LLVM Register Allocation

Kai
kai@skymizer.com

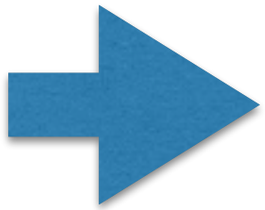
Outline

- Introduction to Register Allocation Problem
- LLVM Base Register Allocation Interface
- 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

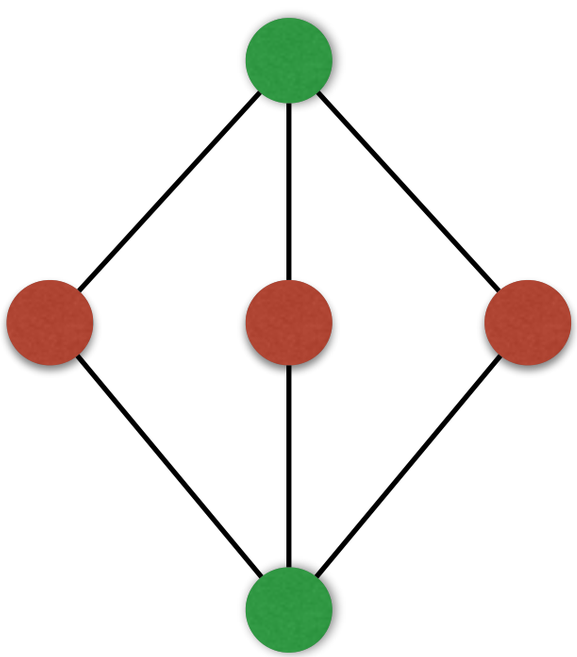
```c
int main()
{
    int i, j;
    int answer;

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

    return 0;
}
```



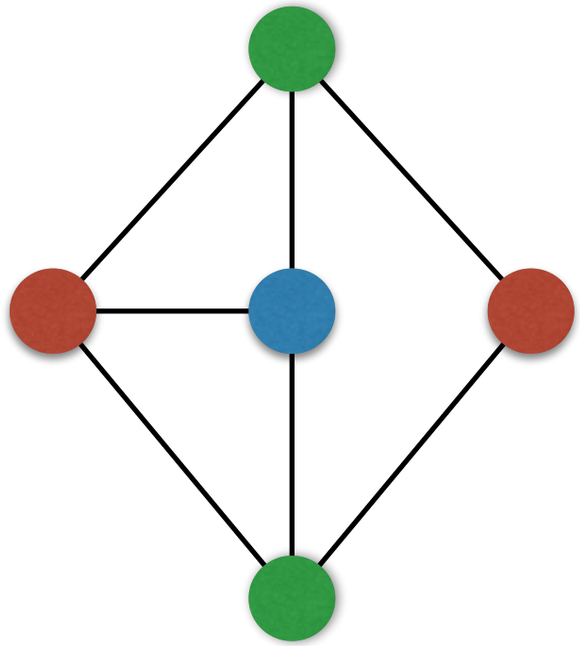
```
_main:
@ BB#0:
    sub  sp, #16
    movs r0, #0
    str  r0, [sp, #12]
    movs r0, #1
    str  r0, [sp, #8]
    b    LBB0_2
LBB0_1:

    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.



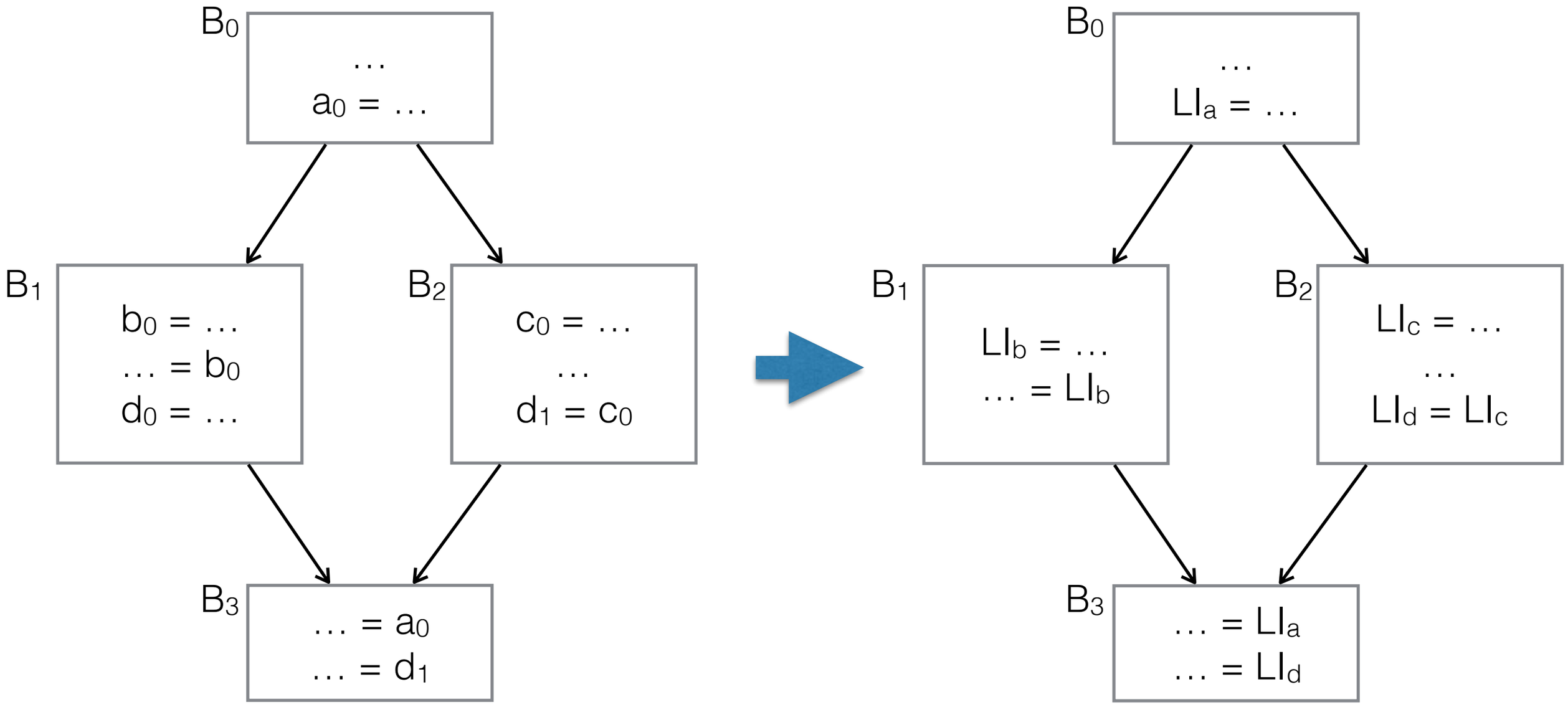
2-colorable

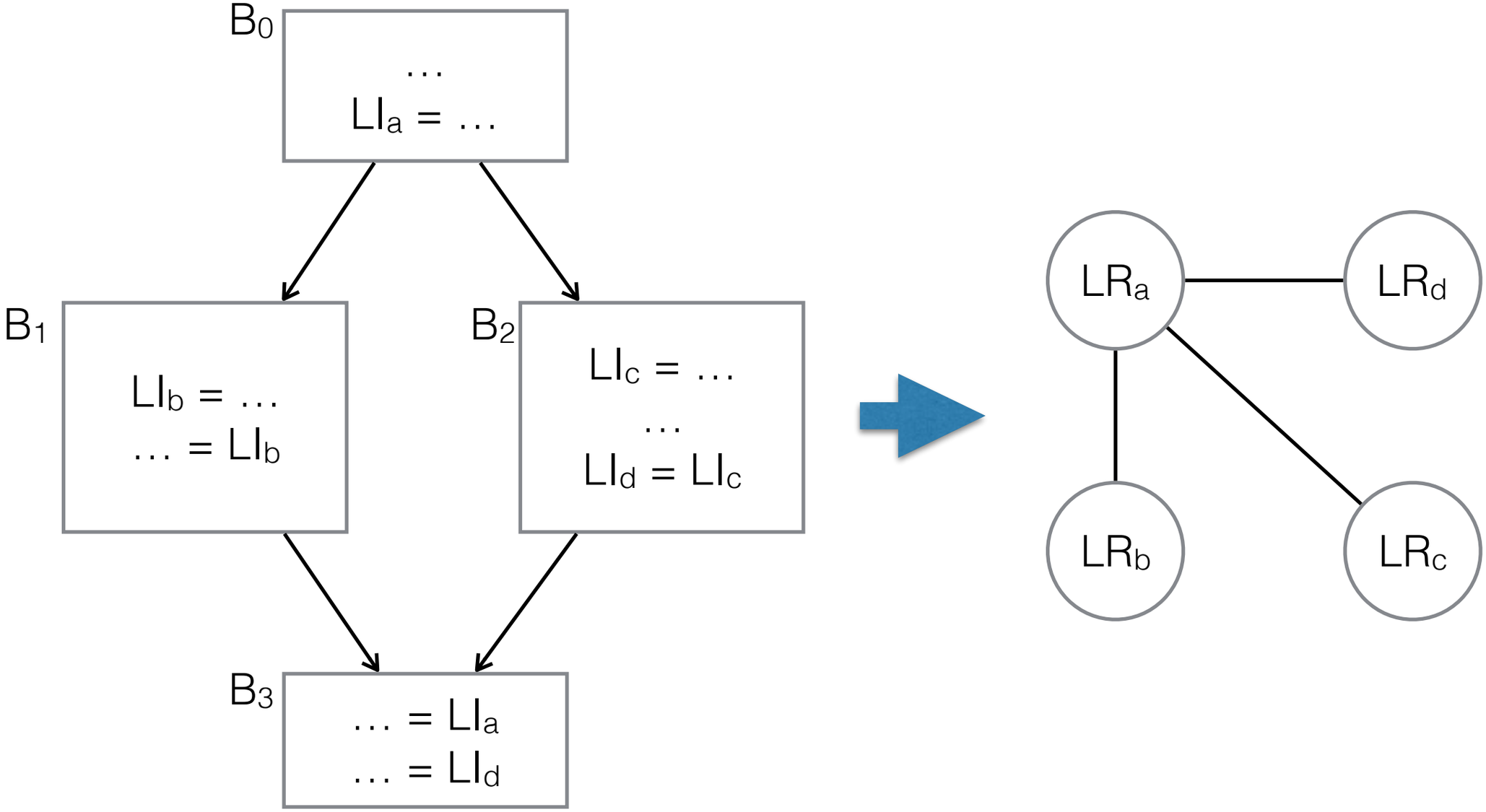


3-colorable

Graph Coloring for RA

- Node: Live interval
- Edge: Two live intervals have interference
- Color: Physical register
- Find a feasible 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

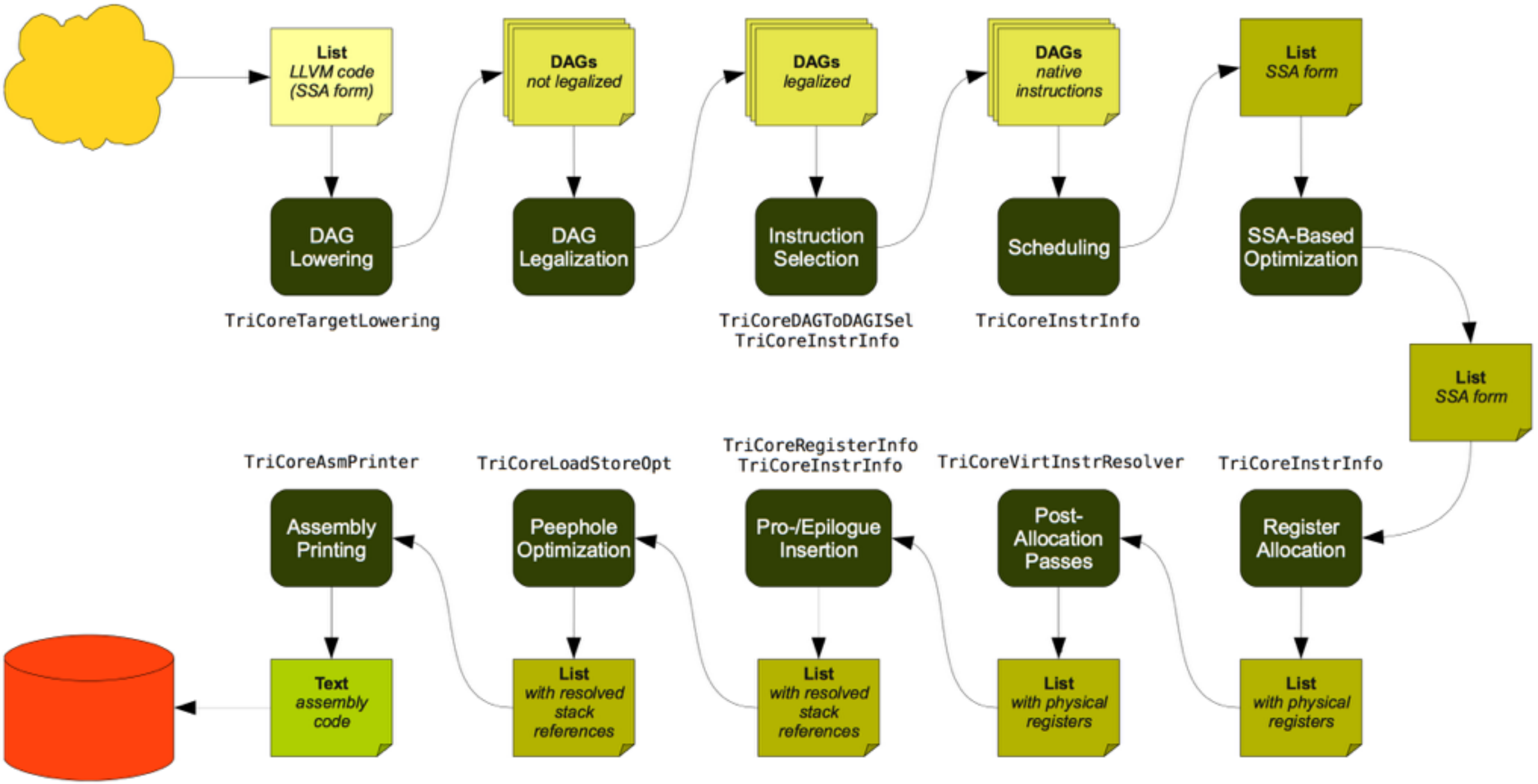


An Example from “Engineering A Compiler”

Why Not Graph Coloring

- Interference graph is expensive to build
- Spill code placement is more important than colouring
- Need to model aliases and overlapping register classes
- Flexibility is more important than the coloring algorithm

(Adopted from “Register Allocation in LLVM 3.0”)



Excerpt from tricore_llvm.pdf

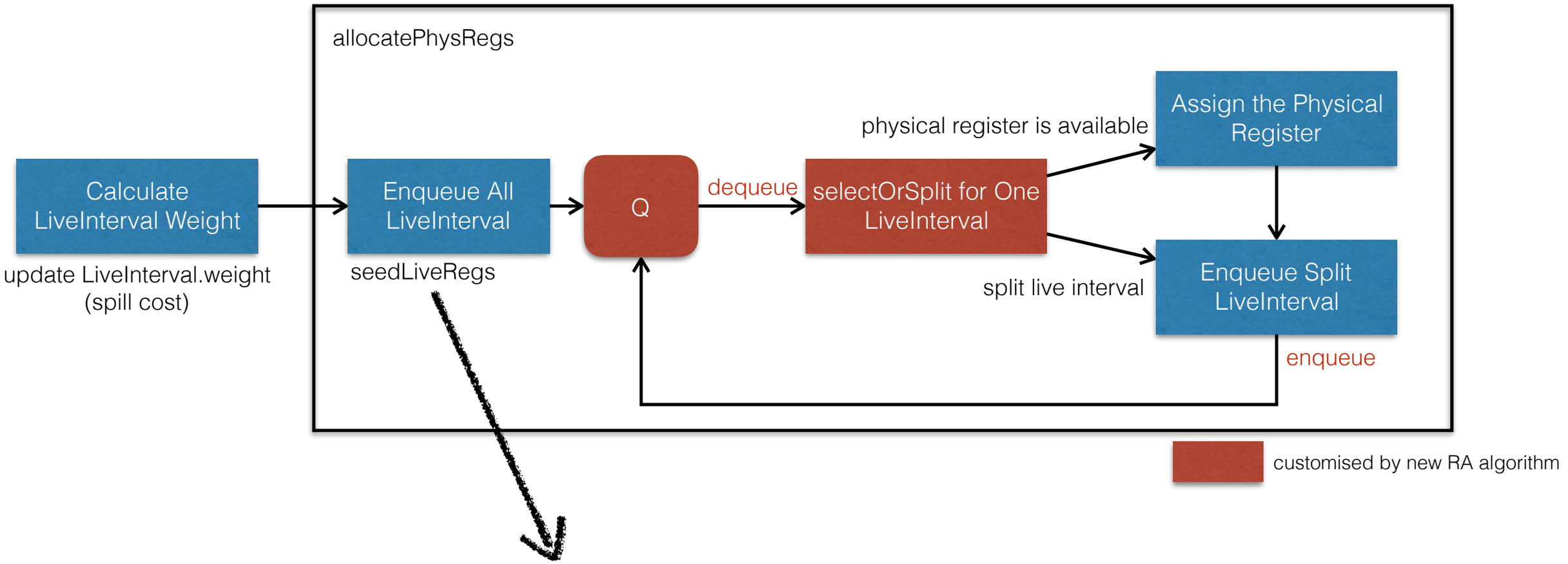
SSA Properties

* Each definition in the procedure creates a unique name.
* Each use refers to a single definition.

LLVM Register Allocation

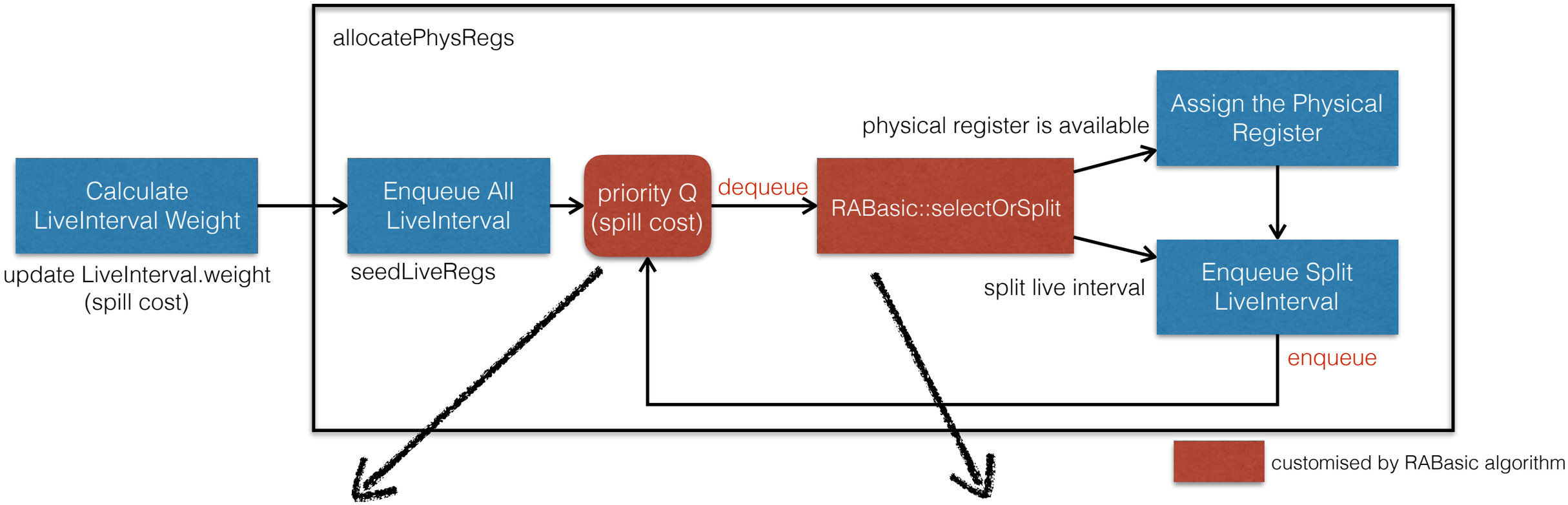
- Basic
  - Provide a minimal implementation of the basic register allocator
- Greedy
  - Global live range splitting.
- Fast
  - This register allocator allocates registers to a basic block at a time.
- PBQP
  - Partitioned Boolean Quadratic Programming (PBQP) based register allocator for LLVM

LLVM Base Register Allocation Interface




```
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));
}
```

LLVM Basic Register Allocation




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

```
// Check for an available register in this class.
AllocationOrder Order(VirtReg.reg, *VRM, RegClassInfo);
while (unsigned PhysReg = Order.next()) {
  // Check for interference in PhysReg
  switch (Matrix->checkInterference(VirtReg, PhysReg)) {
  case LiveRegMatrix::IK_Free:
    // PhysReg is available, allocate it.
    return PhysReg;

  case LiveRegMatrix::IK_VirtReg:
    // Only virtual registers in the way, we may be able to spill them.
    PhysRegSpillCands.push_back(PhysReg);
    continue;

  default:
    // RegMask or RegUnit interference.
    continue;
  }
}
```

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

Matrix->checkInterference()

- How to represent live/dead points?
  - SlotIndex
- How to represent a value?
  - VNInfo
- How to represent a live interval?
  - LiveInterval
- How to check interference between live intervals?
  - LiveIntervalUnion & LiveRegMatrix

Liveness Slot

- There are four kind of slots to describe a position at which a register can become live, or cease to be live.
  - Block (B)
    - entering or leaving a block
    - PHI-def
  - Early Clobber (e)
    - kill slot for early-clobber def
    - A = A op B (誤)
  - Register (r)
    - normal register use/def slot
  - Dead (d)
    - dead def

