Intro to the LLVM MC Project

The LLVM Machine Code (aka MC) sub-project of LLVM was created to solve a number of problems in the realm of assembly, disassembly, object file format handling, and a number of other related areas that CPU instruction-set level tools work in. It is a sub-project of LLVM which provides it with a number of advantages over other compilers that do not have tightly integrated assembly-level tools.

This blog post talks about how the MC project evolved, describes a few different aspects of it , talks about the improvements/capabilities it brings to LLVM, and finishes off with the current status of the project.

History and Background

Before MC came around, LLVM was already a mature compiler system that support both static compilation (through a normal assembler) and as well as JIT compilation (which emitted encoded instruction bytes directly to memory) on many different targets and sub-targets. However, despite these capabilities, the implementation of these subsystems was not designed wonderfully: why should the X86 instruction encoder (used by the JIT) have to know how to encode the weird pseudo instructions that the X86 code generator uses for its convenience? Why was this code emitter specific to the one code model used by the JIT?

Beyond lack of elegance, it has been a goal for quite a while (for example, see Bruno's 2009 devmtg talk) to directly integrate an assembler into the compiler. Doing so would solve a number of problems: For fast compiles, it is incredibly wasteful for the compiler to carefully format a massive text file, then fork/exec an assembler, which then has to lex/parse/validate the file. In fact, since the rest of Clang is so fast, the assembler takes ~20% of compile time for C code at -O0 -g. In addition to winning back this performance, not relying on an external assembler is attractive for several other reasons: the compiler doesn't have to worry about lots of different buggy assemblers which have similar but inconsistent syntax, Windows users typically don't have assemblers (and MASM is not fully-featured enough to be targeted by a compiler), some systems like FreeBSD have old versions of binutils that lack modern features, etc.

Previously, we tried implementing a "direct object emission" system to allow the code generator to write .o files without an assembler, but this project hit a number of problems. One of the biggest problems is that it tried to re-implement all of the logic in AsmPrinter which handled "lowering" of the codegen IR to machine level constructs. This includes the logic that handled selection of linkage types, section selection, constant lowering and emission, lowering of debug info and exception info, etc. While these may not sound like a big deal, this logic is very delicate and having subtle differences between the .o file writer and .s file writer (e.g. one emitted a strong symbol and one emitting a weak symbol) wasn't acceptable. This previous project never even got to the interesting parts of an assembler, like relaxation support.

Finally, beyond the previous scope of LLVM, there is a bigger toolchain out there that has to deal with things like assembly and disassembly of instructions. When a new instruction set extension comes around (e.g. SSE 4.2) the new instructions need to be added to the assembler, the disassembler, and the compiler. Since LLVM already had a JIT, it already knew how to do encoding of instructions, and we speculated that it might be reasonable to produce an assembler from the textual descriptions of the instructions in the code generator's tables. It would be great to be able to add an instruction description to one place and get the assembler, disassembler, and compiler backend support all at once.

Primary MC Components

You can break down the MC-related data structures and components into "that which operates on instructions" and "that which does other stuff". To provide a consistent abstraction we introduced the new **MCInst** class to represent an instruction with operands (e.g. registers, immediates, etc) which is separate from the code generator's existing notion of an instruction (MachineInstr). The "other stuff" includes a variety of classes like MCSymbol (which represents a label in a .s file), MCSection, MCExpr etc. You can read the headers in the llvm/include/MC directory for more information on these.

These MC classes live at a very low level in the LLVM system, depending only on the support and system libraries. The idea of this is that you can build something low-level (like a disassembler library) that doesn't have to link in the entire LLVM code generator (a disassembler doesn't need a register allocator :).

Given this background, I'll talk about the major components of the MC project: the instruction printer, the instruction encoder, the instruction parser, the instruction decoder, the assembly parser, the assembler backend, and the compiler integration. LLVM already had these (except the instruction decoder and assembly parser), but the existing code has been refactored out and reworked substantially.

Instruction Printer

The instruction printer is a very simple target-specific component that implements the MCInstPrinter API. Given a single MCInst, it formats and emits the textual representation of the instruction to a raw_ostream. Targets can have multiple MCInstPrinters, for example the X86 backend includes an AT&T and an Intel Syntax instruction printer. MCInstPrinters don't know anything about sections, directives, or other things like that so they are independent of object file format.

Instruction printing is somewhat fiddly code (formatting all the operands exactly right, dealing with inconsistencies in the syntax of various instructions, etc) but LLVM already had a TableGen backend that auto-generated much of this from the .td files. Getting an old-style LLVM target to support this requires introducing a new MachineInstr → MCInst lowering pass, which then passes the MCInst to the InstPrinter.

