LLVM Greedy Register Allocation

Kai
hsiangkai@gmail.com

Outline

- Introduction to Register Allocation Problem
- LLVM Register Allocation Template Method
- LLVM Basic Register Allocation
- LLVM Greedy Register Allocation

Introduction to Register Allocation

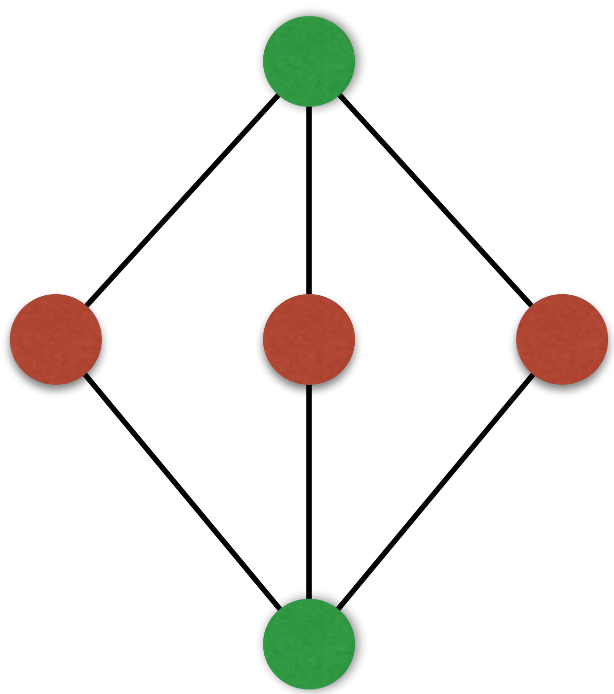
- Definition
  - Register allocation is the problem of mapping program variables to either machine registers or memory addresses.
- Best solution
  - minimise the number of loads/stores from/to memory
- NP-complete

```c
int main()
{
    int i, j;
    int answer;

    for (i = 1; i < 10; i++)
        for (j = 1; j < 10; j++) {
            answer = i * j;
        }

    return 0;
}
```

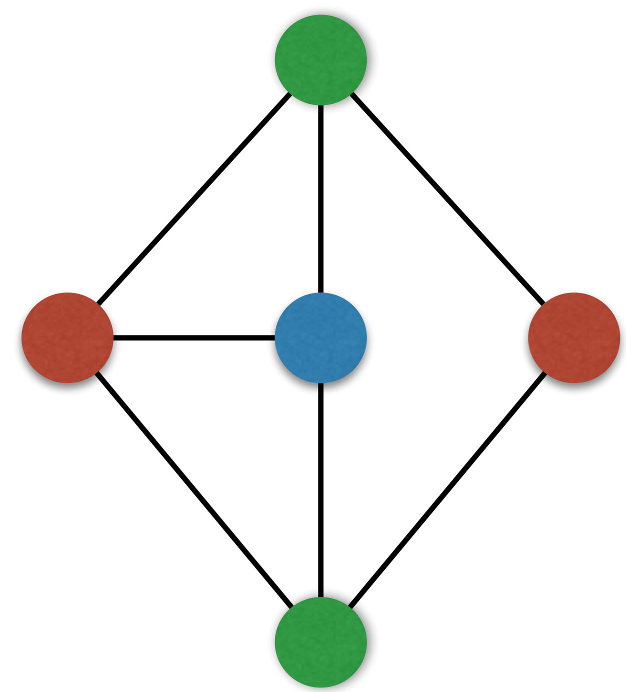
```
_main:
@ BB#0:
        sub  sp, #16
        movs r0, #0
        str  r0, [sp, #12]
        movs r0, #1
        str  r0, [sp, #8]
        b    LBB0_2
LBB0_1:

        adds r1, #1
        str  r1, [sp, #8]
LBB0_2:


        ldr  r1, [sp, #8]
        cmp  r1, #9
        bgt  LBB0_6
@ BB#3:

        str  r0, [sp, #4]
        b    LBB0_5
LBB0_4:

        ldr  r2, [sp, #4]
        muls r1, r2, r1
        str  r1, [sp]
        ldr  r1, [sp, #4]
        adds r1, #1
```

Graph Coloring

- For an arbitrary graph G; a coloring of G assigns a color to each node in G so that no pair of adjacent nodes have the same color.



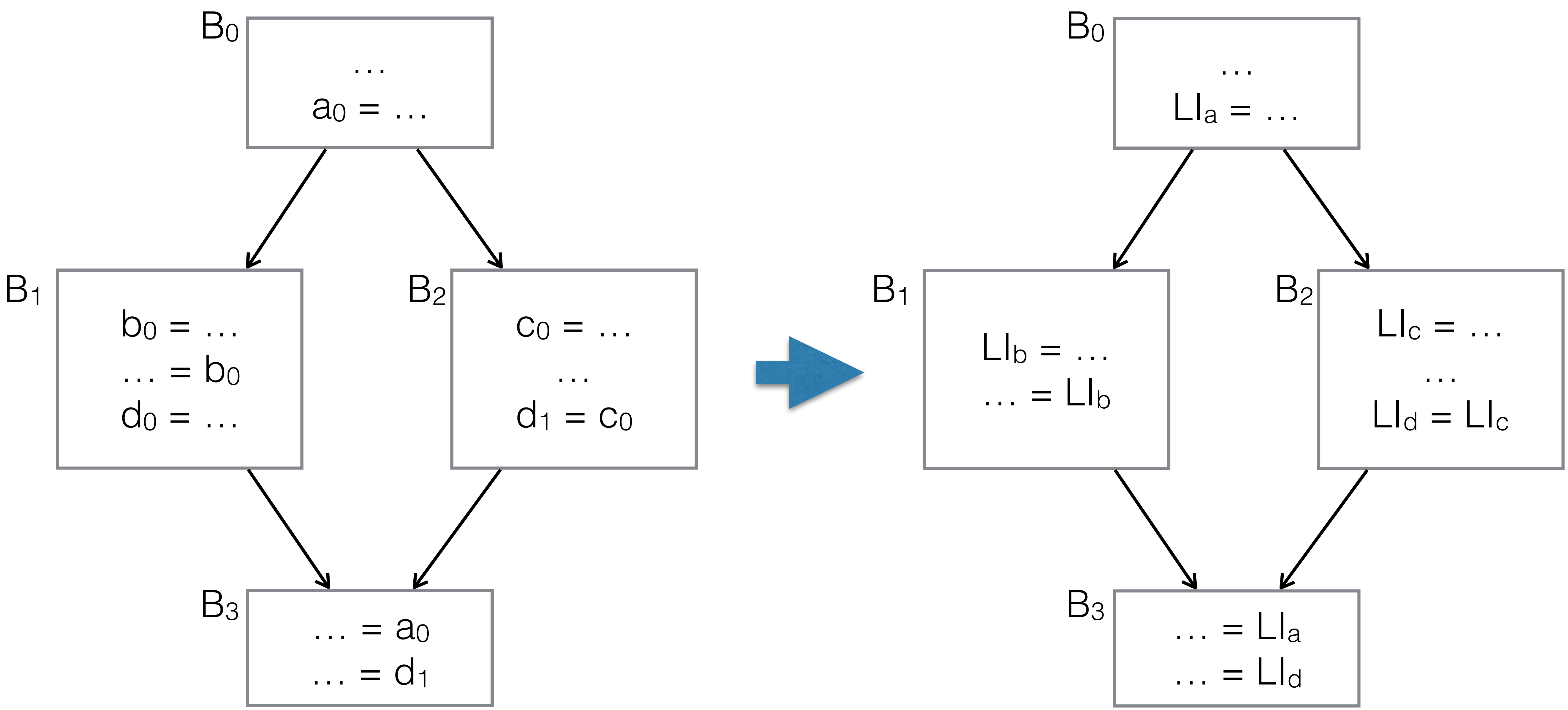
2-colorable



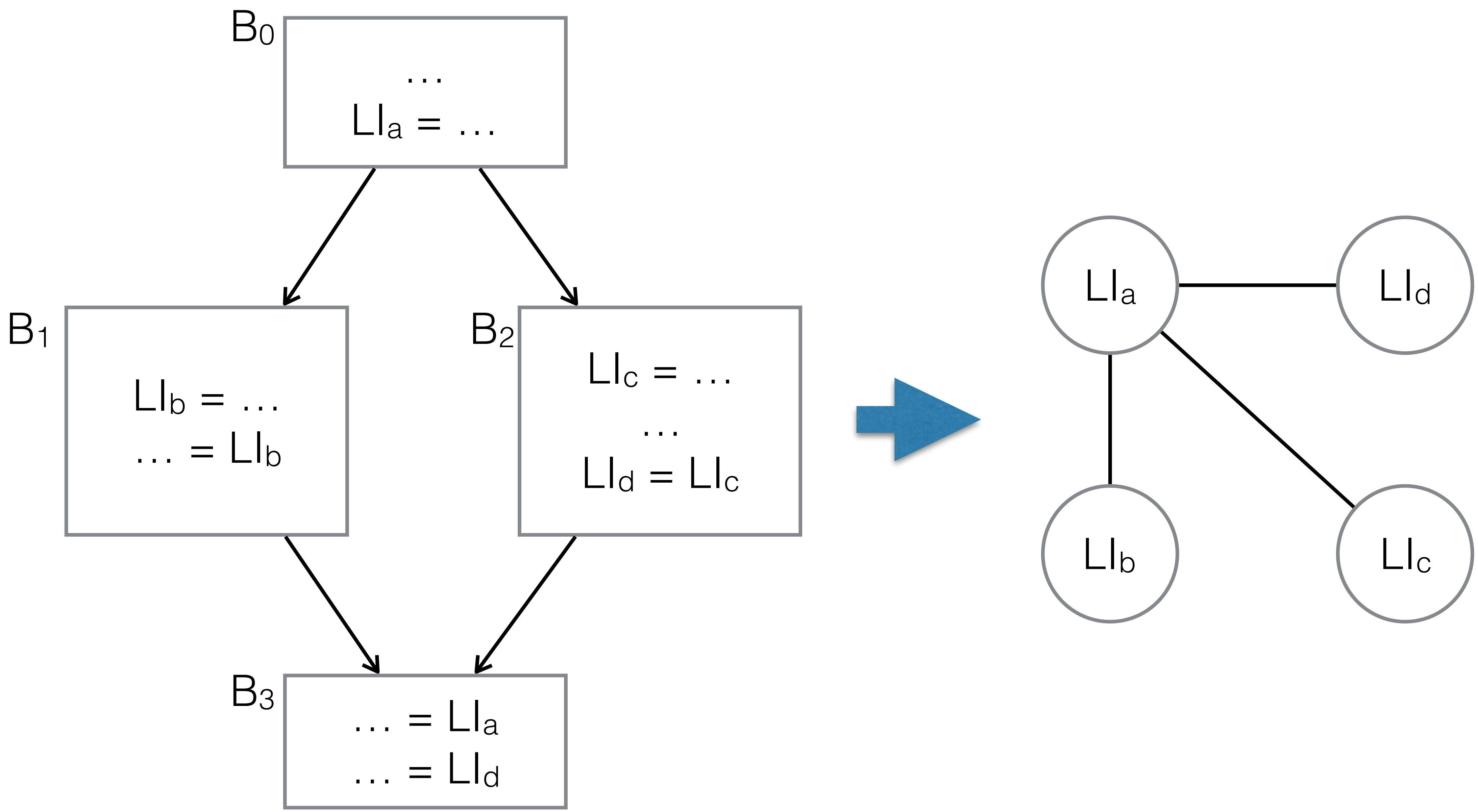
3-colorable

Graph Coloring for RA

- Node: Live interval
- Edge: Two live intervals have interference
- Color: Physical register
- Find a optimal colouring for the graph



B0
...
a0 = ...
B1
b0 = ...
... = b0
d0 = ...
B2
c0 = ...
...
d1 = c0
B3
... = a0
... = d1
B0
...
LIa = ...
B1
LIb = ...
... = LIb
B2
LIc = ...
...
LId = LIc
B3
... = LIa
... = LId

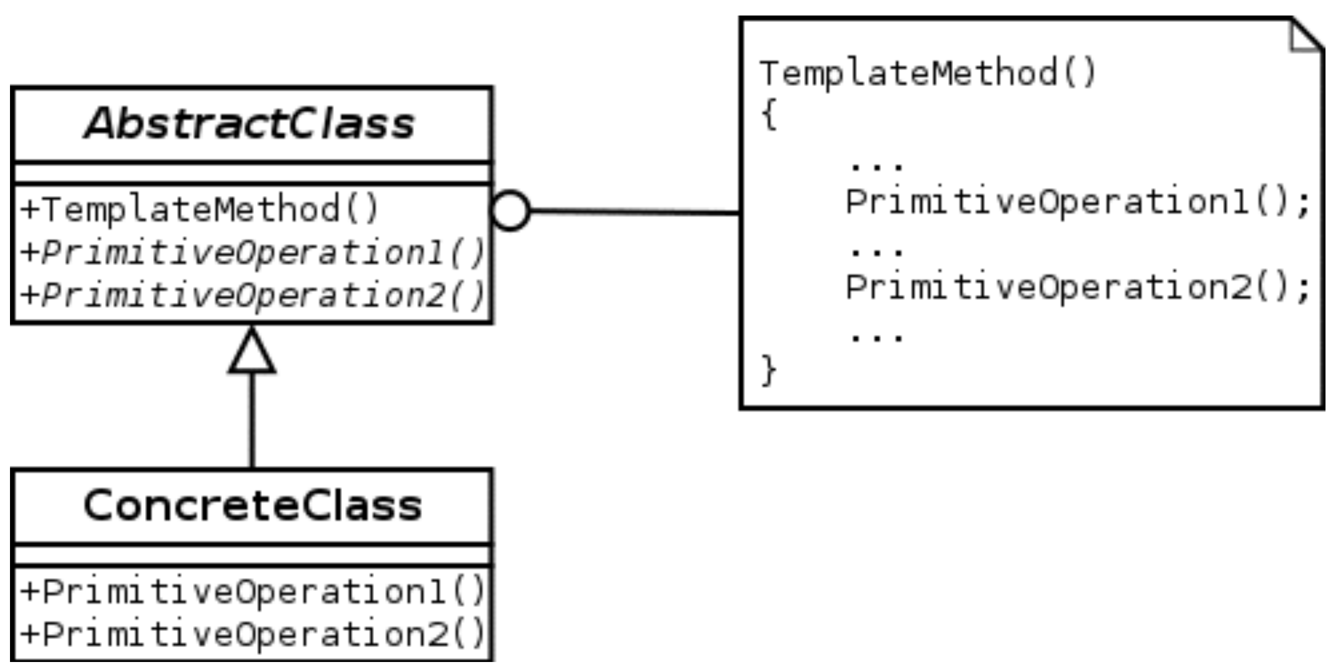

B0
...
LIa = ...
B1
LIb = ...
... = LIb
B2
LIc = ...
...
LId = LIc
B3
... = LIa
... = LId
LIa
LId
LIb
LIc

LLVM Register Allocation

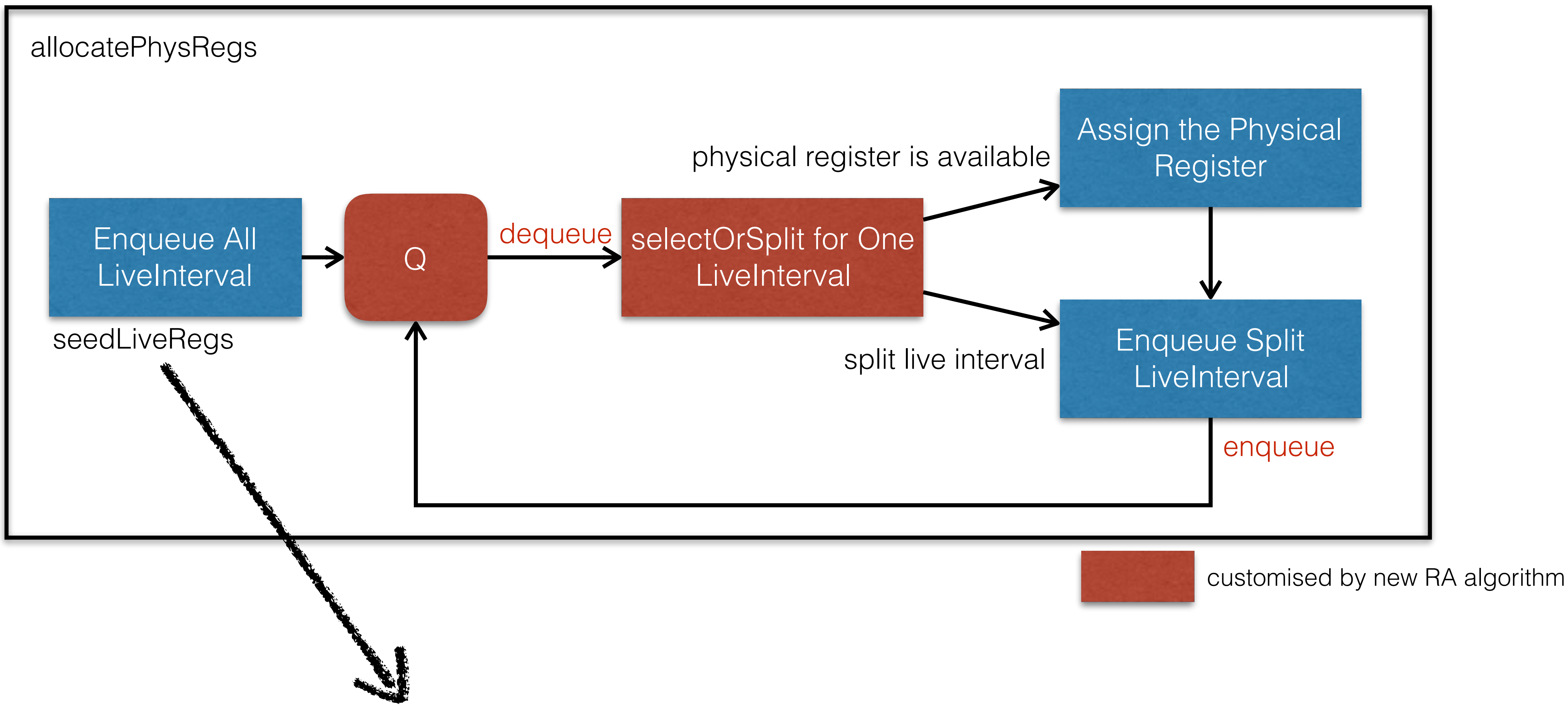
- Basic
  - Provide a minimal implementation of the basic register allocator
- Greedy
  - Global live range splitting.
- Fast
  - This register allocator allocates registers to a basic block at a time.
- PBQP
  - Partitioned Boolean Quadratic Programming (PBQP) based register allocator for LLVM

Template Method

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.



