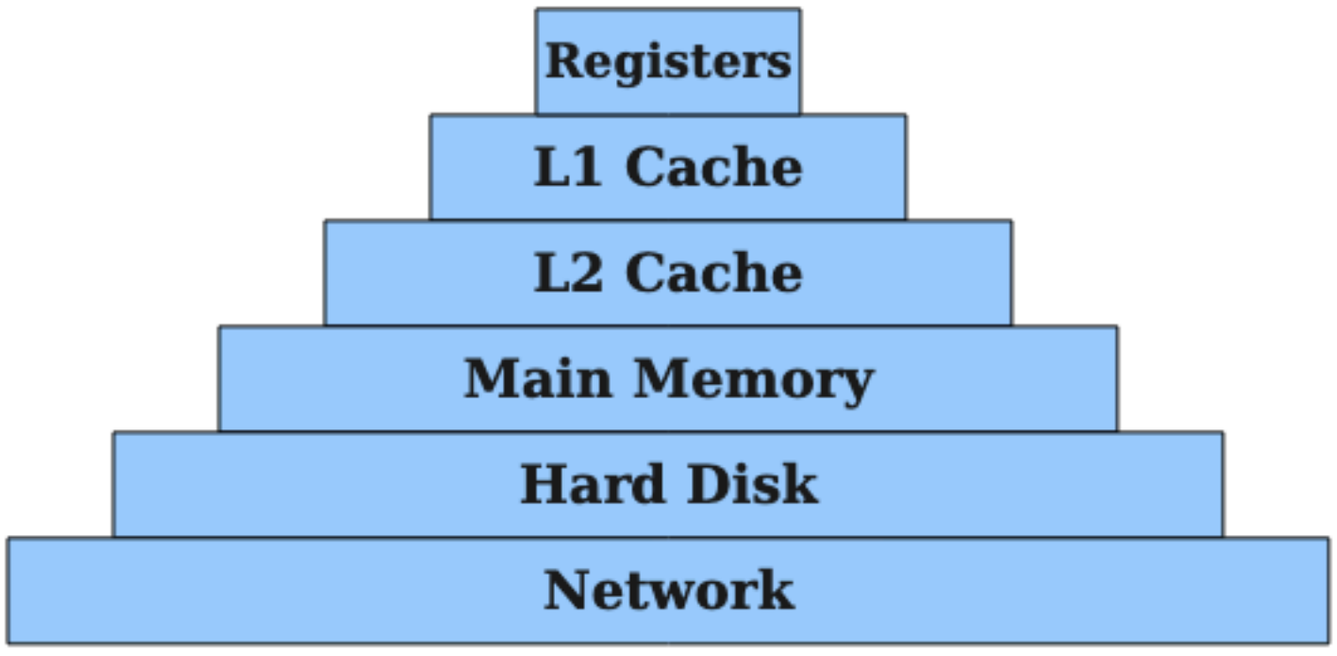
Register Allocation Part I: Liveness and Conflicts