```
********** INTERVALS **********
%vreg0 [208r,320r:0)[416B,432r:0)  0@208r
%vreg1 [16r,32r:0)  0@16r
%vreg2 [48r,480B:0)  0@48r
%vreg3 [96r,112r:0)  0@96r
%vreg4 [496r,512r:0)  0@496r
%vreg6 [224r,240r:0)  0@224r
%vreg7 [432r,448r:0)  0@432r
%vreg8 [304r,320r:0)  0@304r
%vreg9 [320r,336r:0)  0@320r
%vreg10 [352r,368r:0)  0@352r
%vreg11 [368r,384r:0)  0@368r
```

SlotIndex

((MachineInstr *, index), slot)

listEntry()

```
Slot_Block
Slot_EarlyClobber
Slot_Register
Slot_Dead
```

```
unsigned getIndex() const {
  return listEntry()->getIndex() | getSlot();
}
```

Numbering of Machine Instruction

```
for (MachineBasicBlock::iterator miItr = mbb->begin(), miEnd = mbb->end();
     miItr != miEnd; ++miItr) {
  MachineInstr *mi = miItr;
  if (mi->isDebugValue())
    continue;

  // Insert a store index for the instr.
  indexList.push_back(createEntry(mi, index += SlotIndex::InstrDist));

  // Save this base index in the maps.
  mi2iMap.insert(std::make_pair(mi, SlotIndex(&indexList.back(),
                                              SlotIndex::Slot_Block)));
}
```

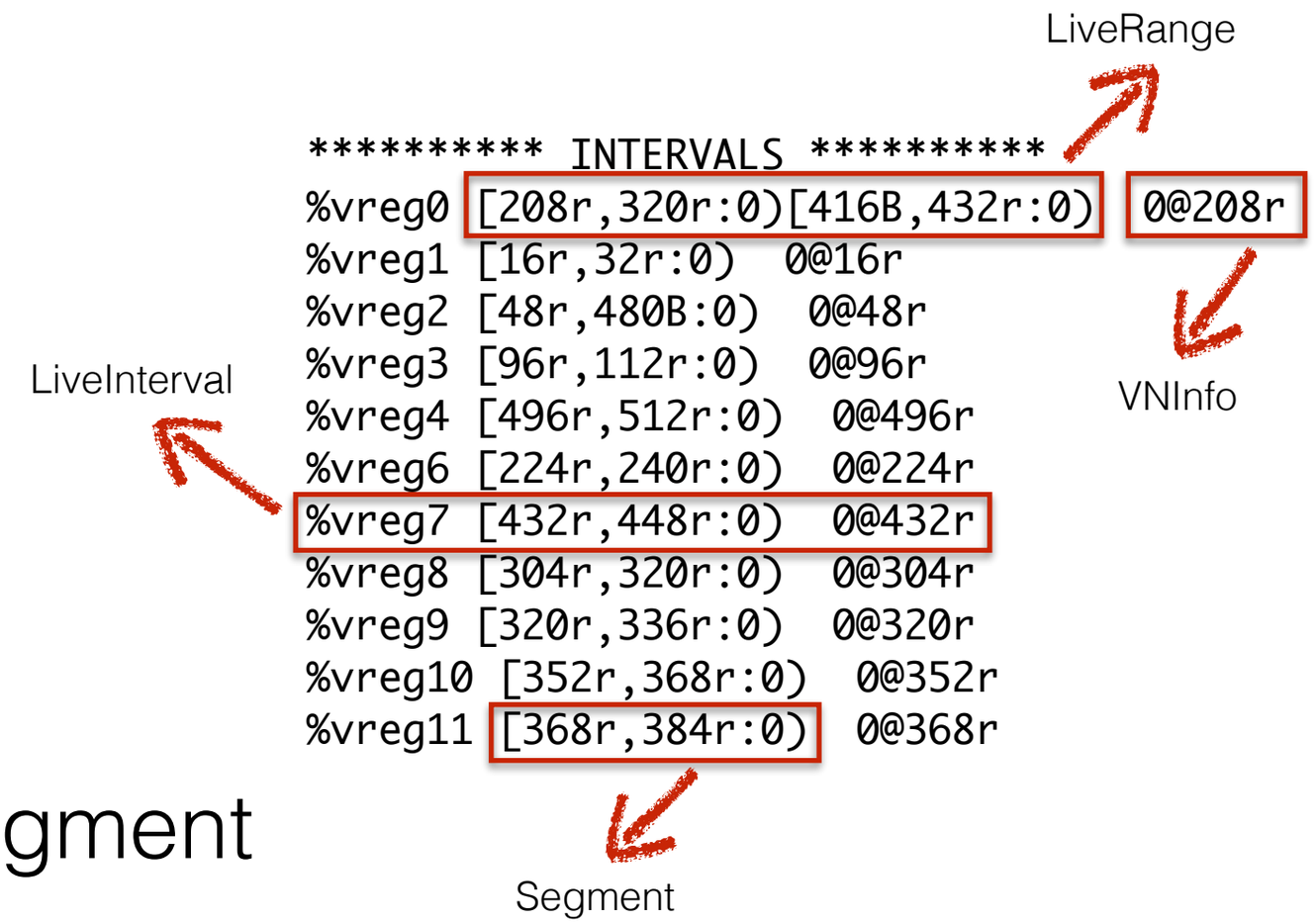
```
0B    BB#0: derived from LLVM BB %entry
16B       %vreg1<def> = t2MOVi 0, pred:14, pred:%noreg, opt:%noreg; rGPR:%vreg1
32B       t2STRi12 %vreg1, <fi#0>, 0, pred:14, pred:%noreg; mem:ST4[%retval] rGPR:%vreg1
48B       %vreg2<def> = t2MOVi 1, pred:14, pred:%noreg, opt:%noreg; rGPR:%vreg2
64B       t2STRi12 %vreg2, <fi#1>, 0, pred:14, pred:%noreg; mem:ST4[%i] rGPR:%vreg2
          Successors according to CFG: BB#1
```

VNInfo

- hold information about a machine level value
- (id, def)
  - def: SlotIndex of the defining instruction

Live Interval

- Segment
  - start, end, valno
- LiveRange
  - an ordered list of Segment
- LiveInterval
  - LiveRange with register and weight (spill cost)



Example

```
192B   BB#3: derived from LLVM BB %for.cond.1
208B          %vreg0<def> = t2LDRi12 <fi#1>, 0
224B          %vreg6<def> = t2LDRi12 <fi#2>, 0
240B          t2CMPri %vreg6, 9
256B          t2Bcc <BB#5>
272B          t2B <BB#4>
```

208r

```
416B   BB#5: derived from LLVM BB %for.inc.4
432B          %vreg7<def> = t2ADDri %vreg0, 1
448B          t2STRi12 %vreg7, <fi#1>, 0
```

416B

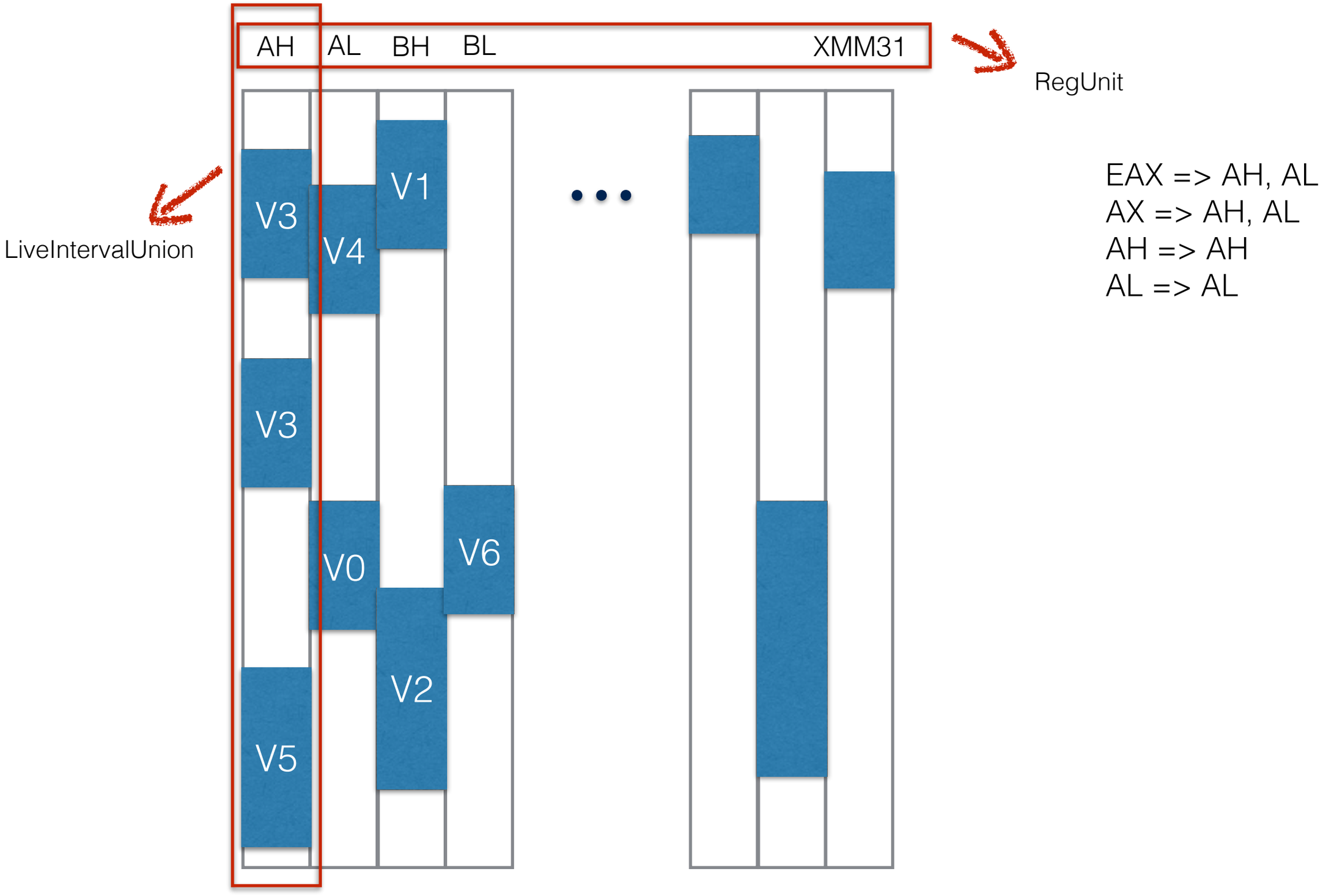
432r

```
288B   BB#4: derived from LLVM BB %for.body.3
304B          %vreg8<def> = t2LDRi12 <fi#2>, 0
320B          %vreg9<def> = t2MUL %vreg0, %vreg8
336B          t2STRi12 %vreg9, <fi#3>, 0
352B          %vreg10<def> = t2LDRi12 <fi#2>, 0
368B          %vreg11<def> = t2ADDri %vreg10, 1
384B          t2STRi12 %vreg11, <fi#2>, 0
400B          t2B <BB#3>
```

320r

```
********** INTERVALS **********
%vreg0 [208r,320r:0)[416B,432r:0)  0@208r
%vreg1 [16r,32r:0)  0@16r
%vreg2 [48r,480B:0)  0@48r
%vreg3 [96r,112r:0)  0@96r
%vreg4 [496r,512r:0)  0@496r
%vreg6 [224r,240r:0)  0@224r
%vreg7 [432r,448r:0)  0@432r
%vreg8 [304r,320r:0)  0@304r
%vreg9 [320r,336r:0)  0@320r
%vreg10 [352r,368r:0)  0@352r
%vreg11 [368r,384r:0)  0@368r
```

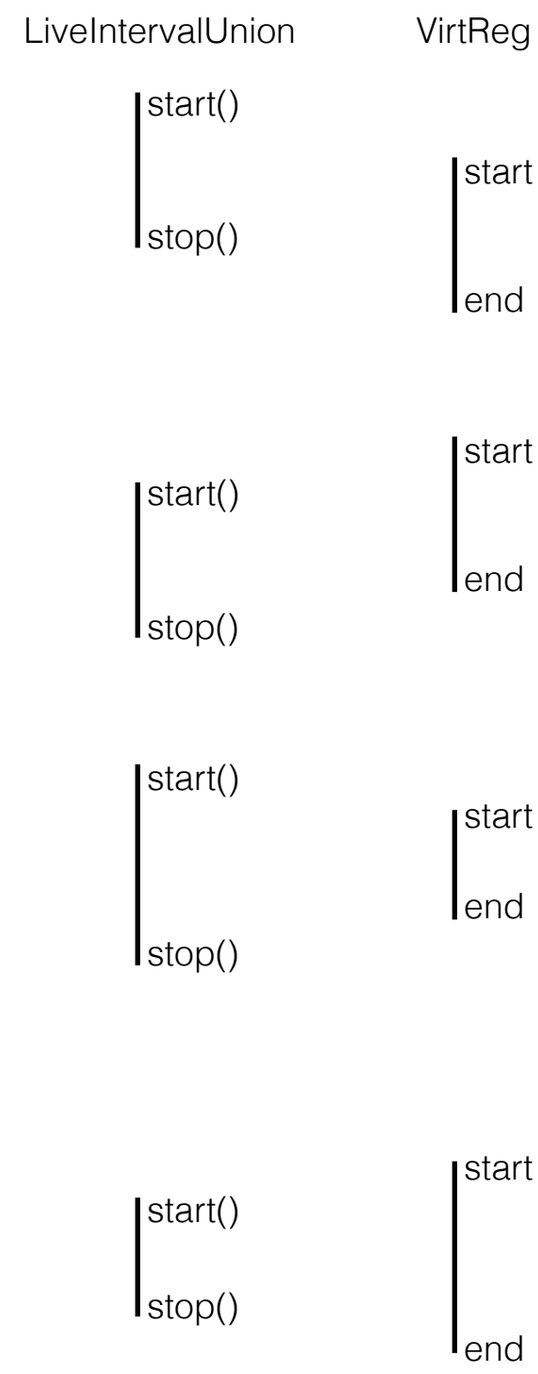
LiveRegMatrix



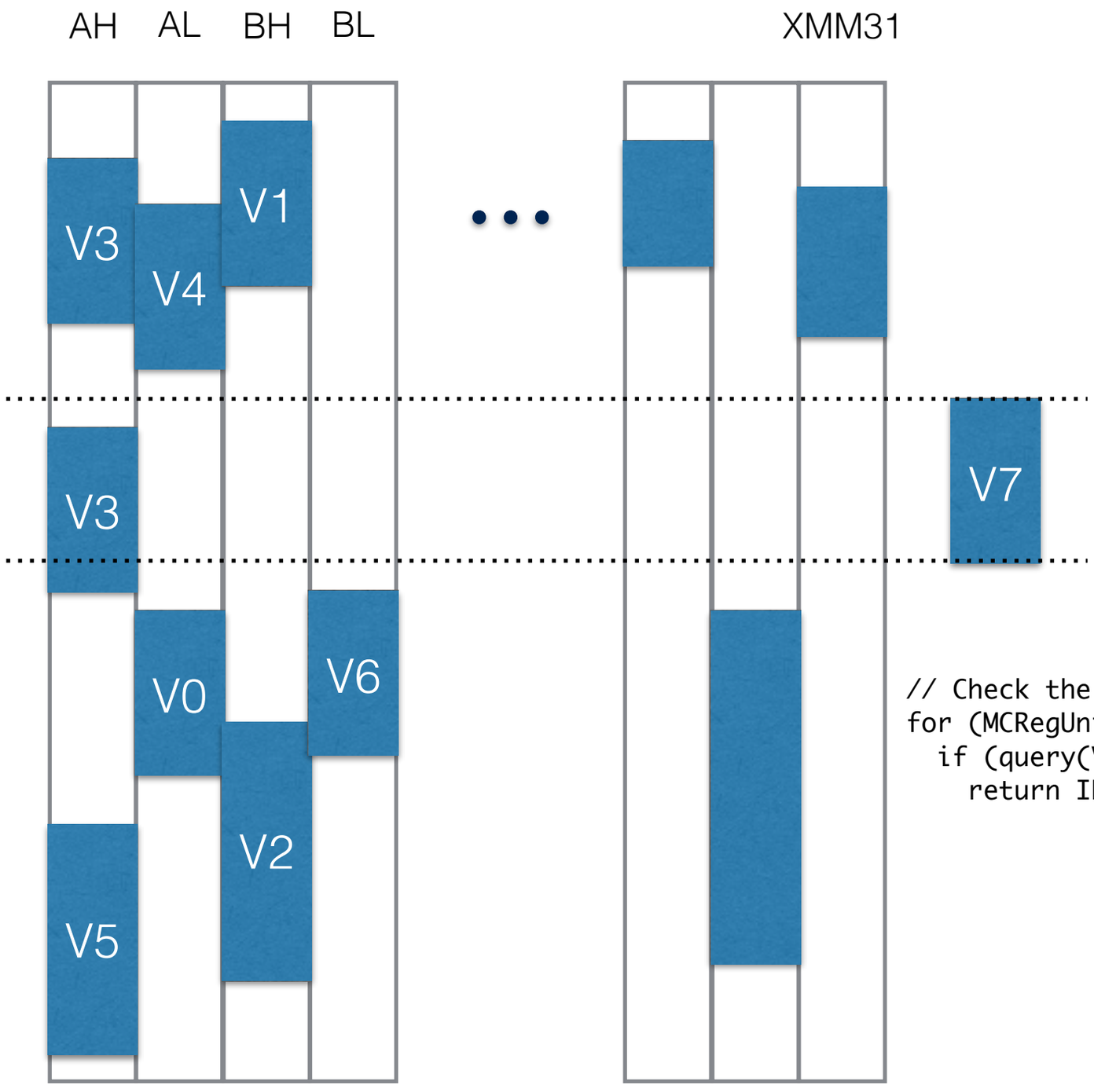
EAX => AH, AL
AX => AH, AL
AH => AH
AL => AL

Check Interference

```
unsigned LiveIntervalUnion::Query::
collectInterferingVRegs(unsigned MaxInterferingRegs) {
    …
    // Check for overlapping interference.
    while (VirtRegI->start < LiveUnionI.stop() &&
           VirtRegI->end > LiveUnionI.start()) {
      // This is an overlap, record the interfering register.
      LiveInterval *VReg = LiveUnionI.value();
      if (VReg != RecentReg && !isSeenInterference(VReg)) {
        RecentReg = VReg;
        InterferingVRegs.push_back(VReg);
        if (InterferingVRegs.size() >= MaxInterferingRegs)
          return InterferingVRegs.size();
      }
      // This LiveUnion segment is no longer interesting.
      if (!(++LiveUnionI).valid()) {
        SeenAllInterferences = true;
        return InterferingVRegs.size();
      }
    }
    …
}
```



Check Interference



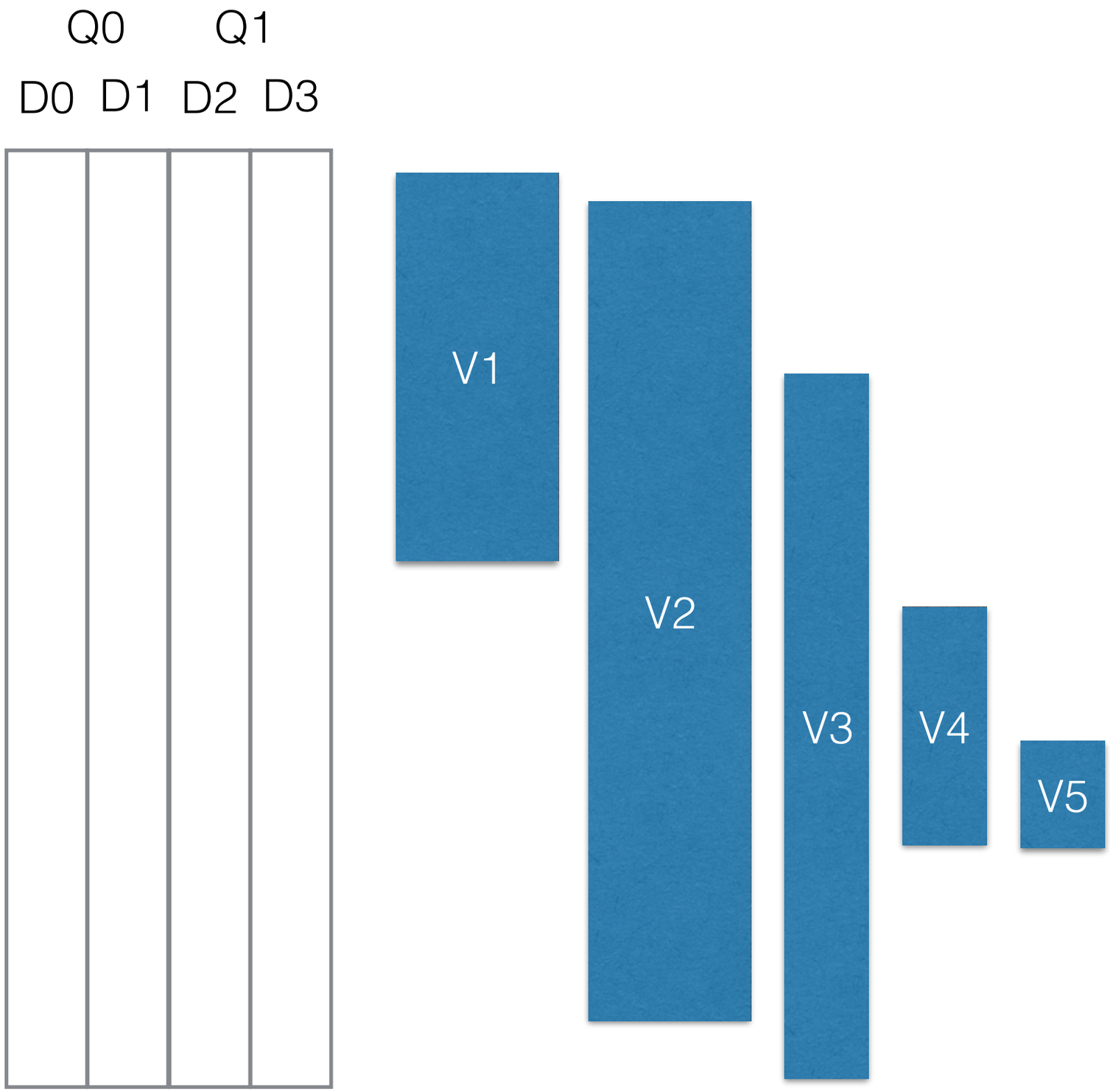

