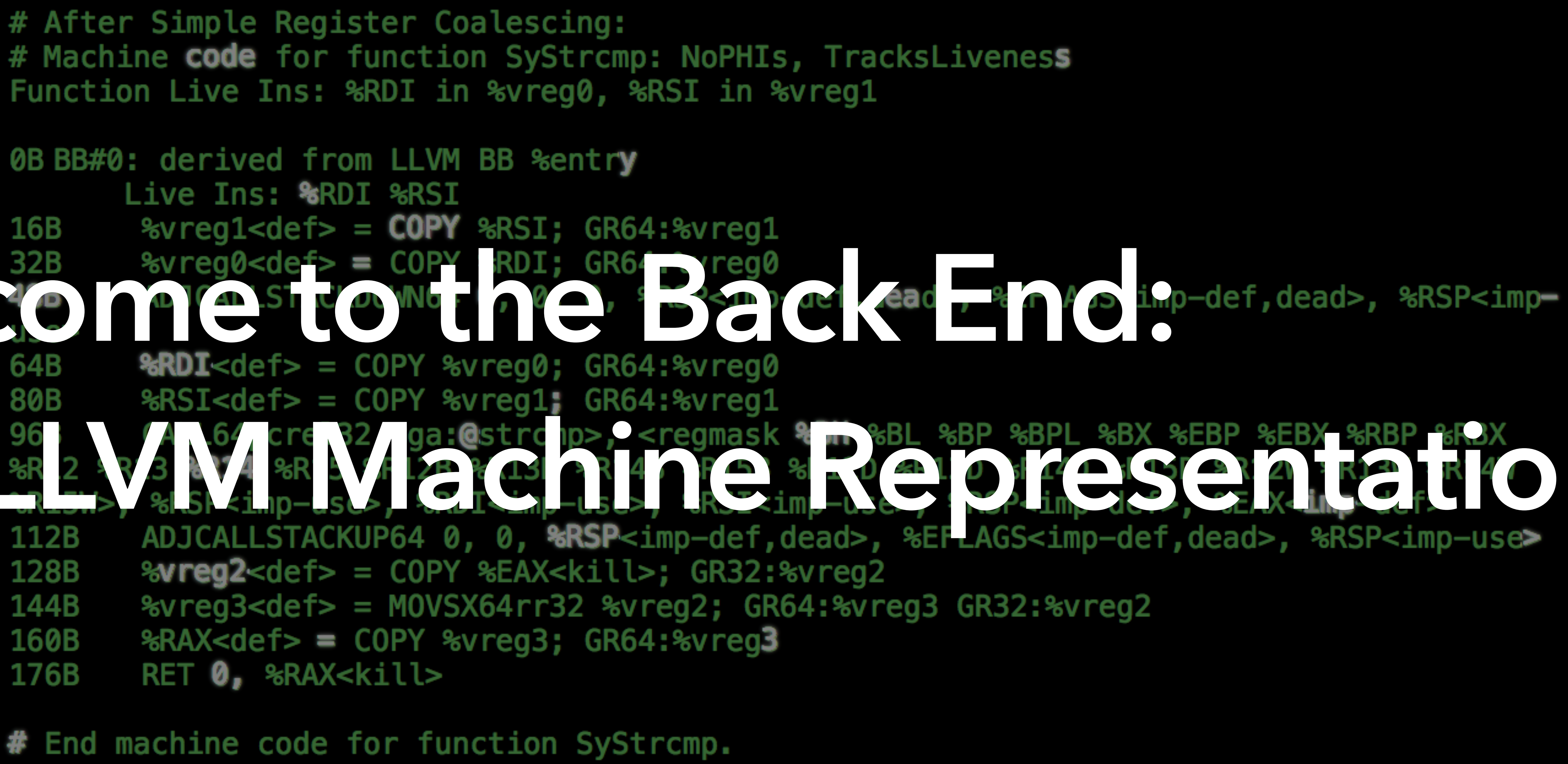
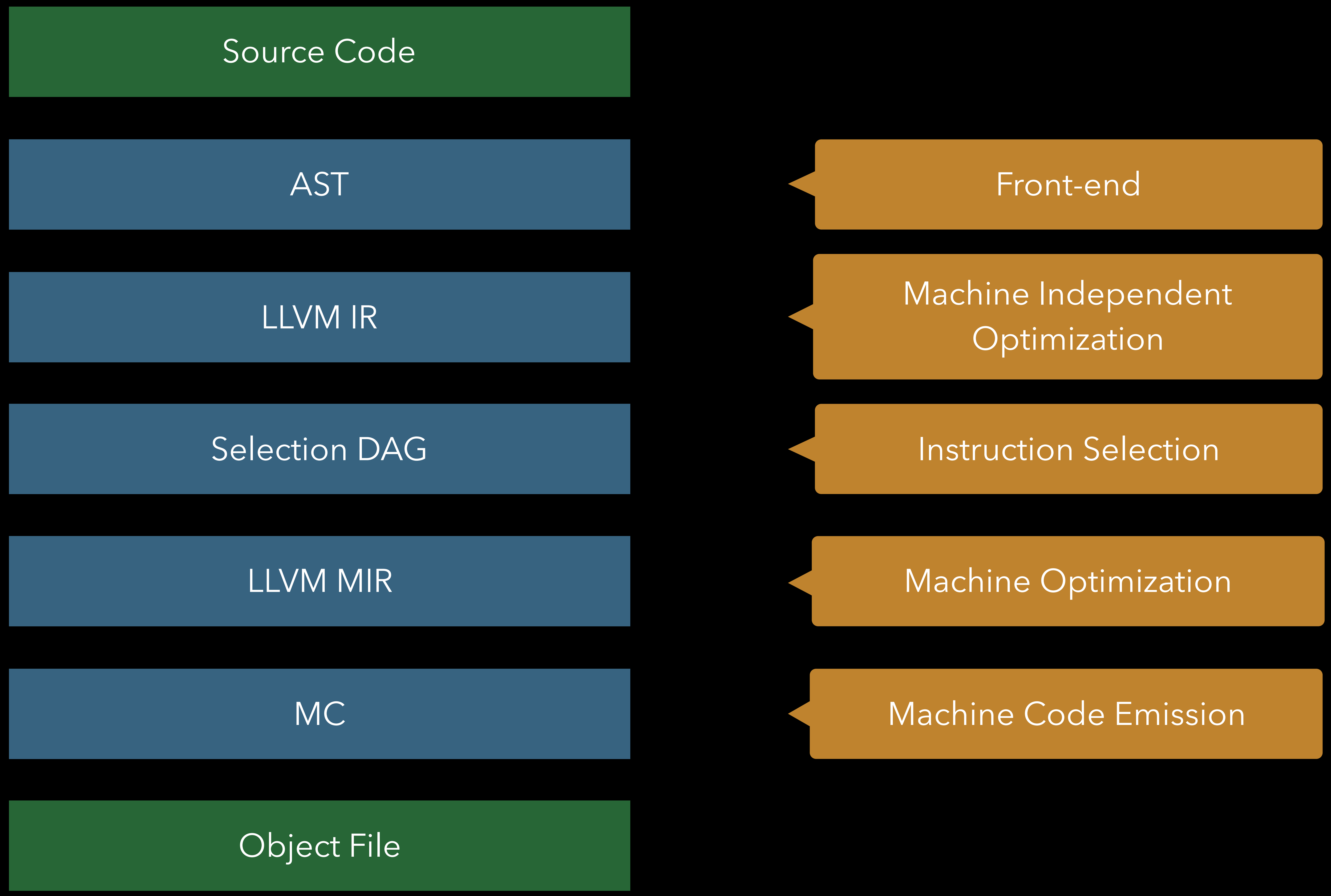
Welcome to the Back End: The LLVM Machine Representation



