CS 107 Final Review Session

Some Assembly Required

mov operand: scaling, indexing, displacement

mov 0x4(%eax, %ebx, 2), %ecx => %ecx = *((%eax + %ebx*2) + 0x4)

%eax -> base (default 0)

%ebx -> index (default 0)

2 -> scale (default 1)

0x4 -> displacement (default 0)

Data always flows from left to right!

mov examples

mov $1, (%eax, %ebx) => *(%eax + %ebx * 1 + 0) = 1

Scale and displacement defaulted

mov $1, (, %ebx, 4) => *(0 +%ebx * 4 + 0) = 1

Base and displacement defaulted

movl $1, (%rax, %rcx, 8) => *(%rax + %rcx * 8) = 1

Displacement defaulted

lea vs. mov (Lecture 12, Slide 32)

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

Calling Convention Summary

Arguments (in order): %rdi, %rsi, %rdx, %rcx, %r8, %r9, (stack)

Return value: %rax

Stack pointer: %rsp

Instruction pointer: %rip

On the x86-64 guide and reference sheet! (You will have the reference sheet during the exam).

Conditional Execution

Arithmetic/comparison instructions set flags: (cmp, xor, test, etc.).

Instructions with condition codes will check those flags and conditionally perform an operation.

C Code

```
int if_then(int param1) {
    if (param1 == 6) {
        param1++;
    }

    return param1 * 2;
}
```

What does this assembly code translate to?

```
0000000004004d6 <if_then>:
  4004d6:       cmp    $0x6,%edi
  4004d9:       jne    4004de
  4004db:       add    $0x1,%edi
  4004de:       lea    (%rdi,%rdi,1),%eax
  4004e1:       retq
```

What order does cmp do things?

In the previous slide we saw cmp $6, %edi.

For jne, %edi == 6 and 6 == %edi are the same.

What if we have a jle for example? Is it %edi <= 6 or 6 <= %edi?

The best way to think about it is that cmp <op1>, <op2> will perform <op2> - <op1>.

So jle will check if ZF=1 or SF!=OF (so it'll check if the cmp result is <= 0).

So <op2> - <op1> <= 0 => <op2> <= <op1> => %edi <= 6.

Conditional Jumps (Lecture 13, Slide 35)

There are also variants of **jmp** that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero (ZF = 1)
`jne` *Label*	`jnz`	Not equal / not zero (ZF = 0)
`js` *Label*		Negative (SF = 1)
`jns` *Label*		Nonnegative (SF = 0)
`jg` *Label*	`jnle`	Greater (signed >) (SF = 0 and SF = OF)
`jge` *Label*	`jnl`	Greater or equal (signed >=) (SF = OF)
`jl` *Label*	`jnge`	Less (signed <) (SF != OF)
`jle` *Label*	`jng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`ja` *Label*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *Label*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb` *Label*	`jnae`	Below (unsigned <) (CF = 1)
`jbe` *Label*	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Common If-Else Construction (Lecture 13, Slide 42)

If-Else In C

```
if (num > 3) {
    x = 10;
} else {
    x = 7;
}

num++;
```

If-Else In Assembly

```
Test
Jump past if-body if test fails
If-body
Jump past else-body
Else-body
Past else body
```

Common For Loop Construction (Lecture 13, Slide 62)

C

```
for (init; test; update) {
        body
}
```

C Equivalent While Loop

```
init
while(test) {
    body
    update
}
```

For Loop Assembly

```
Init
Jump to test
Body
```

```
Update
Test
Jump to body if success
```

FP stands for Free Pizza

FP Motivation

Goal: Represent real numbers (decimals) on a computer.

Idea: Use scientific notation (can potentially represent large numbers very compactly; 1.0*10^300 vs 1000000000000….)!

But computers work well in binary, so use base-2 scientific notation instead of base-10.

We'll work with 32-bit (single-precision) FP numbers, but the same reasoning holds for FP numbers of any other bitwidth.

Fixed Point (Lecture 10, Slide 26)

Idea: Like in base 10, let's add binary decimal places to our existing number representation.

1	0	1	1	.	0	1	1
8s	4s	2s	1s		1/2s	1/4s	1/8s

Pros: arithmetic is easy! And we know exactly how much precision we have.

FP Motivation

Scientific notation number: -2.3*10^20

Three parts:

+/-: sign

2.3: fraction

20: exponent

(Let’s fix our base so we don’t have to store it, we’ll be using base-2 for FP).

FP Representation

Same thing in FP!

