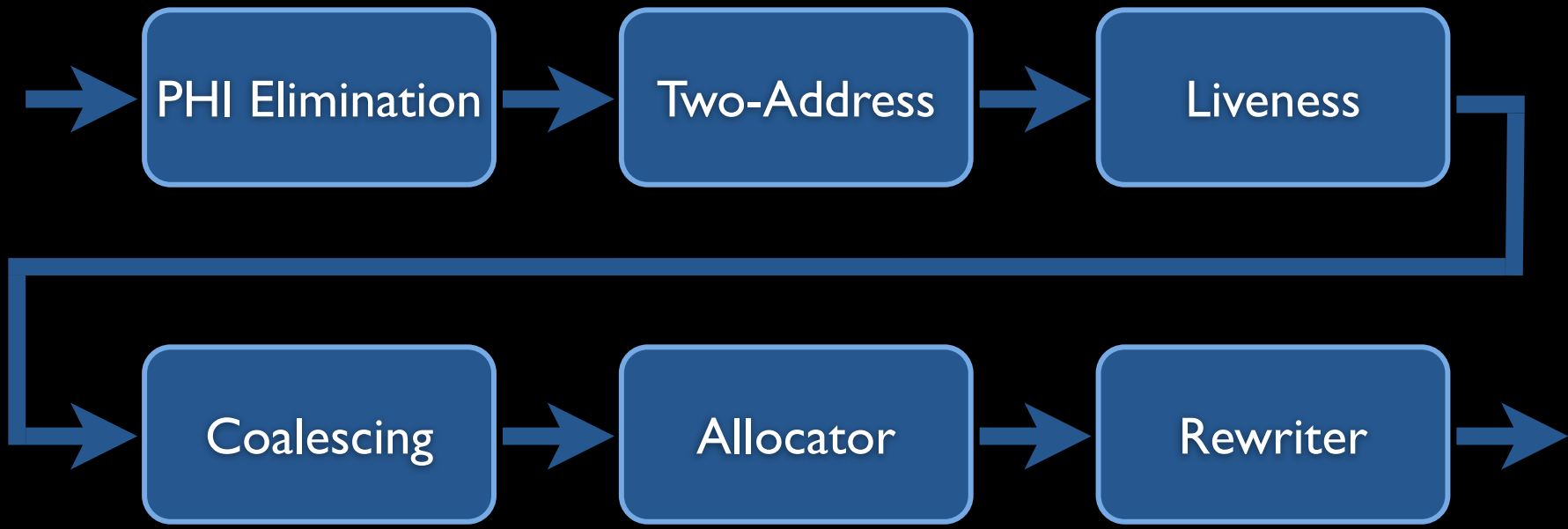
Future Works in LLVM Register Allocation

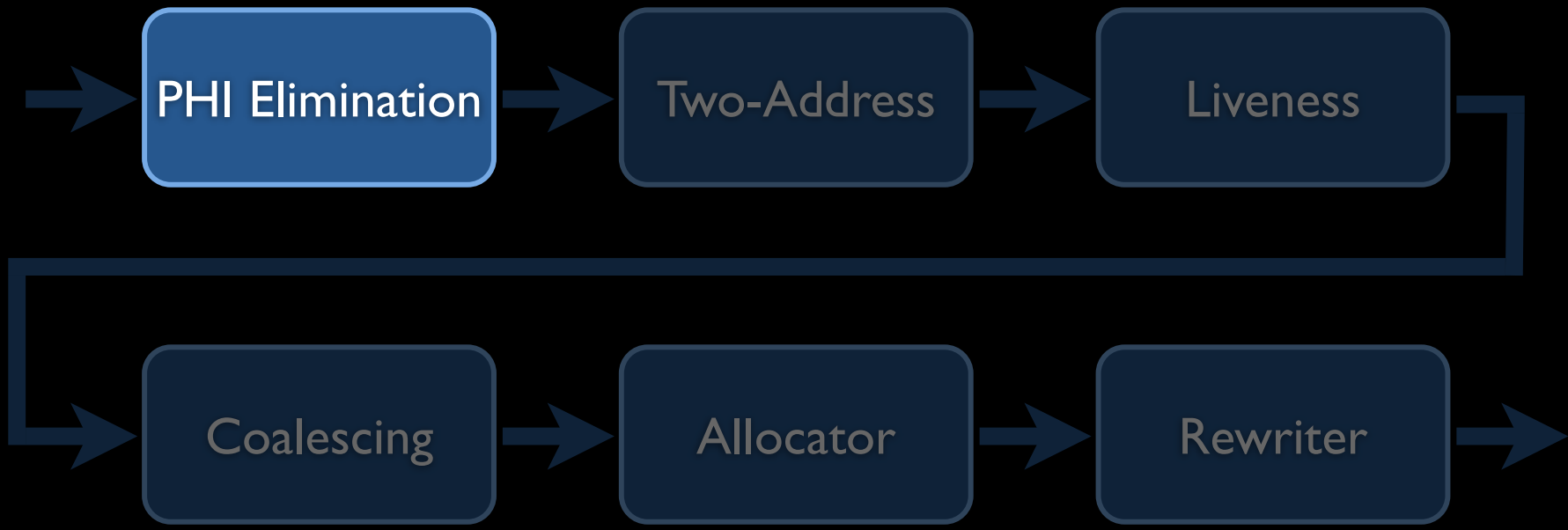
Talk Overview

1. Introduction
2. Upcoming Changes
3. PBQP

Register Allocation in LLVM

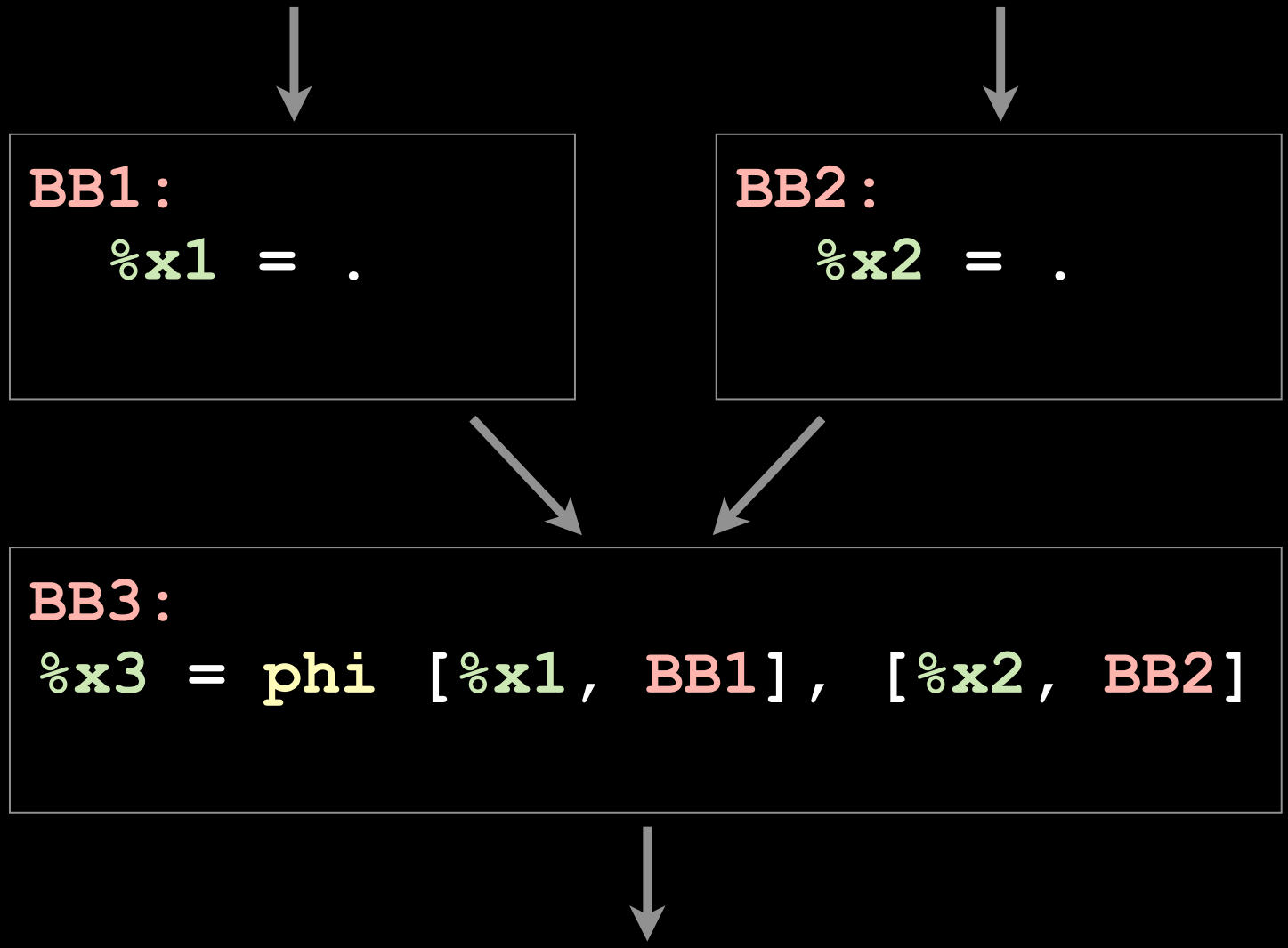


Register Allocation in LLVM

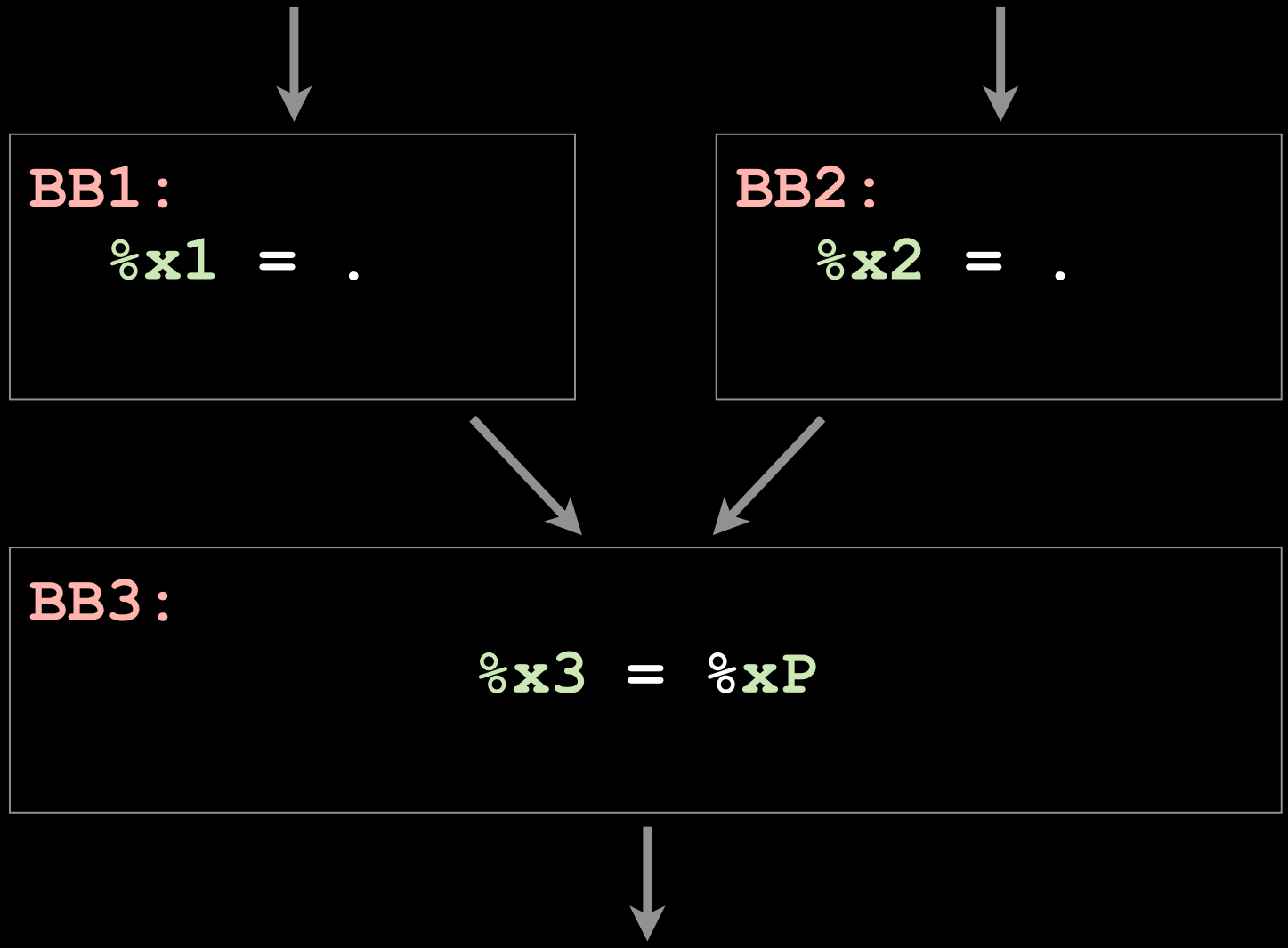


Lower PHI-instructions to copies

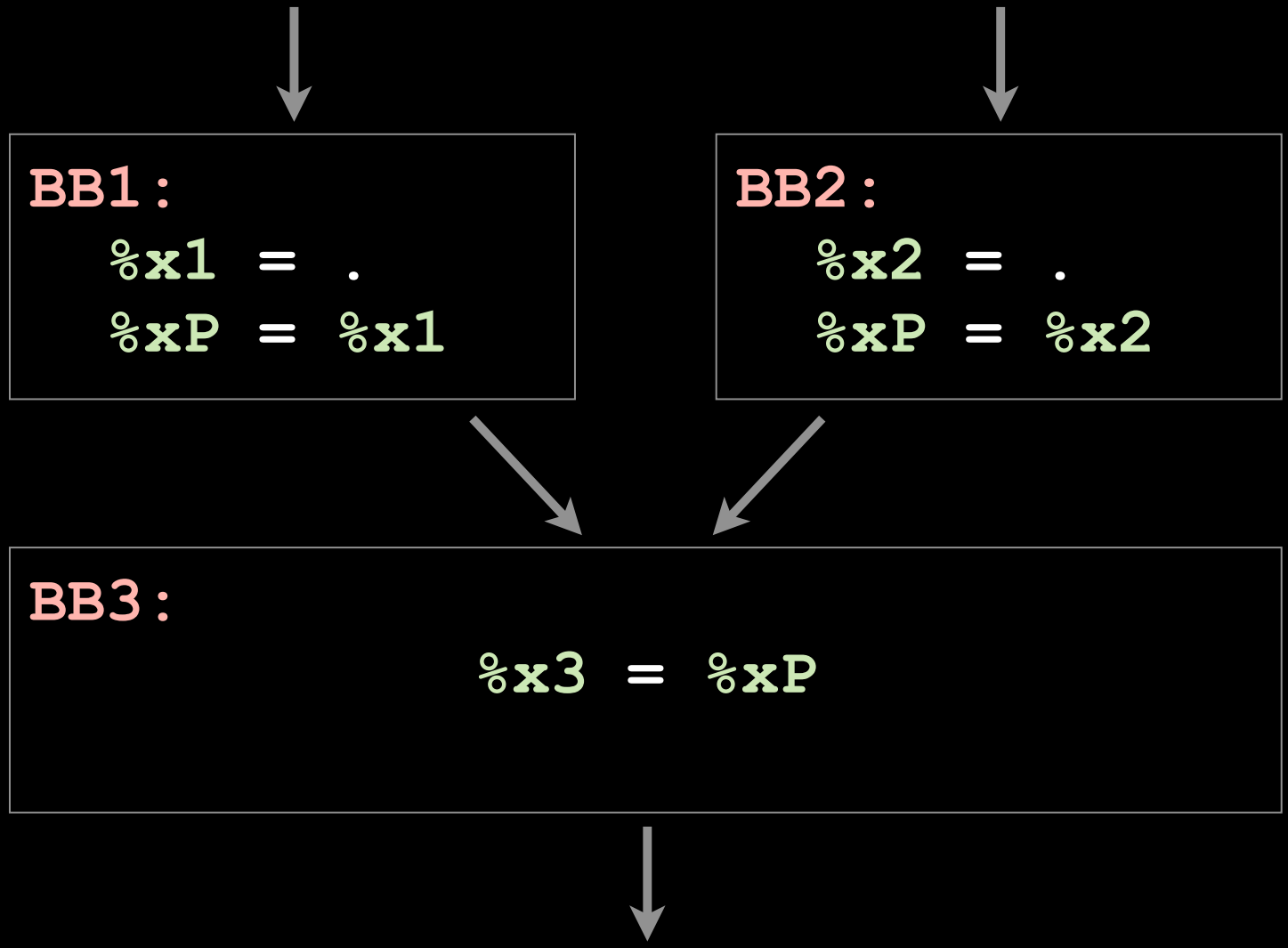
PHI Lowering



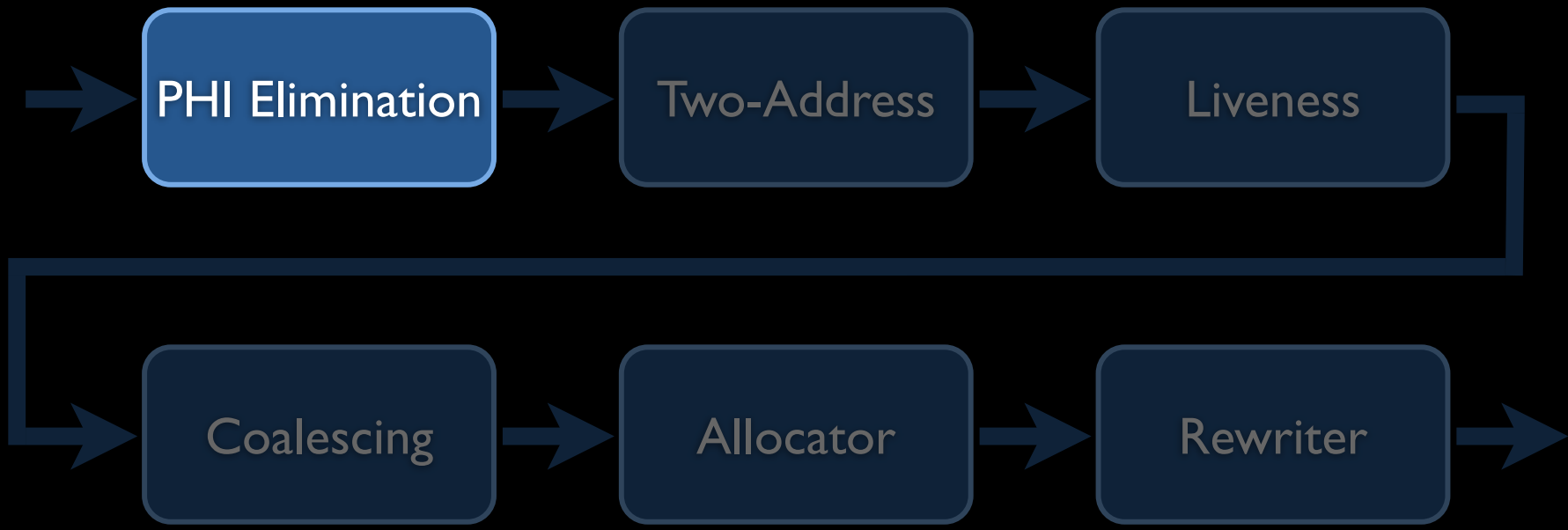
PHI Lowering



PHI Lowering

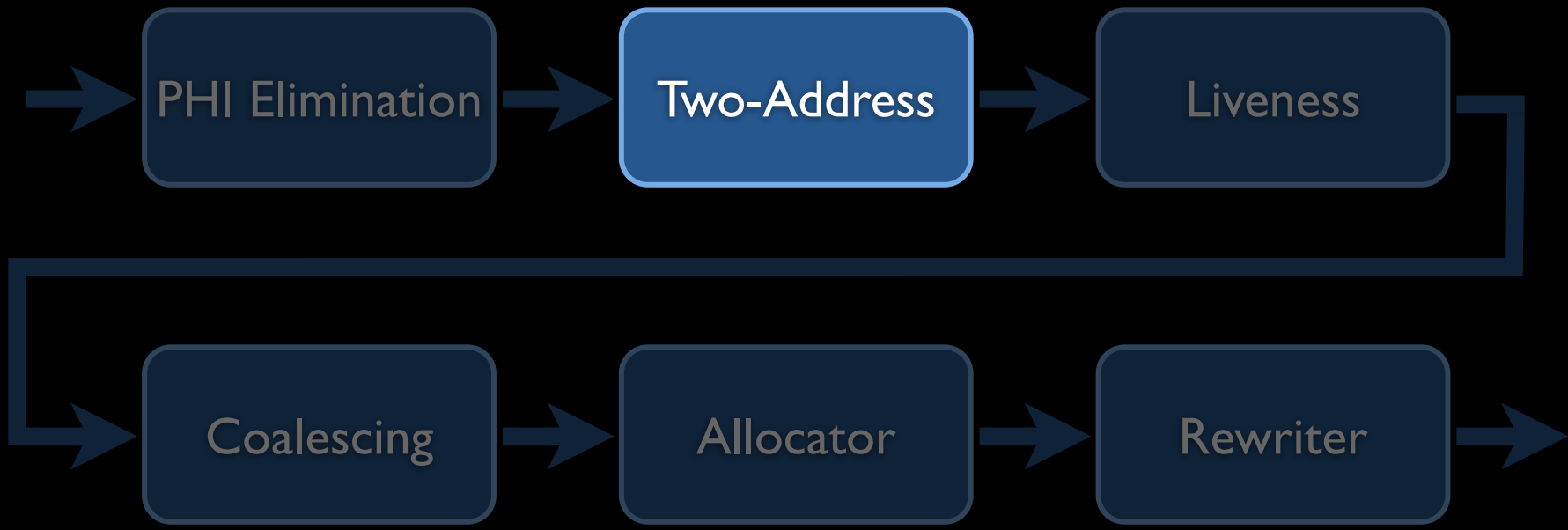


Register Allocation in LLVM



Lower PHI-instructions to copies

Register Allocation in LLVM



Lower Three-Address Instructions

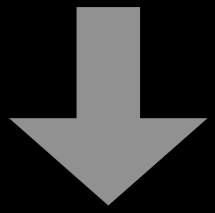
Two-Address Instructions

Two-Address Instructions

```
x3 = x2 + x1
```

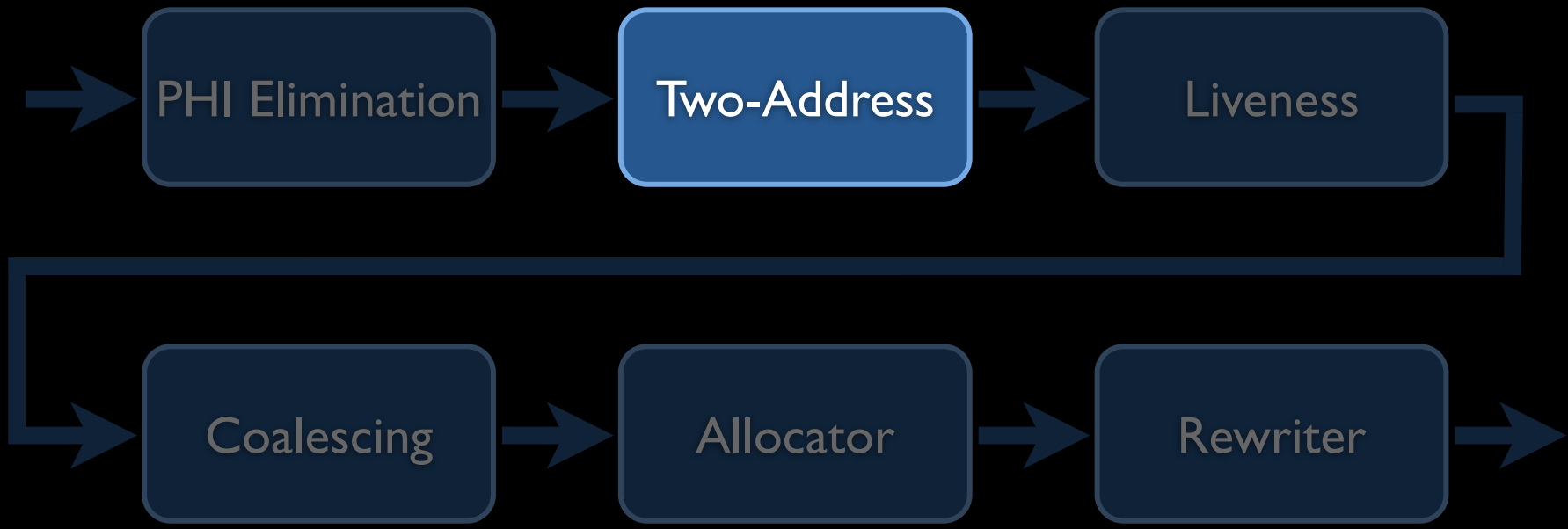
Two-Address Instructions

```
x3 = x2 + x1
```