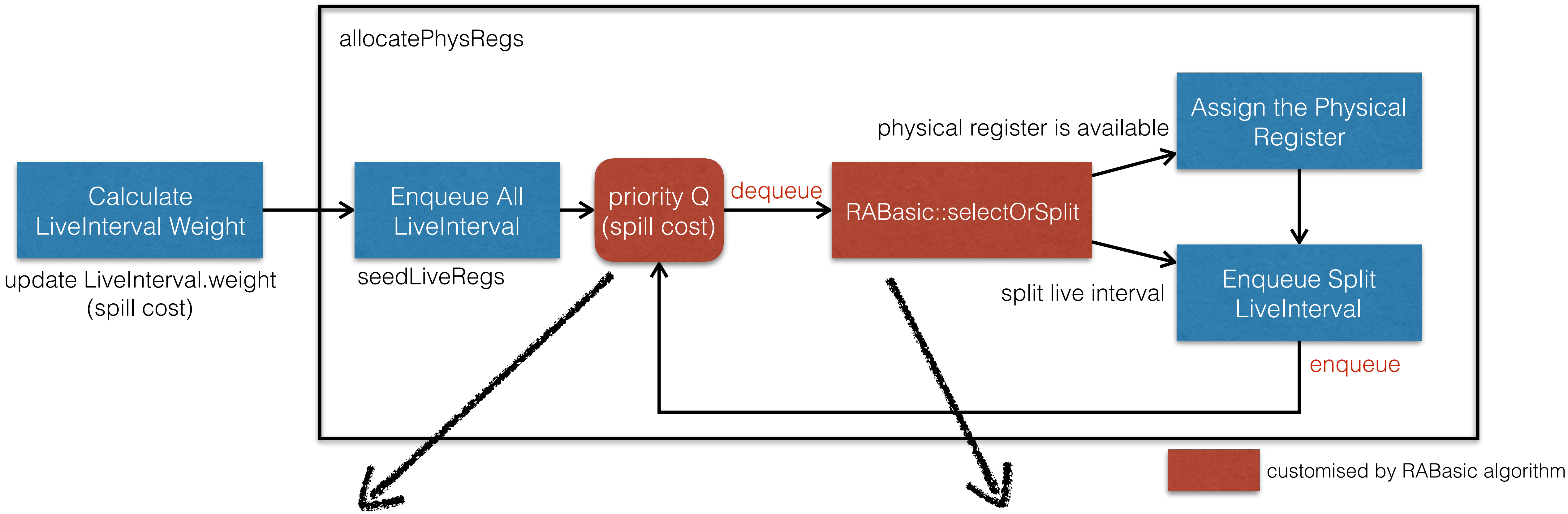
LLVM Register Allocation Template Method




```
for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {
  unsigned Reg = TargetRegisterInfo::index2VirtReg(i);
  if (MRI->reg_nodbg_empty(Reg))
    continue;
  enqueue(&LIS->getInterval(Reg));
}
```

Basic Register Allocation

LLVM Basic Register Allocation




```
struct CompSpillWeight {
  bool operator()(LiveInterval *A, LiveInterval *B) const {
    return A->weight < B->weight;
  }
};
```

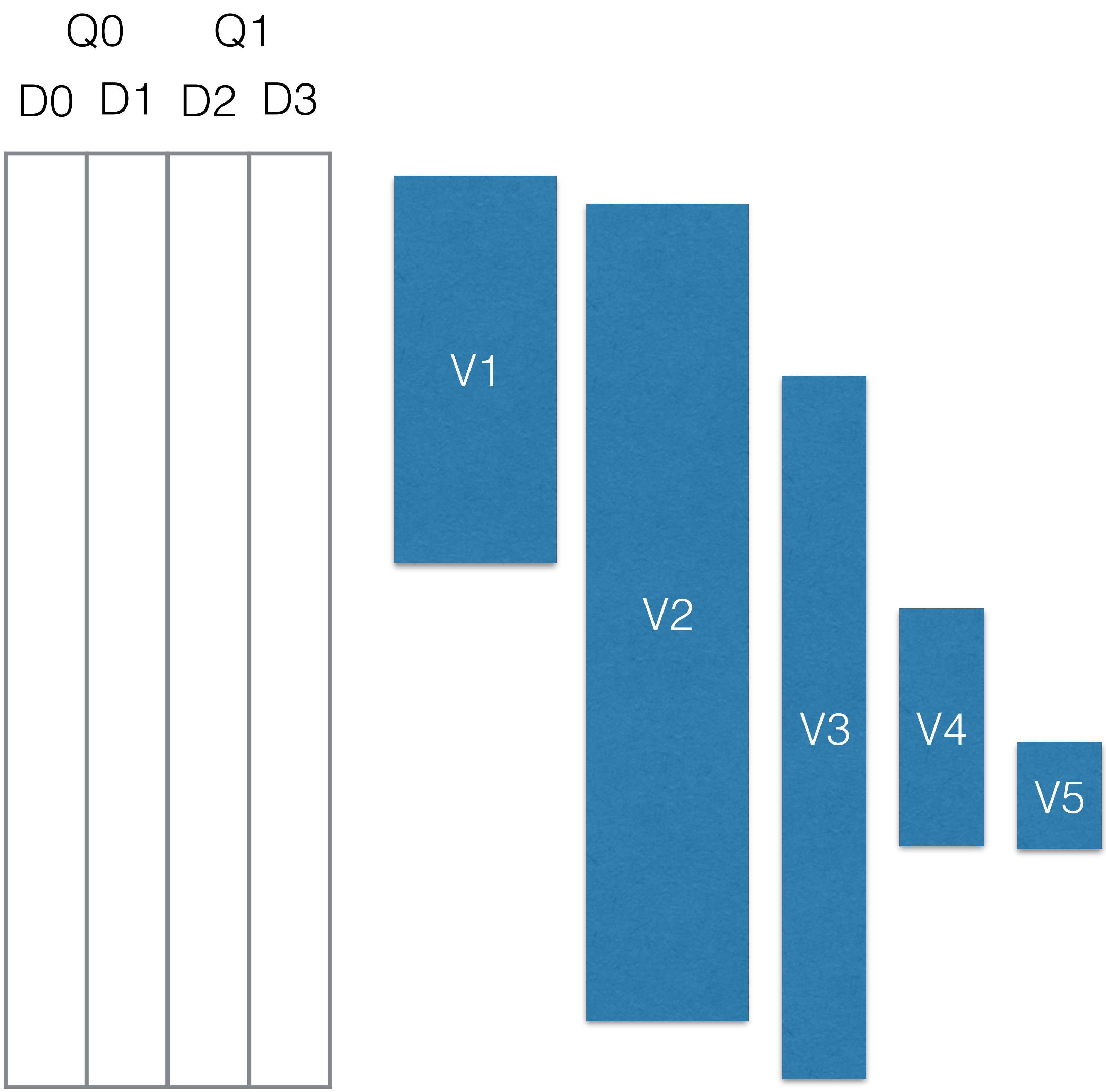
1. Assign physical registers to Live Interval with highest spill cost.
2. If there is no physical registers for current Live Interval, select the highest spill cost Live Interval between current one and interferences to assign physical registers.
3. Spill the unassigned Live Intervals.

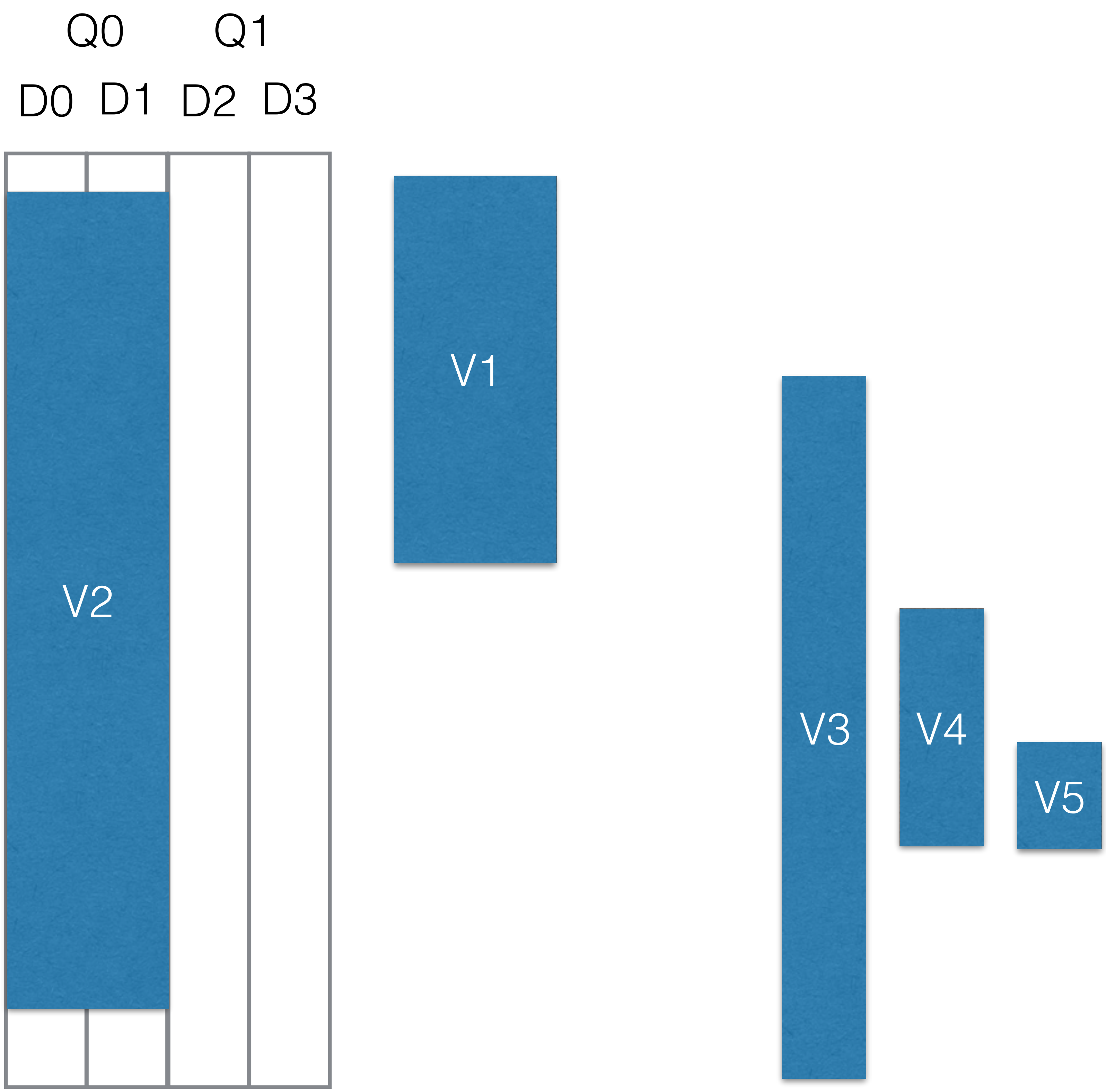
LiveInterval Weight

- Weight for one instruction with the register
  - weight = (isDef + isUse) * (Block Frequency / Entry Frequency)
  - loop induction variable: weight *= 3
- For all instructions with the register
  - totalWeight += weight
- Hint: totalWeight *= 1.01
- Re-materializable: totalWeight *= 0.5
- LiveInterval.weight = totalWeight / size of LiveInterval

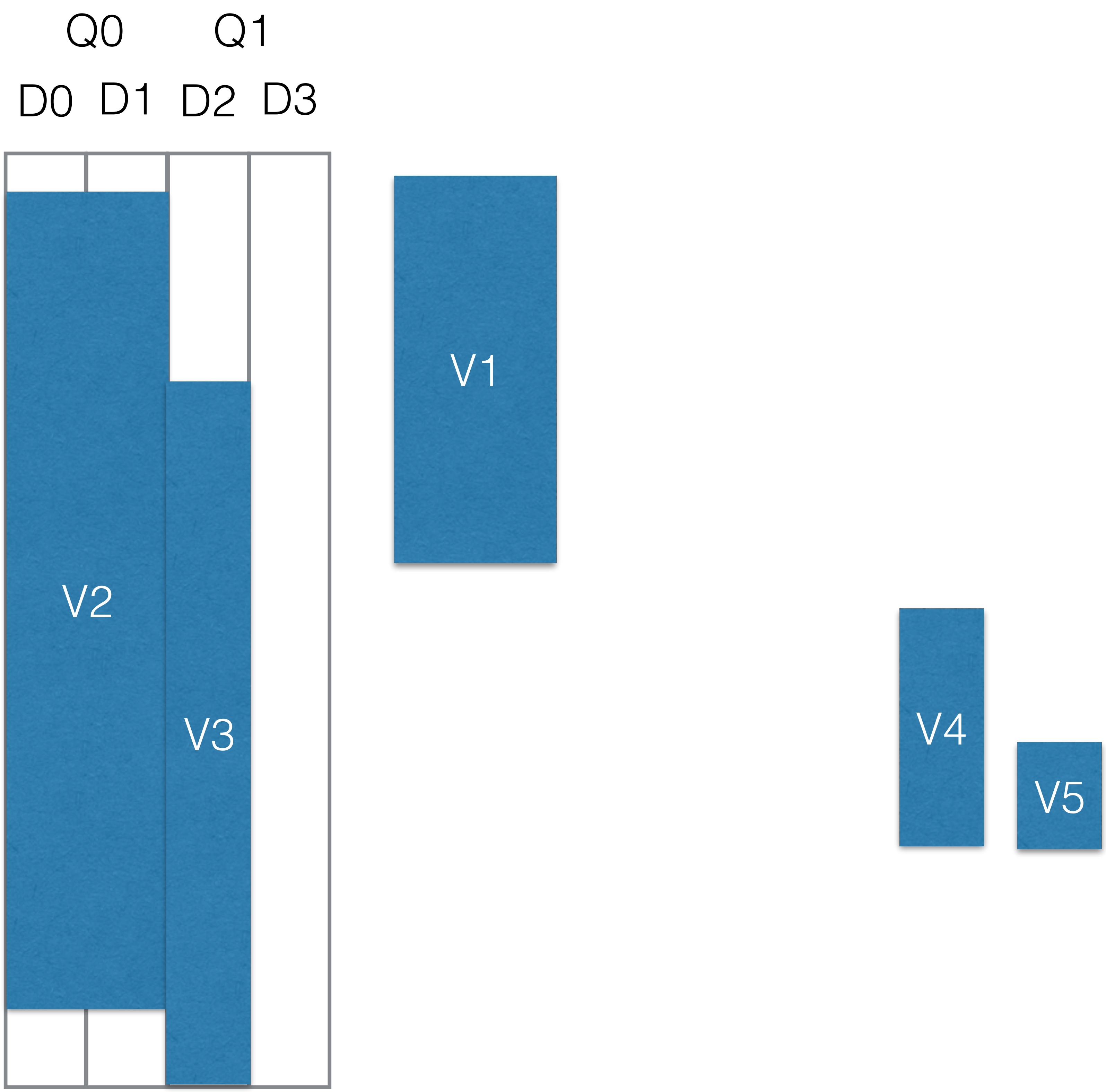
Greedy Register Allocation

- Example (assign physical registers by length)