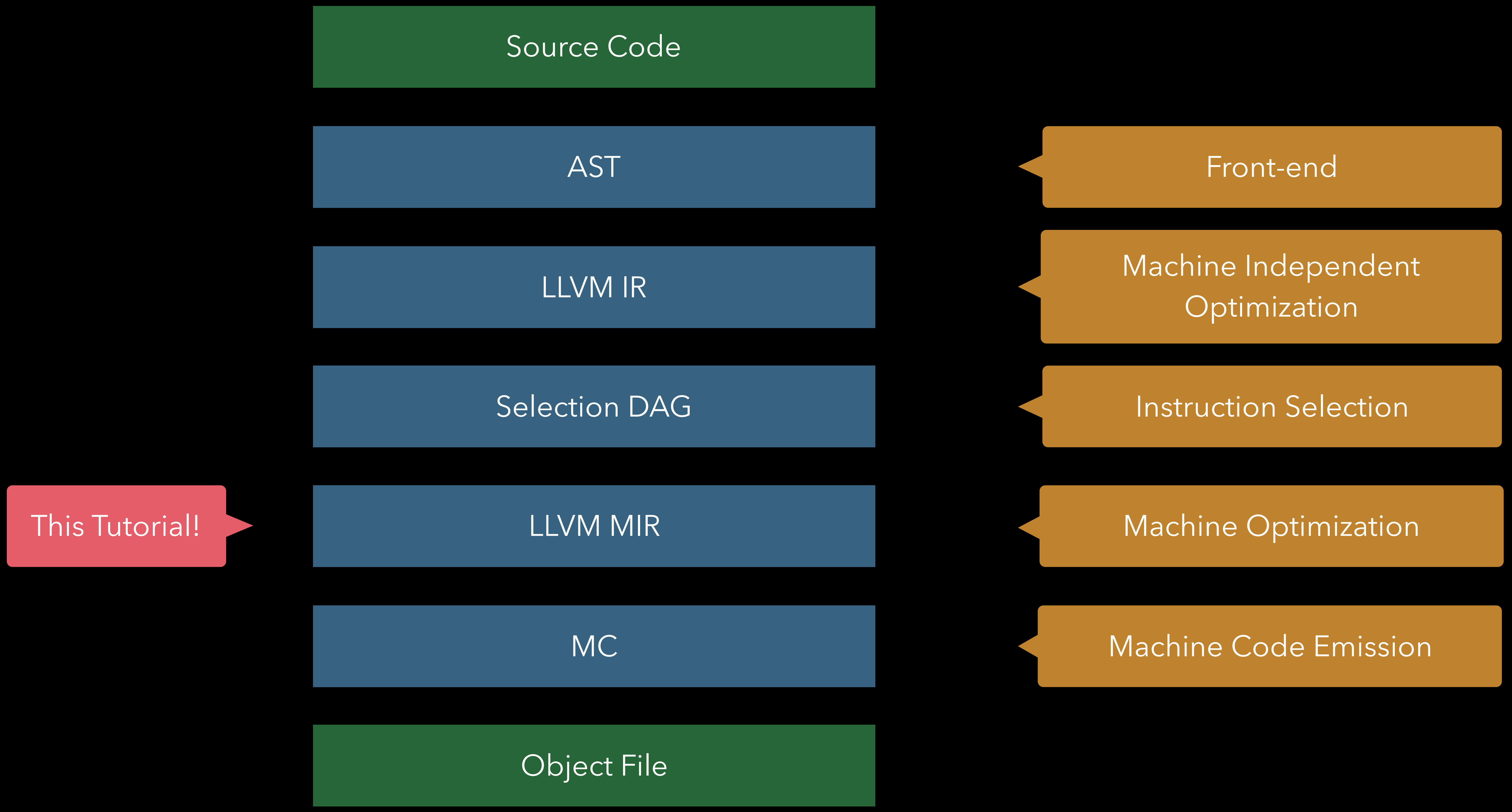
```
# After Simple Register Coalescing:
# Machine code for function SyStrcmp: NoPHIs, TracksLiveness
Function Live Ins: %RDI in %vreg0, %RSI in %vreg1

0B BB#0: derived from LLVM BB %entry
        Live Ins: %RDI %RSI
16B      %vreg1<def> = COPY %RSI; GR64:%vreg1
32B      %vreg0<def> = COPY %RDI; GR64:%vreg0
[illegible] imp-def,dead>, %RSP<imp-
[illegible]
64B      %RDI<def> = COPY %vreg0; GR64:%vreg0
80B      %RSI<def> = COPY %vreg1; GR64:%vreg1
96[illegible] <regmask [illegible] %BL %BP %BPL %BX %EBP %EBX %RBP %RBX
[illegible]
[illegible]
112B     ADJCALLSTACKUP64 0, 0, %RSP<imp-def,dead>, %EFLAGS<imp-def,dead>, %RSP<imp-use>
128B     %vreg2<def> = COPY %EAX<kill>; GR32:%vreg2
144B     %vreg3<def> = MOVSX64rr32 %vreg2; GR64:%vreg3 GR32:%vreg2
160B     %RAX<def> = COPY %vreg3; GR64:%vreg3
176B     RET 0, %RAX<kill>

# End machine code for function SyStrcmp.
```

Matthias Braun, Apple

Program Representations



Program Representations



LLVM MIR

- Machine Specific Instructions
- Tasks
  - Resource Allocation: Registers, Stack Space, …
  - Lowering: ABI, Exception Handling, Debug Info, …
  - Optimization: Peephole, Instruction/Block Scheduling, …
- Tighten constraints along pass pipeline

A Tutorial on the LLVM Machine Representation

Part 1: The Basics

- Introduce data structures, important passes
- Usage examples, debugging tips

Part 2: Register Allocation

- Registers, register operand flags, Highlight assumptions, and constraints
- Liveness analysis before and after allocation, frame lowering scavenging

Part I: The Basics

Writing an LLVM Target

- Implement `TargetMachine` interface:

```
class TargetMachine { // ...
  const MCAsmInfo *getMCAsmInfo() const { return AsmInfo; }
  // ... more getMCxxx() functions
  virtual bool addPassesToEmitFile(PassManagerBase &, raw_pwrite_stream &,
                                   /*...*/);
  virtual bool addPassesToEmitMC(PassManagerBase &, MCContext *&,
                                 raw_pwrite_stream &, /*...*/);
  virtual TargetPassConfig *createPassConfig(PassManagerBase &PM);
  virtual const TargetSubtargetInfo *getSubtargetImpl(const Function &);
};
```

Code Generation Pipeline

MachineSSA

- ExpandISelPseudo
- MachineLICM
- MachineCSE
- MachineSink
- PeepholeOptimizer
- DeadMachineInstrElim

Optimized RegAlloc

- PHIElimination
- TwoAddressInstruction
- RegisterCoalescer
- MachineScheduler
- RegAllocGreedy
- VirtRegRewriter
- StackSlotColoring

Late Passes, Emission

- ShrinkWrap
- PrologEpilogInserter
- ExpandPostRAPseudos
- PostMachineScheduler
- BlockPlacement
- LiveDebugValues
- AsmPrinter

