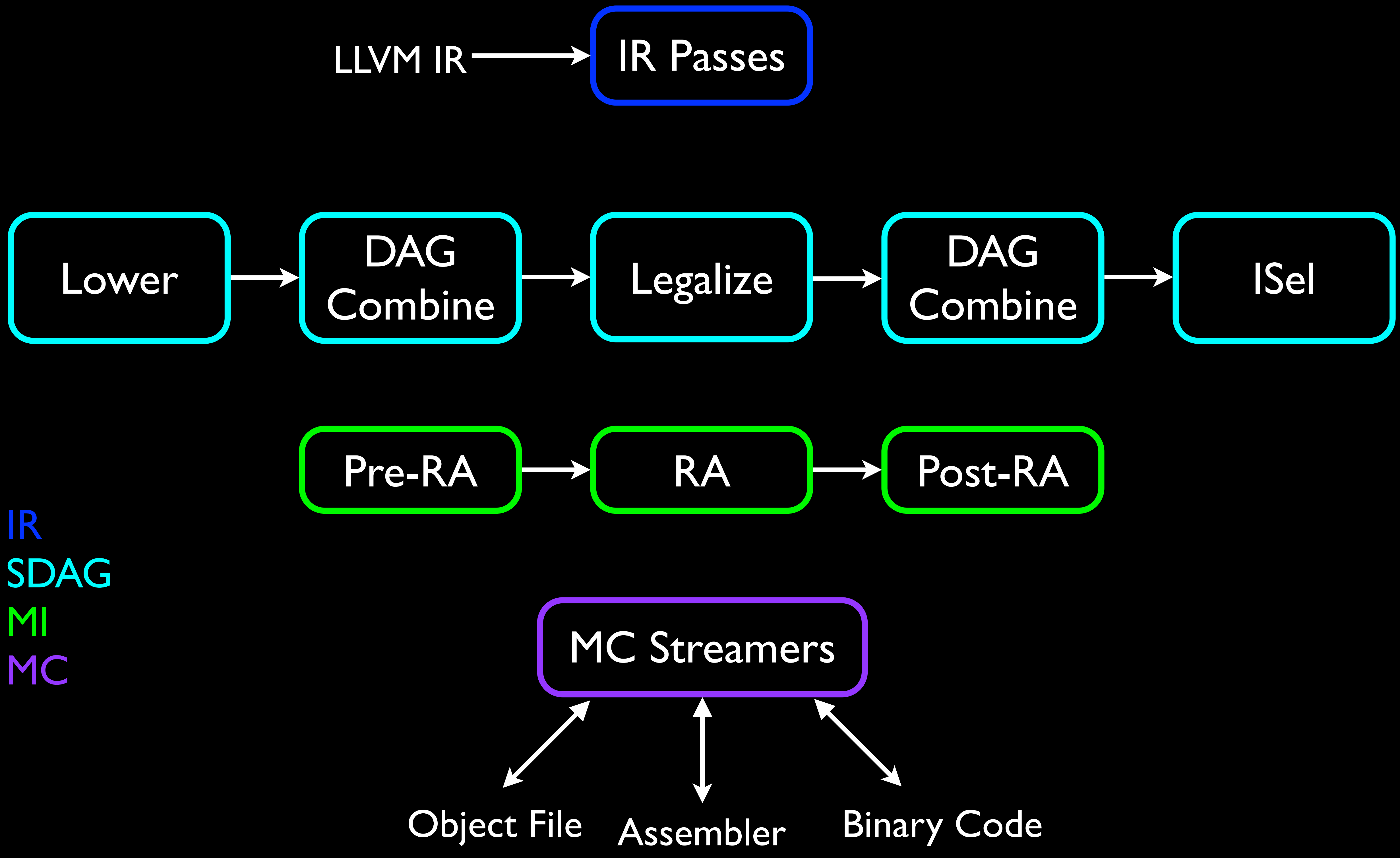
Tutorial: Building a backend in 24 hours

