Object Code Emission & llvm-mc

LLVM Developers Meeting, 2009
Cupertino, CA

Bruno Cardoso Lopes
bruno.cardoso@gmail.com
Introduction

• Motivation
• Background
• Actual Code Emission
• Object Code Emission
• llvm-mc
Motivation

- **Known path**
  
  Compiler → .s → Assembler → .o → Linker

- **Object code path**
  
  Compiler → .o → Linker
Motivation

• Why direct object code emission?
• Bypass the external assembler.
• Speed-up compile time.
Background

- Current code emission:
  - Asm printers
  - JIT engine
Asm Printer

- AsmPrinter
- Instructions are described on .td files.
- Auto-generated method is used to print instructions.
Asm Printer

```cpp
void X86AsmPrinter::printMCInst(const MCInst *MI) {
  if (MAI->getAssemblerDialect() == 0)
    X86ATTInstPrinter(0, *MAI).printInstruction(MI);
  else
    X86IntelInstPrinter(0, *MAI).printInstruction(MI);
}
```
JIT

- JIT emits binary code.
- Blobs are emitted to memory by a target specific code emitter class.
• The code is emitted per-function

JIT

MachineFunctionPass

X86 Emitter

PPC Emitter

...
• Only PPC has an auto-generated code emitter.

```cpp
...  
class Emitter : public MachineFunctionPass {  
  ...  
  bool runOnMachineFunction(MachineFunction &MF);  
  void emitInstruction(const MachineInstr &MI,  
                       const TargetInstrDesc *Desc);  
...  
```
MachineCodeEmitter

- The actual binary code emission is done by calls to the **MachineCodeEmitter**.

```c
void ...emitInstruction(const MachineInstr &MI, ...) {
  // Emit the lock opcode prefix as needed.
  if (Desc->TSFlags & X86II::LOCK)
    MCE.emitByte(0xF0);
  ...
```

**MachineCodeEmitter**
JITCodeEmitter

- JIT code emission is implemented in the JITCodeEmitter.
- A specialization from MCE.
- Implement methods to actually write to memory:

  - emitByte(..)
  - emitULEB128Bytes(..)
  - emitDWordLE(...)
  - emitAlignment(....)
Object Code

- Object Code support is implemented using this scenario.
- Specialize the MCE as JIT does.
- MCE is an instance of ObjectCodeEmitter.
Object Code

- The specific formats (e.g. ELF) are specializations of ObjectCodeEmitter.

```
ObjectCodeEmitter
  \_ ELFCodeEmitter
  \_ COFFCodeEmitter
  \_ ...
```
Object Code

- Blobs of code and data are written to **BinaryObjects**.
- High level abstraction of "Sections" or "Segments".

```cpp
class ELFSection : public BinaryObject {
public:
    ...
...}
```
ELFCodeEmitter

- Handling of ConstantPools and Jumptables.
  - On each binary format a different section.
- Generic target relocations to ELF specific ones.

`llvm::reloc_absolute_word`  
`R_X86_64_32`
The ELFCodeEmitter emits code to BinaryObjects.
ELFWriter

• Emits the symbol table, string table, header and relocations into binary objects.

• Dump binary objects to a final file.
Limitations

- Inline assembly not handled by emitters.
- That demands an assembly parser.
- Solution: **llvm-mc**.
llvm-mc

- Machine code driver.
- Current playground for an assembly parser, assembler and disassembler.
• **Goals:**
  
  • Extract all info from .td files.
  
  • Auto-generate a assembler, disassembler and code generator.
  
  • Integrate the assembler into the compiler.
llvm-mc

- **Goals:**
  - At least ~20% speedup at “-O0 -g”.
  - Share binary writers code base as much as possible among different formats.
- **In progress:**
  - Parse an assembly file and dump the Lex tokens.

```
.data
.ascii "hello"
```

```
identifier: .data
EndOfFileStatement
identifier: .ascii
string: "hello"
EndOfFileStatement
```

```
-as-lex
```
In progress:

- Parse and assemble a .s file, emitting asm again or object code.

```
$ llvm-mc -assemble -output-asm-variant=0
   -show-encoding x86.s

   .section __TEXT,__text,regular,pure_instructions
   subb   %al, %al
   # encoding: [0x28,0xc0]
   addl   $24, %eax
   # encoding: [0x83,0xc0,0x18]
```
• In progress:
  • A complete assembler: includes relaxation phases, which allows late optimizations
  • Example: Jump instruction encoding on x86.
• In progress:
  • Interactive disassembler: makes easier to write regression tests for instruction encoding

```
$ llvm-mc -disassemble
74 22 ← user input
1 instruction:
  74 22 je 34
```
• **Architecture**
  
  • The asm parser emits code through a generic streamer, **MCStreamer**.
  
  • The streamer is specialized to emit asm or object code.
llvm-mc

AsmParser.Run() -> EmitInstruction() -> EncodeInstruction() -> MachObjectWriter() -> WriteObject()

.MCStreamer

.MCAsmStreamer

.MCMachOStreamer
llvm-mc

- **Current limitations**
  - Quite new and experimental.
  - Demands lots of clean up and refactoring.
  - Hardcoded for MachO.
**Current limitations**

- ELF emission is not integrated into the llvm-mc architecture.
- ELF assembly parsing bits not implemented.
- The Assembly printer is not entirely converted to use **MCAsmStreamer**.
llvm-mc

- **MCStreamer future:**
  - Support other binary formats.
  - New specializations for: JIT, dwarf EH and debug info.
• **MCStreamer future:**

  • JIT and asm printers will eventually be merged into only one “emitter”.
  
  • “-S” could generate “verbose assembly” by default (loop depth, encoding info, ...)

---

**llvm-mc**
Future Design

- Printing .s
Future Design

- JIT

```
CodeGen -> Code Emitter

Code Emitter -> MCStreamer

MCJITStreamer -> Memory
```
Future Design

• .o writing

- CodeGen ➔ Code Emitter ➔ MCStreamer ➔ MCOObjectStreamer ➔ AssemblerBackend ➔ ELF, MachO, ...

- .o
Future Design

- Inline asm for .o file writing

Diagram:

- CodeGen → Code Emitter
- AsmParser
- MCSStreamer
- MCObjectStreamer
- ELF, MachO, ...
- .o → AssemblerBackend
Future Design

AssemblerBackend

MCMachOWriter

Layout

Relaxation

MCELFWriter

...
Object Code Emission & llvm-mc

Questions?

Bruno Cardoso Lopes
bruno.cardoso@gmail.com