# Register-based Its

## And bytecode

# Stack-Based Bytecode

```
push 3

push a

push b

send +

send /

send ceiling
```

```
(3 / (a + b)) . ceiling()
```

- Each operation produces or consumes values into/from the stack
- Compact linear representation
- Execution “falls” from one instruction to the next one

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3

push a

push b

send +

send /

send ceiling
```

the stack

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```


the stack

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```

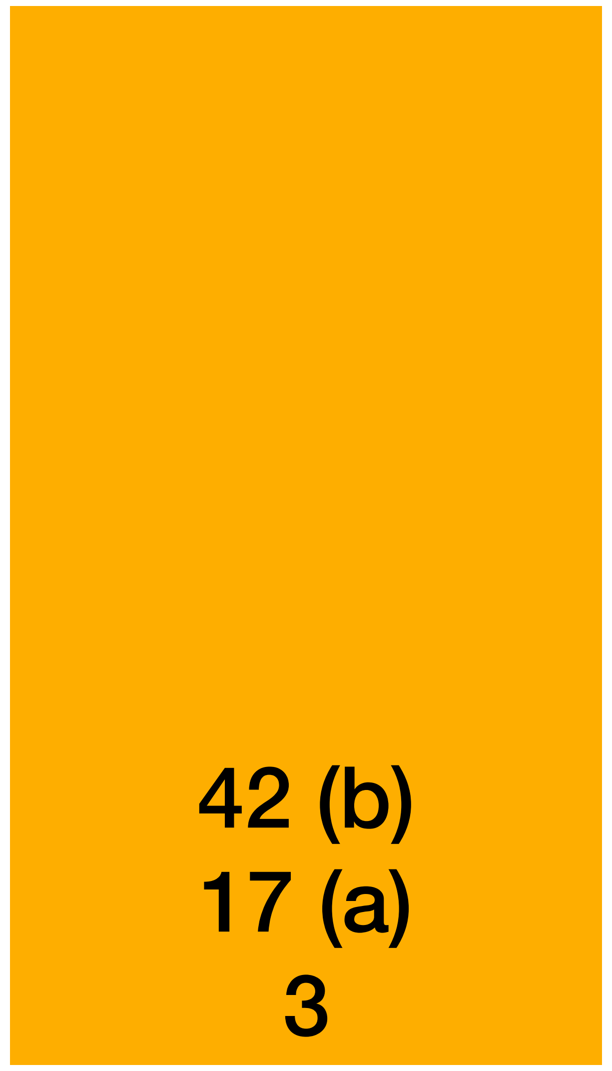

the stack

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```




the stack

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```


the stack

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```


the stack

# Stack-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3

push a

push b

send +

send /

send ceiling
```


the stack

# Register Based IRs

- Operands are exchanged through *explicit registers*
- Registers **need not** to be machine registers
  - => Registers represent *variables*
  - that could eventually be mapped to actual registers

```
ADD A, B, 1
```

opcode

registers

- Registers could be fixed vs *infinite*
- Registers could be physical vs virtual

# Three-Address-Code

## TAC, 3AC

- Instruction have generally three parameters: destination and sources
- Many notations you'll find in the wild:

```
A := B + C
```

I like this notation! (less to think...)

- Destination could be at the left (aka intel notation)

```
ADD A, B, C
```

- Destination could be at the right (aka AT&T notation)

```
ADD B, C, A
```

# Two-Address-Code

`A := A + B`

## 2AC, two-address-instruction

- One of the sources is also destination

  Meaning you'll overwrite one register always!