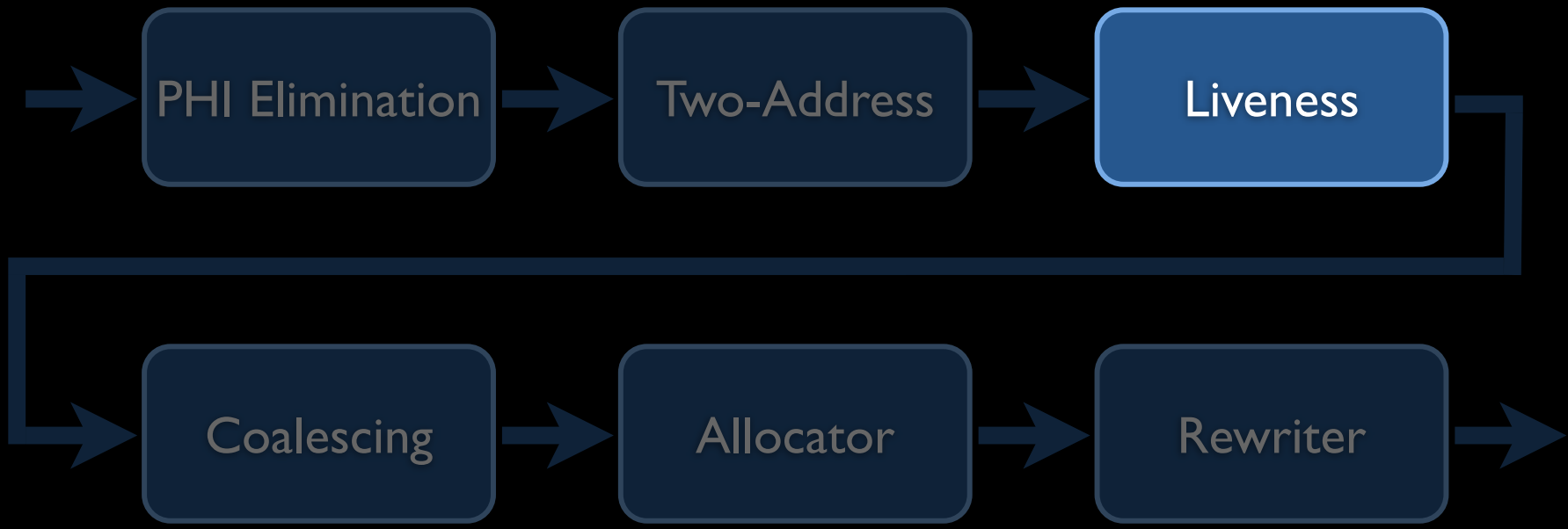
```
x3 = x2
x3 += x1
```

Register Allocation in LLVM



Lower three-address instructions

Register Allocation in LLVM



Construct live intervals

Live Intervals

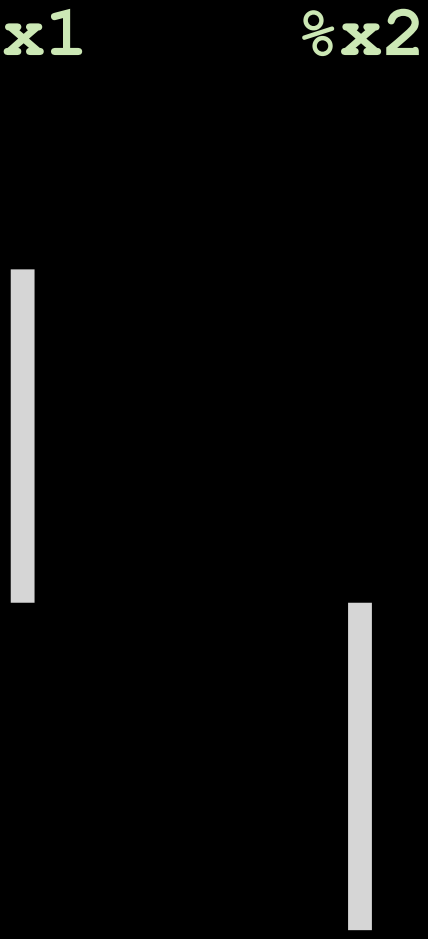
```
BB:
     %x1 = ...
           .
           .
           .
     %x2 = %x1
           .
           .
           .
     ... = %x2
```

Live Intervals

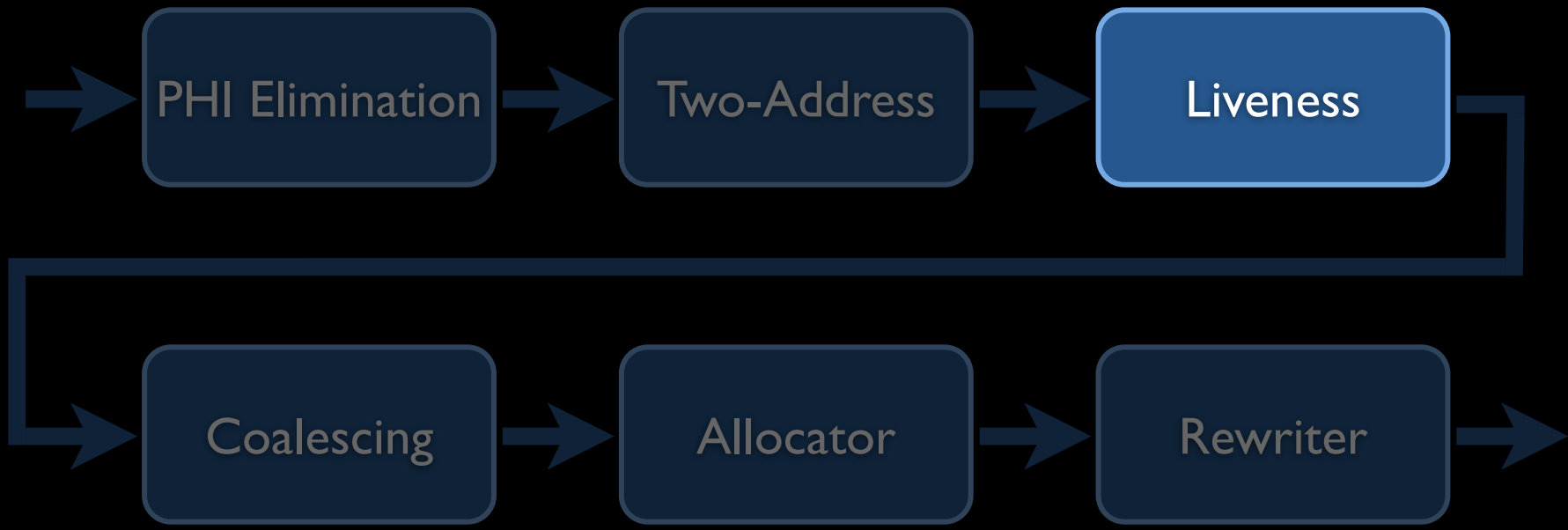
```
BB:
     %x1 = ...
           .
           .
           .
     %x2 = %x1
           .
           .
           .
     ... = %x2
```

Live Intervals

```
BB:
     %x1 = ...
           .
           .
           .
     %x2 = %x1
           .
           .
           .
     ... = %x2
```

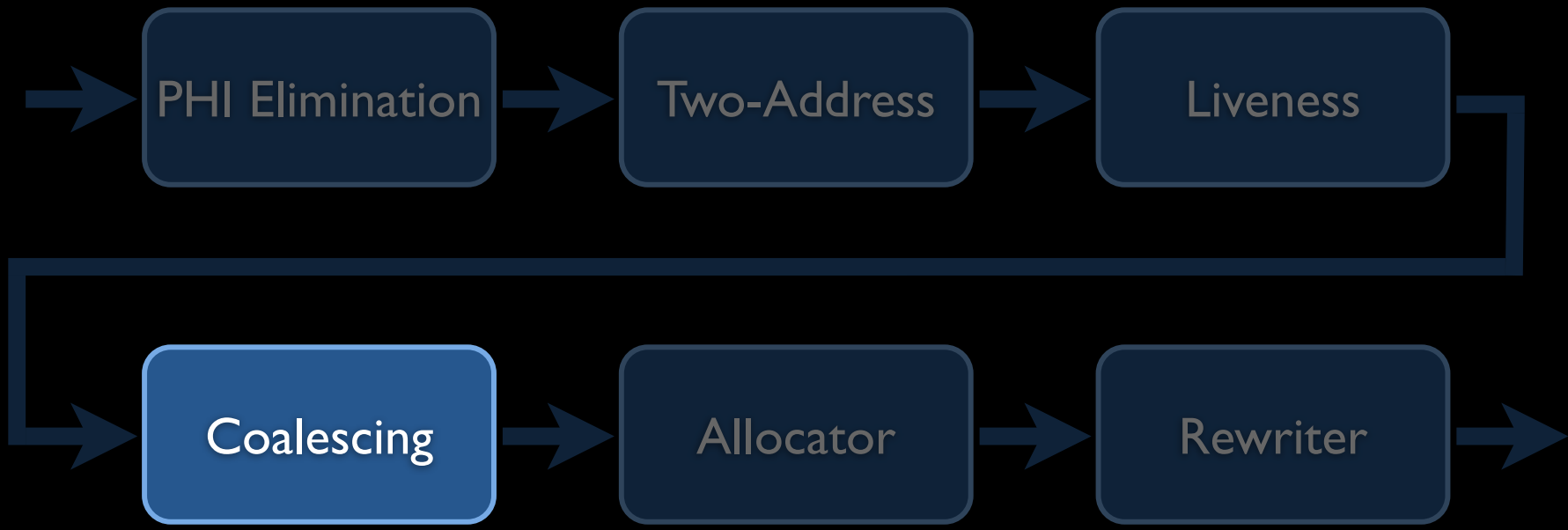


Register Allocation in LLVM



Construct Live Intervals

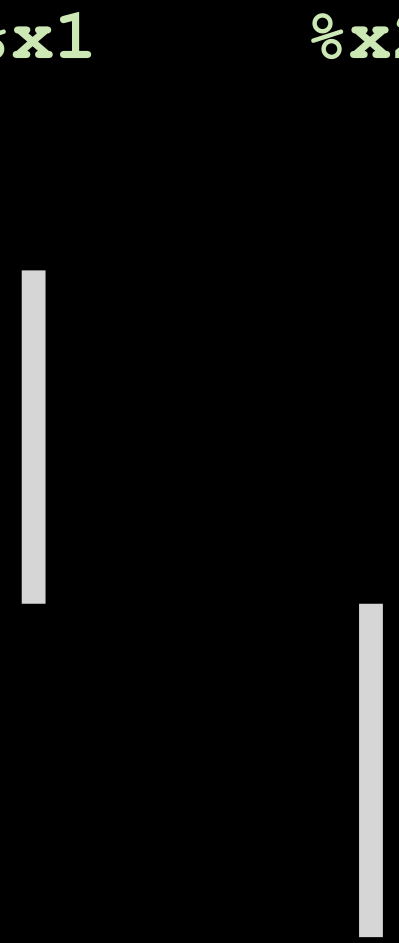
Register Allocation in LLVM



Aggressively eliminate copies

Coalescing

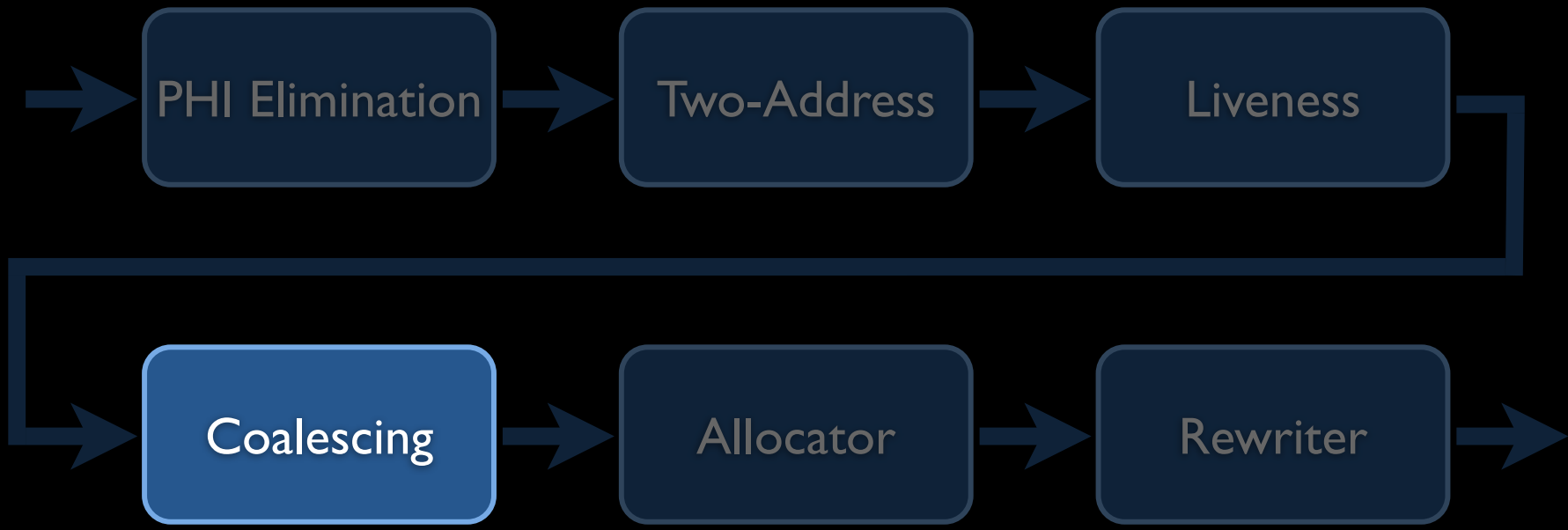
```
BB:
     %x1 = ...
           .
           .
           .
     %x2 = %x1
           .
           .
           .
     ... = %x2
```



Coalescing

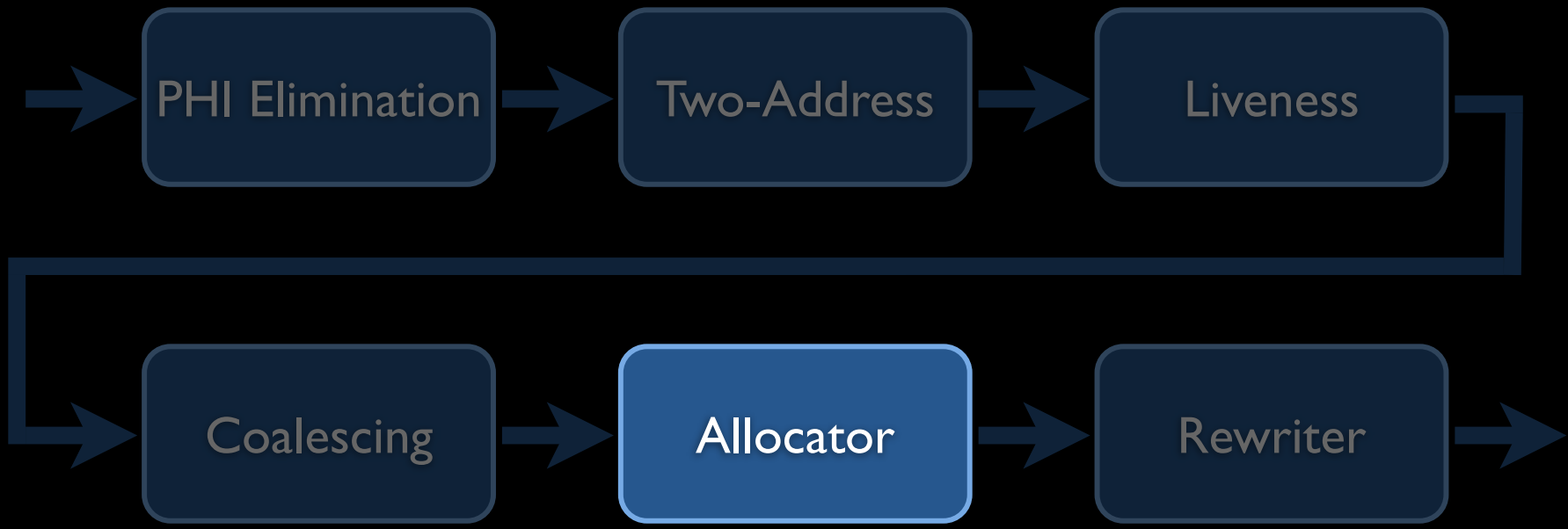
```
BB:
     %x1 = ...
           .
           .
           .
           .
           .
           .
     ... = %x1
```

Register Allocation in LLVM



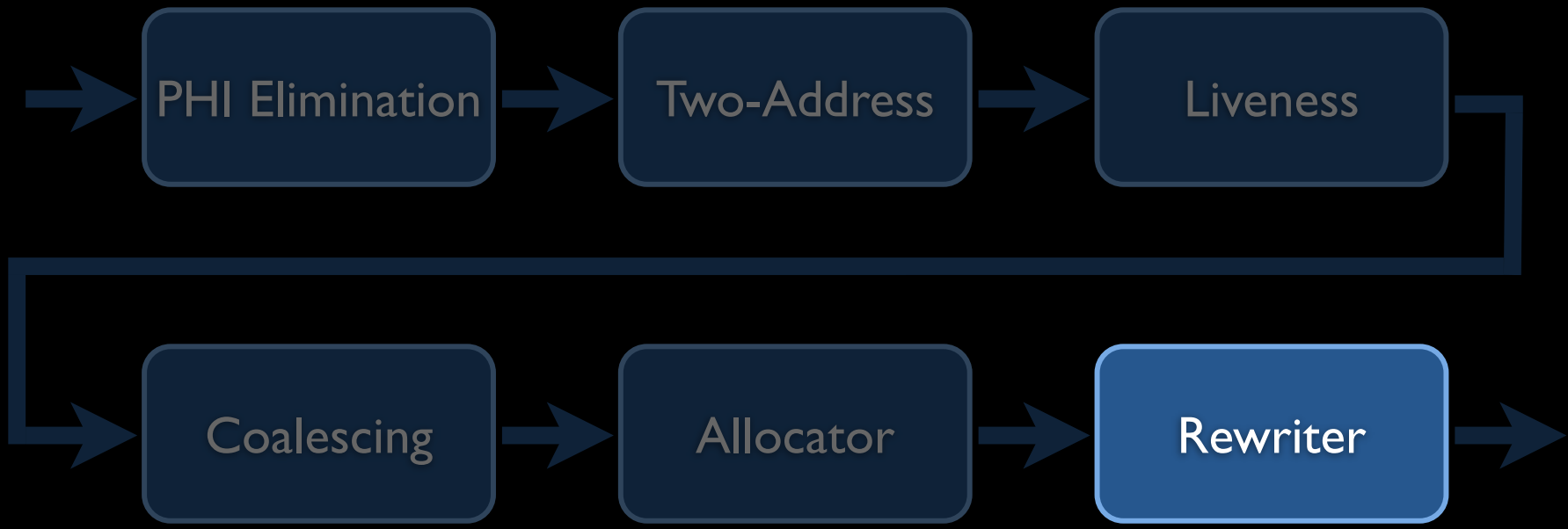
Aggressively eliminate copies

Register Allocation in LLVM



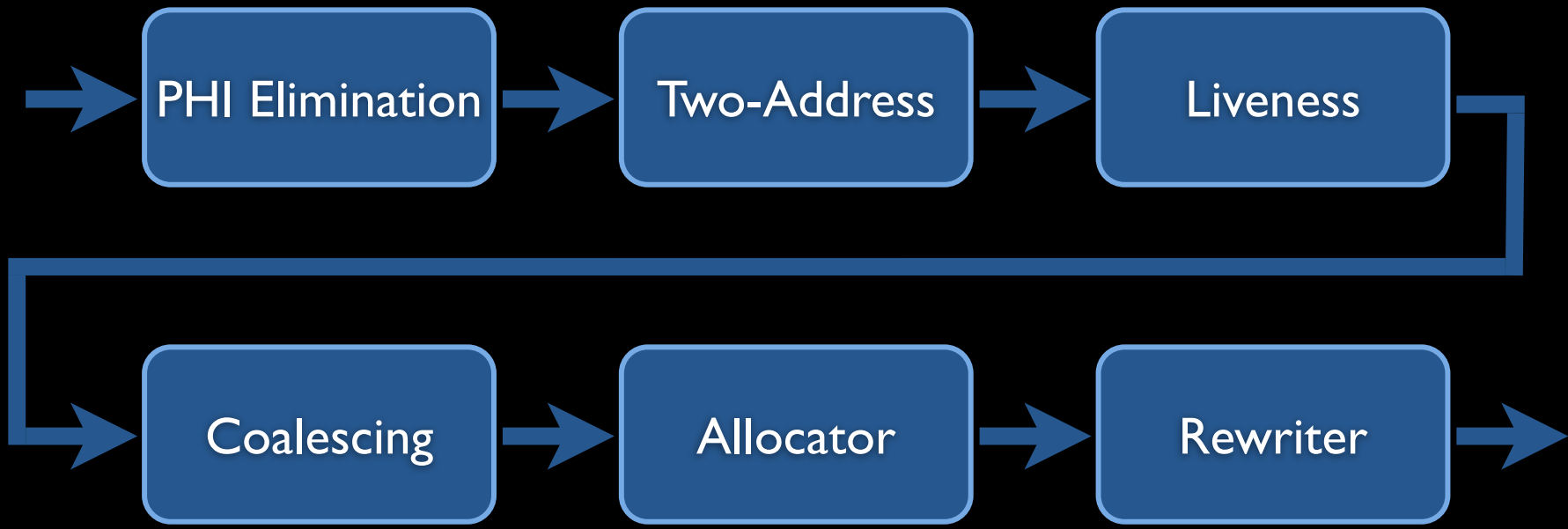
Compute register assignment

Register Allocation in LLVM



Apply register assignment

Register Allocation in LLVM



Improvements

- New and better optimizations
- New allocators
- Cleaner infrastructure

1. Optimizations

Rematerialization

Rematerialization

```
vr1  = <expr>
  .  // stuff
 ... = vr1
  .  // more stuff
 ... = vr1
```

Rematerialization

```
vr1  = <expr>
.    // stuff
...  = vr1
.    // more stuff
...  = vr1
```

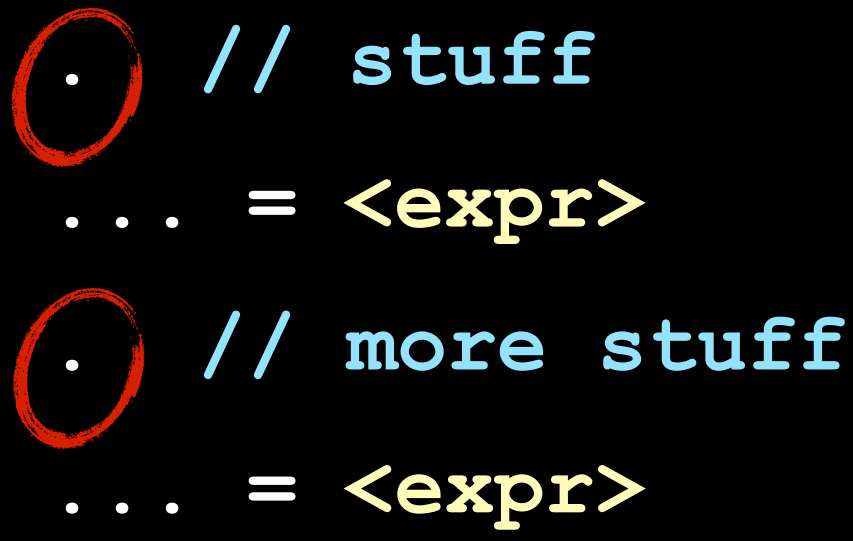
Rematerialization

```
[M]   = <expr>
  .   // stuff
 ...  =  [M]
  .   // more stuff
 ...  =  [M]
```

Rematerialization

```
vr1  = <expr>
.    // stuff
...  = vr1
.    // more stuff
...  = vr1
```

