Welcome back$_3$ to CS429H!

Week 3

Ed memes of the week:

pragma once shame on me, pragma twice shame on you

I forgot cri means something else lol.

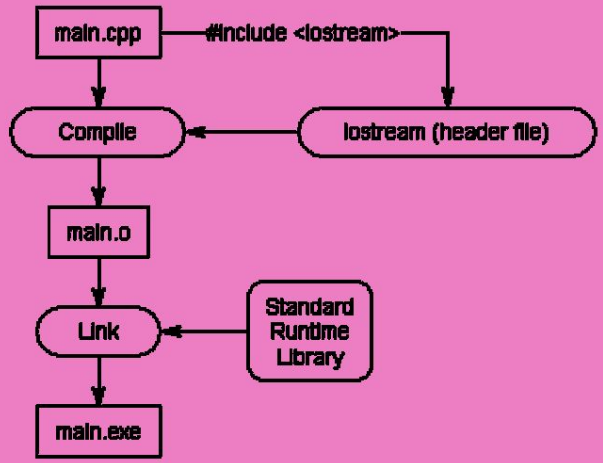
```
t2 ... pass [0.00]
simple_func_test ... pass [0.00]
t3 ... pass [0.00]
t0 ... womp womp womp womp womp womp [0.21]
t1 ... pass [0.00]
testing-partial ... pass [0.00]
```

Discord meme of the week:

We should link up

To:

From:



Be quiet during lecture pls

Questions on lecture content?
Or about cats?

Quiz everyone say YAY!

Poll

```
uint64_t entry =
loadElf("feedback.txt");
```

How was the quiz?

A. easy
B. mostly fine
C. mostly fine, but not enough time
D. too hard, but finished mostly in time
E. too hard and not enough time
F. too hard regardless of time

Stress

- 429H is not an easy class
  - Lots of new materials
  - Unfamiliar programming environments
  - Fast, often relentless pace
- Struggling in this course is normal
  - There will be times you won't know the answer of the solution
  - This is expected—we want we everyone to succeed, but the only way we can help is if you ask for it
- If you find yourself overly overwhelmed or spending more time on this class than you think you should be, please reach out to Dr. Gheith or the TAs
  - We can help out as far as the class goes
  - We can provide other resources where we are not able to help

Mental health resource available at UT

P2 Postmortem

- Grades will be released by Eventually
- Correctness
  - Not bad!
- Test cases
  - Pretty good!
- Code quality
  - Good!
- Reports
  - Okay!
- We are still working on benchmarking for the fastest interpreter

A Note on Regrade Requests

- Please do not submit regrades until after we have released grades for an assignment
- We will by default take the last commit before the soft deadline (which is not known)
- Private Ed post in Regrades category - be sure to include which project and which commit hash you want to be graded

P3

Upload your .s files

- Easier for other people to debug using your test case!
- https://drive.google.com/drive/folders/1UxiOZpXiQgT3oONYKxmL5ihx3lJJshRV?usp=sharing

- Other useful tips for debugging:
  - Run "`git clone git@gheith.csres.utexas.edu:cs429h_s24_p3__tests `" to download matrix test cases
  - If there's no .s file available, use objdump to translate .arm files into human-readable assembly code using the following command
    - `~gheith/public/gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu-objdump -d csid.arm`

Post-index/pre-index/unsigned offset

- Three ways to use a load/store instruction
  - Why might these be useful
- What do they do?
  - Post: load(x++);
  - Pre: load(++x);
  - Offset: load(x + 1);

What is going on in ~~GDC~~ ORR?

TAs: “Why did you include this instruction?”

Gheith: “Oh, because it’s nasty”

It’s a difficult instruction because we have a 4 byte instruction, but we would like to be able to encode an 8 byte immediate value. ORR gives 13 bits to encode up to 64 bits.

What is going on in ~~GDC~~ ORR?

We start with a bit pattern (which can be of size 2, 4, 8, 16, 32, or 64 bits). This bit pattern starts with some amount of 0s, then ends with some amount of 1s (at least one of each).

Bit pattern: 0… 1…

We encode this bit pattern by storing the overall length of the bit pattern and the number of 1s in the bit pattern (we actually store the number of 1s minus 1).

Then we concatenate copies of this bit pattern however many times it takes to fill up a 32 bit or 64 bit immediate.

What is going on in ~~GDC~~ ORR?

After we have our the value from the previous slide, we right-rotate the whole thing by an amount that is encoded.

00101 right rotated by 2 is 01001

Why is the encoding... like that?

13 bits could theoretically represent 8192 13-bit values, but this encoding only allows us to represent 5334 immediates (but spread over a different range).