Oct 10, 2023

Memory Hierarchy

faster, smaller

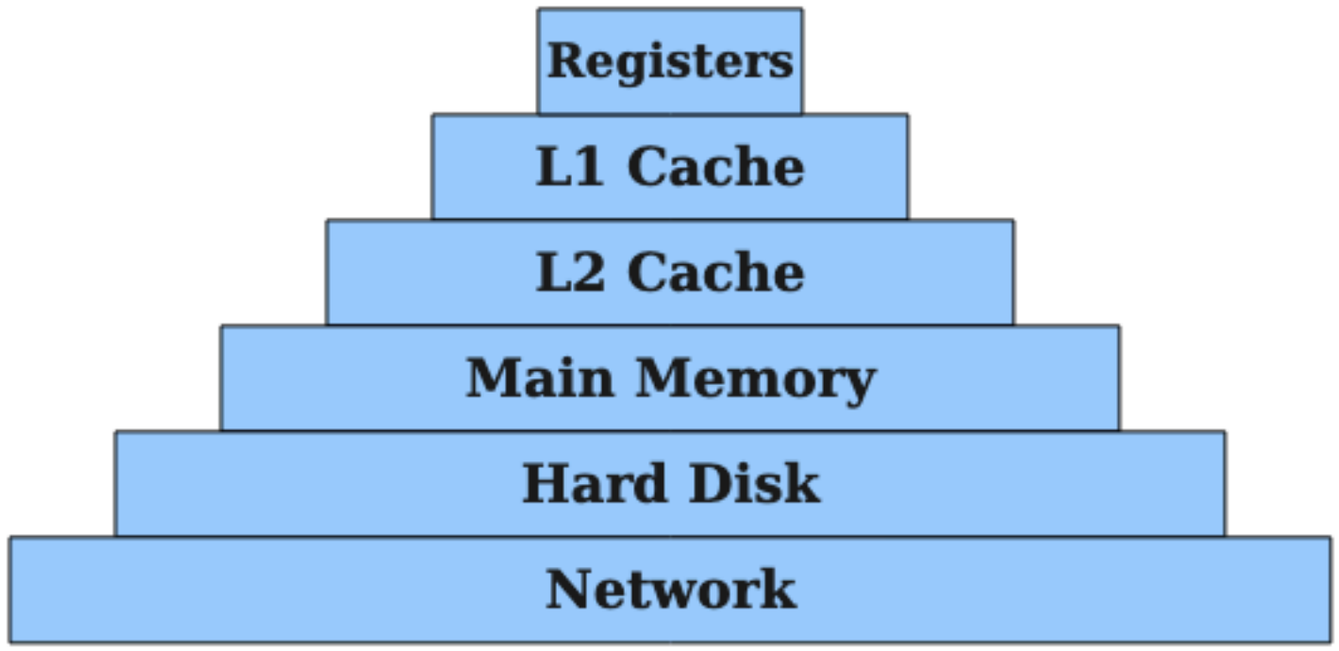


256B - 8KB	0.25 – 1ns
16KB – 64KB	1ns – 5ns
1MB - 4MB	5ns – 25ns
4GB – 256GB	25ns – 100ns
500GB+	3 – 10ms
HUGE	10 – 2000ms

slower, bigger

Memory Hierarchy

Systems view of memory:



Snake Program view of memory

variables?

Register Allocation

Goal:

- Associate each variable in the program to a memory location
  - Register if possible, stack if necessary

Current strategy:

- All variables on the stack (easy, but slow)

Performance gains:

- 3-10x+ faster variable accesses
- Space gain: smaller stack frames

High computational complexity: often the slowest part of the compiler

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

```
Currently:
a: stack
x: stack (rbp - 8)
y: stack (rbp - 16)
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

```
With register
alloc:
a: stack
x: rax
y: rax
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

```
With register
alloc:
a: stack
x: rax
y: rax
```

```
mov rax, [rbp + ..]
sar rax, 1
imul rax, 4
add rax, 14
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  g(x, y)
end
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  g(x, y)
end
```

```
With register
alloc:
a: stack
x: rax
y: rcx
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  g(x, y)
end
```

```
With register
alloc:
a: stack
x: rax
y: rcx
```

Can’t put x and y in the same register
Say they are **in conflict** or **interfering**

Register Allocation

3(.5) Steps

1. **Liveness analysis**: identify when each variable's value is needed in the program
2. **Conflict analysis**: identify which variables interfere with each other
3. **Graph Coloring**: assign variables to registers so that interfering registers are assigned different registers.
    1. Spilling: if necessary, assign some variables to stack slots

Shadowing

```
def f(a):
  let x = a * 2 in
  let y = let x = 14 in x + 7 in
  f(x, y)
end
```

Shadowing

```
def f(a):
  let x = a * 2 in
  let y = let x = 14 in x + 7 in
  f(x, y)
end
```

two different “x” are in conflict here

Shadowing

```
def f(a):
  let x0 = a * 2 in
  let y = let x1 = 14 in x1 + 7 in
  f(x0, y)
end
```

Before reg allocation:
make all variable names unique.

Register Allocation

When to do register allocation?