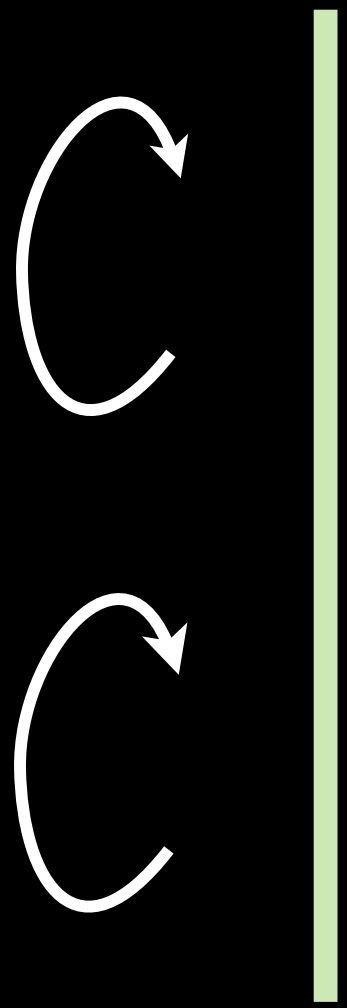
Rematerialization



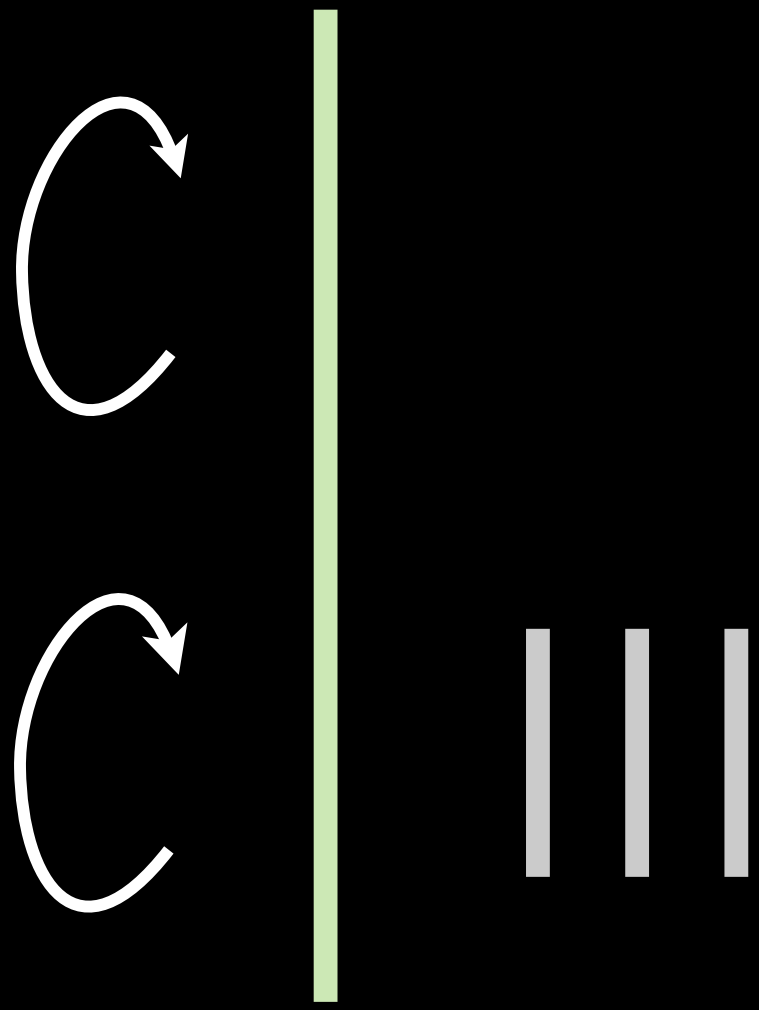
Splitting

Splitting

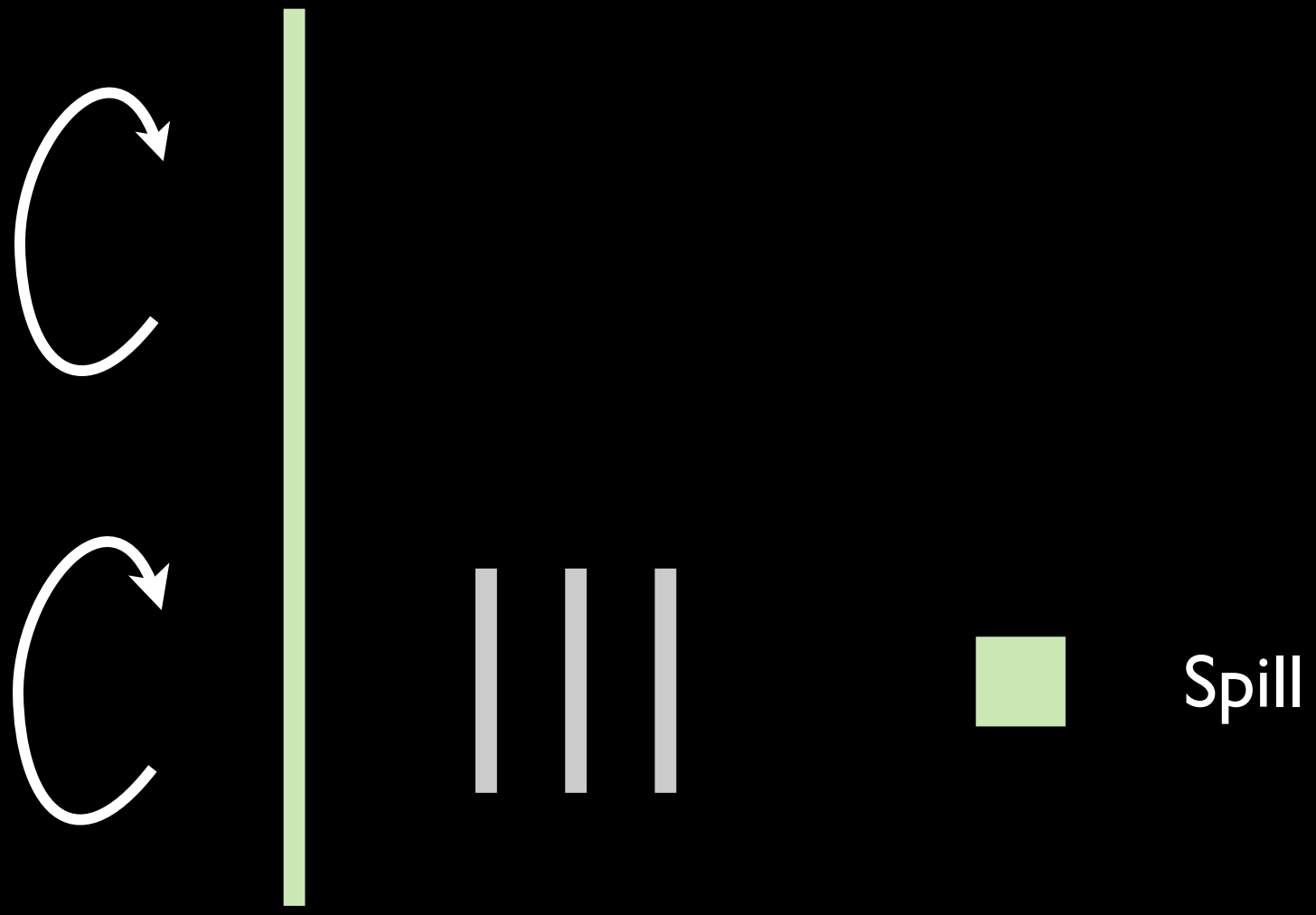
Splitting



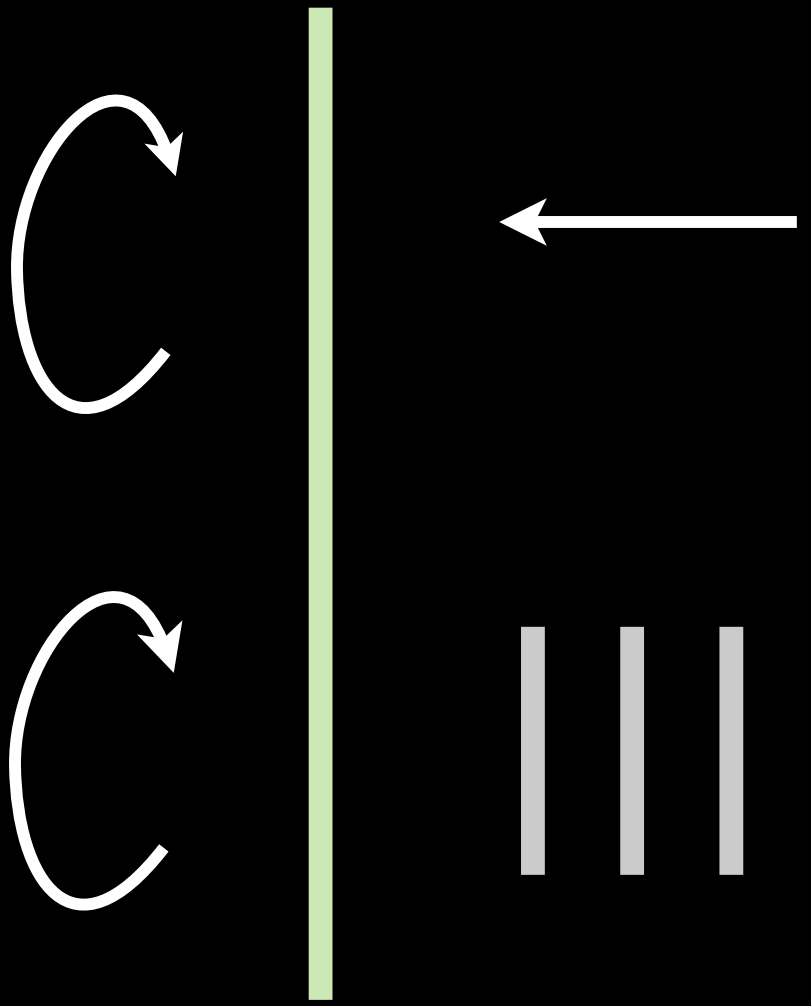
Splitting



Splitting



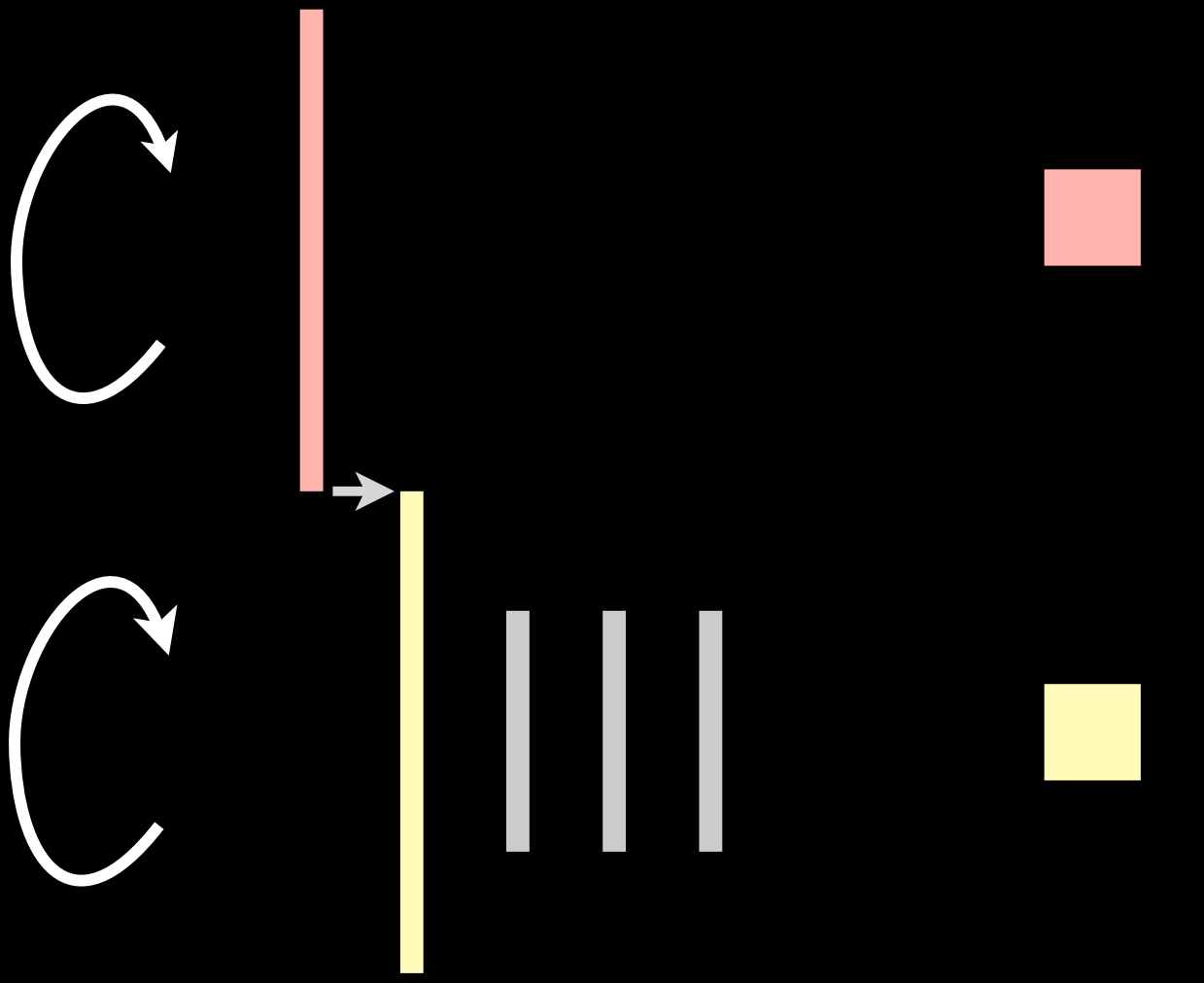
Splitting



Spills introduced
in to other loops

Spill

Splitting



2. New Allocators

New Allocators

New Allocators

- "Linear Scan" is not, in fact, linear

New Allocators

- "Linear Scan" is not, in fact, linear
- We want something faster

New Allocators

- "Linear Scan" is not, in fact, linear
- We want something faster
- Priority coloring?

New Allocators

- "Linear Scan" is not, in fact, linear
- We want something faster
- Priority coloring?
- *Linear* Scan?

New Allocators

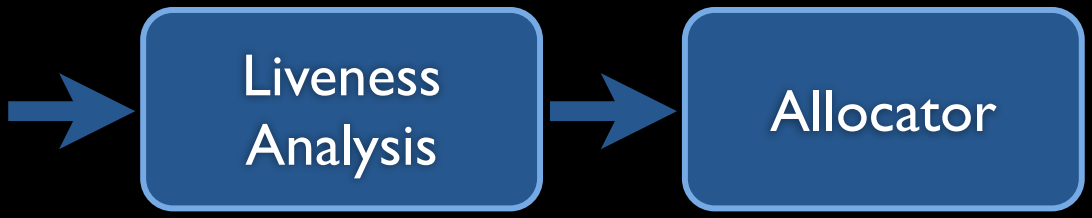
- "Linear Scan" is not, in fact, linear
- We want something faster
- Priority coloring?
- *Linear* Scan?
- Need to tidy the infrastructure

3. Cleaner Infrastructure

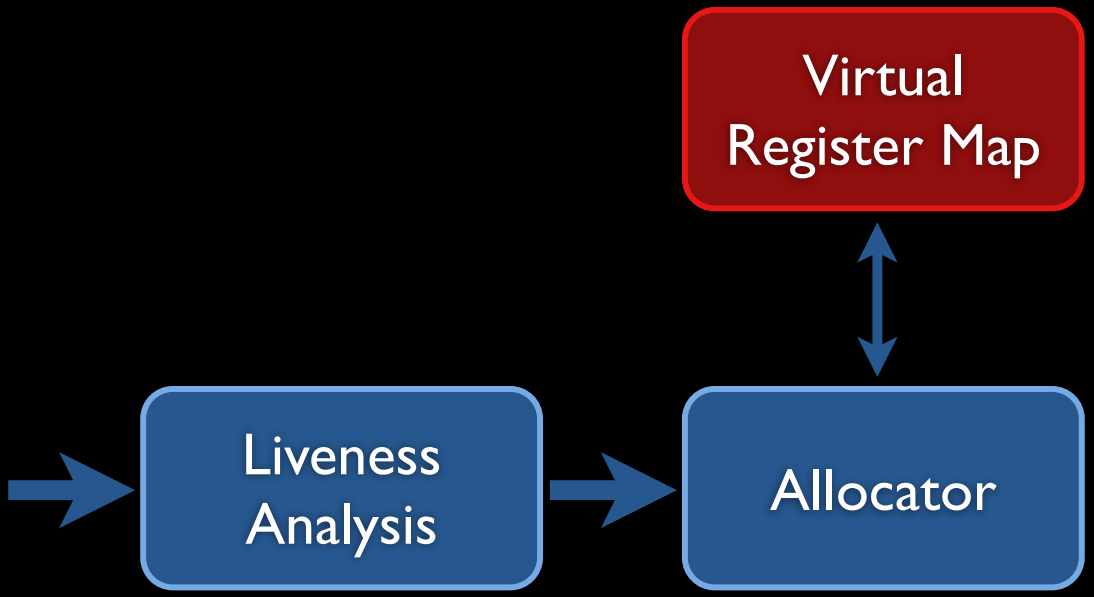
Currently...

Currently...

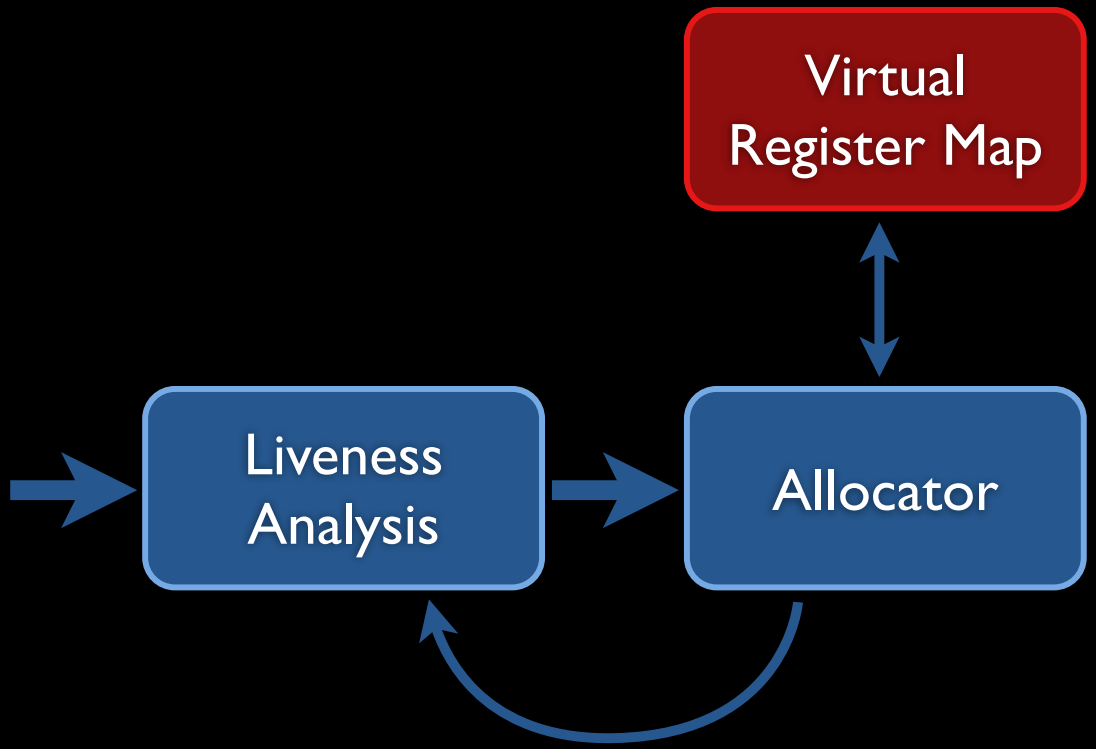
Currently...



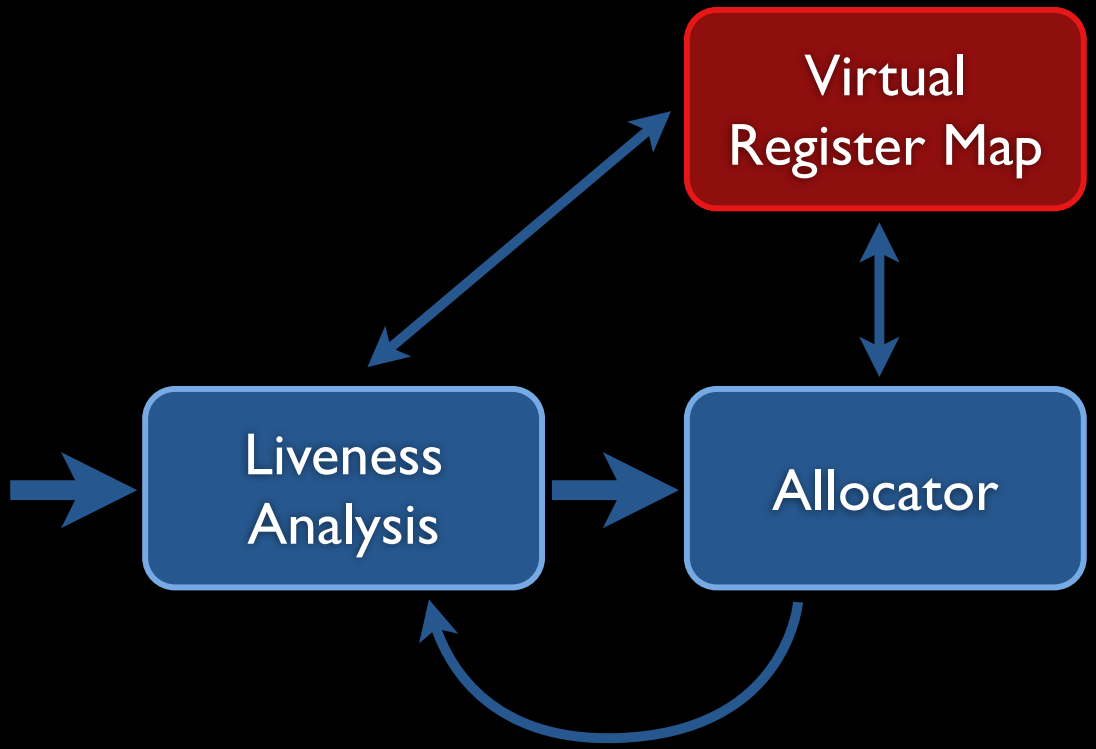
Currently...



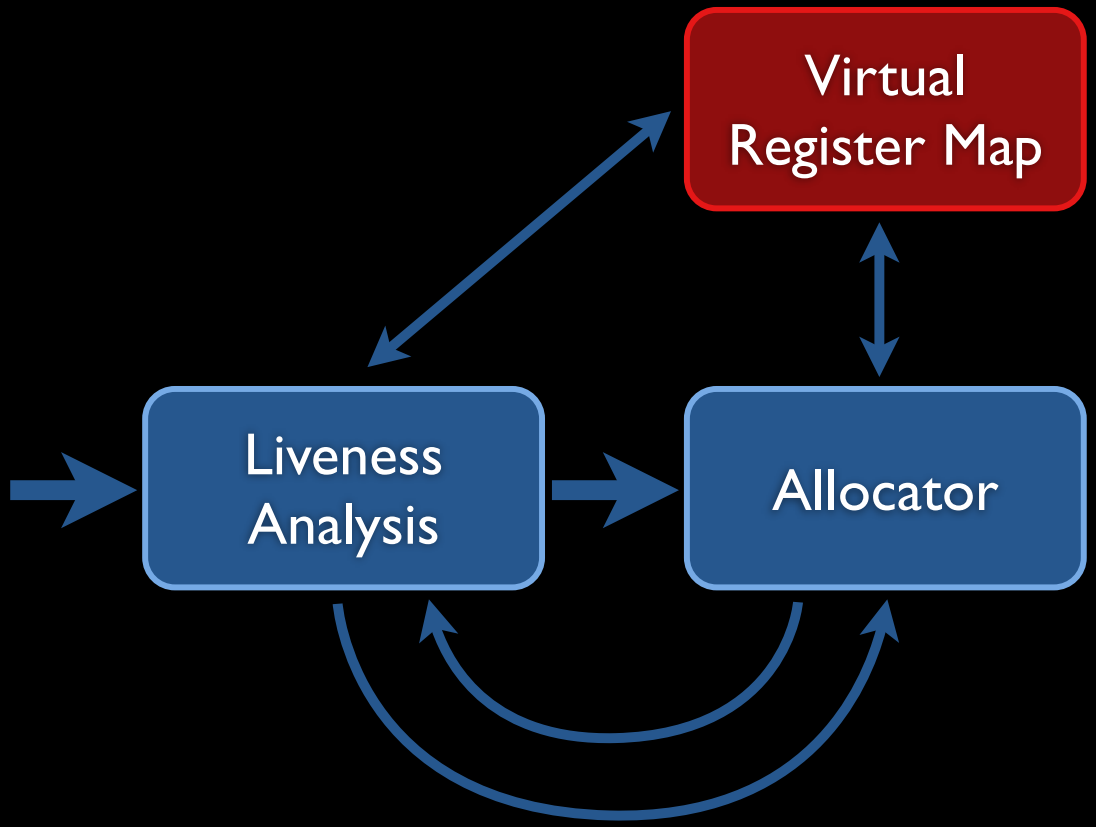
Currently...



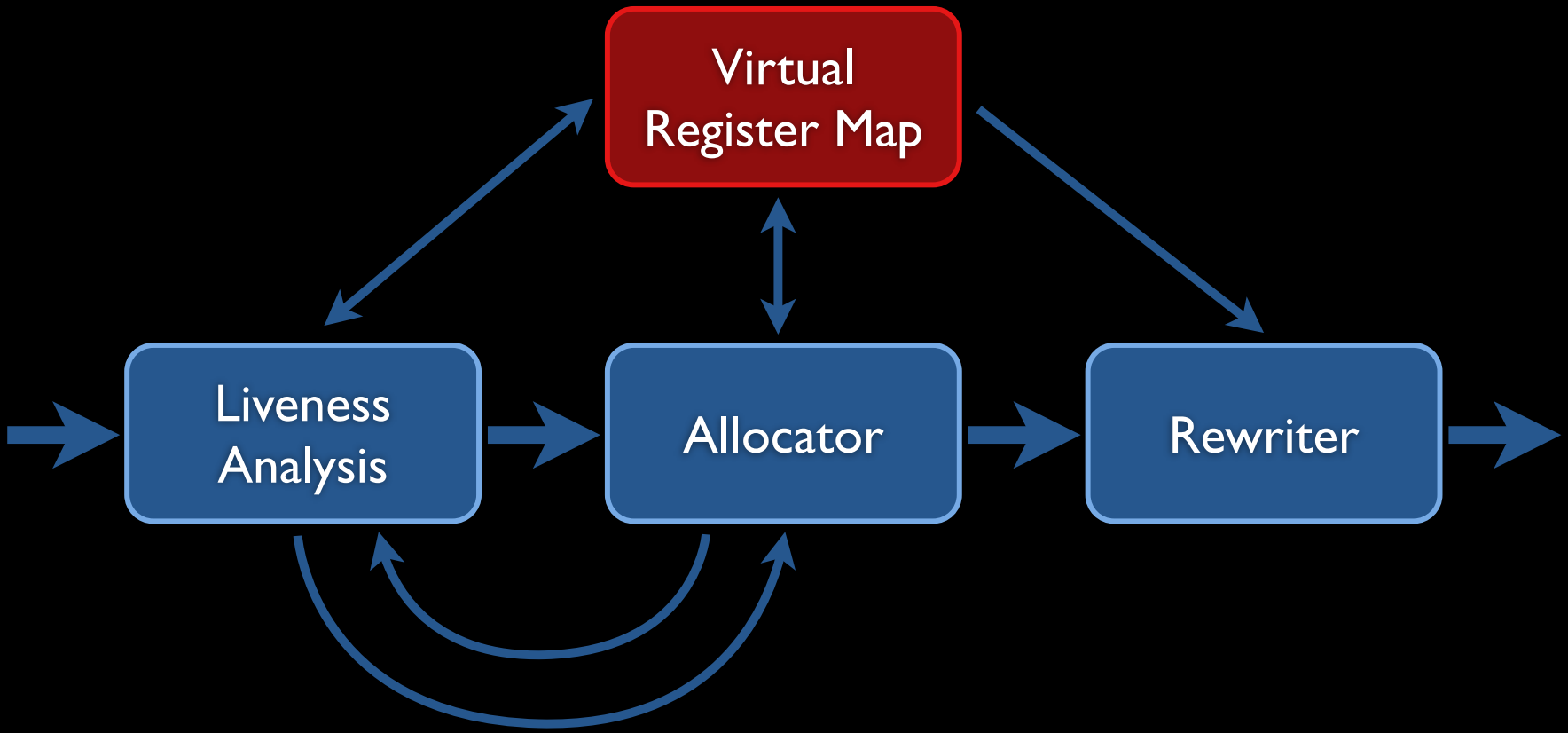
Currently...



Currently...