Somebody used statistical analysis to show this encoding allowed for immediate values that were popular in programs.

Fun fact: Dr. Gheith used to work for ARM, and saw the design document in which usage patterns were analyzed and this scheme was developed. But it's confidential :(

Further ORR Resources

High level explanation:
https://devblogs.microsoft.com/oldnewthing/20220802-00/?p=106927

Helpful details in the logical immediates section:
https://dinfuehr.github.io/blog/encoding-of-immediate-values-on-aarch64/

Poll

Where did the names “Big Endian” and “Little Endian” originate?

A. The Grapes of Wrath
B. Gulliver’s Travels
C. Lord of the Flies
D. Great Expectations
E. The Adventures of Huckleberry Finn

In computing, **endianness** is the order in which bytes within a word of digital data are transmitted over a data communication medium or stored (upwardly) in computer memory, counting only byte significance compared to earliness. Endianness is primarily expressed as **big-endian** (BE) or **little-endian** (LE), terms introduced by Danny Cohen into computer science for data ordering in an Internet Experiment Note published in 1980.[1] The adjective *endian* has its origin in the writings of 18th century Anglo-Irish writer Jonathan Swift. In the 1726 novel *Gulliver's Travels*, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end.[2][3] By analogy, a CPU may read a digital word big end first, or little end first.

How to read more than one byte?

Consider the 32-bit integer `0x12345678`

- What is the big endian representation?
- What is the little endian representation?

How to read more than one byte?

Consider the 32-bit integer `0x12345678`

- What is the big endian representation?
- What is the little endian representation?

Big endian

`0x12`	`0x34`	`0x56`	`0x78`

How to read more than one byte?

Consider the 32-bit integer `0x12345678`

- What is the big endian representation?
- What is the little endian representation?

Little endian

`0x78`	`0x56`	`0x34`	`0x12`

Why?

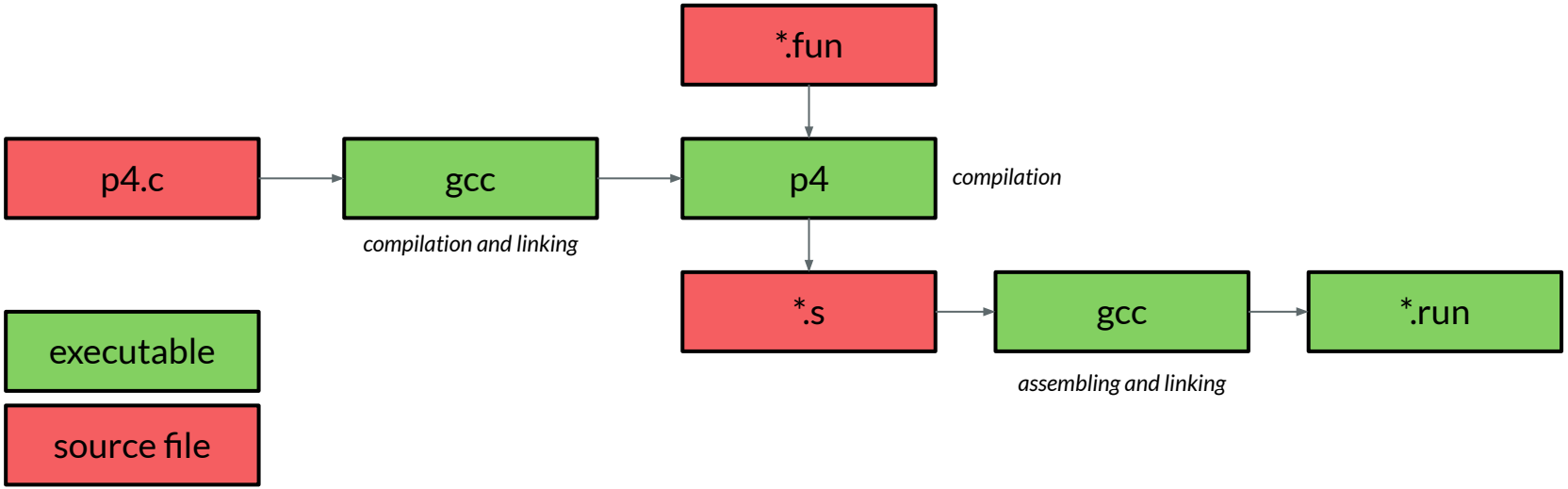
```
uint32_t x = 0x12345678;

uint8_t y = (uint8_t) x;
```

If x and y point to the same address, what's the value of y?

Bonus Round: P4

*These slides were last minute stolen from last year so they are in x86, sorry

what is p4?


