CMSC 430, March 3rd 2020

Hustle

Stacks

Stacks

- Let’s review stacks a little bit

Stacks

- Let’s review stacks a little bit
- One thing I was trying to get across, but may have failed:

Stacks

- Let’s review stacks a little bit
- One thing I was trying to get across, but may have failed:
    - There are many ways to use stacks to store temporaries!

Stacks

- Let’s review stacks a little bit
- One thing I was trying to get across, but may have failed:
  - There are many ways to use stacks to store temporaries!
  - Only thing that matters: that it works.

Stacks: Part 1

Stacks: Part 1

- In AMD64, there are two registers normally used for the stack:

Stacks: Part 1

- In AMD64, there are two registers normally used for the stack:
  - **`rsp`** and **`rbp`**

Stacks: Part 1

- In AMD64, there are two registers normally used for the stack:
  - **`rsp`** and **`rbp`**
- Importantly, these registers are not special!

Stacks: Part 1

- In AMD64, there are two registers normally used for the stack:
  - **`rsp`** and **`rbp`**
- Importantly, these registers are not special!
  - In fact, in the architecture specification they are explicitely called out as *general purpose*

Stacks: Part 2

Stacks: Part 2

- The idea behind having two:

Stacks: Part 2

- The idea behind having two:
  - The stack pointer points to the "top" of the stack

Stacks: Part 2

- The idea behind having two:
  - The stack pointer points to the "top" of the stack
  - The base pointer points to the "bottom" of the stack

Stacks: Part 2

- The idea behind having two:
  - The stack pointer points to the "top" of the stack
  - The base pointer points to the "bottom" of the stack
- The ’distance’ between the determines how many things are currently on the stack.

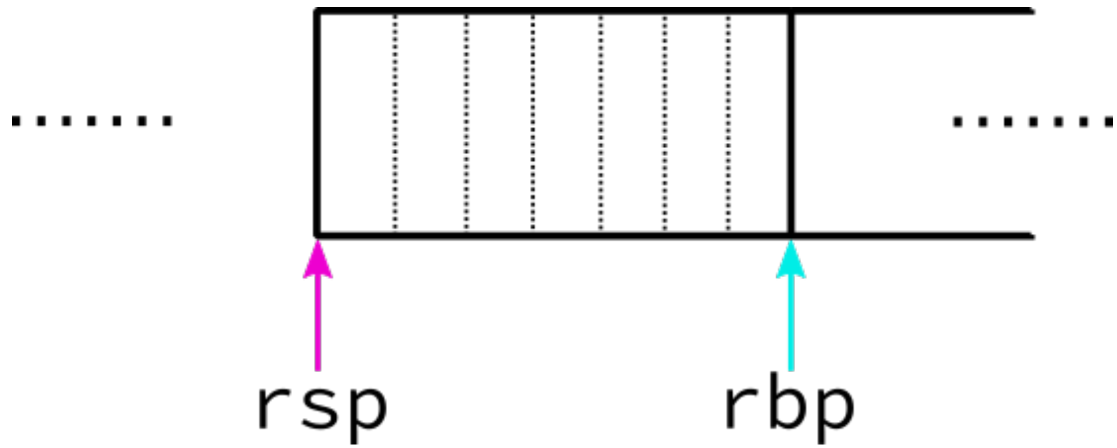
Stacks: Part 3

Stacks: Part 3

- Let’s take a look:

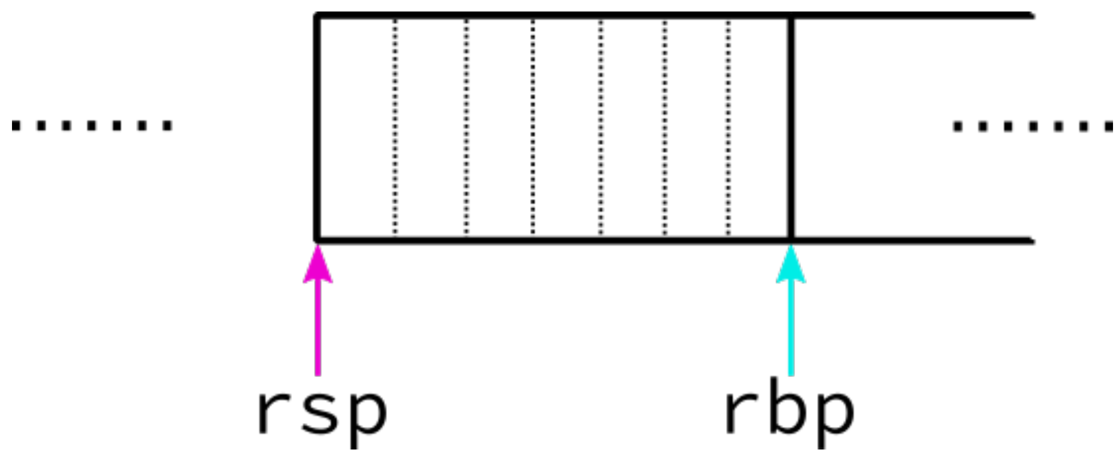
Stacks: Part 3

- Let's take a look:



Stacks: Part 3

- Let's take a look:



- Even with both **rsp** and **rbp** we have to keep track of things

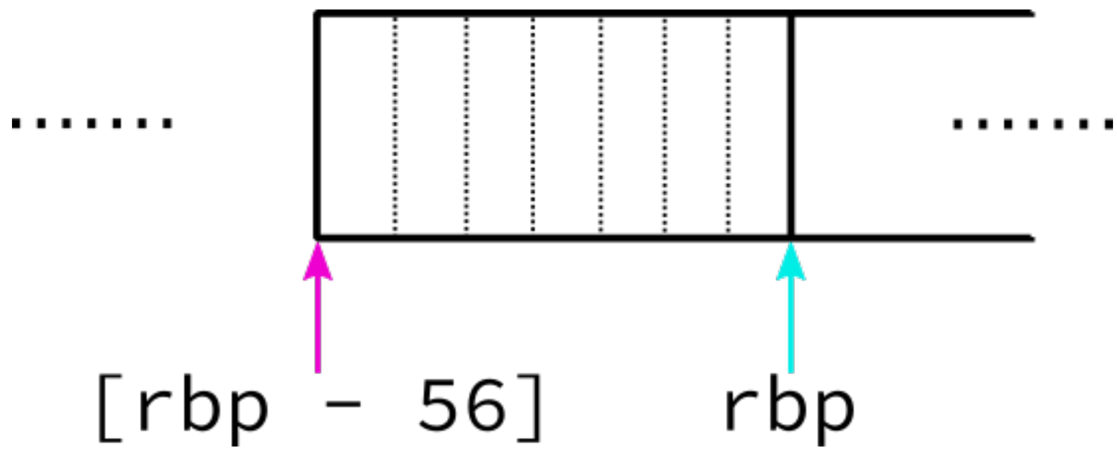
Stacks: Part 4

Stacks: Part 4

- Since we’re keeping track of things, the following are all equivalent:

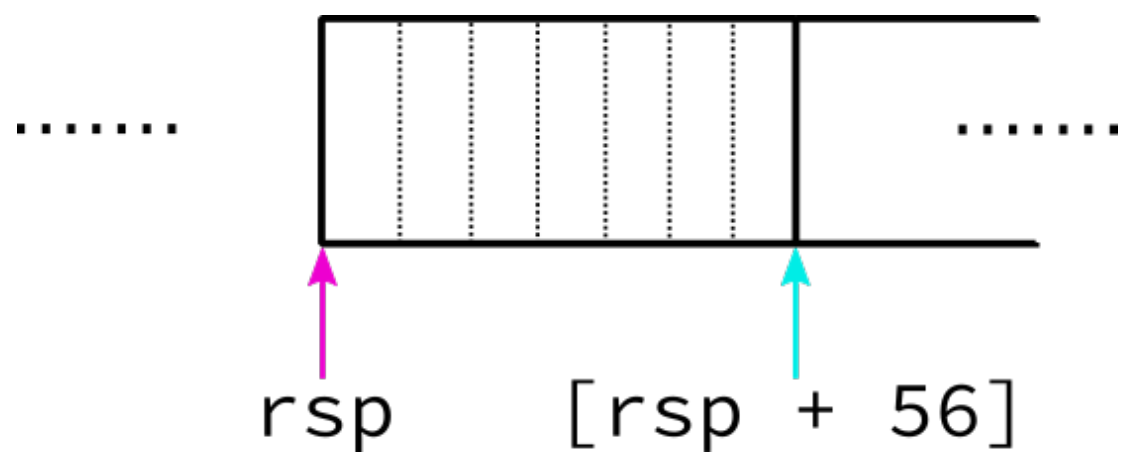
Stacks: Part 4

- Since we're keeping track of things, the following are all equivalent:



Stacks: Part 4

- Since we're keeping track of things, the following are all equivalent:



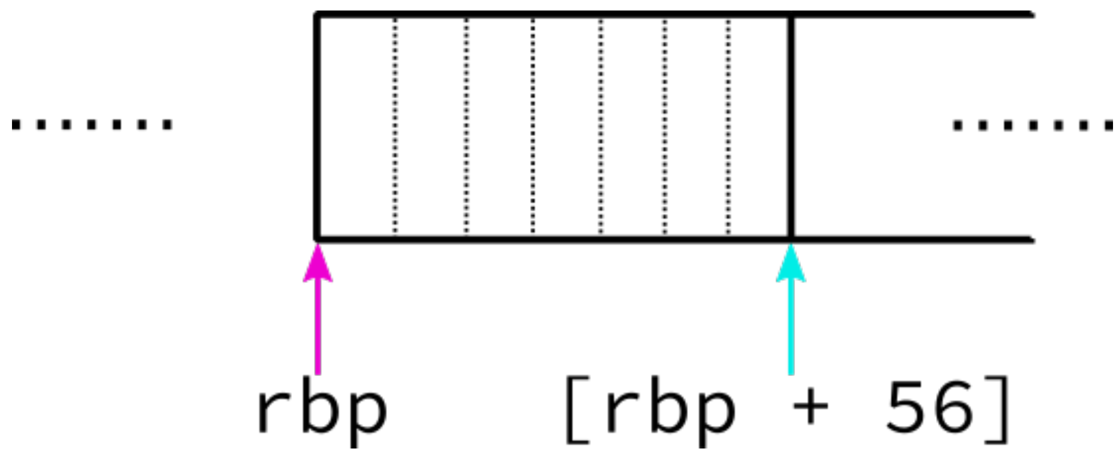
Stacks: Part 5

Stacks: Part 5

- But I also said there was nothing special about **`rsp`** and **`rbp`**

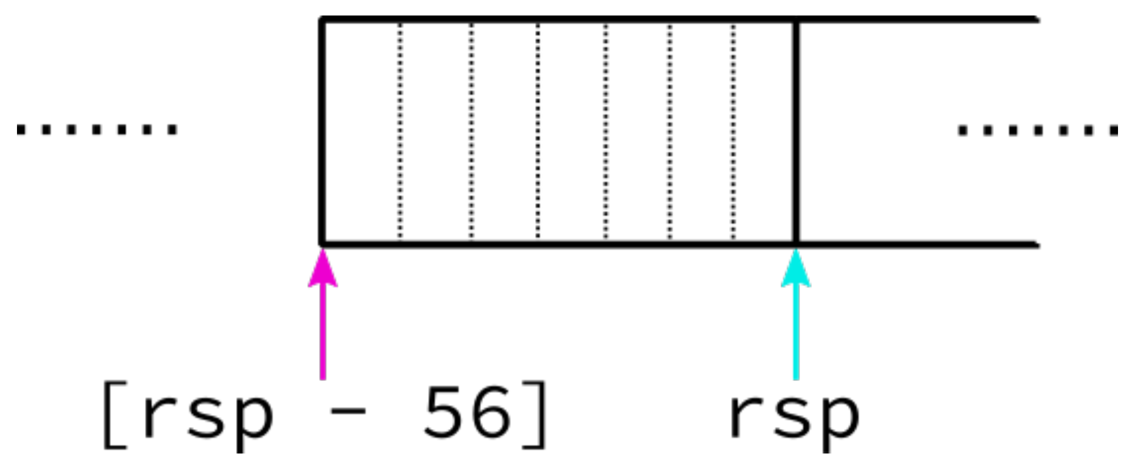
Stacks: Part 5

- But I also said there was nothing special about **`rsp`** and **`rbp`**



Stacks: Part 5

- But I also said there was nothing special about **`rsp`** and **`rbp`**



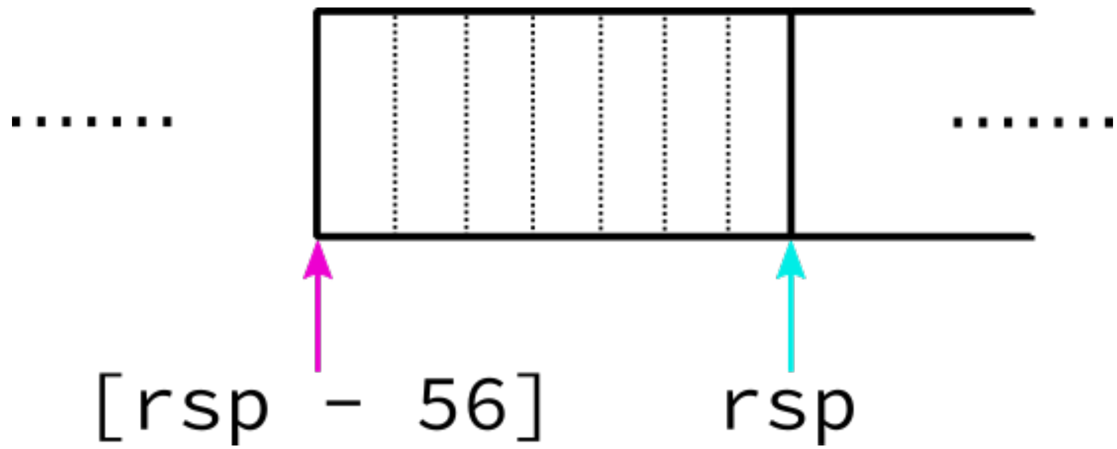
Stacks: Part 6

Stacks: Part 6

- We went with the last one:

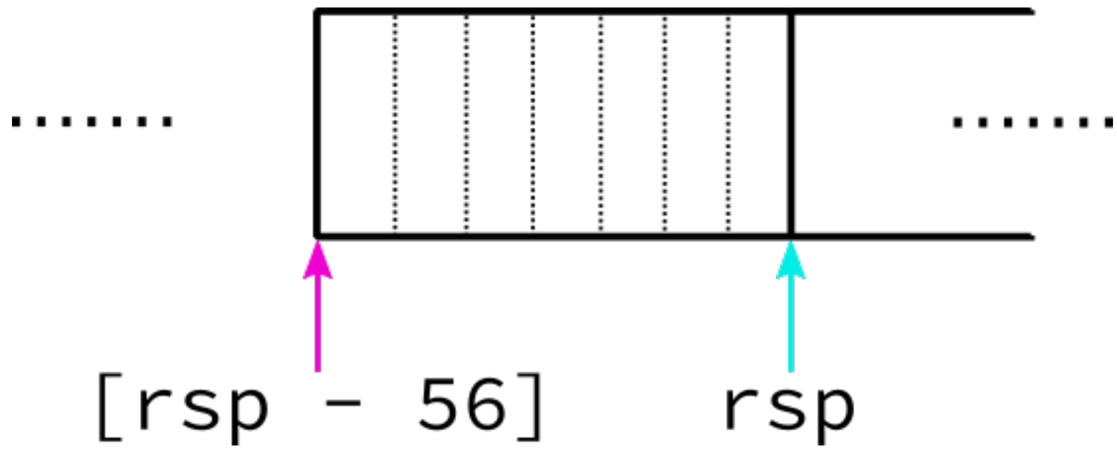
Stacks: Part 6

- We went with the last one:



Stacks: Part 6

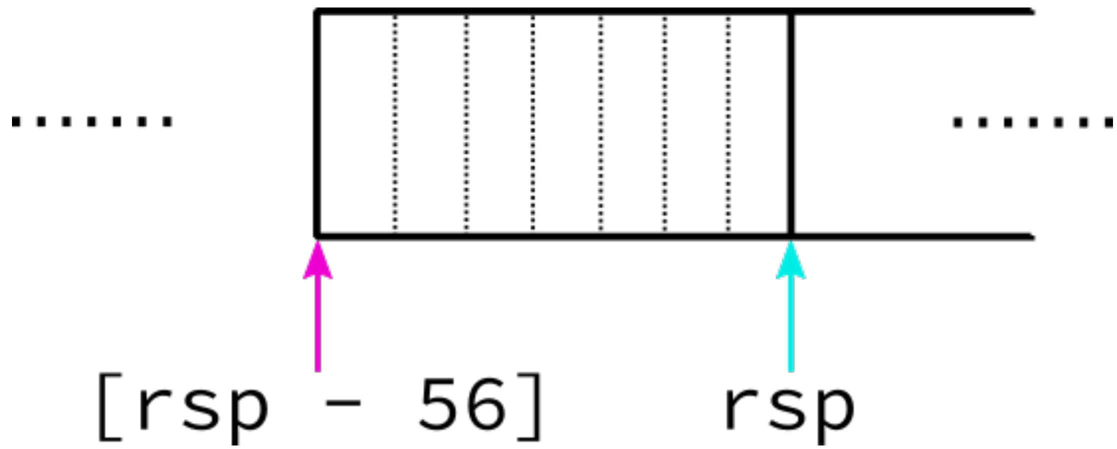
- We went with the last one:



- Why not use **`rbp`**?

Stacks: Part 6

- We went with the last one:



- Why not use **rbp**?
    - Because **rbp** is special to C
    - :(

Our languages so far:

Our languages so far:

- Each lecture we're seeing the complexities of our language grow

Our languages so far:

- Each lecture we’re seeing the complexities of our language grow
- Most of the time these new features change things in our interpreter/compiler but not in our RTS

Our languages so far:

- Each lecture we're seeing the complexities of our language grow
- Most of the time these new features change things in our interpreter/compiler but not in our RTS
- Today is an RTS day.

Our languages so far:

- Each lecture we're seeing the complexities of our language grow
- Most of the time these new features change things in our interpreter/compiler but not in our RTS
- Today is an RTS day.
  - Which is also a compiler day, to take advantage of our new RTS!

Hustle

Hustle

- Hustle is going to introduce a notion of a *heap* to our RTS

Hustle

- Hustle is going to introduce a notion of a *heap* to our RTS
- We will use the heap to implement *boxed values*

What's in the box?

What's in the box?

- A good short-hand:

What's in the box?

- A good short-hand:
  - Box = not on the stack

What’s in the box?

- A good short-hand:
  - Box = not on the stack
- In general, boxed values are things you need to derefence a pointer to get.

What's in the box?

- A good short-hand:
    - Box = not on the stack
- In general, boxed values are things you need to derefence a pointer to get.
- But not all things that you need to dereference a pointer are 'boxed'

Boxing Day

```
racket> ; show box and unbox
```

What’s in the box?