After sequentialization and lambda lifting

Register Allocation vs Calling Conventions

Register Allocation and all of our analyses will take place on a **per function** basis. We distinguish between

- **internal** calls (always tail calls, subject to register allocation)
- **external** calls (possibly non-tail, use a pre-determined calling convention)

Register Allocation vs Calling Conventions

Register Allocation and all of our analyses will take place on a **per function** basis. We distinguish between

- **internal** calls (always tail calls, subject to register allocation)
- **external** calls (possibly non-tail, use a pre-determined calling convention)

```
def f(a):
  def loop(i): e
  end
  let j = g(x,y)
  in loop(i)
end
```

```
def f(a):
  def loop(i): e
  end
  let j = ecall(g; x,y)
  in icall(loop;i)
end
```

Register Allocation vs Calling Conventions

```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      i' = i - 1;
      a' = a + x;
      icall(loop; i', a')
end
icall(loop; y, 0)
```

Register Allocation vs Calling Conventions

```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      i' = i – 1;
      a' = a + x;
      icall(loop; i', a')
end
icall(loop; y, 0)
```

x,y,z: predetermined by CC

i, a, i', a': subject to reg alloc

Liveness Analysis

Goal: determine for every sub-expression, which variables are "live".

Liveness Analysis

Goal: determine for every sub-expression, which variables are "live".

Definition: a variable **x** is live in an expression **e** if the observable behavior of **e** depends on the value of **x.**

Liveness Analysis

Goal: determine for every sub-expression, which variables are "live".

Definition: a variable **x** is live in an expression **e** if the observable behavior of **e** depends on the value of **x.**

intuition: work "backwards" from uses and propagate them up the AST

Liveness Example

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
    y * x
end
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { }
2: { }
3: { }
4: { }
5: { }
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { }
2: { }
3: { }
4: { }
5: { x, y }
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { }
2: { }
3: { }
4: { x }
5: { x, y }
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { }
2: { }
3: { x }
4: { x }
5: { x, y }
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { }
2: { a }
3: { x }
4: { x }
5: { x, y }
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { a }
2: { a }
3: { x }
4: { x }
5: { x, y }
```

Liveness Example

```
def f(a):
  1          2
   let x = a * 2 in
  3          4
   let y = x + 7 in
   5
   y * x
end
```

```
1: { a }
2: { a }
3: { x }
4: { x }
5: { x, y }
```

x and y are in conflict in expression 5, need separate registers

Liveness Example

is x live?

Liveness Example

is x live?

x

Liveness Example

is x live?

x

Yes

Liveness Example

is x live?

```
x * y
```

Liveness Example

is x live?

x * y

Yes

Liveness Example

is x live?

```
if b: x else: y
```

Liveness Example

is x live?

```
if b: x else: y
```

not clear without more context

Liveness Example

is x live?

```
let b = true in
if b: x else: y
```

Liveness Example

is x live?

```
let b = true in
if b: x else: y
```

yes

Liveness Example

is x live?

```
let b = false in
if b: x else: y
```

Liveness Example

is x live?

```
let b = false in
if b: x else: y
```

no

Liveness Example

is x live?

```
let b = read_input() in
if b: x else: y
```

Liveness Example

is x live?

```
let b = read_input() in
if b: x else: y
```

yes

Liveness Example

is x live?

```
let b = read_input() in
if b: x else: y
```

yes

Liveness Example

is x live?

