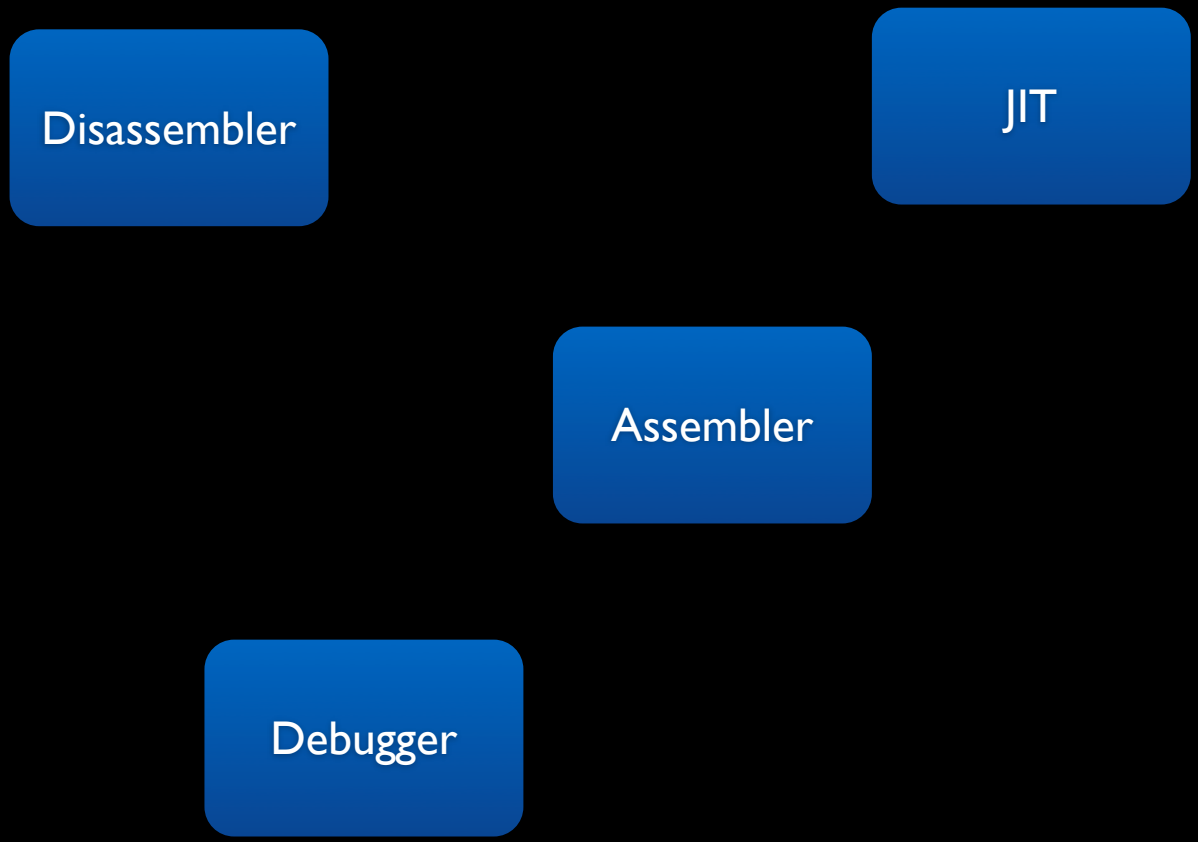
LLVM MC in Practice

Jim Grosbach and Owen Anderson

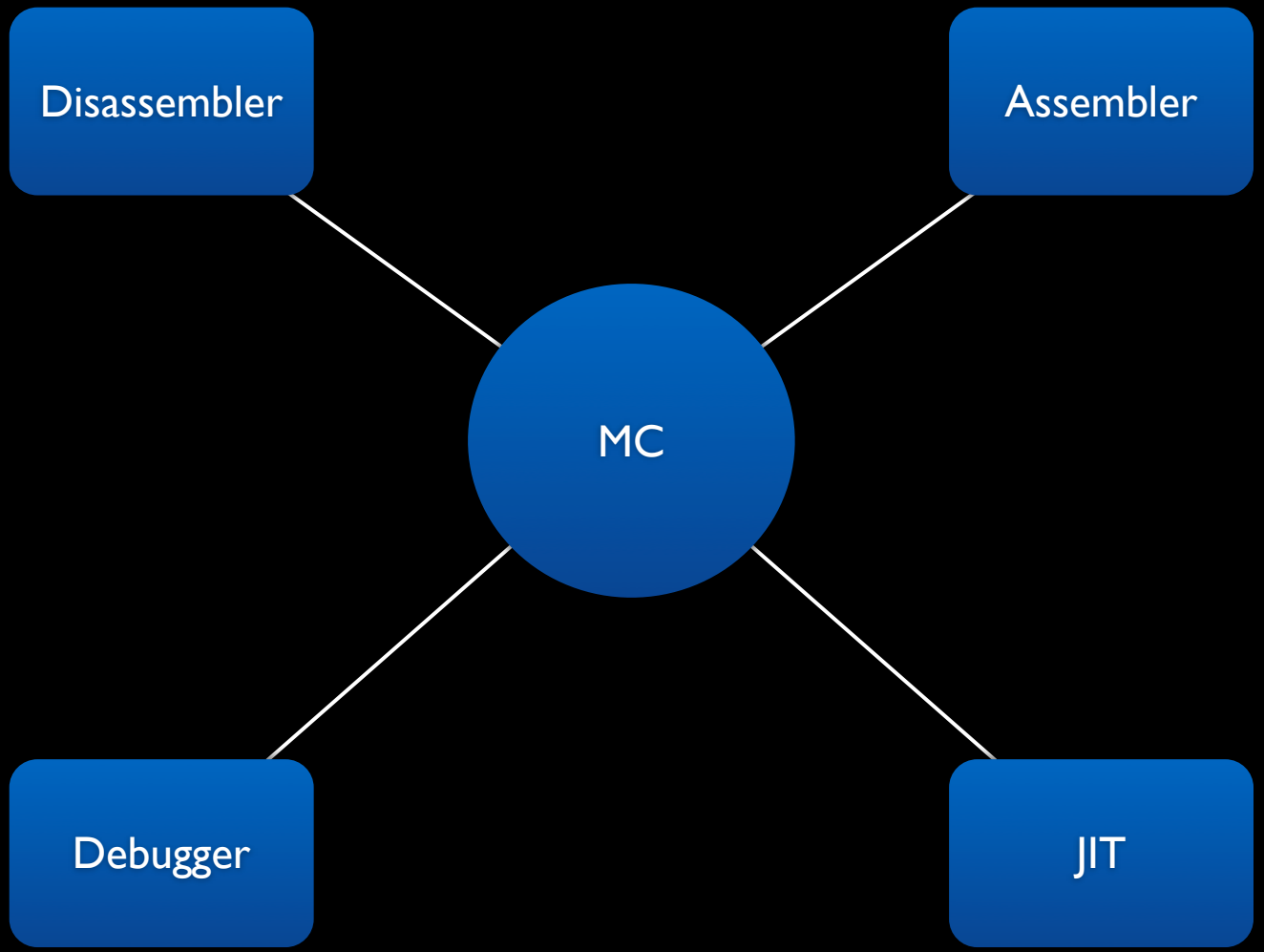
What is MC?

- LLVM Machine Code
- Details of encodings and object file output

Why MC?