Simplified Picture

Pass Manager Setup

- Pass manager pipeline is setup in `TargetPassConfig`
- Target overrides methods to add, remove or replace passes.

```
class TargetPassConfig { // ...
  virtual void addMachinePasses();
  virtual void addMachineSSAOptimization();
  virtual bool addPreISel() { return false; }
  virtual bool addILPOpts() { return false; }
  virtual void addPreRegAlloc() {}
  virtual bool addPreRewrite() { return false; }
  virtual void addPostRegAlloc() {}
  void addPass(Pass *P, bool verifyAfter = true, bool printAfter = true);
};
```

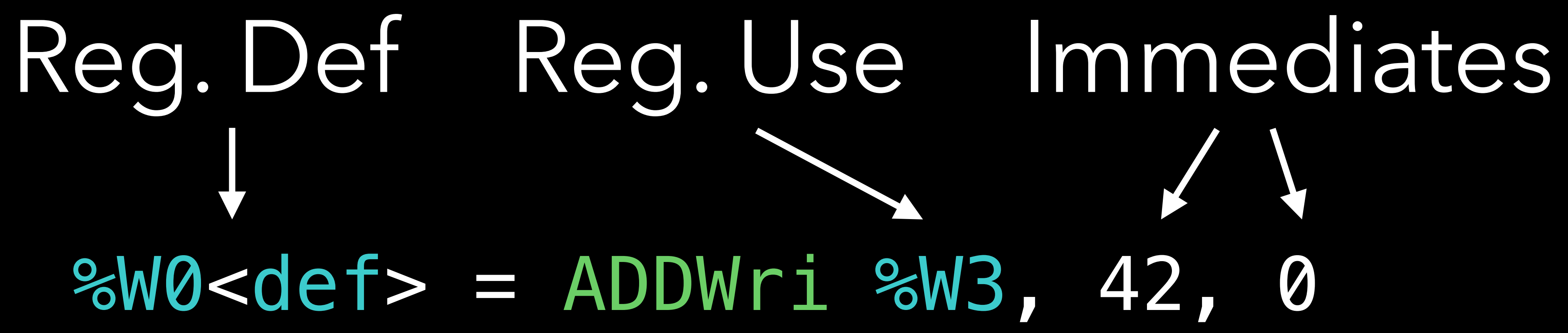
- There is also `insertPass` and `substitutePass`.

Instructions

- `class MachineInstruction` (MI)
- Opcode
- Pointer to Machine Basic Block
- Operand Array; Memory Operand Array
- Debugging Location

Operands

- `class MachineOperand` (MOP)
- Register, RegisterMask
- Immediates
- Indexes: Frame, ConstantPool, Target…
- Addresses: ExternalSymbol, BlockAddress, …
- Predicate, Metadata, …



```
%W0<def> = ADDWri %W3, 42, 0
```

Opcodes

- `class MCInstrDesc`
  Opcode/Instruction Description
- Describes operand types, register classes
- Flags describing instruction:
  - Format
  - Semantic
  - Filter for target callbacks
  - Side Effects
  - Transformation Hints/Constraints

Variadic hasOptionalDef RegSequence
Return Barrier Terminator Branch Add
IndirectBranch MoveImm Bitcast Select
Compare Call DelaySlot Commutable
FoldableAsLoad ConvertibleTo3Addr
Rematerializable UsesCustomInserter
HasPostISelHook Pseudo MayLoad
UnmodeledSideEffects MayStore
CheapAsAMove ExtraSrcRegAllocReq
ExtraDefRegAllocReq NotDuplicable
Convergent Predicable

Basic Blocks

- `class MachineBasicBlock` (MBB)
- List of instructions, sequentially executed from beginning; branches at the end.
- PHIs come first, terminators last
- Double Linked List of Instructions
- Arrays with predecessor/successor blocks with execution frequency
- Pointer to Machine Function and IR Basic Block
- Numbered (for dense arrays)

Functions

- `class MachineFunction` (MF)
- Double Linked List of Basic Blocks
- Pointers to IR `Function`, `TargetMachine`, `TargetSubtargetInfo`, `MCContext`, …
- State: `MachineRegisterInfo`, `MachineFrameInfo`, `MachineConstantPool`, `MachineJumpTableInfo`, …

Example: Print All Instructions

```
void SimplePrinter::runOnMachineFunction(MachineFunction &MF) {
  for (MachineBasicBlock &MBB : MF)
    for (MachineInstruction &MI : MBB)
      out() << MI;
}
```

Example: Print Registers Used

```
void PrintUsedRegisters::runOnMachineFunction(MachineFunction &MF) {
  DenseSet<unsigned> Registers;

  // Fill set with used registers.
  for (MachineBasicBlock &MBB : MF)
    for (MachineInstruction &MI : MBB)
      for (MachineOperand &MO : MI.operands())
        if (MO.isReg())
          Registers.insert(MO.getReg());

  // Print set.
  TargetRegisterInfo *TRI = MF.getSubtarget().getRegsiterInfo();
  for (unsigned Reg : Registers)
    out() << "Register: " << PrintReg(Reg, TRI) << '\n';
}
```

Development Tips

- Produce .ll file, then use llc:

```
$ clang -O1 -S -emit-llvm a.c -o a.ll
$ llc a.ll
```

- Enable debug output:

```
$ llc -debug ...
```

- Debug output for passes `foo` and `bar`:

```
$ llc -debug-only=foo,bar ...
```

- Also available in clang:

```
$ clang -mllvm -debug-only=foo,bar ...
$ clang -mllvm -print-machineinstrs ...
```

Print Instructions after Passes

```
$ llc -print-machineinstrs a.ll
# After Instruction Selection:
# Machine code for function FU: IsSSA, TracksLiveness
BB#0: derived from LLVM BB %0
    Live Ins: %EDI
  %vreg0<def> = COPY %EDI; GR32:%vreg0
  %EAX<def> = COPY %vreg0; GR32:%vreg0
  RET 0, %EAX
# ...
# After Post-RA pseudo instruction expansion pass:
# Machine code for function FU: NoPHIs, TracksLiveness, NoVRegs
BB#0: derived from LLVM BB %0
    Live Ins: %EDI
  %EAX<def> = MOV32rr %EDI<kill>
  RET 0, %EAX<kill>
# ...
```

Testing: MIR File Format

- Stop after pass `isel` and write .mir file:

```
$ llc -stop-after=isel a.ll -o a.mir
```

- Load .mir file, run pass, write .mir file:

```
$ llc -run-pass=machine-scheduler a.mir -o a_scheduled.mir
```

- Load .mir file and start code generation pipeline after pass `isel`:

```
$ llc -start-after=isel a.mir -o a.s
```

⚠ Not all target state serialized yet (`-start-after` often fails)

Writing a Machine Function Pass

```
#define DEBUG_TYPE "mypass"
namespace {
class MyPass : public MachineFunctionPass {
public:
  static char ID;
  MyPass() : MachineFunctionPass(ID) {
    initializeMyPass(*PassRegistry::getPassRegistry());
  }
  bool runOnMachineFunction(MachineFunction &MF) override {
    if (skipFunction(*MF.getFunction()))
      return false;
    // ... do work ...
    return true;
  }
};
}
INITIALIZE_PASS(MyPass, DEBUG_TYPE, "Reticulates Splines", false, false)
```