```
let b = read_input() in
if b: x else: y
```

Limitation: Computability

Determining correct liveness information can be arbitrarily complicated...

Limitation: Computability

Determining correct liveness information can be arbitrarily complicated...

Rice's Theorem:

Any non-trivial semantic property of programs in a Turing-complete language is undecidable

Determining liveness of variables is undecidable!

Limitation: Computability

Determining **correct** liveness information can be arbitrarily complicated...

Limitation: Computability

Determining **correct** liveness information can be arbitrarily complicated...

What if we determined **incorrect** liveness information sometimes?

Limitation: Computability

Determining **correct** liveness information can be arbitrarily complicated...

What if we determined **incorrect** liveness information sometimes?

- false positives: sometimes we say a variable is live when it's not
- false negatives: sometimes we say a variable is not live when it is

Limitation: Computability

Determining **correct** liveness information can be arbitrarily complicated...

What if we determined **incorrect** liveness information sometimes?

- false positives: sometimes we say a variable is live when it's not
- false negatives: sometimes we say a variable is not live when it is

False positives are ok: we will just use more registers/space than necessary

Limitation: Computability

Goal: **Overapproximate**

The output of our liveness analysis should include every variable that is live, but possibly some that are not live.

Limitation: Computability

Goal: **Overapproximate**

The output of our liveness analysis should include every variable that is live, but possibly some that are not live.

Approach so far in class: big overapproximation

- Essentially consider all variables in scope to be live, assign them different stack slots to be safe

Limitation: Computability

Goal: **Overapproximate**

The output of our liveness analysis should include every variable that is live, but possibly some that are not live.

Approach so far in class: big overapproximation

- Essentially consider all variables in scope to be live, assign them different stack slots to be safe

We can do much better

- Only consider variables live if they actually get used
- But consider all execution paths (i.e. branches) to be possible

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...
- LIVE(if imm: e1 else: e2) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...
- LIVE(if imm: e1 else: e2) = LIVE(imm) U LIVE(e1) U LIVE(e2)

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...
- LIVE(if imm: e1 else: e2) = LIVE(imm) U LIVE(e1) U LIVE(e2)
- LIVE(let x = e1 in e2) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...
- LIVE(if imm: e1 else: e2) = LIVE(imm) U LIVE(e1) U LIVE(e2)
- LIVE(let x = e1 in e2) = (LIVE(e2) - x) U LIVE(e1)

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...
- LIVE(if imm: e1 else: e2) = LIVE(imm) U LIVE(e1) U LIVE(e2)
- LIVE(let x = e1 in e2) = (LIVE(e2) - x) U LIVE(e1)
- LIVE(icall(f; imm,...)) =

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(ecall(f; imm1,...)) = LIVE(imm1) U ...
- LIVE(if imm: e1 else: e2) = LIVE(imm) U LIVE(e1) U LIVE(e2)
- LIVE(let x = e1 in e2) = (LIVE(e2) - x) U LIVE(e1)
- LIVE(icall(f; imm,...)) = (LIVE(f.body) - f.args) U LIVE(imm1) U ...

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
    P let a' = a + x in
      icall(loop; i', a')
end
icall(loop; y, 0)
```

In the sub-expression P, which variables are

In scope:

Syntactically occurring:

Live:

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
end
icall(loop; y, 0)
```

In the sub-expression P, which variables are

In scope: `x,y,z,i,a,i'`

Syntactically occurring: `x,a,a',i'`

Live: `x,z,a,i'`

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	?
5:	{ }	?
6:	{ }	?
7:	{ }	?
8:	{ }	?

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	?
5:	{ }	?
6:	{ }	?
7:	{ }	?
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	?
5:	{ }	?
6:	{ }	?
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	?
5:	{ }	?
6:	{ }	{x,a}
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
 icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	?
5:	{ }	{x,a,i'}
6:	{ }	{x,a}
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	{i}
5:	{ }	{x,a,i'}
6:	{ }	{x,a}
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	{x,i,a}
4:	{ }	{i}
5:	{ }	{x,a,i'}
6:	{ }	{x,a}
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	{z,a}
3:	{ }	{x,i,a}
4:	{ }	{i}
5:	{ }	{x,a,i'}
6:	{ }	{x,a}
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 0	Round 1
1:	{ }	{x,z,i,a}
2:	{ }	{z,a}
3:	{ }	{x,i,a}
4:	{ }	{i}
5:	{ }	{x,a,i'}
6:	{ }	{x,a}
7:	{ }	{i',a'}
8:	{ }	{y}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	1: ?
2:	{z,a}	2: ?
3:	{x,i,a}	3: ?
4:	{i}	4: ?
5:	{x,a,i'}	5: ?
6:	{x,a}	6: ?
7:	{i',a'}	7: ?
8:	{y}	8: ?

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
 icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	?
3:	{x,i,a}	?
4:	{i}	?
5:	{x,a,i'}	?
6:	{x,a}	?
7:	{i',a'}	?
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	?
3:	{x,i,a}	?
4:	{i}	?
5:	{x,a,i'}	?
6:	{x,a}	?
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	?
3:	{x,i,a}	?
4:	{i}	?
5:	{x,a,i'}	?
6:	{x,a}	{x,a}
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3                4
      let i' = i - 1 in
      5                6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	?
3:	{x,i,a}	?
4:	{i}	?
5:	{x,a,i'}	{x,z,a,i'}
6:	{x,a}	{x,a}
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
 icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	?
3:	{x,i,a}	?
4:	{i}	{i}
5:	{x,a,i'}	{x,z,a,i'}
6:	{x,a}	{x,a}
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	?
3:	{x,i,a}	{x,z,i,a}
4:	{i}	{i}
5:	{x,a,i'}	{x,z,a,i'}
6:	{x,a}	{x,a}
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3                 4
      let i' = i - 1 in
      5               6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	?
2:	{z,a}	{z,a}
3:	{x,i,a}	{x,z,i,a}
4:	{i}	{i}
5:	{x,a,i'}	{x,z,a,i'}
6:	{x,a}	{x,a}
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3                 4
      let i' = i - 1 in
      5               6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
 icall(loop; y, 0)
```