Anton Korobeynikov
anton@korobeynikov.info

Outline

1. From IR to assembler: codegen pipeline
2. MC
3. Parts of a backend
4. Example step-by-step

The Pipeline



LLVM IR

IR Passes

Lower

DAG Combine

Legalize

DAG Combine

ISel

Pre-RA

RA

Post-RA

IR

SDAG

MI

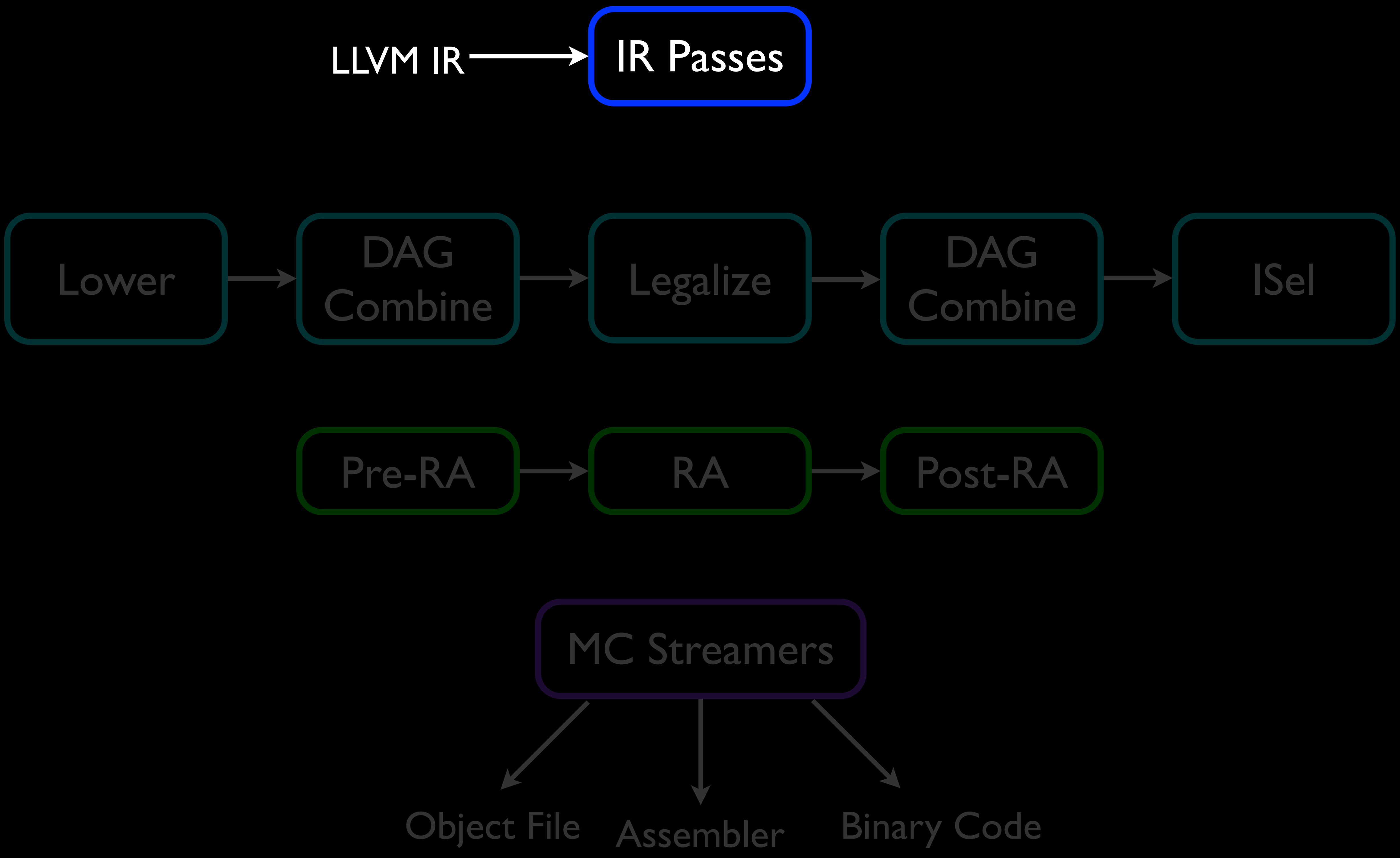
MC

MC Streamers

Object File

Assembler

Binary Code



LLVM IR

IR Passes

Lower

DAG Combine

Legalize

DAG Combine

ISel

Pre-RA

RA

Post-RA

MC Streamers

Object File

Assembler

Binary Code

IR Level Passes

Why?