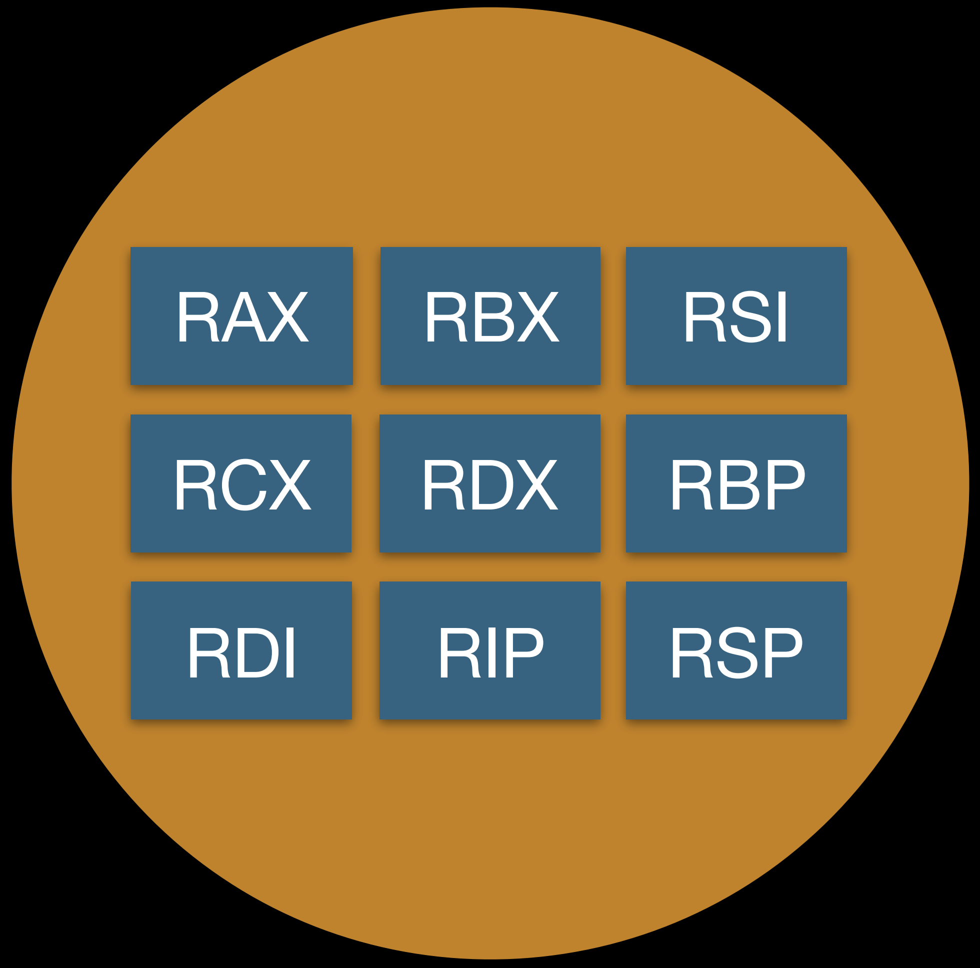
Writing a Machine Module Pass

```
class MyModulePass : public ModulePass { // ...
  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<MachineModuleInfo>();
    AU.setPreservesAll();
    ModulePass::getAnalysisUsage(AU);
  }
  bool runOnMachineFunction(Module &M) override {
    MachineModuleInfo &MMI = getAnalysis<MachineModuleInfo>();
    // Example: Iterate over all machine functions.
    for (Function &F : M) {
      MachineFunction &MF = MMI.getOrCreateMachineFunction(F);
      // ...
    }
    return true;
  }
};
```

Part II: Register Allocation

Physical Registers

- Defined by target, Numbered (`typedef uint16_t MCPhysReg`)
- Can have sub- and super-registers (or arbitrary aliases)
- `MachineRegisterInfo` maintains list of uses and definitions per register
- Register Classes are sets of registers
- Register constraints modeled with classes



X86 GR64_NOREX

Virtual Registers

- Managed by `MachineRegisterInfo`
- Have a register class assigned
- Virtual+Physical Registers stored in `unsigned`
- Differentiate with `TargetRegisterInfo::isVirtualRegister(Reg)` `TargetRegisterInfo::isPhysicalRegister(Reg)`
- Register == 0: No register used (neither physical nor virtual)

Register Operands

Reg. Def   Reg. Use   Immediate

```
%W0<def> = ADDWri %W3, 42, 0
```

AArch64 addition

- `<imp>` Flag: Not emitted
- `<tied>` Flag: Same register for Def+Use (Two Address Code)

```
%EDI<def,tied1> = ROL32rCL %EDI<kill,tied0>, %EFLAGS<imp-def>, %CL<imp-use>
```

X86 rotate left

Assembly:

```
roll %cl, %edi
```

Register Operands

- `<undef>` Flag: Register Value doesn't matter

```
%EAX<def,tied1> = XOR32rr %EAX<undef,tied0>, %EAX<undef>, %EFLAGS<imp-def>
```

X86 XOR / Zero Register

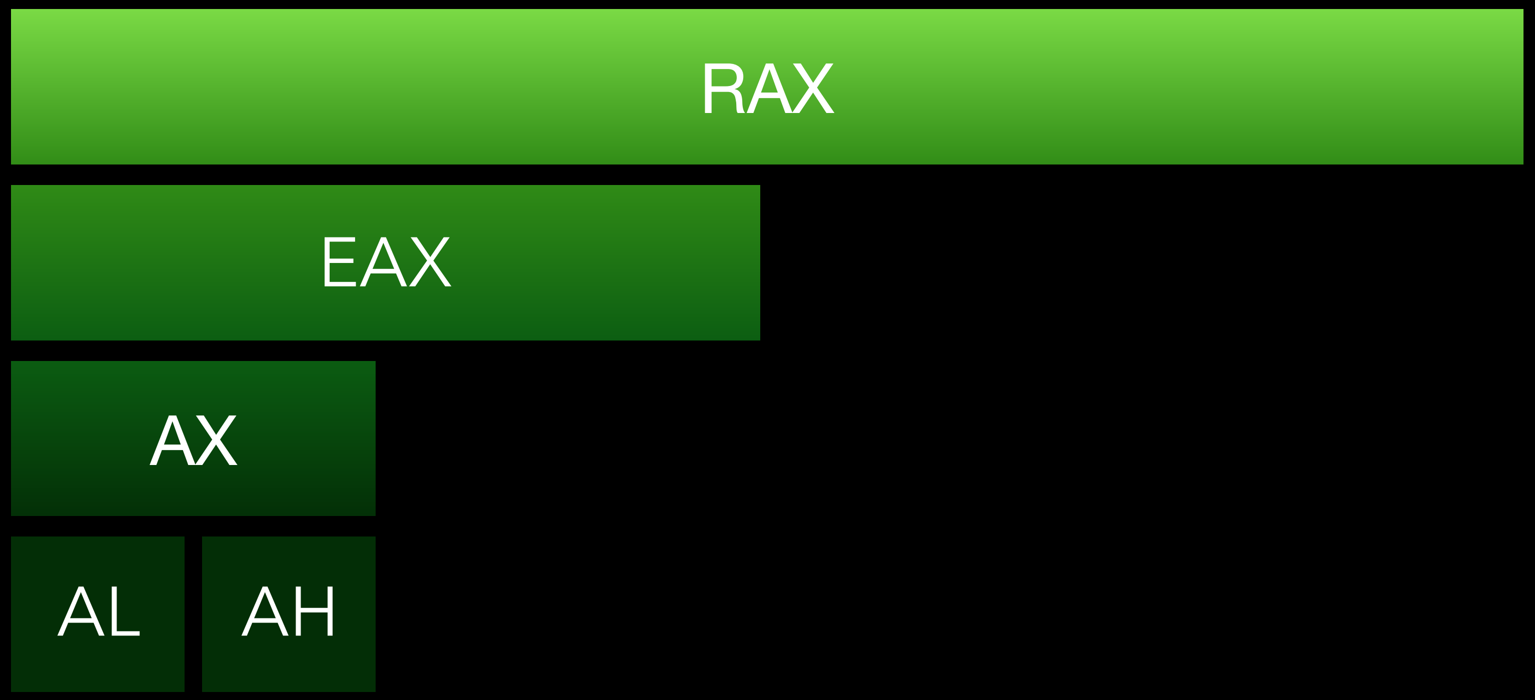
- `<earlyclobber>` Flag: Register definition happens before uses.