	Round 1	Round 2
1:	{x,z,i,a}	{x,z,i,a}
2:	{z,a}	{z,a}
3:	{x,i,a}	{x,z,i,a}
4:	{i}	{i}
5:	{x,a,i'}	{x,z,a,i'}
6:	{x,a}	{x,a}
7:	{i',a'}	{x,z,i',a'}
8:	{y}	{x,y,z}

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 2	Round 3
1:	{x,z,i,a}	1: ?
2:	{z,a}	2: ?
3:	{x,z,i,a}	3: ?
4:	{i}	4: ?
5:	{x,z,a,i'}	5: ?
6:	{x,a}	6: ?
7:	{x,z,i',a'}	7: ?
8:	{x,y,z}	8: ?

Liveness Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

	Round 2	Round 3
1:	{x,z,i,a}	{x,z,i,a}
2:	{z,a}	{z,a}
3:	{x,z,i,a}	{x,z,i,a}
4:	{i}	{i}
5:	{x,z,a,i'}	{x,z,a,i'}
6:	{x,a}	{x,a}
7:	{x,z,i',a'}	{x,z,i',a'}
8:	{x,y,z}	{x,y,z}

Implementation Concerns

How to store live sets?

- Use the Ann
- `init_liveness(e: Exp<T>) -> Exp<HashSet<String>>`
- `update_liveness(e: Exp<HashSet<String>>) -> Exp<HashSet<String>>`
- iterate until you reach a fixed point
  - `update_liveness(e) == e`

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time?

```
def f(y):
  let y = x in
  x + y
end
```

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables **truly** conflict when

- They are live at the same time
- with different values

Again, **over**approximate

false positives (spurious conflicts) are ok

false negatives (missing conflicts) are not

Conflict Analysis: Algorithm

Input: A top-level function definition, annotated with liveness information in every subexpression

Ouput: Conflict graph

Nodes: all variables that are in scope in any subexpression of the function

Edges: conflicts

Algorithm:

At each point where a variable x is written,

Add a conflict between x and every y that is live immediately after the write

Unless x and y can be proven to have the same value

Conflict Analysis: Algorithm

Algorithm:

- At each point where a variable x is written,
- Add a conflict between x and every y that is live immediately after the write
- Unless x and y can be proven to have the same value

When are variables written to?