```
// Check the matrix for virtual register interference.
for (MCRegUnitIterator Units(PhysReg, TRI); Units.isValid(); ++Units)
  if (query(VirtReg, *Units).checkInterference())
    return IK_VirtReg;
```

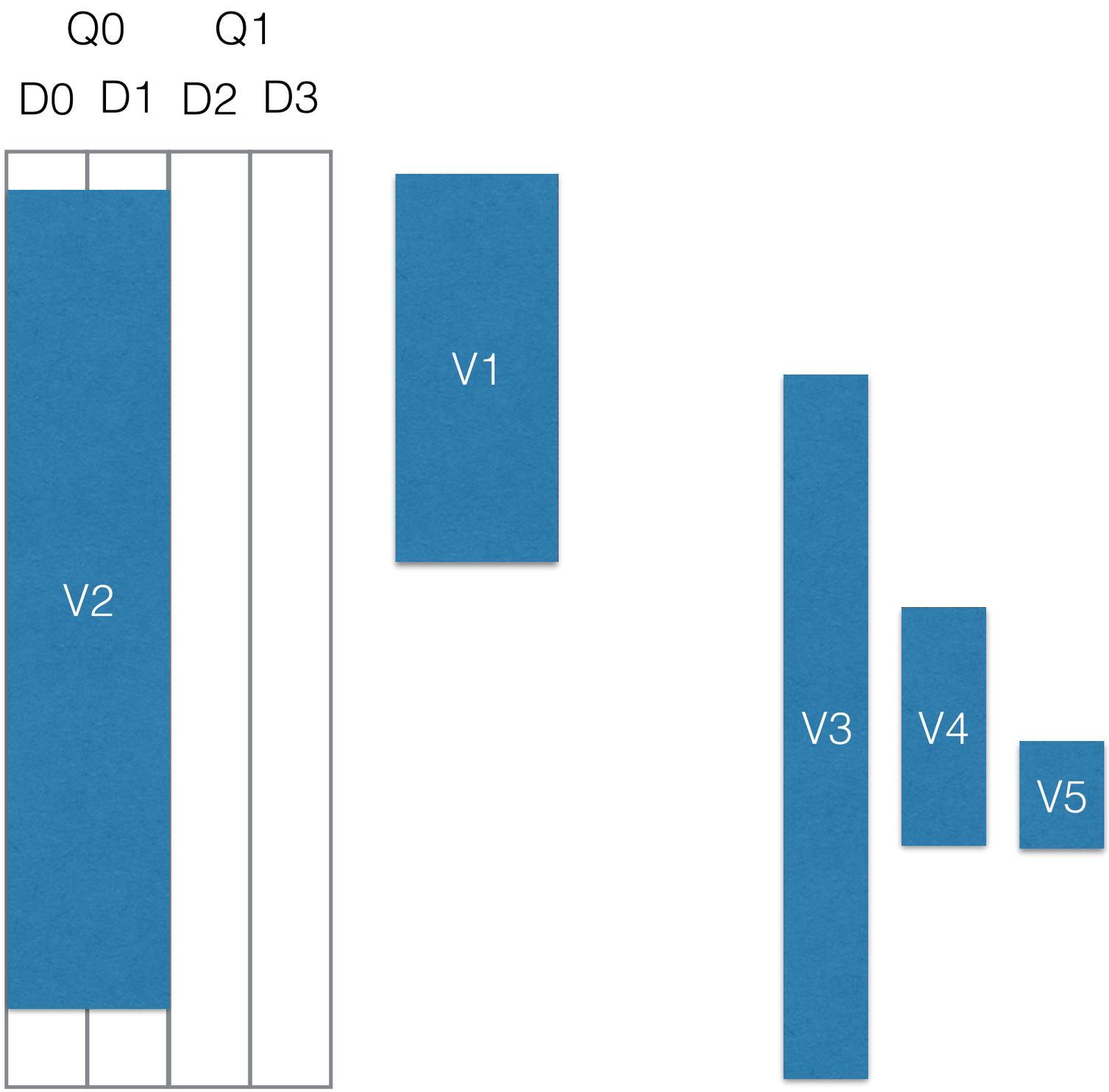
Greedy Register Allocation

Use Split to Improve RA

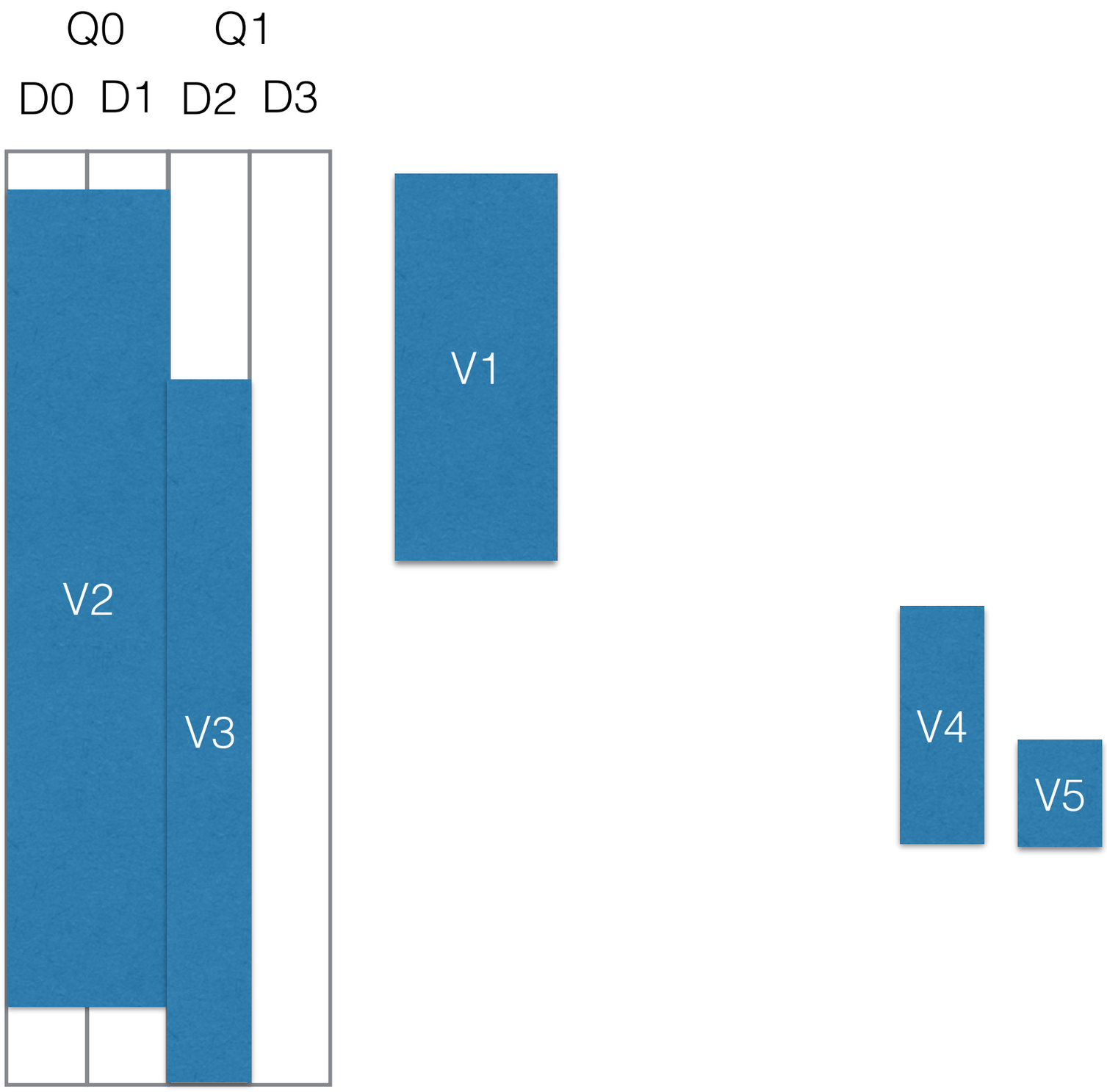
- Live Range Splitting
  - Insert copy/re-materialize to split up live ranges
    - hopefully reduces need for spilling
  - Also control spill code placement

- Example

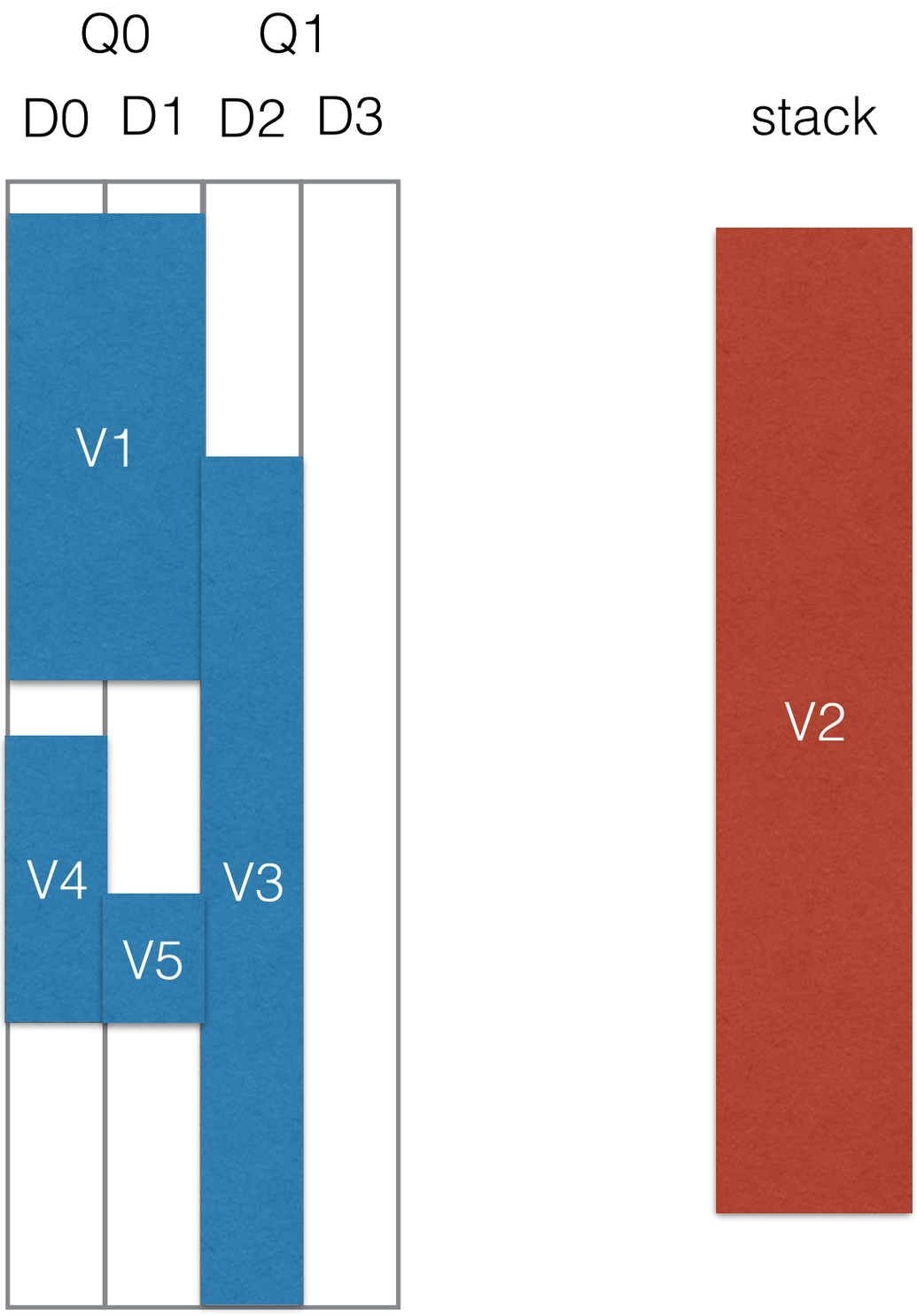



Q0
Q1
D0
D1
D2
D3
V1
V2
V3
V4
V5

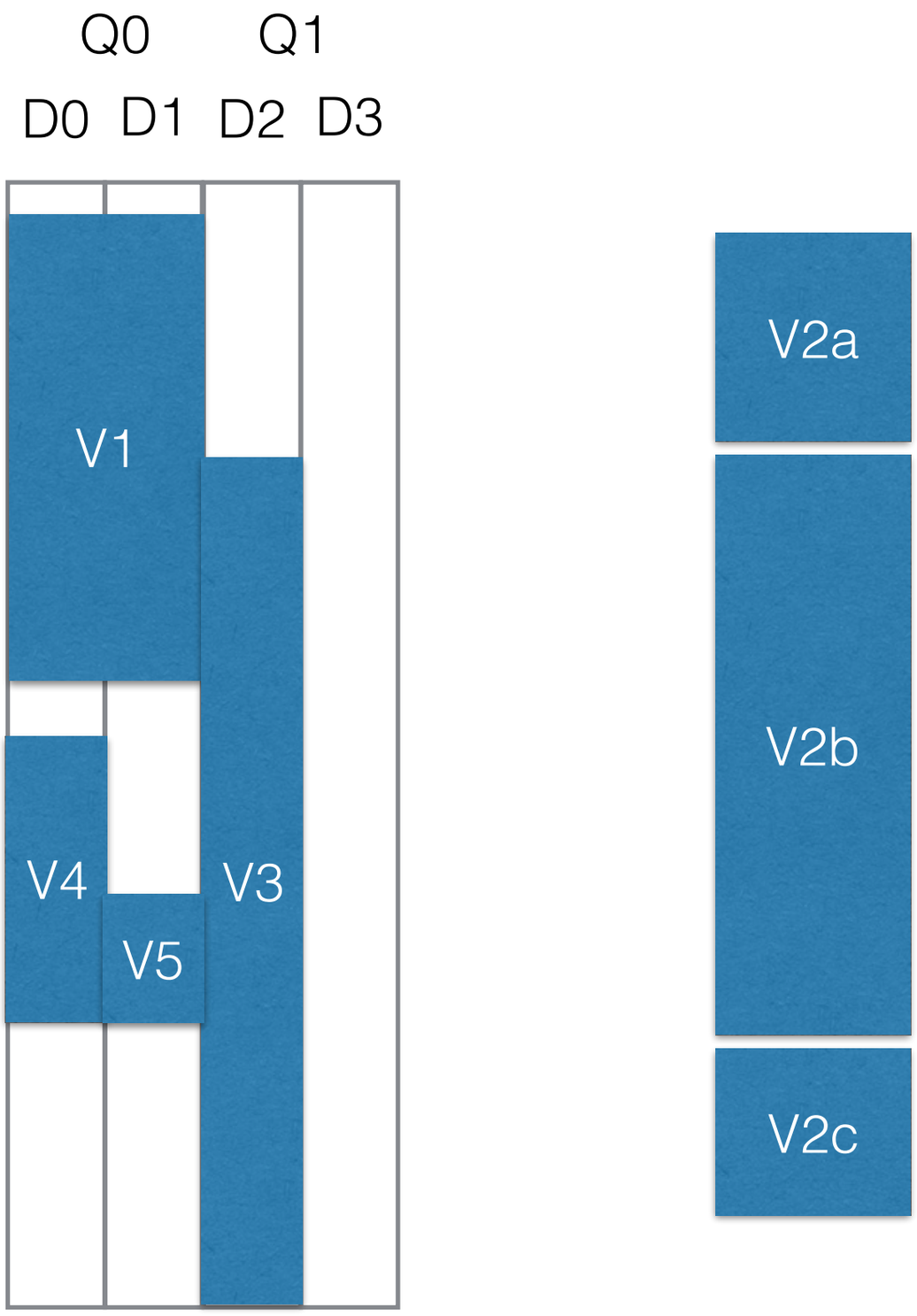
- No physical register for V1



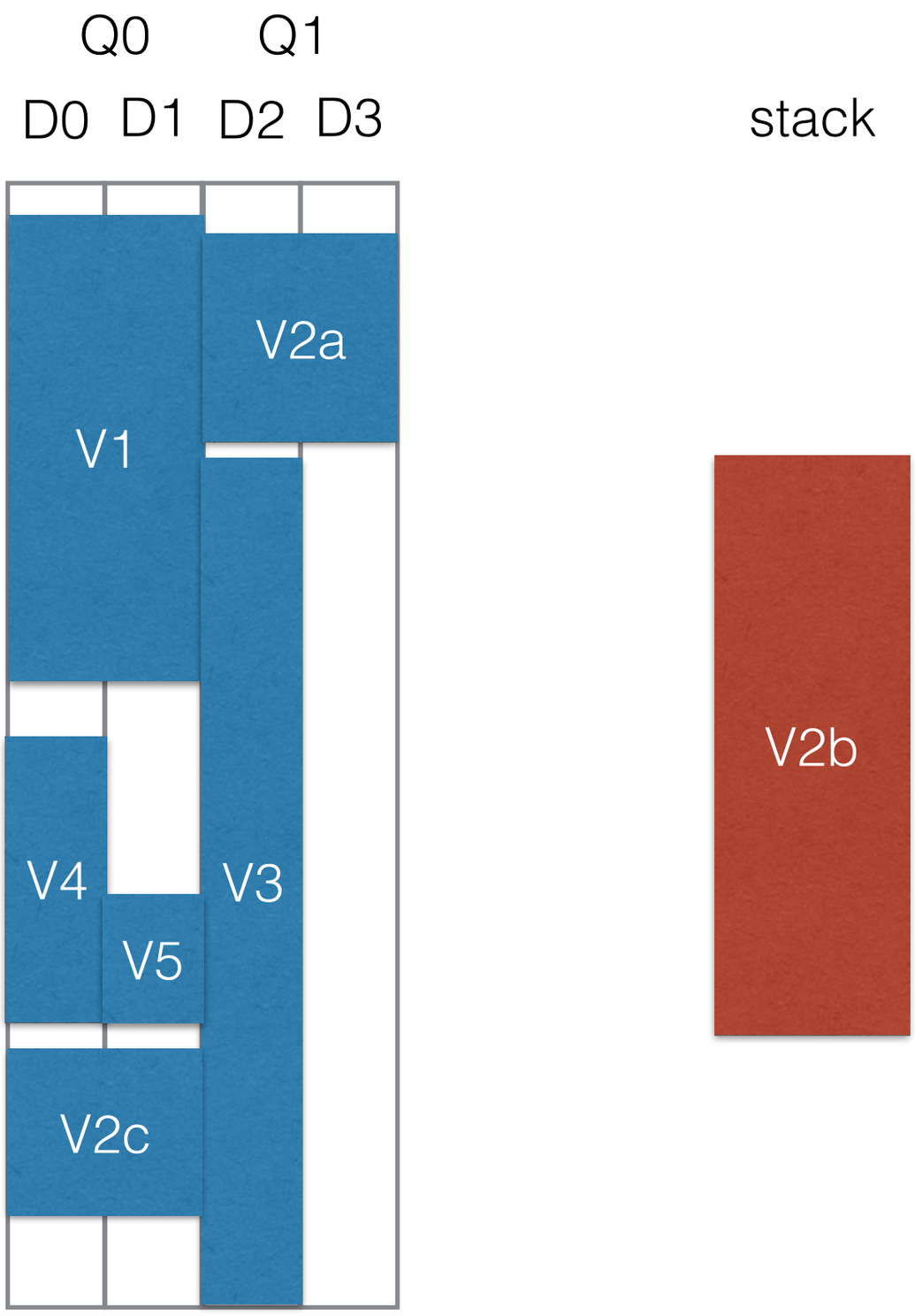
- Evict V2



- Split V2



- Split V2

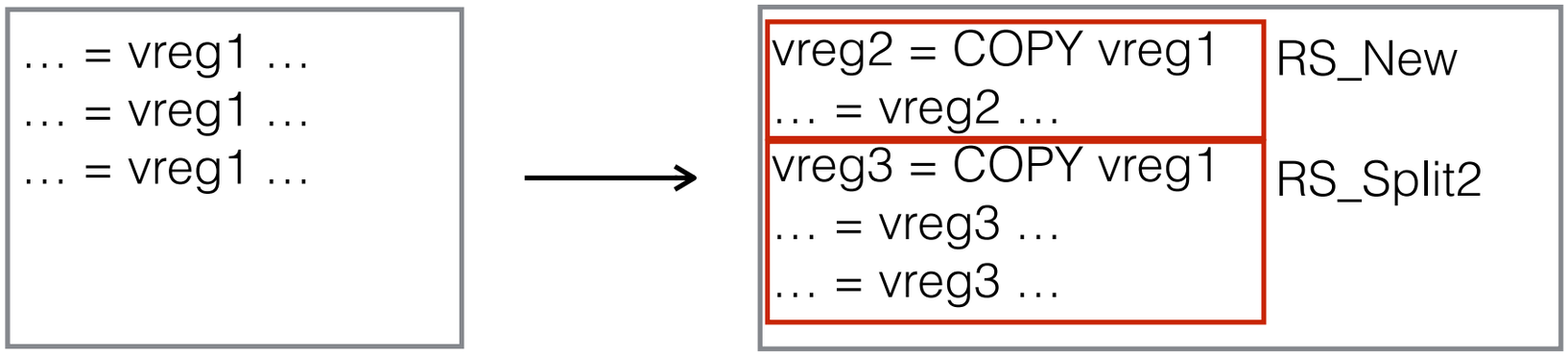


Greedy RA Stages

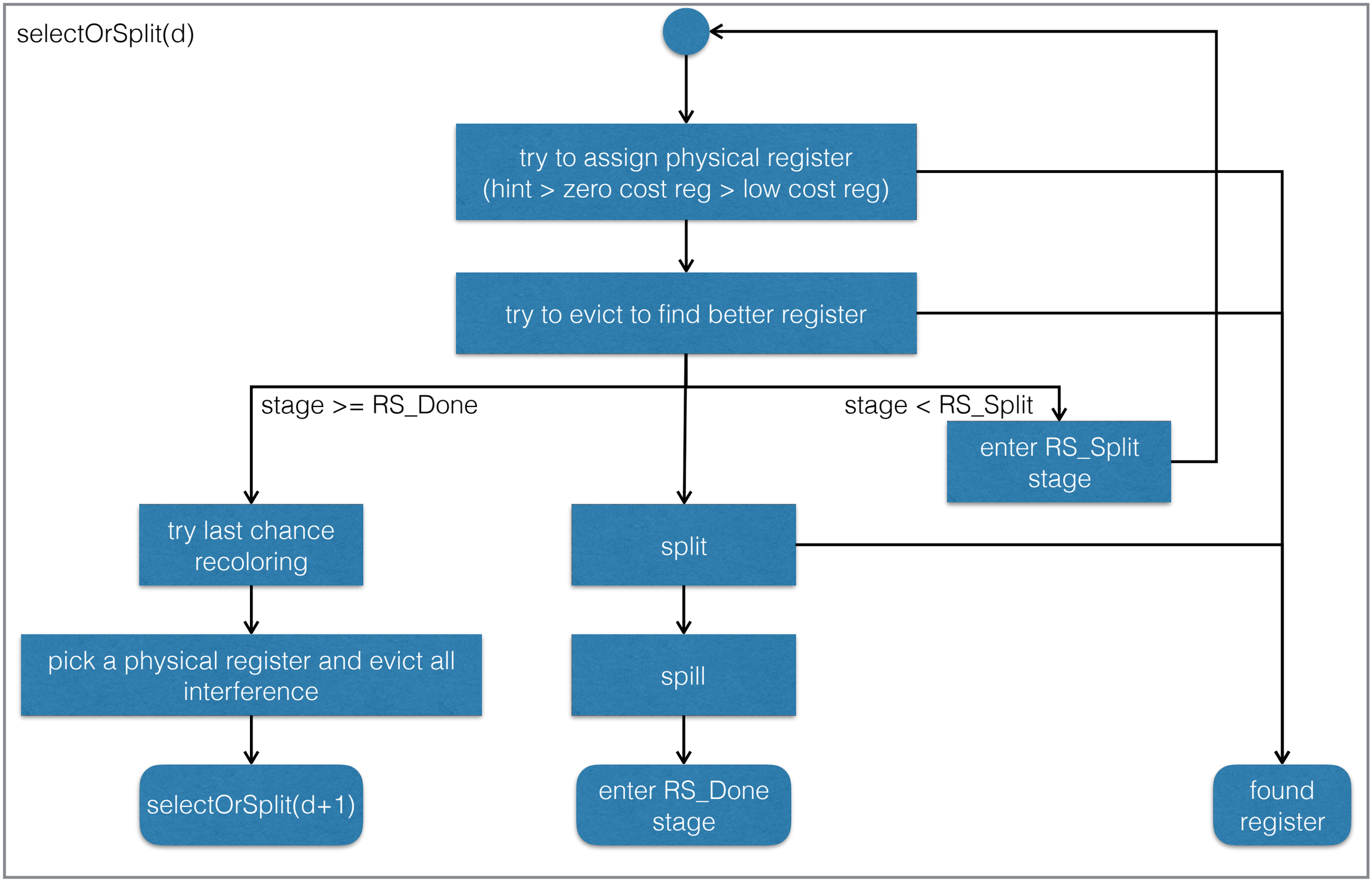
- RS_New: created
- RS_Assign: enqueue
- RS_Split: need to split
- RS_Split2
  - used for split products that may not be making progress
- RS_Spill: need to spill
- RS_Done: assigned a physical register or created by spill

RS_Split2

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



Greedy Register Allocation



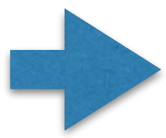
Last Chance Recoloring

- Try to assign a color to VirtReg by recoloring 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.

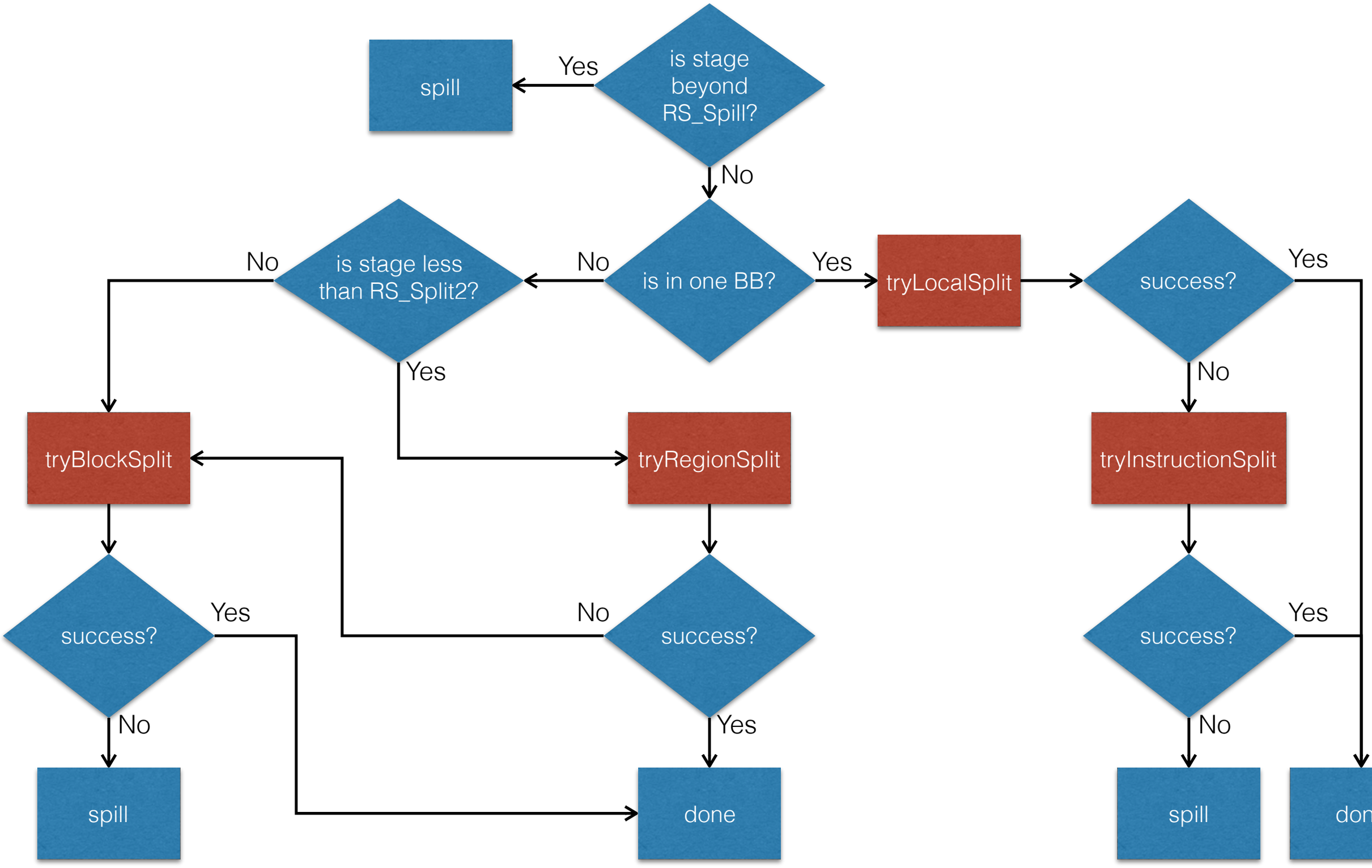
vA can use {R1, R2 }
vB can use { R2, R3}
vC can use {R1 }

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

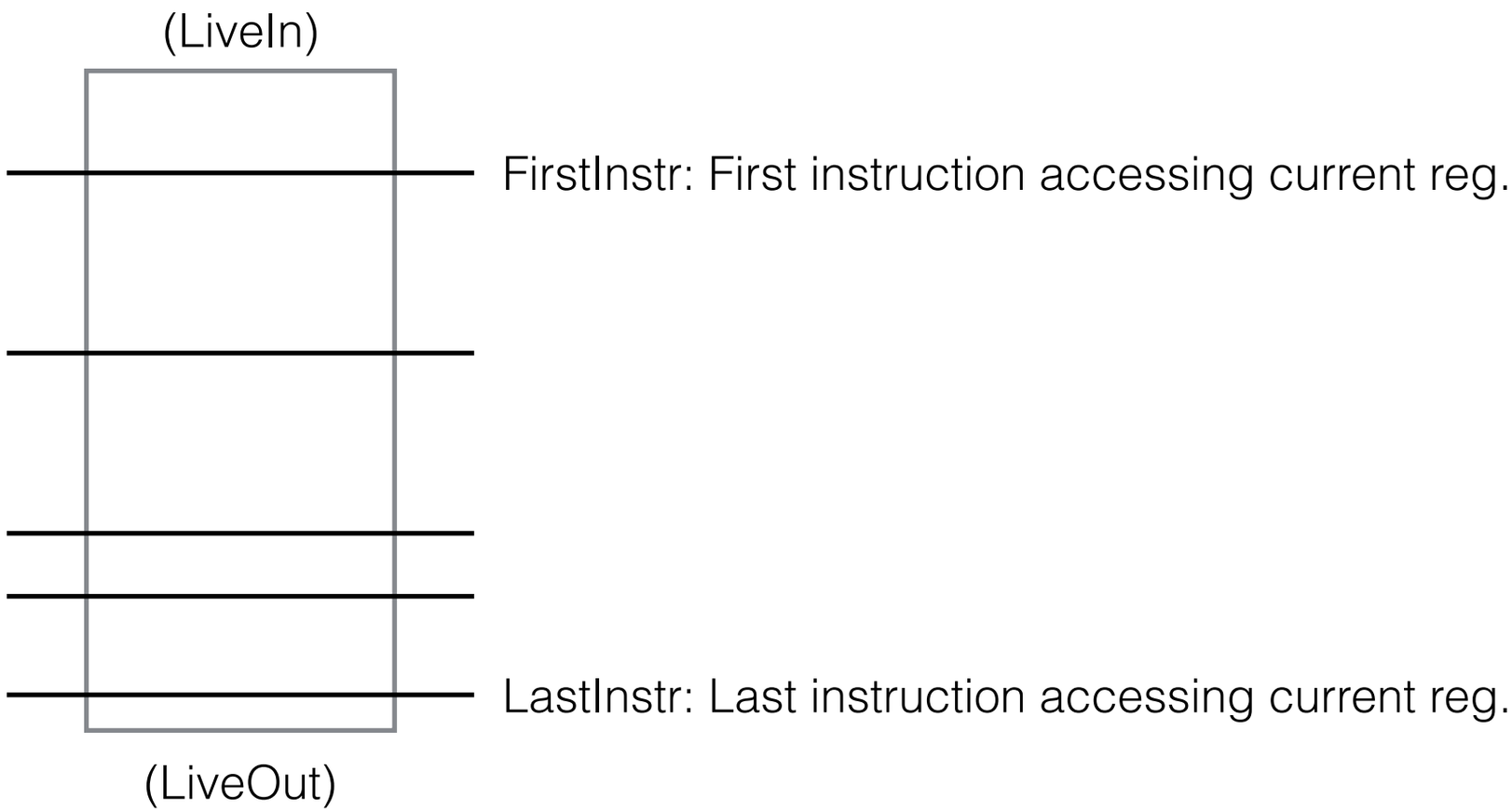


vA => R2
vB => R3
vC => R1 (fixed)

How to Split?


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

BlockInfo

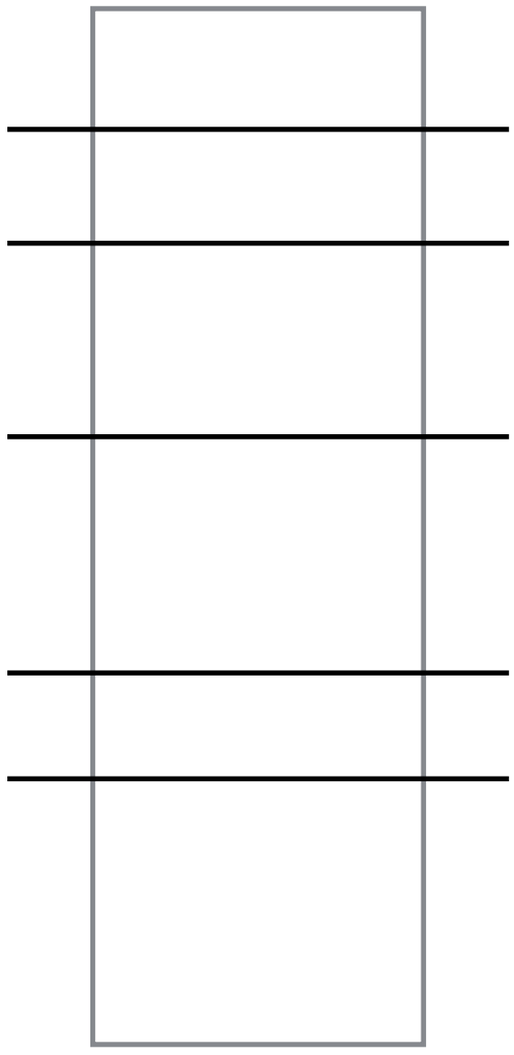


Live-through blocks without any uses don’t get BlockInfo entries.

tryLocalSplit

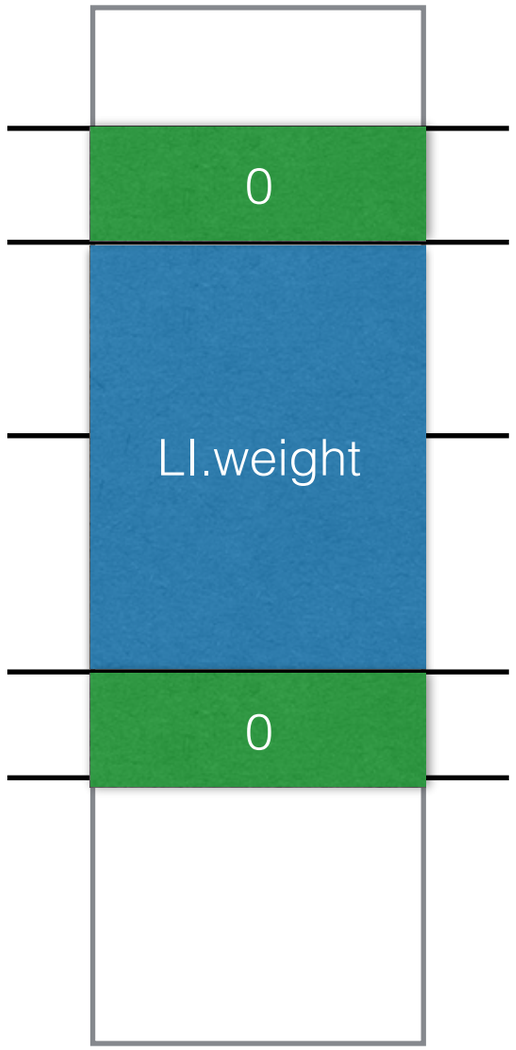
- Try to split virtual register interval into smaller intervals inside its only basic block.