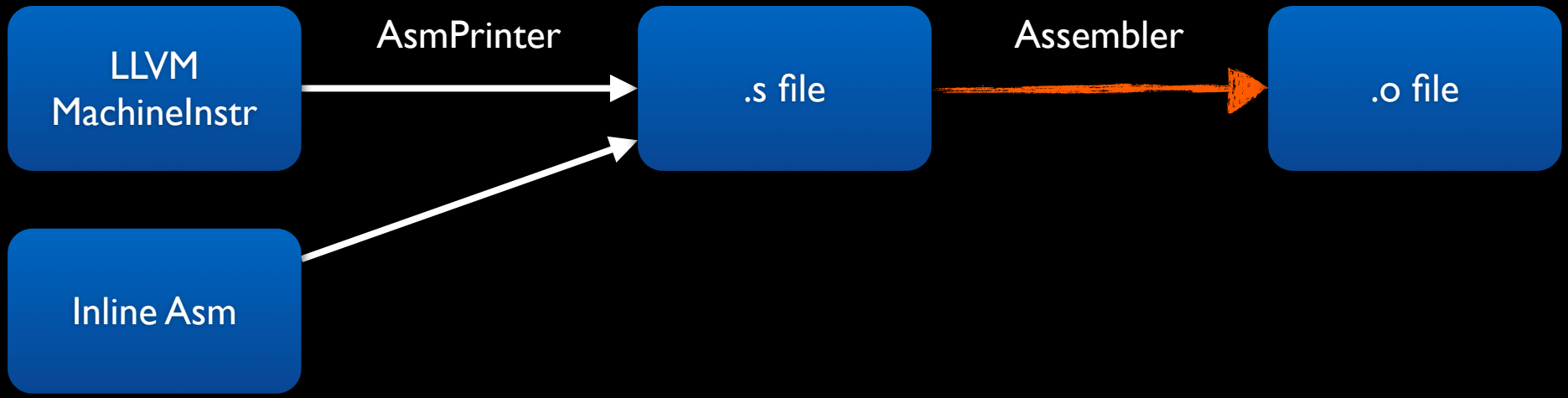


LLVM Machine Code

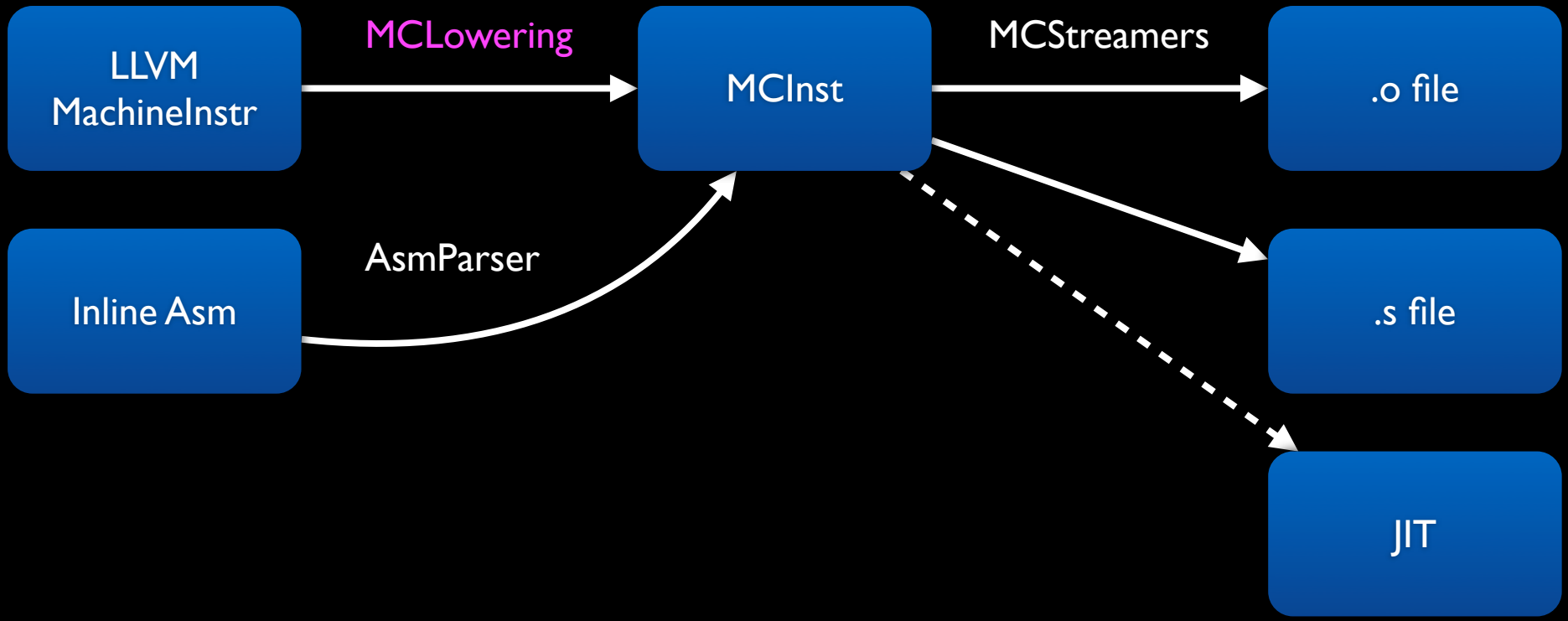


Integrated Assembler

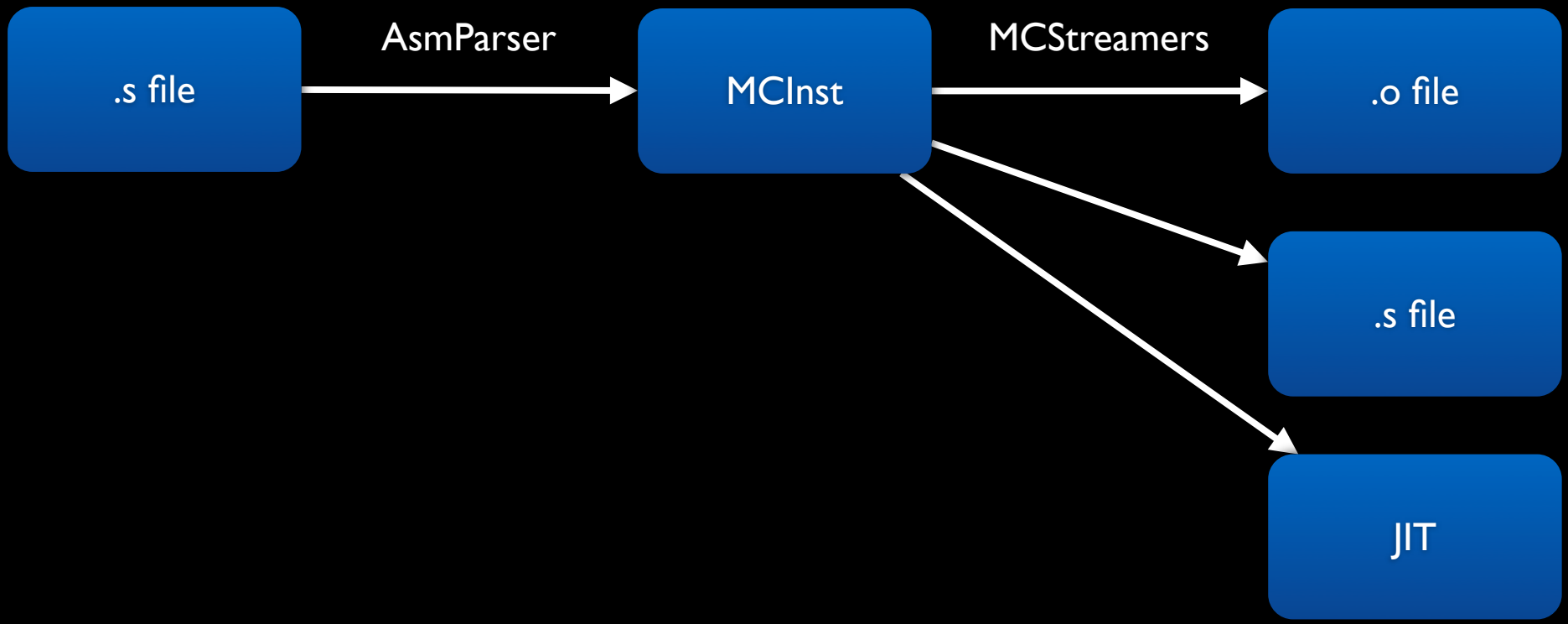
Traditional Assembler



Integrated Assembler



Integrated Assembler



Assembly Parser

- Generated from .td file assembly syntax
- Table driven by mnemonic
- Match by type of operands
- Custom hooks for complex operands