- Let bindings
- Function calls

Conflict Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
 end
8
 icall(loop; y, 0)
```

Liveness Info

1: {x,z,i,a}

2: {z,a}

3: {x,z,i,a}

4: {i}

5: {x,z,a,i'}

6: {x,a}

7: {x,z,i',a'}

8: {x,y,z}

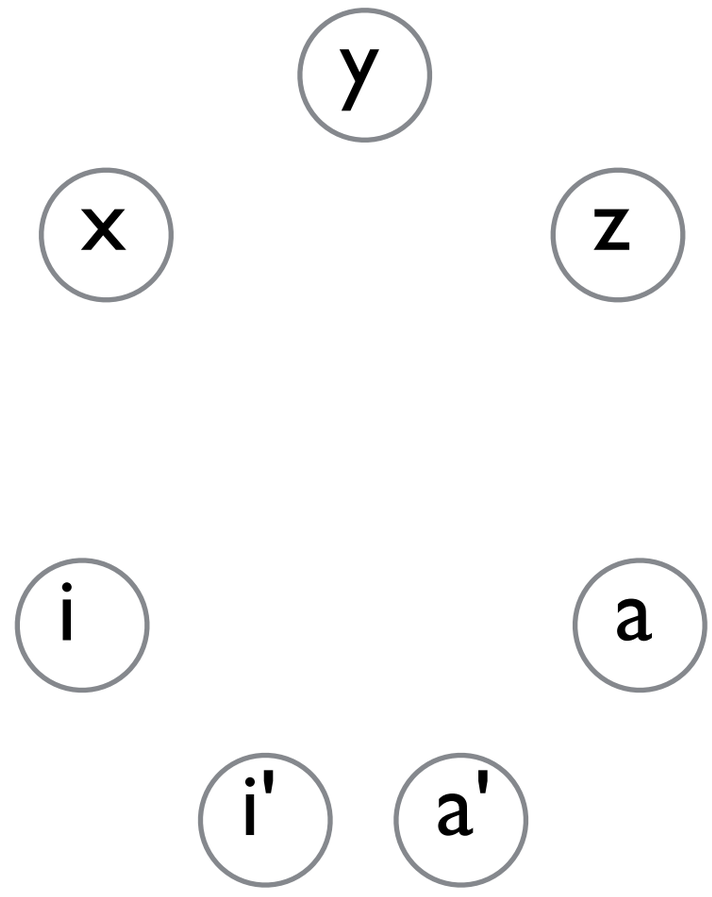
Conflict Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3                 4
      let i' = i - 1 in
      5               6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
  icall(loop; y, 0)
```

Liveness Info

```
1: {x,z,i,a}
2: {z,a}
3: {x,z,i,a}
4: {i}
5: {x,z,a,i'}
6: {x,a}
7: {x,z,i',a'}
8: {x,y,z}
```

Interference Graph



Initialize with all variables

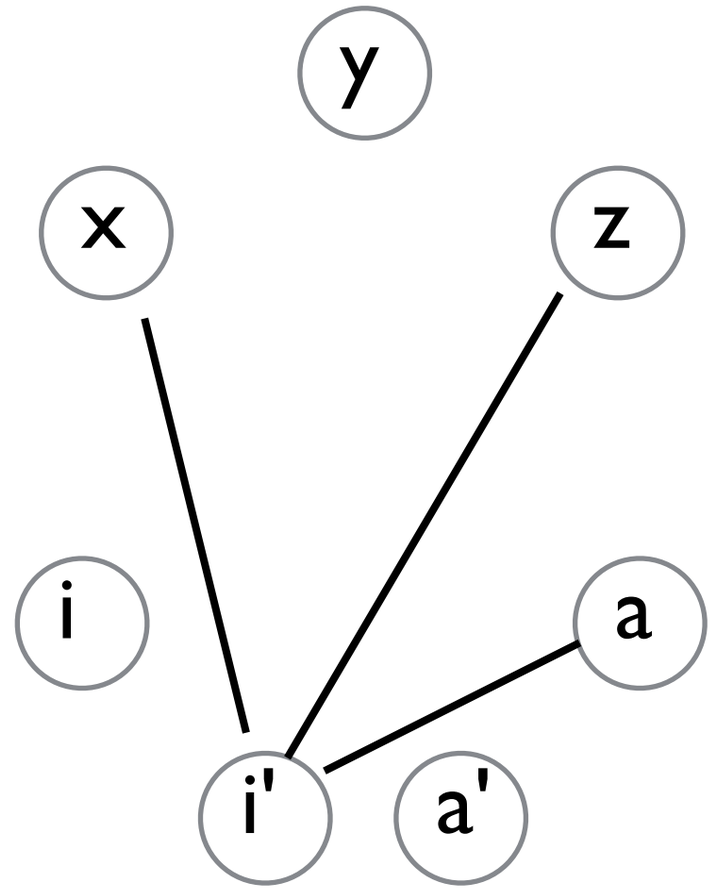
Conflict Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3                 4
      let i' = i - 1 in
      5                 6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

Liveness Info

1: {x,z,i,a}
2: {z,a}
3: {x,z,i,a}
4: {i}
5: {x,z,a,i'}
6: {x,a}
7: {x,z,i',a'}
8: {x,y,z}

Interference Graph



[3]let i' = ...