- Destination could be at the left (aka intel notation)

```
ADD A, B
```

- Destination could be at the right (aka AT&T notation)

```
ADD B, A
```

The rest of these slides
we will use 3AC

# Register-Based Bytecode

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```

```
T1 := A + B
T2 := 3 / T1
T3 := T2 SEND ceiling
```

# Executing Register Based Bytecode

## Overview

- Each variable is mapped to a physical register or memory position
- In an **interpreter**, a method activation could
  - pre-allocate in memory one slot per register
  - the size is bound to the max depth of the execution stack
- In a **compiler**, we will require *register allocation* algorithms
  - => for a class on itself

# From Stack to Registers

## Overview

- We simulate the stack-based execution
- Each position in the stack represents a register
- Pushing to the stack creates sets values to registers
- Popping from the stack gives us the register to use

# From Stack To Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```

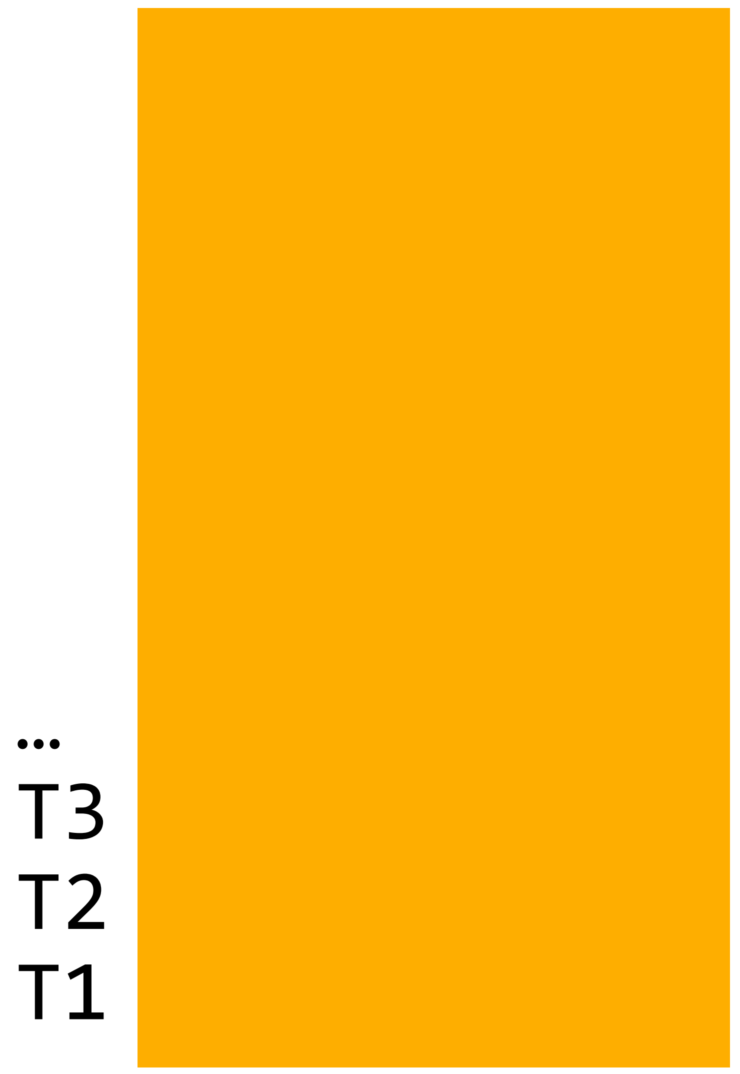
```
T1 := A + B
T2 := 3 / T1
T3 := T2 SEND ceiling
```