    - calculate gap weights
    - adjust the split region

Calculate Gap Weights



NumGaps = 4

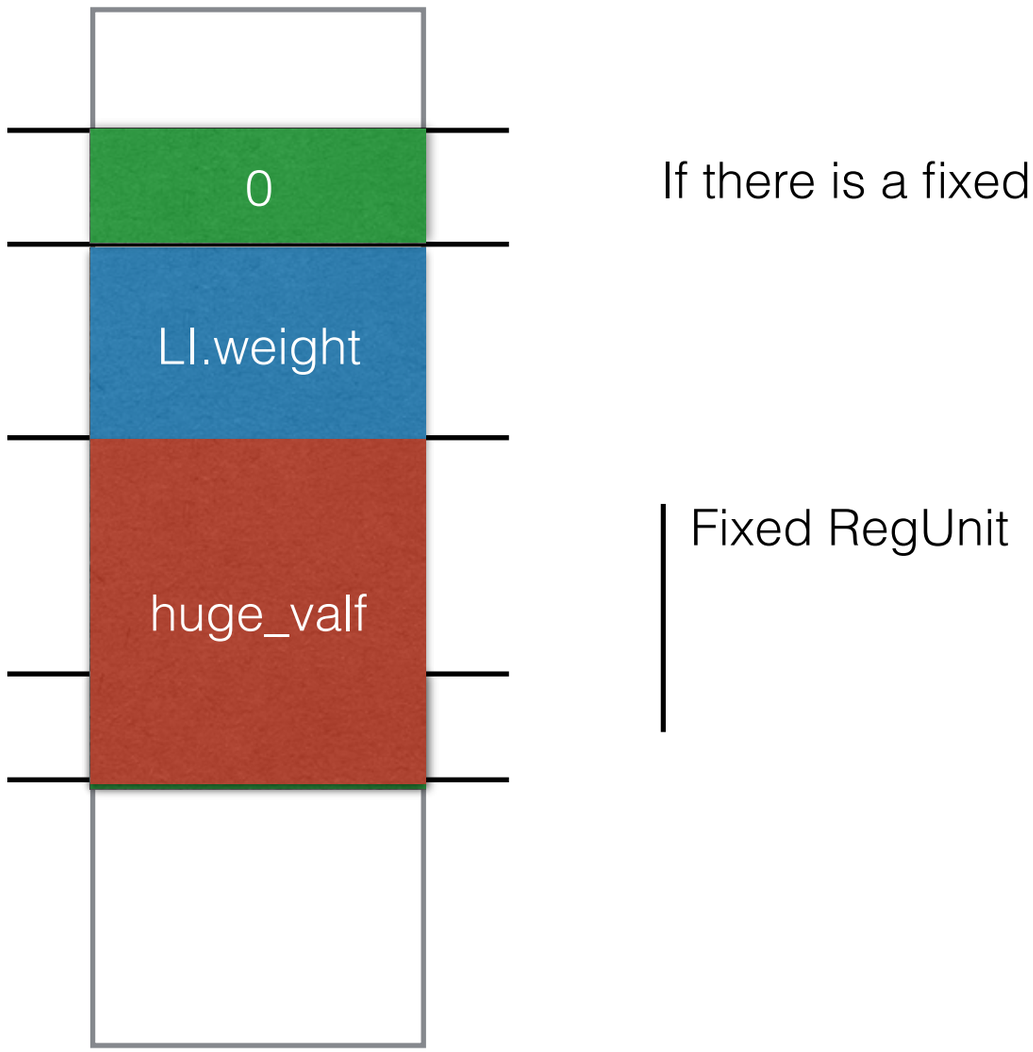
Calculate Gap Weights



If there is a RegUnit occupied by VirtReg:

VirtReg LI

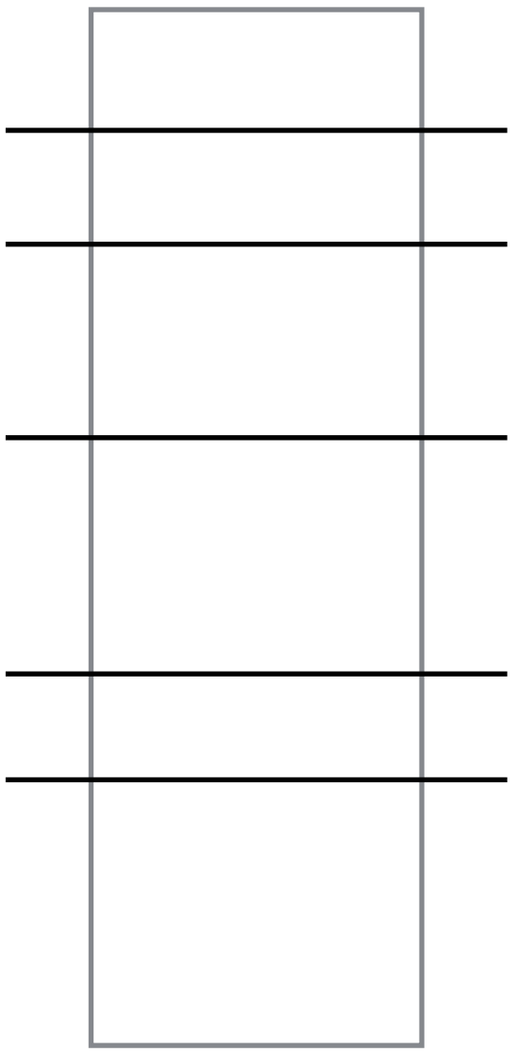
Calculate Gap Weights

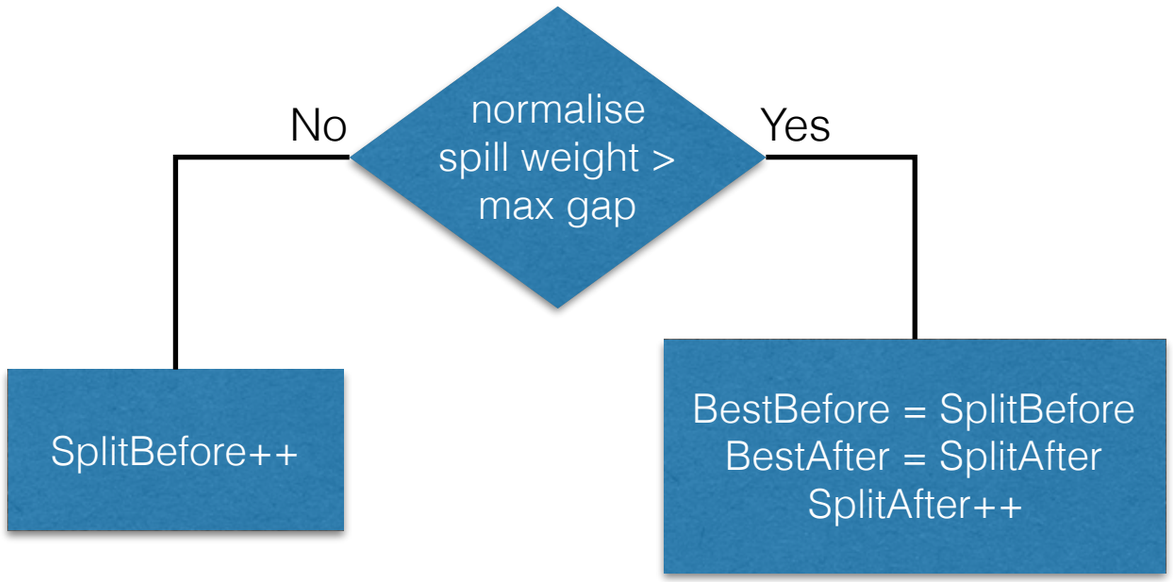


If there is a fixed RegUnit:

Fixed RegUnit

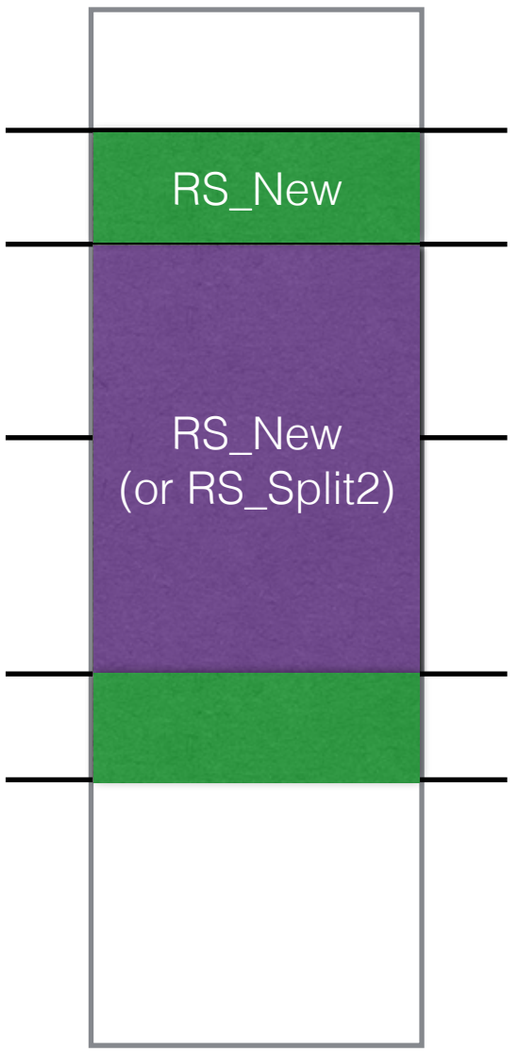
Adjust Split Region





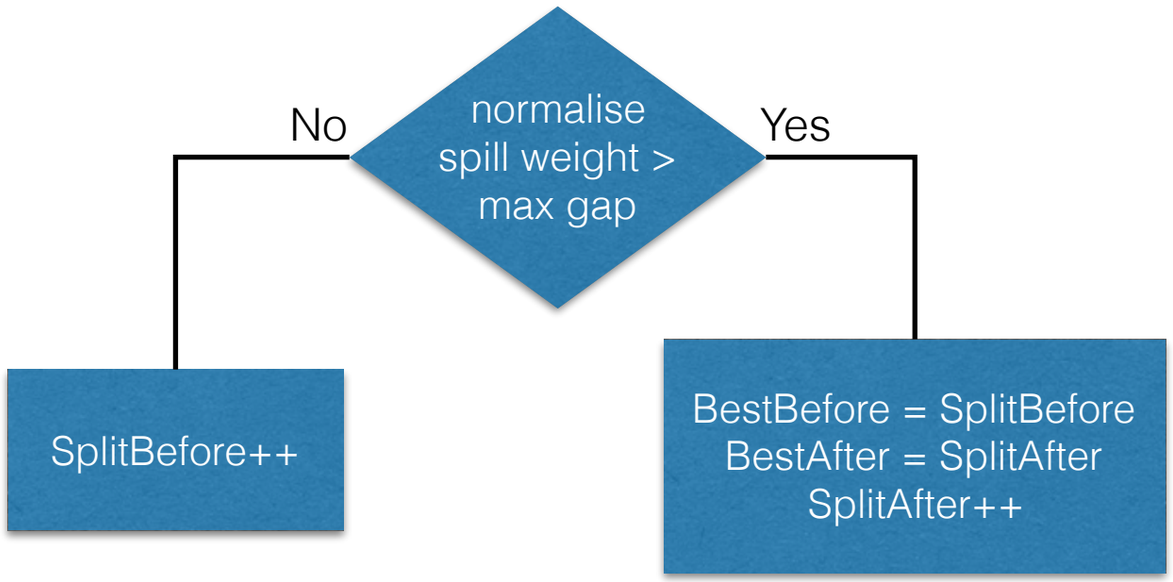
normalise spill weight = spill cost / distance
= (#gap * block_freq) / distance(SplitBefore, SplitAfter)

Adjust Split Region



BestBefore

BestAfter



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

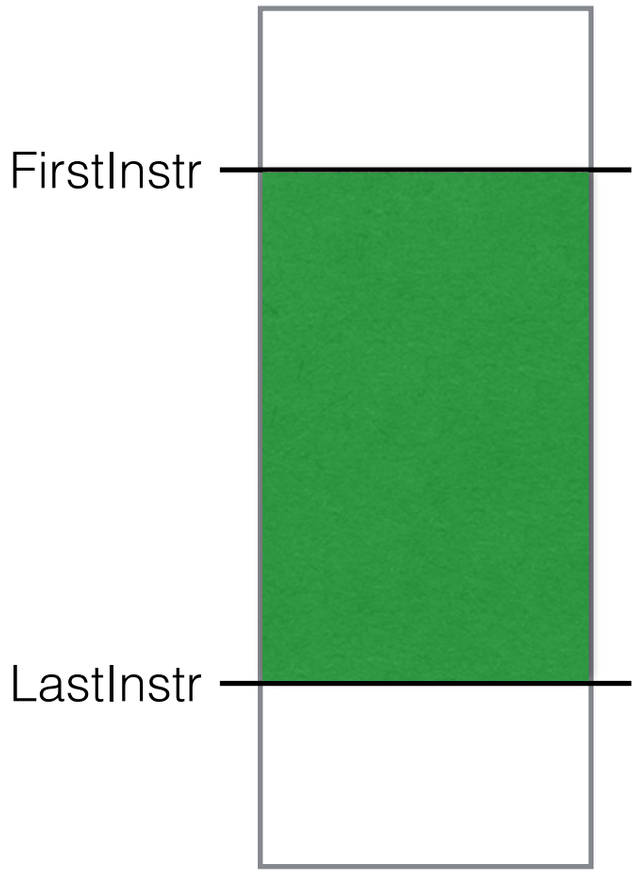
Find the most critical range.

tryInstructionSplit

- Split a live range around individual instructions.
- Every “use” instruction has its own live interval.

tryBlockSplit

- Split a global live range around every block with uses.



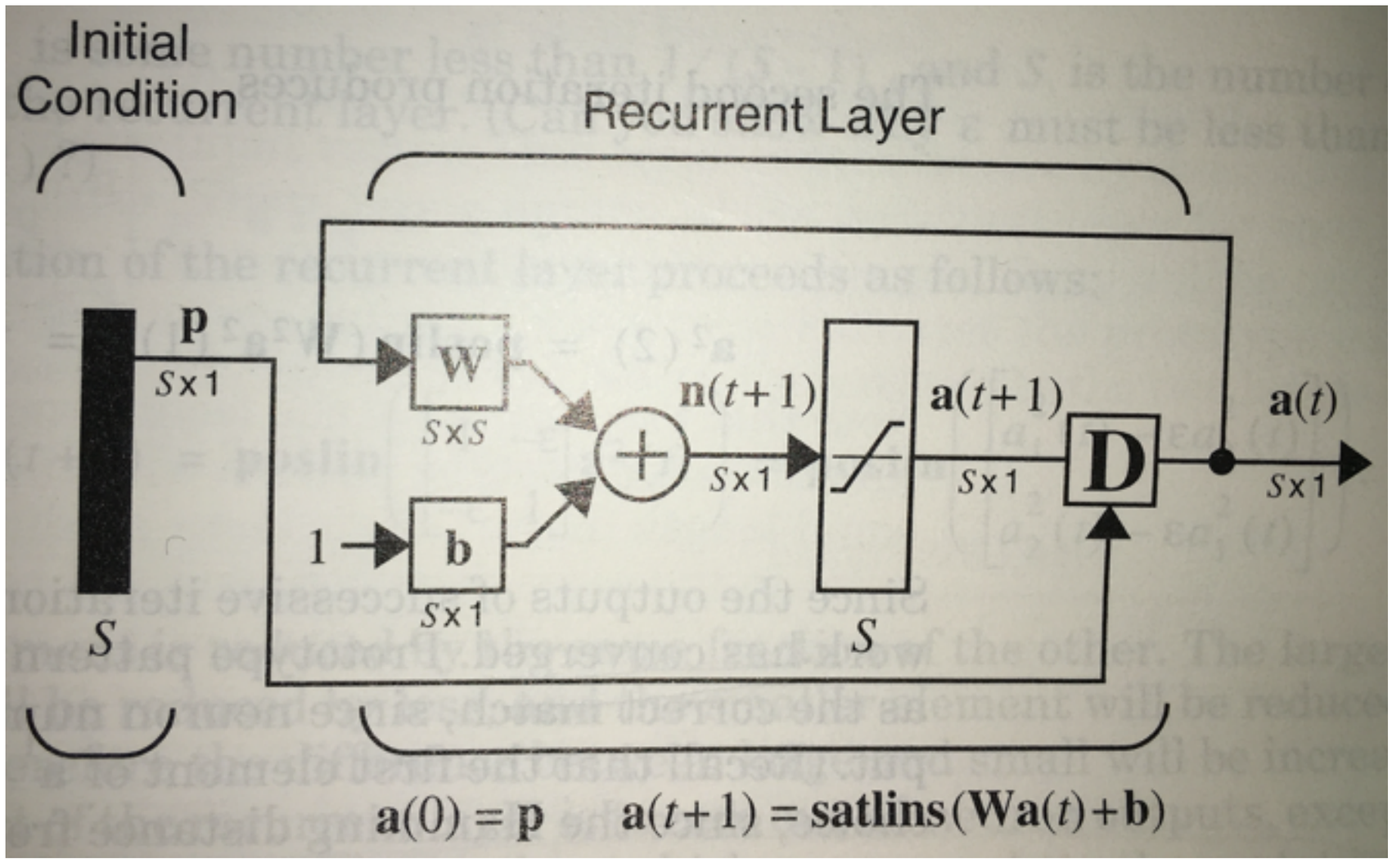
tryRegionSplit

- For every physical register
    - Prepare interference cache
    - Construct Hopfield Network
        - Construct block constraints
        - Update Hopfield Network biases and values according to block constraints
        - Add links in Hopfield Network and iterate
- Get the best candidate (minimize split cost + spill cost)
- Do region split

Hopfield Network

- A form of recurrent artificial neural network popularised by John Hopfield in 1982.

- Guaranteed to converge to a local minimum.

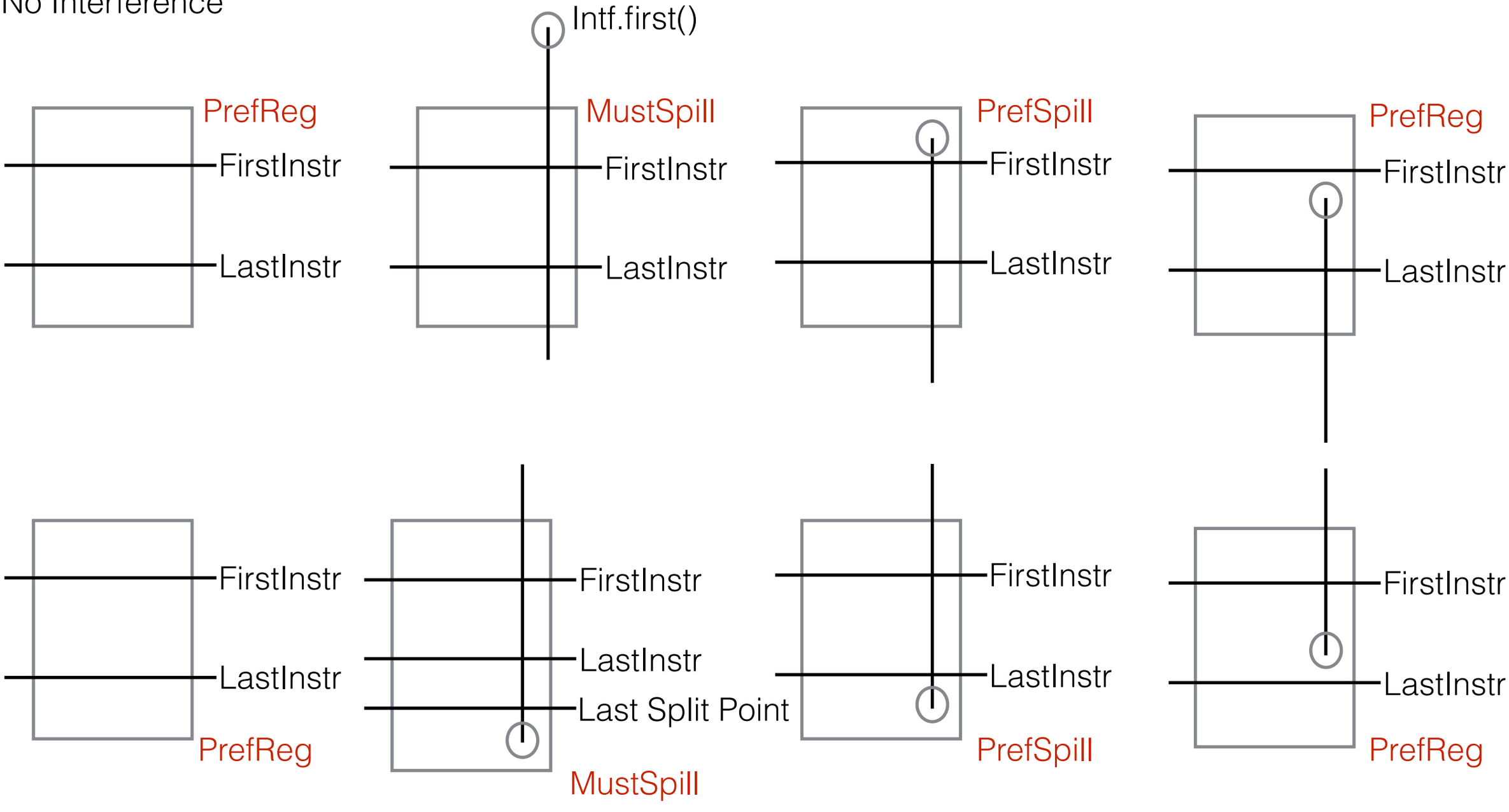


Hopfield Network

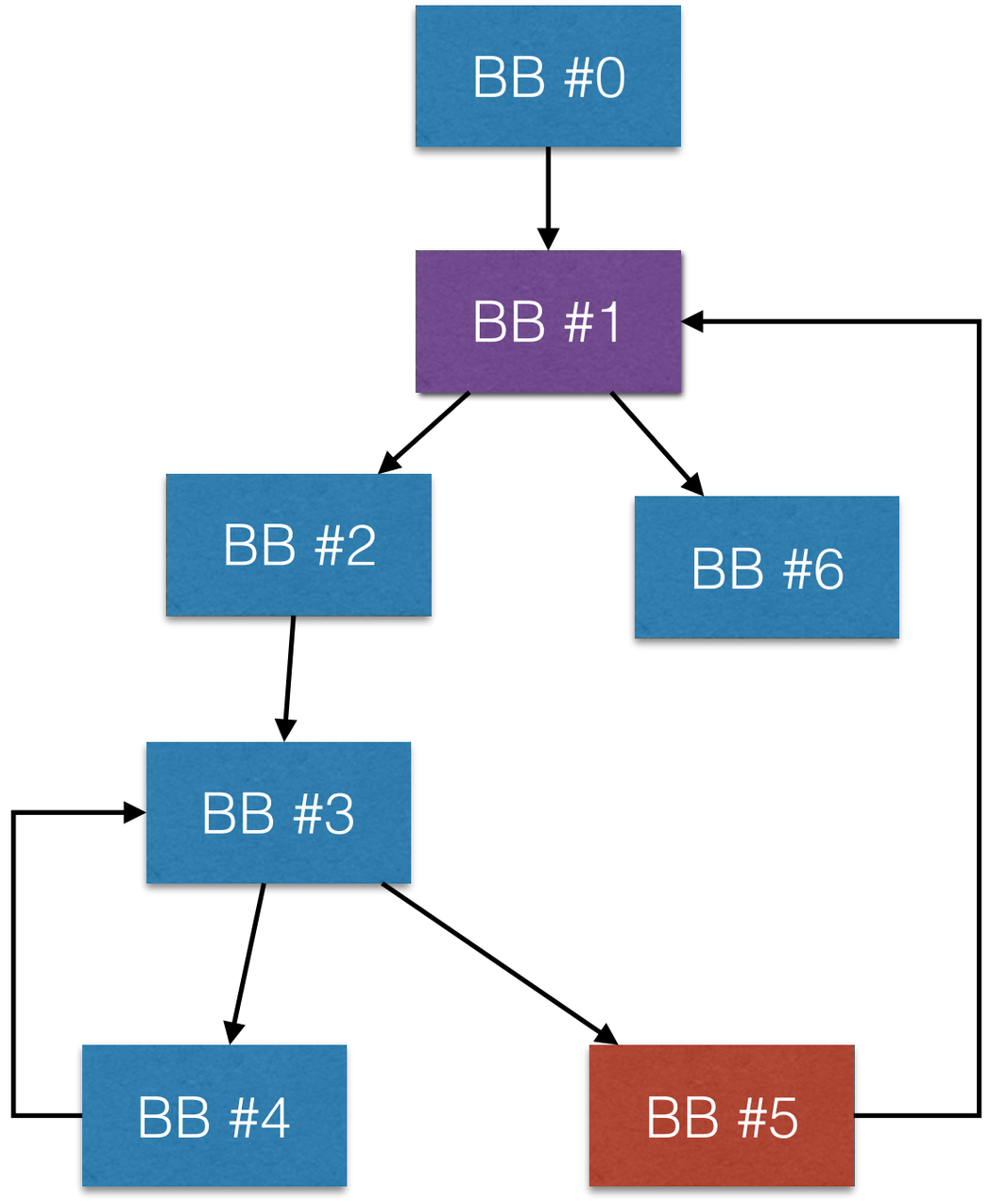
- Node: edge bundle
- Link: transparent basic blocks have the variable live through.
- Energy function (the cost of spilling)
- Weight: block frequency
- Bias: according to block constraints

Block Constraints

No Interference



Edge Bundle

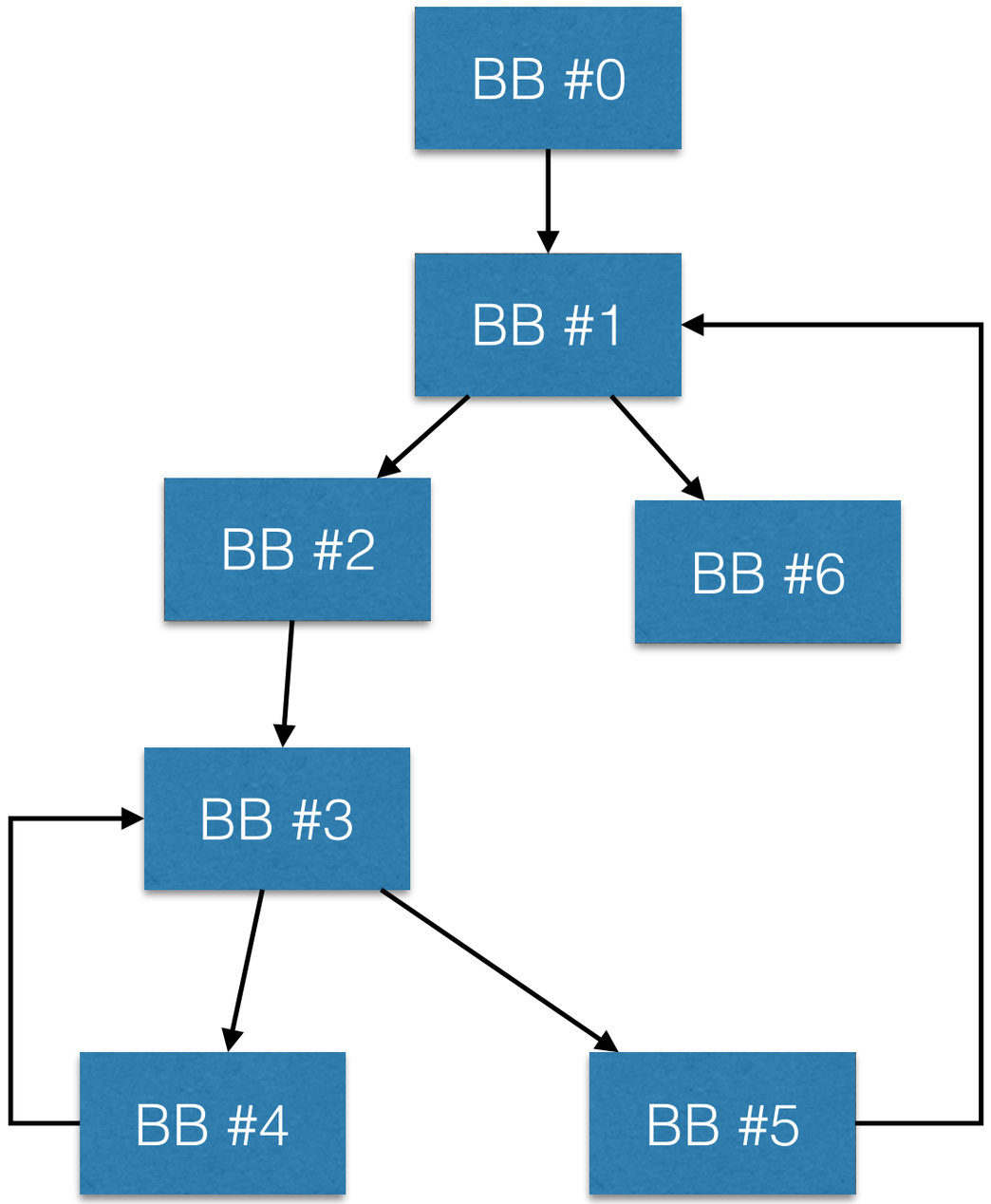