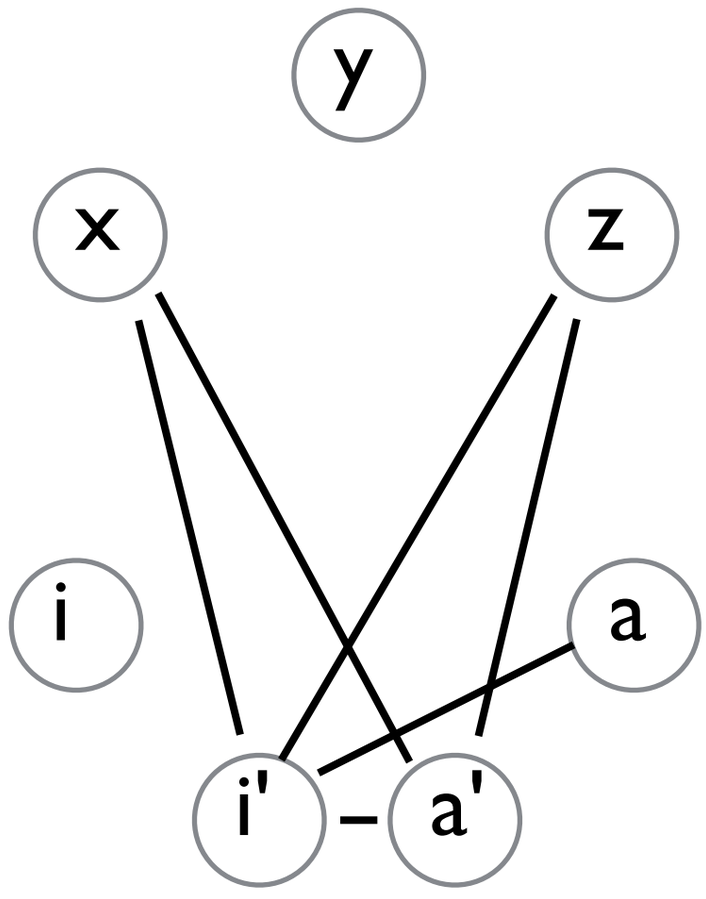
Conflict Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3                 4
      let i' = i - 1 in
      5               6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

Liveness Info

1: {x,z,i,a}
2: {z,a}
3: {x,z,i,a}
4: {i}
5: {x,z,a,i'}
6: {x,a}
7: {x,z,i',a'}
8: {x,y,z}

Interference Graph



5 let a' = ...

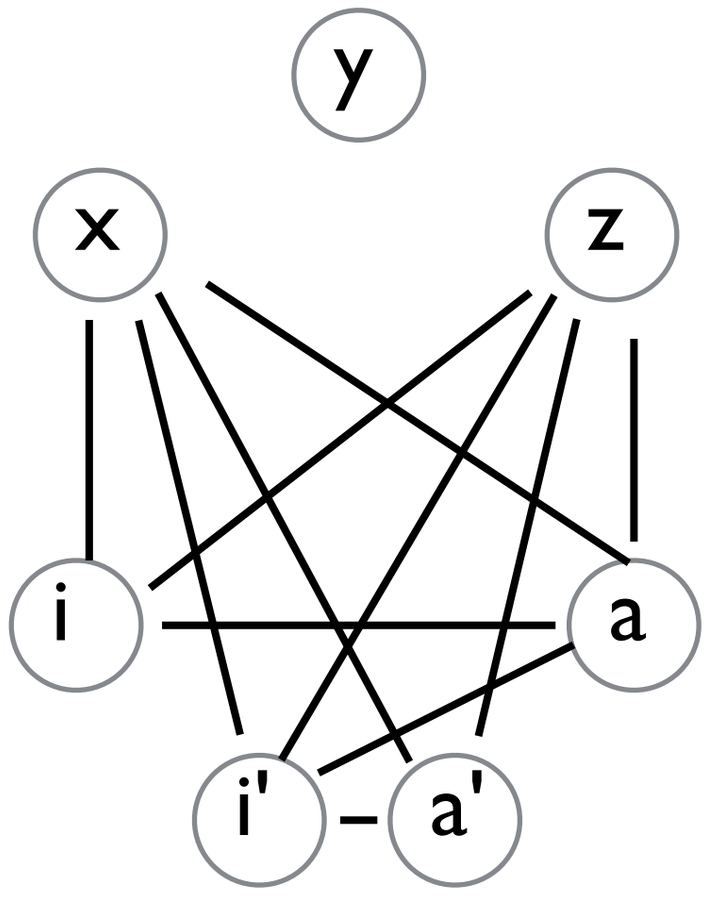
Conflict Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3              4
      let i' = i - 1 in
      5              6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
icall(loop; y, 0)
```

Liveness Info