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```



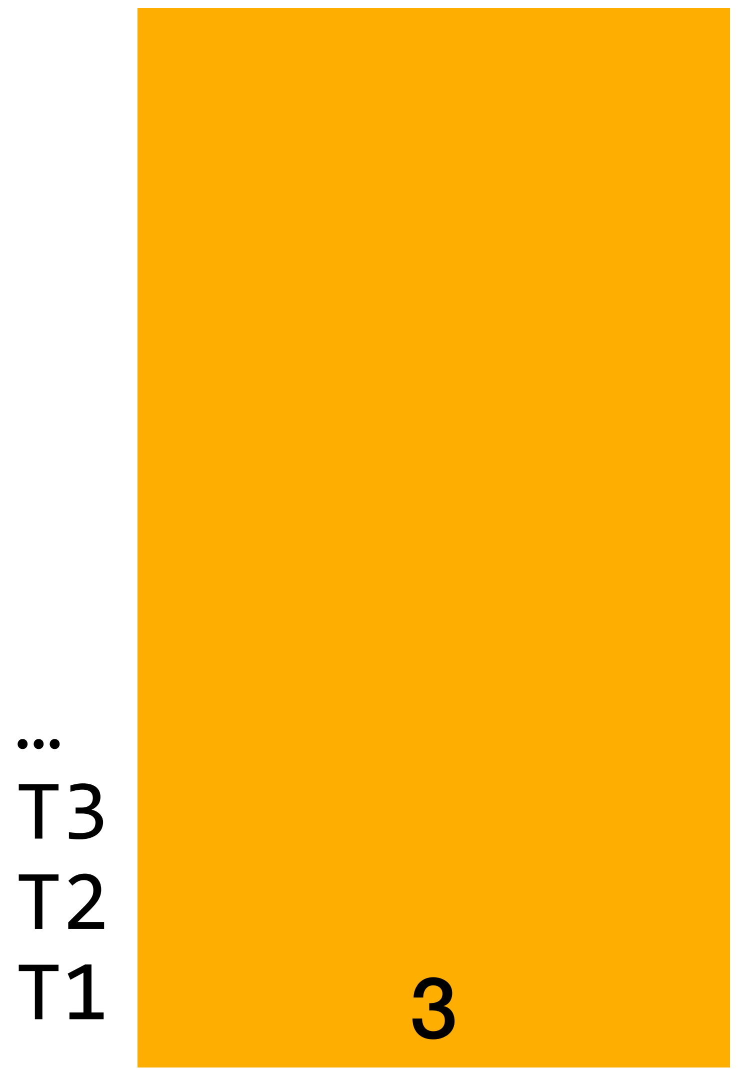

the stack

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```



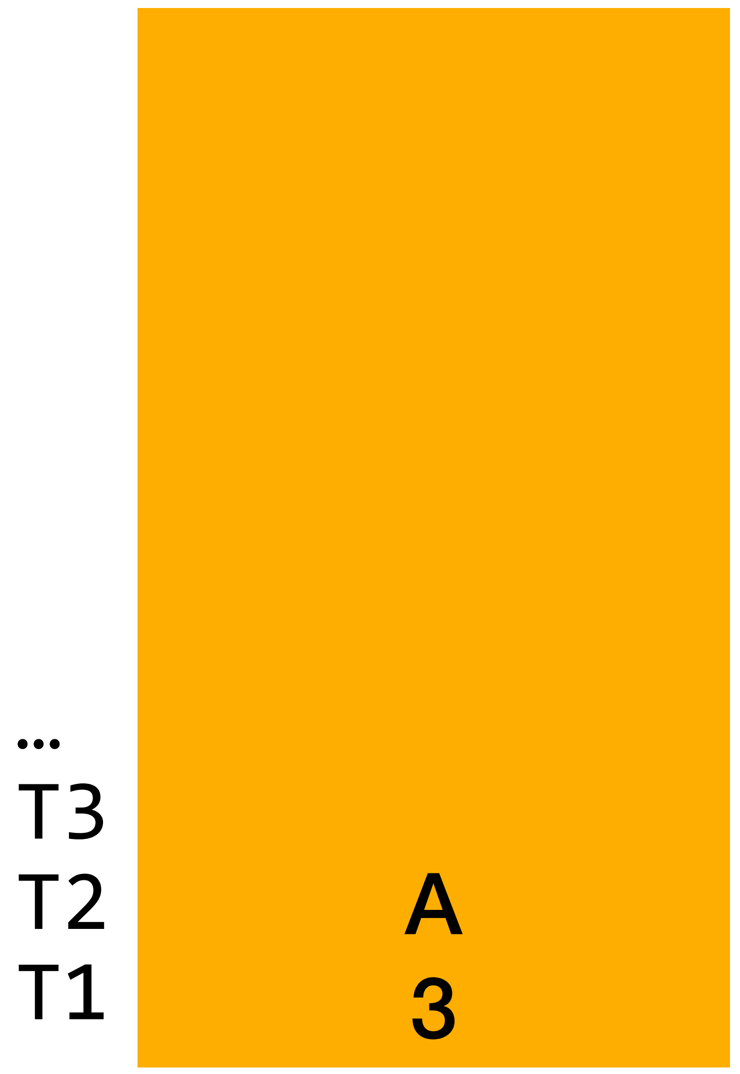

the stack

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```



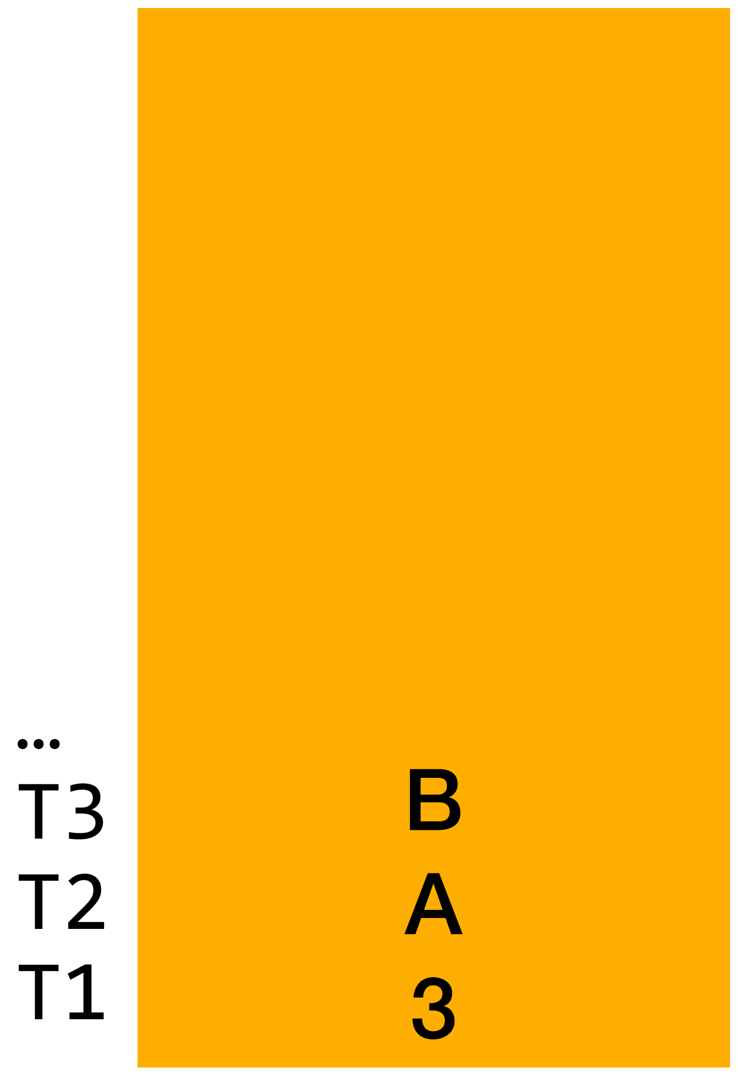

the stack

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```



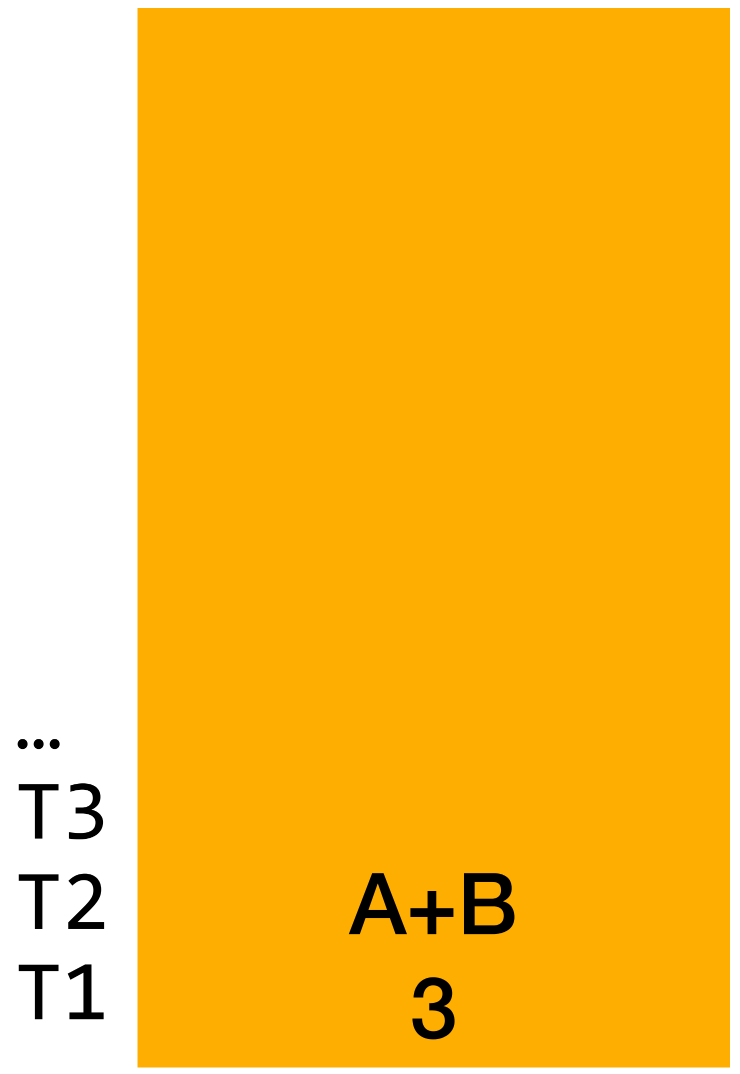
the stack

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```