Register Operands

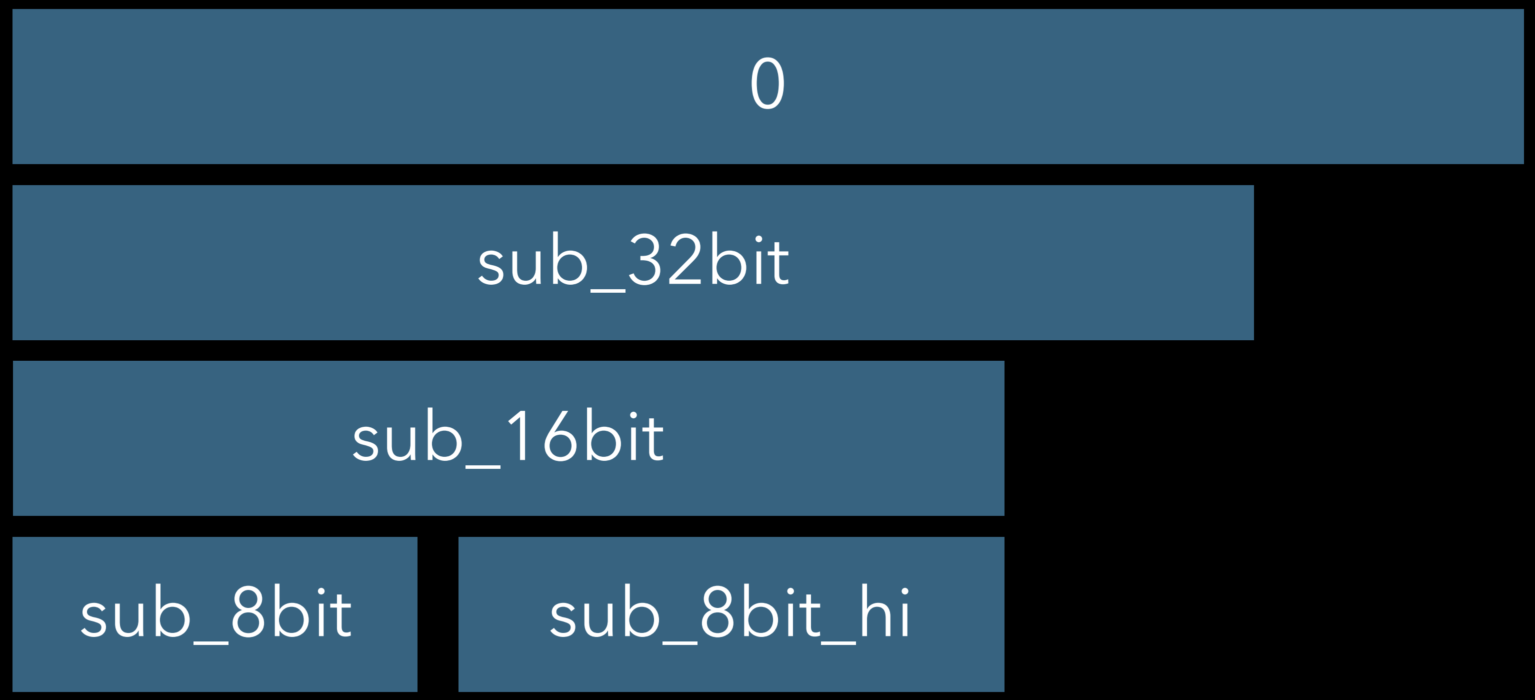
- `:subregindex`: Read/Write part of a virtual register

```
%vreg0<def> = MOV32r0 ...
%vreg0:sub_8bit<def> = SETLr %EFLAGS<imp-use>
```

X86 Set 0/1



X86 GP Registers



Subregister Indexes

⚠ A sub-register write "reads" the other parts (unless it is `<undef>`)

Liveness Indicator Flags

- `<dead>` Flag: Unused definition
- `<kill>` Flag: Last use of a register; ⚠ Optional, Do Not Use

```
int get0(int x, int y) { return x/y; }
```

```
DIV32r %ESI<kill>, %EAX<imp-def>, %EDX<imp-def,dead>,
       %EFLAGS<imp-def,dead>, %EAX<imp-use>, %EDX<imp-use,kill>
```

X86 Division

Register Mask Operands

- `<regmask>`: Bitset of preserved registers, others are clobbered

Global Address Register Mask

```
CALL <ga:@func>, <regmask %LR, %FP, %X19, %X20, ...>, ...
```

- Preserves `%LR`, `%FP`, `%X19` and `%X20`, clobbers every other register

Examples

Legal?

```
%vreg0<def> = OP
 = OP %vreg0
```

```
 = OP %vreg0
```

```
 = OP %vreg1<undef>
```

```
%vreg1<def,dead> = OP
 = OP %vreg1
```

```
%vreg2<def> = OP
 = OP %vreg2<kill>
 = OP %vreg2
```

i Enable verifier with `llc -verify-machineinstrs`

Examples

Legal?

`%vreg0<def> = OP`
` = OP %vreg0`
	✓

` = OP %vreg0`	✖ No definition
` = OP %vreg1<undef>`	✓

`%vreg1<def,dead> = OP`
` = OP %vreg1`
	✖ Use of dead value

`%vreg2<def> = OP`
` = OP %vreg2<kill>`
` = OP %vreg2`
	✖ Use after kill

i Enable verifier with `llc -verify-machineinstrs`

Examples

Legal?

```
%vreg0:sub_8<def> = OP
```

```
%vreg1 = OP
%vreg1:sub_8 = OP
```

```
%vreg2:sub_8<def,undef> = OP
```

```
%vreg3<def> = OP
  = OP %vreg3:sub_8<kill>
  = OP %vreg3:sub_8_hi<kill>
```

```
%sp<def,dead> = ADD %sp, 4
 = OP %sp
```

Examples

Legal?

```
%vreg0:sub_8<def> = OP
```

✖ No definition for rest of register

```
%vreg1 = OP
%vreg1:sub_8 = OP
```

✓

```
%vreg2:sub_8<def,undef> = OP
```

✓

```
%vreg3<def> = OP
  = OP %vreg3:sub_8<kill>
  = OP %vreg3:sub_8_hi<kill>
```