Status

- Enabled by default for x86
- Beta on ARM, will be default soon
- Work ongoing for other targets

Summary

- Direct to object file compiler output
- Supports inline assembler
- Standalone assembler (clang driver)

New Disassembler Framework

A New Disassembler

- Many targets share basic features
  - Fixed-length instructions
  - Simple operand ⇔ encoding mapping
- ARM, Sparc, PowerPC, ...

Fixed Length Disassembler

- Goals
  - Leverage existing encoding data
  - Massively auto-generated
  - Manual hooks when auto-generated code goes astray

Existing Encoding Data

```
def MSRi : ABI<0b0011, (outs),
               (ins msr_mask:$mask,  so_imm:$a),
               NoItinerary,
               "msr", "\t$mask, $a", []> {
  bits<5> mask;
  bits<12> a;

  let Inst{23} = 0;
  let Inst{22} = mask{4}; // R bit
  let Inst{21-20} = 0b10;
  let Inst{19-16} = mask{3-0};
  let Inst{15-12} = 0b1111;
  let Inst{11-0} = a;
}
```

Existing Encoding Data

```
def MSRi : ABI<0b0011, (outs),
                       (ins msr_mask:$mask,  so_imm:$a),
                       NoItinerary,
                       "msr", "\t$mask, $a", []> {
  bits<5> mask;
  bits<12> a;

  let Inst{23} = 0;
  let Inst{22} = mask{4}; // R bit
  let Inst{21-20} = 0b10;
  let Inst{19-16} = mask{3-0};
  let Inst{15-12} = 0b1111;
  let Inst{11-0} = a;
}
```

Auto-generated Disassembler

- Tree of "filters"
  - Each layer switches on a range of bits that distinguishes the most classes of instructions
- Leaves hold concrete decoders

```
switch (Inst{28-17}) {
    ...
    case 0x123:
    switch (Inst{5-4}) {
      ...
      case 0x1:
      decodeInst();
      ...
    }
    ...
}
```

Concrete Decoders

```
MI.setOpcode(243);
tmp = 0;
tmp |= (fieldFromInstruction32(insn, 16, 4) << 0);
tmp |= (fieldFromInstruction32(insn, 22, 1) << 4);
DecodeMSRMask(MI, tmp, Address, Decoder)
tmp = fieldFromInstruction32(insn, 0, 12);
DecodeSOImmOperand(MI, tmp, Address, Decoder)
tmp = fieldFromInstruction32(insn, 28, 4);
DecodePredicateOperand(MI, tmp, Address, Decoder)
return S; // MSRi
```

Concrete Decoders

```
MI.setOpcode(243);
tmp = 0;
tmp |= (fieldFromInstruction32(insn, 16, 4) << 0);
tmp |= (fieldFromInstruction32(insn, 22, 1) << 4);
DecodeMSRMask(MI, tmp, Address, Decoder)
tmp = fieldFromInstruction32(insn, 0, 12);
DecodeSOImmOperand(MI, tmp, Address, Decoder)
tmp = fieldFromInstruction32(insn, 28, 4);
DecodePredicateOperand(MI, tmp, Address, Decoder)
return S; // MSRi
```

Concrete Decoders

```
MI.setOpcode(243);
tmp = 0;
tmp |= (fieldFromInstruction32(insn, 16, 4) << 0);
tmp |= (fieldFromInstruction32(insn, 22, 1) << 4);
DecodeMSRMask(MI, tmp, Address, Decoder)
tmp = fieldFromInstruction32(insn, 0, 12);
DecodeSOImmOperand(MI, tmp, Address, Decoder)
tmp = fieldFromInstruction32(insn, 28, 4);
DecodePredicateOperand(MI, tmp, Address, Decoder)
return S; // MSRi
```

Manual Operand Hooks

- Needed to handle operands more complex than an immediate
  - Most are simple: reg # → reg enum
  - Some are complex
  - Range checking

```
def postidx_reg : Operand<i32> {
  let EncoderMethod =
    "getPostIdxRegOpValue";
  let DecoderMethod =
    "DecodePostIdxReg";
  let PrintMethod =
    "printPostIdxRegOperand";
  let ParserMatchClass =
    PostIdxRegAsmOperand;
  let MIOperandInfo =
    (ops GPR, i32imm);
}
```

Manual Instruction Decoders

- Necessary to handle very complex instructions
  - Tied sub-operands of ComplexOperand's
  - Under-specified encodings

```
def CPS3p :
  CPS<(ins imod_op:$imod,
           iflags_op:$iflags,
           imm0_31:$mode),
       "$imod\t$iflags, $mode"> {
  let DecoderMethod =
    "DecodeCPSInstruction";
}
```

Challenge Going Forward

- Compile time
  - ARMGenDisassemblerTables.inc is 32K LOC
    - Handwritten code is only 4K LOC!
  - De-duplicating leaf decoders
  - Make it table-driven?