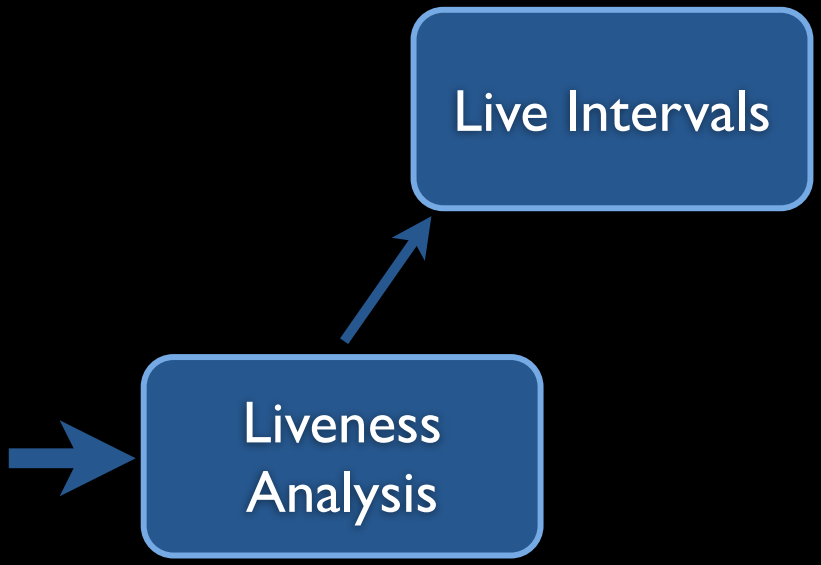
Currently...


Virtual
Register Map
Liveness
Analysis
Allocator
Rewriter

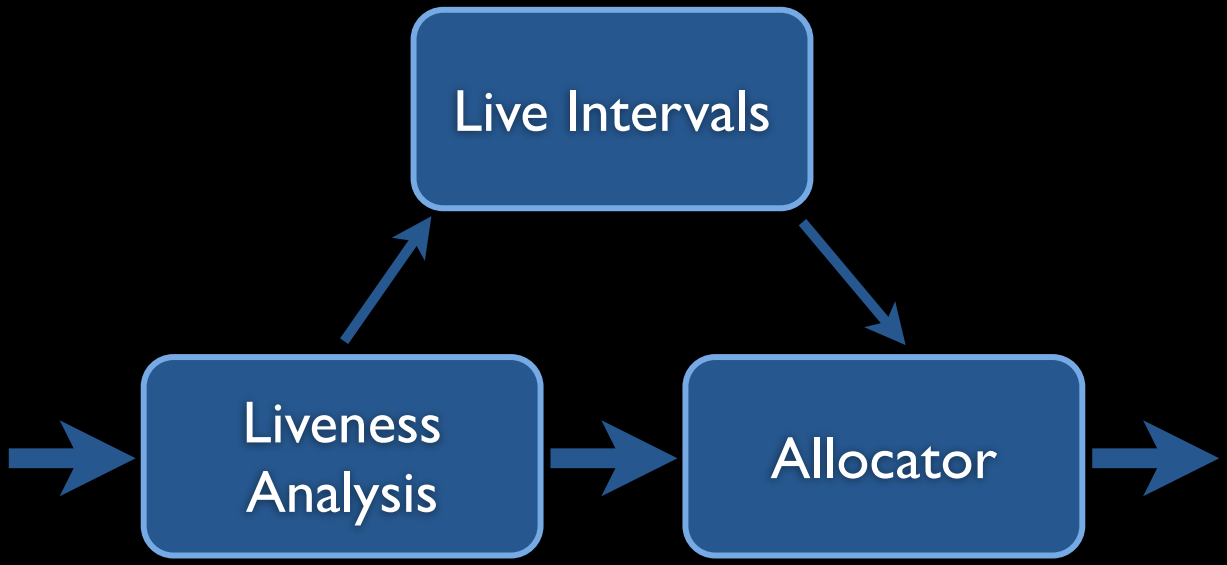
Rewrite In Place

Rewrite In Place

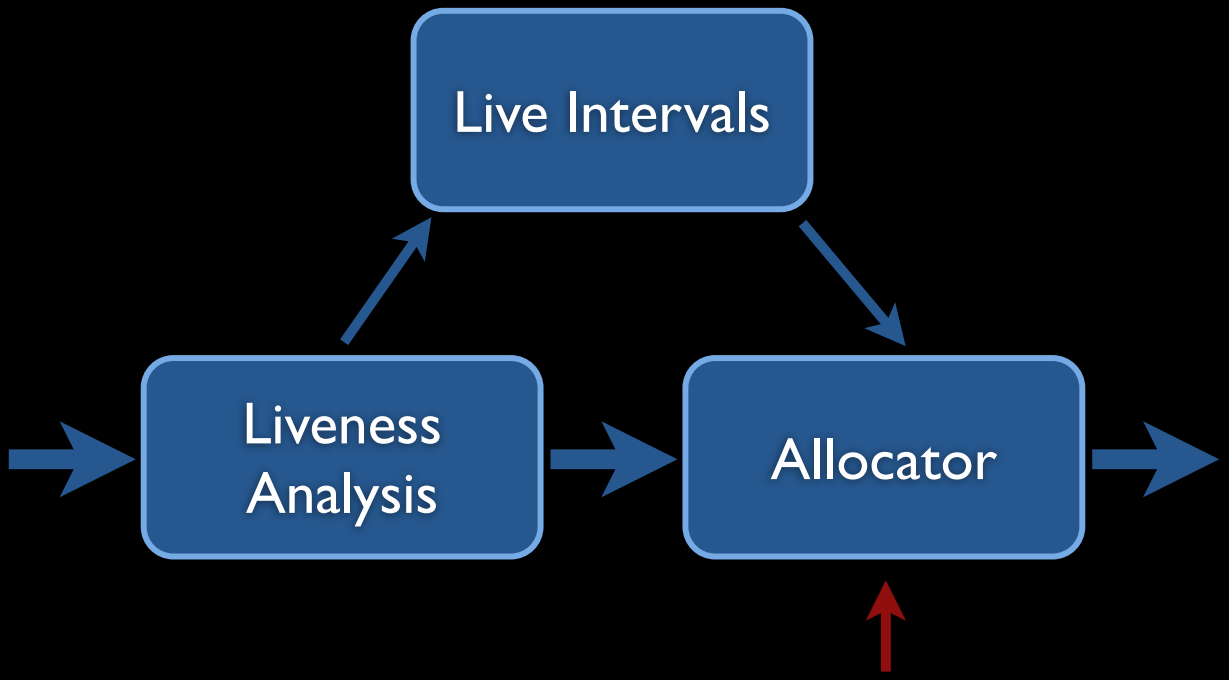
Rewrite In Place



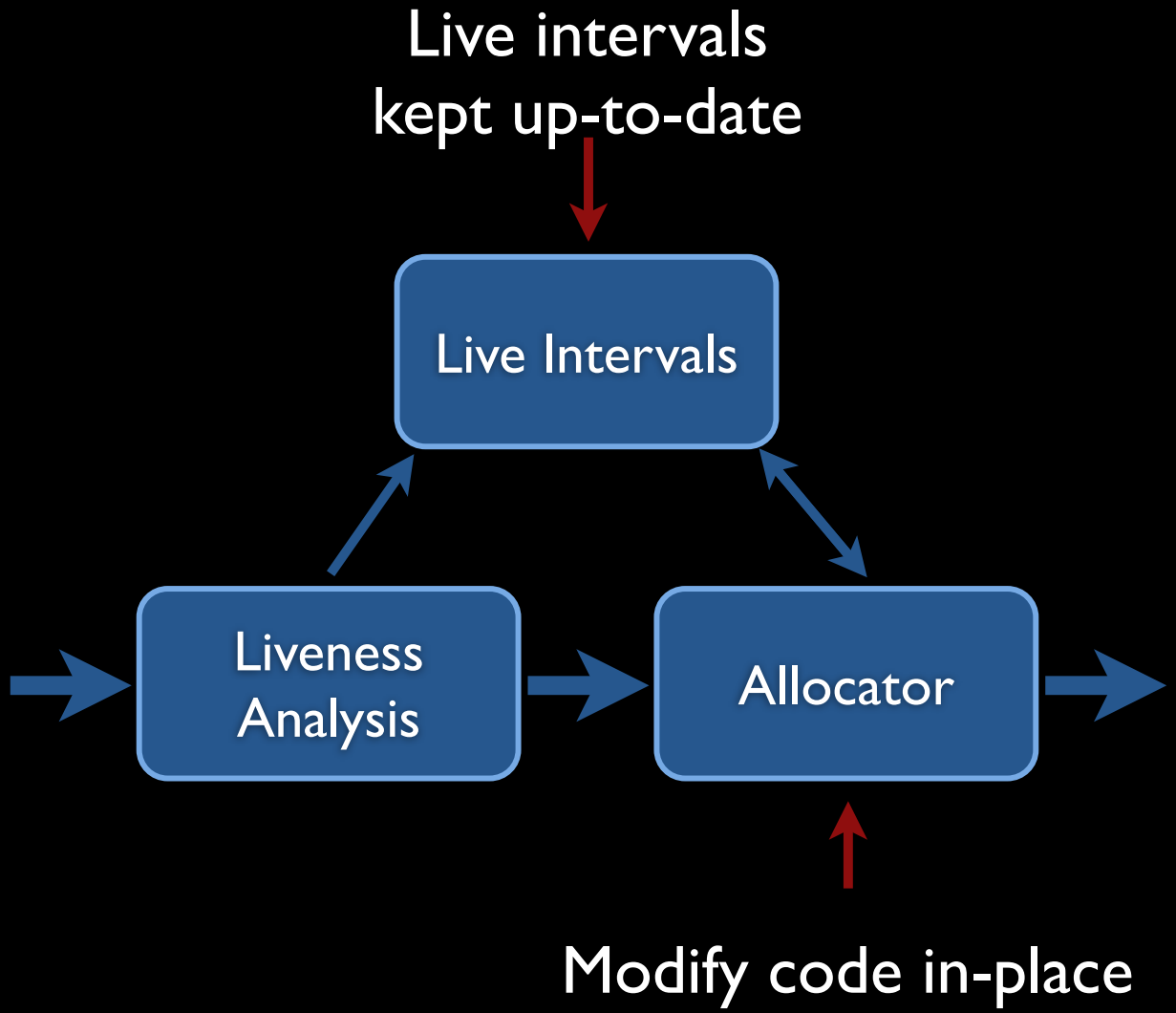
Rewrite In Place



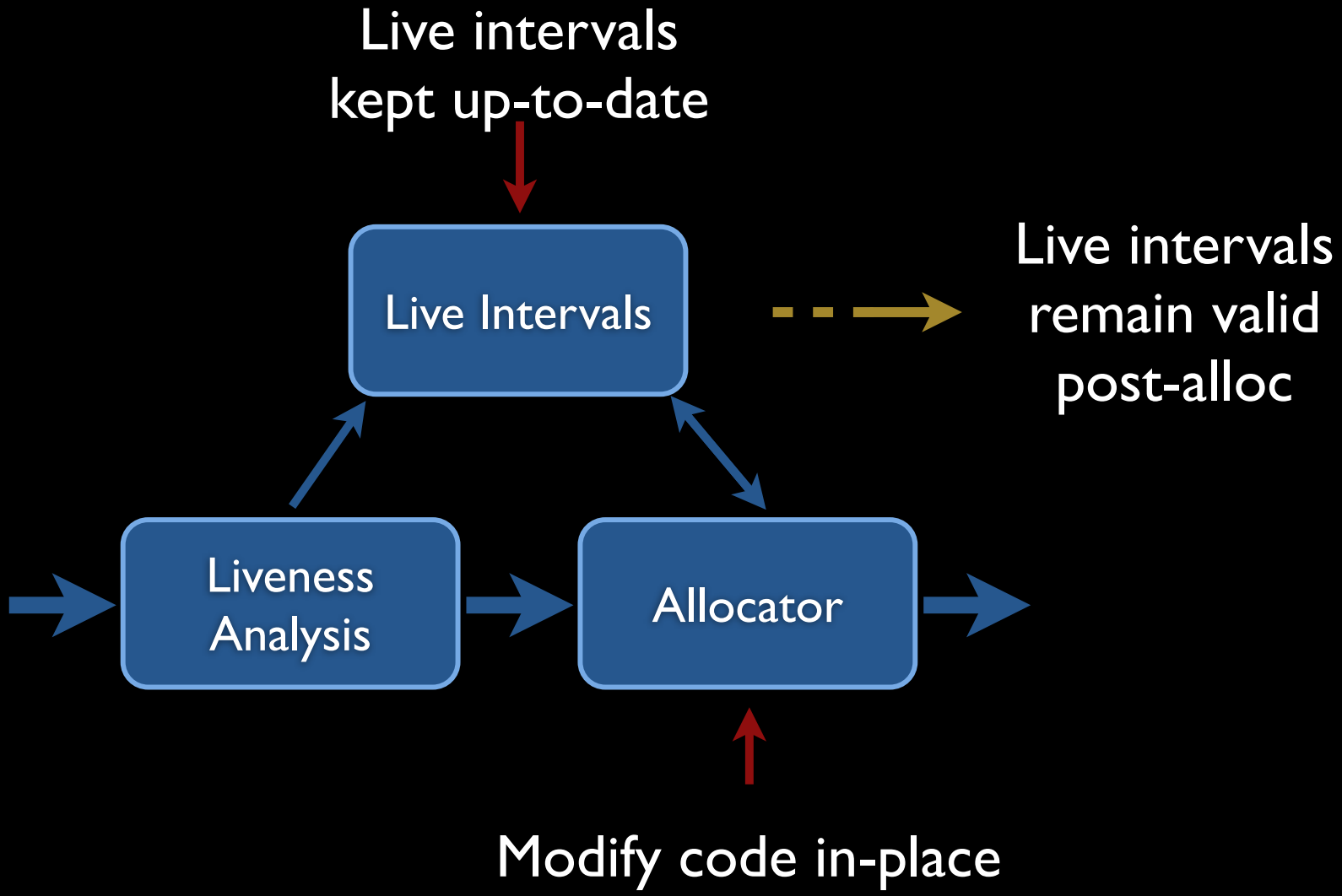
Rewrite In Place



Rewrite In Place



Rewrite In Place



Improvements

- New and better optimizations
- New allocators
- Cleaner infrastructure

Upcoming Changes

Upcoming Changes

- Live index renumbering
- Improved splitting
- Better def/kill tracking for values

Live Indexes

Live Indexes

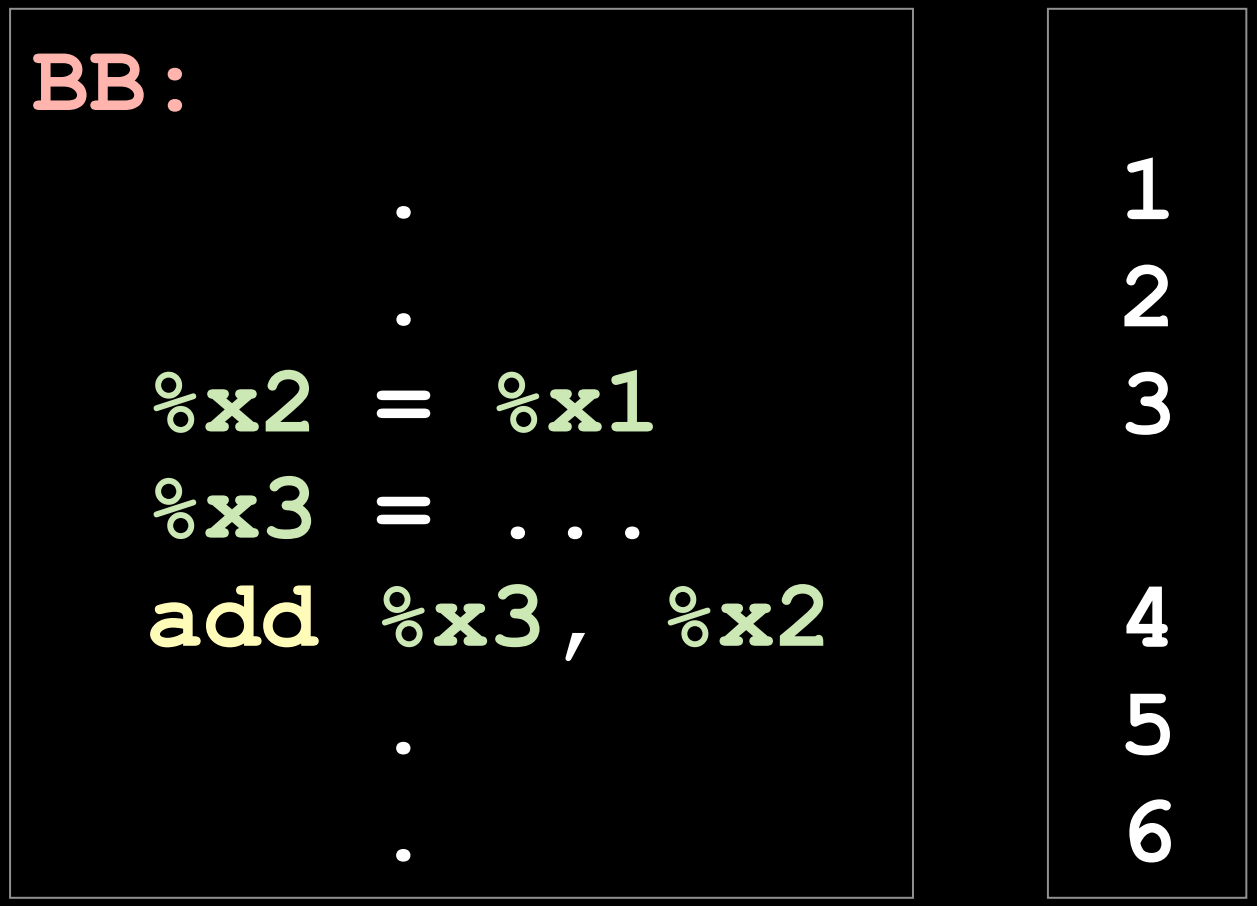
```
BB:
        .
        .
   %x2 = %x1
   add %x3, %x2
        .
        .
```

Live Indexes

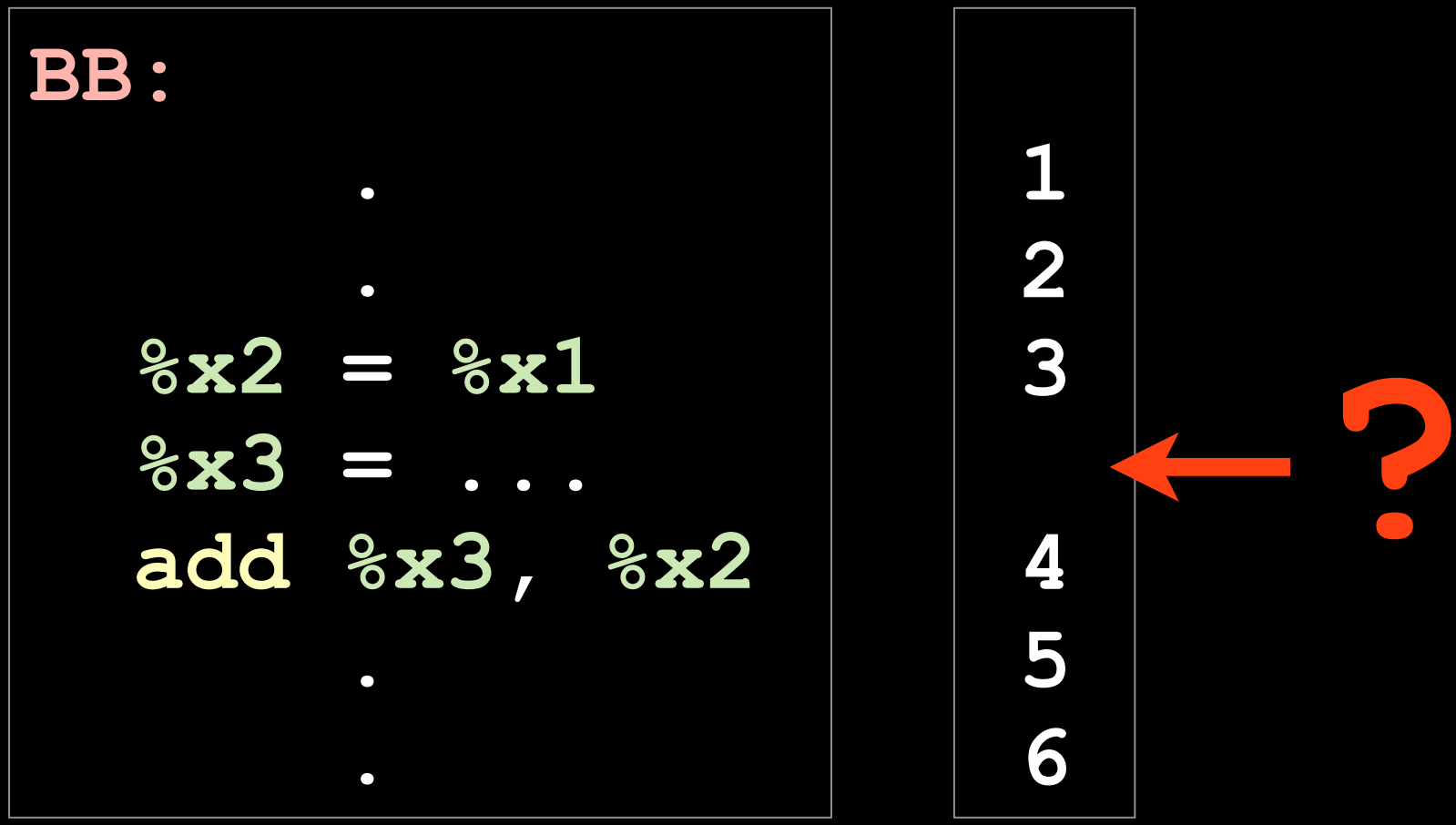
```
BB:
          .
          .
   %x2 = %x1
   add %x3, %x2
          .
          .
```

```
1
2
3
4
5
6
```

