Notes on more datastructures

- This assignment should be fairly straightforward - but there are a lot of details that you need to remember
- What I want to review mainly is:
  - How does the "stack" work?
  - Allocations / Heaps. Arrays + Indexing.
  - Machine details (padding, offset, etc)

AT&T Syntax vs Intel syntax

- Use the **AT&T** one
- Look at: https://en.wikibooks.org/wiki/X86_Assembly

Get started with the "Simple" Instructions

- `Arith`, `Load` and `Store`, `Cmp`
- What do these mean *denotationally* (after running this instruction, what is the *new state* of your stack / heap) and *operationally* (*how* - what assembly instructions do you need to achieve the end points)?

Draw out a computer

- Stack is a **byte addressable** array. A variable will occupy a **word** or **8 bytes**.
- Variables are **stored on the stack** - ordered in the way that they are declared
- Registers are **much faster** to acess than the stack - and the only way your CPU does computation
  - So when accessing a variable - you need to load them off of the stack into a register before doing anything

Suffixes

- "word" in the 64-bit case means 64-bits - but an alternate meaning of "word" is 16. Don't use `movw` cus that's wrong.
- X86 instructions only work on certain bit-width integers. The suffix of each instruction determines what bit-width the instruction works on:
  - q = "quad" or 64-bit
  - l = "long" or 32-bits
  - w = "word" or 16-bits
  - b = "byte" or 8-bits
- For the most part - the solution will always use the `<instr>q` version of the assembly instruction.

Example of calling "add" (drawn out)

- Example program:

This:

```
fn main() -> int {
        let x: int = 3;
        let j: int = 4;

        x = x + j;
        return 0;
}
```

Gets lowered to:

```
Externs

Globals

Function main() -> Int {
  Locals
    _t1 : Int
    j : Int
    x : Int

  entry:
    Copy(x, 3)
    Copy(j, 4)
    Arith(_t1, add, x, j)
    Copy(x, _t1)
    Ret(0)
}
```

Now, how do we do codegen for this program?

- First - **set up the stack** by "pushing" the stack pointer to allocate space for the variables. The stack **grows downward** from the "base pointer" - which denotes the location of the base of this frame.
- Now the stack will look like: image drawn on whiteboard
- Finally, we deal with the `entry` block of main:
  - make a label for it and jump to it (the label will be in the LIR)
- What should we do on `Copy`? Since we are copying constants - this is easy. **Mutate the stack**. Semantically, what `Copy` should do is:
  - Mutate the location in the stack where the `lhs` is stored with the value of the `rhs`
  - **You need to codegen in a way that respects this** (this is what ben's solution does)
  - This is just done with a simple `movq $0, -<loc>(%rbp)` - where `loc` is the location where the variable is stored.
- What about `Arith`?
  - The **semantics** of Arith is to take the two operands (the last 2 arguments), load their values, do the operation, then store the result in the `lhs` (first argument).
  - You can achieve this with these instructions:

    ```
    ; load the value of x
    movq -24(%rbp), %r8
    ; the source of "addq" can be a memory location - in this case y
    addq -16(%rbp), %r8
    ; move the result into the temp variable
    movq %r8, -8(%rbp)
    ```

  - Note that because **at most one operand may be a memory location** - we use the `%r8` register.
- Finally - how about `Ret`?
  - `Ret` should just jump to the epilouge
  - the epilouge resets the stack to what it was, then calls the `ret` assembly instruction

A quirk with Compare

- In x86 assembly, the `cmp` instruction sets a flag rather than storing the result in a visible register.
- That's why compare needs to be done into two instructions:

```
asm
  ;; do the comparison
  cmpq v1, v2
```

```
  ;; set the value to 0 or 1 based on the result of the comparison
  ;; where c is one of the condition codes:  (`e`, `ne`, `z`, `nz`, `s`, `ns`, `g`,
`ge`, `l`, `le`)
  set<c> %r8b
```

Function pointers

- **Implicitly** every declared function has a pointer to it with itself as the name.
- i.e:

```
fn foo() -> int {
  return 0;
}
fn main() -> int {
  let x: &() -> int;
  let y: int;
  x = foo;
  y = x();
  return 0;
}
```

is implicitly

```
let foo: &() -> int;
fn foo() -> int {
  return 0;
}
fn main() -> int {
  let x: &() -> int;
  let y: int;
  x = foo;
  y = x();
  return 0;
}
```

- what happens when I have a global variable that has the same name as a function? I need to mangle the global. This is described in the lecture notes.

  ```
  let foo: int;
  fn foo() -> int {
    return 0;
  }
  fn main() -> int {
    let x: &() -> int;
    let y: int;
    x = foo;
    y = x();
    return 0;
  }
  ```
- A **label is just a location in memory**. What ben does for each function pointer global is store the pointer to the function label in that global.
- You can run `call` on an adress directly as well.

Structs and Arrays review

- Structs and arrays are **heap allocated** - meaning you can use an "allocator" to give you access to memory
  - Allocators are very interesting but the "classic" allocator is `malloc` from `glibc`.

    - The interface of the allocator is much less restricted (i.e. you can `malloc` and `free`) - than a stack - but that also makes it harder to manage (i.e. you need to figure out when to `free` or you'll have a memory leak).
- Arrays are run-time checked by size.
- All fields in the struct are 8-bytes (64-bits). This means that a **struct is almost like an array** (at least, has the same offset logic as an array).
    - Map the `field` to an `index` - remember that this is **alphabetically ordered** - this matters b/c you'll get a wrong offset from the reference solution if you don't respect this.
    - Since **all fields accesses are valid** (due to type checking) - we can just access directly. The code should be similar to array access.

Calling Conventions

- Internal functions:
    - Save any caller saved regiesters before calling
    - push all args on the stack
    - call the function
- External functions:
    - Follow what's on the lecture notes
    - This is the **C calling convention**. This is not dictated by us - but by how C fuctions work.