Fixed Length Disassembler

- Operational today in the ARM disassembler
- Eases the task of writing/maintaining the disassembler
- Will continue to improve in the future

Rich Disassembly

Rich Disassembly

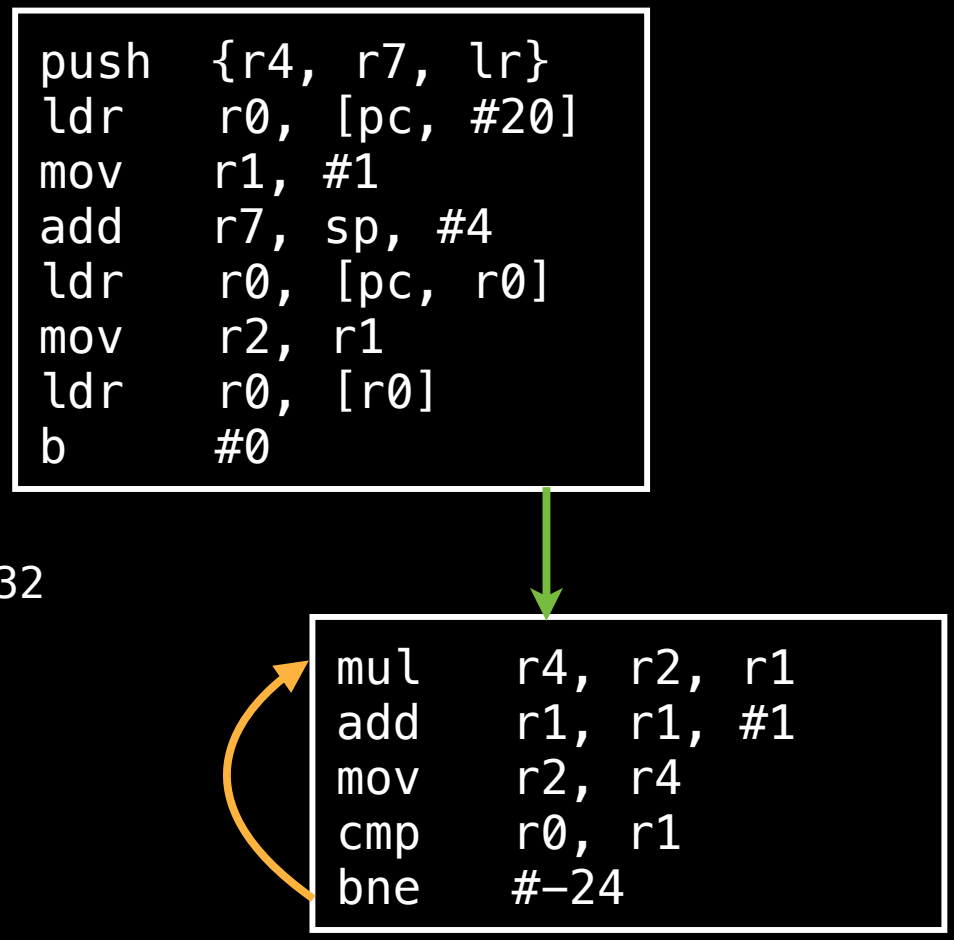
- FixedLengthDisassembler improved life for disassembler writers...
- What can we do for disassembler *users*?

```
fac:
push {r4, r7, lr}
ldr  r0, [pc, #20]
mov  r1, #1
add  r7, sp, #4
ldr  r0, [pc, r0]
mov  r2, r1
mov  r3, r1
ldr  r0, [r0]
b    #0
b    #0
andeq  r0, r0, r8, asr #32
mul  r4, r2, r1
add  r1, r1, #1
mov  r2, r4
cmp  r0, r1
bne  #-24
ldr  r0, [pc, #16]
add  r0, pc, r0
bl   #-72 # _puts
mov  r0, r4
pop  {r4, r7, lr}
bx   lr
andeq  r0, r0, r0, lsl r0
```