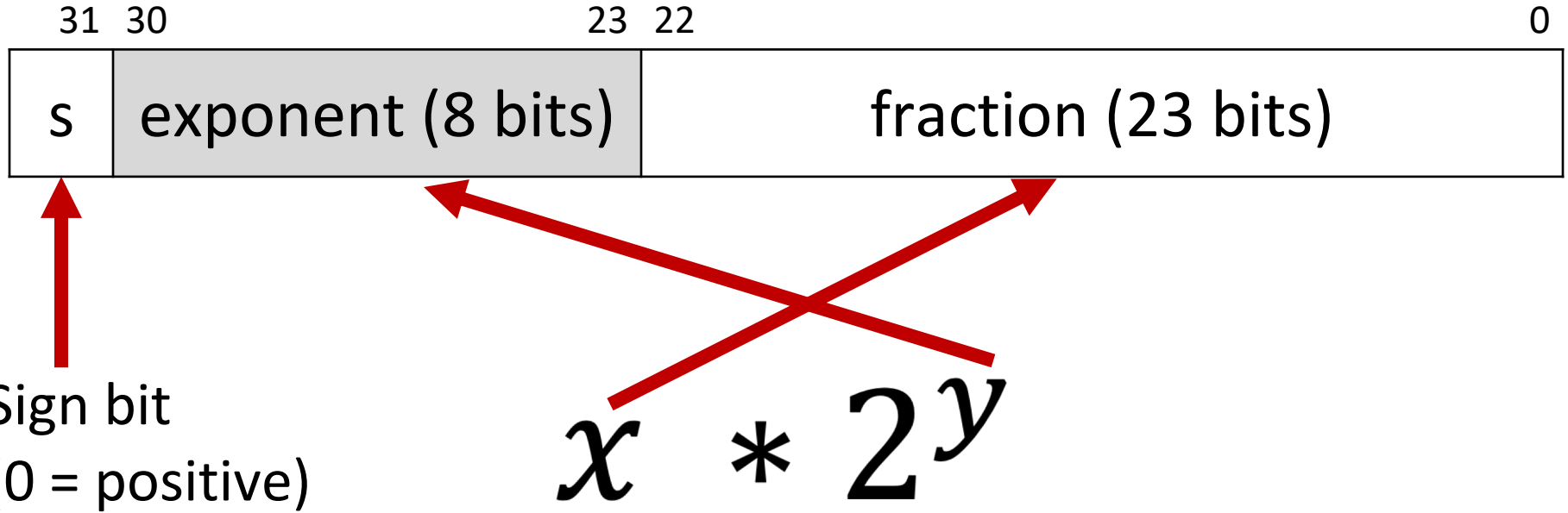
Three parts:

Sign bit - 0 => positive, 1 => negative

Exponent - Same role as before

Fraction/Mantissa/Significand - Same role as before

IEEE Single Precision Floating Point (Lecture 10, Slide 35)



FP Exponents: Summary

Let’s say we have n bits for our exponent E. To get the true value of our exponent, we compute:

2^(E - B)

E - The value of our exponent as an unsigned number

B - Bias = ((2^n) / 2) - 1 (So for IEEE 32-bit FP, n = 8, so B = ((2^8) / 2) - 1 = 127)

Example:

E = 10010001 => our true exponent will be E - B = 145 - 127 = 18.

FP Fractions: Summary

Let F be our fraction.

Idea: Let’s just assume that we always write our number as +/- 1.F * 2^(E - B).

We can get 1 additional bit of precision.

This is for normalized numbers, there are other types!

Different types of FP numbers!

Type of number	**Exponent (0 <= E <= 255)**	**Fraction (F)**	**Sign (S)**	**Value**
Zero	0	0	Any	+/- 0
Denormalized	0	!= 0	Any	+/- 0.F * 2^(-B + 1)
Normalized (regular)	0 < E < 255	Any	Any	+/- 1.F * 2^(E - B)
Infinity	255	0	Any	+/- Inf
Not a Number (NaN)	255	!= 0	Any	NaN

Can I get some Pointers on the Final?

Generics: Motivation

How do we write a search function in C (return pointer to found element or NULL otherwise).

```
int* search_int(int* haystack, int needle, size_t sz);

char* search_char(char* haystack, char needle, size_t sz);

long* search_long(long* haystack, long needle, size_t sz);
```

…

Strings? Custom types?

This is tedious: can we just have one function?

Generics: Motivation

Idea: Want to be able to store data of any type

=> Define void* to be a pointer to any type!

Attempt 1: void* search(void* haystack, void?? needle, size_t nelems);

What should the type of our needle be? Our needle can be any type as well, so we should let the user pass it in as a pointer to the needle so it can be a void* (a variable can't have type void).