- Some things are easier to do at IR level
- Simplifies codegen
- Safer (pass pipeline is much more fixed)

IR Level Passes

Why?

- Some things are easier to do at IR level
- Simplifies codegen
- Safer (pass pipeline is much more fixed)

What is done?

- Late opts (LSR, elimination of dead BBs)
- IR-level lowering: GC, EH, stack protector
- Custom pre-isel passes
- CodeGenPrepare

EH Lowering

Why?

- To simplify codegen

EH Lowering

Why?

- To simplify codegen

What is done?

- Lowering of EH intrinsics to unwinding runtime constructs (e.g. sjlj stuff)

CodeGenPrepare

Why?

- To workaround BB-at-a-time codegen

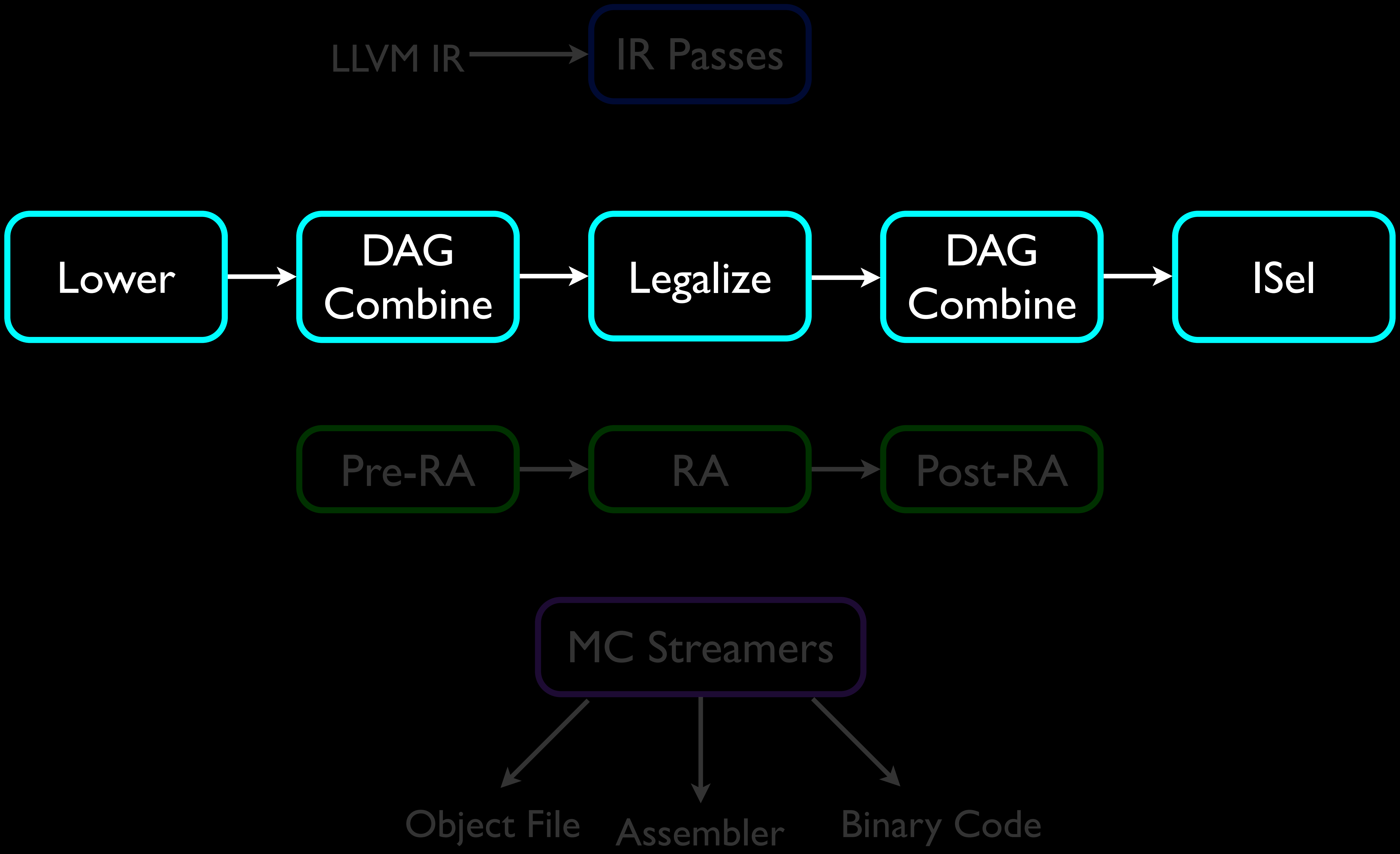
CodeGenPrepare

Why?

- To workaround BB-at-a-time codegen

What is done?

- Addressing mode-related simplifications