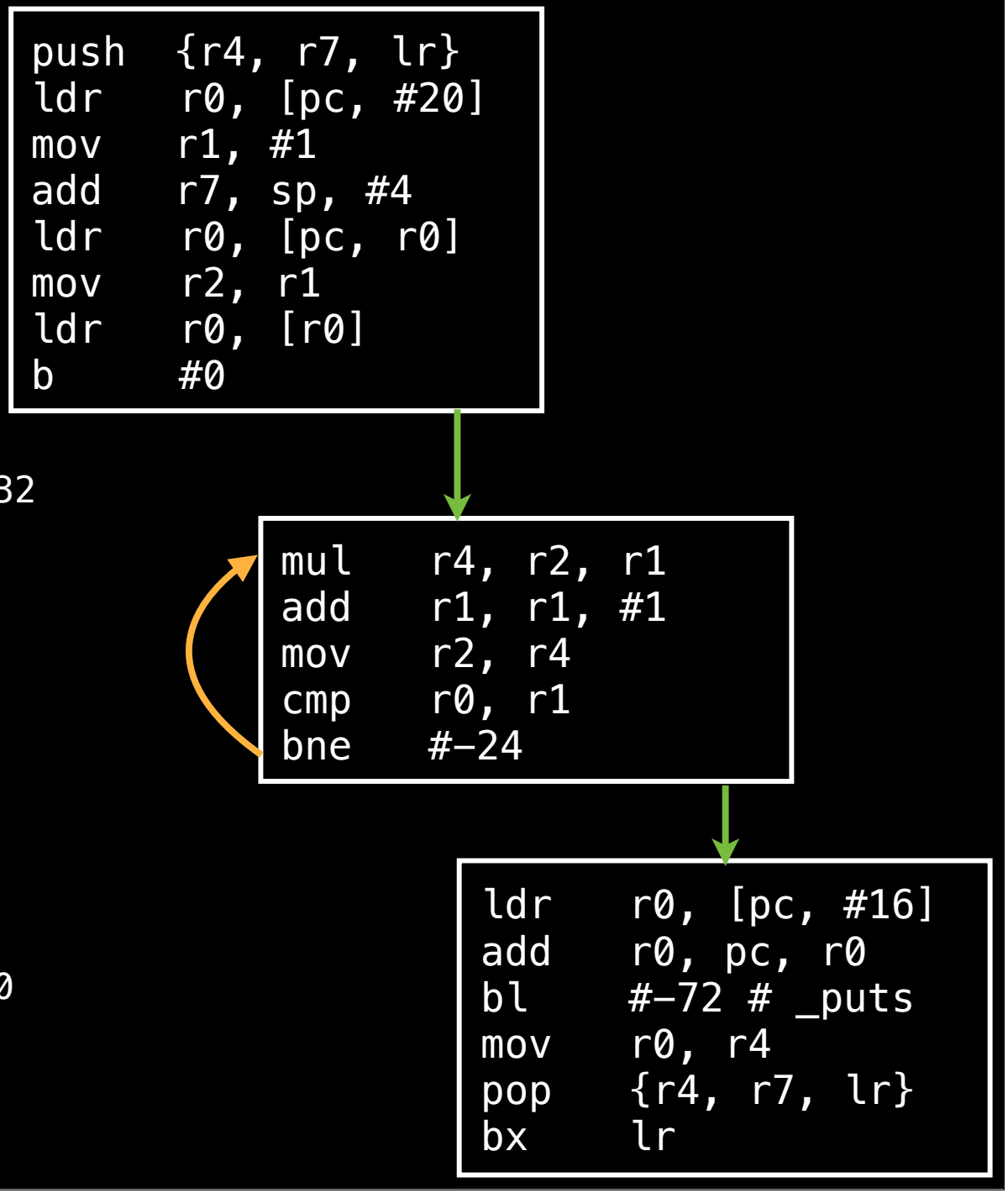
```
_fac:
push  {r4, r7, lr}
ldr   r0, [pc, #20]
mov   r1, #1
add   r7, sp, #4
ldr   r0, [pc, r0]
mov   r2, r1
ldr   r0, [r0]
b     #0
andeq  r0, r0, r8, asr #32
mul   r4, r2, r1
add   r1, r1, #1
mov   r2, r4
cmp   r0, r1
bne   #-24
ldr   r0, [pc, #16]
add   r0, pc, r0
bl    #-72 # _puts
mov   r0, r4
pop   {r4, r7, lr}
bx    lr
andeq  r0, r0, r0, lsl r0
```

```
push  {r4, r7, lr}
ldr   r0, [pc, #20]
mov   r1, #1
add   r7, sp, #4
ldr   r0, [pc, r0]
mov   r2, r1
ldr   r0, [r0]
b     #0
```