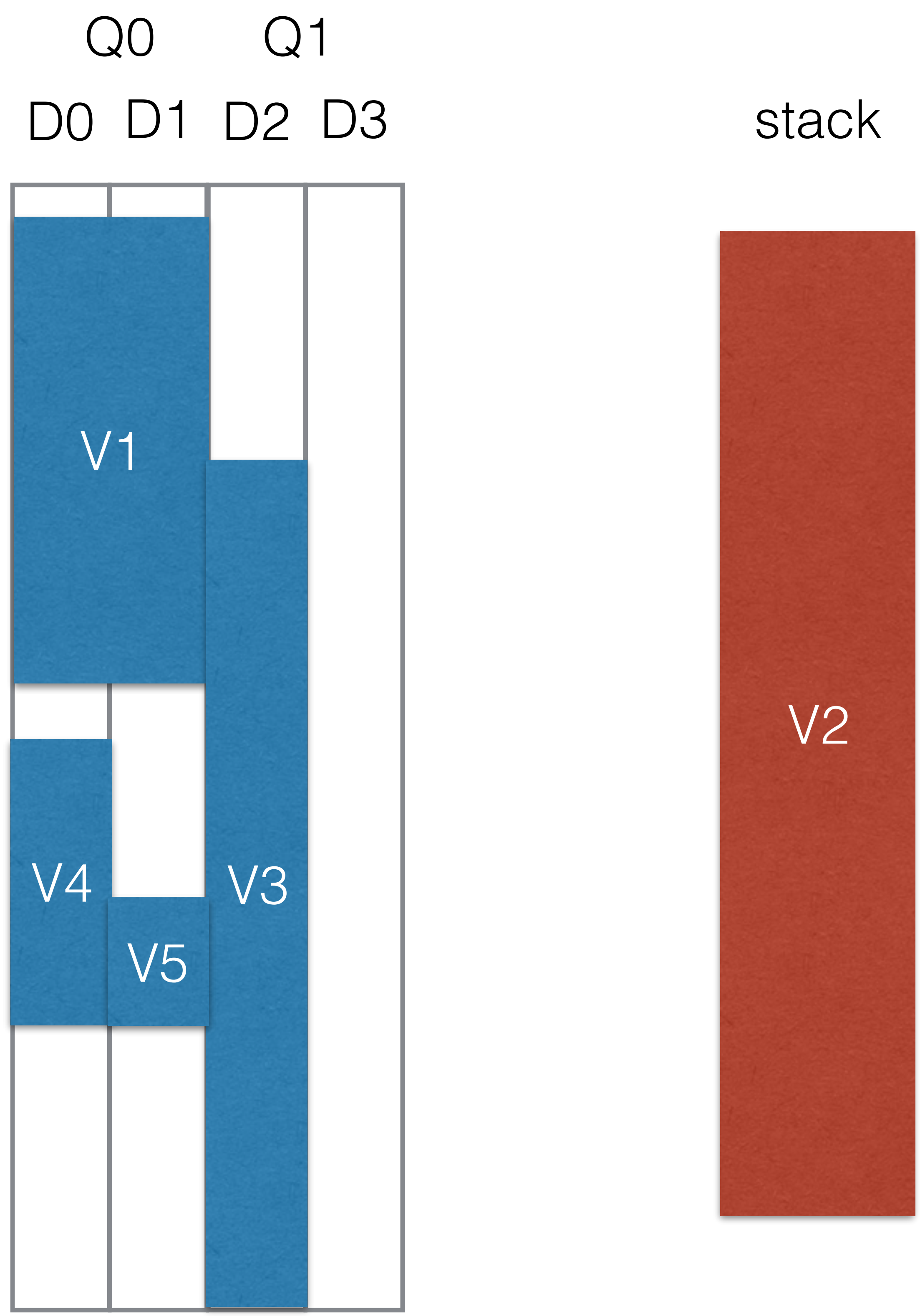



Q0
Q1
D0
D1
D2
D3
V2
V1
V3
V4
V5

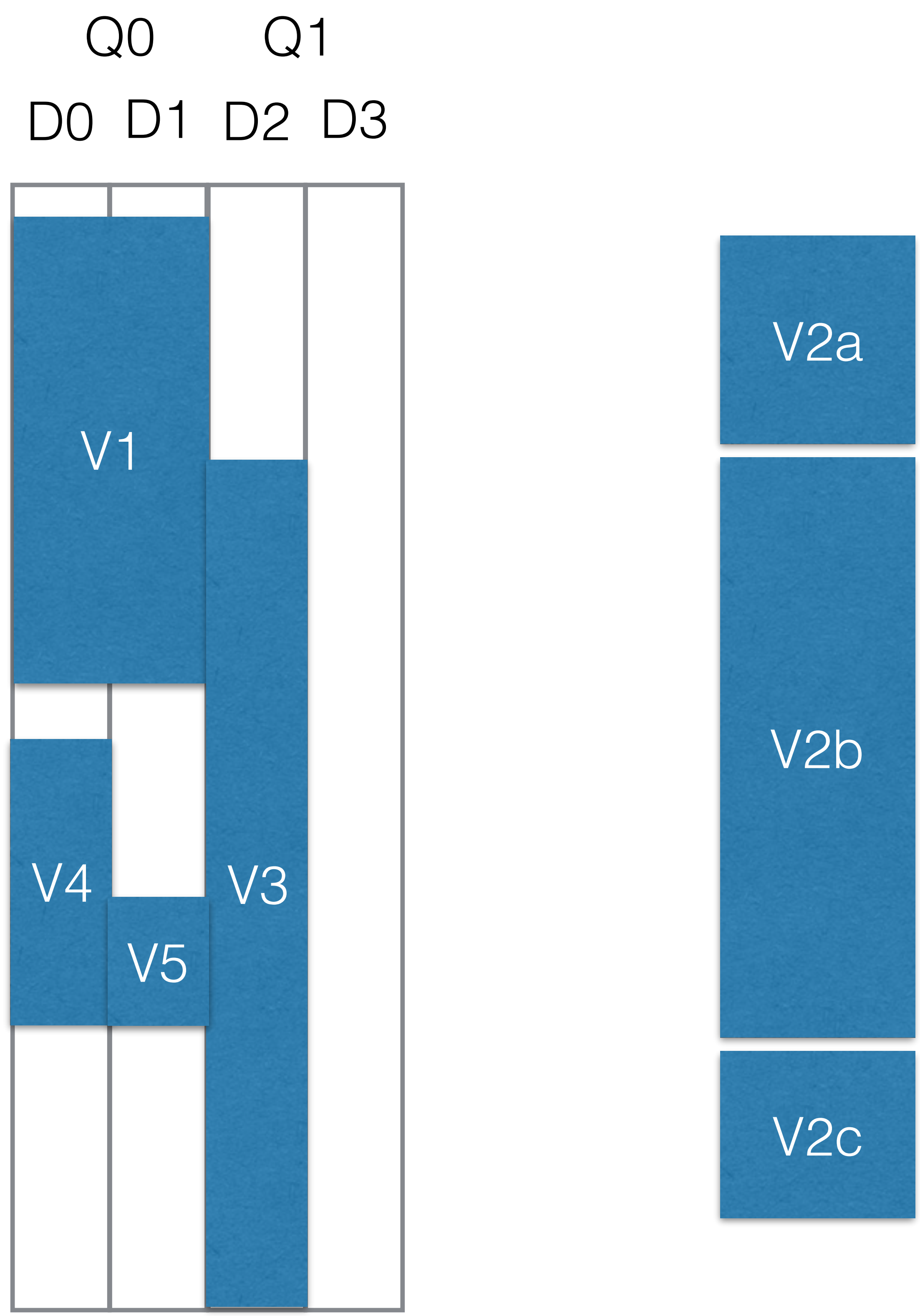
- No physical register for V1



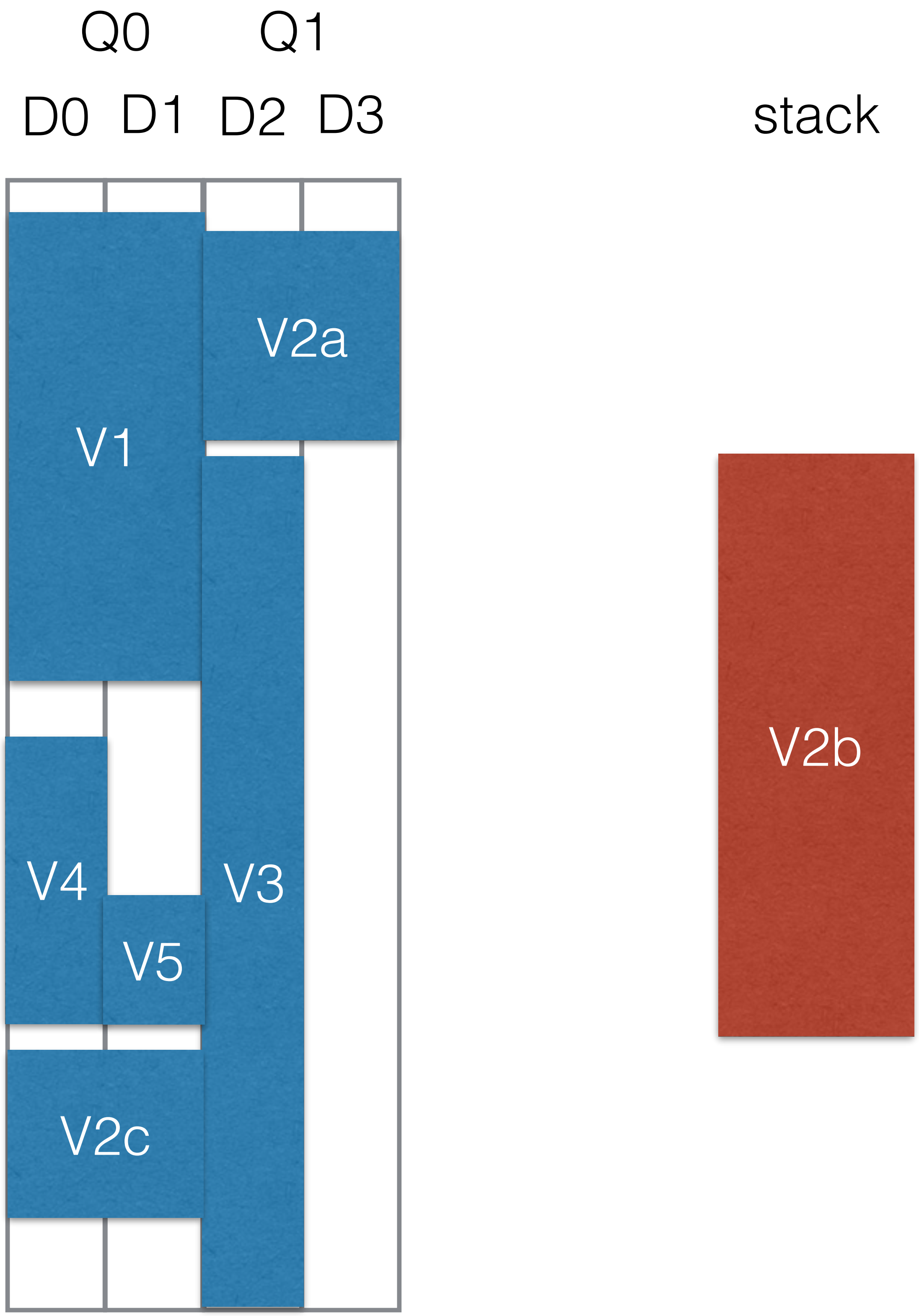
- Evict V2 (evict Live Interval with lower spill cost)



- Split V2



- Split V2

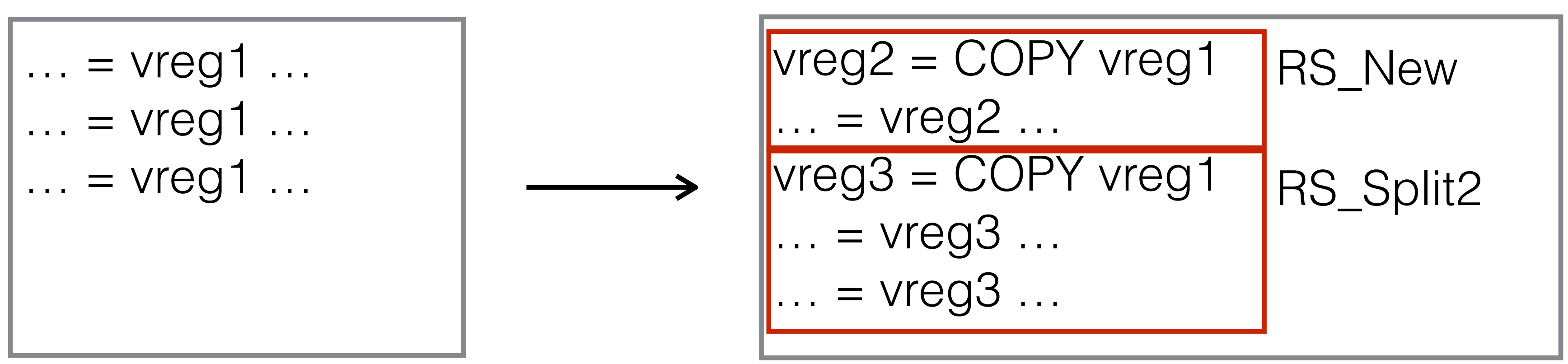


Greedy RA Stages

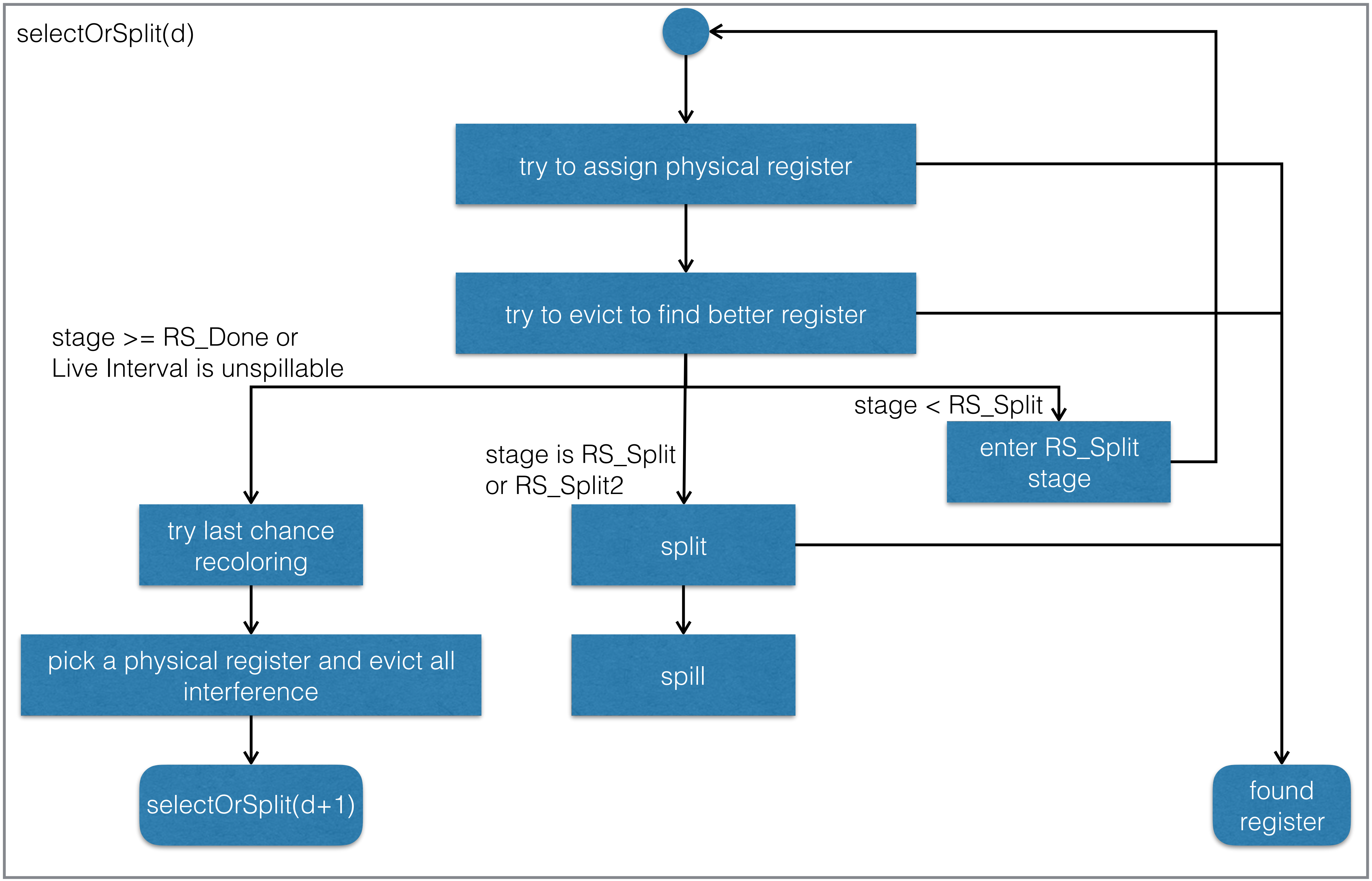
- RS_New: created
- RS_Assign: enqueue
- RS_Split: need to split
- RS_Split2
  - used for split products that may not be making progress
- RS_Spill: need to spill
- RS_Done: assigned a physical register or created by spill

RS_Split2

- The live intervals created by split will enqueue to process again.
    - There is a risk of creating infinite loops.



Greedy Register Allocation



Last Chance Recoloring

- Try to assign a physical register to Live Interval by evicting all its interferences.