- Inline asm simplification (e.g. bswap patterns)
- Move debug stuff closer to defs


LLVM IR
IR Passes
Lower
DAG Combine
Legalize
DAG Combine
ISel
Pre-RA
RA
Post-RA
MC Streamers
Object File
Assembler
Binary Code

Selection DAG

- First strictly backend IR
- Even lower level than LLVM IR
- Use-def chains + additional stuff to keep things in order
- Built on per-BB basis

DAG-level Passes

- Lowering
- Combine
- Legalize
- Combine
- Instruction Selection

DAG Combiner

- Optimizations on DAG
- Close to target
- Runs twice - before and after legalize
- Used to cleanup / handle optimization opportunities exposed by targets

DAG Legalization

Turn non-legal operations into legal one

DAG Legalization

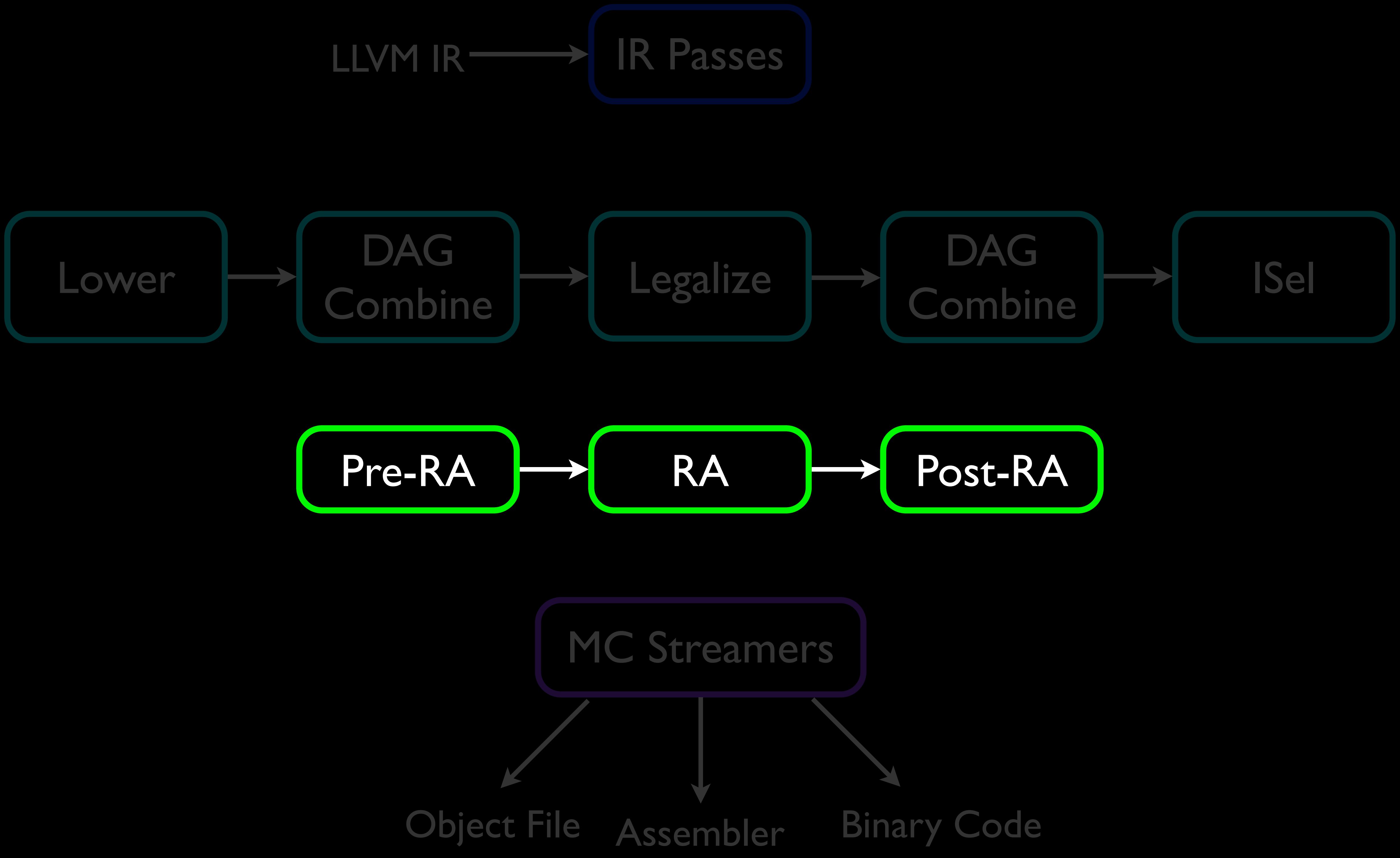
Turn non-legal operations into legal one