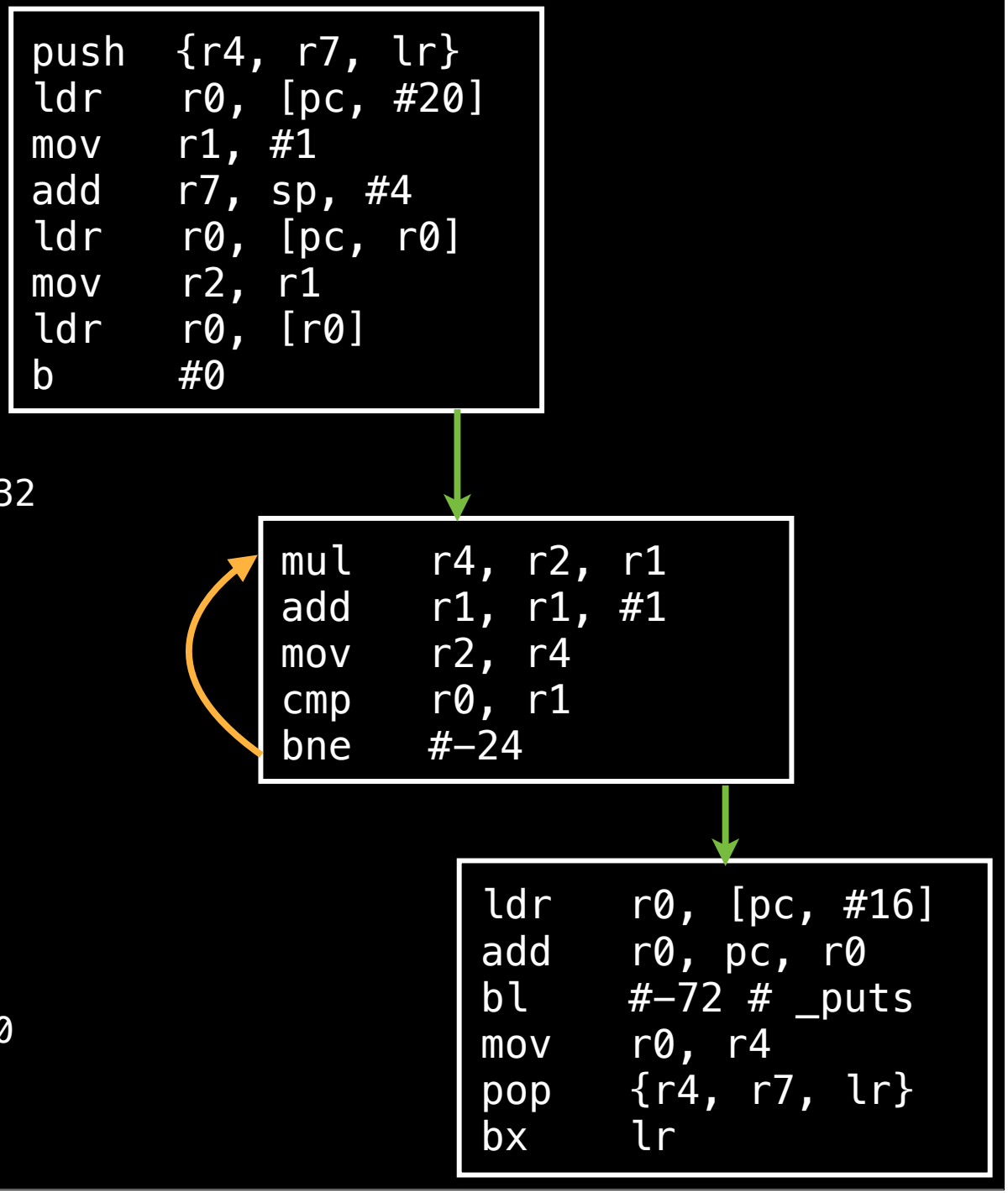
```
_fac:
push  {r4, r7, lr}
ldr    r0, [pc, #20]
mov    r1, #1
add    r7, sp, #4
ldr    r0, [pc, r0]
mov    r2, r1
ldr    r0, [r0]
b      #0
andeq  r0, r0, r8, asr #32
mul    r4, r2, r1
add    r1, r1, #1
mov    r2, r4
cmp    r0, r1
bne    #-24
ldr    r0, [pc, #16]
add    r0, pc, r0
bl     #0x72 <_puts>
mov    r0, r4
pop    {r4, r7, lr}
bx     lr
andeq  r0, r0, r0, lsl r0
```