```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8: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

```
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];
}
```

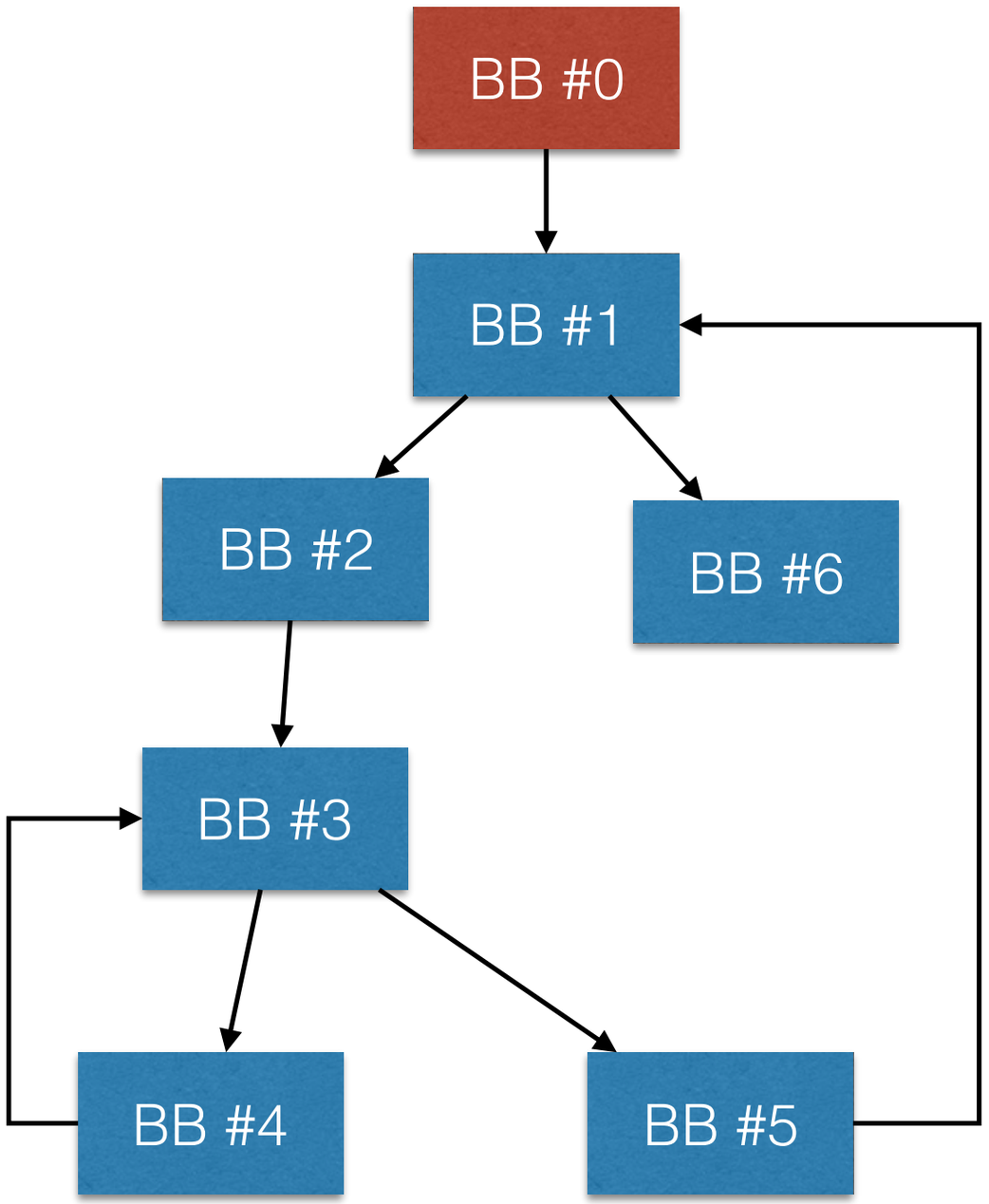
```
// Join the outgoing bundle with the ingoing bundles of all successors.
for (MachineBasicBlock::const_succ_iterator SI = MBB.succ_begin(),
       SE = MBB.succ_end(); SI != SE; ++SI)
  EC.join(OutE, 2 * (*SI)->getNumber());
```

