgez rs1, label`	Branch ≥ 0 (Signed)	pc <= (reg[rs1] >=$_s$ 0) ? label : pc + 4
`bgtz rs1, label`	Branch > 0 (Signed)	pc <= (reg[rs1] >$_s$ 0) ? label : pc + 4
`blez rs1, label`	Branch ≤ 0 (Signed)	pc <= (reg[rs1] <=$_s$ 0) ? label : pc + 4

- Shorthand
- converted to actual RISC-V instructions that's equivalent

	pseudo:	assem:
Ex:	mv x2, x1	addi x2, x1, 0
	ble x1, x2, label	bge x2, x1, label
	j label	jal x0, label

load immediate:

(32 bits)
smaller #: li x2, 3 → addi x2, x0, 3

larger #: li x3, 0x4321 → lui x3, 0x4
lui & addi addi x3, x3, 0x321
(64 bits) → takes up 2 words in mem.

DATA VS. INSTRUCTION MEM

- different sections of memory
- if program counter pointing to something, we assume it's pointing to instructions
- don't overwrite!
- pc keeps track of next instruction
  - The program counter (pc) register keeps track of the address of your next instruction
    - Default (non control-flow instructions): pc = pc + 4 ← consecutive locations in mem. are 4 bytes apart
    - Control flow instructions might update pc by a different amount.

.= blt is stored @ addy 0x0

```
. = 0x0
blt x1, x2, label      # if x1 < x2: pc = 0x8
addi x1, x1, 1         # else: pc = pc + 4
label:
lw x3, 0x100(x0)       # x3 = 0x12345678
. = 0x100
.word 0x12345678
```

going to 0x100, getting data val, putting into x3

@ 0x100, 0x12345678 is stored there

Ex: Sum array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
  addi x5, x0, 0
  addi x6, x0, 0
  addi x7, x0, 40
loop:
  bge x5, x7, end
  lw x8, 0x700(x5)
  add x6, x6, x8
  addi x5, x5, 4
  jal x0, loop
end:
```

array len is 10 · 4 = 40 bytes apart in mem.

memory addy + bytes offset (i)

incr. by 4 bytes

Ex: conditional statement

C code

```
if (expr) {
  if-code;
} else {
  else-code;
}
```

1. Compile *expr* into *xN*
2. Conditional branch instruction that will jump to *else* when *expr* is 0.
3. Compile *if-code* into assembly.
4. Add unconditional jump past else branch.
5. Compile *else-code* into assembly.

Ex:

```
if (x > y) {
  x = x – y;
} else {
  y = y – x;
}
```

We'll use x10 for x and x11 for y

```
  slt x12, x11, x10
  beq x12, x0, else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
```

← unconditionally jump to skip else portion

Ex: diff. way:

C code

```
if (expr) {
  if-code;
} else {
  else-code;
}
```

1. Conditional branch instruction that will jump when the *opposite* of *expr* is true.
2. Compile *if-code* into assembly.
3. Add unconditional jump past else branch.
4. Compile *else-code* into assembly.

Ex:

```
if (x > y) {
  x = x – y;
} else {
  y = y – x;
}
```

```
ble x10, x11, else
 sub x10, x10, x11
 j endif
else:
 sub x11, x11, x10
endif:
```

Ex: while loop

C code

```
while (expr) {
  loop-code
}
```

1. Conditional branch instruction that will jump past loop body when *opposite* of *expr* is true.
2. Compile *loop-code* into assembly.
3. Add unconditional jump to top of loop at end of *loop-code*.

```
while (x != y) {
   if (x > y) {
    x = x – y;
   } else {
    y = y – x;
   }
}
```

```
loop:
 beq x10, x11, endloop
 ble x10, x11, else
 sub x10, x10, x11
 j endif
else:
 sub x11, x11, x10
endif:
 j loop
endloop:
```

← control flow instructions can be costly... instead of having 2...

diff. way:

C code

```
while (expr) {
  loop-code
}
```

1. Compile *loop-code* into assembly.
2. Put a conditional branch instruction after *loop-code* that jumps to top of loop when *expr* is true.
3. Before starting loop, jump to that conditional branch instruction.

```
while (x != y) {
   if (x > y) {
    x = x – y;
   } else {
    y = y – x;
   }
}
```

```
j compare
loop:
 ble x10, x11, else
 sub x10, x10, x11
 j endif
else:
 sub x11, x11, x10
endif:
compare:
 bne x10, x11, loop
```

← only one control flow!

PROCEDURES: gcd

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal arguments
  - Local storage
  - Returns to the caller when finished
- Using procedures enables abstraction and reuse
  - Compose large programs from collections of simple procedures

```
int gcd(int a, int b) {
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
      x = x – y;
    } else {
      y = y – x;
    }
  }
  return x;
}

bool coprimes(int a, int b) {
  return gcd(a, b) == 1;
}

coprimes(5, 10); # false
coprimes(9, 10); # true
```

← callee

← caller

CALLER: procedure that called other procedure

CALLEE: procedure that was called by caller

* a procedure can be both caller & callee!

How to transfer control to callee & back to caller?

- register x1 (ra) used to hold a procedure's RETURN ADDRESS

- once finished, callee will use ra to jump back to correct instruction within caller procedure

Jump and **Link**

jal: Unconditional jump and link

Format: jal rd, label

- Jump to instruction at label
  - program counter (pc) updates to be address of that instruction
- Link: stored in register rd ← ***NEW THIS WEEK***
  - The *link* is the address of the instruction ***after*** the jal instruction (rd = pc + 4)
  - This is how we remember where we came from!