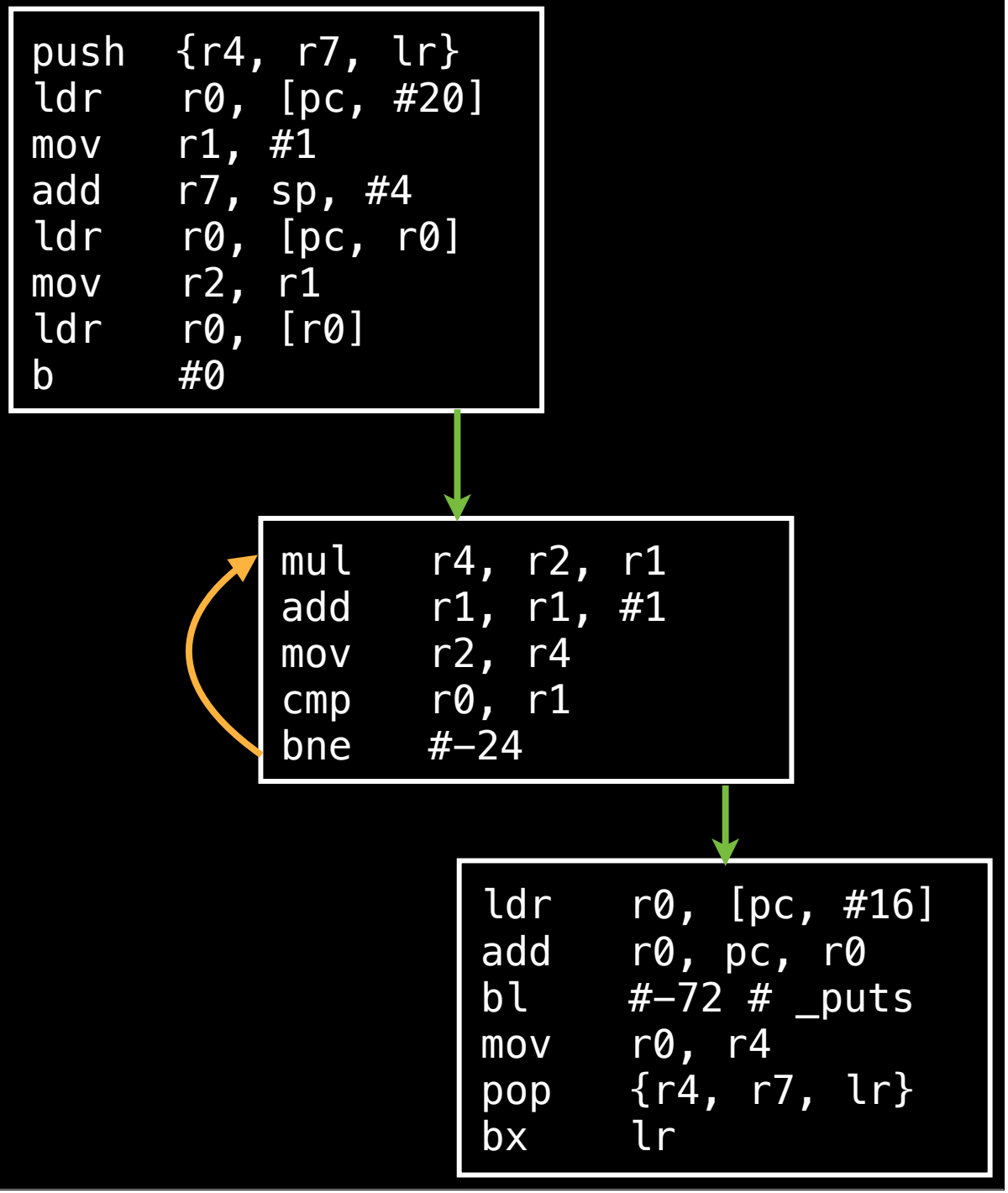
```
_fac:
push  {r4, r7, lr}
ldr   r0, [pc, #20]
mov   r1, #1
add   r7, sp, #4
ldr   r0, [pc, r0]
mov   r2, r1
ldr   r0, [r0]
b     #0
andeq  r0, r0, r8, asr #32
mul   r4, r2, r1
add   r1, r1, #1
mov   r2, r4
cmp   r0, r1
bne   #-24
ldr   r0, [pc, #16]
add   r0, pc, r0
bl    #-72 # _puts
mov   r0, r4
pop   {r4, r7, lr}
bx    lr
andeq  r0, r0, r0, lsl r0
```



```
_fac:
push  {r4, r7, lr}
ldr    r0, [pc, #20]
mov    r1, #1
add    r7, sp, #4
ldr    r0, [pc, r0]
mov    r2, r1
ldr    r0, [r0]
b      #0
andeq  r0, r0, r8, asr #32
# Loop begin:
  mul      r4, r2, r1
  add      r1, r1, #1
  mov      r2, r4
  cmp      r0, r1
  bne      #-24
ldr    r0, [pc, #16]
add    r0, pc, r0
bl     #-72 # _puts
mov    r0, r4
pop    {r4, r7, lr}
bx     lr
andeq  r0, r0, r0, lsl r0
```



```
_fac:
push  {r4, r7, lr}
ldr    r0, [pc, #20]
mov    r1, #1
add    r7, sp, #4
ldr    r0, [pc, r0]
mov    r2, r1
ldr    r0, [r0]
b      #0
# 4 bytes of data:
.long 0x00000048
# Loop begin:
  mul      r4, r2, r1
  add      r1, r1, #1
  mov      r2, r4
  cmp      r0, r1
  bne      #-24
ldr    r0, [pc, #16]
add    r0, pc, r0
bl     #-72 # _puts
mov    r0, r4
pop    {r4, r7, lr}
bx     lr
# 4 bytes of data:
.long 0x00000010
```



Disassembly Annotations

- Integration with libObject
- Use relocation and/or DWARF information
  - Display printf strings
  - Decode objc_msgSend
  - Print file/line information

Work-in-Progress

- Initial work done by Benjamin Kramer
  - MachO-specific
- Future Work: Make it more generic
  - Enhancing libObject APIs
  - Improved integration between disassembler and libObject

Work-in-Progress

- Future Work: Expose an API
  - Clients: LLDB, profilers, binary analysis/rewriting
  - “Programmatic objdump”
  - Human-friendly disassembly

State of the Disassembler

- Right Now: New Disassembler Framework
- The Future: Rich Disassembly API

MC JIT

MC JIT

- Why do we want or need a new JIT?
  - Duplicated functionality for encoding
  - Need to handle Inline assembly

MC JIT

- Uses same MC path as static compiler
- Object file emitted to memory
- Runtime dynamic linker

MC JIT Addressing

- Address space independence
- JIT process separate from target execution process
- Arbitrary object file inputs

MC JIT Today

- OSX LLDB for expression evaluation
- Remote process
- Cross compile for remote target

Summary

- x86 and ARM integrated assemblers are here and awesome!
- Disassembler framework makes things easier!
- Rich disassembly is magical and growing fast!
- MC JIT is on the way and enabling new features.

Questions?