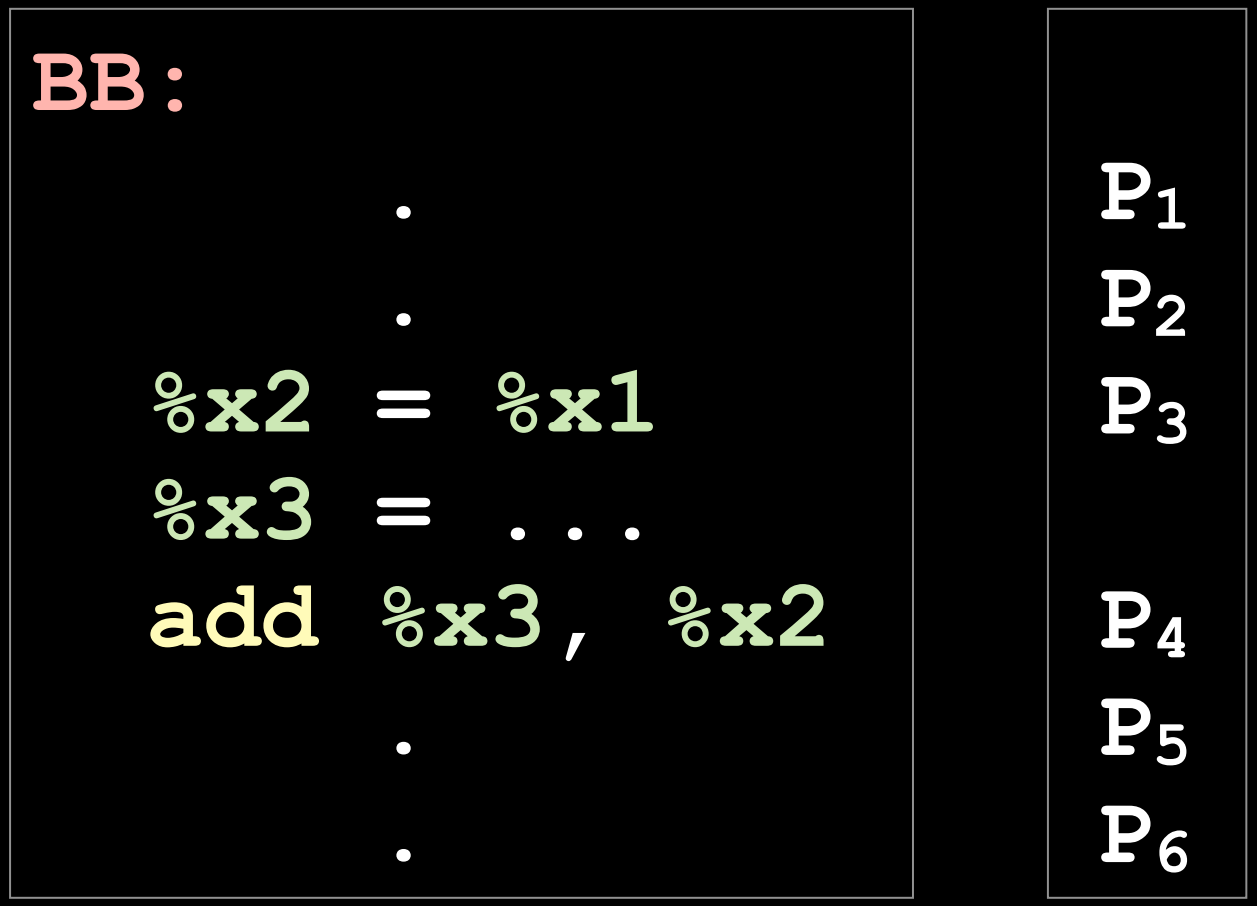
Live Indexes



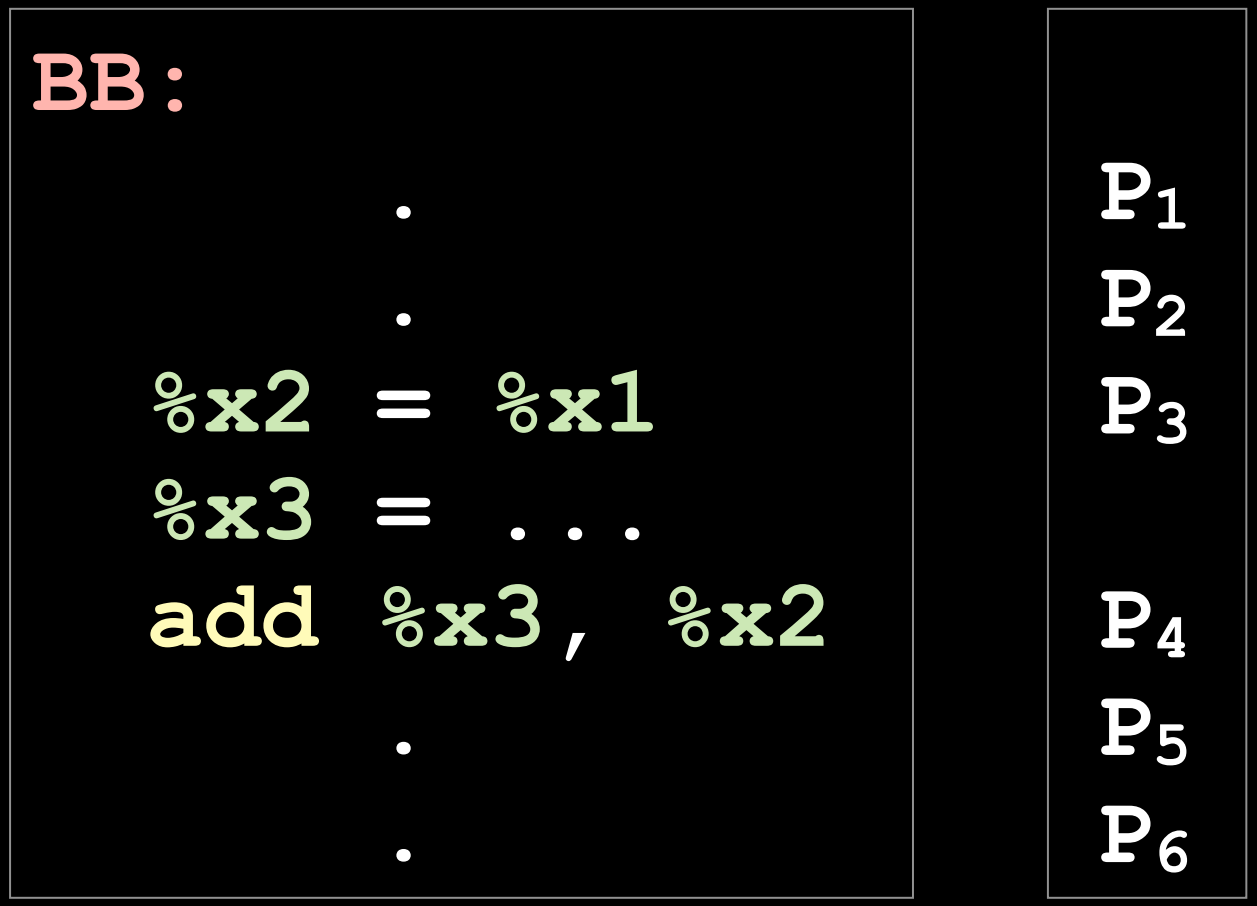
Live Indexes



Live Indexes

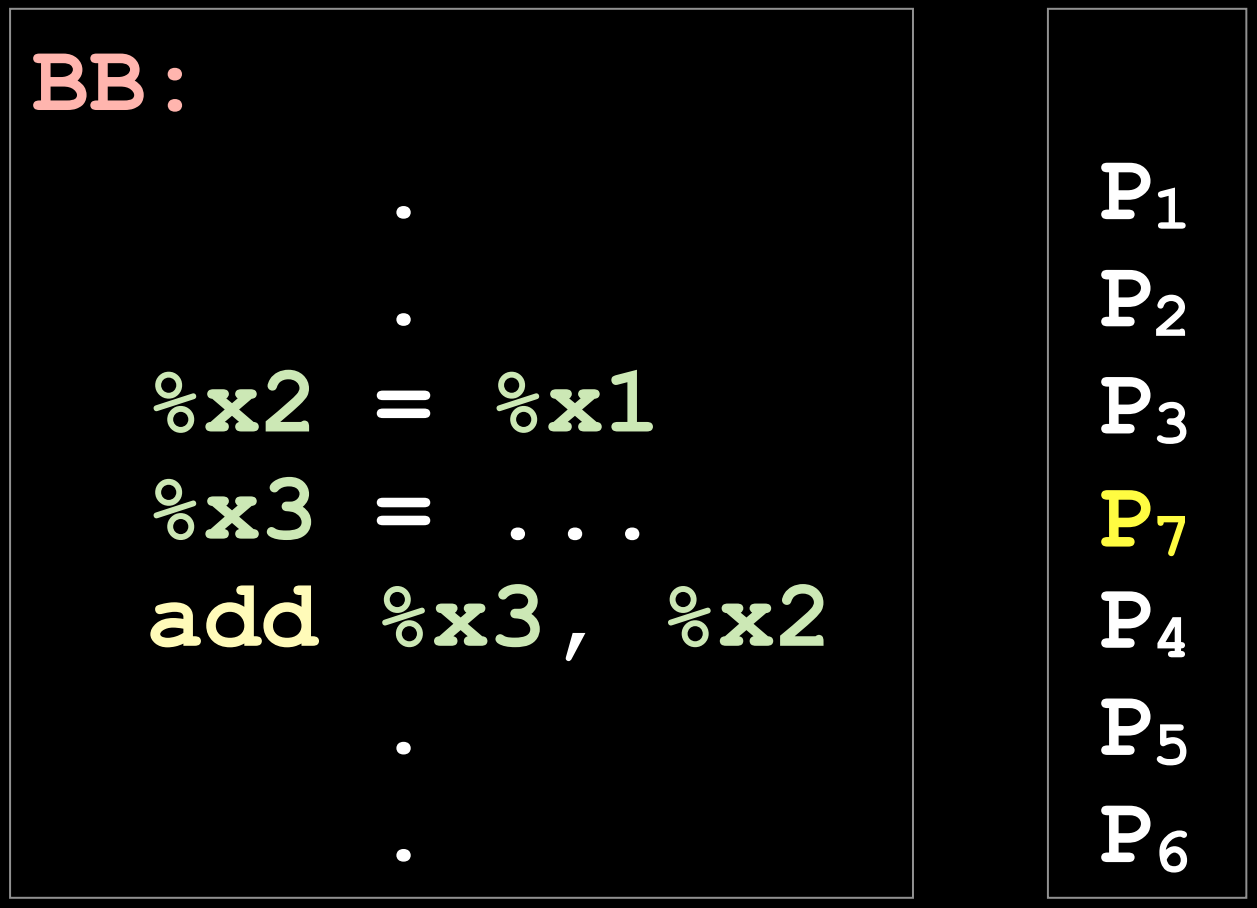


Live Indexes



$P_1 < P_2 < P_3 < P_4 < P_5 < P_6$

Live Indexes



$P_1 < P_2 < P_3 < P_4 < P_5 < P_6$

Live Indexes

```
BB:
        .
        .
   %x2 = %x1
   %x3 = ...
   add %x3, %x2
        .
        .
        .
```

P_1
P_2
P_3
P_7
P_4
P_5
P_6

$P_1 < P_2 < P_3 < P_7 < P_4 < P_5 < P_6$

Live Indexes

- *unsigned* → *LiveIndex*

- Index Renumbering

Improved Splitting

Improved Splitting

- Break multi-value intervals into component values.

Improved Splitting

- Break multi-value intervals into component values.
- Each value gets a 2nd chance at allocation.

Improved Splitting

- Break multi-value intervals into component values.
- Each value gets a 2nd chance at allocation.
- ... but not a 3rd.

Improved Splitting

- Break multi-value intervals into component values.
- Each value gets a 2nd chance at allocation.
- ... but not a 3rd.
- 13% reduction in static memory references on test case (a pathological SSE kernel).

Better Def/Kill Tracking

Better Def/Kill Tracking

For Values

- Defined by a PHI - Track the def block.

Better Def/Kill Tracking

For Values

- Defined by a PHI - Track the def block.
- Killed by a PHI - Track the appropriate predecessor.

Future Work

- Better value def/kill tracking 
- LiveIndex renumbering 
- Improved splitting 
- Rewrite-in-place 
- New allocators 

PBQP

PBQP

Partitioned Boolean Quadratic Problems

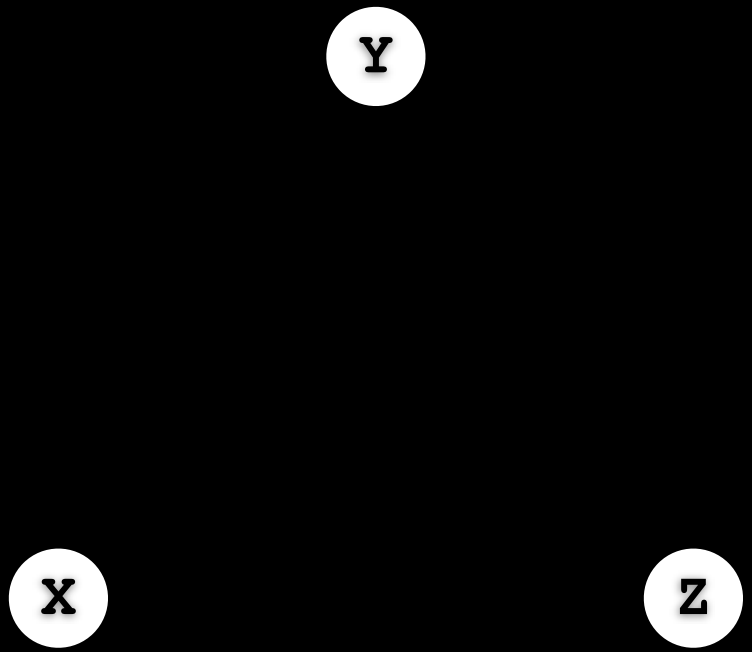
- Discrete optimization problems
- NP-complete
- Subclass solvable in linear time

Irregular Architectures

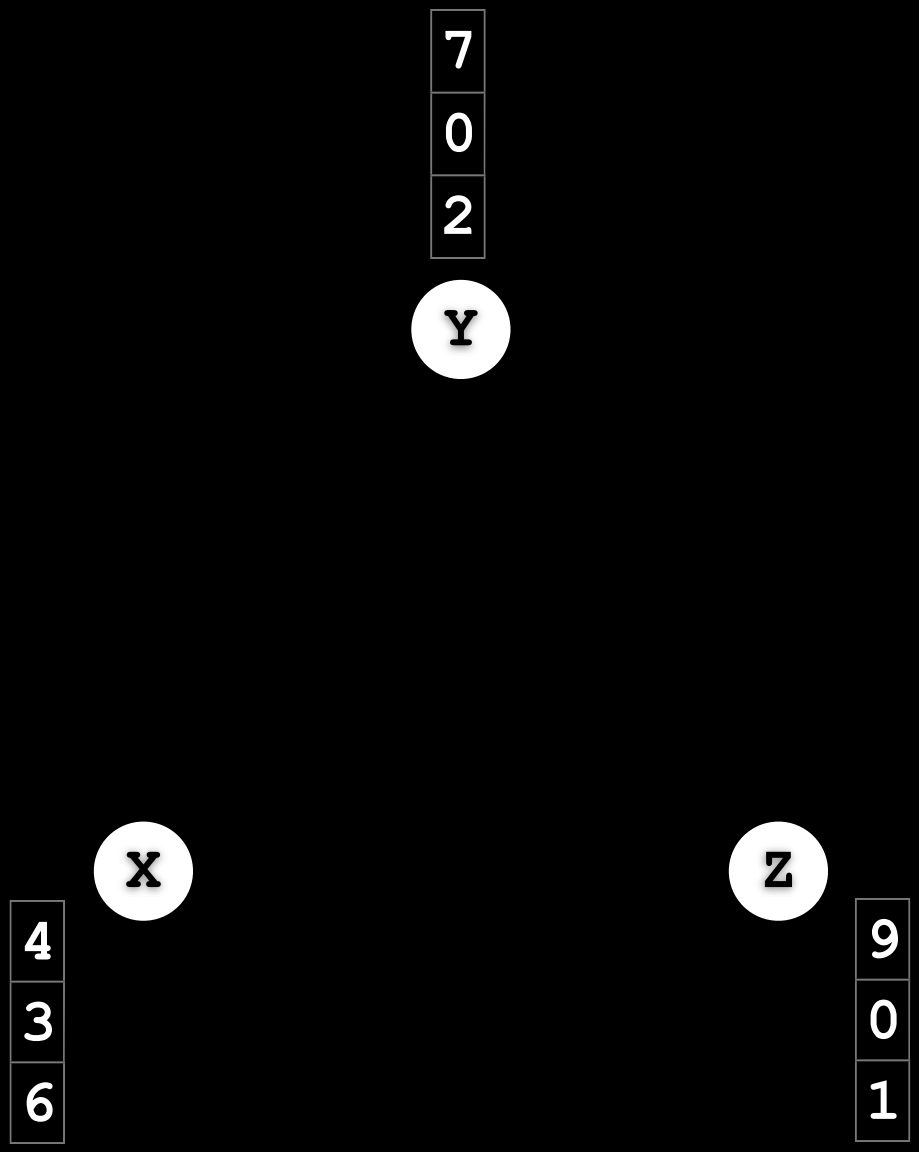
- Multiple register classes.
- Register aliasing.
- Register pairing.
- ...

PBQP Example

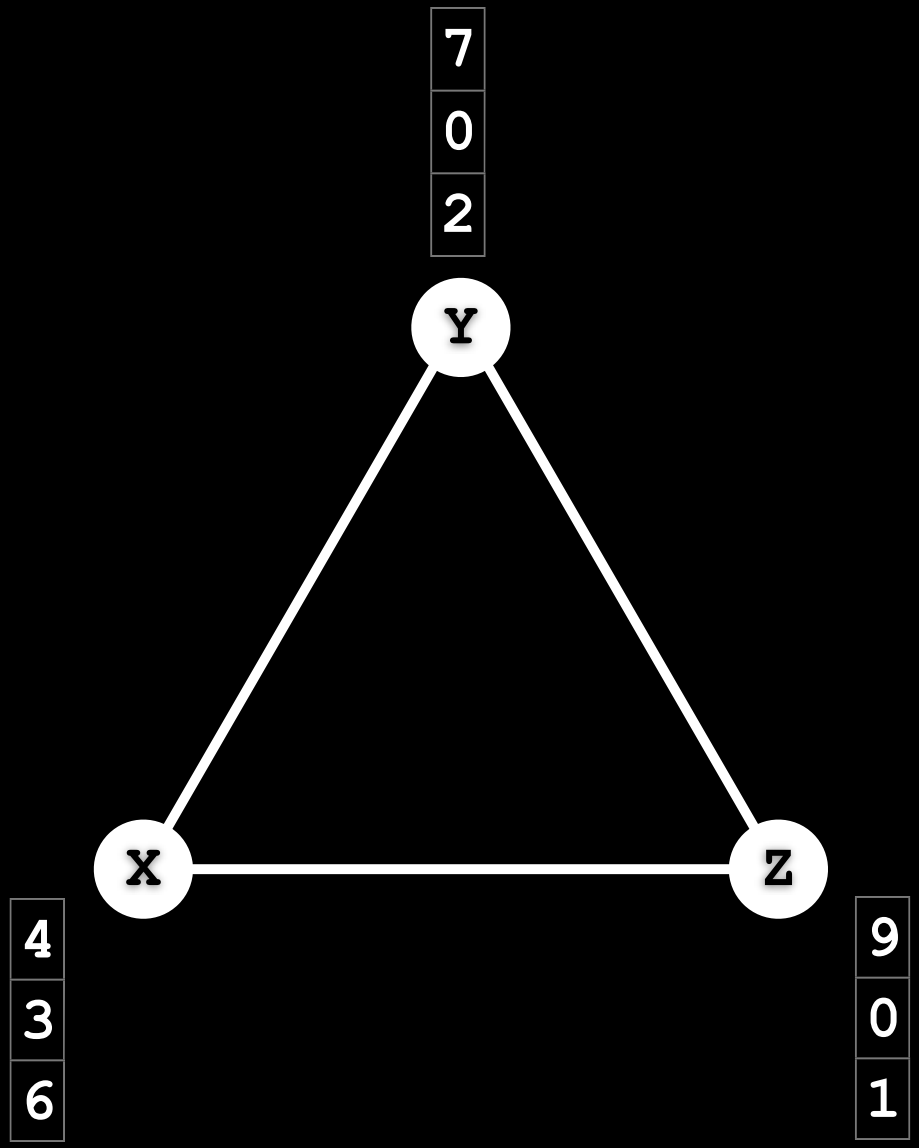
PBQP Example



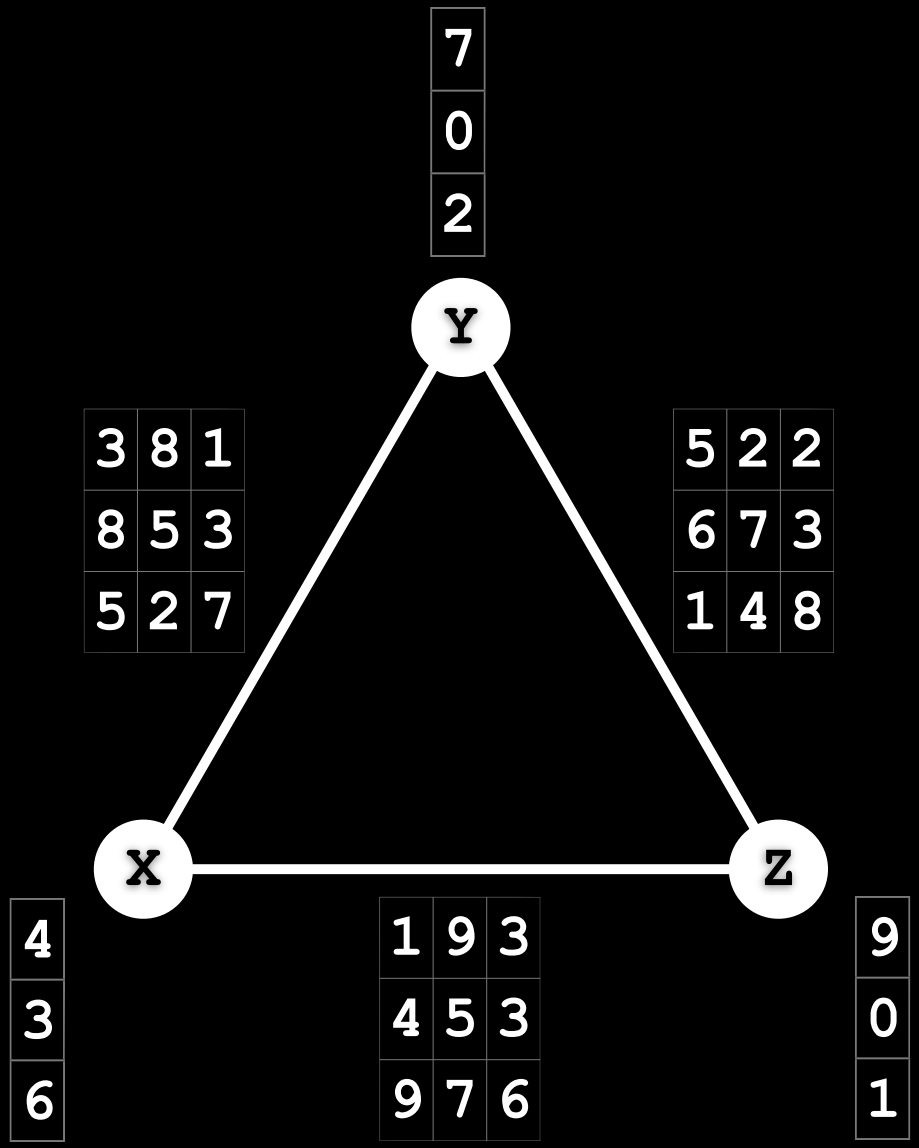
PBQP Example



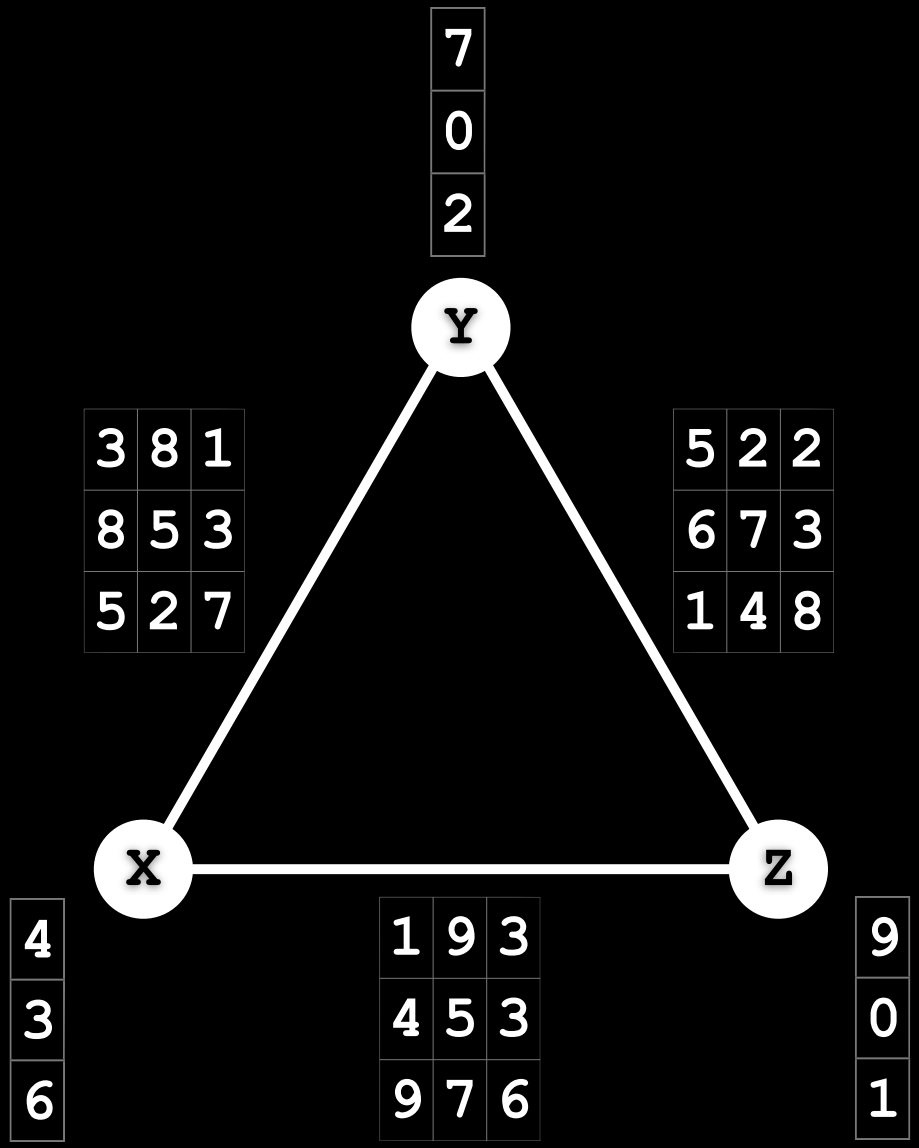
PBQP Example



PBQP Example

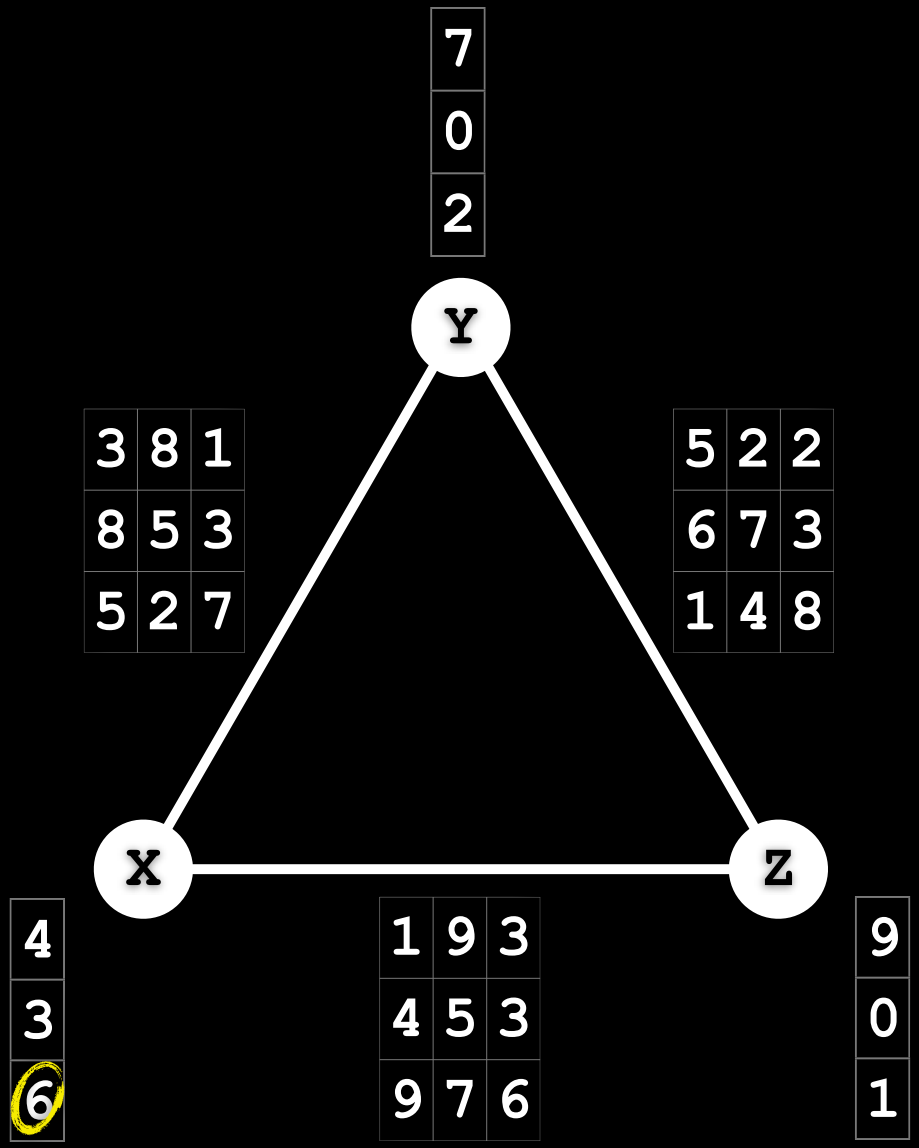


PBQP Example



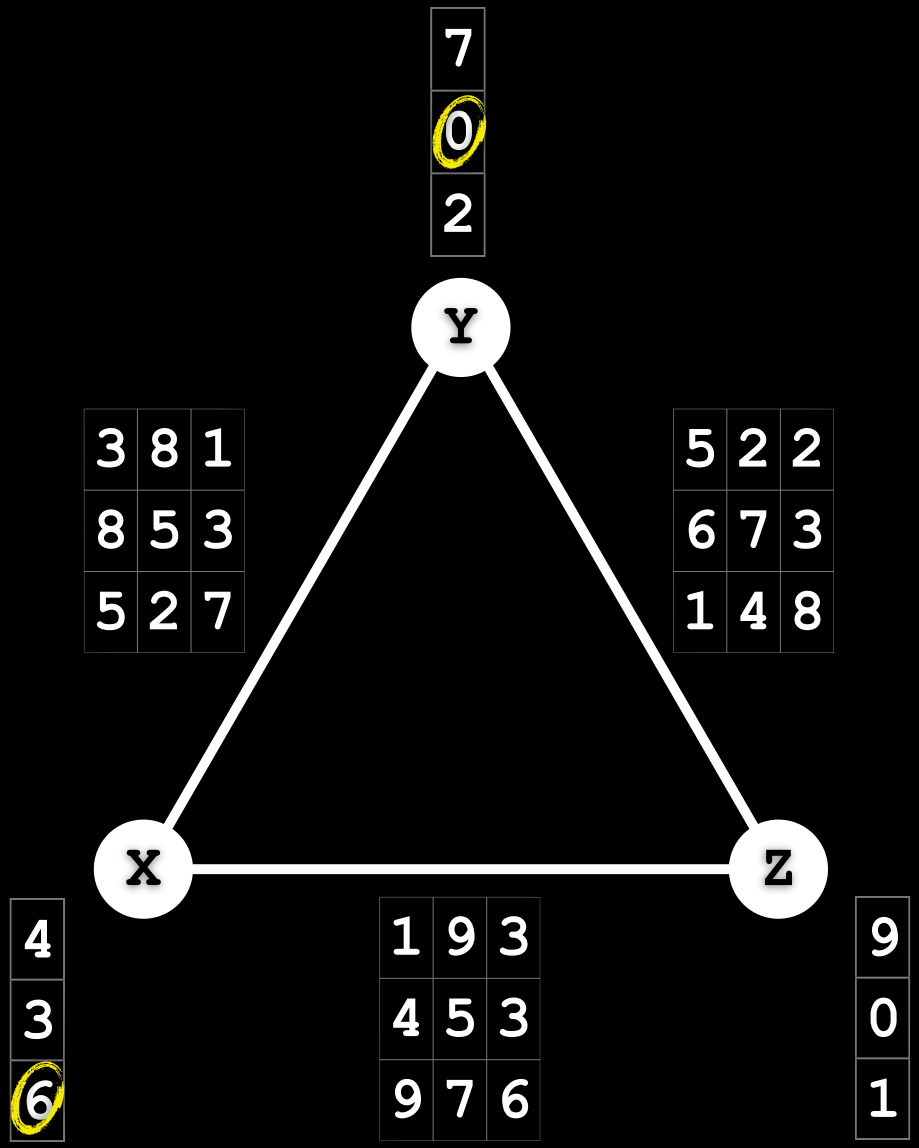
Solution [3,2,1]:

PBQP Example



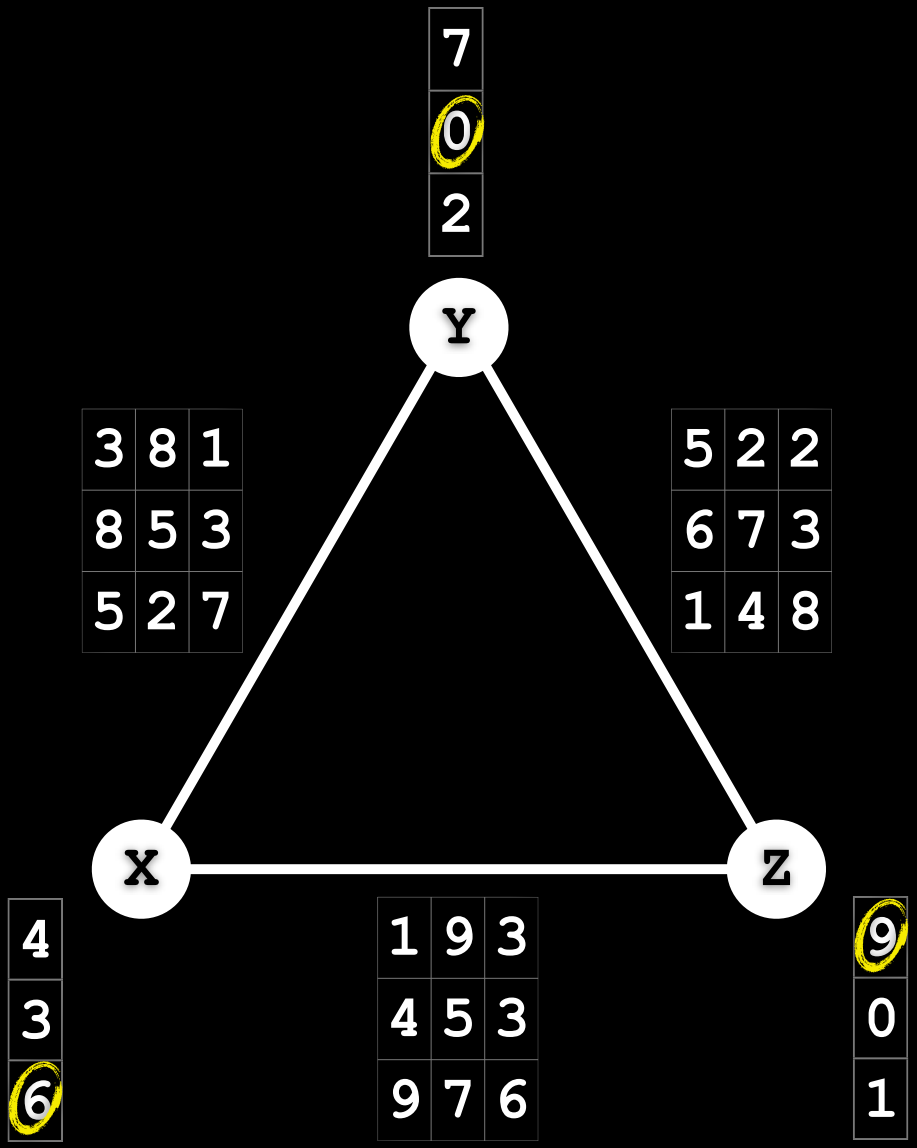
Solution [3,2,1]:

PBQP Example



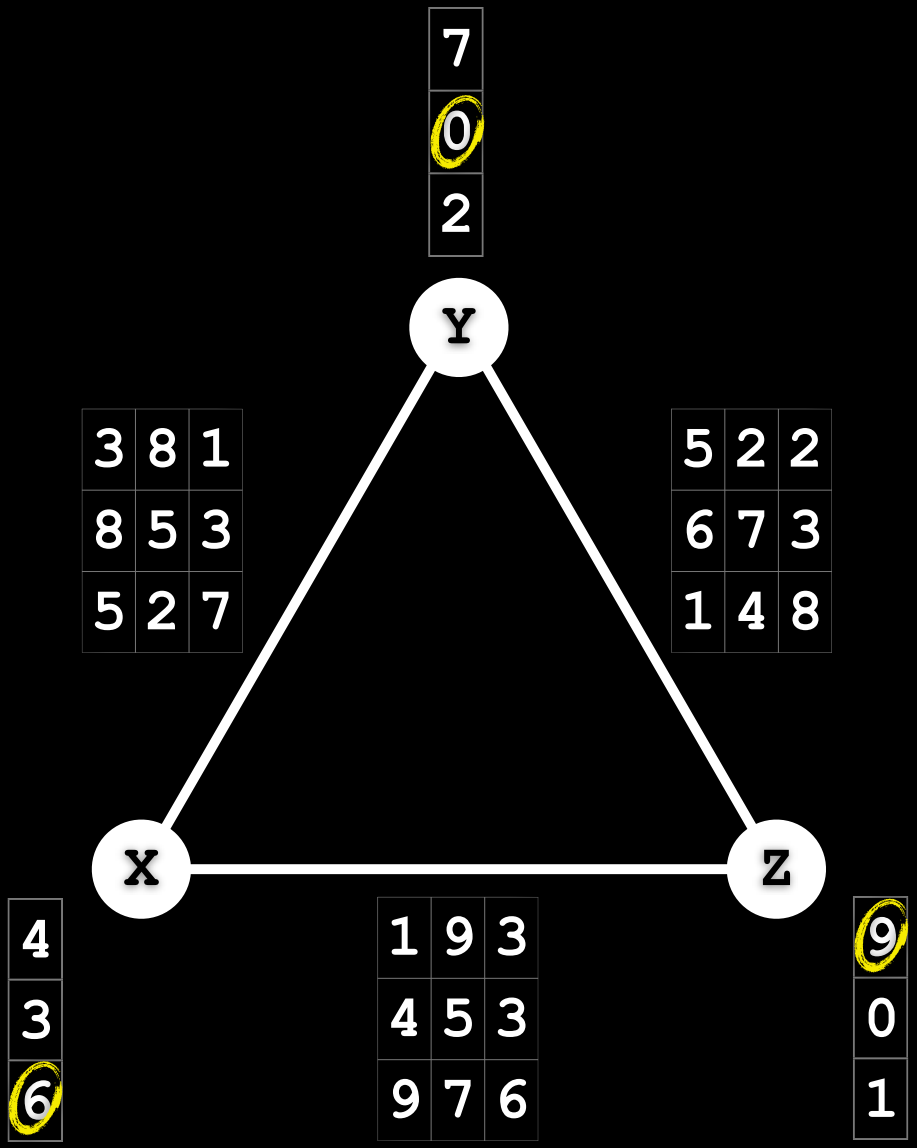
Solution [3,2,1]:

PBQP Example



Solution [3,2,1]:

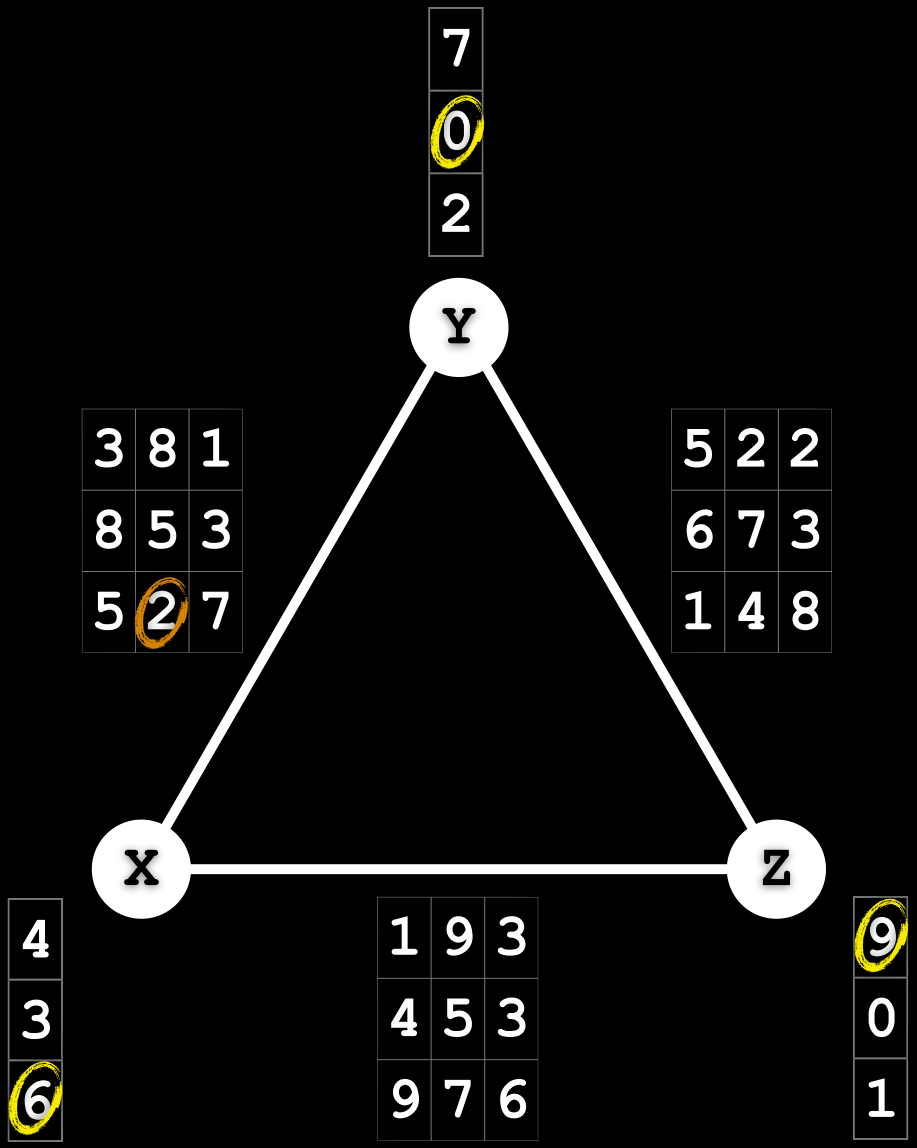
PBQP Example



Solution [3,2,1]:

Node Costs: 6+0+9 = 15

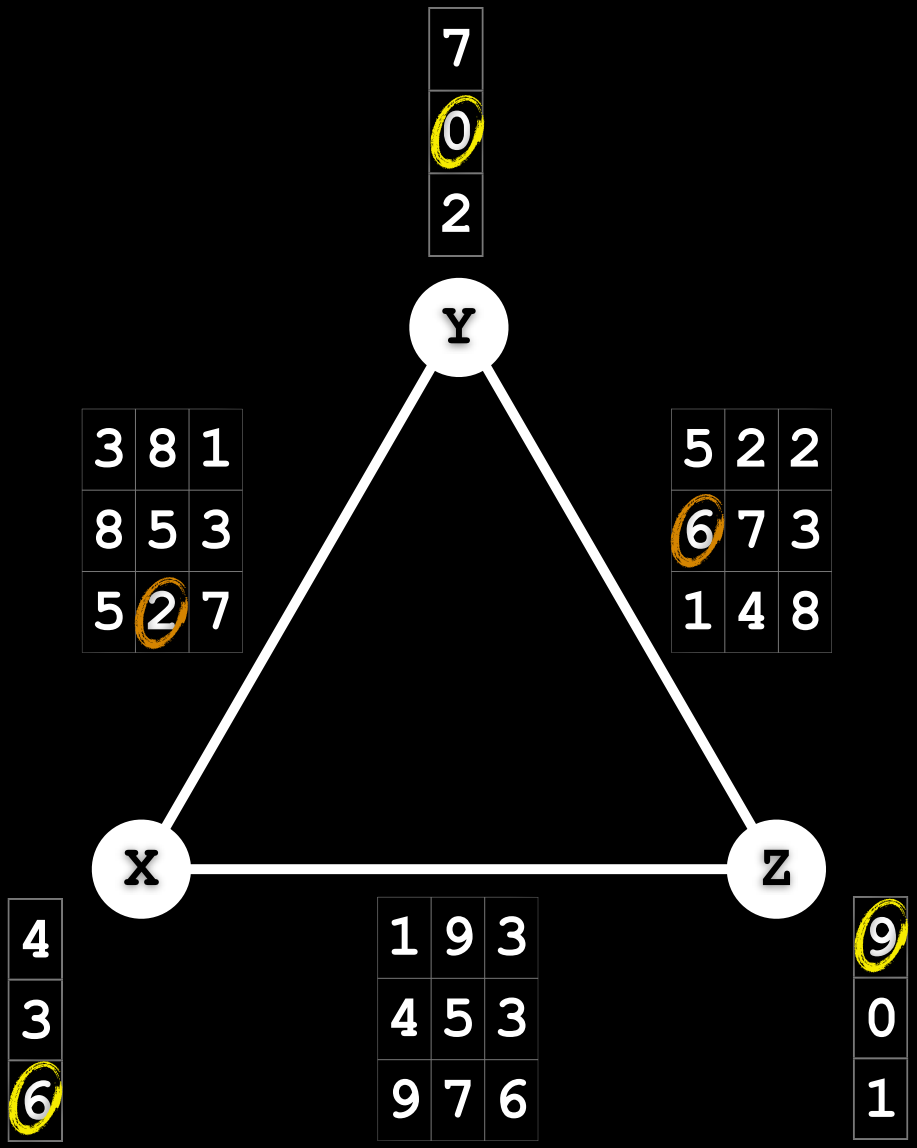
PBQP Example



Solution [3,2,1]:

Node Costs: 6+0+9 = 15

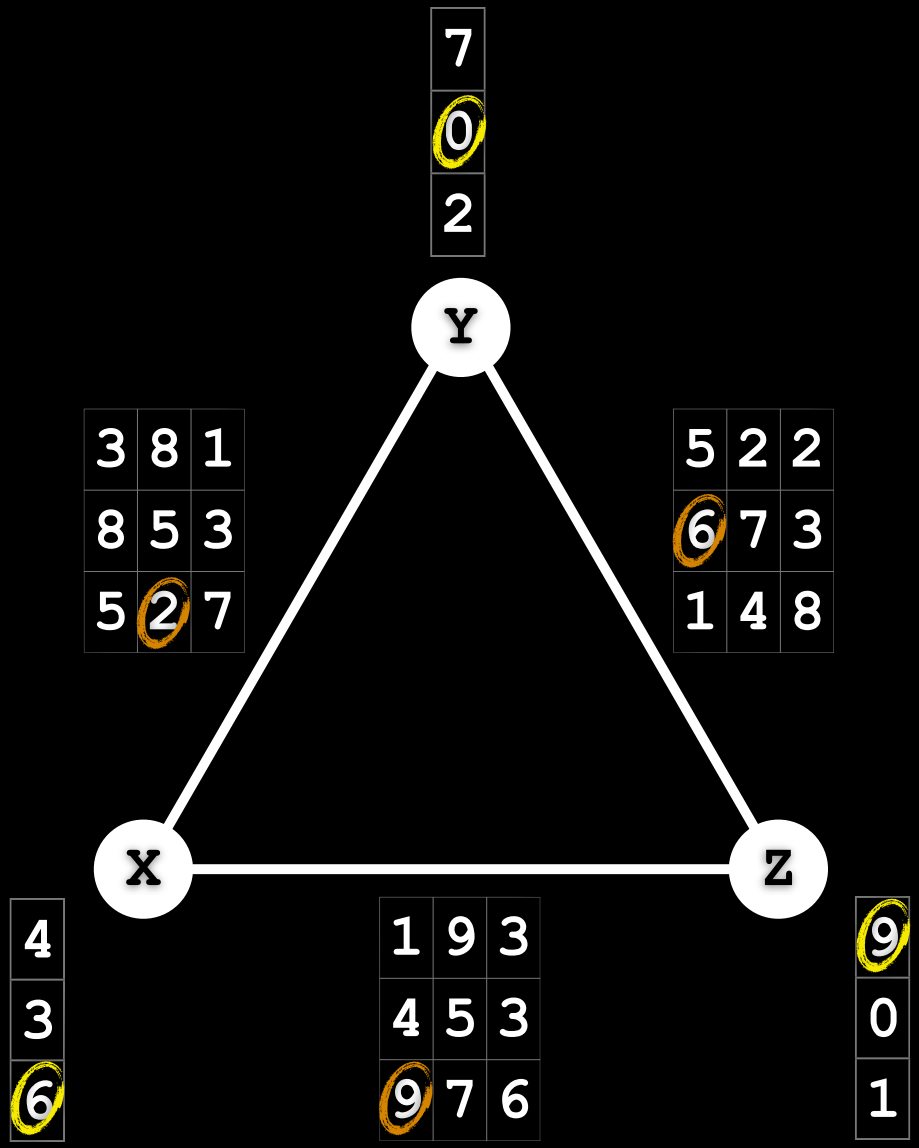
PBQP Example



Solution [3,2,1]:

Node Costs: 6+0+9 = 15

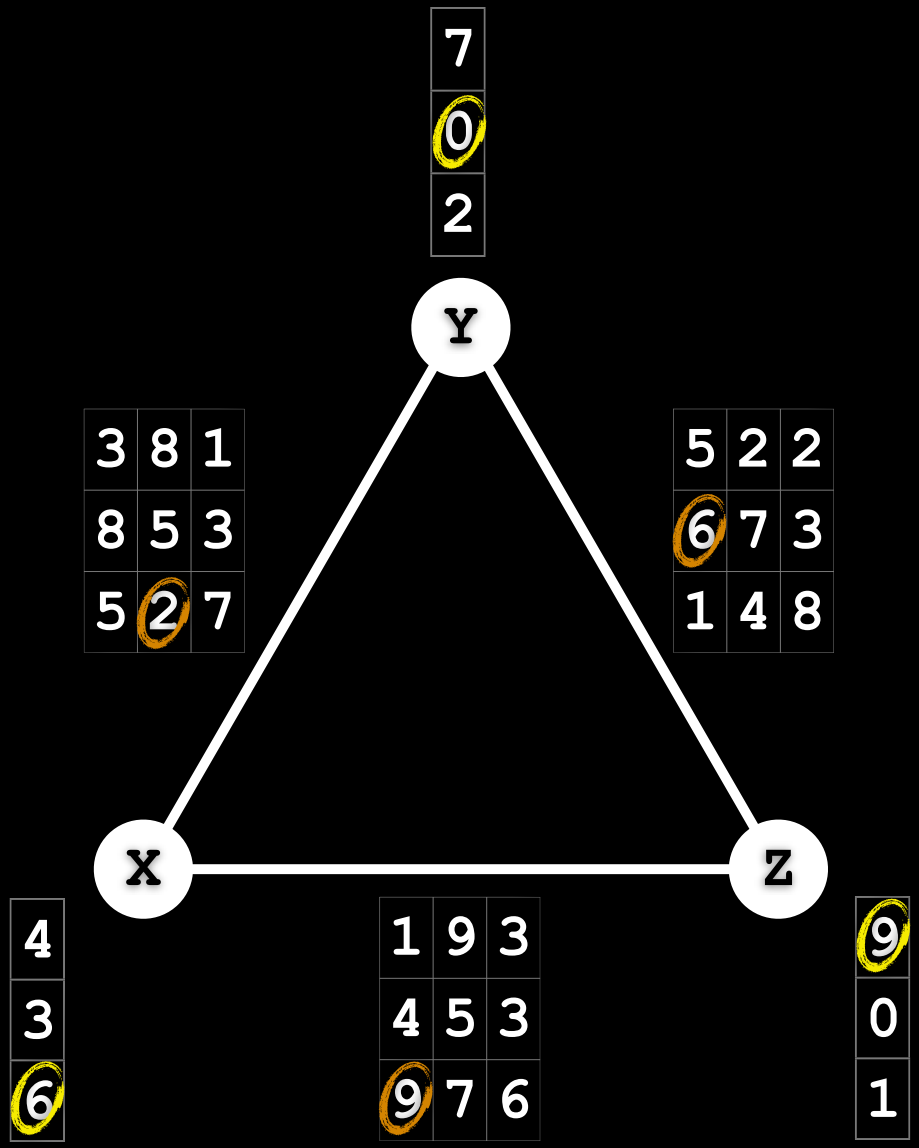
PBQP Example



Solution [3,2,1]:

Node Costs: 6+0+9 = 15

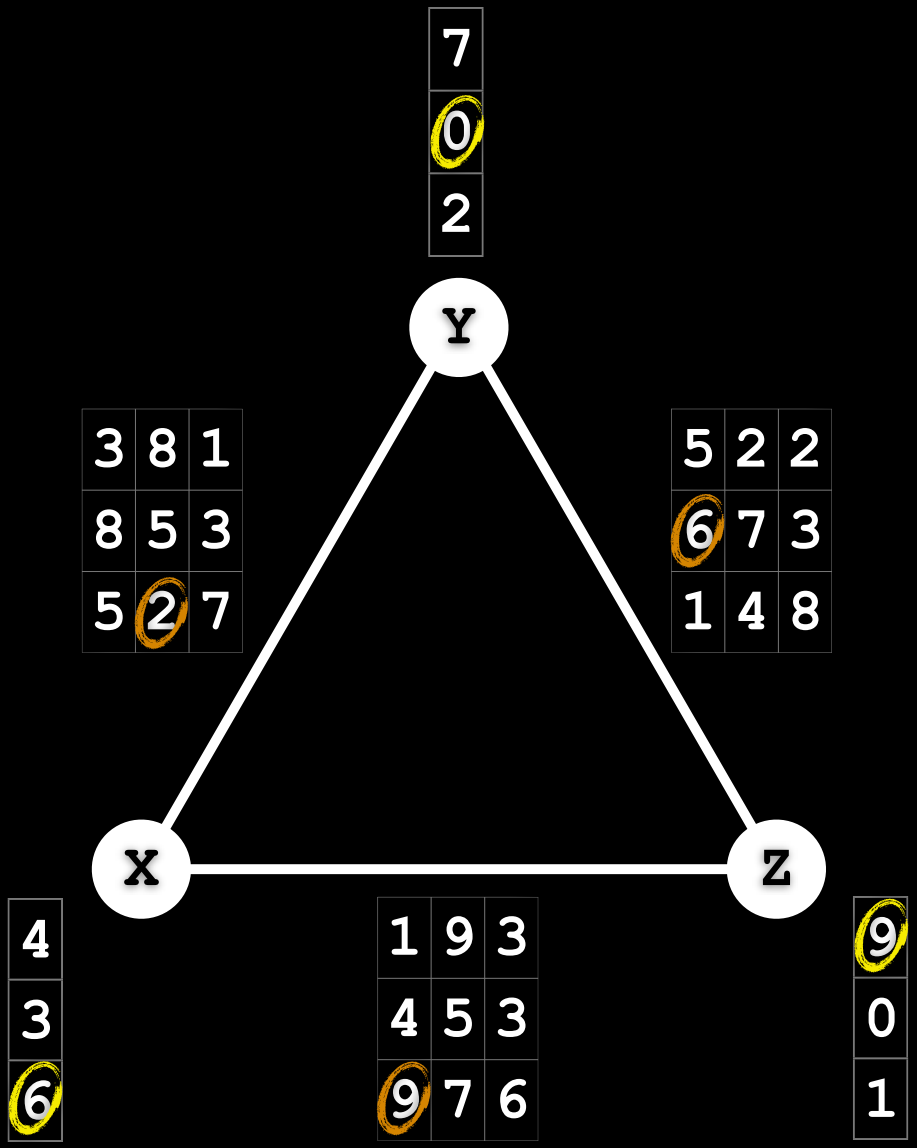
PBQP Example



Solution [3,2,1]:

Node Costs: 6+0+9 = 15
Edge Costs: 2+6+9 = 17

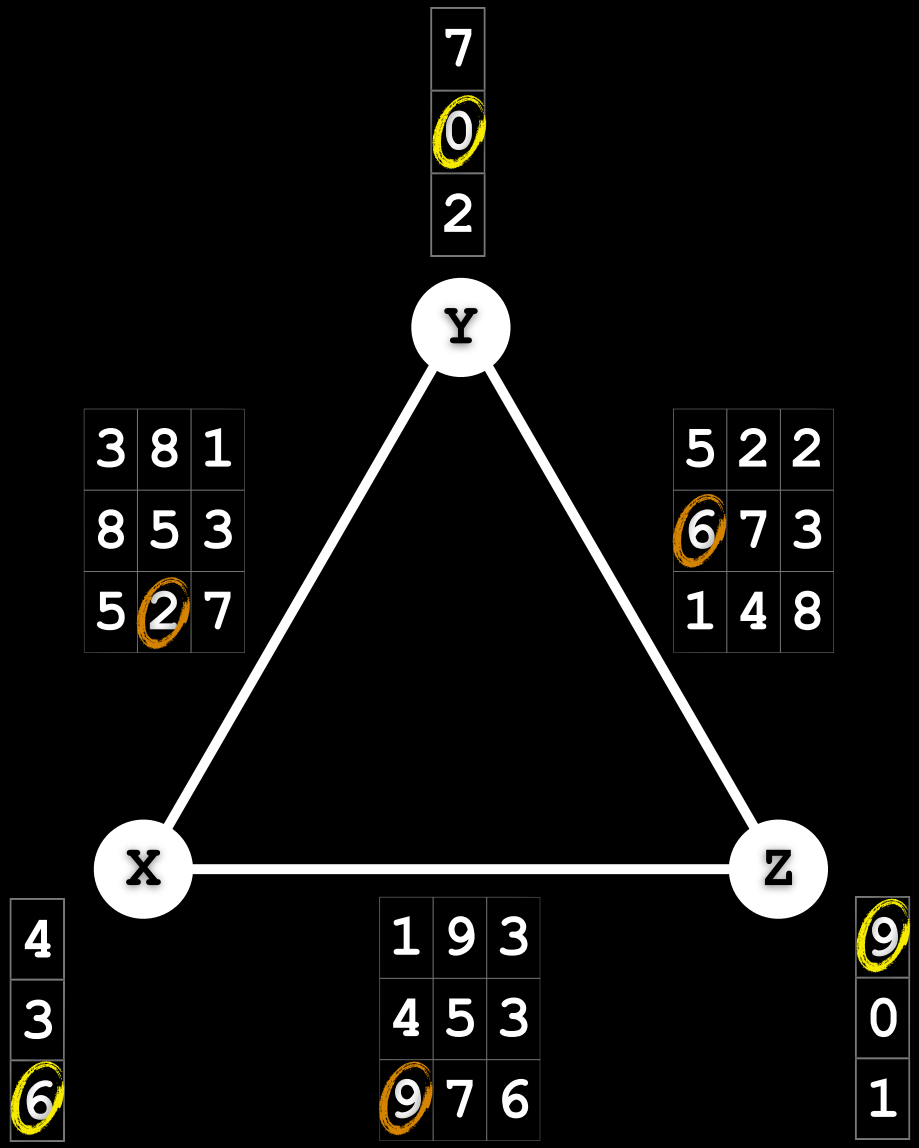
PBQP Example




```
Solution [3,2,1]:

Node Costs:   6+0+9 = 15
Edge Costs:   2+6+9 = 17
      Total:        32
```

PBQP Example

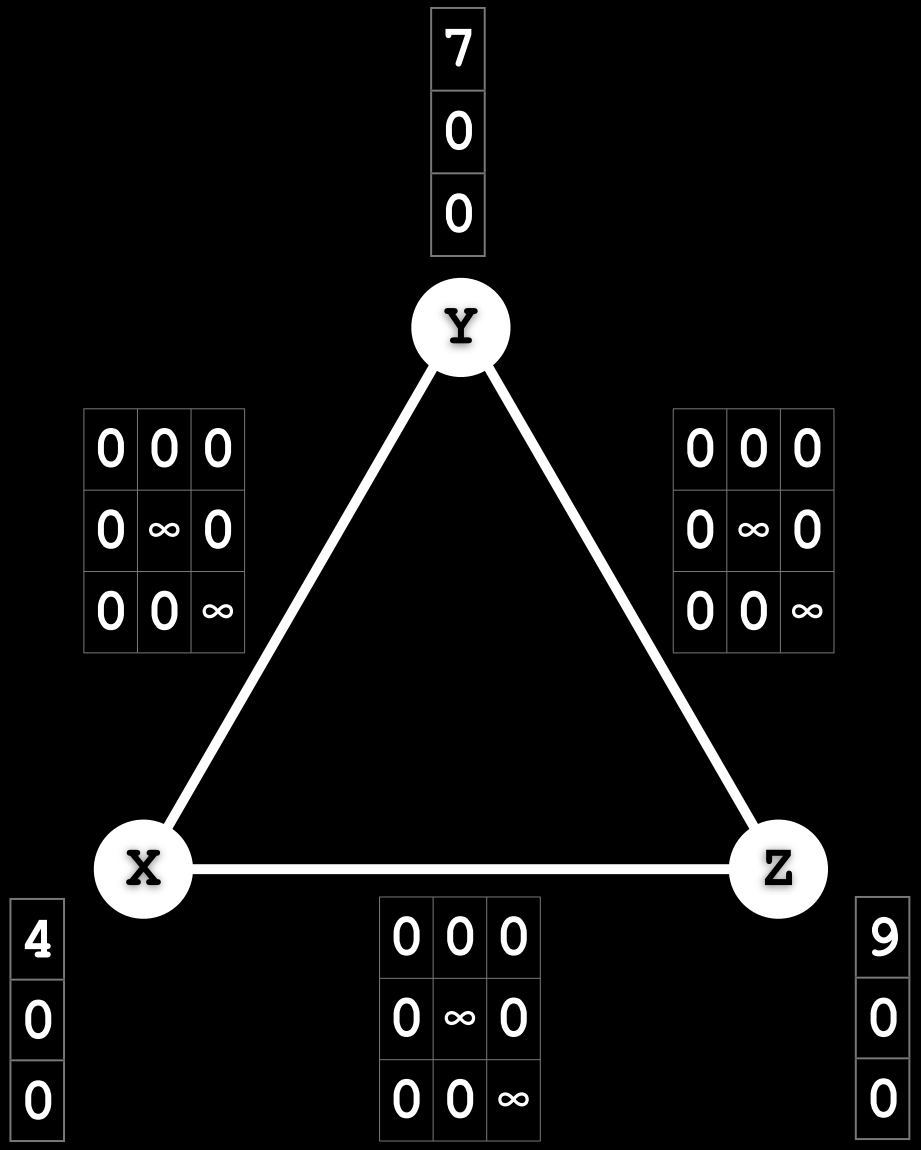


Solution [3,2,1]:

Node Costs: 6+0+9 = 15
Edge Costs: 2+6+9 = 17
Total: 32

Solution [1,2,3]: 19

PBQP Example



For Register Allocation:

Nodes represent virtual registers.

Options reflect storage locations.

Option costs:

Typically zero cost for registers, spill cost estimate for stack slot.

Edge costs:

Depends on the constraint.