What’s in the box?

- Boxes, without a notion of pointer equality, are uninteresting.

What's in the box?

- Boxes, without a notion of pointer equality, are uninteresting.
- In our language, boxes are single-element vectors

What's in the box?

- Boxes, without a notion of pointer equality, are uninteresting.
- In our language, boxes are single-element vectors
- For now, we can see boxes as an important stepping stone to something much more important:

What's in the box?

- Boxes, without a notion of pointer equality, are uninteresting.
- In our language, boxes are single-element vectors
- For now, we can see boxes as an important stepping stone to something much more important:
  - **cons**

Getting Box/Car on track

Getting Box/Car on track

- Goal for today:

Getting Box/Car on track

- Goal for today:
- Understand how things like **box** and **cons** are implemented

Hustle’s AST

Hustle's AST

- We're only showing the new stuff:

Hustle’s AST

- We’re only showing the new stuff:
  - `e = ...`

Hustle's AST

- We're only showing the new stuff:
  - `e = ...`
- Expressions are unchanged!

Hustle's AST

- We're only showing the new stuff:
  - **`e = ...`**
- Expressions are unchanged!
  - **`p1 = ... | box | unbox | car | cdr`**

Hustle's AST

- We're only showing the new stuff:
  - **`e = ...`**
- Expressions are unchanged!
  - **`p1 = ... | box | unbox | car | cdr`**
  - **`p2 = ... | cons`**

Hustle's AST

- We're only showing the new stuff:
    - **`e = ...`**
- Expressions are unchanged!
    - **`p1 = ... | box | unbox | car | cdr`**
    - **`p2 = ... | cons`**
- Is this enough?

Hustle's AST

- We're only showing the new stuff:
  - **`e = ...`**
- Expressions are unchanged!
  - **`p1 = ... | box | unbox | car | cdr`**
  - **`p2 = ... | cons`**
- Is this enough?
  - Not if we want programs to have boxed results.

Hustle's AST

- We're only showing the new stuff:
  - **`e = ...`**
- Expressions are unchanged!
  - **`p1 = ... | box | unbox | car | cdr`**
  - **`p2 = ... | cons`**
- Is this enough?
  - Not if we want programs to have boxed results.
  - v = ... | (box v) | (cons v v) | ’()

Find value in the hustle

Find value in the hustle

- We’ve got 3 new values, what do we do about representation?

Find value in the hustle

- We’ve got 3 new values, what do we do about representation?
- Before: All values were ’flat’

Find value in the hustle

- We’ve got 3 new values, what do we do about representation?
- Before: All values were ’flat’
- Now: values can be arbitrarily big

Find value in the hustle

- We’ve got 3 new values, what do we do about representation?
- Before: All values were ’flat’
- Now: values can be arbitrarily big
  - So they won’t all fit in a machine word!

Find value in the hustle

- We've got 3 new values, what do we do about representation?
- Before: All values were 'flat'
- Now: values can be arbitrarily big
  - So they won't all fit in a machine word!
- Idea:

Find value in the hustle

- We’ve got 3 new values, what do we do about representation?
- Before: All values were ’flat’
- Now: values can be arbitrarily big
  - So they won’t all fit in a machine word!
- Idea:
  - Make distinction between flat and boxed values

Find value in the hustle

- We've got 3 new values, what do we do about representation?
- Before: All values were 'flat'
- Now: values can be arbitrarily big
  - So they won't all fit in a machine word!
- Idea:
  - Make distinction between flat and boxed values
  - Then make distinctions between the flat (immediate) and boxed values

Wait a bit... what about the heap?

Wait a bit... what about the heap?

- Early I had mentioned that we’ll get a heap, then I never addressed it

Wait a bit... what about the heap?

- Early I had mentioned that we’ll get a heap, then I never addressed it
- I didn’t forget, we just had to lay the groundwork

Wait a bit... what about the heap?