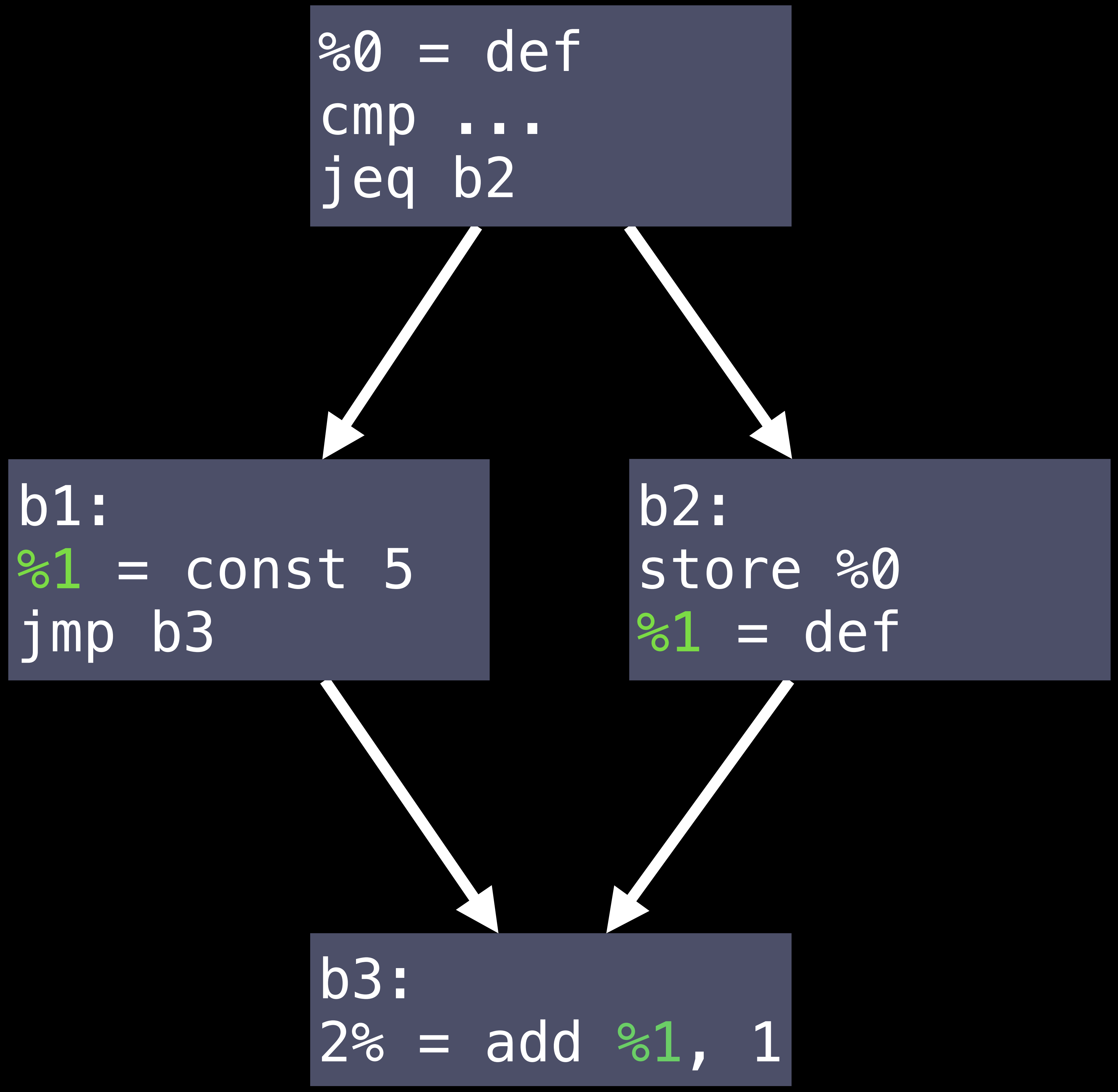
✖ Use after kill (affects whole register)

```
%sp<def,dead> = ADD %sp, 4
 = OP %sp
```

✓ No rules for reserved registers

Liveness Tracking

- Linearize program



```
%0 = def
cmp ...
jeq b2
```

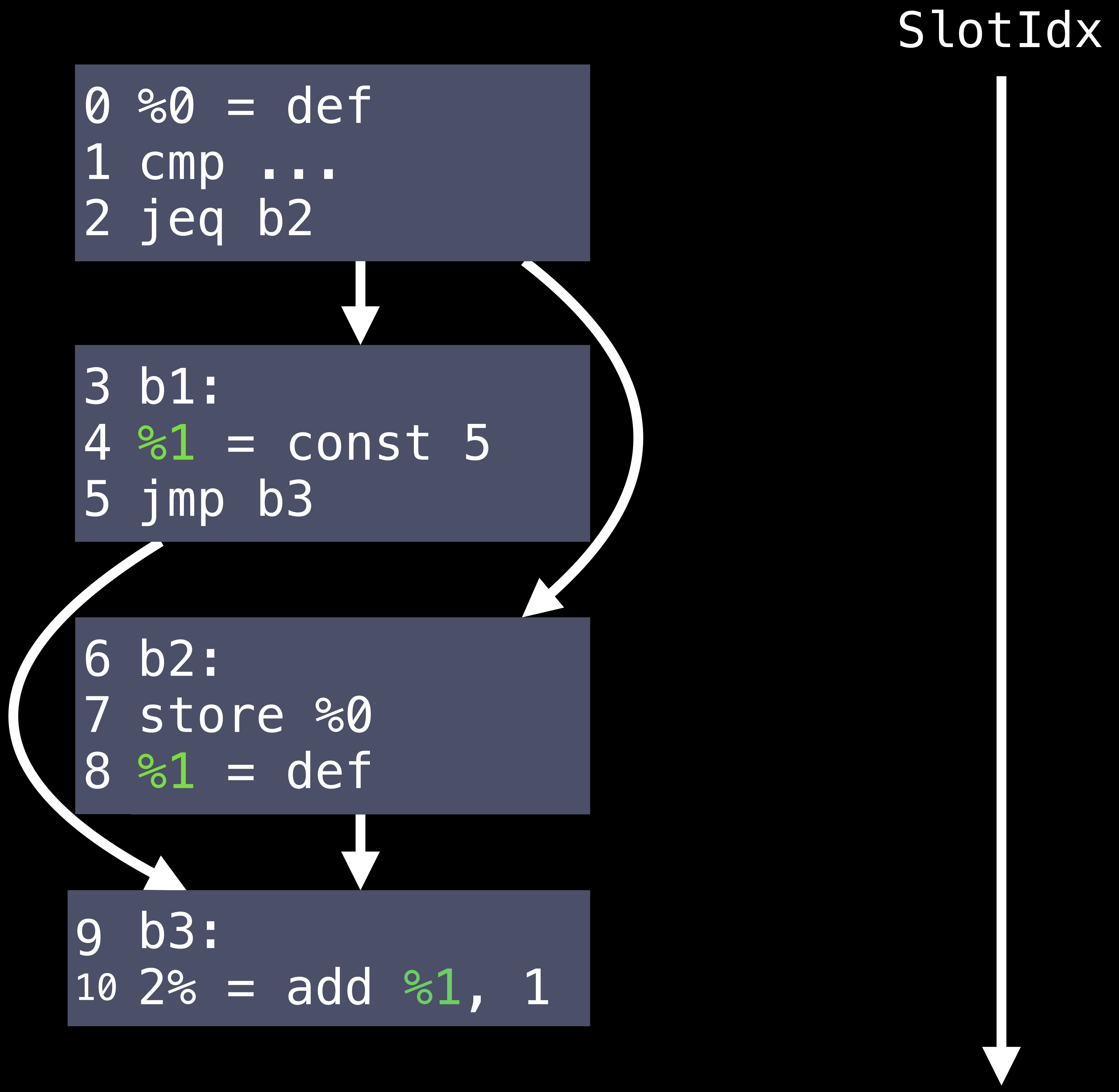
```
b1:
%1 = const 5
jmp b3
```

```
b2:
store %0
%1 = def
```

```
b3:
2% = add %1, 1
```

Liveness Tracking

- Linearize program
- `SlotIndexes` maintains numbering of instructions.



```
0 %0 = def
1 cmp ...
2 jeq b2
```