  - The recoloring process may recursively use the last chance recoloring. Therefore, when a virtual register has been assigned a color by this mechanism, it is marked as Fixed.

```
vA can use {R1, R2    }
vB can use {    R2, R3}
vC can use {R1        }
```

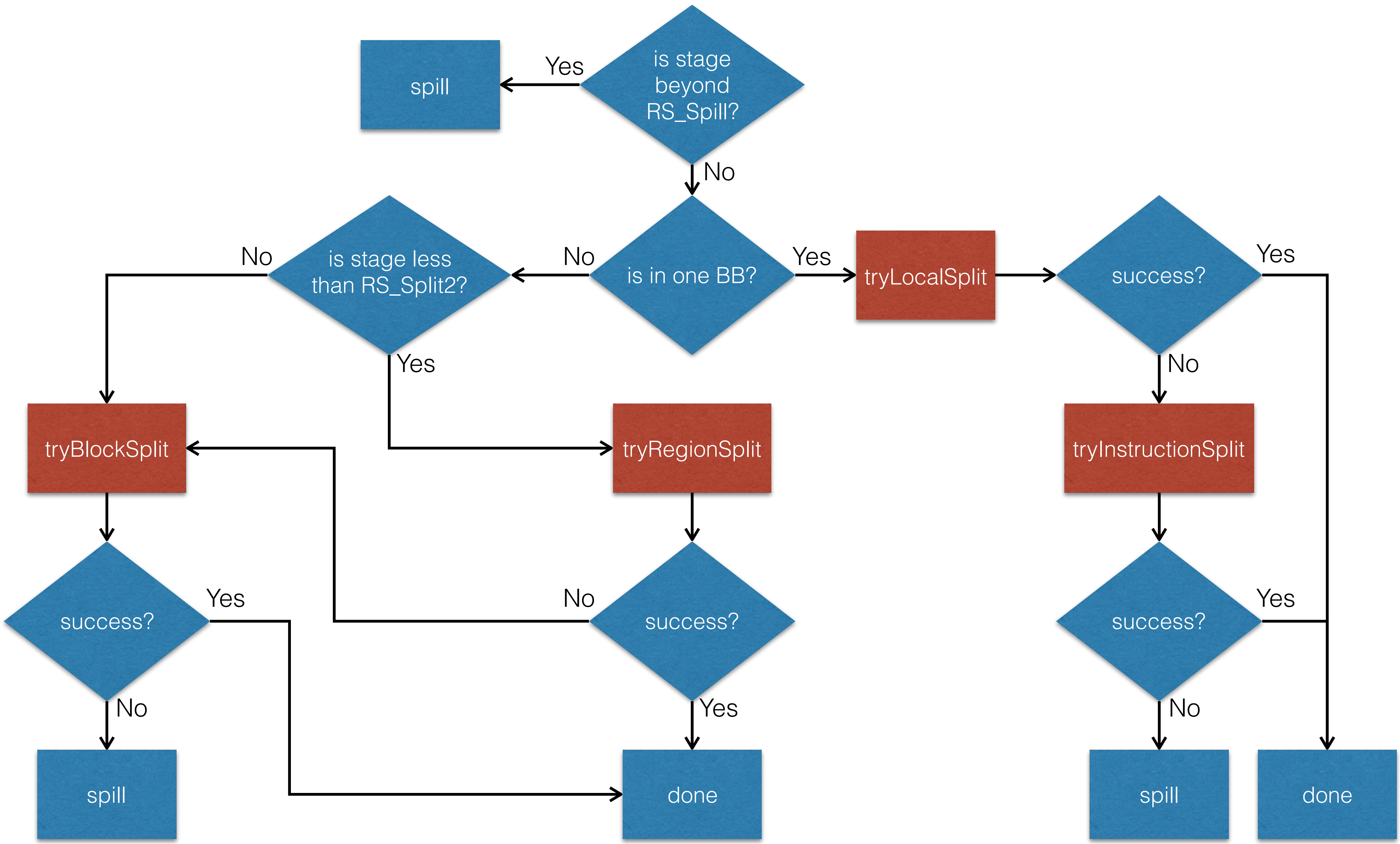
selectOrSplit(d)

vA => R1
vB => R2
vC => fails

selectOrSplit(d + 1)

vA => R2
vB => R3
vC => R1 (fixed)

How to Split?

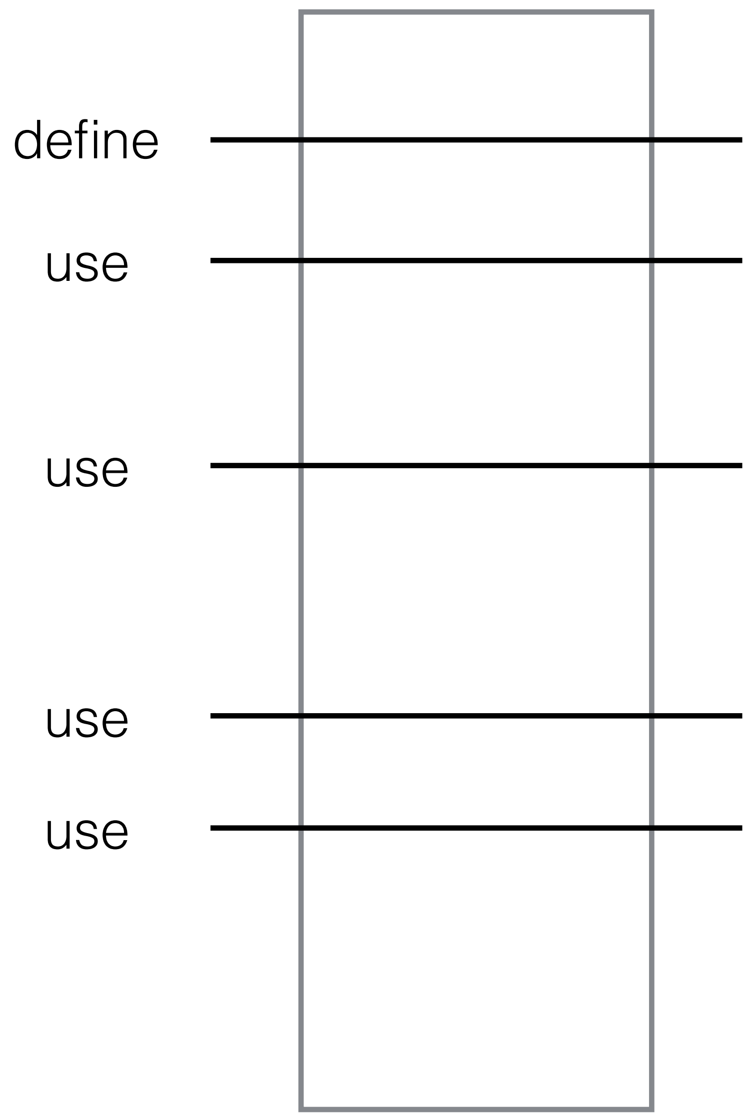


spill
Yes
is stage beyond RS_Spill?
No
No
is stage less than RS_Split2?
No
is in one BB?
Yes
tryLocalSplit
success?
Yes
Yes
No
tryBlockSplit
tryRegionSplit
tryInstructionSplit
success?
Yes
No
success?
Yes
success?
No
No
Yes
No
spill
done
spill
done

tryLocalSplit

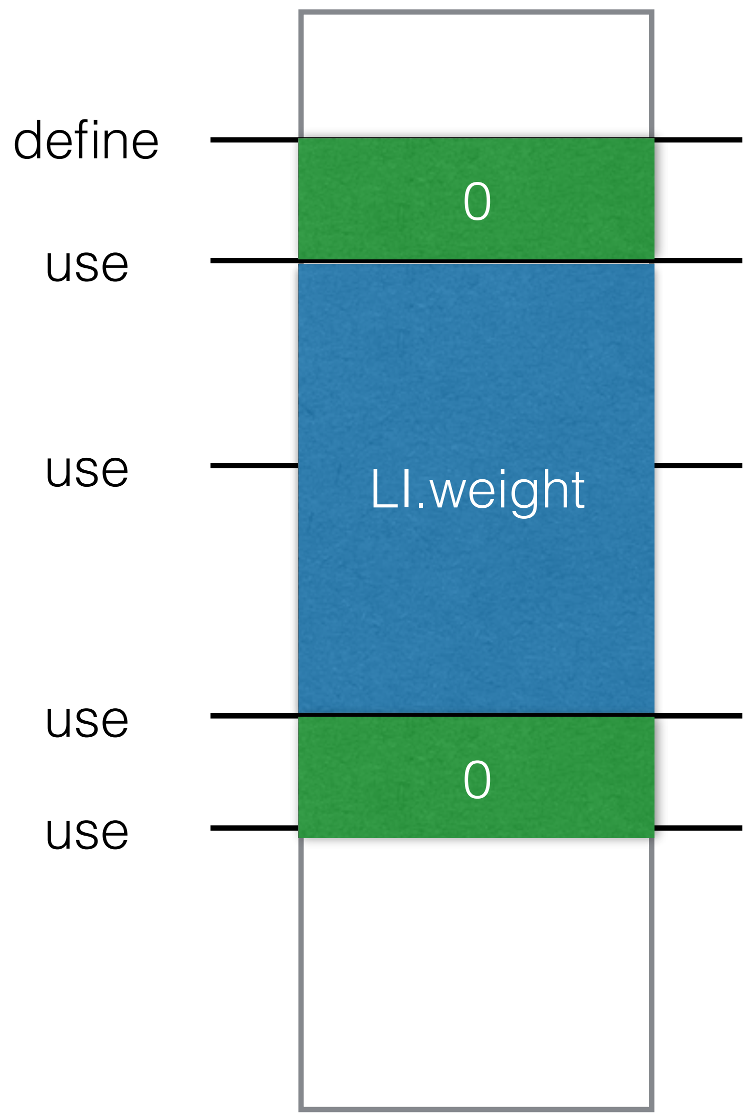
- Try to split virtual register interval into smaller intervals inside its only basic block.
  - calculate gap weights
  - adjust the split region

Calculate Gap Weights

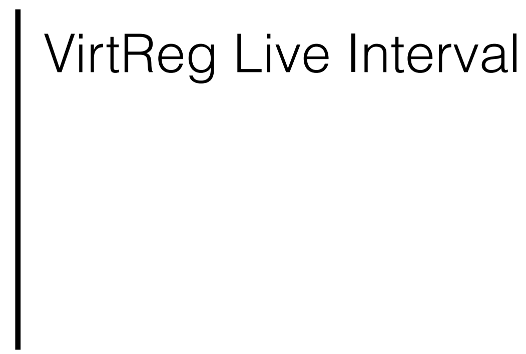


NumGaps = 4

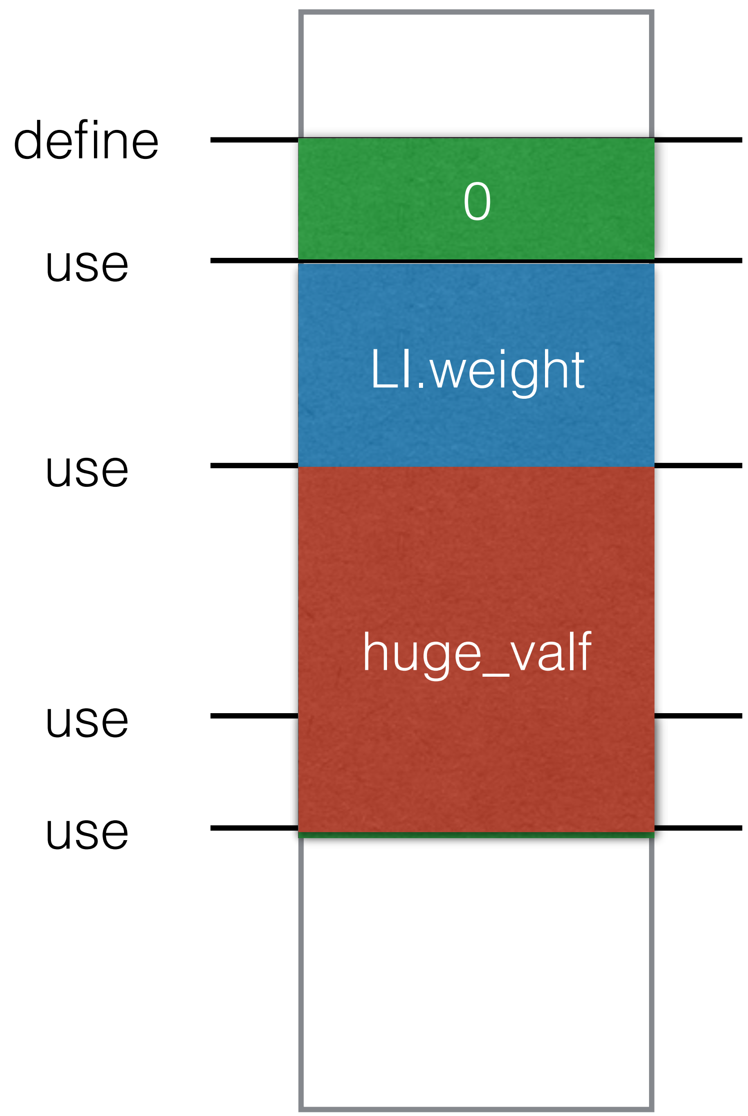
Calculate Gap Weights



If there is a physical register occupied by VirtReg.



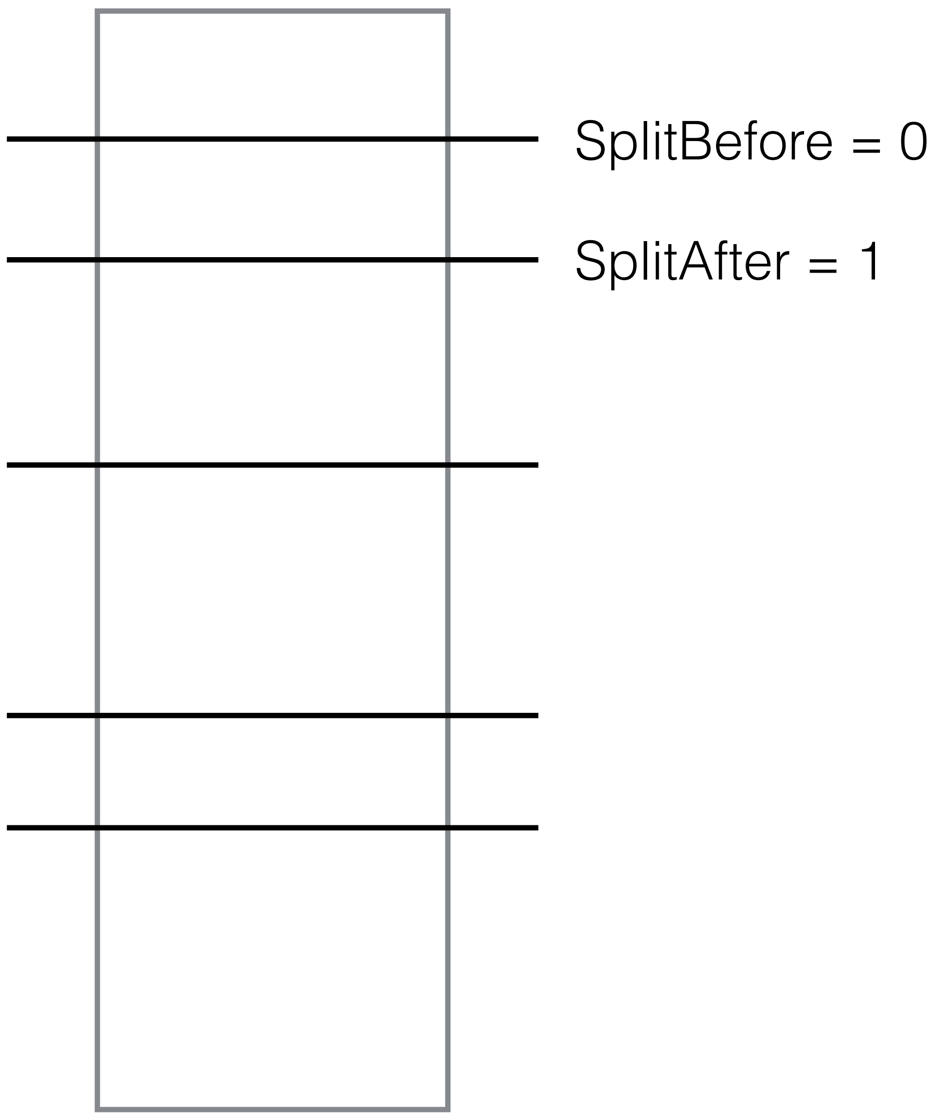
Calculate Gap Weights

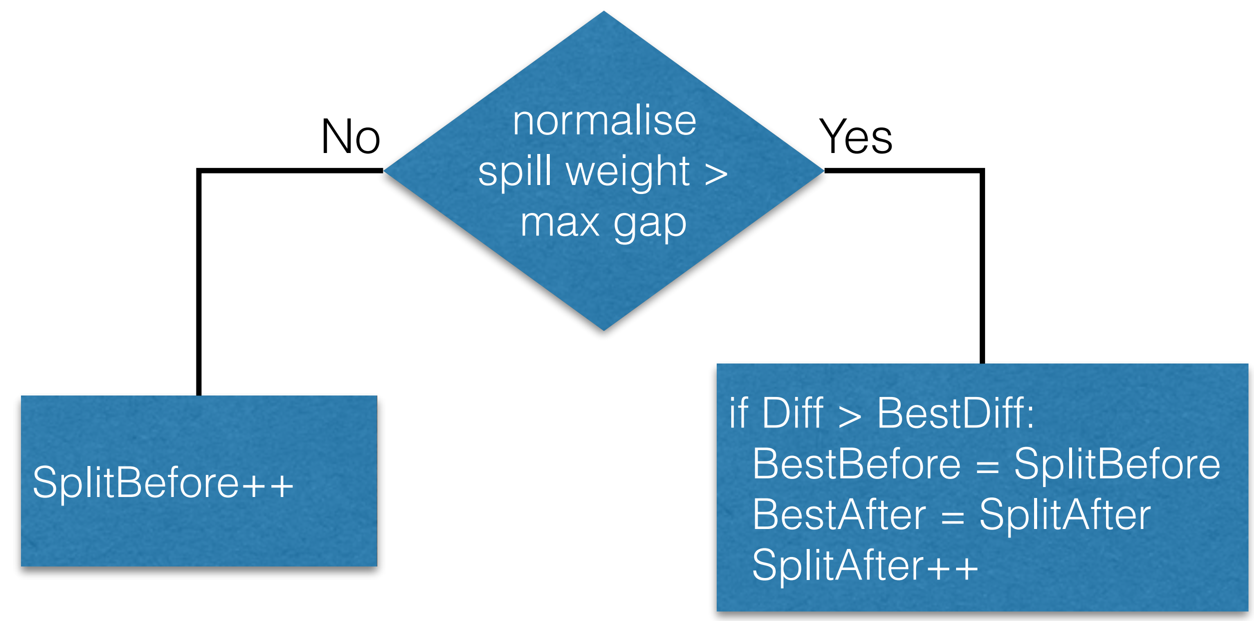


If there is a fixed physical register.

