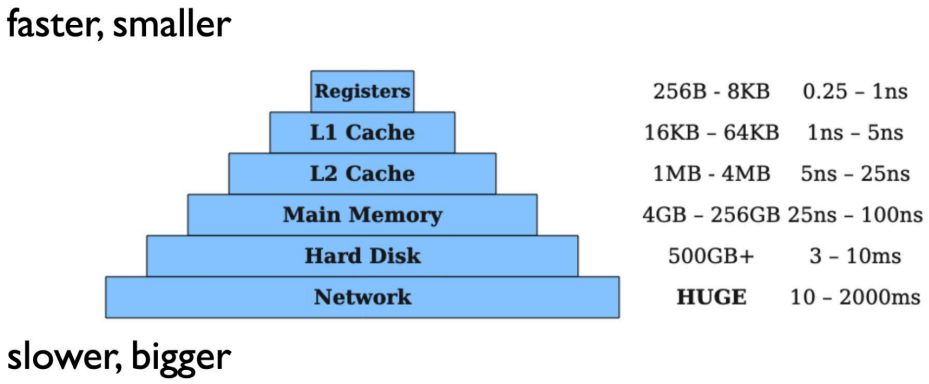
CSE 231: Register Allocation

Ranjit Jhala *May 21, 2026*

Memory Hierarchy ...

Not all *memory* is created equal!



(via Max New)

... Stack is an Expensive Place to Store Values

Source

```
(let ((a0 92)
      (a1 (add1 a0))
      (a2 (add1 a1))
      (a3 (add1 a2))
      (a4 (add1 a3))
      (a5 (add1 a4)))
 a5)
```

Asm

```
mov rax, 184
mov [rbp - 8*2], rax
mov rax, [rbp - 8*2]
add rax, 2
mov [rbp - 8*3], rax
mov rax, [rbp - 8*3]
add rax, 2
mov [rbp - 8*4], rax
mov rax, [rbp - 8*4]
add rax, 2
mov [rbp - 8*5], rax
mov rax, [rbp - 8*5]
add rax, 2
mov [rbp - 8*6], rax
mov rax, [rbp - 8*6]
add rax, 2
mov [rbp - 8*7], rax
mov rax, [rbp - 8*7]
```

Asm-Opt

```
mov rbx, 184
add rbx, 2
add rbx, 2
add rbx, 2
add rbx, 2
add rbx, 2
mov rax, rbx
```

Much faster to access *registers* than *main memory*!

```
$ time ./test/reg_slow.fun.run 1000000000
97
Executed in    1.84 secs    ...

$ time ./test/reg_opt.fun.run 1000000000
97
Executed in  763.37 millis   ...
```

Optimization: Register Allocation

Lets use **registers** instead of defaulting to **stack** storage.

Why is this tricky?

QUIZ: Register Optimized Code

Program

```
(let ((a1 (+ 10 10))
      (a2 (* 2 a1))
      (a3 (* 3 a2)))
  (* 10 a3))
```

Optimized Asm

```
```

```
(let ((n (* 5 5))
      (m (* 6 6))
      (x (+ n 1))
      (y (+ m 1)))
    (+ x y))
```

```
```

```
(defn (f a)
  (let ((x (* a 2))
        (y (+ x 7)))
    y))
```

```
; a --> [rbp + 16]
```

```
(defn (f a)
  (let ((x (* a 2))
        (y (+ x 7)))
    (g x y)))
```

```
; a --> [rbp + 16]
```

Optimization Relies on Variables?

What if the programmer *instead* wrote code like

Example 1

```
(* 10 (* 3 (* 2 (+ 10 10))))
```

Example 2

```
(+ (+ (* 5 5) 1) (+ (* 6 6) 1))
```

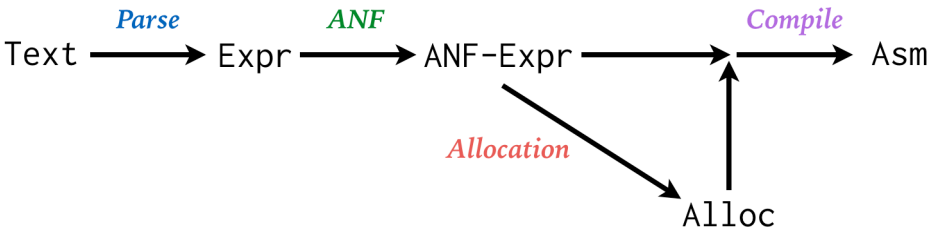
Example 3

```
(defn (f a)
  (+ (* a 2) 7))
```

Yikes, how to optimize without *names*?

Register Optimization Pipeline

Change the compiler with 3 new steps.



1. **Transform** to ANF *intermediate representation*,
2. **Allocation** of variables to *registers*,
3. **Compile** using *allocation*.

Administrative Normal Form (ANF) Transform

- **Immediate:** *constants or variable lookups* whose values can be loaded with a single machine instruction
- **ANF:** every call or prim-op's arguments are *immediate*.

Step 1 transforms `(+ (+ (* 5 5) 1) (+ (* 6 6) 1))` to ANF

```
(let ((n (* 5 5))
      (m (* 6 6))
      (x (+ n 1))
      (y (+ m 1)))
   (+ x y))
```

Lets look at these steps *backwards* starting with Step 2 (allocation), then Step 3 (compilation), *after which* we will circle back to (Step 1) ANF transformation.