Example 1

Interference on a Regular Architecture

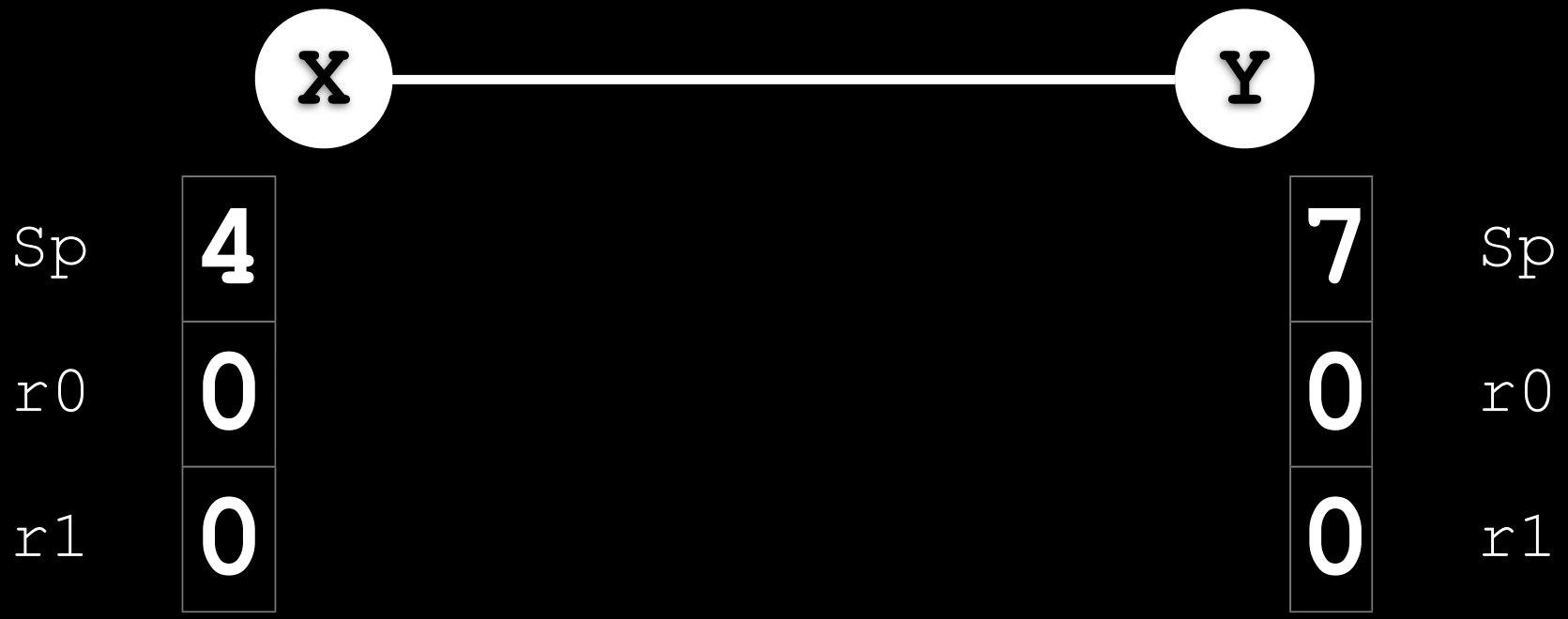
Example 1

Interference on a Regular Architecture

Sp	4
r0	0
r1	0

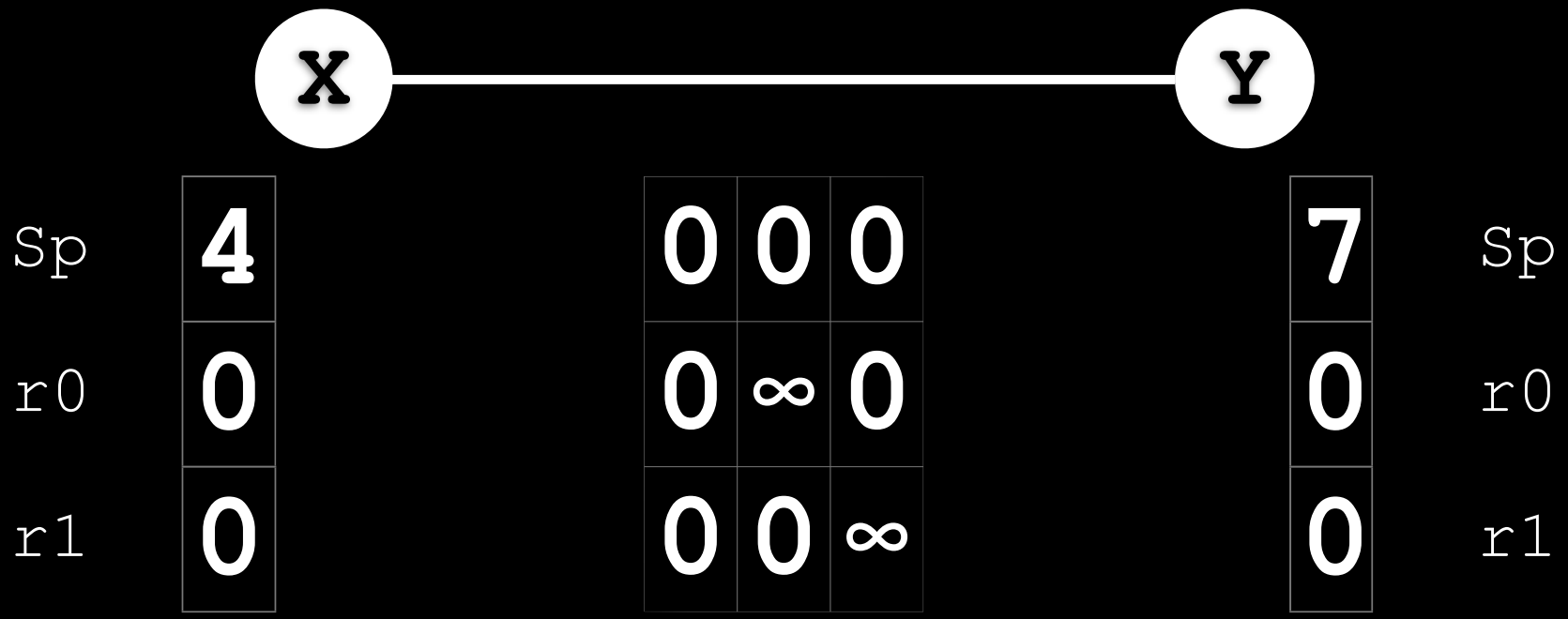
Example 1

Interference on a Regular Architecture



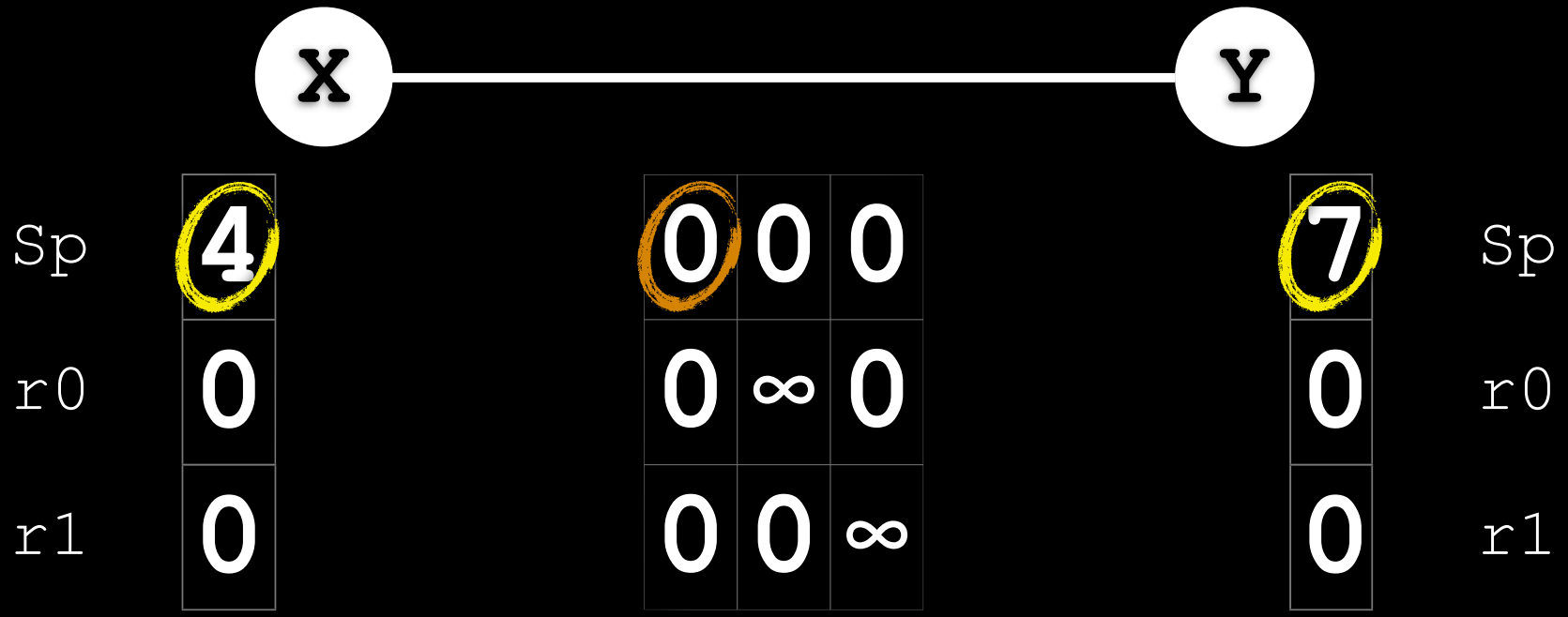
Example 1

Interference on a Regular Architecture



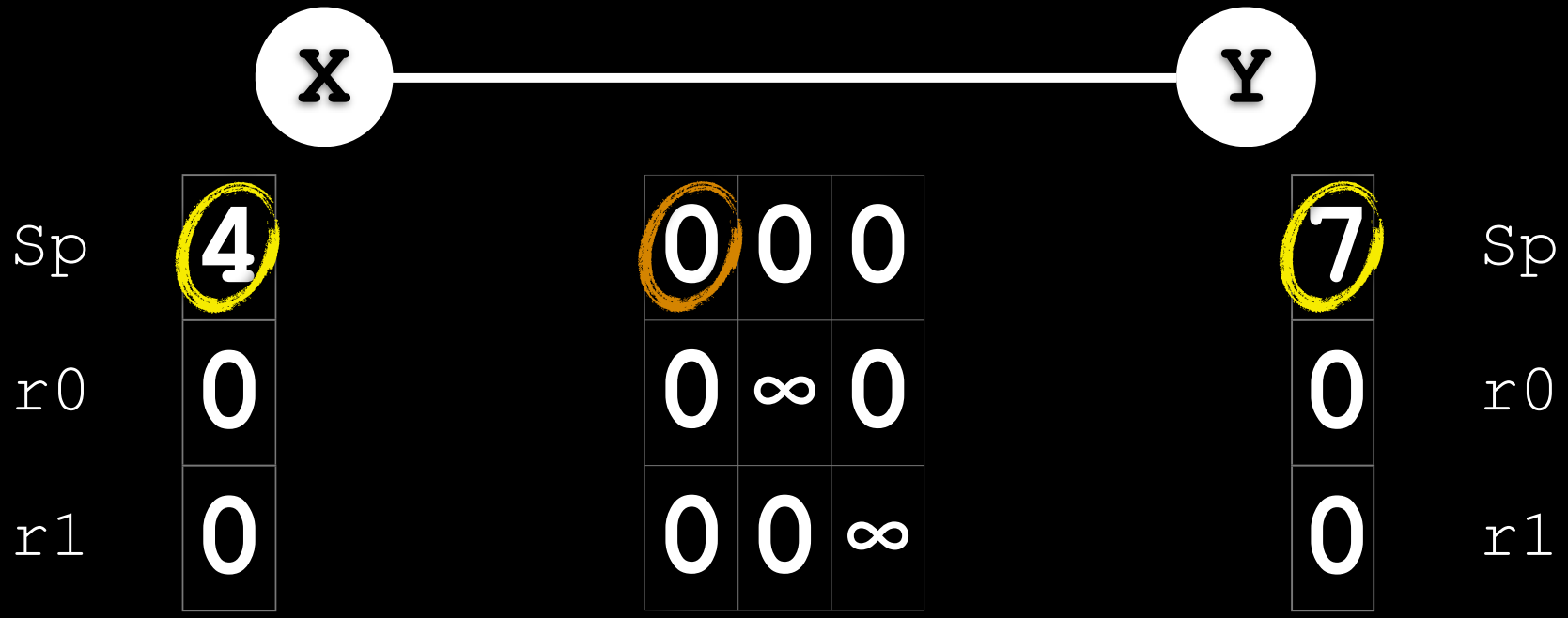
Example 1

Interference on a Regular Architecture



Example 1

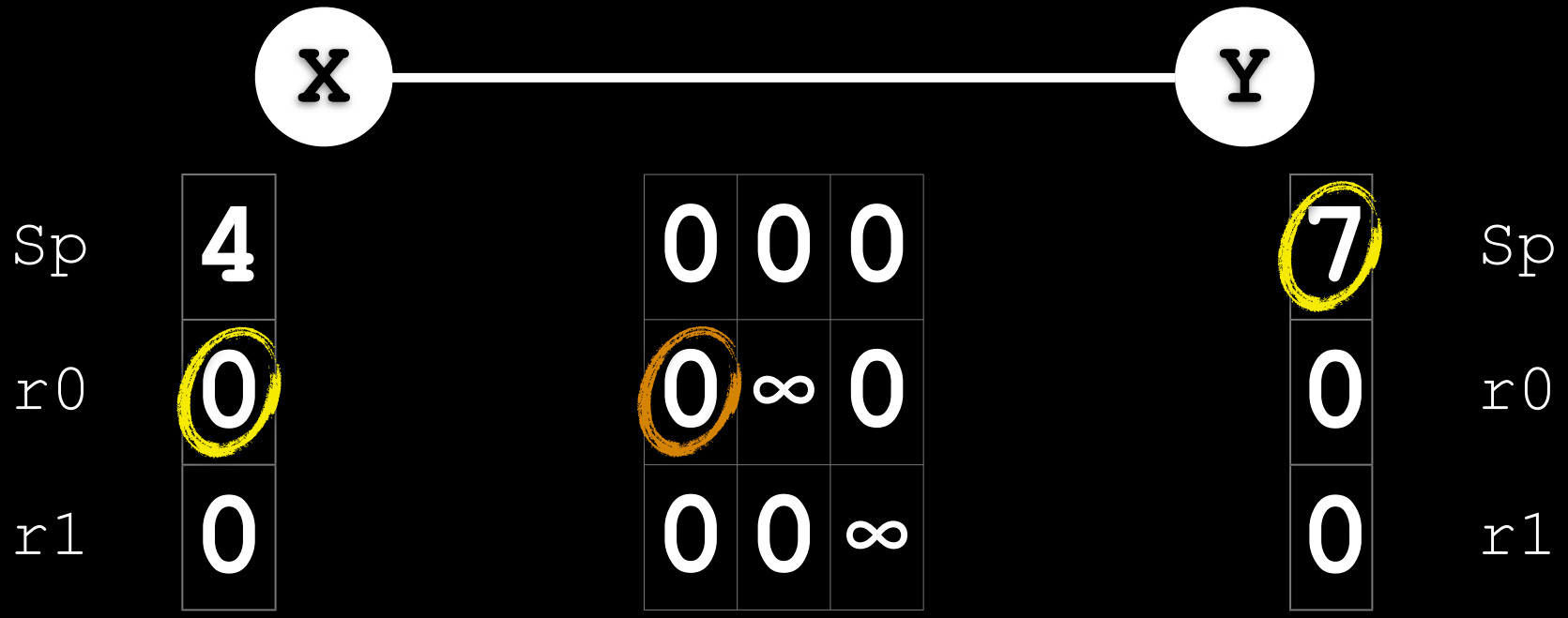
Interference on a Regular Architecture



Cost (Sp, Sp) = 11

Example 1

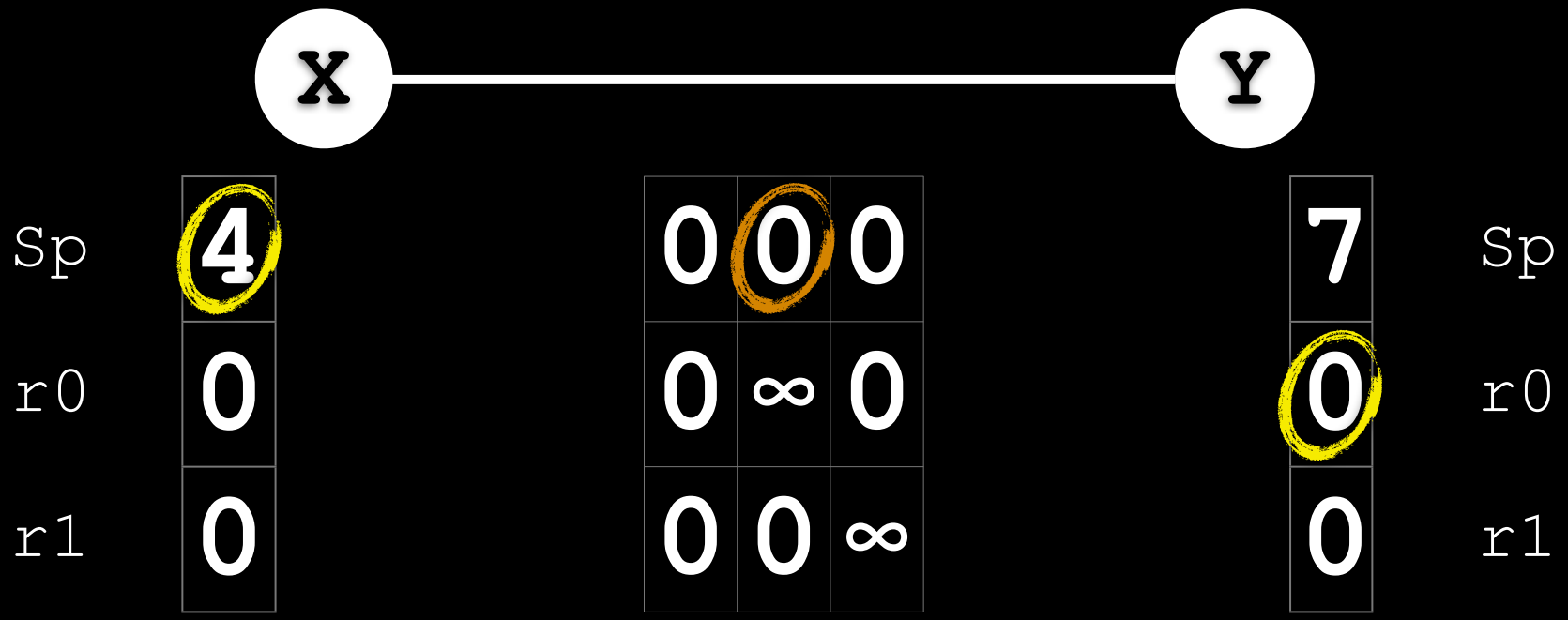
Interference on a Regular Architecture



Cost (r0, Sp) = 7

Example 1

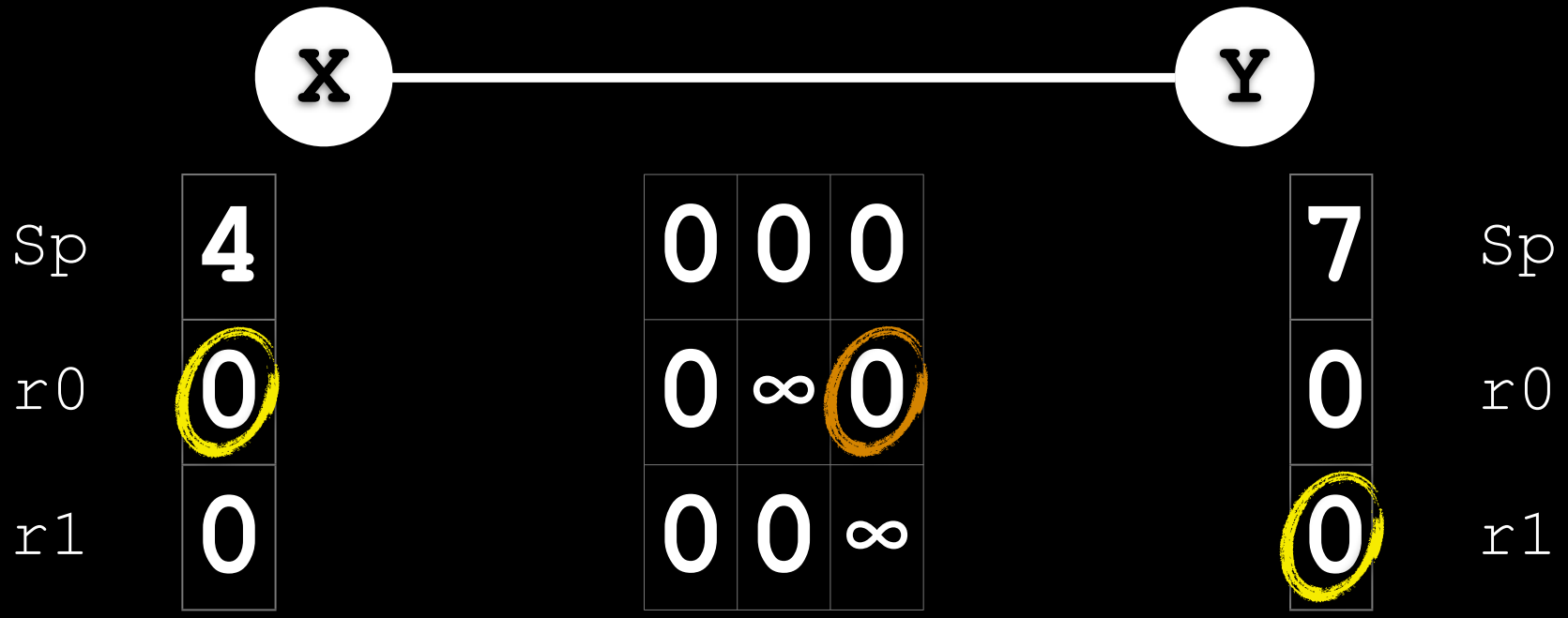
Interference on a Regular Architecture



Cost (Sp, r0) = 4

Example 1

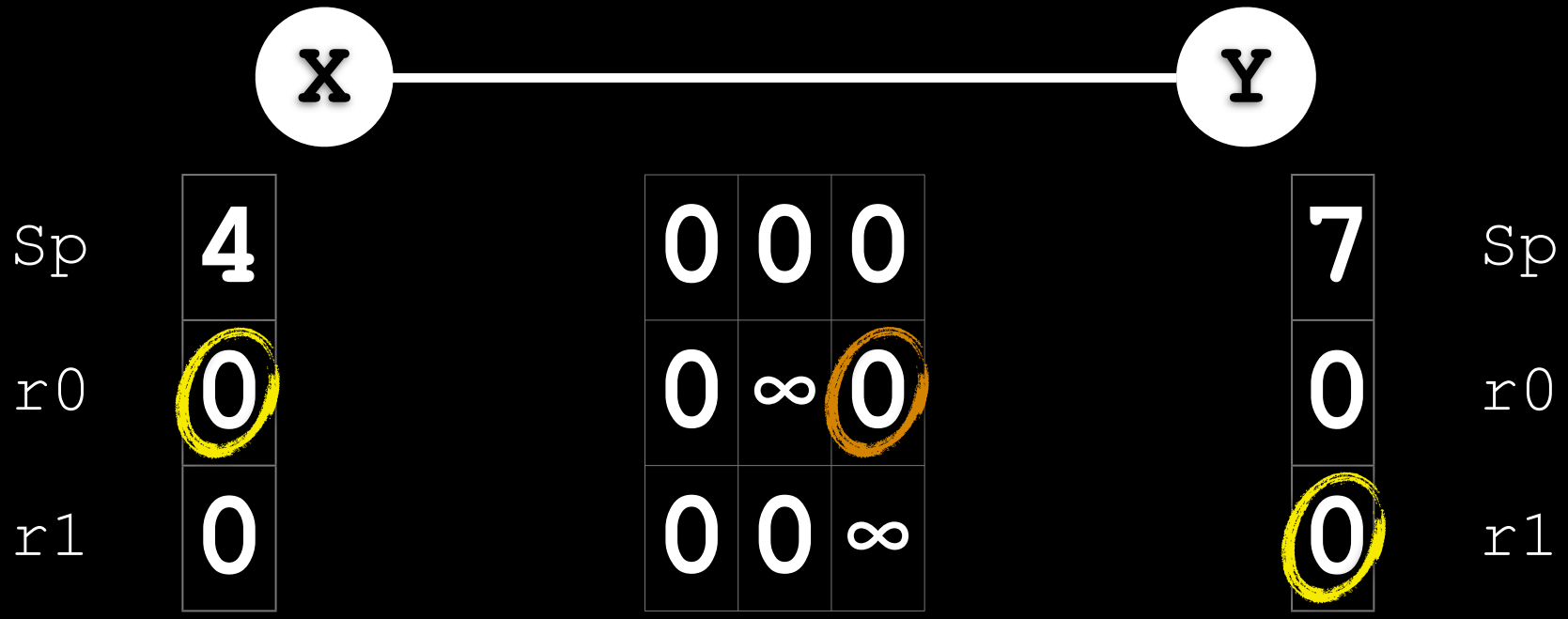
Interference on a Regular Architecture



Cost (r0, r1) = 0

Example 1

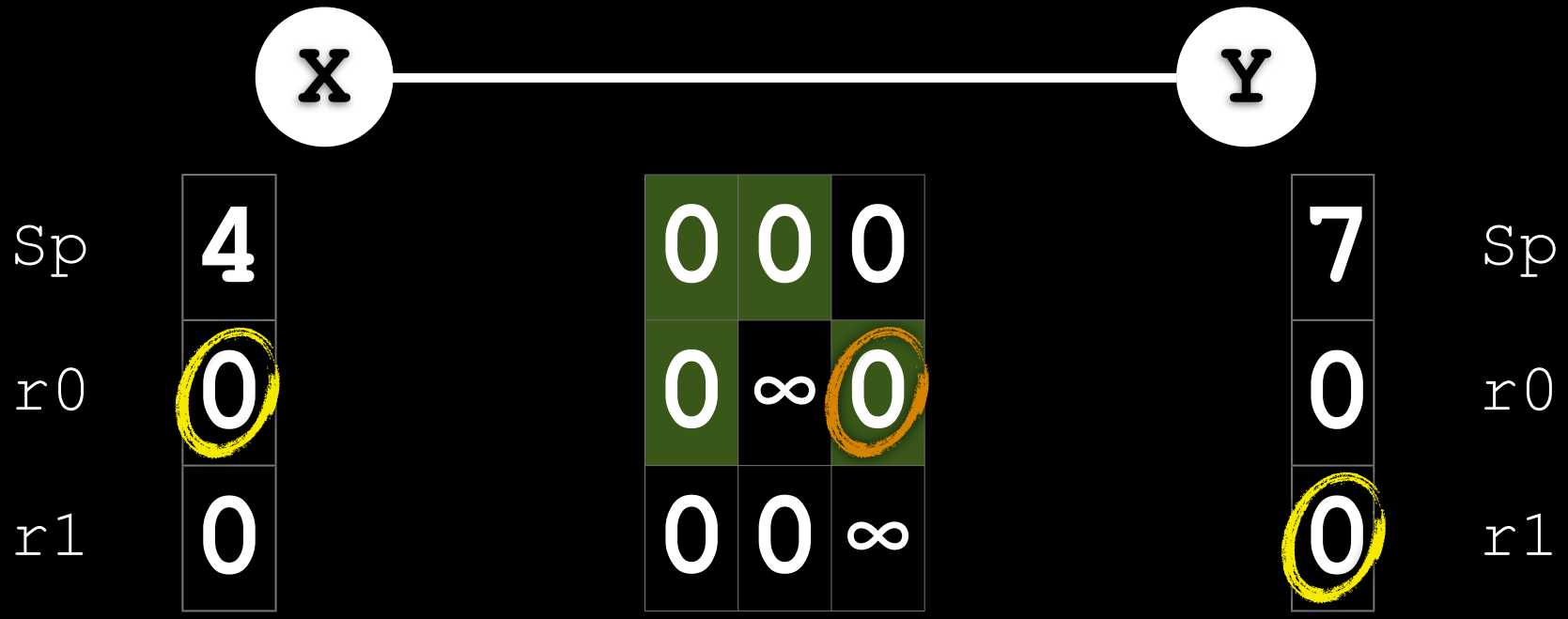
Interference on a Regular Architecture



Cost (r0, r1) = 0 ✓

Example 1

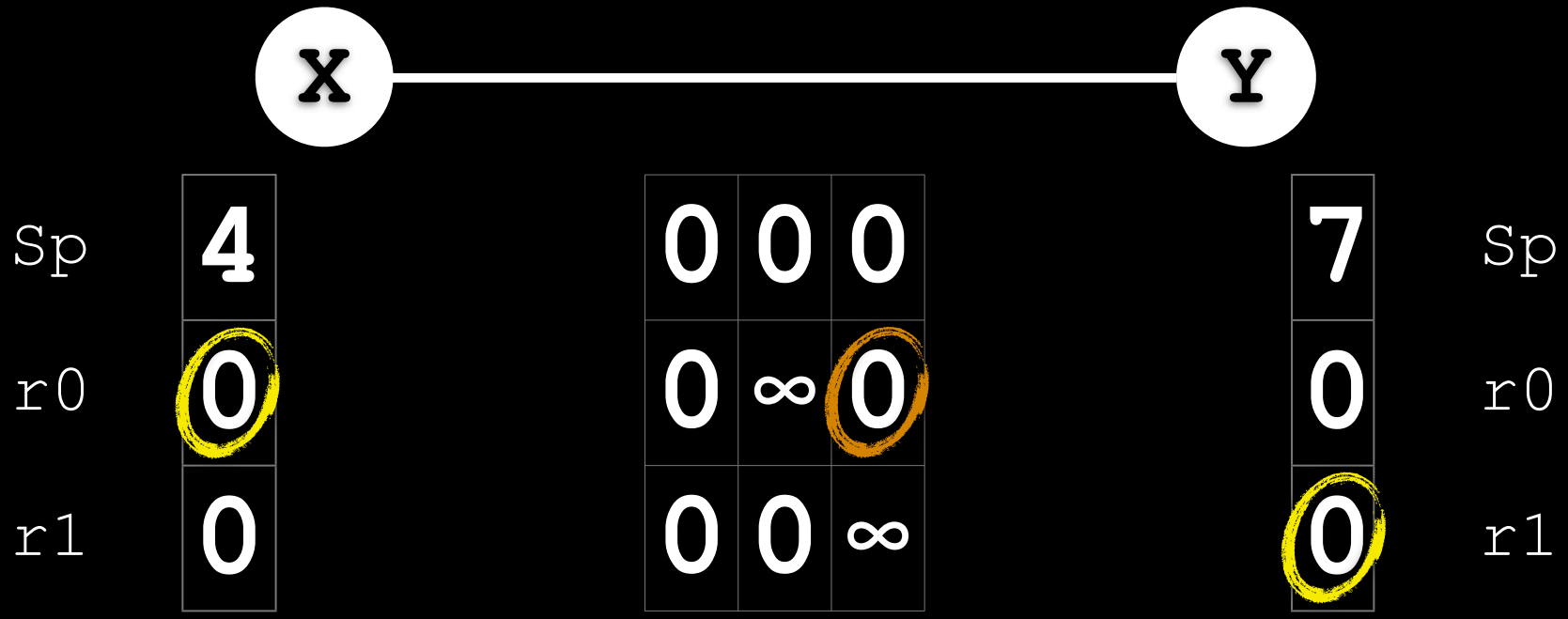
Interference on a Regular Architecture



Cost (r0, r1) = 0 ✓

Example 1

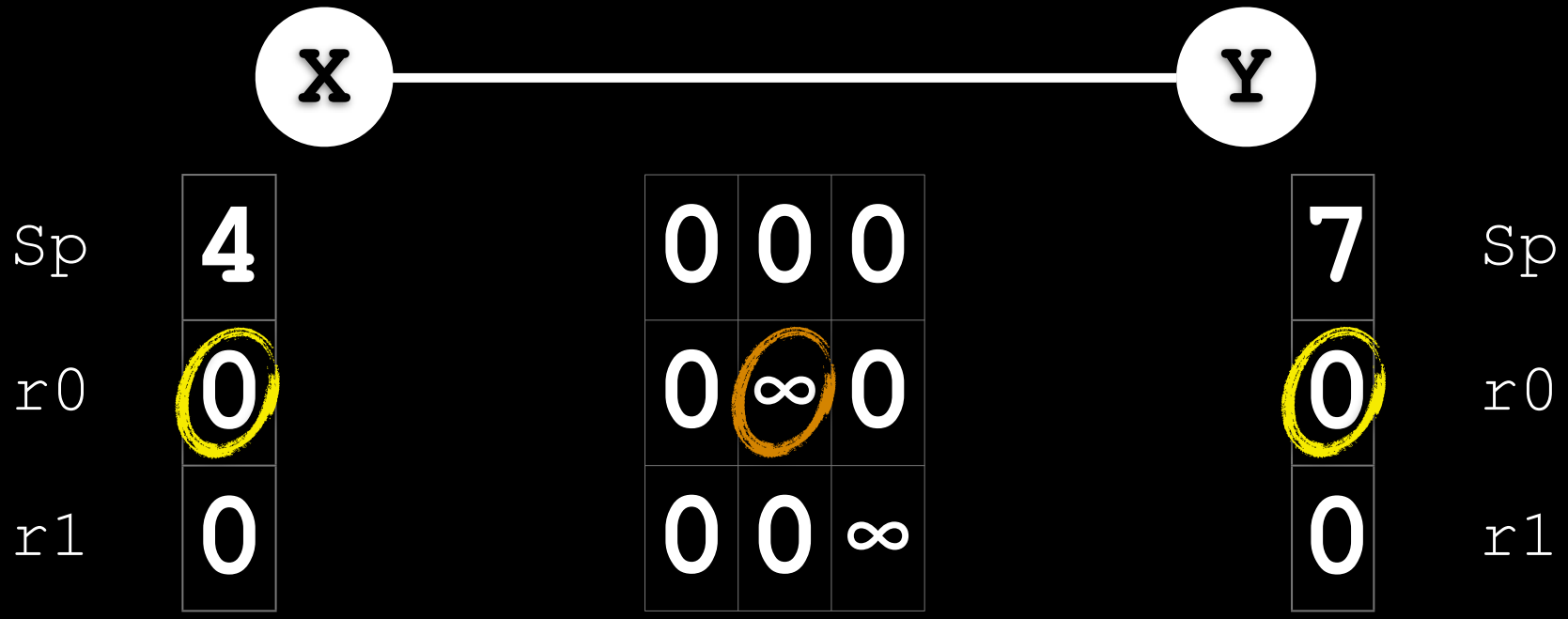
Interference on a Regular Architecture



Cost (r0, r1) = 0 ✓

Example 1

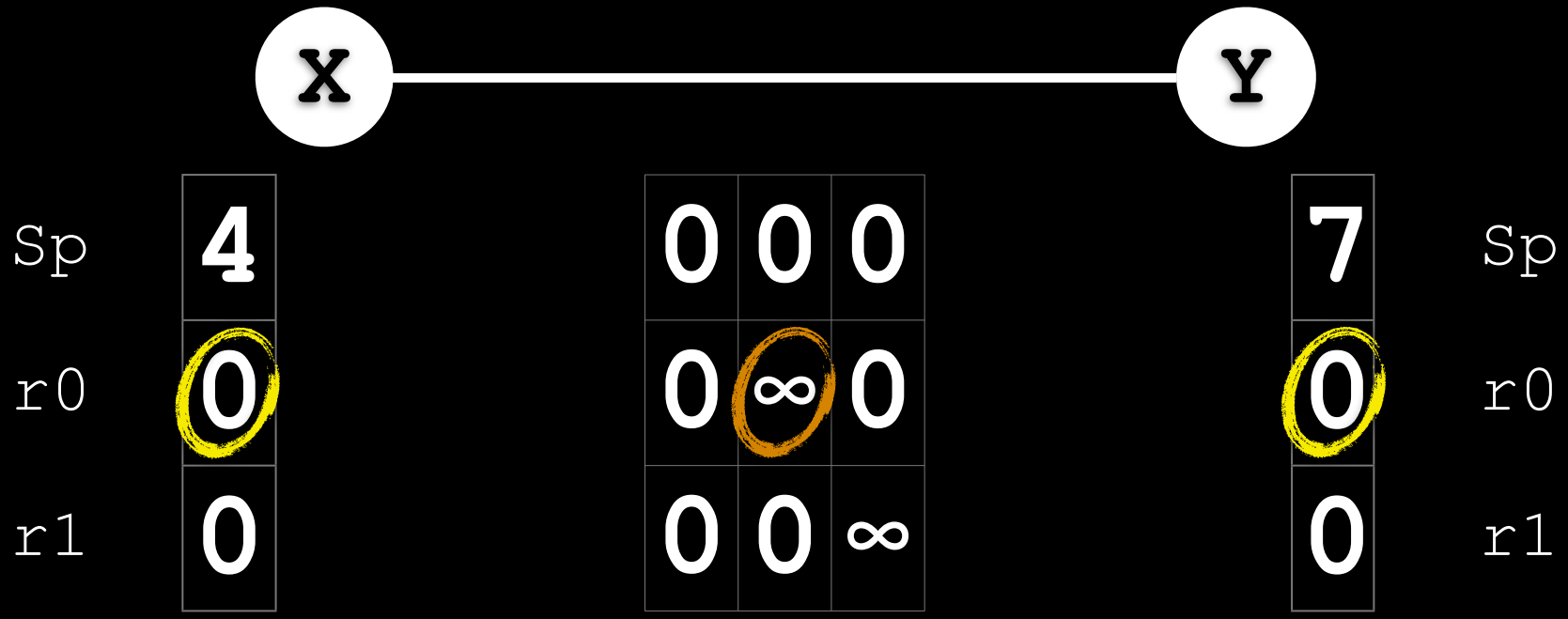
Interference on a Regular Architecture



Cost (r0, r0) = ∞

Example 1

Interference on a Regular Architecture



Cost (r0, r0) = ∞ ✗

Example 1

Interference on a Regular Architecture

0	0	0
0	∞	0
0	0	∞

Example 2

Interference on an Irregular Architecture

	Sp	AL	AH	BL	CL
Sp	0	0	0	0	0
AX	0	∞	∞	0	0
BX	0	0	0	∞	0

Example 3

Coalescing

	Sp	AX	BX
Sp	0	0	0
AX	0	-c	0
BX	0	0	-c

Example 4

Register Pairing (R_i, R_{i+1})

	Sp	s0	s1	s2	s3
Sp	0	0	0	0	0
s0	0	∞	0	∞	∞
s1	0	∞	∞	0	∞
s2	0	∞	∞	∞	0

The PBQP Allocator

```
regalloc -> pbqp
solution = solve pbqp
solution -> allocation
```

The PBQP Allocator

```
regalloc -> pbqp
solution = solve pbqp
solution -> allocation
```

The PBQP Allocator

```
regalloc -> pbqp
solution = solve pbqp
solution -> allocation
```

The PBQP Allocator

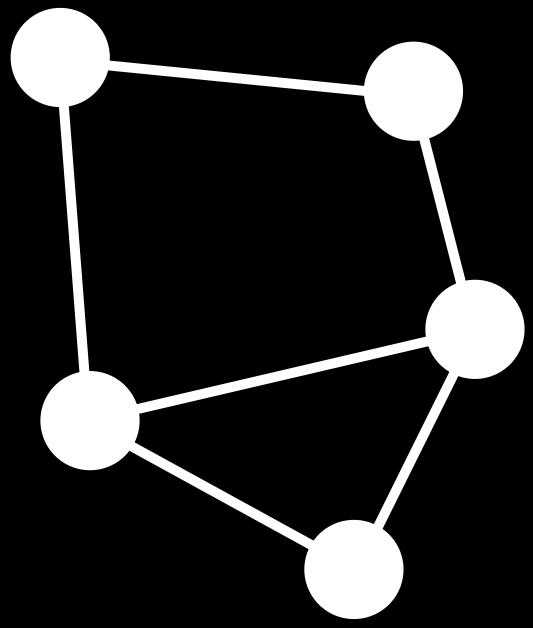
```
regalloc -> pbqp
solution = solve pbqp
solution -> allocation
```

`llvm/lib/CodeGen/RegAllocPBQP.cpp`

How does it work?

Solver uses a graph reduction algorithm.

Reduce problem to the empty graph with reduction rules, then reconstruct it.



PBQP

PBQP

PROS

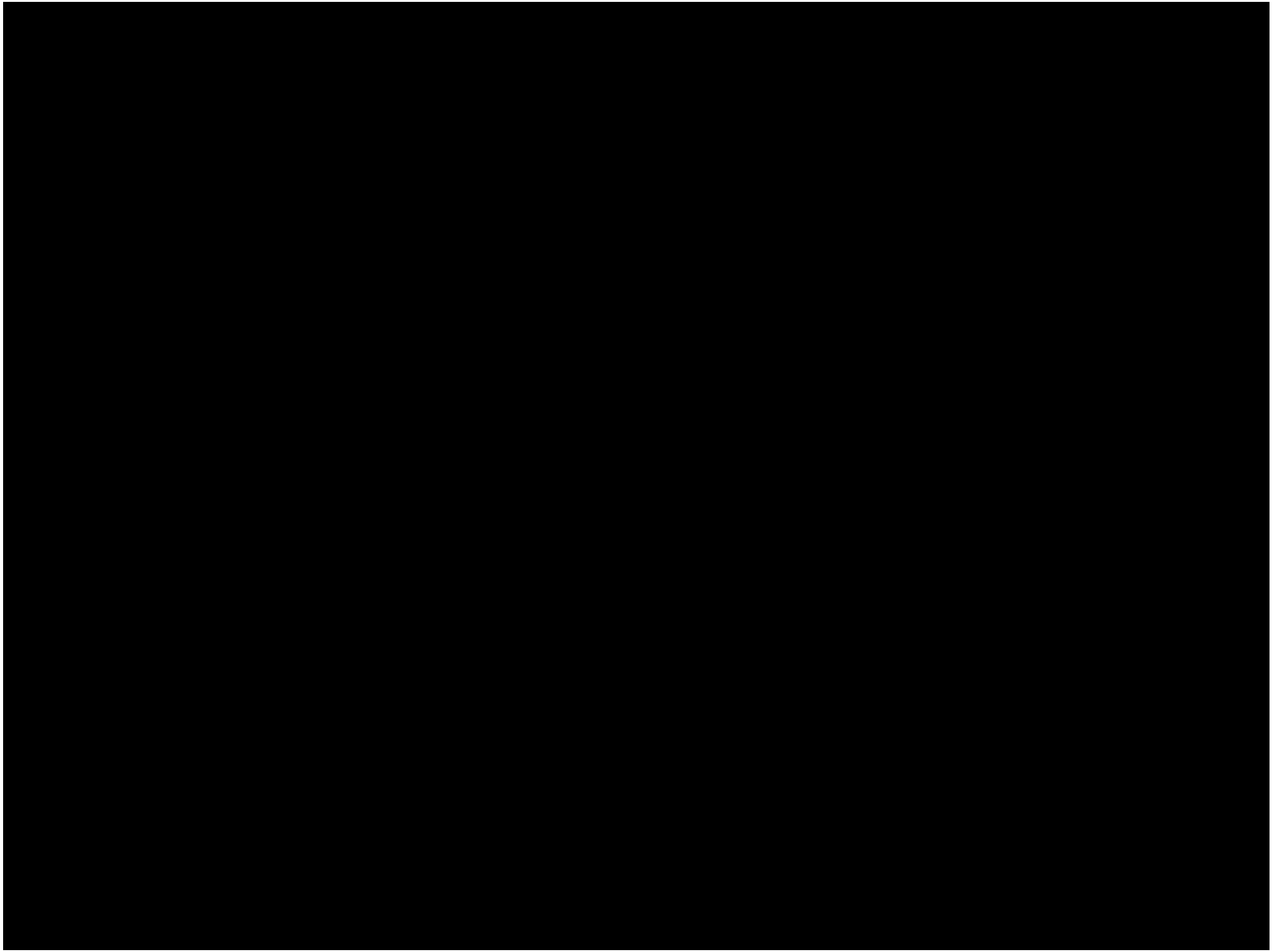
- Ideal for Irregularity
- Very Simple
- Reasonable quality

PBQP

PROS	CONS
• Ideal for Irregularity • Very Simple • Reasonable quality	• Slooooooow

PBQP

PROS	CONS
• Ideal for Irregularity • Very Simple • Reasonable quality • Perfect opportunity for a coffee	• Sloooooooow



- Improved Optimizations.
- New Allocators.
- Cleaner Architecture.
- PBQP.

the end.