physical Live Interval

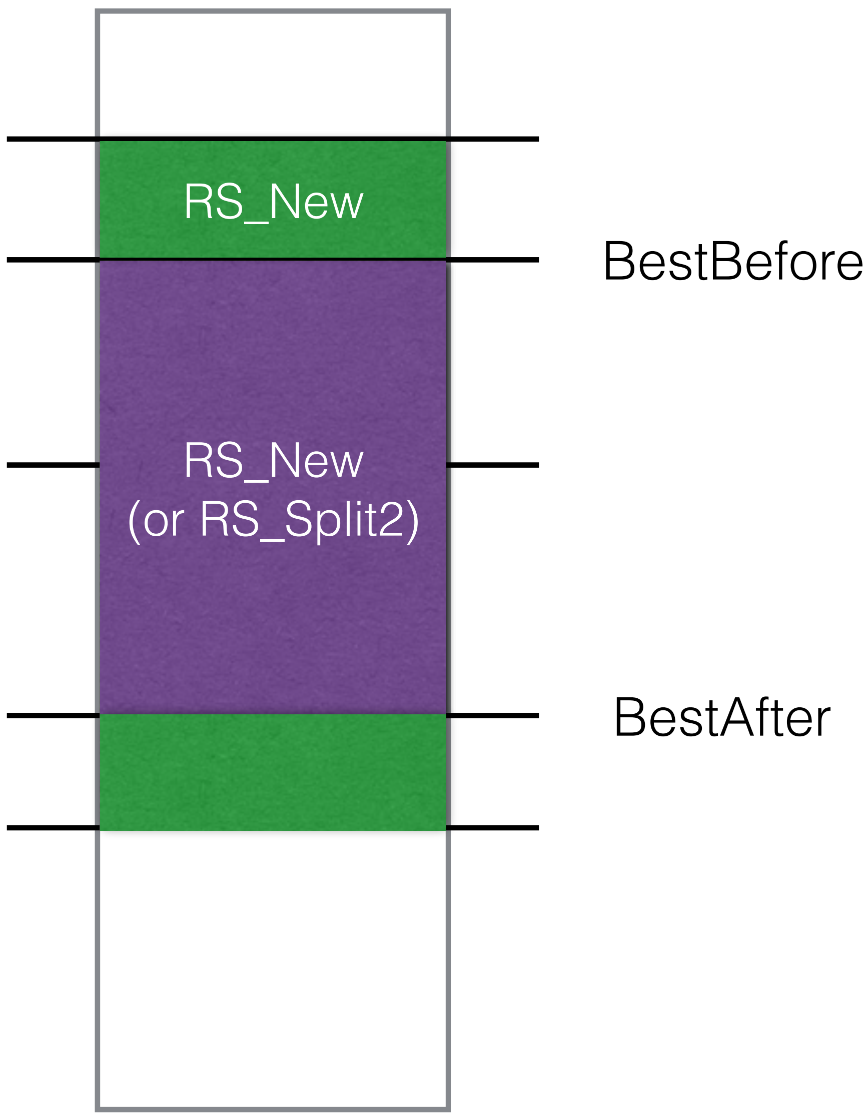
Adjust Split Region

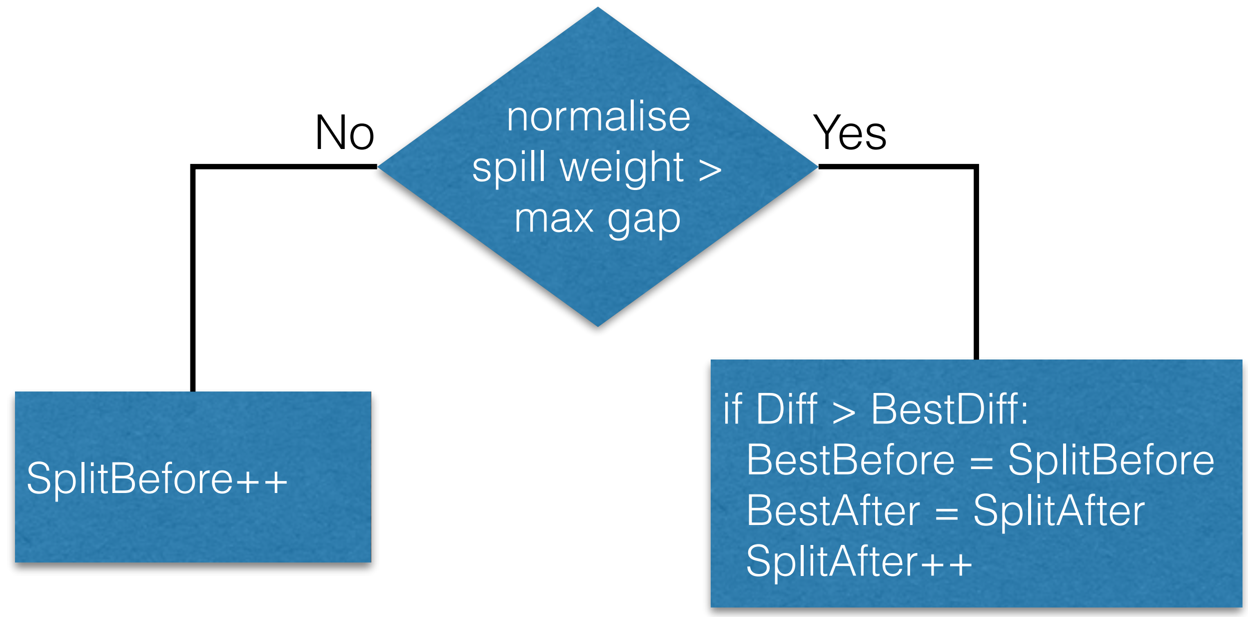




normalise spill weight = spill cost / distance
= (#gap * block_freq) / distance(SplitBefore, SplitAfter)

Adjust Split Region





normalise spill weight = spill cost / distance
= (#gap * block_freq) / distance(SplitBefore, SplitAfter)

Go through all physical registers.
Find the most critical range.

tryRegionSplit

- Use Hopfield Network to find optimal splits.
  - Guaranteed to converge to a local minimum.

Hopfield Network

$$a(t)_{s\times 1} = \begin{cases} p_{s\times 1} & : t = 0 \\ S(W_{s\times s} \times a(t-1)_{s\times 1} + b_{s\times 1}) & : t \geq 1 \end{cases}$$

$$S(x) = \begin{cases} +1 & : x \geq \theta \\ -1 & : x < \theta \end{cases}$$

tryRegionSplit

1. For every physical register, construct Hopfield Network
   - Initialize border constraints
   - Initialize Hopfield Network nodes according to border constraints
   - Add links to Hopfield Network and iterate
2. Get the best candidate
3. Do region split

Initialize Border Constraints

- No Interference.

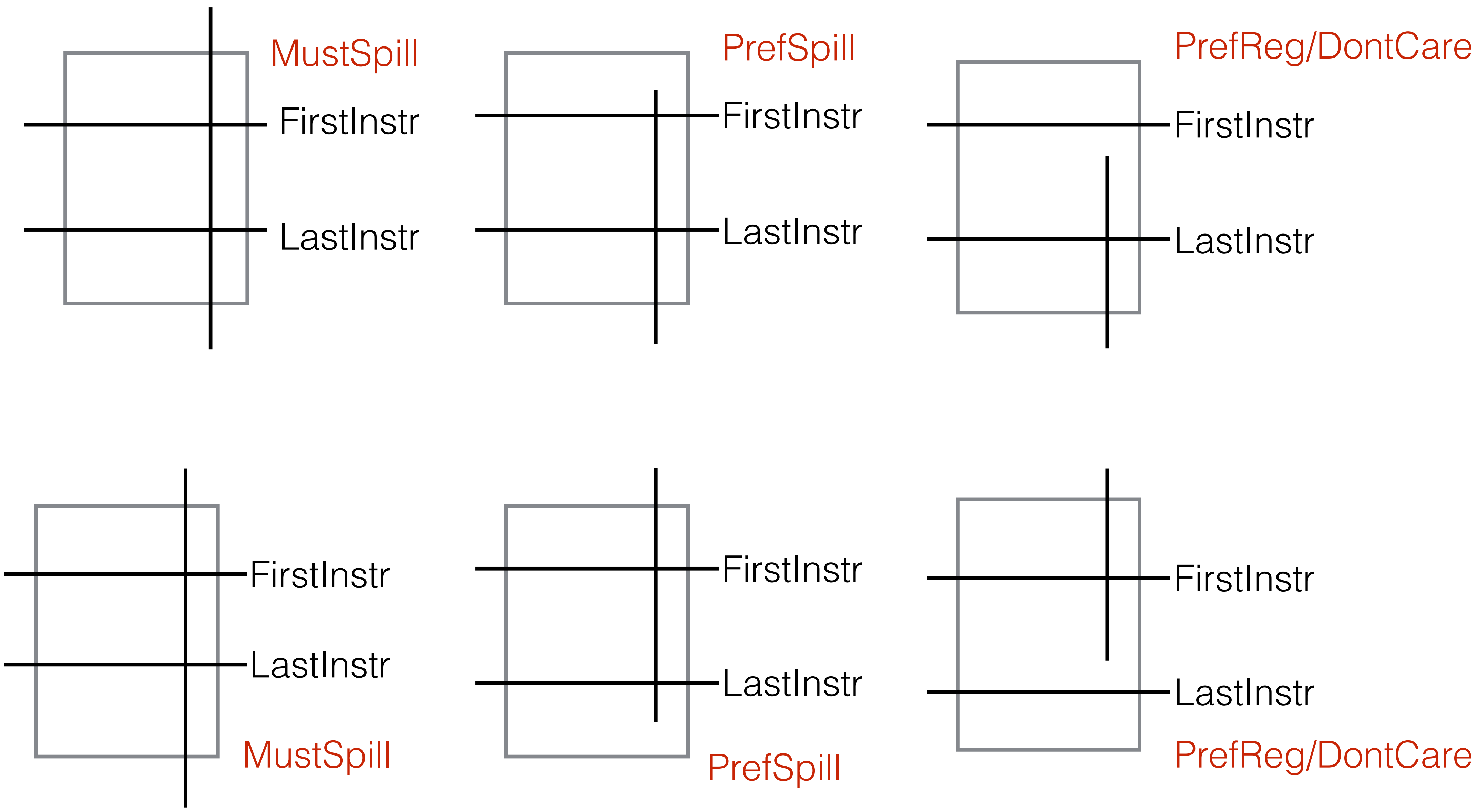
LiveIn ? PrefReg : DontCare;

LiveOut ? PrefReg : DontCare;

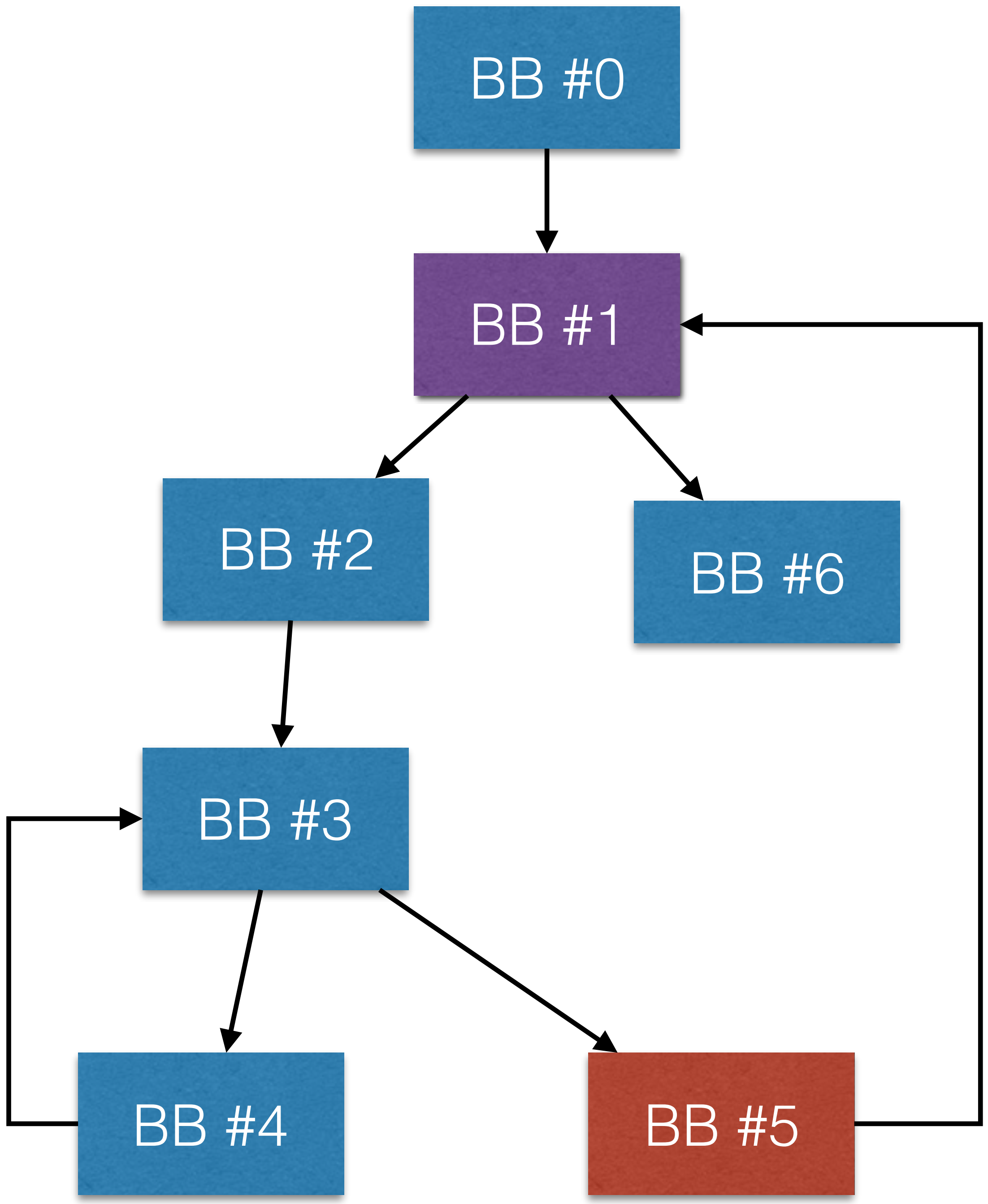
```
enum BorderConstraint {
  DontCare,
  PrefReg,
  PrefSpill,
  PrefBoth,
  MustSpill
};
```

Initialize Border Constraints

- There are Interferences.



Edge Bundle

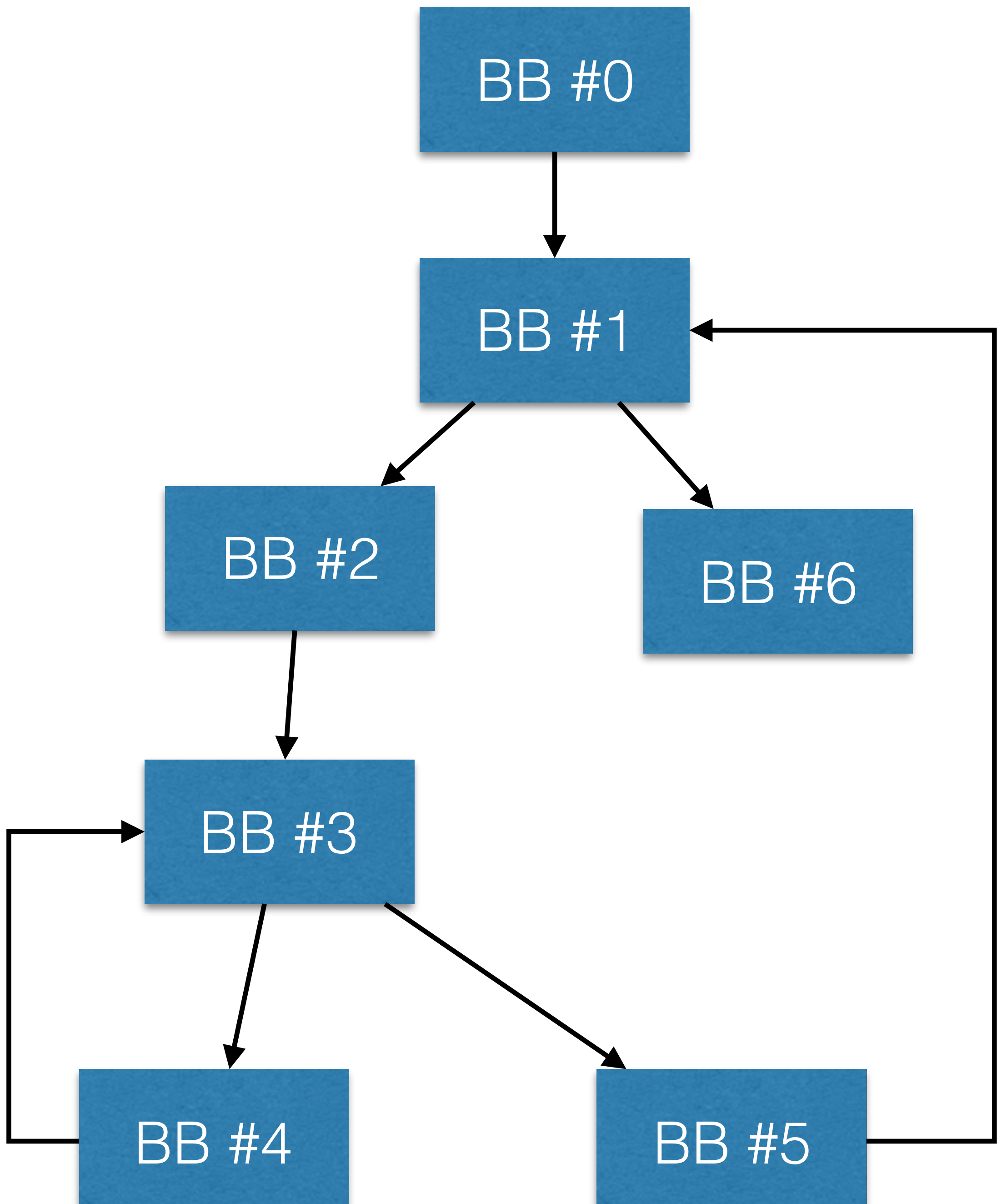



```
EC:
(BB#0, in)  Bundle #0:  0   0         0
(BB#0, out) Bundle #1:  1   1         1
(BB#1, in)  Bundle #2:  2   1         1
(BB#1, out) Bundle #3:  3   3         2
(BB#2, in)  Bundle #4:  4   3         2
(BB#2, out) Bundle #5:  5   5         3
(BB#3, in)  Bundle #6:  6   5         3
(BB#3, out) Bundle #7:  7   7         4
(BB#4, in)  Bundle #8:  8   7         4
(BB#4, out) Bundle #9:  9   5         3
(BB#5, in)  Bundle #10: 10  7         4
(BB#5, out) Bundle #11: 11  11 -> 1   1
(BB#6, in)  Bundle #12: 12  3         2
(BB#6, out) Bundle #13: 13  13        5
```

```
void join(unsigned a, unsigned b) {
  unsigned eca = EC[a];
  unsigned ecb = EC[b];
  while (eca != ecb)
    if (eca < ecb)
      EC[b] = eca, b = ecb, ecb = EC[b];
    else
      EC[a] = ecb, a = eca, eca = EC[a];
}
```