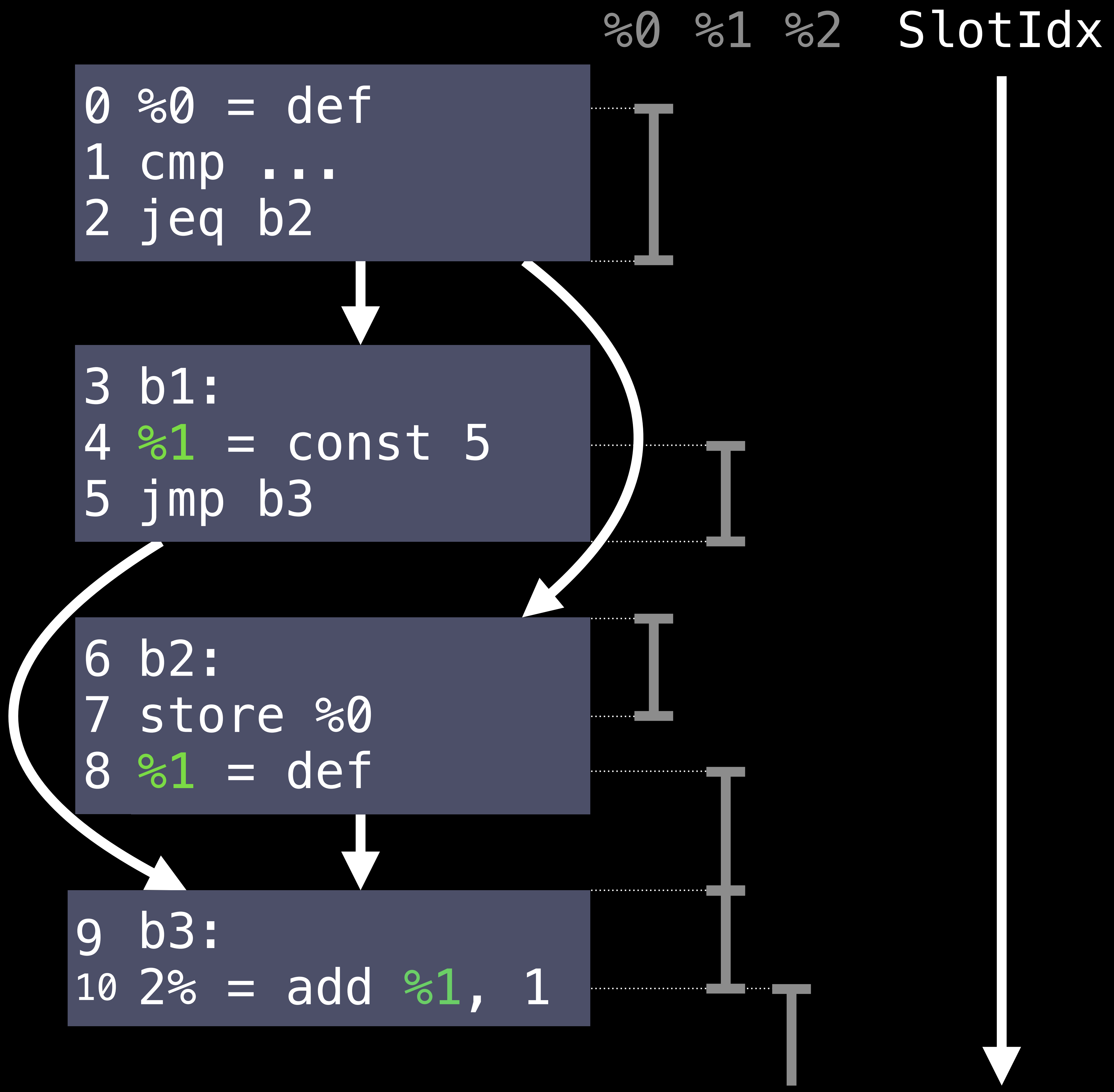
```
3 b1:
4 %1 = const 5
5 jmp b3
```

```
6 b2:
7 store %0
8 %1 = def
```

```
9 b3:
10 2% = add %1, 1
```

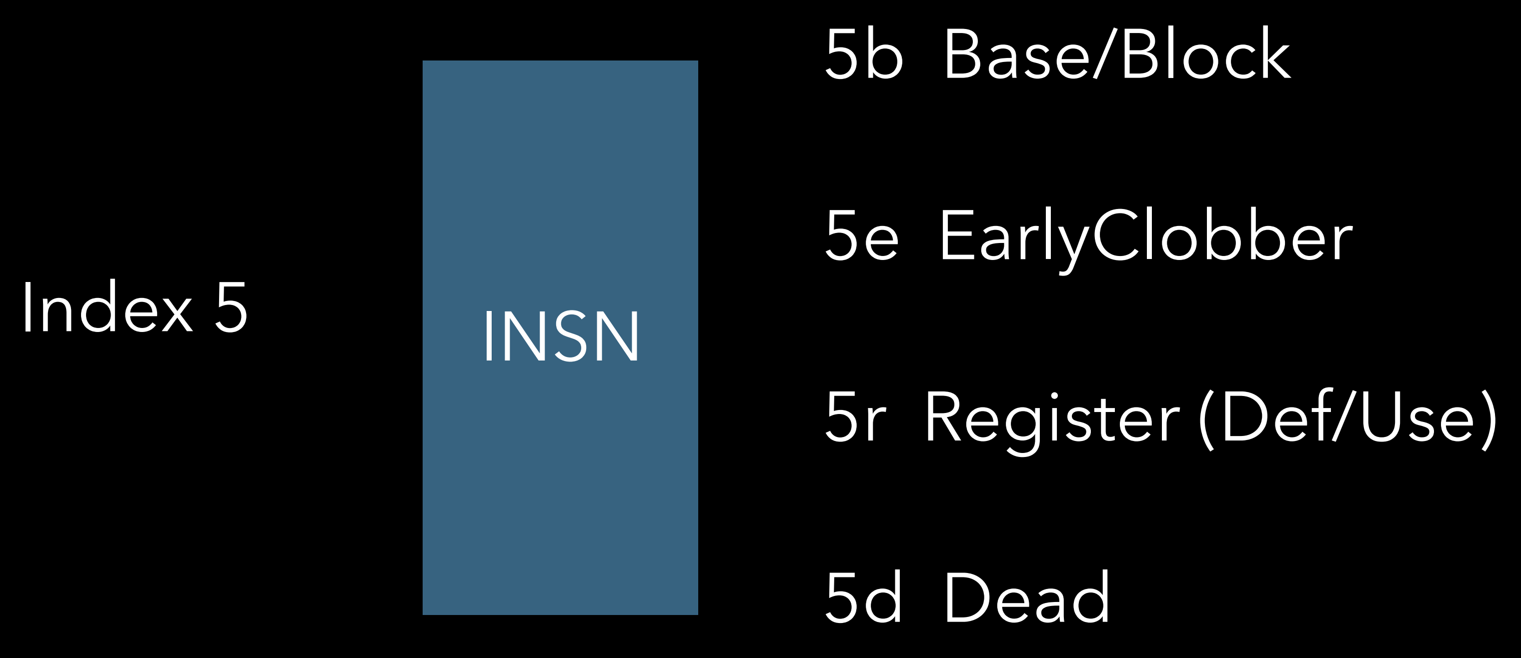
Liveness Tracking

- Linearize program
- `SlotIndexes` maintains numbering of instructions.