```
jal x1, label
addi x7, x7, 1
```

1. Jump to instruction following label
2. x1 holds address of addi instruction

- jal ra, sum
  - When calling a procedure, we store the link in ra

```
proc:
    ...
[0x100] jal ra, sum # ra = 0x104
    ...
[0x678] jal ra, sum # ra = 0x67C
    ...
```

```
sum:
  ...
  j ??
```

Equivalent pseudoinstructions:

- jal sum
- call sum

How can the callee use ra to return back to the caller?

Jump and **Link** Register

jalr: Unconditional jump via register and link

Format: jalr rd, offset(rs1)

- Jump to instruction stored at address Reg[rs1] + offset
  - program counter (pc) updates to be address of that instruction
- Link: stored in register rd ← ***NEW THIS WEEK***
  - The *link* is the address of the instruction ***after*** the jalr instruction (rd = pc + 4)
  - This is how we remember where we came from!

```
jalr x3, 0(x1)
addi x7, x7, 1
```

1. Jump to instruction at memory address in x1
2. x3 holds address of addi instruction

How to transfer control back to caller?

- jalr x0, 0(ra)
  - ra indicates where the program should return to
    - As long as sum was called using jal ra, sum or equivalent, and *ra hasn't been overwritten since.*
  - Store link in x0
    - Effectively discarding it... we have no need to remember that we came from the end of a procedure.

* ra remembers return addy from procedure call

```
proc:
    ...
[0x100] jal ra, sum # ra = 0x104   ← putting into ra
    ...
[0x678] jal ra, sum # ra = 0x67C
    ...
```

```
sum:
  ...
  jalr x0, 0(ra)
```

Equivalent pseudoinstructions:

- jr ra
- ret

summary:

Summary: Transferring control between caller and callee

- To call a procedure: jal ra, label
  - or jal label
  - or call label
- To return from a procedure: jalr x0, 0(ra)
  - or jr ra
  - or ret

*Switching control back & forth b/t caller/callee

```
proc:
    ...
[0x100] jal ra, sum # ra = 0x104
    ...
[0x678] jal ra, sum # ra = 0x67C
    ...
```

```
sum:
  ...
  jalr x0, 0(ra)
```

How to communicate arguments & return values?

- we need rules (calling convention)

CALLING CONVENTION: rules for register usage across procedures

function args/return vals:

Symbolic name	Registers	Description
a0 to a7	x10 to x17	Function arguments
a0 and a1	x10 and x11	Function return values

- 1st arg in a0, 2nd a1, ...
- callee can find them!

- The caller is responsible for putting function arguments in the necessary registers before calling a procedure.

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

coprimes responsible for making sure a0 = a, a1 = b before calling gcd...

```
int gcd(int x, int y){
  ...
}
```

... so that gcd can assume a0 = x and a1 = y

- first return val in a0, second a1, ...
- callee responsible for putting return vals in necessary registers

- The callee is responsible for putting return values in the necessary registers before returning.

```
int gcd(int x, int y){
  ...
  return val;
}
```

gcd responsible for putting val in a0 before returning...

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

...so coprimes can assume gcd(a,b) will be in a0, ready to use

How to let procedures use more storage than can fit in registers?

- allocate special section of memory (stack) that isn't for registers

RISC-V STACK:

- grows down from high→low addys (LIFO)
- sp points to top of stack
- push & pull operations
- must make room before pushing, must move ptr after pulling

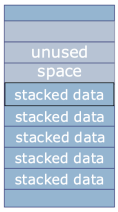
- Two operations: push and pop
  - Push: put something on top of stack

```
addi sp, sp, -4 # allocate space
sw a1, 0(sp)    # put element on
```

  - Pop: take something off top of stack

```
lw a1, 0(sp)    # take element off
addi sp, sp, 4  # deallocate space
```

Stack ptr no longer pts to val we just removed



· any procedure can use stack, but before returning, must put stack back how it was

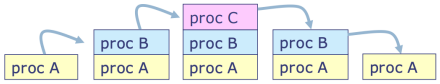
· sp must be reset to value that it was @ beginning of procedure

ACTIVATION RECORD:

· holds all storage needs of procedure that do not fit in registers

· activation records go on stack (LIFO)

· **stack frame:** current procedure's activation record always @ top of stack



What if caller & callee need to use same register?

· RISC-V calling conventions specify rules for sharing

- If a register will (potentially) be overwritten across a function call, its value must be saved on the stack.
  - The caller procedure is responsible for saving some registers…
  - …and the callee procedure is responsible for saving others.
- RISC-V ISA tells us which ones are which!

one or the other (wasteful if both)

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–x7	t0–t2	Temporary registers	Caller
x8–x9	s0–s1	Saved registers	Callee
x10–x11	a0–a1	Function arguments and return values	Caller
x12–x17	a2–a7	Function arguments	Caller
x18–x27	s2–s11	Saved registers	Callee
x28–x31	t3–t6	Temporary registers	Caller

Caller-saved Registers (aN, tN, ra)

→ assumed to not be preserved across procedure calls!

Registers not preserved across a procedure call.
- The callee can use them freely
- The caller doesn't know what registers the callee will use – must assume it'll use them all.

If the caller needs a caller-saved register's value after the procedure call, it must:
- *Before calling the procedure:* save the register's value on the stack.
- *After returning from the procedure:* restore the old value from the stack before next using the register.

Any callee can use these registers freely because it can safely assume the caller will save any register values it will need.

Callee-saved Registers (sN, sp*)

→ IS PRESERVED guaranteed across calls

Preserved across a procedure call!
- The caller can assume that the register's value will be the same before and after the procedure call.

If the callee wants to use a callee-saved register, it must:
- *Before using:* save the register's value on the stack.
- *After using:* restore the old value from the stack before returning to the caller.

*The stack pointer (sp) doesn't go on the stack, just needs to be reset to it's original value.

Any caller can safely assume that these register values will be "untouched" because the callee will make sure to restore them to their original state before returning.

calling convention:

- t-registers (temporary - not guaranteed to be preserved)
- s-registers (saved - guaranteed preservation

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---
zero	x0	Hardwired zero	---

Let's put this all together

Caller: (use freely)

- Saves any aN, tN, or ra registers ***whose values need to be maintained past the procedure call*** on the stack prior to the procedure call
- Restores them ***before using them again*** after returning from the callee.

Callee: (preserved)

- Saves original values of sN registers on the stack ***before using them*** in a procedure.
- Must restore sN registers and stack pointer when done using them, ***before exiting procedure***.

Ex: callee-saved

- Implement f using s0 and s1 to store intermediate values

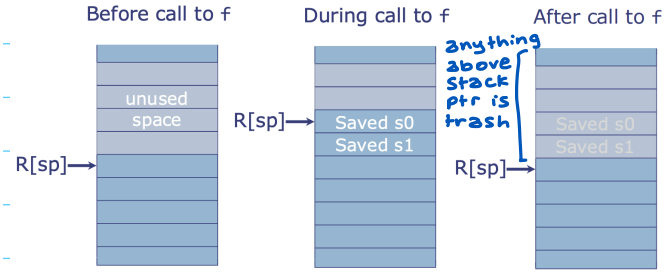
```
int f(int x, int y) {
    return (x + 3)|(y + 0x123456);
}
```