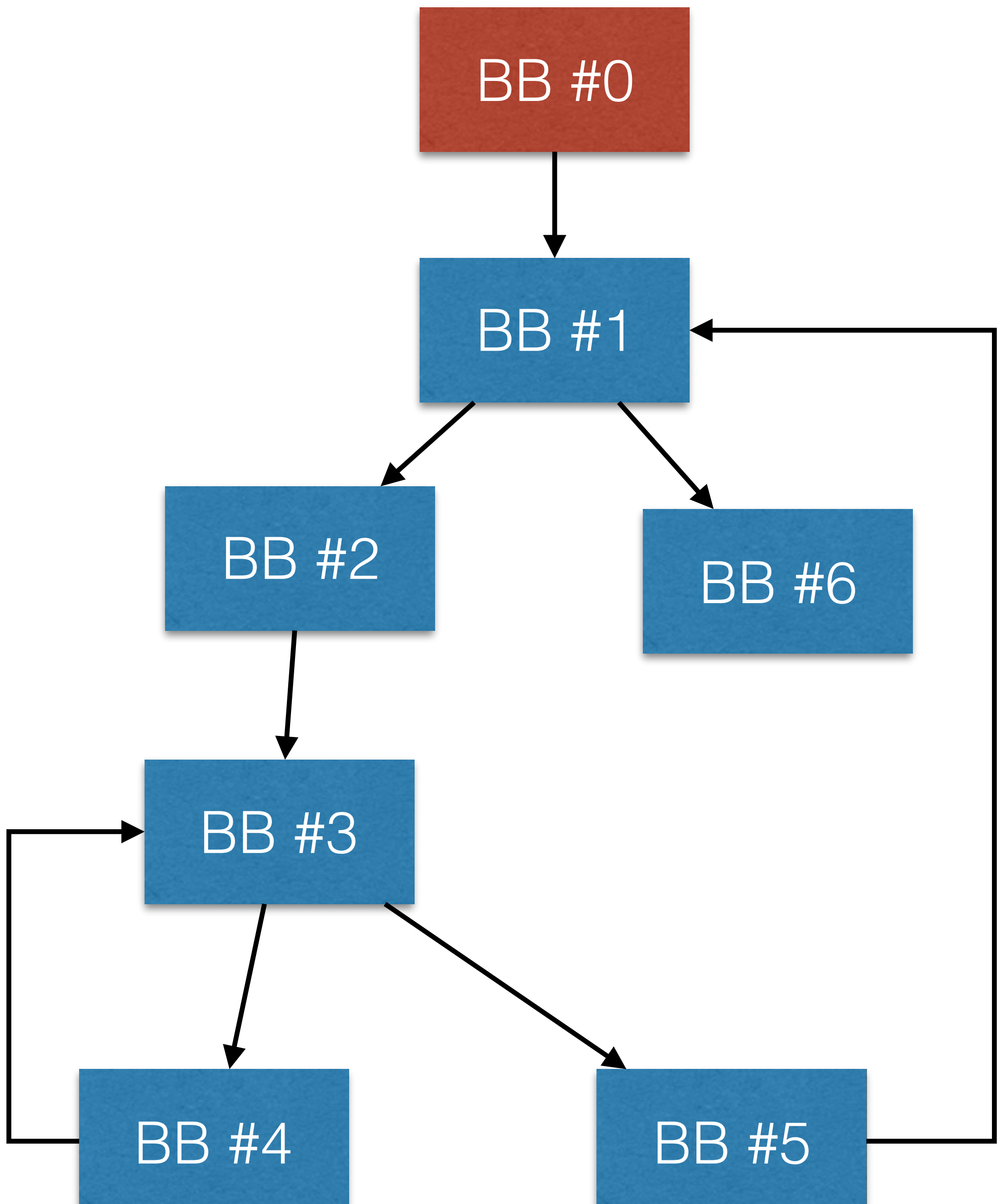
```
// Join the outgoing bundle with the ingoing bundles of all successors.
for (MachineBasicBlock::const_succ_iterator SI = MBB.succ_begin(),
       SE = MBB.succ_end(); SI != SE; ++SI)
  EC.join(OutE, 2 * (*SI)->getNumber());
```

Edge Bundle



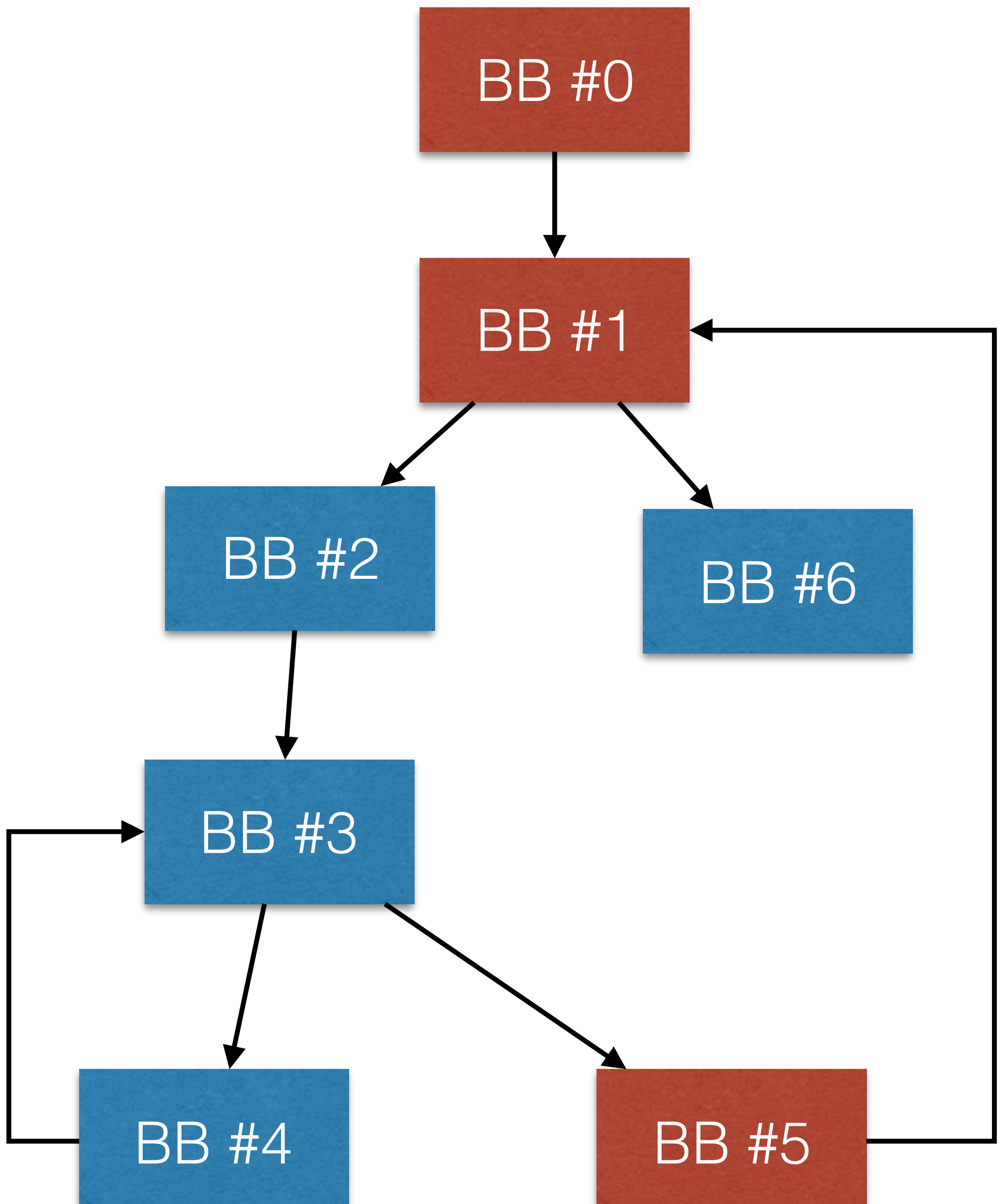

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5

Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

Edge Bundle



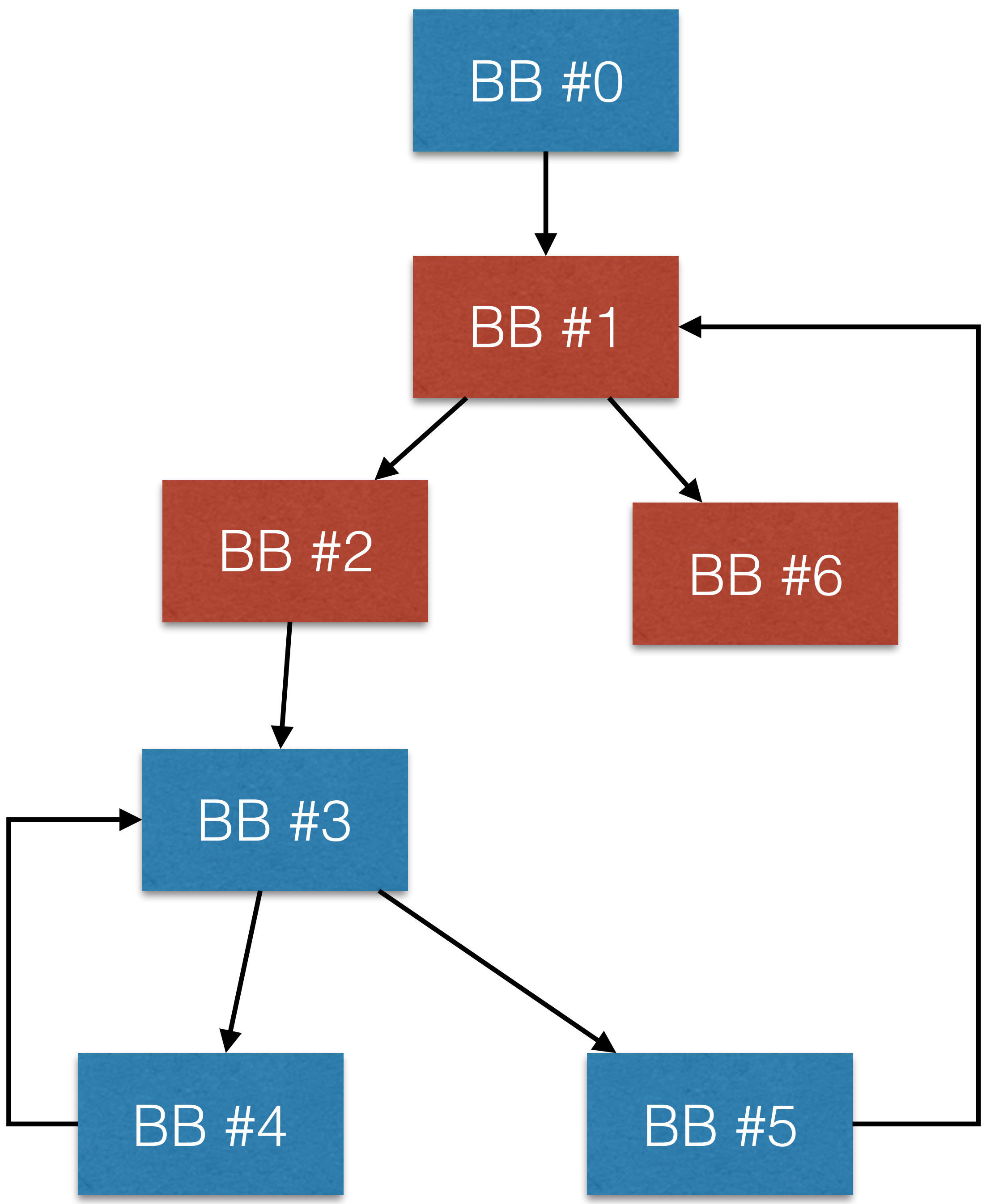

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5

Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

Edge Bundle




```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5

Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

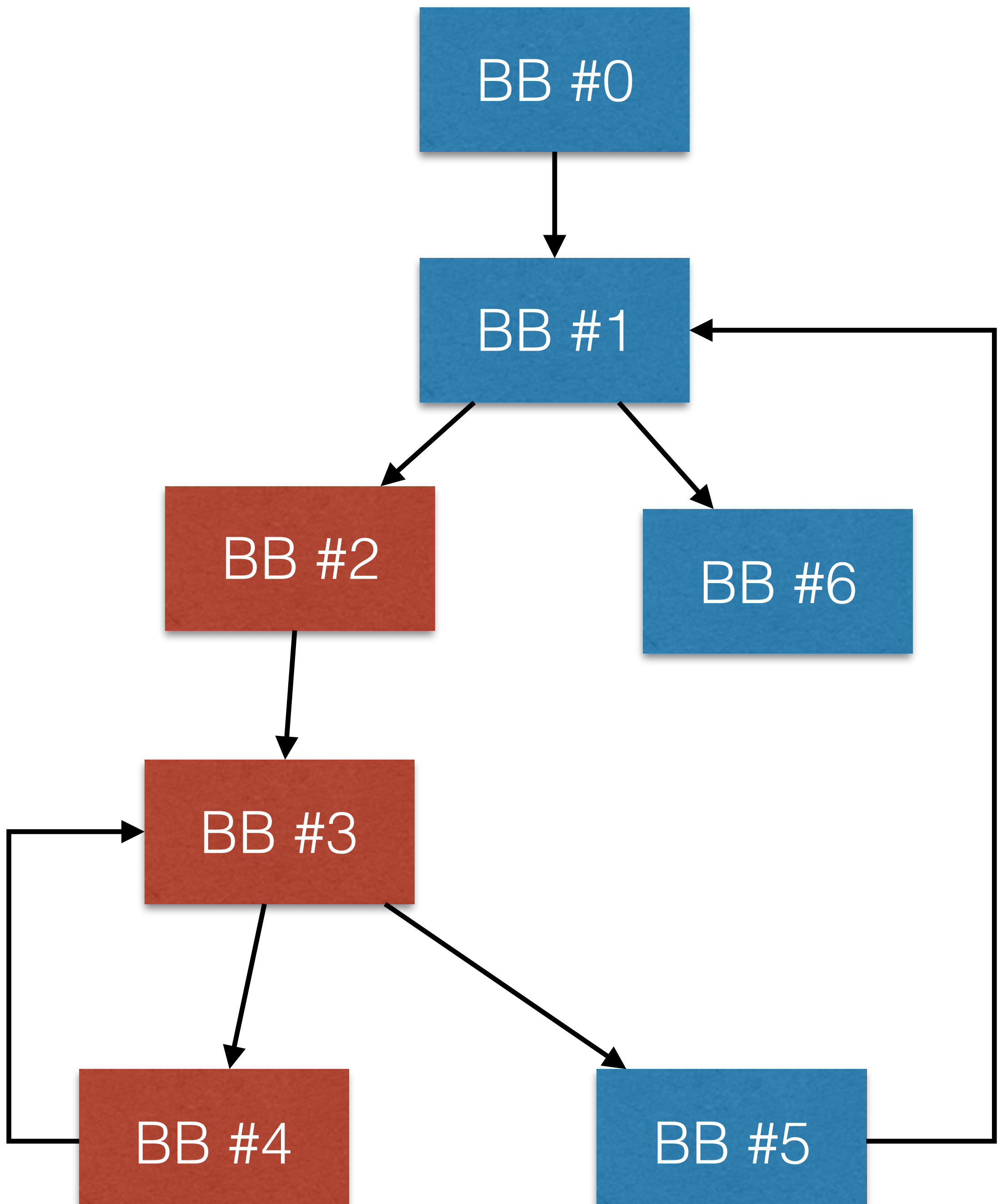
Edge Bundle