*.fun
p4.c
gcc
p4
compilation
compilation and linking
*.s
gcc
*.run
assembling and linking
executable
source file

how much does my compiler have to do?

- is a compiler that only has print statements okay?

how much does my compiler have to do?

- is a compiler that only has print statements okay?
- in general, you can make any optimizations as long as your compiler makes no assumptions about the program state at runtime
  - e.g. the expression `5 + 3 * 4` can be simplified to 17 but `5 + x * 4` cannot be simplified even if it is possible for your compiler to figure out the value of x
  - if/else/while conditions, function calls, etc. can't be precomputed
- your compiled code shouldn't be reading any input
  - don't write a "compiler" that just writes a generic interpreter in assembly, actually compile the test case

how do i turn my interpreter into a compiler?

- find all the places you modify the program state
  - printing, assigning variables, calling functions, taking if/else branches, etc.
- come up with some assembly that has the same effect
- instead of modifying the program state, output that assembly instead
- the resulting binary will contain all the instructions it needs to execute the fun program

debugging a compiler?

how to use gdb with assembly?

- layout asm → like `layout src` but for assembly
- ni → like `next` but instead of next statement, it goes to the next instruction
- si → like `step` but instead of stepping into statements, it steps into calls and jumps
- info reg → display the contents of all the registers

Compilers are complicated

How do you map variables to 16 registers & memory locations?

For example, int c = a + b;

- Where is a? b? Where should c go? Are both a and b only in memory? What regs can we modify?

p4 pro tip — do not hold values in registers

How do you map variables to 16 registers & memory locations?

For example, int c = a + b;

- Where is a? b? Where should c go? Are both a and b only in memory? What regs can we modify?

Stack Machines

Let's say our architecture has only one general purpose register: %rsp. To make up for this, we are changing the ISA to only include the following instructions:

```
PUSH val   // pushes a value onto the stack
ADD        // pops 2 values, adds them, and pushes the result
NEGATE     // pops a value, negates it, and pushes the result
PRINT      // pops from the stack and prints
```

Calling Convention

- C ABI for functions defines which registers are for parameters and returning
- Only necessary to call external functions (that you don't compile)
- Calling your own functions can use whatever convention you want
  - Can you change your convention based on anything?
  - Does it have to be consistent with itself?
  - What are some tradeoffs of staying true to the x86 calling convention?

Using Labels

```
.section .data
variable: .quad 0x0123456789ABCDEF

.section .text
function: ...
    mov variable, %rax
    mov %rax, variable
    call function
```

Printing Things!

```
//data segment
puts("    .data");
puts("format: .byte '%', 'l', 'u', 10, 0");
```

What does this do?

Printing Things!

```
//data segment
puts("    .data");
puts("format: .byte '%', 'l', 'u', 10, 0");
```

```
$ cat test.c
#include <stdio.h>


int main(int argc, char **argv) {
    printf("%lu\n", argc);
    return 0;
}
$ gcc test.c
```

```
$ objdump -d a.out
...
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   89 7d fc                mov    %edi,-0x4(%rbp)
 655:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
 659:   8b 45 fc                mov    -0x4(%rbp),%eax
 65c:   89 c6                   mov    %eax,%esi
 65e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi
 665:   b8 00 00 00 00          mov    $0x0,%eax
 66a:   e8 b1 fe ff ff          callq  520 <printf@plt>
 66f:   b8 00 00 00 00          mov    $0x0,%eax
 674:   c9                      leaveq
 675:   c3                      retq
 676:   66 2e 0f 1f 84 00 00    nopw
%cs:0x0(%rax,%rax,1)
 67d:   00 00 00
...
0x9f(%rip): 25 6c 75 0a 00
```

Printing Things!

```
//data segment
puts("    .data");
puts("format: .byte '%', 'l', 'u', 10, 0");
```

```
$ cat test.c
#include <stdio.h>


int main(int argc, char **argv) {
    printf("%lu\n", argc);
    return 0;
}
$ gcc test.c
```

```
$ objdump -d a.out
...
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   89 7d fc                mov    %edi,-0x4(%rbp)
 655:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
 659:   8b 45 fc                mov    -0x4(%rbp),%eax
 65c:   89 c6                   mov    %eax,%esi
 65e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi
 665:   b8 00 00 00 00          mov    $0x0,%eax
 66a:   e8 b1 fe ff ff          callq  520 <printf@plt>
 66f:   b8 00 00 00 00          mov    $0x0,%eax
 674:   c9                      leaveq
 675:   c3                      retq
 676:   66 2e 0f 1f 84 00 00    nopw
%cs:0x0(%rax,%rax,1)
 67d:   00 00 00
...
0x9f(%rip): 25 6c 75 0a 00
```

Questions?