```
f:
  addi sp, sp, -8    # allocate 2 words (8 bytes) on stack
  sw s0, 0(sp)       # save s0
  sw s1, 4(sp)       # save s1
  addi s0, a0, 3     # x (1st arg)
  li s1, 0x123456
  add s1, a1, s1
  or a0, s0, s1
  lw s0, 0(sp)       # restore s0
  lw s1, 4(sp)       # restore s1
  addi sp, sp, 8     # deallocate 2 words from stack
                     # (restore sp)
  ret
```

reverses

return stack how it was before

- Stack contents:



Ex: caller-saved

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -4
sw a1, 4(sp) # save y
jal ra, sum
# a0 = sum(x, y) = z
lw a1, 4(sp) # restore y
jal ra, sum
# a0 = sum(z, y) = w
addi sp, sp, 4
```

cannot assume what callee did. caller must save 1 onto stack & restore 1 after the procedure call

Callee

```
int sum(int a, int b) {
    return a + b;
}
```

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn't see implementation of sum!)

Why didn't we save a0?

We don't need the original value after either procedure call. (no need to save)

Ex: nested procedures

- If a procedure calls another procedure, it needs to save its own return address
  - Remember that `ra` is caller-saved
- Example:

```
int coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

```
coprimes:
  addi sp, sp, -4
  sw ra, 0(sp)
  call gcd   ← overwrites ra   # overwrites ra
  addi a0, a0, -1 ]
  sltiu a0, a0, 1 ] restores ra
  lw ra, 0(sp)
  addi sp, sp, 4
  ret   ← jumping to ra   # where will the program return to?
```

Computing with large data structures:

- Suppose we want to write a procedure that finds the maximum value in an array a[].
  - The array is too large to be stored in registers
- How do we pass the array a[] as an argument?
  - Pass the base address (pointer to array) and the size of the array as arguments (len)

Just like we've been doing in C!

Ex:

```
# Finds maximum element in an
# array with size elements
int maximum(int *a, int size) {
   int max = 0;
   for (int i = 0; i < size;
    i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}

int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

```
main:  li a0, ages
       li a1, 5
       call maximum
       # max returned in a0

ages:  23
       4
       6
       81
       16
```

```
int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

```
# Finds maximum element in an
# array with size elements
int maximum(int *a, int size) {
   int max = 0;
   for (int i = 0; i < size;
    i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}

int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

```
maximum:
    mv t0, zero    # t0: i
    mv t1, zero    # t1: max
    j compare
loop:
    slli t2, t0, 2 # t2: i*4
    add t3, a0, t2
    # t3: addr of a[i]=base+i*4
    lw t4, 0(t3)   # t4: a[i]
    ble t4, t1, endif
    mv t1, t4      # max = a[i]
endif:
    addi t0, t0, 1 # i++
compare:
    blt t0, a1, loop

    mv a0, t1      # a0 = max
    ret
```

EXERCISES:

pseudo → non-P

Ex: call function_name → jal ra, function_name

non → pse

Ex: sub a4, zero, a2 → neg a4, a2

ex: add a4, zero, a3 → mv a4, a3

ex: addi x2, zero, 0xF2 (15)

0000 1111 0010 12 bits → +!

addi x3, zero, 0xFF4

1111 1111 0010 ← NEG: interp. as FFFF... F4 (12 bits)

The `s0–s11` registers are callee-saved registers. This means that: (select all correct answers)

- [ ] Prior to making a procedure call, all `s` registers must be saved on the stack.
- [ ] At the beginning of each procedure, all `s` registers must be saved to the stack.
- [x] All `s` registers that are overwritten within a procedure must first be saved to the stack.
- [ ] At the end of each procedure, all `s` registers must be restored from the stack.
- [x] At the end of each procedure, all modified `s` registers must be restored from the stack.

Submit View Answer 100.00%

You have infinitely many submissions remaining.

Which of the following statements are True in code that follows the RISC-V calling convention?

- [ ] Since `a` registers are caller-saved, you never need to save them onto the stack in your procedure implementation.
- [ ] All `a` registers must be saved to the stack prior to making a procedure call.
- [x] Only the `a` registers whose value you will need after returning from a procedure must be saved onto the stack.
- [ ] The `ra` register must be saved to the stack at the beginning of every procedure.
- [x] The `ra` register must be saved to the stack prior to making a procedure call.
- [ ] The `s` registers must be saved to the stack prior to making a procedure call.
- [ ] The `t` registers never need to be saved on the stack.
- [x] The `s` registers are guaranteed to be returned from a procedure call with their values unchanged.
- [ ] A procedure may modify the stack without restoring it to its original state.

Submit View Answer 100.00%

You have infinitely many submissions remaining.

Ex:

```
f:      // a0 a1
        // 4,6
        // (1)
        mv a2, a1   a2 = 6
        mv s0, a1   s0 = 6
        li a1, 2    a1 = 2
        // (2)
        call mul
        // (3)
        add a0, a0, a2
        add a0, a0, s0
        // (4)
        ret
        // (5)
```

mul (4,2)

a2 = 2
s0 = 2
a1 = 2

Given that `mul` follows the calling convention, which of the following register(s) are guaranteed to hold the same value at both points `(2)` and `(3)` in the code execution? Select all correct answers.