```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5
```

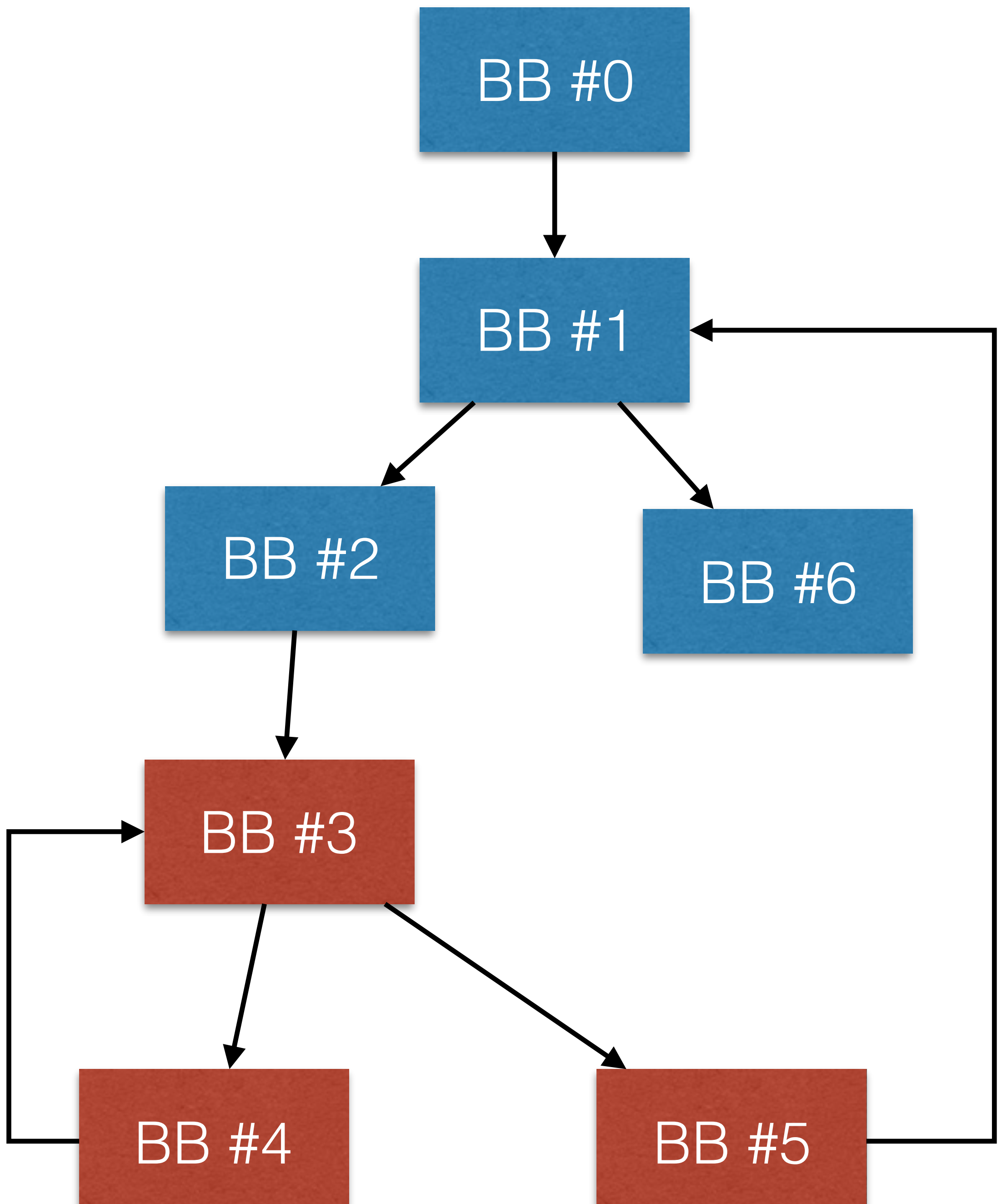
```
Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

Edge Bundle



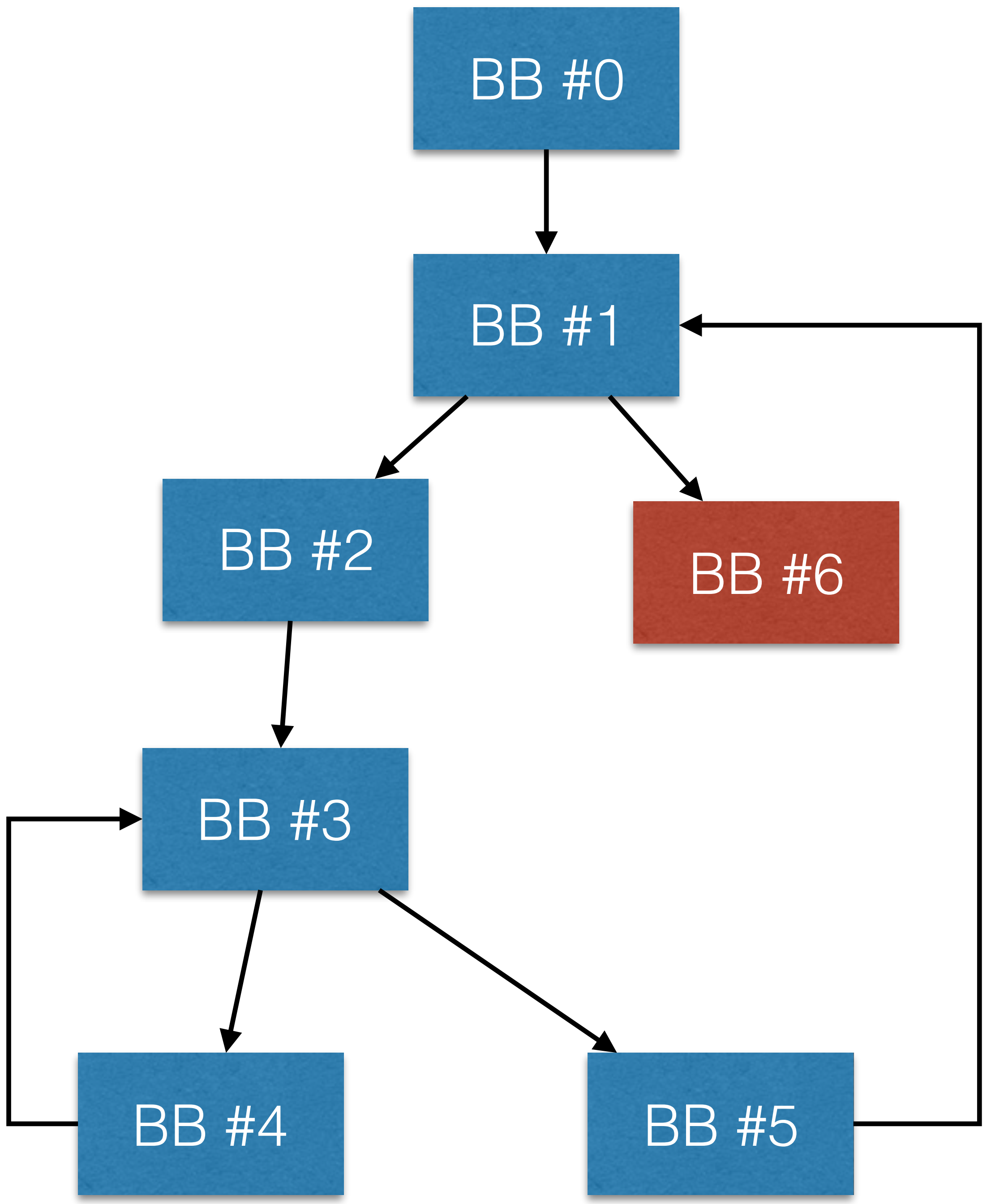

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5

Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

Edge Bundle




```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5

Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

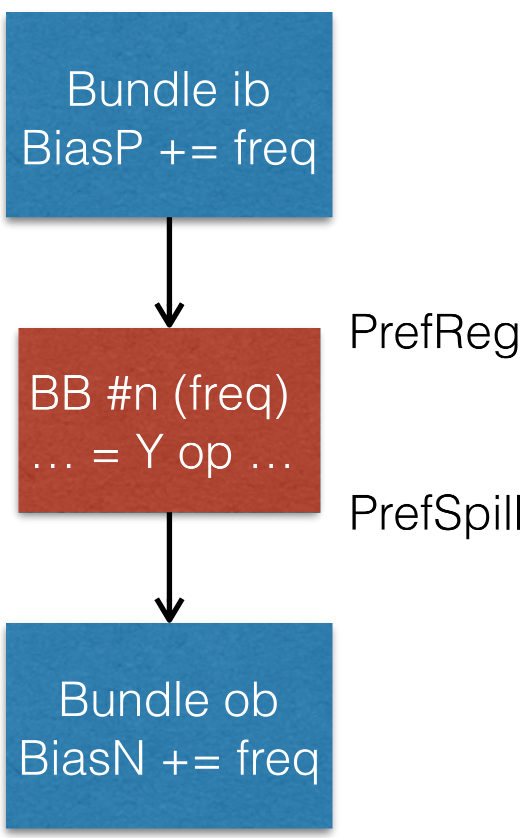
Edge Bundle




```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8:  8   7   4
(BB#4, out) Bundle #9:  9   5   3
(BB#5, in)  Bundle #10: 10  7   4
(BB#5, out) Bundle #11: 11  1   1
(BB#6, in)  Bundle #12: 12  3   2
(BB#6, out) Bundle #13: 13  13  5

Blocks:
Bundle #0: BB#0
Bundle #1: BB#0, BB#1, BB#5
Bundle #2: BB#1, BB#2, BB#6
Bundle #3: BB#2, BB#3, BB#4
Bundle #4: BB#3, BB#4, BB#5
Bundle #5: BB#6
Bundle #6:
Bundle #7:
Bundle #8:
Bundle #9:
Bundle #10:
Bundle #11:
Bundle #12:
Bundle #13:
```

Initialize Hopfield Network Node

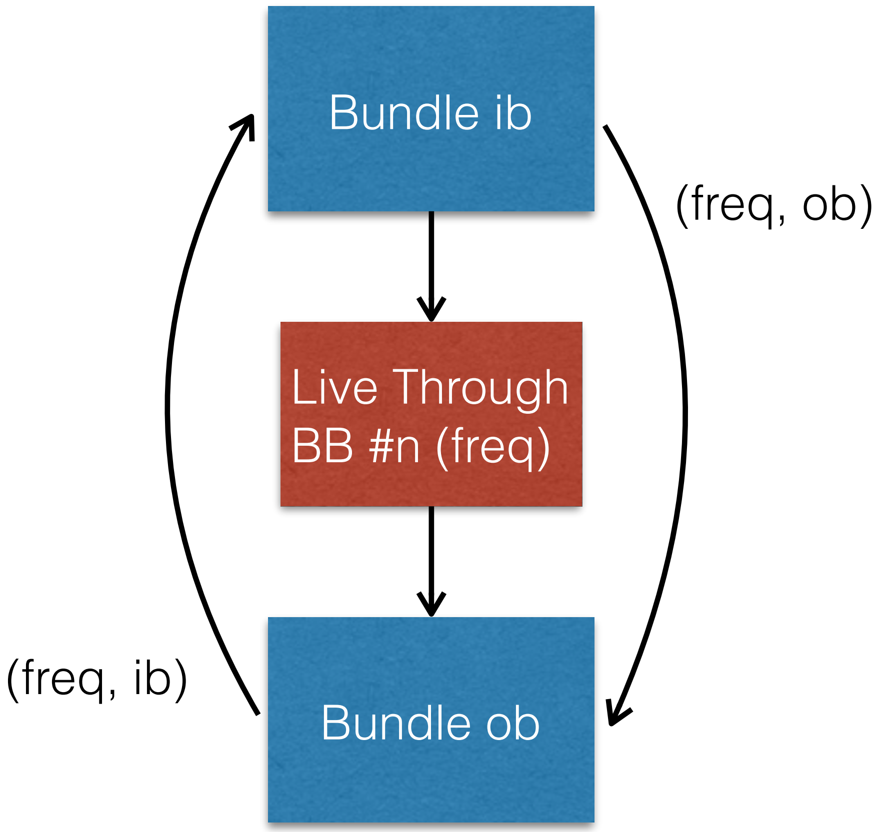
- update BiasN, BiasP according to BorderConstraint




```
void addBias(BlockFrequency freq, BorderConstraint direction) {
  switch (direction) {
  default:
    break;
  case PrefReg:
    BiasP += freq;
    break;
  case PrefSpill:
    BiasN += freq;
    break;
  case MustSpill:
    BiasN = BlockFrequency::getMaxFrequency();  // (uint64_t)-1ULL
    break;
  }
}
```

Add Links to Hopfield Network

- add weight to links

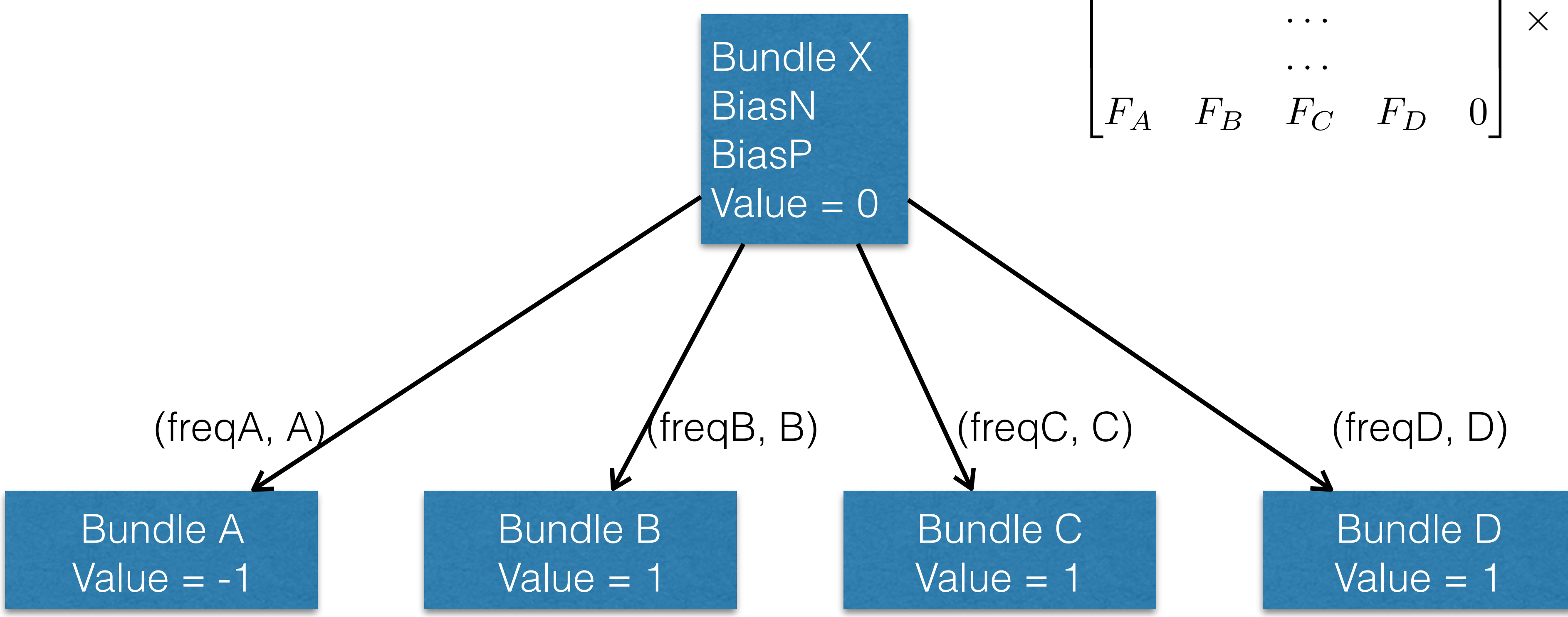



```
void addLink(unsigned b, BlockFrequency w) {
  // Update cached sum.
  SumLinkWeights += w;

  // There can be multiple links to the same bundle, add them up.
  for (LinkVector::iterator I = Links.begin(), E = Links.end(); I !=
    if (I->second == b) {
      I->first += w;
      return;
    }
  // This must be the first link to b.
  Links.push_back(std::make_pair(w, b));
}
```

Update Hopfield Network

$$a(t)_{s\times1} = \begin{cases} p_{s\times1} & : t = 0 \\ S(W_{s\times s} \times a(t-1)_{s\times1} + b_{s\times1}) & : t \geq 1 \end{cases}$$

$$\begin{bmatrix} & & \cdots & & \\ & & \cdots & & \\ & & \cdots & & \\ & & \cdots & & \\ F_A & F_B & F_C & F_D & 0 \end{bmatrix} \times \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} \vdots \\ Bias_p - Bias_n \end{bmatrix}$$



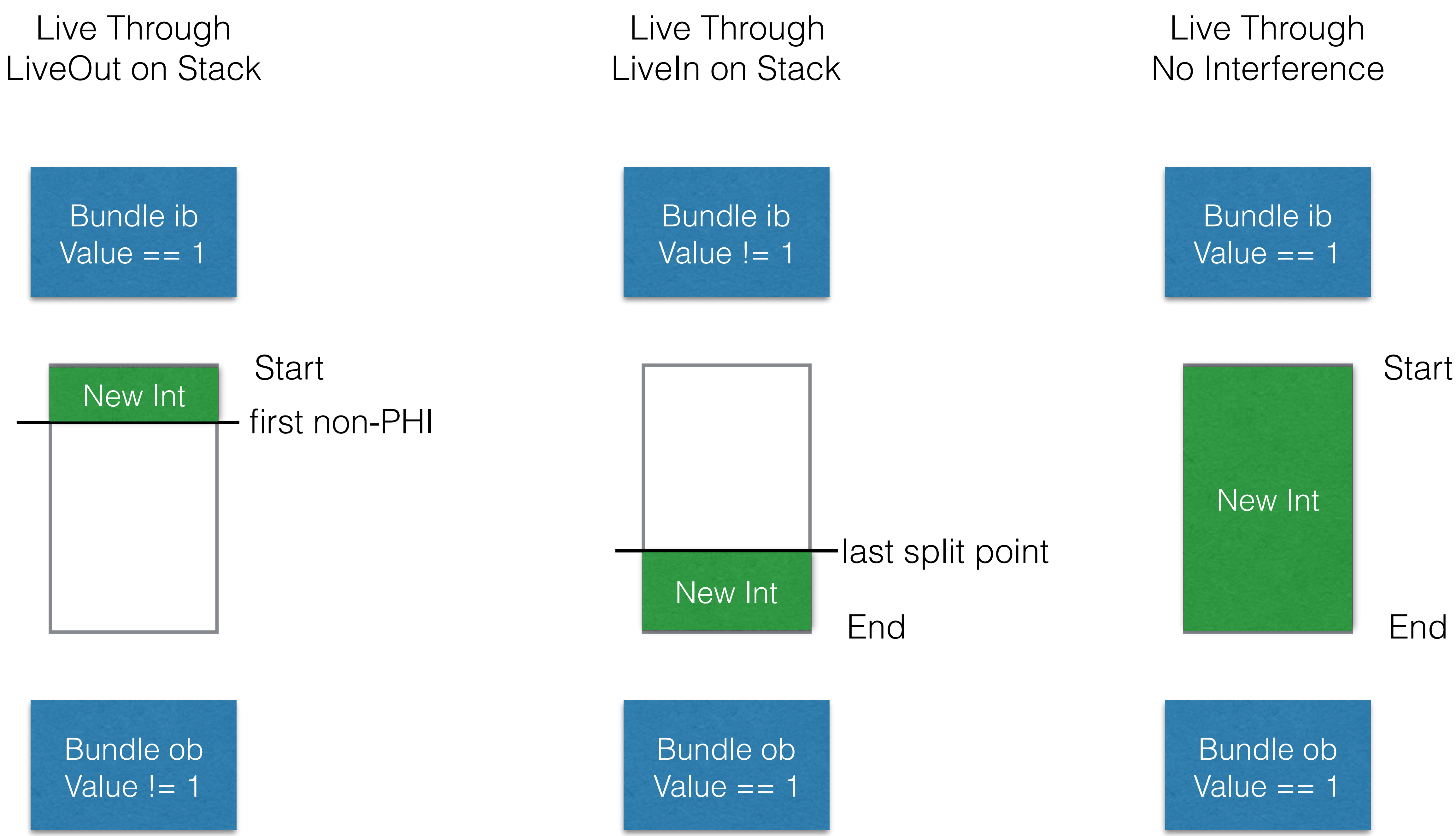