Edge Bundle



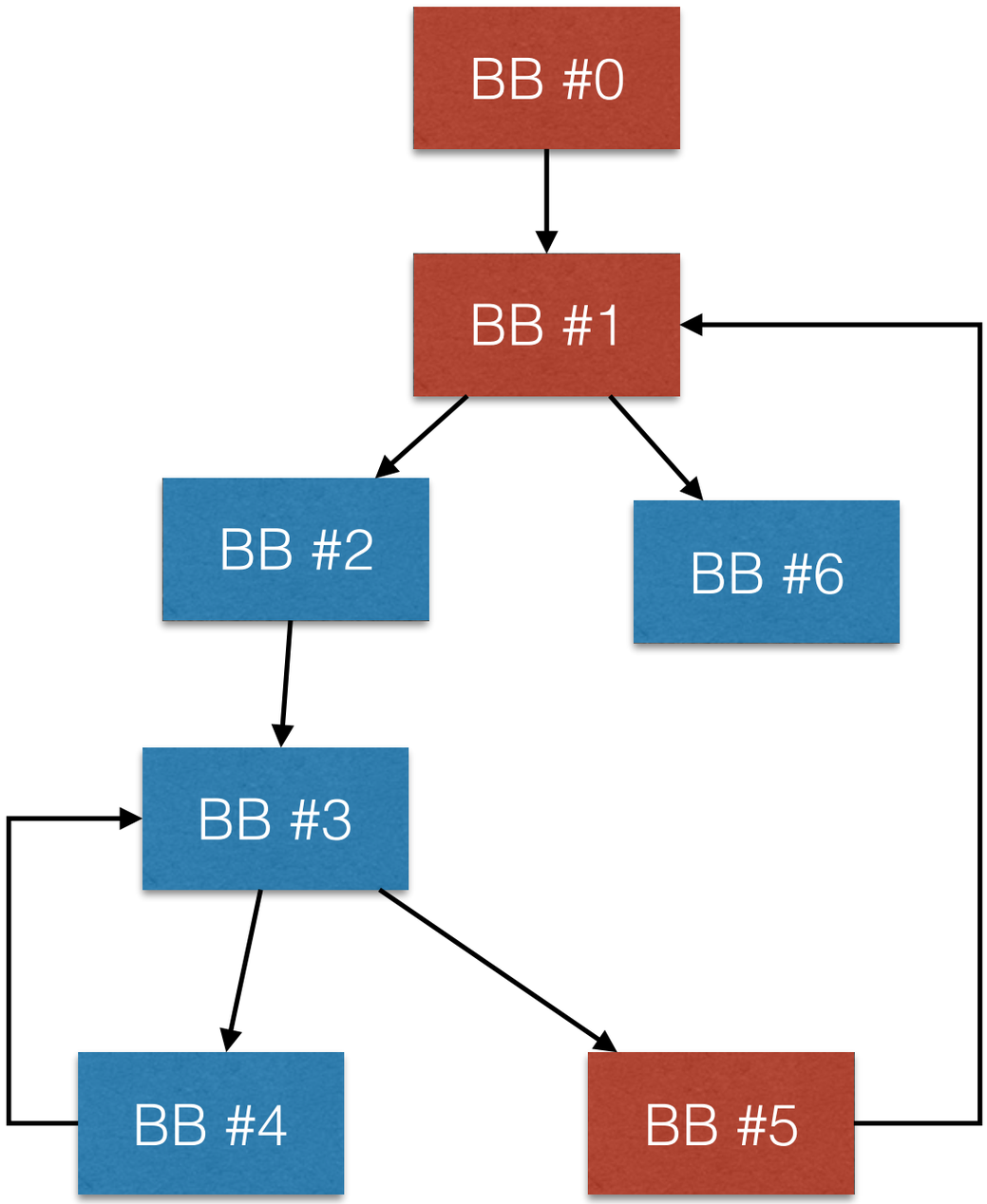

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8: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



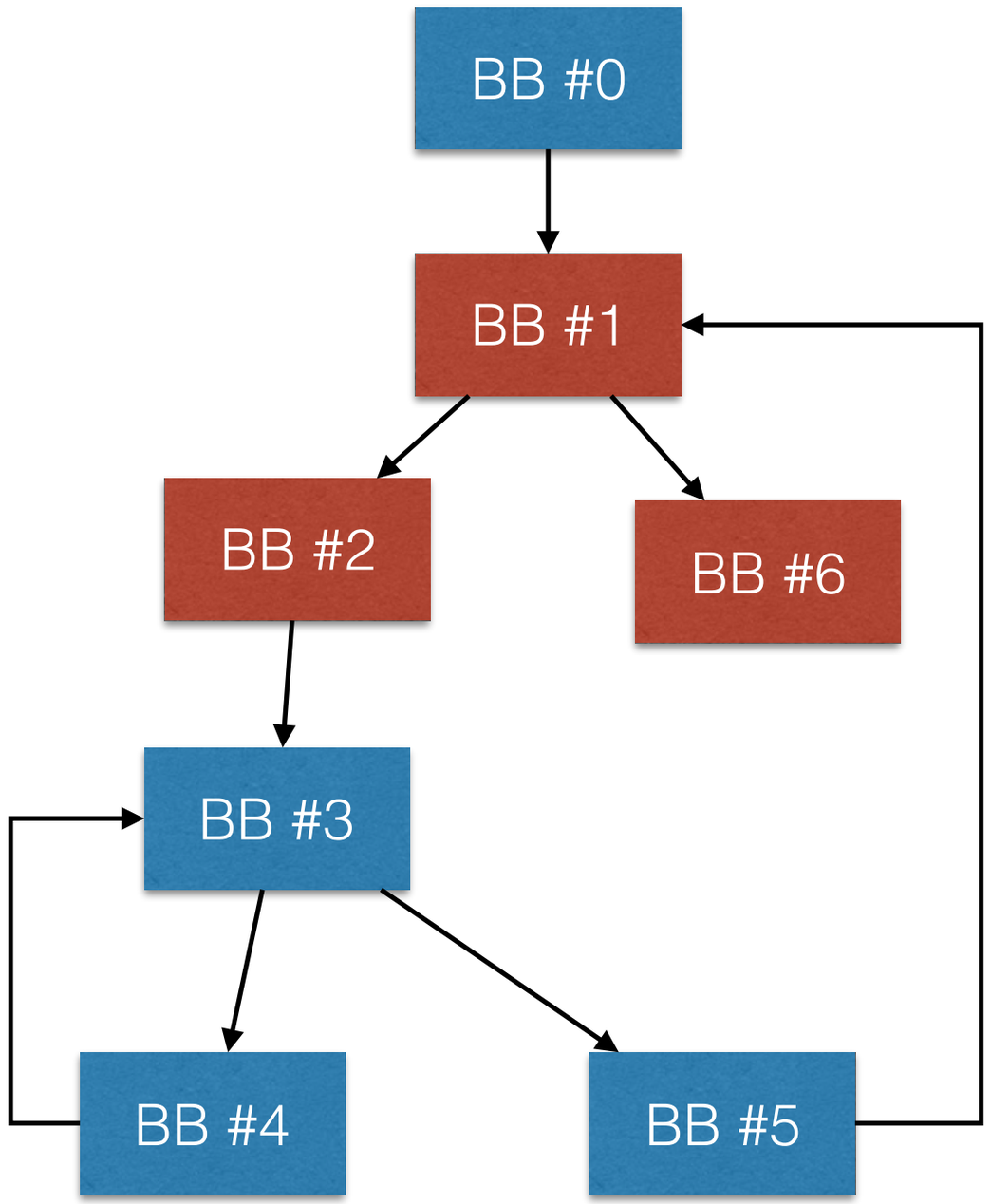

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8: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



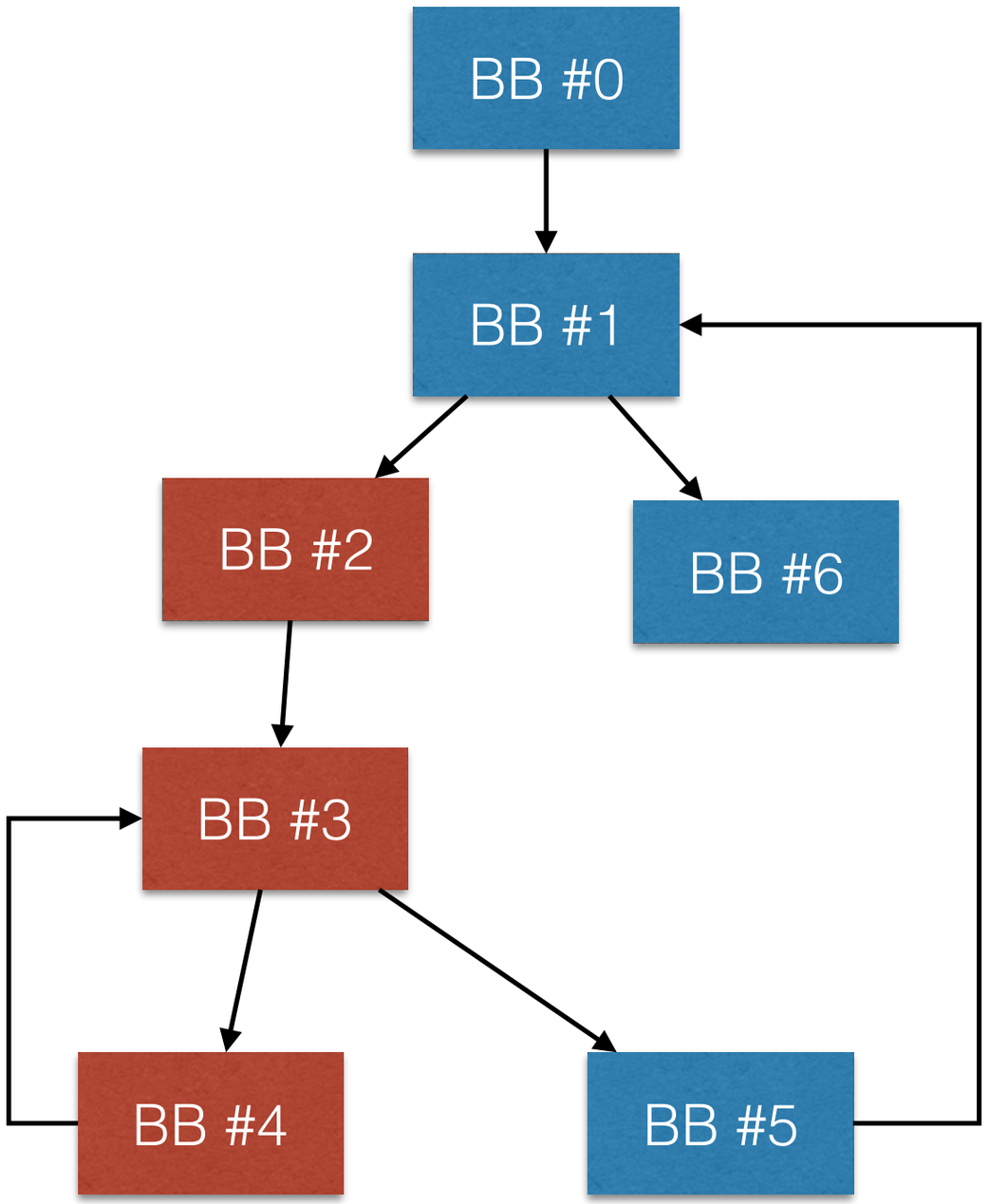

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8: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



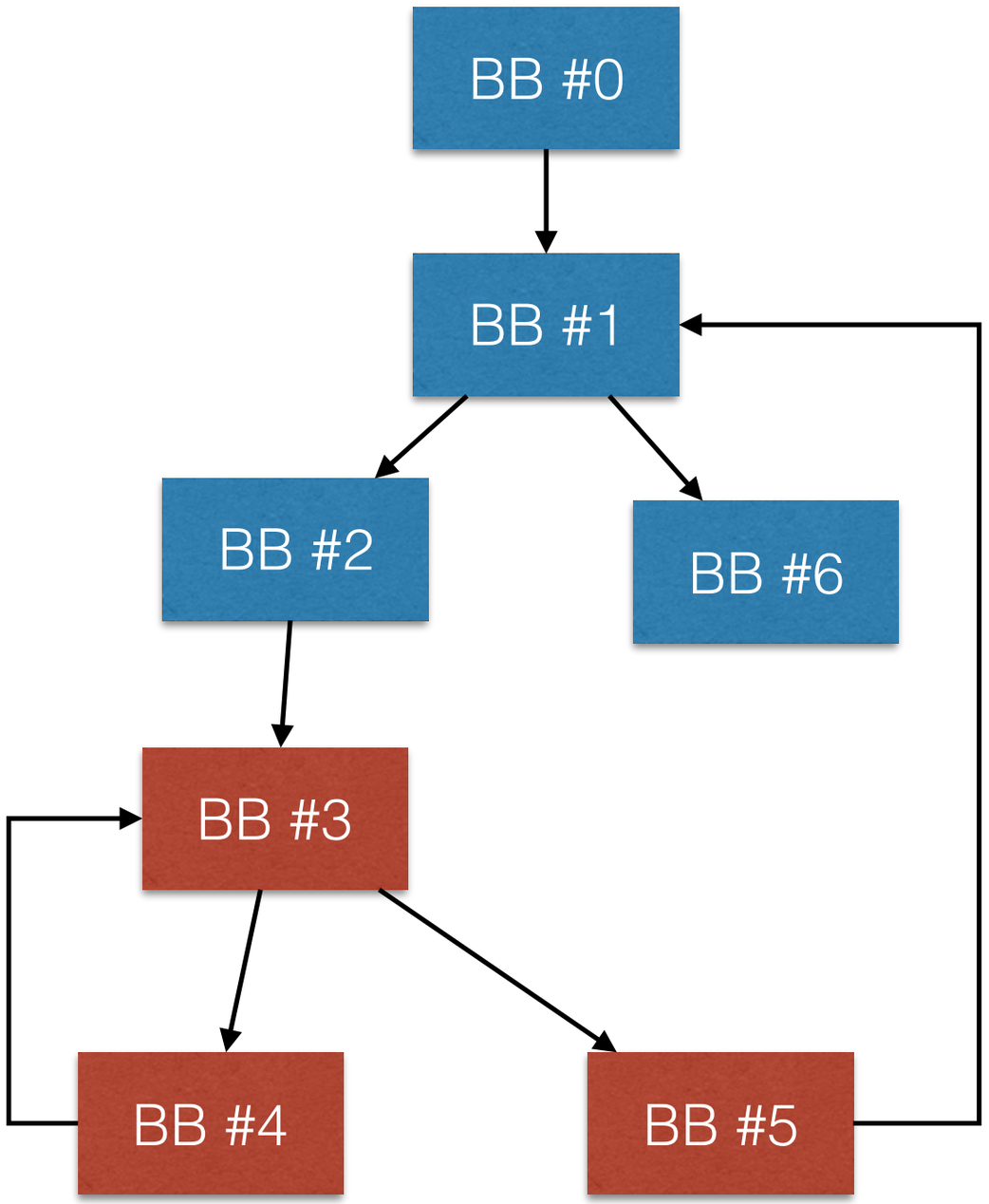

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8: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