- [x] zero
- [ ] ra
- [ ] a0
- [ ] a1
- [ ] a2
- [ ] t0
- [ ] t1
- [x] s0
- [x] s1

Submit View Answer 100.00%

★ Ex: jal ra, label

0_0001010000_ _00000_ 1101111 (ra = 1, 7)

0_19-12_0

20 bit imm: 0....1 0 1 0 0 0 0 (80, 64)

320

Ex: lw s8, 4(sp)

rd rs1

sp = x2 = 06010

s8 = x 24 = 06011000

imm = 4 = 060100

0_0010100000_0_ 0

256 = 2^8

64 = 2^6

000000000100_ 00010_ 010_ 11000_ 0000011

Ex: `beq a5, t5, 12b` — rs1 rs2 7→28 — imm: 11100

a5 → x15 → 01111 ; t5 → x30 → 11110

Ex: `sw t3, 4(a0)` → im: 0100

t3 → x28 → 11100 (rs2) ; a0 → x10 → 01010 (rs1)

Ex: 0b0000000_00101_10000_101_10110_0110011 — rd

SRL — rs2 x5 t0 — x16 rs1 a6 — x22 s6

```
int quad_sum(int w, int x, int y, int z) {
   int a = sum(w, x);
   int b = sum(y, z);
   return sum(a, b);
}
```

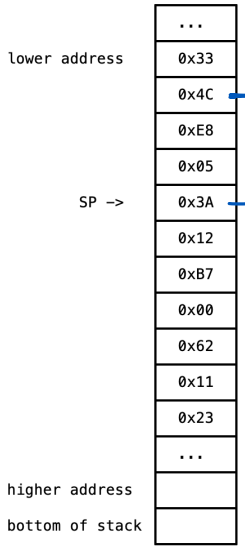
```
quad_sum:
  ??????????
  sw ra, 0(sp)
  sw s0, 4(sp)
  sw s1, 8(sp)

  mv s0, a2
  mv s1, a3
  call sum

  mv t0, a0
  mv a0, s0
  mv a1, s1
  mv s0, t0
  call sum

  mv a1, s0
  call sum

  lw ra, 0(sp)
  lw s0, 4(sp)
  lw s1, 8(sp)
  ???????????
    ←
  ret
```



ra jal ra+4 → ra = 48

previous func. call's ra

64+16+3 = 83

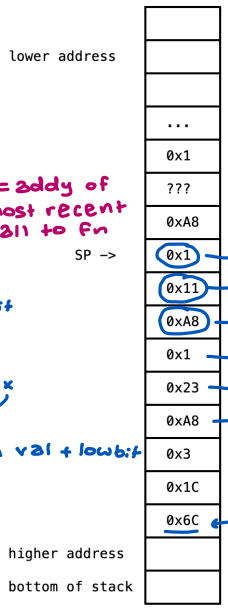
```
int fn(int x) {
    int lowbit = x & 1;
    int rest = x >> 1;
    if (x == 0) return 0;
    else return ???;
}
```

```
fn:
  addi sp, sp, -12
  sw s0, 0(sp)
  sw s1, 4(sp)
  sw ra, 8(sp)
  andi s0, a0, 1   s0 = lowbit
  srai s1, a0, 1   s1 = rest
    ← stop
yy:
  beqz a0, rtn
  mv a0, s1  ← a0 = rest
  jal ra, fn
  add a0, a0, s0 ← return val + lowbit

rtn:
  lw s0, 0(sp)
  lw s1, 4(sp)
  lw ra, 8(sp)
  addi sp, sp, 12
  jr ra
```

a0 = s1 — recursion

ra = addy of most recent call to fn



from previous

← most recent

x = s1 << 1 + s0 = 010_00110 0001 = 0100_0111 (4 7)

jal = ra - 4 = A4

jal points to next line

1st call (diff ra) — ra = jal - 4 = 6C - 4 = 68

a0 = 0x53 = 0b0101_0011 (64 16 3)

lowbit = 0000_0001

rest = 0010_1001

return fn( ) + 1

01001001.11111111 → 8

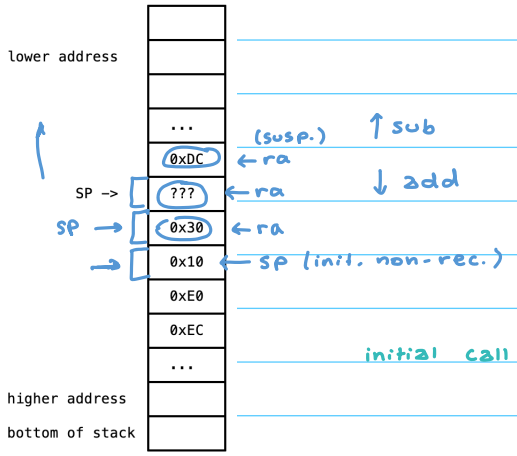
← 01001001

01001001 11111111 — 8 8

```
int get_remainder(int x, int y) {
   int difference = x - y;
   if (difference < 0) return x;
   else return ???;
}
```

```
get_remainder:
  sub t0, a0, a1
  bltz t0, rtn
  addi sp, sp, -4
  sw ra, 0(sp)
  mv a0, t0
  jal get_remainder
  lw ra, 0(sp)
  addi sp, sp, 4
rtn:
  jr ra
```

a0 = x
a1 = y
t0 = diff
a0 = x-y
a0 = 0x7
get_rem(diff, y)
← suspend.



initial call instruction: 0x30 - 4 (jal)
= 0x2c

getrem(0x67, 0x20)