- Early I had mentioned that we'll get a heap, then I never addressed it
- I didn't forget, we just had to lay the groundwork
- For now, it's easy:

Wait a bit... what about the heap?

- Early I had mentioned that we'll get a heap, then I never addressed it
- I didn't forget, we just had to lay the groundwork
- For now, it's easy:
  - Just have the RTS allocate a big block of memory. That's it. That's the heap.

Wait a bit... what about the heap?

- Early I had mentioned that we'll get a heap, then I never addressed it
- I didn't forget, we just had to lay the groundwork
- For now, it's easy:
  - Just have the RTS allocate a big block of memory. That's it. That's the heap.
- We gotta keep track of it too...

Wait a bit... what about the heap?

- Early I had mentioned that we'll get a heap, then I never addressed it
- I didn't forget, we just had to lay the groundwork
- For now, it's easy:
  - Just have the RTS allocate a big block of memory. That's it. That's the heap.
- We gotta keep track of it too...
  - Uhh... let's use **`rdi`**

Wait a bit... what about the heap?

- Early I had mentioned that we'll get a heap, then I never addressed it
- I didn't forget, we just had to lay the groundwork
- For now, it's easy:
  - Just have the RTS allocate a big block of memory. That's it. That's the heap.
- We gotta keep track of it too...
  - Uhh... let's use **`rdi`**
- Moving on.

From grifters to hustlers

From grifters to hustlers

- Before we had the following:

From grifters to hustlers

- Before we had the following:

```
(define imm-shift        1)
(define imm-type-mask    (sub1 (shift 1 imm-shift)))
(define imm-type-int     0)
(define imm-val-true     3)
(define imm-val-false    1)
```

From grifters to hustlers

- Which becomes:

```
(define result-shift     3)
(define result-type-mask (sub1 (shift 1 result-shift))
(define type-imm         0)
(define type-box         1)
(define type-pair        2)
```

We need more

We need more

- However, this only helps us determine the types

We need more

- However, this only helps us determine the types
- We need more in order to disambiguate the values

All the bits

All the bits

```
(define result-shift     3)
(define result-type-mask (sub1 (shift 1 result-shift))
(define type-imm         0)
(define type-box         1)
(define type-pair        2)
(define imm-shift         (+ 3 result-shift))
(define imm-type-mask     (sub1 (shift 1 imm-shift)))
(define imm-type-int      (shift 0 result-shift))
(define imm-val-true     (shift 1 result-shift))
(define imm-val-false    (shift 2 result-shift))
(define imm-val-empty    (shift 3 result-shift))
```

Follow these instructions

- Here is a quick overview of some useful facts
- **`rdi`**

Follow these instructions

- Here is a quick overview of some useful facts
- **`rdi`**

```
MOV RAX, [RDI]
```

Follow these instructions

- Here is a quick overview of some useful facts
- **`rdi`**

```
MOV RAX, [RDI]
MOV RAX, [RDI + 8]
```

Follow these instructions

- Here is a quick overview of some useful facts
- **`rdi`**

```
MOV RAX, [RDI]
MOV RAX, [RDI + 8]
MOV [RDI + 8], RAX
```

Follow these instructions

- Here is a quick overview of some useful facts
- **`rdi`**

```
MOV RAX, [RDI]
MOV RAX, [RDI + 8]
MOV [RDI + 8], RAX
```

- we call this **`offset`**

Follow these instructions

- Here is a quick overview of some useful facts
- someone asked about how many 'lets' we can have:

Follow these instructions

- Here is a quick overview of some useful facts
- someone asked about how many ’lets’ we can have:
- run the following at your terminal
  - **`ulimit -a`**

Follow these instructions

- Here is a quick overview of some useful facts
- someone asked about how many ’lets’ we can have:
- run the following at your terminal
    - **`ulimit -a`**
- If I did my math right (always questionable), we should be able to store ~1 million let-bound variables.

Let’s write it!