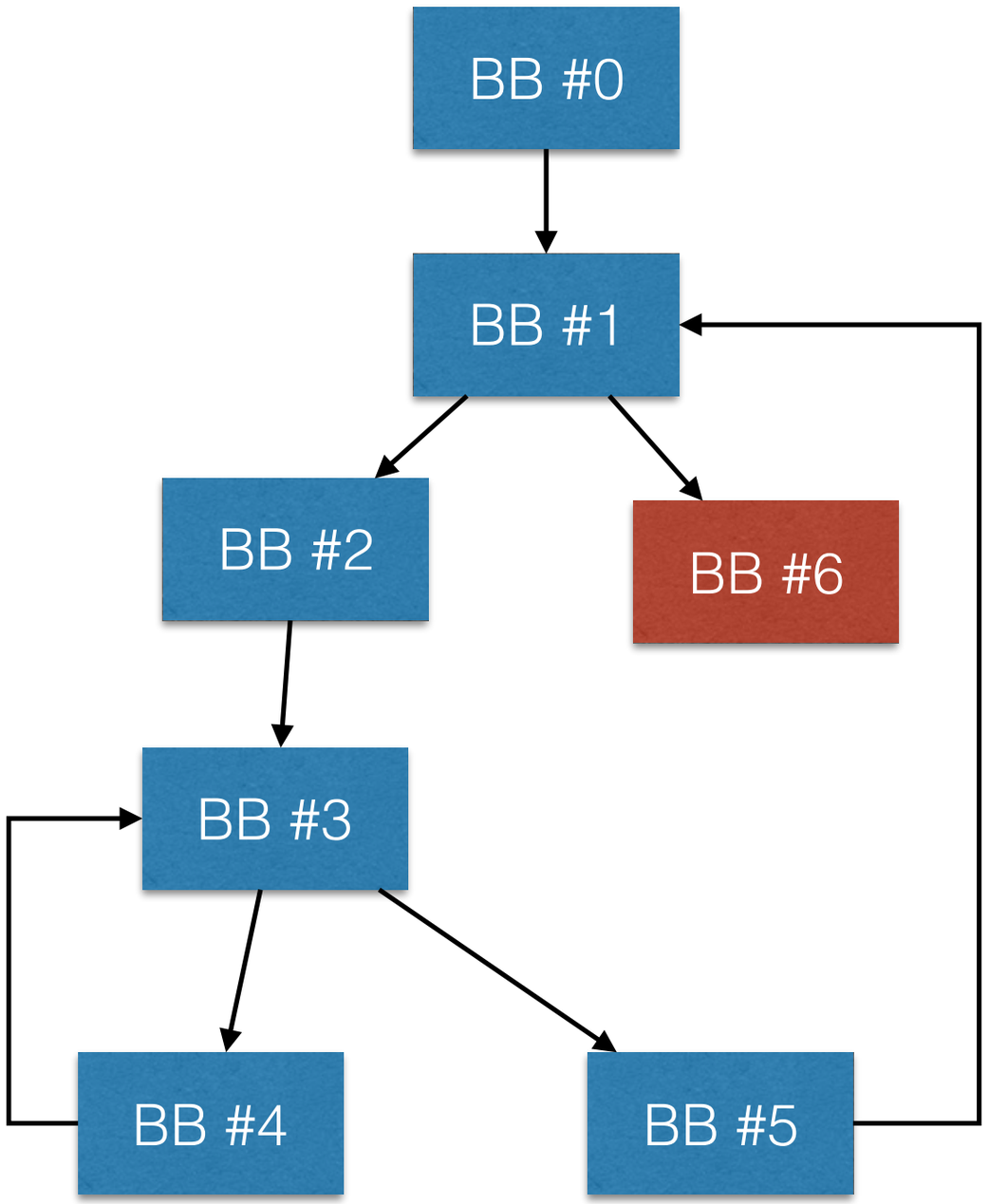

```
EC:
(BB#0, in)  Bundle #0:  0   0   0
(BB#0, out) Bundle #1:  1   1   1
(BB#1, in)  Bundle #2:  2   1   1
(BB#1, out) Bundle #3:  3   3   2
(BB#2, in)  Bundle #4:  4   3   2
(BB#2, out) Bundle #5:  5   5   3
(BB#3, in)  Bundle #6:  6   5   3
(BB#3, out) Bundle #7:  7   7   4
(BB#4, in)  Bundle #8: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

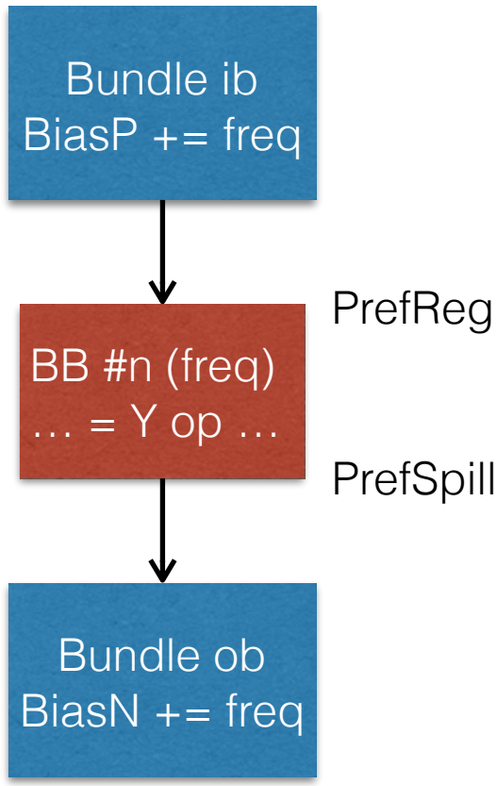
Edge Bundle




```
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:
```

SpillPlacement::addConstraints

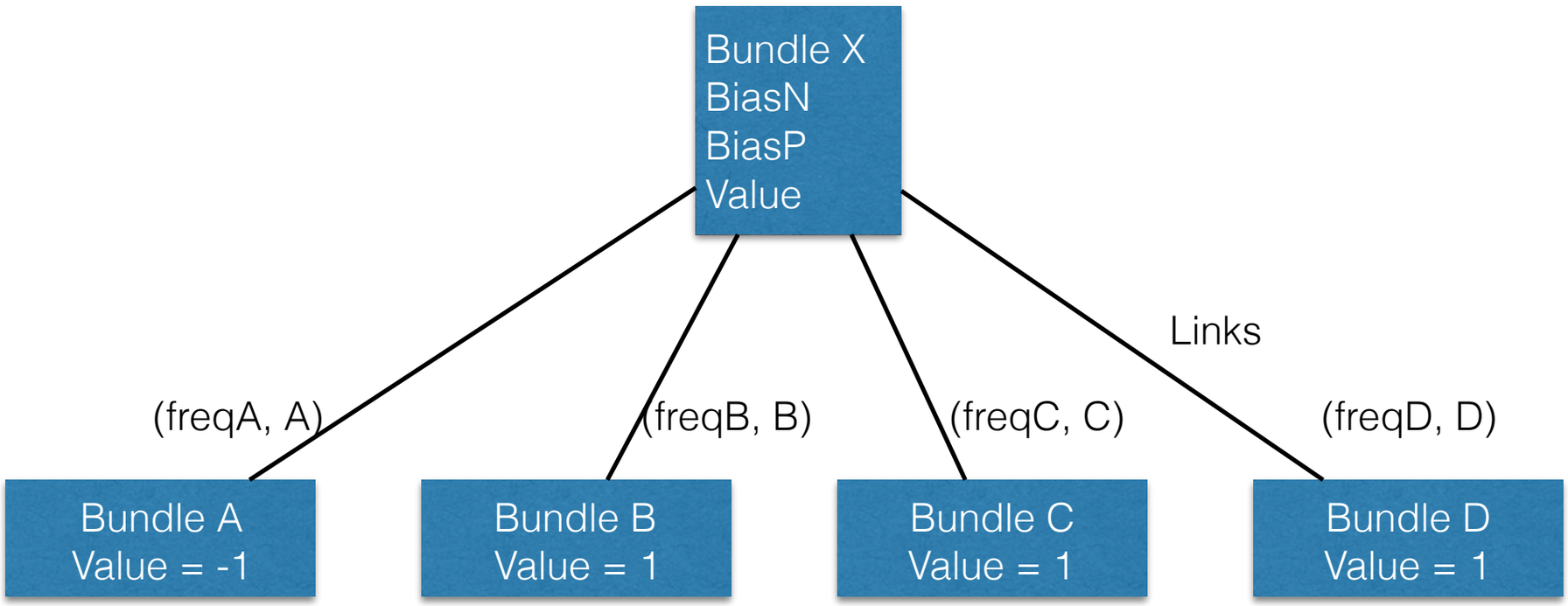
- update BiasN, BiasP according to BorderConstraint




```
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;
  }
}
```

Hopfield Network Node

- Node.update(nodes, Threshold)

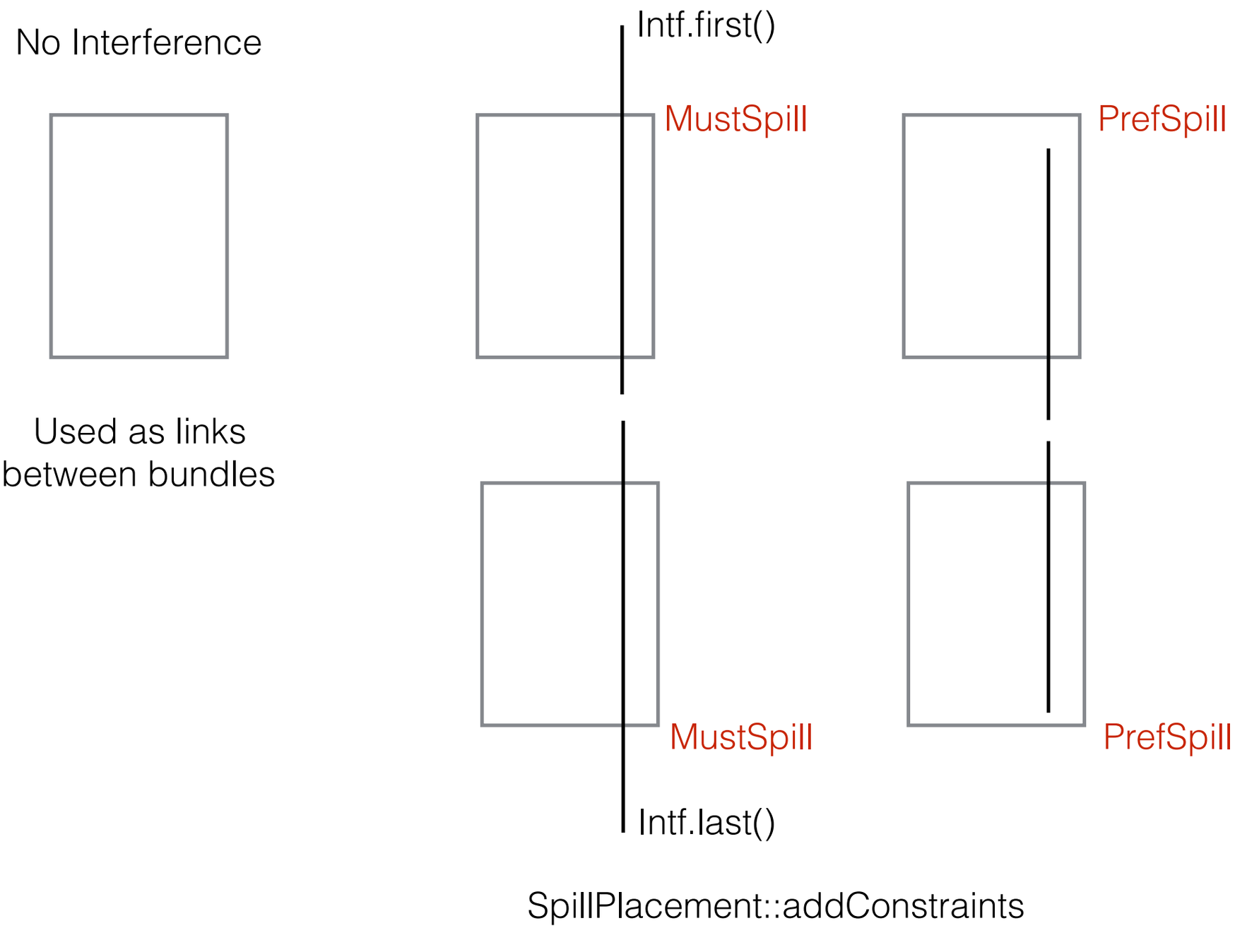



```
SumN = BiasN + freqA
SunP = BiasP + freqB + freqC + freqD
```

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

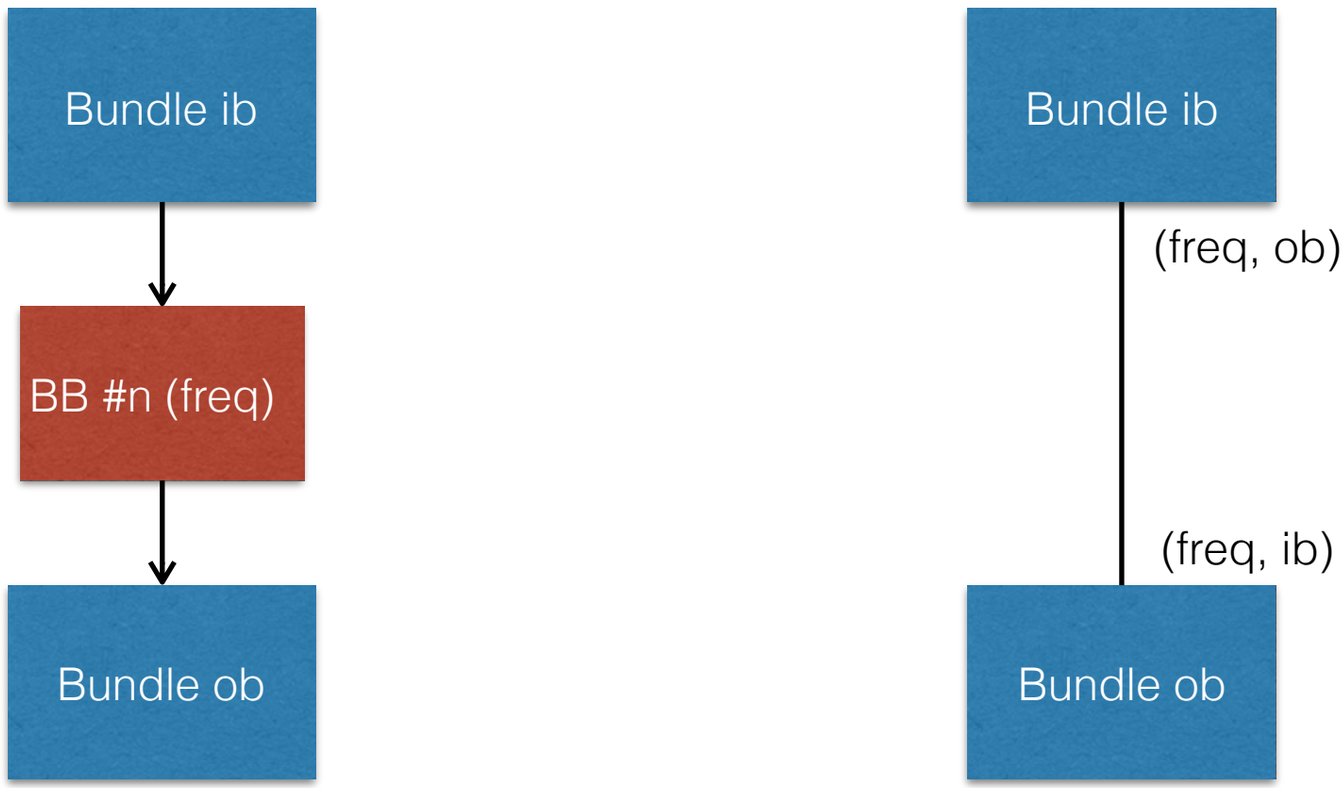
Grow Region

- Live through blocks in positive bundles.



SpillPlacement::addConstraints

SpillPlacement::addLinks

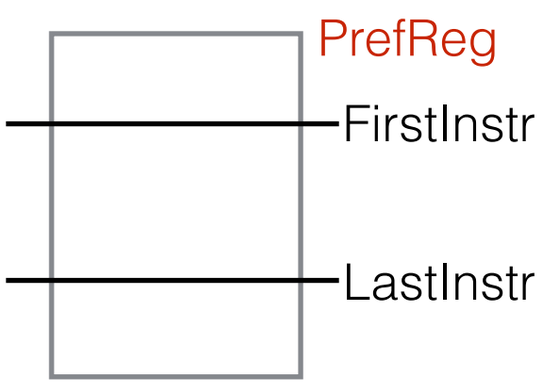


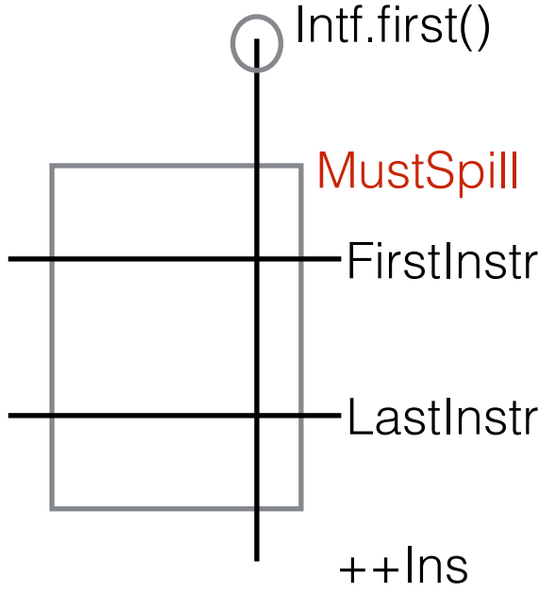
SpillPlacement::iterate

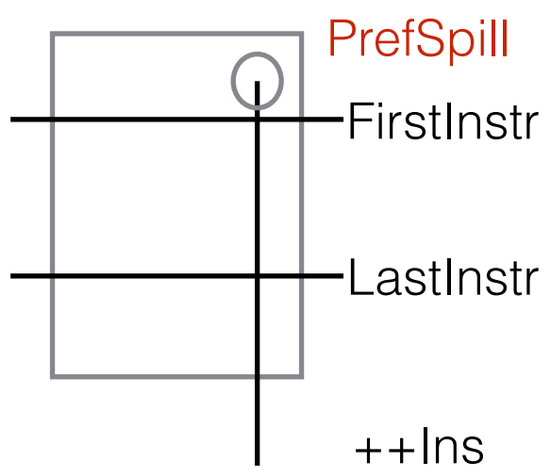
```
for (unsigned iteration = 0; iteration != 10; ++iteration) {
  bool Changed = false;
  for (SmallVectorImpl<unsigned>::const_reverse_iterator I =
         iteration == 0 ? Linked.rbegin() : std::next(Linked.rbegin()),
         E = Linked.rend(); I != E; ++I) {
    unsigned n = *I;
    if (nodes[n].update(nodes, Threshold)) {
      Changed = true;
      if (nodes[n].preferReg())
        RecentPositive.push_back(n);
    }
  }
  if (!Changed || !RecentPositive.empty())
    return;