0110_0111 — 64+32+7 = 103
0010_0000 — 32

rem = 7 — 0111 — 7 → a0

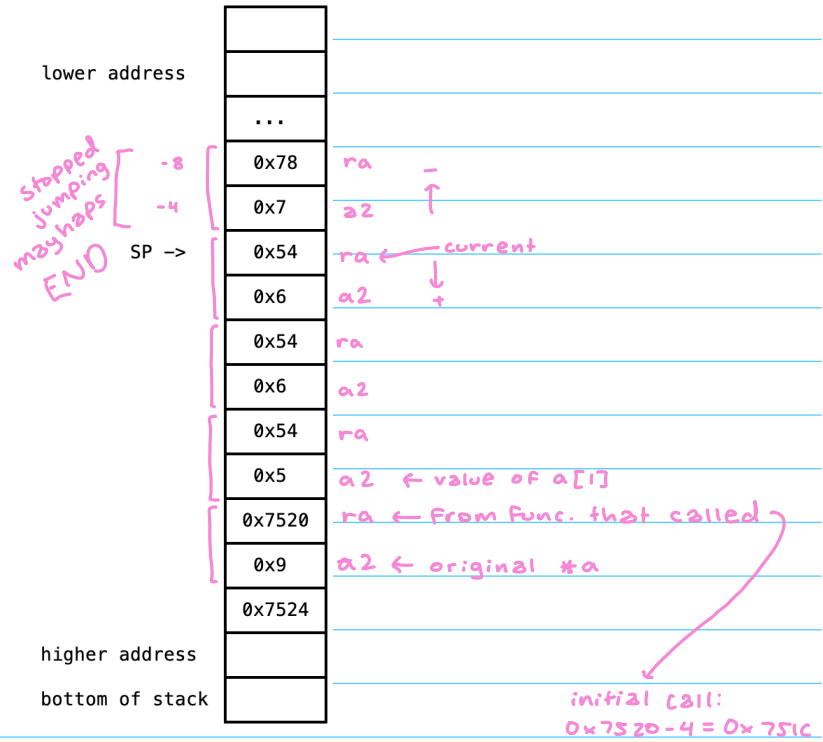
y = 32
00100000
2 0

The following C program computes the sum of all elements in an array.

```
int arraySum(int* a, int b){
    // int *a: pointer to array
    // int b: length of array
    if (b == 1) return a[0];
    else {
        return a[0] + arraySum(a+1, b-1);
    }
}
```

The corresponding assembly program is shown below:

```
                        a1 = b
arraySum:
     lw a2, 0(a0)       ← a2 = *a
     li a3, 1             a3 = 1
     beq a1, a3, end
     addi sp, sp, -8
     sw ra, 0(sp)       store ra
     sw a2, 4(sp)       store [0] addy
     addi a0, a0, 4     a0 + 4
     #REPLACE <---       b-1 : addi a1, a1, -1
     jal arraySum
     lw a2, 4(sp)
     add a2, a2, a0
     lw ra, 0(sp)
     addi sp, sp, 8
end:                    → Stopped!
mv a0, a2
ret
```



initial call:
0x7520 - 4 = 0x751c

b = len = how many times recursion happens = 5

[4/29/25] - rec 5

Pseudoinstructions unlocked!

MIT 6.191 (6.004) ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
li rd, liConstant	Load Immediate	reg[rd] <= liConstant
mv rd, rs1	Move	reg[rd] <= reg[rs1] + 0
not rd, rs1	Logical Not	reg[rd] <= reg[rs1] ^ -1
neg rd, rs1	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
j label	Jump	pc <= label
jal label call label	Jump and Link (with ra)	reg[ra] <= pc + 4 pc <= label
jr rs	Jump Register	pc <= reg[rs1] & ~1
jalr rs	Jump and Link Register (with ra)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
ret	Return from Subroutine	pc <= reg[ra]
bgt rs1, rs2, label	Branch > (Signed)	pc <= (reg[rs1] >, reg[rs2]) ? label : pc + 4
ble rs1, rs2, label	Branch ≤ (Signed)	pc <= (reg[rs1] <=, reg[rs2]) ? label : pc + 4
bgtu rs1, rs2, label	Branch > (Unsigned)	pc <= (reg[rs1] >, reg[rs2]) ? label : pc + 4
bleu rs1, rs2, label	Branch ≤ (Unsigned)	pc <= (reg[rs1] <=, reg[rs2]) ? label : pc + 4
beqz rs1, label	Branch = 0	pc <= (reg[rs1] == 0) ? label : pc + 4
bnez rs1, label	Branch ≠ 0	pc <= (reg[rs1] != 0) ? label : pc + 4
bltz rs1, label	Branch < 0 (Signed)	pc <= (reg[rs1] <, 0) ? label : pc + 4
bgez rs1, label	Branch ≥ 0 (Signed)	pc <= (reg[rs1] >=, 0) ? label : pc + 4
bgtz rs1, label	Branch > 0 (Signed)	pc <= (reg[rs1] >, 0) ? label : pc + 4
blez rs1, label	Branch ≤ 0 (Signed)	pc <= (reg[rs1] <=, 0) ? label : pc + 4

Note: *liConstant* is a 32-bit value.

For examples,

x=5

Before: addi a0, x0, 5

After li a0, 5

a1=a2

Before: addi a1, a2, 0

After mv a1, a2

Return "jump to ra"

Before: jal x0, 0(ra)

After: ret

Calling Convention

Why? We have limited shared registers so we don't want to mess up other processes if there are multiple going on.

```
main:
#assume a0 is the address of arr
a5 is index i
loop:
    (sth)
    slli a2, a5, 2 #a2 =4*i
    add a2, a2, a0 #a2 = address arr[i]
    lw a1, 0(a2)
    call mod
    add a1, a0, a1
    j loop CALLER
    (sth)
```

```
mod: CALLEE
(do sth)
mv a0, a2 #put return value in a0
ret
```

they are all same a0 (not like in C where we have scope)

pc: program counter

ra: return addy

pc, ra, jalr

pc = program counter = where we are

ra = return address = where should we go back if we "return"

```
caller:
    …
    addi sp, sp, -N
    sw a0,  0(sp)
    sw t0, 4(sp)
    sw ra, 8(sp)
pc → call callee #i.e. jal ra, callee
pc+4 lw a0,  0(sp)   ← ra: come back after func. call
pc+8 lw t0, 4(sp)
    lw ra, 8(sp)
    addi sp, sp, N
    …
```

```
pc → callee:
    …
    addi sp, sp, -N
    sw s0, 0(sp)
    …
    something
    …
    lw s0,  0(sp)
    addi sp, sp, N
pc → ret  # jal x0, 0(ra)
```

goes back to ra

Stack and sp "backup memory"