Attempt 2: void* search(void* haystack, void* needle, size_t nelems);

Generics: Motivation

Q: This will type check, but now we don’t know how big each element is (remember, we can’t do haystack[i] if haystack is a void*!), so we can’t iterate through haystack. How do we solve this?

A: User should tell us!

Attempt 3: void* search(void* haystack, void* needle, size_t width, size_t nelems);

Q: Almost there! What does == mean on void*?

A: User should tell us!

Generics: Motivation

Attempt 4: void* search(void* haystack, void* needle, size_t width, size_t nelems, int (*cmp)(void*, void*));

Looks good to me! (cmp is a custom comparison function:

> 0 => greater than

< 0 => less than

== 0 => equals).

As a side-note, technically we could define the comparison function however we wanted. However, the above definition is commonly how comparison functions are structured. We take in two elements and we return an ordering (less than, greater than, or equal to).

Let’s see how it looks! (Demo)

Practice Makes Perfect!

Final Extra Practice

Consider the following x86-64 code output by gcc using the settings we use for this class (-Og):

```
<ham>:
  mov    (%rdi),%eax
  lea    (%rax,%rax,2),%esi
  add    %esi,%esi
  mov    $0x0,%ecx
  imul   $0x31,%esi
  jmp    L1

L3:
  lea    (%rcx,%rax,1),%edx
  movslq %edx,%rdx
  mov    %esi,(%rdi,%rdx,4)
  add    $0x2,%eax
  jmp    L2

L4:
  mov    %ecx,%eax

L2:
  cmp    $0x9,%eax
  jle    L3
  add    $0x3,%ecx

L1:
  cmp    $0x9,%ecx
  jle    L4
  mov    $0xa,%eax
  retq
```

Fill in the Blanks!

```
eliza[3] = _____________ * burr[0];

for (int i = 0; i < ____________; i+=____________) {

      for (int j = ____________; j < ____________; j+=____________) {

            burr[____________] = eliza[0]*eliza[1]*eliza[2]*eliza[3];


      }

}

   return ____________;

}
```

```
int ham(int *burr) {
    int eliza[4];
    eliza[0] = 7;
    eliza[1] = 7;
    eliza[2] = 1;


    eliza[3] = 6 * burr[0];
    for (int i = 0; i < 10; i+= 3) {

      for (int j = i; j < 10; j+=2) {

            burr[i + j] = eliza[0]*eliza[1]*eliza[2]*eliza[3];

      }

    }



    return 10;

}
```

Final Exam #3, Problem 1: Setup

A normalized IEEE 32-bit float is stored as the following bit pattern:

N EEEEEEEE SSSSSSSSSSSSSSSSSSSSSSS

where N is the sign bit (1 if negative), E is the 8-bit exponent (with a bias of 127), and S is the 23-bit significand, with an implicit leading “1”.

Beyoncé and Jay-Z set up two bank accounts, one for each of their twins, but instead of using IEEE 32-bit floats to represent the account balances, they created a new made-up “minifloat” as a fittingly “mini” way to represent the account balances. A minifloat is structured the same as an IEEE float, but is 8 bits instead of 32, with 1 sign bit, 4 exponent bits, and 3 mantissa bits, and an exponent bias of 7.

Final Exam #3, Problem 1

(a) (3pts) Twin A's account balance is stored as the minifloat **0 1110 010**. What is the corresponding (simplified) decimal number that this represents?

(b) (3pts) Twin B's account balance is stored as the minifloat **0 1011 001**. What is the corresponding (simplified) decimal number that this represents?

(c) (4pts) After much thought, they decide to merge the two bank accounts together. The bank correctly performs minifloat addition to get the summed balance of **0 1110 011**, but this seems off to Beyoncé and Jay-Z. What is the corresponding (simplified) decimal number that this represents? Why is this the resulting total balance? Why do or don't you predict that this would have been an issue if they had used regular IEEE 32-bit floats instead?

Problem 1: Floating Point

(a) 160

(b) 18

(c) 176 – we lose precision because minifloat does not have enough bits to store the entire sum of the number. This likely would not have been an issue with IEEE 32-bit floats, since there are many more bits to store numbers with more precision.

Practice Final #2, Problem 2

2a) The function `extract_min` extracts the smallest element from a generic array according to a comparison function. The client supplies an address as the first argument; the smallest element is written to this address and that element is removed from the array. As an example, `extract_min` on the array `{8, 5, 19, 11}` with ordinary integer comparison writes `5` to the client's address, changes the array contents to `{8, 19, 11}`, and decrements the number of elements from 4 to 3. The order of the remaining array elements is preserved and the array's storage is not resized. Assume the array has at least one element.