Step 2: Allocation (Assigning Vars to Registers)

Registers: Fast Storage for Variables

Some fixed set of registers that we can use to store variables.

```
enum Reg {
    RAX, RBX, RCX, RDX,
    R8, R9, R10, R11, ...
}
```

Locations: Where to Store a Variable

A `Loc` is a *location* where a variable can be stored,

- a *register* or
- a *stack offset*

```
pub enum Loc {
  /// Register
  Reg(Reg),
  /// Stack local:  [rbp - 8 * offset]
  Stack(i32),
}
```

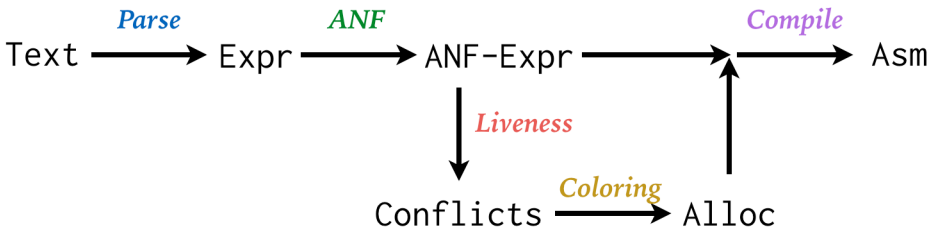
Allocation: Mapping Variables to Locations

An `Alloc` is a mapping from variable names to `Loc`:

```
pub struct Alloc(HashMap<String, Loc>);
```

Computing an Allocation

How to compute `Alloc` for a given function body (`Expr`)?



Step 1: Build *conflict graph* over function variables

Step 2: Color variables so *adjacent vars have different* colors.

1. Conflict Graph

Undirected graph where:

- **Vertices** are *variables*
- **Edges** between `x`, `y` if both *must be available at same time*

QUIZ: Fill in the Conflict Graphs

Program

```
(let ((a0 92))
      (a1 (+ a0 1))
      (a2 (+ a1 1))
      (a3 (+ a2 1))
      (a4 (+ a3 1))
      (a5 (+ a4 1)))
  a5)
```

Conflict Graph

```
(let ((n (* 5 5))
      (m (* 6 6))
      (x (+ n 1))
      (y (+ m 1)))
    (+ x y))
```

2. Coloring

- Label each *vertex* with a **color** (register)
- Ensuring **adjacent** vertices have **different** colors.

What if we *require* n colors, but only *have* $r < n$ registers? ***Spill*** the remaining $n - r$ variables onto the stack!

Program

```
(let ((n (5 * 5))
      (m (6 * 6))
      (x (n + 1))
      (y (m + 1))
      (z (x + y))
      (k (z * z))
      (g (k + 5)))
  (+ k 3))
```

Conflict Graph

```
n --- m
  \ / |
   X  |
  / \ |
y --- x
|   /
|  /
| /
z -- k -- g
```

Coloring

```
VAR         REG
-------------
n    ---->
m    ---->
x    ---->
y    ---->
z    ---->
k    ---->
g    ---->
```

Conflict Graph from Live Variables

We *stared* at the code to find "conflicts". How to *automate*?

Vars `{x1...xn}` are **live for** `e` if we *need* the values of `x1...xn` to evaluate `e`.

1. Identify **live variables** at each point,
2. Conflicts between **pairs of concurrently live** variables.

QUIZ: Live Variables

Lets identify the live/conflicts in the quiz above.

```
                          ;      LIVE Variables


                          ;
(let (a0 92)
                          ;
 (let (a1 (+ a0 1))
                          ;
  (let (a2 (+ a1 1))
                          ;
   (let (a3 (+ a2 1))
                          ;
    (let (a4 (+ a3 1))
                          ;
     (let (a5 (+ a4 1))
                          ;
        a5))))))
```

```
                          ;      LIVE Variables


                          ;
(let (n (* 5 5))
                          ;
 (let (m (* 6 6))
                          ;
  (let (x (+ n 1))
                          ;
   (let (y (+ m 1))
                          ;
      (+ x y)))))
```

Similar to **free variables** for closure (but not quite same...)

Implementing `live` in your compiler

Your compiler will implement a function

```
fn live(
 graph: &mut ConflictGraph, /// graph of edges
 e: &Expr,                  /// expression to analyze
 binds: &HashSet<String>,   /// let-binds outside e
 params: &HashSet<String>,  /// funparams (on stack)
 out: &HashSet<String>,     /// vars "LIVE" after `e`
) -> HashSet<String>        /// vars "LIVE" *before*
```

`graph` is `mut`: Add *conflict edges* while traversing `e`.

Step 3: Compilation

Use Allocated Registers (not only `rax`)

Suppose that

- `imm1` and `imm2` *immediate* values,
- `Alloc` maps `imm1` to `<imm1>` and `imm2` to `<imm2>`,
- we want to compile `(+ imm1 imm2)` to `<dst>`.

How to compile ANF code using the `Alloc` mapping?

What if `dst` is on the stack?
What if `dst` is a register?

ANF Code

```
(+ imm1 imm2)
```

Compiled Asm

```
```

```
(let ((a0 92))
      (a1 (+ a0 1))
      (a2 (+ a1 1))
      (a3 (+ a2 1))
      (a4 (+ a3 1))
      (a5 (+ a4 1)))
  a5)
```

```
; ALLOC a0...a5 --> rax
```

Saving Registers

Callee functions can overwrite *caller* registers!

Starter code does "callee save"; beware calls into *runtime*.

1. Callee saves Each function saves & restores all of the registers it uses; callers do not have to store anything.

2. Caller saves Each caller saves & restores all of the registers it uses; callees do not have to store anything.

3. Hybrid Combine the two options; some registers are *caller-saved* others are *callee-saved*