```
SumN = BiasN + freqA
SunP = BiasP + freqB + freqC + freqD
```

```
if (SumN >= SumP + Threshold)
  Value = -1;
else if (SumP >= SumN + Threshold)
  Value = 1;
else
  Value = 0;
```

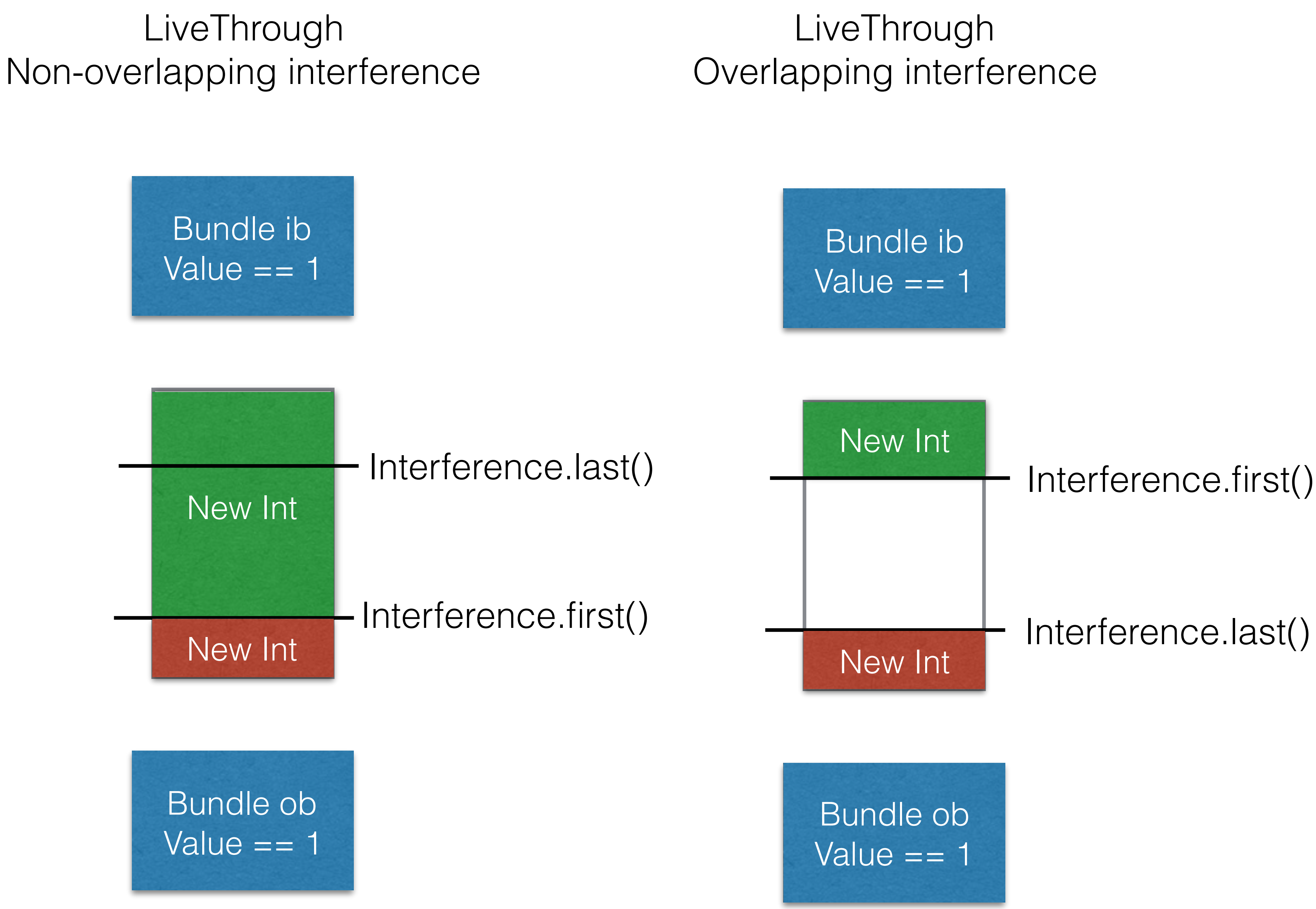
Region Split

- splitLiveThroughBlock
- splitRegInBlock
- splitRegOutBlock

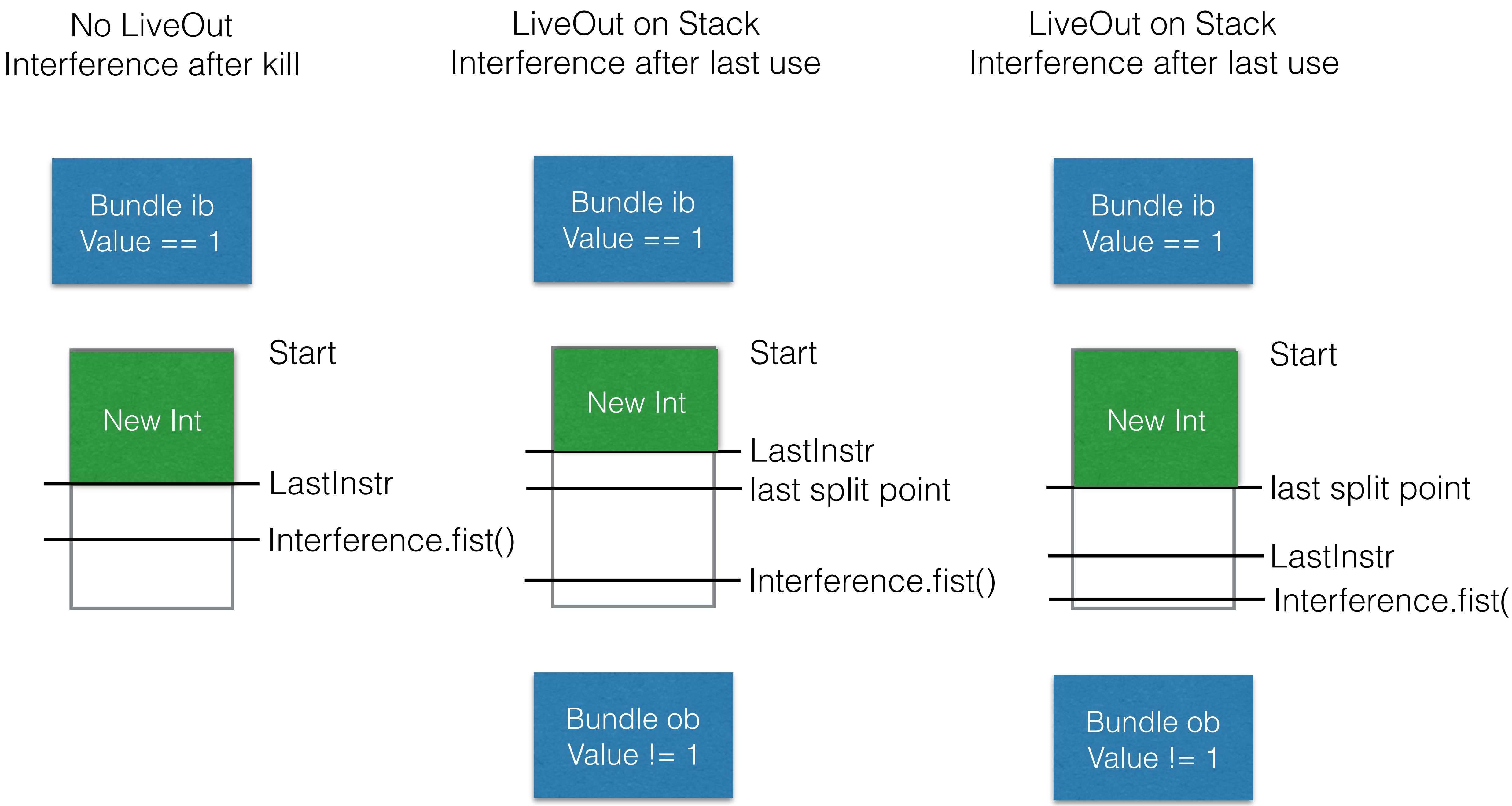
splitLiveThroughBlock



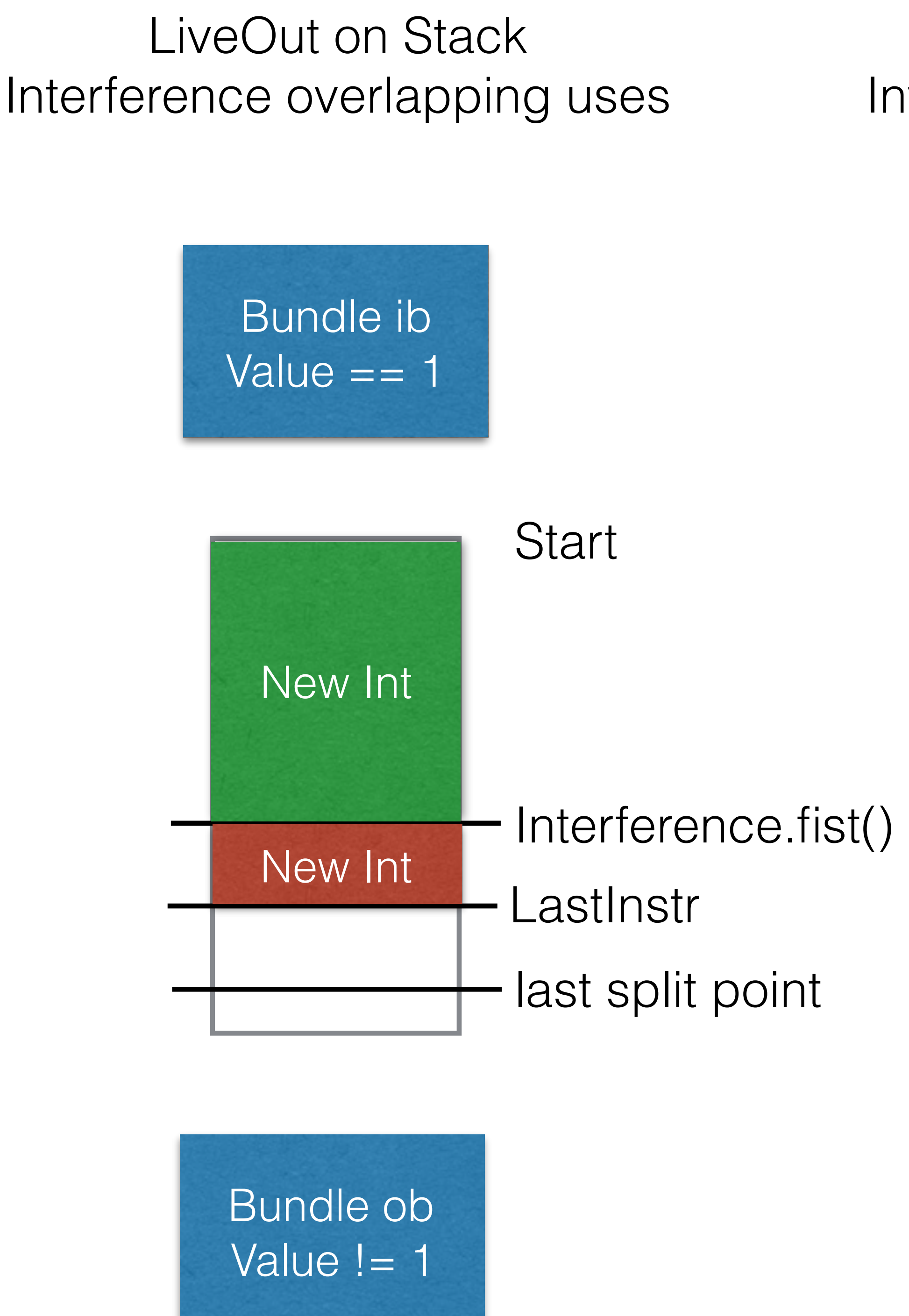
splitLiveThroughBlock

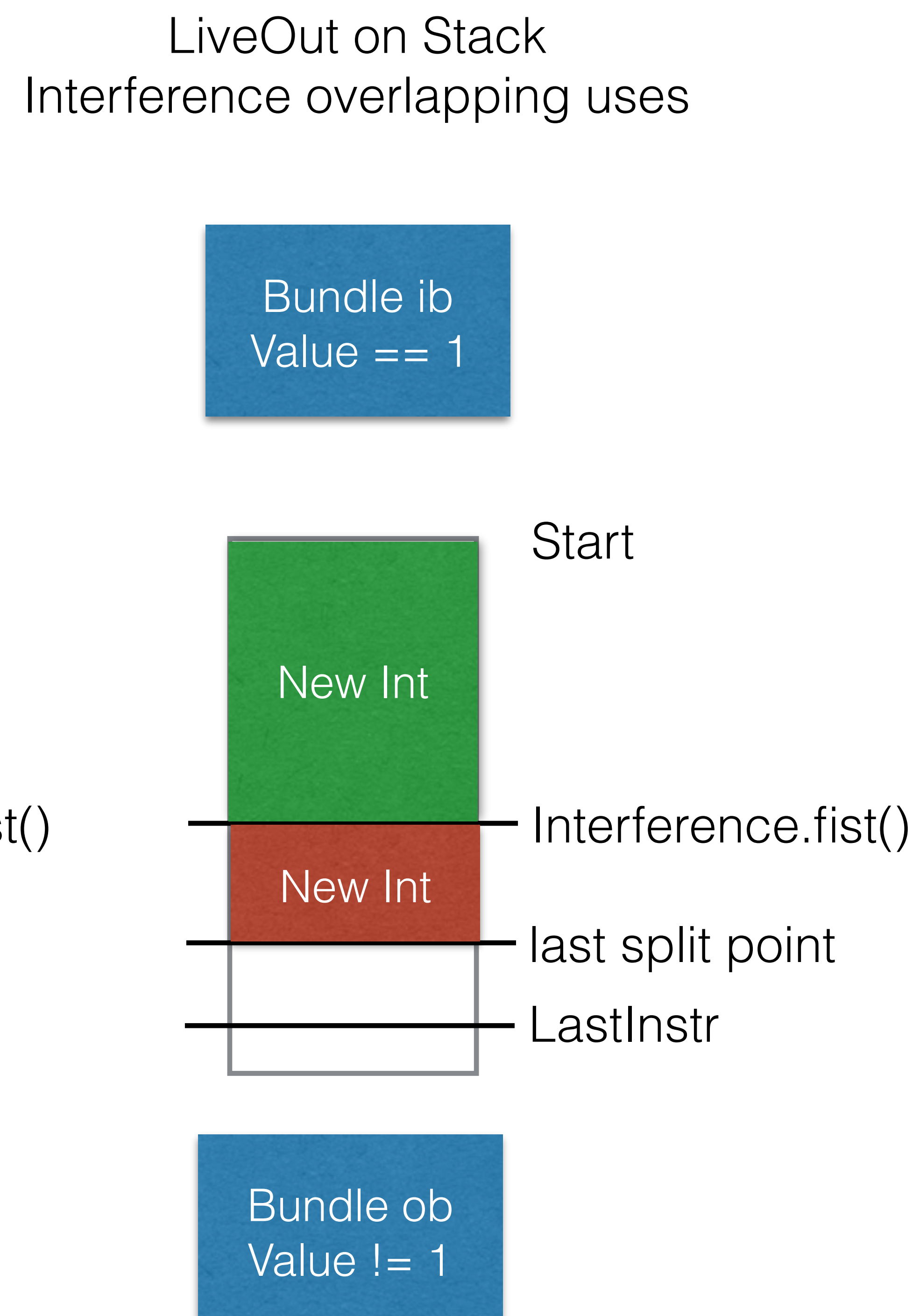


splitRegInBlock



splitRegInBlock





splitRegOutBlock