the stack

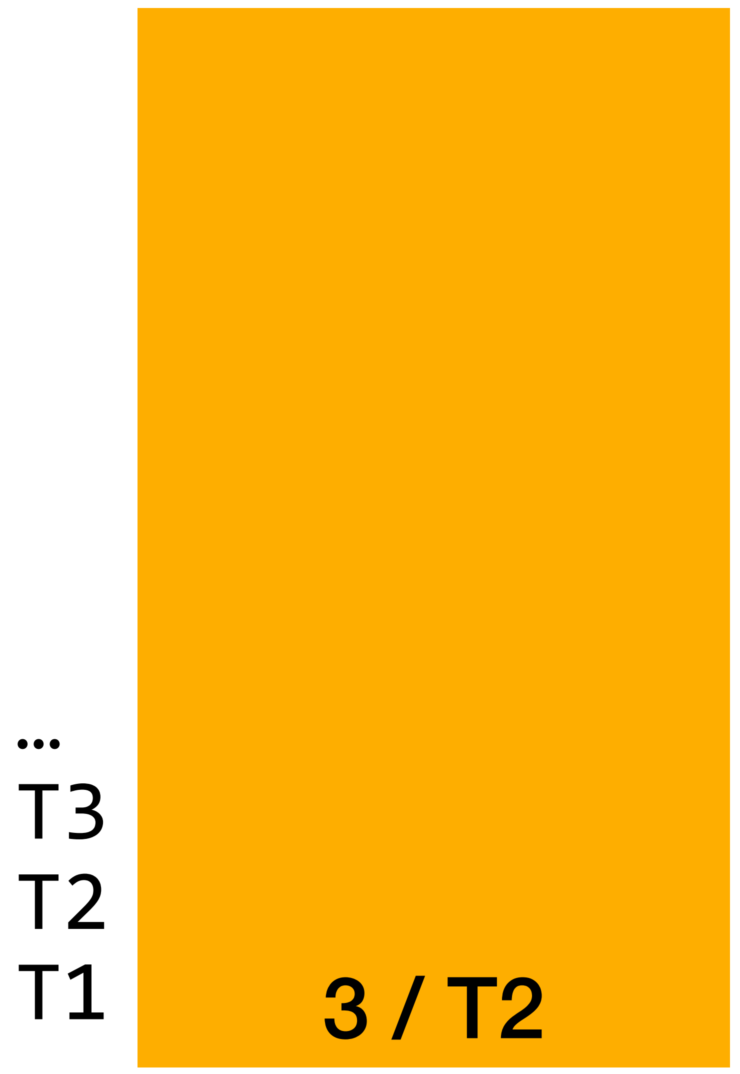
```
T2 := A + B
```

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```




the stack

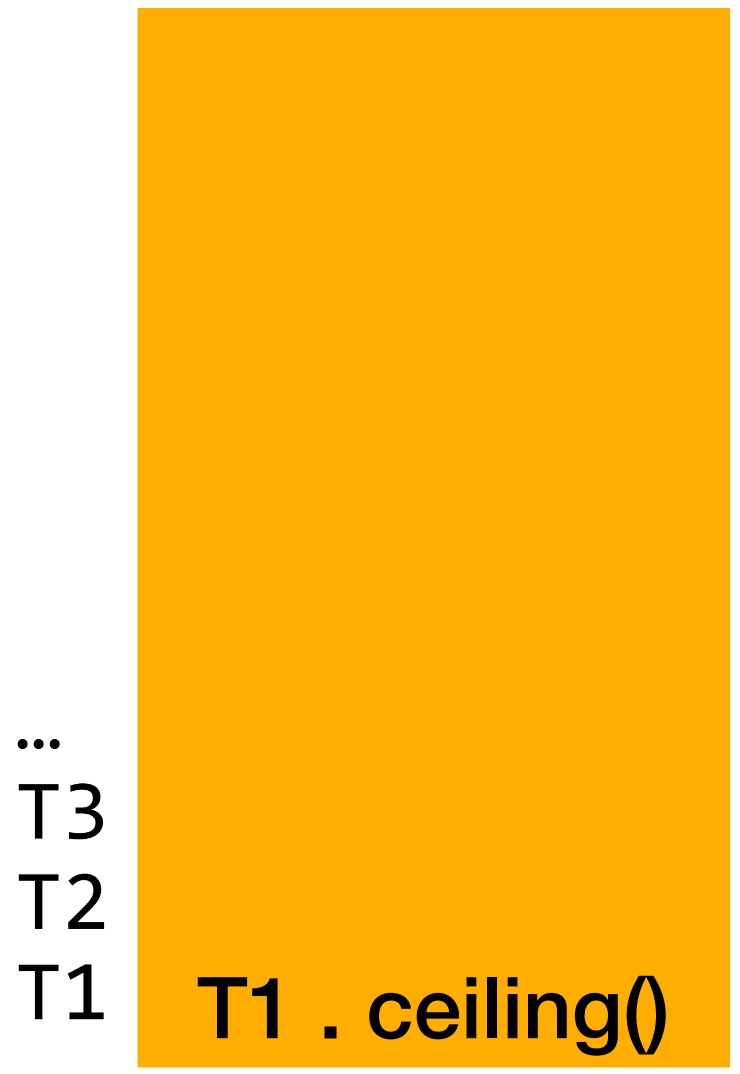
```
T2 := A + B
T1 := 3 / T2
```

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```




the stack

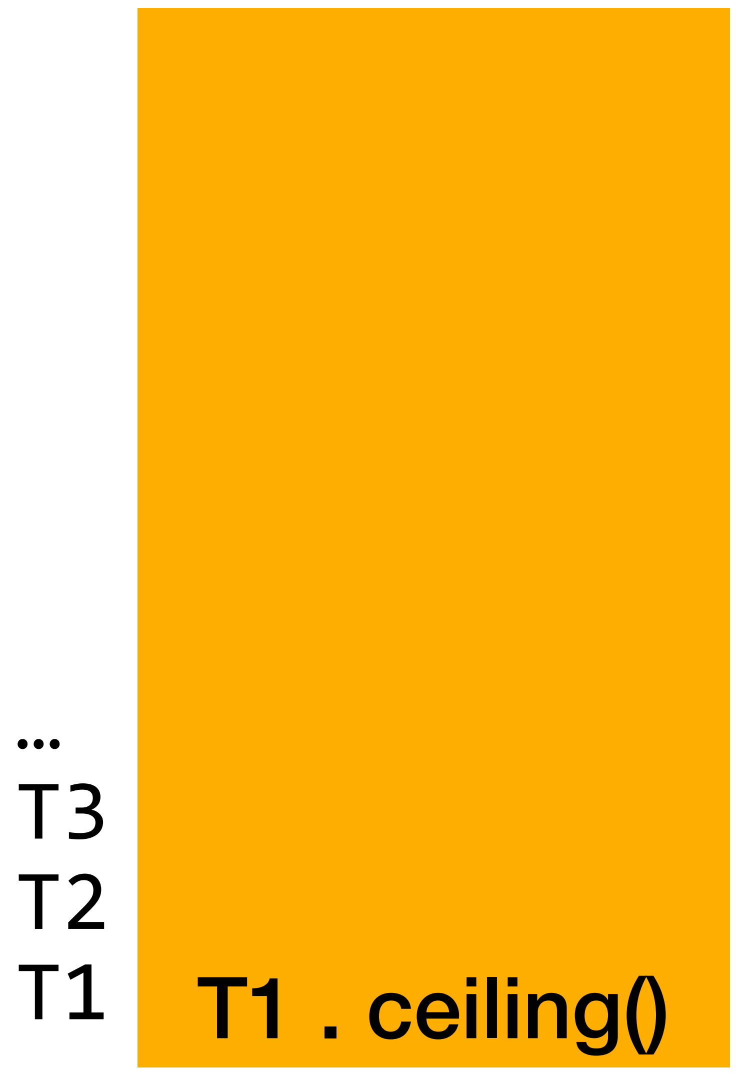
```
T2 := A + B
T1 := 3 / T2
T1 := T1 SEND ceiling
```

# From Stack to Registers

## An example

```
(3 / (a + b)) . ceiling()
```

```
push 3
push a
push b
send +
send /
send ceiling