Stack = part of memory address where we use to store some values, going from higher to lower a

Stack pointer (sp) = point to the top of stack

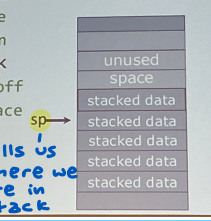
Two operations: push and pop

- Push: put something on top of stack

```
addi sp, sp, -4 # allocate space
sw a1, 0(sp)    # put element on
```

- Pop: take something off top of stack

```
lw a1, 0(sp)    # take element off
addi sp, sp, 4  # deallocate space
```



tells us where we are in stack

sw a0, 0(sp) write a0 to mem.

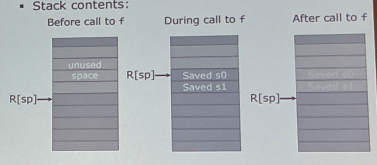
lw a0, 0(sp) move stack ptr back

Sample,

This function used s (callee saved) -> need to restore it back at the end

```
f:
  addi sp, sp, -8   # allocate 2 words (8 bytes) on stack
  sw s0, 0(sp)      # save s0
  sw s1, 4(sp)      # save s1
  addi s0, a0, 3
  li s1, 0x123456
  add s1, a1, s1
  or a0, s0, s1
  lw s0, 0(sp)      # restore s0
  lw s1, 4(sp)      # restore s1
  addi sp, sp, 8    # deallocate 2 words from stack
                    # (restore sp)
  ret
```

- Stack contents:



Who save what?

```
caller:
    …
    addi sp, sp, -N
    sw a0,  0(sp)
    sw t0, 4(sp)
    sw ra, 8(sp)
    call callee
    lw a0,  0(sp)
    lw t0, 4(sp)
    lw ra, 8(sp)
    addi sp, sp, N
    …
```

CALL →

```
callee:
    …
    addi sp, sp, -N
    sw s0, 0(sp)
    …
    something
    …
    lw s0,  0(sp)
    addi sp, sp, N
    ret
```

← RETURN

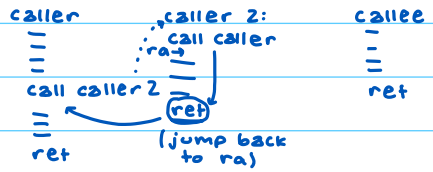
caller save = a's, t's, ra
- save in mem. & load back for safety

callee save = s's, sp
restore vals to og vals

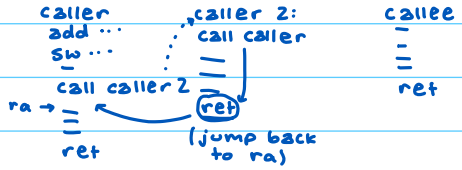
Why does caller need to store ra?

if no store:

Ex:



if store:



Ex: caller & callee

One can be both caller and callee

```
caller_and_callee:
    …
    addi sp, sp, -N
    sw a0,  0(sp)
    sw t0, 4(sp)
    sw ra, 8(sp)
    sw s0, 12(sp)
    …
    call callee
    …

    lw a0,  0(sp)
    lw t0, 4(sp)
    lw ra, 8(sp)
    lw s0, 12(sp)
    addi sp, sp, N
    ret
```

You don't need to save everything, only things that you modify (for s in callee) and use (for a and t in caller)

Blue = callee part
Red = caller part

```c
int function_A(int a, int b) {
  some_other_function();
  return a + b;
}
```

```
function_A:
    addi sp, sp, -8   -12
    sw a0, 8(sp)
    sw a1, 4(sp)
    sw ra, 0(sp)
    jal some_other_function
    lw a0, 8(sp)
    lw a1, 4(sp)
    add a0, a0, a1
    lw ra, 0(sp)
    addi sp, sp, 8
    ret
```

Does the above assembly obey the calling conventions?

- ○ Yes
- ◉ No

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

```c
int function_B(int a, int b) {
  int i = foo((a + b) ^ (a - b));
  return (i + 1) ^ i;
}
```

```
function_B:
    addi sp, sp, -4
    sw ra, 0(sp)
    add t0, a0, a1
    sub a0, a0, a1
    xor a0, t0, a0
    jal foo
    addi t0, a0, 1
    xor a0, t0, a0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

Does the above assembly obey the calling convention?

- (•) Yes
- ( ) No

```c
int function_C(int x) {
  foo(1, x);
  bar(2, x);
  baz(3, x);
  return 0;
}
```

```
function_C:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv a1, a0
    li a0, 1
    jal foo
    li a0, 2
    jal bar
    li a0, 3
    jal baz
    li a0, 0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

sw & lw

Does the above assembly obey the calling convention?