  Changed = false;
  for (SmallVectorImpl<unsigned>::const_iterator I =
         std::next(Linked.begin()), E = Linked.end(); I != E; ++I) {
    unsigned n = *I;
    if (nodes[n].update(nodes, Threshold)) {
      Changed = true;
      if (nodes[n].preferReg())
        RecentPositive.push_back(n);
    }
  }
  if (!Changed || !RecentPositive.empty())
    return;
}
```

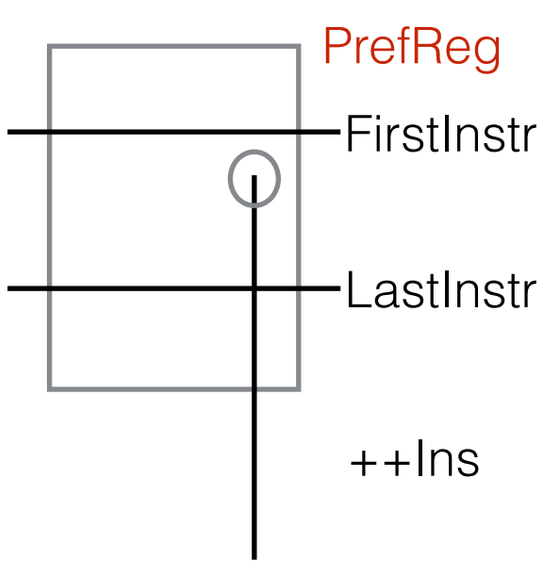
Spill Cost

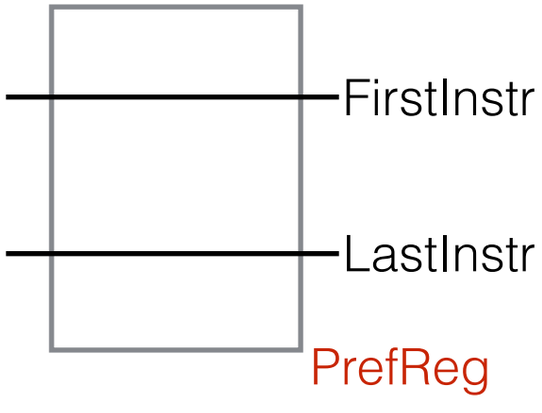
No Interference

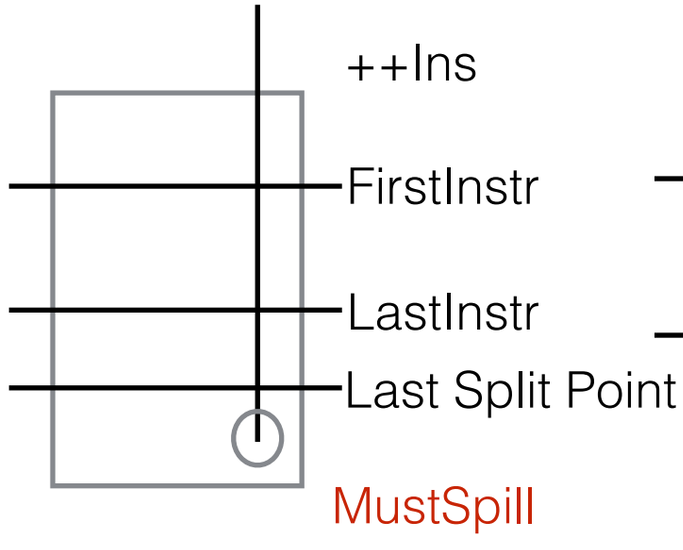


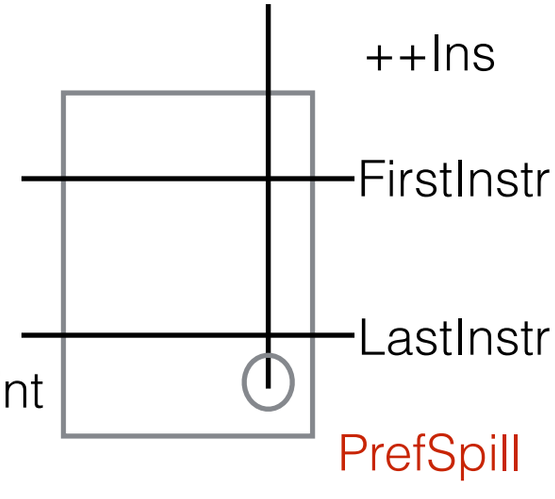


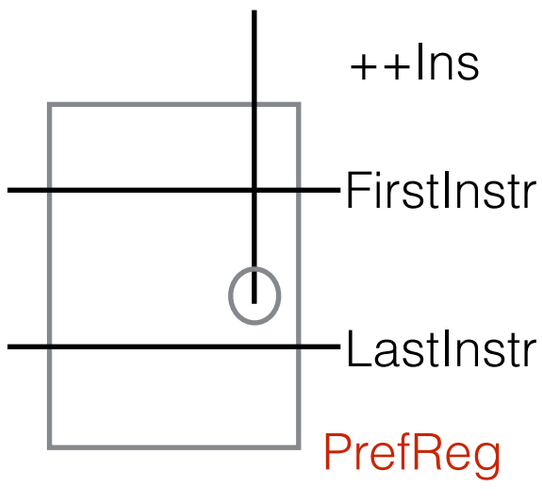








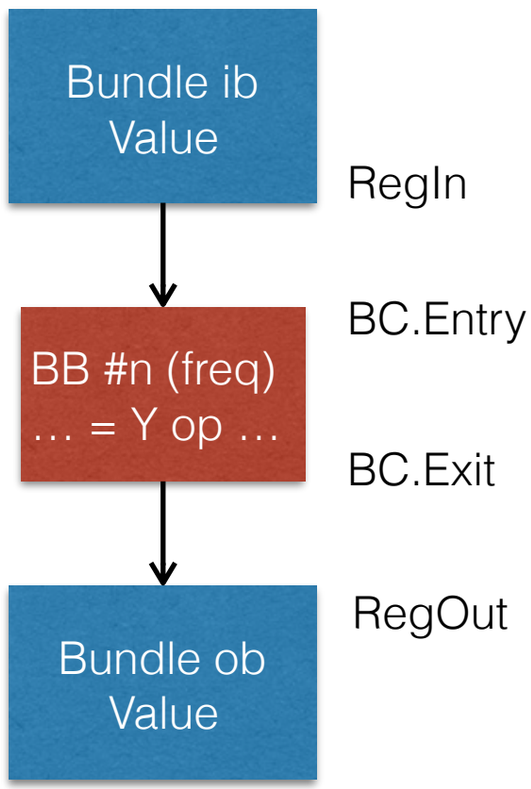




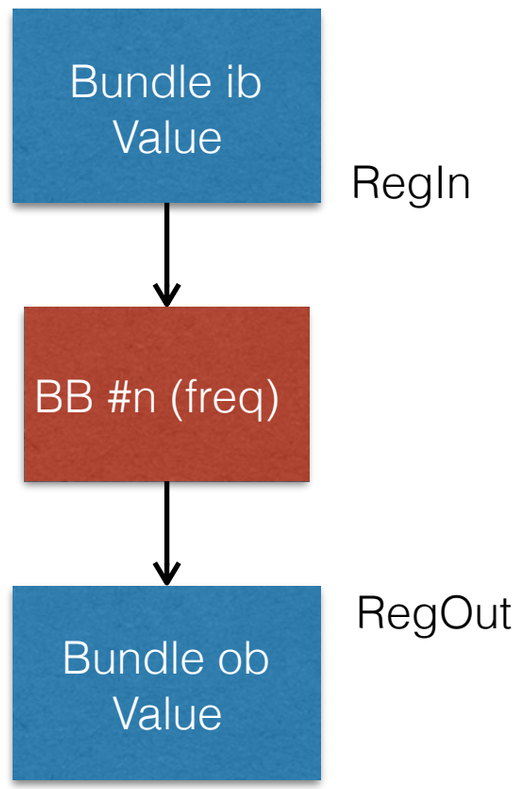
Cost = Block_Frequency * Ins

Split Cost

Use Block



Live Through



RegIn	RegOut	Cost
0	0	0
0	1	freq
1	0	freq
1	1	2 x freq (interfer)

```
if (BI.LiveIn)
  Ins += RegIn != (BC.Entry == SpillPlacement::PrefReg);
if (BI.LiveOut)
  Ins += RegOut != (BC.Exit == SpillPlacement::PrefReg);
while (Ins--)
  GlobalCost += SpillPlacer->getBlockFrequency(BC.Number);
```

The Best Candidate

- For all physical registers, calculate region split cost.
- Cost = block constraints cost (spill cost) + global split cost
- The best candidate has the lowest cost.

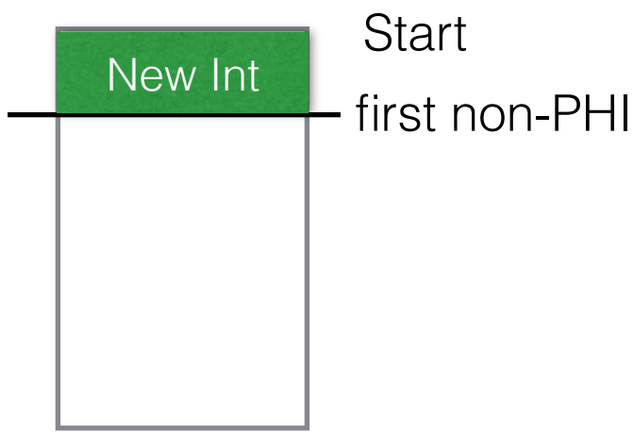
Split

- splitLiveThroughBlock
- splitRegInBlock
- splitRegOutBlock

splitLiveThroughBlock

Live Through
LiveOut on Stack

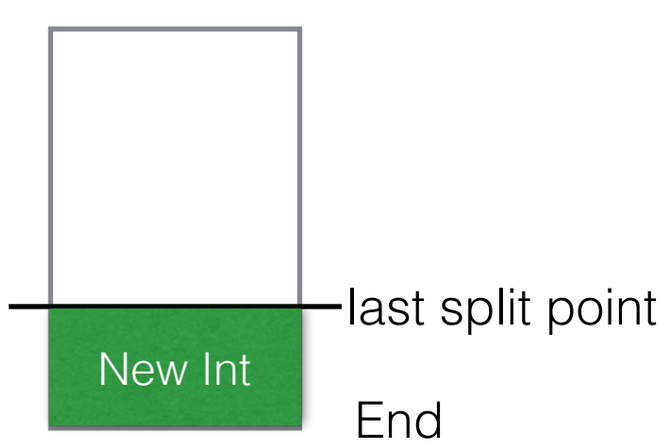
Bundle ib
Value == 1



Bundle ob
Value != 1

Live Through
LiveIn on Stack

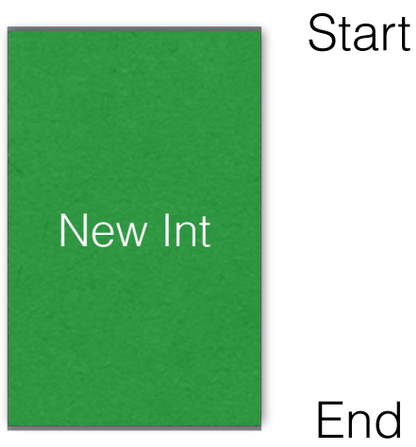
Bundle ib
Value != 1



Bundle ob
Value == 1

Live Through
No Interference

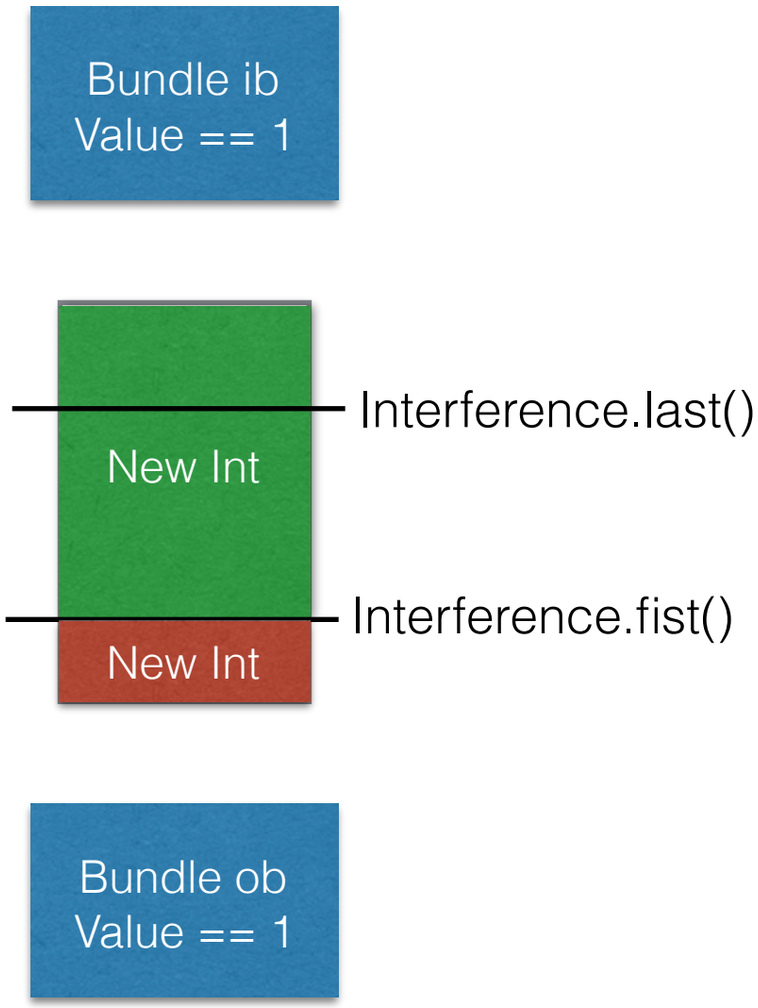
Bundle ib
Value == 1



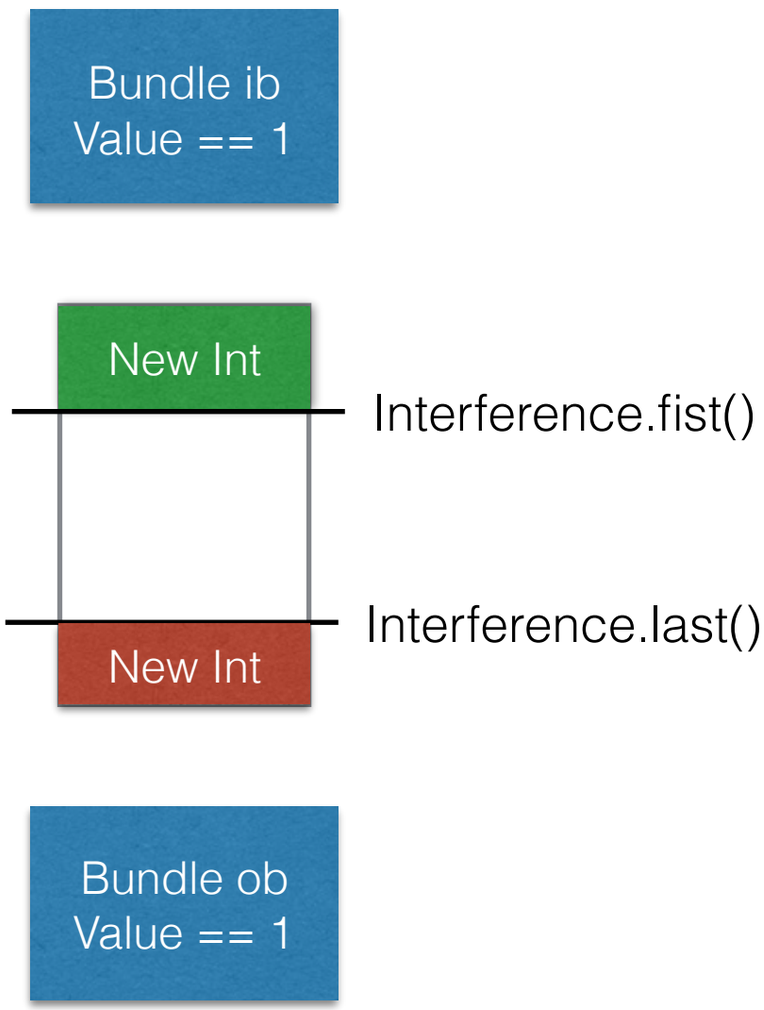
Bundle ob
Value == 1

splitLiveThroughBlock

LiveThrough
Non-overlapping interference



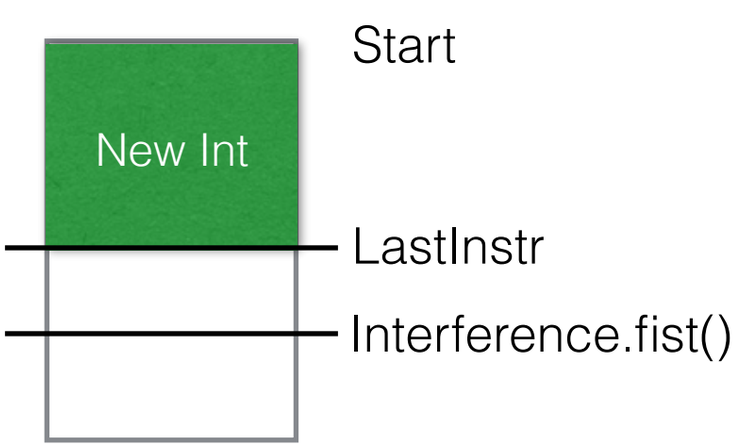
LiveThrough
Overlapping interference



splitRegInBlock

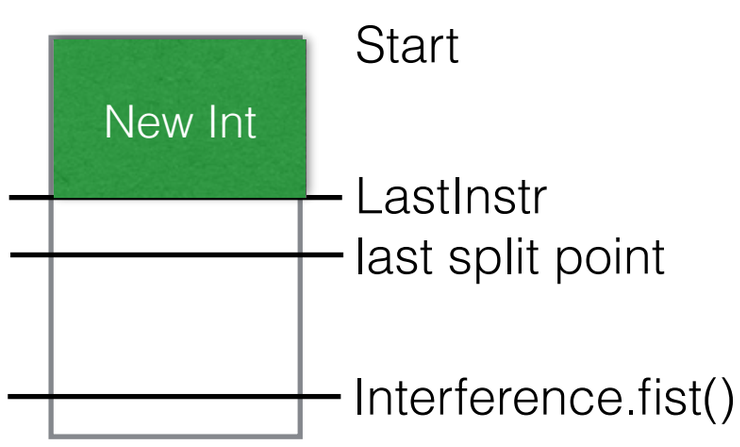
No LiveOut
Interference after kill

Bundle ib
Value == 1



LiveOut on Stack
Interference after last use

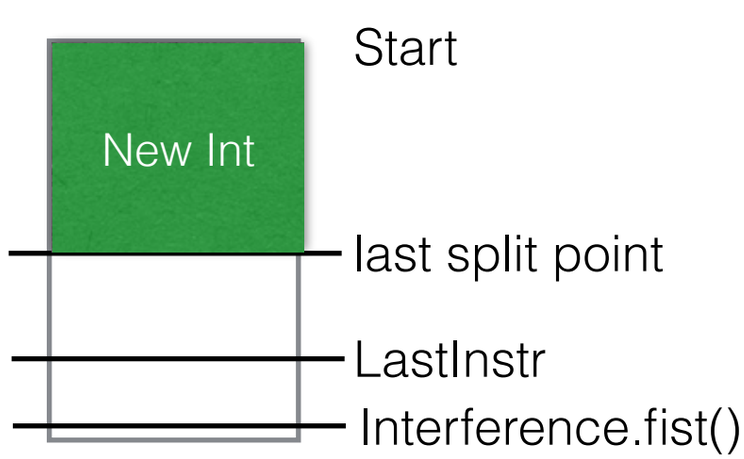
Bundle ib
Value == 1



Bundle ob
Value != 1

LiveOut on Stack
Interference after last use

Bundle ib
Value == 1



Bundle ob
Value != 1

splitRegInBlock

LiveOut on Stack
Interference overlapping uses

Bundle ib
Value == 1

Start

New Int

Interference.fist()

New Int

LastInstr

last split point

Bundle ob
Value != 1

LiveOut on Stack
Interference overlapping uses

Bundle ib
Value == 1

Start

New Int

Interference.fist()

New Int

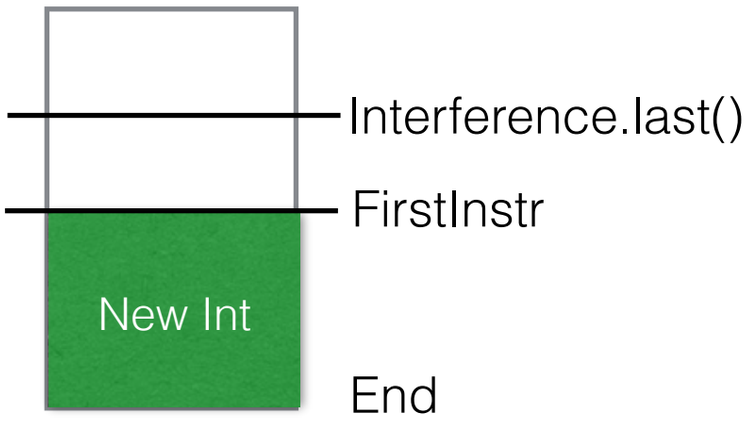
last split point

LastInstr

Bundle ob
Value != 1

splitRegOutBlock

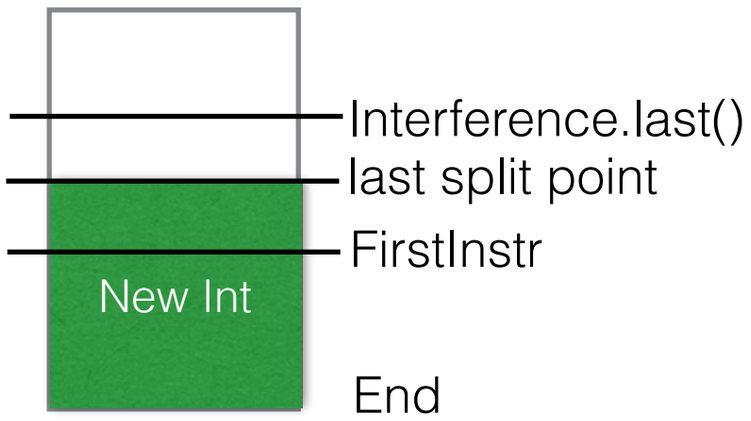
No LiveIn
Interference before def



Bundle ob
Value == 1

Live Through
Interference before def

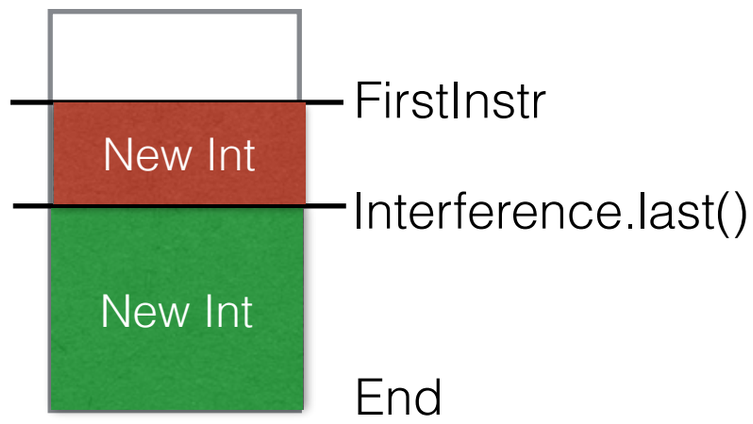
Bundle ib
Value != 1



Bundle ob
Value == 1

Live Through
Interference overlapping uses

Bundle ib
Value != 1



Bundle ob
Value == 1