```




the stack

```
T2 := A + B
T1 := 3 / T2
T1 := T1 SEND ceiling
```

# Single Static Assignment Form

## Aka SSA Form

- A Register IR Form where
  - Every variable is assigned only once
  - Values are merged with *phi functions*

```
T2 := A + B

T1 := 3 / T2

T1 := T1 SEND ceiling
```

Not SSA

```
T1 := A + B

T2 := 3 / T1

T3 := T2 SEND ceiling
```

SSA :) !

# SSA Benefits

## Aka SSA Form

- Every variable is assigned only once!
- Simplify code analyses
    - Register allocation
    - Dead code analysis
    - Instruction dependencies
    - …

```
T1 := A + B

T2 := 3 / T1

T3 := T2 SEND ceiling
```

SSA :) !

# SSA and Phi functions

## A Quick Intro

```
a < b
  ifTrue:  [ c := 1 ]
  ifFalse: [ c := 7 ].
return c
```

- Each assignment is mapped to a different variable
- At *merge points* we insert phi functions

```
  T1 := A < B
  JumpIfTrue T1 truePath
  C1 := 7
  Jump end
truePath:
  C2 := 1
end:
  C3 := phi (C1, C2)
  Return C3
```

# Encoding of Register Based IRs

- Less instructions than stack-based: no need to push to stack
- Fatter instructions than stack-based: arguments are explicit

- Easier to map to machine code!
    - Kind of IR inside a machine code compiler

# Conclusion

- Register based IRs are alternatives to stack-based IRs
- Operands are explicit, registers represent variables
- We can design IRs with infinite or fixed, physical or virtual registers
- Three-address-code has explicit destination
- Two-address-code has implicit destination

- SSA is a special form that simplifies manipulations!