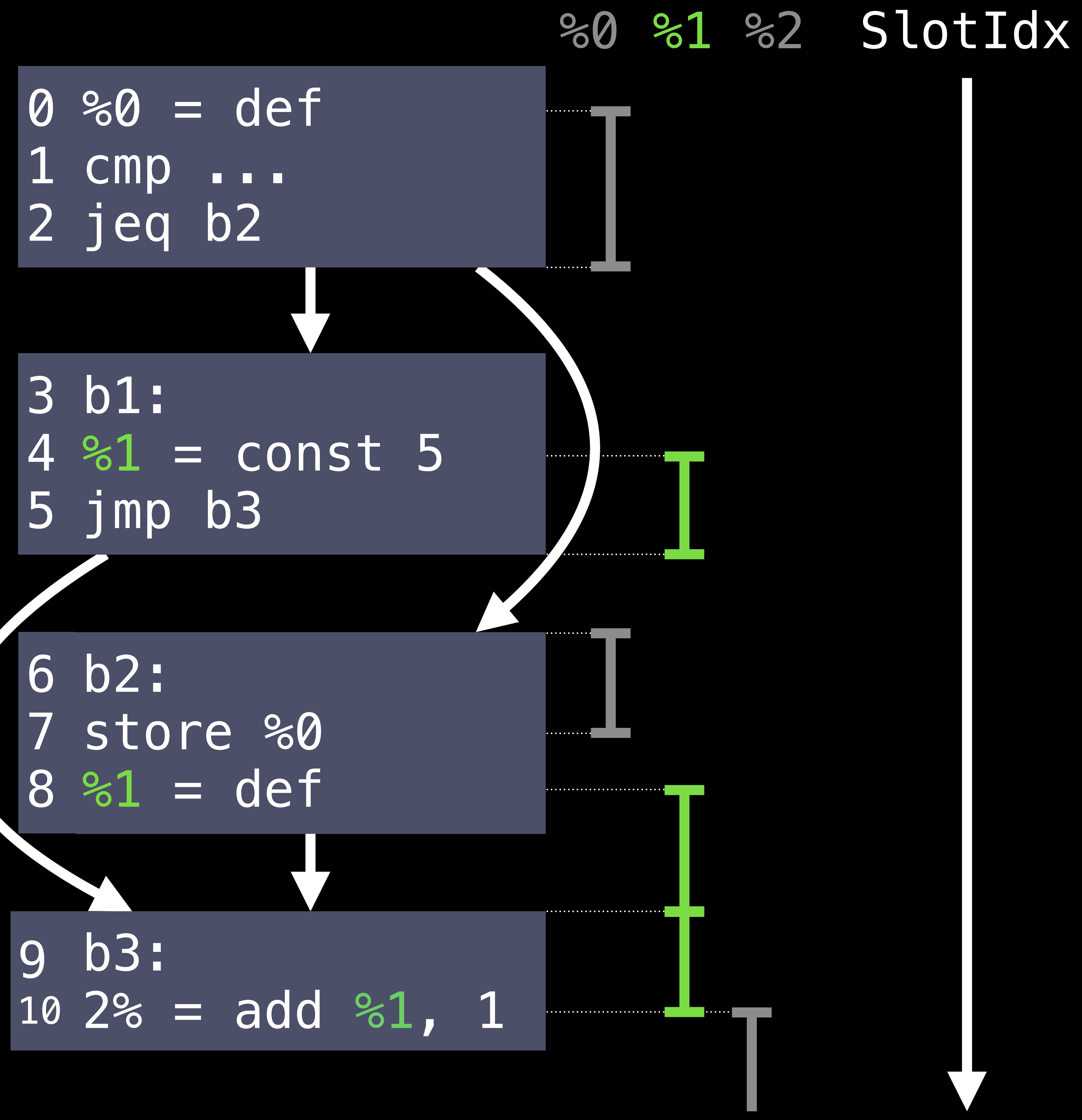


Liveness Tracking

- Linearize program
- `SlotIndexes` maintains numbering of instructions.
- Slots per Instruction:



```
Intervals: %1: [4r:6b)[8r:9b)[9b:10r)
…
```



```
0 %0 = def
1 cmp ...
2 jeq b2

3 b1:
4 %1 = const 5
5 jmp b3

6 b2:
7 store %0
8 %1 = def

9 b3:
10 2% = add %1, 1
```

Register Allocation Tuning

- XXXRegisterInfo.td: Adjust Allocation Order

```
def GR32 : RegisterClass<"X86", [i32], 32,
                         (add EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
                              R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D)>;
```

- Set Register Class Allocation Priority

```
// Tuple of 2 32bit registers.
def VReg_64 : RegisterClass<"AMDGPU", [i64], 32, (add VGPR_64)> {
  let AllocationPriority = 2;
}
```

- Hinting

```
class TargetRegisterInfo { // ...
  virtual void getRegAllocationHints(unsigned VirtReg,
      ArrayRef<MCPhysReg> Order, SmallVectorImpl<MCPhysReg> &Hints, /*...*/);
}
```

Liveness Tracking After Register Allocation

- Live-in list is maintained per Basic Block

```
BB#0:
    Live Ins: %R0, %R3
```

- This also allows to construct Live Out Lists

- Use `LiveRegUnits` (or `LivePhysRegs`) to compute liveness for instructions inside a block.

Prolog Epilog Insertion Pass

- Setup call frames, setup stack frame
- Save/Restore callee saved registers
- Resolve frame indexes
- Register scavenging

```
class TargetFrameLowering { // ...
  virtual void emitPrologue(MachineFunction &MF, MachineBasicBlock &MBB);
  virtual void emitEpilogue(MachineFunction &MF, MachineBasicBlock &MBB);
  virtual void determineCalleeSaves(MachineFunction &MF, BitVector &SavedRegs,
                                    /*...*/);
  virtual void processFunctionBeforeFrameFinalized(MachineFunction &MF,
                                                   /*...*/);
};
```

Frame Indexes

Reg. Use   Frame Index

```
STRi12 %R1, <fi#2>, 0, ...
```

- Prolog epilog insertion resolves frame indexes

```
STRi12 %R1, %SP, 4104, ...
```

- May require temporary registers

```
%vreg0<def> = ADDri %SP, 4096, ...
STRi12 %R1, %vreg0, 8, ...
```

Register Scavenging

- Last step of prolog epilog insertion
- Simulate liveness in basic block, allocate virtual registers
- Insert extra spills and reloads if necessary
- Target must allocate space for spill in advance before frame setup!
  `RegScavenger::addScavengingFrameIndex(int FrameIndex)`

⚠ Precondition: Registers have one definition, all uses in same block

Thank You For Your Attention!

Backup Slides

PEI: Call Frame Setup

```
ADJCALLSTACKDOWN 4, %sp<imp-def>, %sp<imp-use>
STR %sp, 0, 7    ; Store 7 to %sp+0
CALL <ga:@foo>, <regmask ...>, ...
ADJCALLSTACKUP 4, %sp<imp-def>, %sp<imp-use>
ADJCALLSTACKDOWN 8, %sp<imp-def>, %sp<imp-use>
STR %sp, 4, 42   ; Store 42 to %sp + 4
STR %sp, 0, 7    ; Store 7 to %sp + 0
CALL <ga:@bar>, <regmask ...>, ...
ADJCALLSTACKUP 8, %sp<imp-def>, %sp<imp-use>
```

```
%sp<def> = SUB %sp, 8
STR %sp, 0, 7    ; Store 7 to %sp
CALL <ga:@foo>, <regmask ...>, ...
STR %sp, 4, 42   ; Store 42 to %sp+4
STR %sp, 0, 7    ; Store 7 to %sp+0
CALL <ga:@bar>, <regmask ...>, ...
%sp<def> = ADD %sp, 8
```

Target Interfaces

- `TargetMachine`
- `TargetPassConfig`
- `TargetSubtargetInfo`
- `TargetRegisterInfo`, `TargetInstrInfo`
- `TargetLowering`, `TargetFrameLowering`