- ( ) Yes
- (•) No

Submit View Answer 100.00%

```c
int function_D(int x, int y) {
  int i = foo(1, 2);
  return i + x + y;
}
```

```
function_D:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv s0, a0
    mv s1, a1
    li a0, 1
    li a1, 2
    jal foo
    add a0, a0, s0
    add a0, a0, s1
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

-12

sw s0,
sw s1
lw s0,
lw s1

need to store s0 & s1 to stack then restore back

Does the above assembly obey the calling convention?

- ( ) Yes
- (•) No

Submit View Answer 100.00%

Ex: trace shows countBits(10); stopped before rtn label

```
countBits:
        beqz a0, rtn
        addi sp, sp, -8
        sw s0, 4(sp)
        sw ra, 0(sp)
        srli s0, a0, 1
        mv a0, s0
        jal ra, countBits
        addi a0, a0, 1
        lw ra, 0(sp)
        lw s0, 4(sp)
        addi sp, sp, 8
rtn:    jr ra
```

```
/* Counts the minimum numnber of bits required to express a number */
int countBits(unsigned x) {
  unsigned y;
  if (x == 0) return 0;
  else {
    /* Shifts x right by 1 bit. */
    y = x >> 1;
    return countBits(y) + 1;
  }
}
```

pc

allocate 8 bytes & store s0, ra

y = x >> 1

count bit y

ra

deallocate & restore s0, ra

stop →

x = 10

a0 >> 1

= prev a0 >> 1

a0 = s0

call self

ra is updated

0x240
244
248
24C
250

b = s0

0x93	
0x240	ra
0x1	s0 = 1
A = 0x240	ra
B 0x2	s0 =
SP → 0x240	ra
0x5	s0 = 5 = a0 >> 1
0x1108	ra
0x37	s0

same

recursive calls to countBits

5 >> 1 = 101 >> 1 = 10 = 0x2

initial call to countBits

value at...

a0 (return val.) 2

s0 2

sp cannot tell

pc

```
binary_search:  addi sp, sp, -8
                sw ra, 4(sp)
                sw s0, 0(sp)
                mv s0, a2
                beq a1, a2, done
                add t0, a1, a2
                srli t0, t0, 1
                slli t1, t0, 2
                add t1, a0, t1
                lw t1, 0(t1)
if:             bge a3, t1, else
                mv a2, t0
                j recurse

else:           addi t0, t0, 1
                mv a1, t0

recurse:        call binary_search
                mv s0, a0

done:           mv a0, s0
                lw s0, 0(sp)
                lw ra, 4(sp)
L1:             addi sp, sp, 8
                ret
```

s0 = end

t1 = arr[mid]

else arr[mid]

← ra 0xC4

← C8

CC

← stop

D4

pc

D8

a0 a1 a2 a3

```
unsigned binary_search(int* arr, uint32_t start, uint32_t end, int element) {
    if (start == end) {
        return end;
    }
    mid = (start + end) / 2;
    if (element < arr[mid]) {
        end = mid;
    } else {
      start = mid + 1;
    }
    return binary_search(arr, start, end, element);
}
```

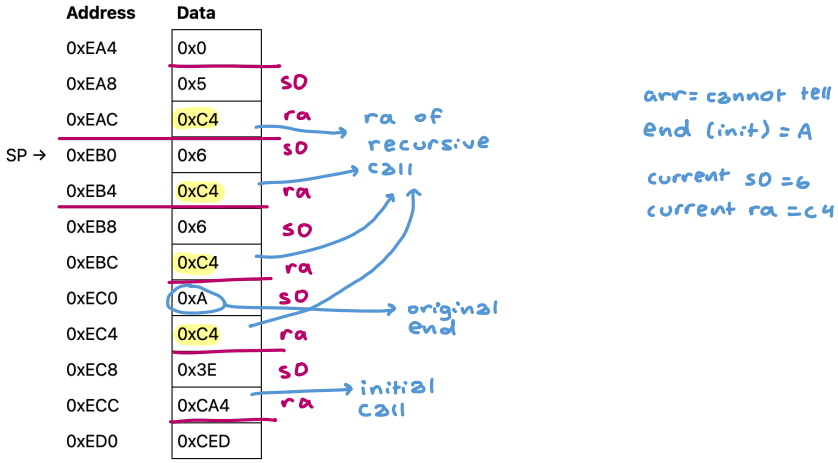
how is it a3?

How many words will be written to the stack before the program makes each recursive call to the function binary_search? 2

Submit View Answer 100.00%

You have infinitely many submissions remaining.

The program's initial call to function `binary_search` occurs outside of the function definition via the instruction `call binary_search`. The program is interrupted *during a recursive call* to `binary_search`, *just prior* to the execution of `addi sp, sp, 8` at label `L1`. The diagram below shows the contents of a region of memory. All addresses and data values are shown in hex. **The current value in the `sp` register is `0xEB0` and points to the location shown in the diagram.**



5.4) Problem 5.4

What is the value in register `s0` right when the execution of `binary_search` is interrupted? Write CAN'T TELL if you cannot tell. 6

Submit View Answer 100.00%

You have infinitely many submissions remaining.

What is the value in register `ra` right when the execution of `binary_search` is interrupted? Write CAN'T TELL if you cannot tell. c4

Submit View Answer 100.00%

You have infinitely many submissions remaining.

5.5) Problem 5.5

What is the hex address of the `call binary_search` instruction that made the *initial call* to `binary_search`?

ca0

Submit View Answer 100.00%

You have infinitely many submissions remaining.

5.6) Problem 5.6

What is the hex address of the `ret` instruction? D8

Submit View Answer 100.00%

[5/6/25] - REC

STACK DETECTIVE:

- find the assembly code corresponding w/ assembly

```c
uint32_t foo(uint32_t x) {
    if (x == 0) {
        return 0;
    } else {
        if (x & 1 == 1) {
            return bar(x >> 1); // diff
        } else {
            return foo(x >> 1); // diff
        }
    }
}
```

```c
uint32_t bar(uint32_t x) {
    if (x == 0) {
        return 0;
    } else {
        if (x & 1 == 1) {
            return 1; // diff
        } else {
            return bar(x >> 1); // diff
        }
    }
}
```

```
foo:
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    beqz a0, foo_end
    andi a1, a0, 1
L1:
    srli s0, a0, 1
    beqz a1, foo_else
    mv a0, s0 # diff
    call bar # diff
    j foo_end
foo_else:
    mv a0, s0 # diff
    call foo # diff
foo_end:
    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret
```