```
1: {x,z,i,a}
2: {z,a}
3: {x,z,i,a}
4: {i}
5: {x,z,a,i'}
6: {x,a}
7: {x,z,i',a'}
8: {x,y,z}
```

Interference Graph



def loop(i, a):

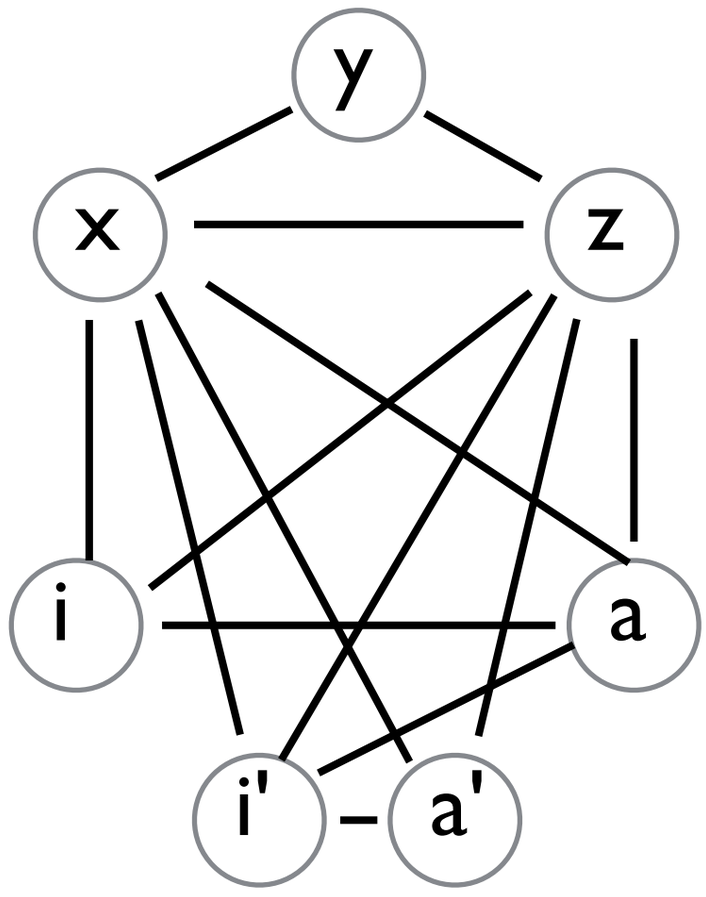
Conflict Analysis

```
def f(x,y,z):
  def loop(i,a):
    1
    if i == 0:
      2
      a * z
    else:
      3          4
      let i' = i - 1 in
      5          6
      let a' = a + x in
      7
      icall(loop; i', a')
  end
8
  icall(loop; y, 0)
```

Liveness Info

1: {x,z,i,a}
2: {z,a}
3: {x,z,i,a}
4: {i}
5: {x,z,a,i'}
6: {x,a}
7: {x,z,i',a'}
8: {x,y,z}

Interference Graph



def f(x,y,z):

Summary so Far

For each top level function in the program

1. Liveness Analysis annotates each expression with which variables are live within it.
2. Conflict Analysis produces a **conflict graph** whose nodes are variables and edges are conflicts (the variables cannot share a register)
3. Next time: Use this conflict graph to assign registers to variables, and generate more efficient code