The provided `find_min` function returns a pointer to the smallest element in a generic array. The use of `find_min` below is correct and you can assume `find_min` is correctly implemented. Complete the implementation of `extract_min` below.

```
void extract_min(void *addr, void *base, size_t *p_nelems, size_t width, int (*cmp)(const void *, const void *)){

	void *min = find_min(base, *p_nelems, width, cmp);
```

2a)

```
void extract_min(void *addr, void *base, size_t *p_nelems, size_t width, int (*cmp)(const void *, const void *)){

	void *min = find_min(base, *p_nelems, width, cmp);

	memcpy(addr, min, width);

	size_t bytes_to_move = (*p_nelems * width) – ((char *)min + width – (char *)base);

	memmove(min, (char *)min + width, bytes_to_move);

		(*p_nelems)--;
}
```

Exam 2 Problem 1: C-strings

The function `substring (char *input, size_t pos, size_t len)` is intended to trim `input` to the sequence of characters starting at `pos` and continuing for `len` characters. If `pos` or `len` is out of range for the input string, the behavior is undefined. The implementation below is buggy:

```
void buggy_substring(char *input, size_t pos, size_t len) {
	input += pos;
	input[len] = '\0';
}
```

You write a program to test the function:

```
int main(int argc, char *argv[]) {
	char name[16];
	strcpy(name, "Tessier-Lavigne"); buggy_substring(name, 3, 2); printf("%s\n", name);
	return 0;
}
```

1a) The above test program is intended to print `"si"`. What is printed instead?

1b) You change the prototype of `substring` to take the input string by reference. Implement the body of the function so that it works correctly and is compatible with the new prototype.

```
void substring(char **p_input, size_t pos, size_t len) {
```

1c) Complete the program started below to test the `substring` function. Compute the substring of `name` starting at position 3 of length 2 and print it.

```
int main(int argc, char *argv[]) {
	char name[16];
	strcpy(name, "Tessier-Lavigne");
```

1a) `"Tessi"`

1b)

```
void substring(char **p_input, size_t pos, size_t len) {

         (*p_input) += pos;

          (*p_input)[len] = '\0';

}
```

1c)

```
int main(int argc, char *argv[]) {

              char name[16];

       strcpy(name, "Tessier-Lavigne"); char *ptr = name; substring(&ptr, 3, 2); printf("%s\n", ptr);


}
```

Practice Exam 2 Problem 5: Runtime stack

The function `concat` is a poorly-coded attempt at concatenation. Not only does it declare the result array with an arbitrary small size, it returns the address of that stack-allocated array. Review the C source below on the left and the generated assembly on the right.

```
char *concat(const char *s, const char *t) concat{

       char buffer[10];
       char *result = buffer;


              strcpy(result, s);
       strcat(result, t);
       return result;
}

int main(int argc, char *argv[]){

       char *title = concat(argv[1], argv[2]);
       printf("%s\n", title);
       return 0;

}
```

```
concat:
    push   %rbx
    sub    $0x10,%rsp
    mov    %rsi,%rbx
    mov    %rdi,%rsi
    mov    %rsp,%rdi
    callq  <strcpy>
    mov    %rbx,%rsi
    mov    %rsp,%rdi
    callq  <strcat>
    mov    %rsp,%rax
    add    $0x10,%rsp
    pop    %rbx
    retq

main:
    sub $0x8,%rsp
    mov %rsi,%rax
    mov 0x10(%rsi),%rsi
    mov 0x8(%rax),%rdi callq <concat>
    mov %rax,%rsi
    mov $0x401850,%edi mov $0x0,%eax callq <printf>
    mov $0x0,%eax
    add $0x8,%rsp
    retq
```

5a) Running `./program LelandStanfordJunior University` crashes during execution. Circle the assembly instruction at which the crash occurs and explain why this instruction fails to execute.

5b) Seeing the array is too small, the implementer enlarges the size of the `buffer` declared within `concat` to:

```
char buffer[4096];
```

After this change, running `./program LelandStanfordJunior University` now prints `LelandStanfordJuniorUniversity`. Explain why the program appears to work correctly despite the fact that `concat` returns the address of a stack-allocated array.

5c) Not wanting to squander memory, they then change the declaration of `buffer` to the proper size:

```
char buffer[strlen(s) + strlen(t) + 1];
```

After this change, `./program LelandStanfordJunior University` now prints garbage. Explain how the change in sizing the array leads to this change in behavior.