Examples:

- Software floating point
- Scalarization of vectors
- Widening of “funky” types (e.g. i42)

Instruction Selection

- Turns SDAGs into MIs
- Uses target-defined patters to select instructions and operands
- Does bunch of magic and crazy pattern-matching
- Target can provide "fast but crude" isel for -O0 (fallbacks to standard one if cannot isel something)



LLVM IR

IR Passes

Lower

DAG Combine

Legalize

DAG Combine

ISel

Pre-RA

RA

Post-RA

MC Streamers

Object File

Assembler

Binary Code

Machine*

- Yet another set of IR: MachineInst + MachineBB + MachineFunction
- Close to target code
- Pretty explicit: set of impdef regs, basic block live in / live out regs, etc.
- Used as IR for all post-isel passes

Pre-RA Passes

- Pre-RA tail duplication
- PHI optimization
- MachineLICM, CSE, DCE
- More peephole opts

Pre-RA Passes

- Pre-RA tail duplication
- PHI optimization
- MachineLICM, CSE, DCE
- More peephole opts

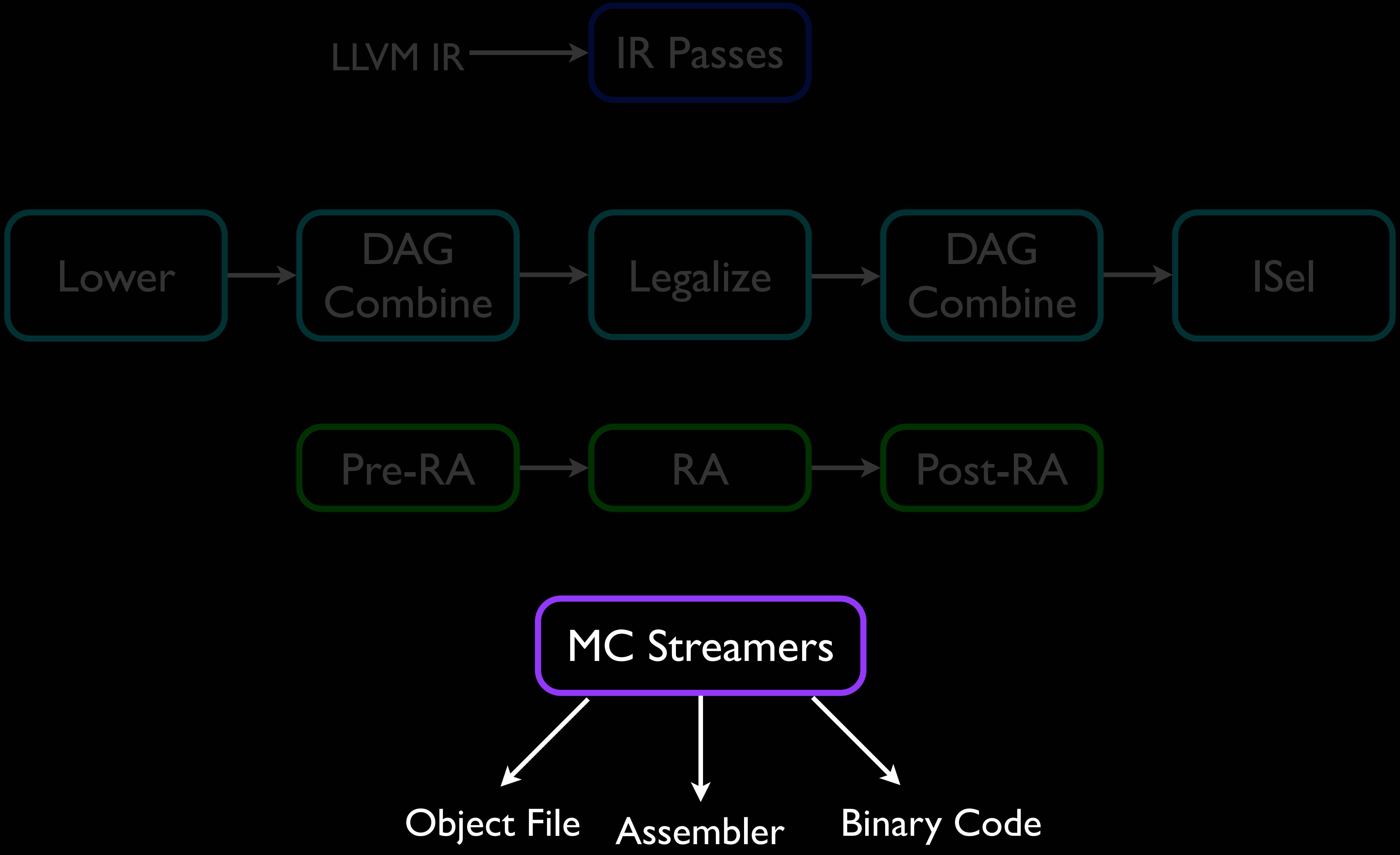
Code is still in SSA form!

Register Allocator

- Fast
- Greedy (default)
- PBQP

Post-RA Passes

1. Prologue / Epilogue Insertion & Abstract Frame Indexes Elimination
2. Branch Folding & Simplification
3. Tail duplication
4. Reg-reg copy propagation
5. Post-RA scheduler
6. BB placement to optimize hot paths



LLVM IR

IR Passes

Lower

DAG Combine

Legalize

DAG Combine

ISel

Pre-RA

RA

Post-RA

MC Streamers

Object File

Assembler

Binary Code

“Assembler Printing”

- Lower MI-level constructs to MCInst