```
bar:
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    beqz a0, bar_end
    andi a1, a0, 1
L2:
    srli s0, a0, 1
    beqz a1, bar_else
    li a0, 1 # diff
    j bar_end
bar_else:
    mv a0, s0 # diff
    call bar # diff
bar_end:
    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret
```

a1 stores last bit and we know func. goes to foo-else ∴ a1 last bit must be 0!

s0 = a0 >> 1

a0 = s0 << 1

← call 0xC00 0xC04 ↑ -4

call foo 0xC0C 0xC10 ↑

← stop here pc ra-6(4)

we call this & don't modify it!

← call 0xC48 ← ra

Pointers	Address	Data	
	0xEA8	???	
	0xEAC	???	
s0 sp ->	0xEB0	0x1	current values
ra	0xEB4	0xC4C	current values
s0	0xEB8	0x2	same addy! so must be bar (bar call bar, not foo call bar)
ra	0xEBC	0xC4C	same addy! so must be bar (bar call bar, not foo call bar)
s0	0xEC0	0x4	
ra	0xEC4	0xC04	← diff ra
s0	0xEC8	0x9	← stored: a0 >> 1
ra	0xECC	0xC10	← diff ra!
s0	0xED0	0x13	
ra	0xED4	0xB08	← init call 0xB08 - 4 = 0xB04

pc = addy of instruction

s0 = 0x9 = 0b1001 << 1

= 0b10010 = 0x12

- find starting point (jumps)

The following line of code is run. At various points throughout the program (denoted `TIME POINT X`), the values in certain memory locations and registers are saved. Some of these values are shown in the table on the next page. Using the code below and those values, fill in the missing cells in the table.

```
jal ra, log_a_x    # <--------- TIME POINT 0 (after this line is executed)
```

```
 1 ilog2: # produce ilog2 of a0
 2   addi sp, sp, -4
 3   sw ra, 0(sp)
 4   addi t1, zero, 1
 5   blt t1, a0, ilog_else
 6   addi a0, zero, 0
 7   beq zero, zero, ilog_ret
 8 ilog_else:
 9   srli a0, a0, 1
10   jal ra, ilog2 ← calls itself (same ra)
11   addi a0, a0, 1
12 ilog_ret:
13   lw ra, 0(sp)
14   addi sp, sp, 4
15   jalr zero, 0(ra)
16
17 idiv: #produce idiv of a0/a1
18   addi sp, sp, -4
19   sw ra, 0(sp)
20   addi t1, zero, 0
21   bge a0, a1, idiv_else
22   addi a0, zero, 0
23   beq zero, zero, idiv_ret
24 idiv_else:
25   sub a0, a0, a1
26   jal ra, idiv
27   addi a0, a0, 1
28 idiv_ret:
29   lw ra, 0(sp)
30   addi sp, sp, 4
31   jalr zero, 0(ra)
32
33 log_a_x: #compute the log of a0 in base a1
34   addi sp, sp, -12 ]
35   sw ra, 0(sp)      | 3 words
36   sw s0, 4(sp)      |
37   sw s1, 8(sp)     ]
38   add s0, a0, zero  # s0 = a0   <--------- TIME POINT 1 (after line 38 executed)
39   addi a0, a1, 0    a0 = a1
40   jal ra, ilog2 ]
41   addi s1, a0, 0      sp remain same
42   addi a0, s0, 0    #   <--------- TIME POINT 2 (after line 42 executed)
43   jal ra, ilog2 ]
44   addi a1, s1, 0
45   jal ra, idiv ]
46   lw s1, 8(sp)      #   <--------- TIME POINT 3 (after line 46 executed)
47   lw s0, 4(sp)
48   lw ra, 0(sp)
49   addi sp, sp, 12
50   jalr zero, 0(ra)  #   <--------- TIME POINT 4 (after line 50 executed)
```

Complete this table for Problem 5, using the code on the previous page.

Address	TIME POINT 0	TIME POINT 1	TIME POINT 2	TIME POINT 3	TIME POINT 4
0x3fc93f00	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
0x3fc93f04	0xa5a5a5a5	0xa5a5a5a5	0xa5a5a5a5	0x42001620	0x42001620
0x3fc93f08	0xa5a5a5a5	0xa5a5a5a5	0xa5a5a5a5	0x42001620	0x42001620
0x3fc93f0c	0x00000004	0x00000004	0x00000004	0x42001620	0x42001620
0x3fc93f10	0x000007c2	0x000007c2	0x000007c2	0x42001620	0x42001620
0x3fc93f14	0x00000123	0x00000123	0x00000123	0x42001650	0x42001650
0x3fc93f18	0xffffffff	0xffffffff	0x42001620	0x42001650	0x42001650
0x3fc93f1c	0x00000123	0x00000123	0x42001620	0x42001650	0x42001650
0x3fc93f20	0x420165b0	0x420165b0	0x4200167c	0x42001690	0x42001690
0x3fc93f24	0x3fc91000	0x4201540a	0x4201540a	0x4201540a	0x4201540a
0x3fc93f28	0x3fc91000	0x0000000f	0x0000000f	0x0000000f	0x0000000f
0x3fc93f2c	0x00000011	0x00000001	0x00000001	0x00000001	0x00000001
0x3fc93f30	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x3fc93f34	0x00000111	0x00000111	0x00000111	0x00000111	0x00000111

Register	TIME POINT 0	TIME POINT 1	TIME POINT 2	TIME POINT 3	TIME POINT 4
a0	0x0000008a	0x0000008a	0x000000 8a	3	3
a1	0x00000005	0x00000005	0x00000005	2	2
s0	0x0000000f	0x000000 8a	0x000000 8a	8a	0xf
s1	0x00000001	0x00000001	2	1	1
ra	0x4201540a	0x4201540a	0x 4200167c	0x 42001690	0x 4201540a
t1	0x0000000a	0x0000000a	0x00000001	0	0
sp	0x3fc93f30	0x3fc93f24	0x3fc93f24	0x3fc93f24	0x3fc93f30

TIPS TO MYSELF:

- pace yourself!
- understand full question before diving in
- pay attention to types
  - signed vs. unsigned
  - hex, decimal, binary
  - pointer?
  - char vs. int
  - 0 vs. \0 null ptr
  - size (8 bits? 32?)
- memory? sw, lw #4 ... array? i, no multiplier