- Let MCStreamer decide what to do next: emit assembler, object file or binary code into memory

Customization

Target can insert its own passes in specific points in the pipeline (e.g. after isel or before scheduler)

Customization

Target can insert its own passes in specific points in the pipeline (e.g. after isel or before scheduler)

Examples:

- IT block formation, load-store optimization on ARM
- Delay slot filling on MIPS or Sparc

The Backend

- Standalone library
- Mixed C++ code + TableGen
- TableGen is a special DSL used to describe register sets, calling conventions, instruction patterns, etc.
- Inheritance and overloading are used to augment necessary target bits into target-independent codegen classes

Stub Backend

How much code we need to create no-op backend?

Stub Backend

How much code we need to create no-op backend?

Some decent amount...

Stub Backend

How much code we need to create no-op backend?

Some decent amount:

- 15 classes
- around 1k LOC (both C++ and TableGen)

FooTargetMachine

- Central class in each backend
- Glues (almost) all the backend classes
- Controls the backend pipeline

FooSubtarget

- Several “subtargets” inside one target
- Usually used to model different instruction sets, platform-specific things, etc.
- Done via “subtarget features”

FooRegisterInfo

Provides various information about register sets:

1. Callee saved regs
2. Reserved (non-allocable) regs
3. Register allocation order
4. Register classes for cross-class copying & coalescing

FooRegisterInfo

Provides various information about register sets:

1. Callee saved regs
2. Reserved (non-allocable) regs
3. Register allocation order
4. Register classes for cross-class copying & coalescing

Partly autogenerated from FooRegisterInfo.td

FooRegisterInfo.td

TableGen description of:

1. Registers,
2. Sub-registers (and aliasing sets for regs)
3. Register classes

FooISelLowering

- Central class for target-aware lowering
- Turns target-neutral SelectionDAG in target-aware (suitable for instruction selection)
- Something can be lowered (albeit not efficiently) in generic way
- Some cases (e.g. argument lowering) always require custom lowering

FooCallingConv.td

Describes the calling convention:

1. What & where & in which order should be passed
2. Not self-containing: used to simplify custom lowering routines
3. Autogenerate set of callee-save registers

FooISelDAGToDAG

- Does most of instruction selection
- Most of C++ code is autogenerated from instruction patterns
- Custom instruction selection code:
  - Complex addressing modes
  - Instructions which require additional care

FooInstrInfo

Hooks used by codegen to:

1. Emit reg-reg copies
2. Save / restore values on stack
3. Branch-related operations
4. Determine instruction sizes

FooInstrInfo.td

Defines the instruction patterns:

- DAG: level of input & output operands
- MI: Instruction Encoding
- ASM: Assembler printing strings

FooInstrInfo.td

Defines the instruction patterns:

- DAG: level of input & output operands
- MI: Instruction Encoding
- ASM: Assembler printing strings

TableGen magic can autogenerate many things

FooInstInfo.td

```
def REV  : AMiscA1I<0b01101011, 0b0011,
                    (outs GPR:$Rd), (ins GPR:$Rm),
                    IIC_iUNAr,
                    "rev", "\t$Rd, $Rm",
                    [(set GPR:$Rd, (bswap GPR:$Rm))]>,
           Requires<[IsARM, HasV6]>;
```

FooInstInfo.td

```
def REV  : AMiscA1I<0b01101011, 0b0011,
                    (outs GPR:$Rd), (ins GPR:$Rm),
                    IIC_iUNAr,
                    "rev", "\t$Rd, $Rm",
                    [(set GPR:$Rd, (bswap GPR:$Rm))]>,
           Requires<[IsARM, HasV6]>;
```

FooInstInfo.td

```
    def REV  : AMiscA1I<0b01101011, 0b0011,
                        (outs GPR:$Rd), (ins GPR:$Rm),
                        IIC_iUNAr,
                        "rev", "\t$Rd, $Rm",
                        [(set GPR:$Rd, (bswap GPR:$Rm))]>,
               Requires<[IsARM, HasV6]>;
```

FooInstInfo.td

```
    def REV  : AMiscA1I<0b01101011, 0b0011,
                        (outs GPR:$Rd), (ins GPR:$Rm),
                        IIC_iUNAr,
                        "rev", "\t$Rd, $Rm",
                        [(set GPR:$Rd, (bswap GPR:$Rm))]>,
               Requires<[IsARM, HasV6]>;
```

FooInstInfo.td

```
    def REV  : AMiscA1I<0b01101011, 0b0011,
                        (outs GPR:$Rd), (ins GPR:$Rm),
                        IIC_iUNAr,
                        "rev", "\t$Rd, $Rm",
                        [(set GPR:$Rd, (bswap GPR:$Rm))]>,
               Requires<[IsARM, HasV6]>;
```

FooFrameLowering

Hooks connected with function stack frames:

1. Prologue & epilogue expansion
2. Function call frame formation
3. Spilling & restoring of callee saved regs

FooMCInstPrinter

- Target part of generic assembler printing code

FooMCInstPrinter

- Target part of generic assembler printing code
- Specifies how a given MCInst should be represented as an assembler string:
  1. Instruction opcodes, operands
  2. Encoding of immediate values,
  3. Workarounds for assembler bugs :)

What’s not covered?

- MC-level stuff: MC{Asm,Instr,Reg}Info
- Assemblers and disassemblers
- Direct object code emission
- MI-level (post-RA) scheduler

OpenRISC

- IP core, not a real CPU chip
- Straightforward 32-bit RISC CPU
- 32 regs
- 3 address instructions
- Rich instruction set

The Goal

Make the following IR to yield the valid assembler:

```
define void @foo() {
entry:
    ret void
}
```

Triple

- Make sure the desired target triple is recognized: include/ADT/Triple.h & lib/Support/Triple.cpp
- Add “or32” entry
- Add “or32 ⇒ openrisc backend” mapping

Stub classes

- Provide stub implementations of all necessary 15 backend classes :(
- Hook them into build system

Stub classes

- Provide stub implementations of all necessary 15 backend classes :(
- Hook them into build system

Maybe it’s a good idea to add ‘stub’ backend to the tree

Registers

Define all registers and register classes:

```
class OpenRISCReg<string n> : Register<n> {
  let Namespace = "OpenRISC";
}

def ZERO : OpenRISCReg<"r0">;
def SP   : OpenRISCReg<"r1">;
...
def R31 : OpenRISCReg<"r31">;

def GR32 : RegisterClass<"OpenRISC", [i32], 32,
    (add (sequence "R%u", 3, 8), (sequence "R%u", 10, 31), LR, SP, FP, ZERO)>;
```

Calling Convention

```
def CC_OpenRISC : CallingConv<[
  // Promote i8 arguments to i32.
  CCIfType<[i8], CCPromoteToType<i32>>,
  // Promote i8 arguments to i32.
  CCIfType<[i16], CCPromoteToType<i32>>,

  // The first 6 integer arguments of non-varargs functions are passed in
  // integer registers.
  CCIfNotVarArg<CCIfType<[i32], CCAssignToReg<[R3, R4, R5, R6, R7, R8]>>>,

  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;
```

Some hooks

- copyPhysReg()
- blank emitPrologue() / emitEpilogue()
- hasFP()
- getReservedRegs()
- getCalleeSavedRegs()

Some boilerplate

- ADJCALLSTACKUP / ADJCALLSTACKDOWN pseudo instructions
- Make sure lowering knows about “native” integer type and register classes

LowerFormalArguments

1. Assign locations to all incoming arguments (depending on their type)
2. Copy arguments passed in registers
3. Create frame index objects for arguments passed on stack
4. Create SelectionDAG nodes for loading of stack arguments

MI to MC

- The lowering MI to MC is straightforward:

```
void OpenRISCMCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) const {
  OutMI.setOpcode(MI->getOpcode());

  for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
    const MachineOperand &MO = MI->getOperand(i);

    MCOperand MCOp;
    switch (MO.getType()) {
    case MachineOperand::MO_Immediate:
      MCOp = MCOperand::CreateImm(MO.getImm());
      break;
    ...
    }
    OutMI.addOperand(MCOp);
  }
```

MCInst Printing

- Printing is easy as well:

```
void OpenRISCInstPrinter::printInst(const MCInst *MI, raw_ostream &O,
                                    StringRef Annot) {
  printInstruction(MI, O);
  printAnnotation(O, Annot);
}

void OpenRISCInstPrinter::printOperand(const MCInst *MI, unsigned OpNo,
                                         raw_ostream &O, const char *Modifier) {
  const MCOperand &Op = MI->getOperand(OpNo);
  if (Op.isReg()) {
    O << getRegisterName(Op.getReg());
  } else if (Op.isImm()) {
    O << Op.getImm();
  } else
    assert(0 && "Unknown operand in printOperand");
}
```

First Instruction

- Add pattern for function return instruction:

```
def return : SDNode<"OpenRISCISD::RET", SDTNone,
                     [SDNPHasChain, SDNPOptInGlue]>;

let isReturn = 1, isTerminator = 1, isBarrier = 1 in {
  def RET   : OpenRISCInst<(outs), (ins),
                          "l.jr r9 # FIXME: delay slot",
                          [(return)]>;
}
```

Clang

- Want to write tescases in C?

Clang

- Want to write tescases in C?
- Hook in clang!
- One has to provide TargetInfo (pretending the toolchain looks binutils-ish)
- Detailed toolchain description can be added later

Next steps

- Reg-reg arithmetic instructions
- Loads / stores: matching address modes
- Proper function frame formation
- Delay slot filling (with NOPs for now)
- Branch folding hooks
- ...

Q & A