Instruction Encoder

The instruction encoder is another target-specific component which transforms an MCInst into a series of bytes and a list of relocations, implementing the MCCodeEmitter API. The API is quite general, allowing any bytes generated to be written to a raw_ostream. Because the X86 instruction encoding is very complex (aka "completely insane"), the X86 backend implements this interface with custom C++ code that is driven from data encoded in the .td files. This is the only realistic way to handle all the prefix bytes, REX bytes etc, and is derived from the old JIT encoder for X86. We hope and expect that encoders for future RISC targets will be directly generated from the .td files. When used by the compiler, the same MachineInst → MCInst lowering code is used as for the instruction printer.

Instruction Parser

In order to parse instructions when reading a .s file, targets can implement lexers and parsers for their syntax, providing the TargetAsmParser API. The lexer is largely shared code that is parameterized based on common assembly syntax features (e.g. what is the comment character?), but the parser is all target-specific. Once an instruction opcode and its operands have been parsed, it goes through a 'matching' process which decides which concrete instruction is being specified. For example, on X86, "addl $4, %eax" is an instruction with a 1-byte immediate, but "addl $256, %eax" uses a 4-byte immediate field and is represented as a different MCInst.

The output of an parsed instruction is an opcode + list of operands, and the parser also exposes the API for matching. Not all instructions can be matched as they are parsed from a .s file (for example, an immediate size might depend on the distance between two labels) so instructions can be held in a more abstract-than-MCInst representation until relaxation is performed if required.

Instruction Decoder

As you might expect, the instruction decoder implements the MCDisassembler API and turns an abstract series of bytes (implemented with the MemoryObject API, to handle remote disassembly) into an MCInst and a size. As with previous components, a target may implement multiple different decoders (the X86 backend implements X86-16, X86-32, and X86-64 decoders). The decoder turns immediate fields into simple integer immediate operands, if there is a reference to a symbol, the relocation representing the symbol has to be handled by a client of the decoder.

Assembly Parser

The assembly parser handles all the directives and other gunk that is in an .s file that is not an instruction (which may be generic or may be object-file specific). This is the thing that knows what .word, .globl etc are, and it uses the instruction parser to handle instructions. The input to the Assembly parser is a MemoryBuffer object (which contains the input file) and the assembly parser invokes actions of an MCStreamer interface for each thing it does.

MCStreamer is a very important API: it is essentially an "assembler API" with one virtual method per directive and one "EmitInstruction" method that takes an MCInst. The MCStreamer API is implemented by the MCAsmStreamer assembly printer (which implements support for printing these directives to a .s file and uses the Instruction Printer to format the MCInsts) as well as the assembler backend (which writes out a .o file). It is impossible to overstate how important the MCStreamer API is for the overall MC picture to fit together as we'll see later.

Assembler Backend

The assembler backend is one implementation of the MCStreamer API (along with the "MCAsmStreamer" text assembly code emitter) which implements all the "hard stuff" that assemblers do. For example, the assembler has to do "relaxation" which is the process that handles things like branch shortening, situations like "the size of this instruction depends on how far apart these two labels are, but the instruction is between the two labels, how do we break the cycle?" etc. It lays out fragments into sections, resolves instructions with symbolic operands down to immediates and passes this information off to object-file specific code that writes out (say) an ELF or MachO .o file.

Compiler Integration

The final major piece of the puzzle is integrating this all into the compiler. In practice this meant making the compiler talk directly to the MCStreamer API to emit directives and instructions instead of emitting a text file. Converting over all the targets, debug info, eh emission, etc was a major project, and doubly so because it was taken as an opportunity to fix a bunch of major design issues in the AsmPrinter implementations (lots of copy and paste code etc). The new system is much better factored and lowers directly to MC constructs like MCSection and MCSymbol instead of passing around strings and using other ad-hoc data structures.

The end result of this work is that the compiler backend now invokes the same MCStreamer interface to emit code that the stand-alone assembler parser does. This gives us high confidence that using the compiler backend to emit a text file (with the "MCAsmStreamer") and reading it back in with the asmparser will result in the same MCStreamer calls as when the code generator directly invokes them.

Building on these Primitive Components

Now that you know about the major components of the MC ecosystem, and you've seen how key data structures like MCInst and MCStreamer are used to communicate between the components, we can talk about building nice things out of them.

Disassembler Library

One major high-level API built on this is the "enhanced disassembler" API, which uses these components in this sequence:

1. It uses the Instruction Decoder to read machine code bytes in memory into an MCInst and determine instruction sizes.
2. It uses the Instruction Printer to print the textual form of the instruction to a buffer.
3. It uses the Instruction Parser to re-parse the textual form to find precise operand boundaries in the text and build a symbolic form of the operand.

This library provides a number of great capabilities. Sean described it and the X86 implementation in his previous blog post, and the interface is in the `llvm/include/llvm-c/EnhancedDisassembly.h` header.

You can also access some of the functionality on the command line by using the llvm-mc tool:

```
$ echo "0xCD 0x21" | llvm-mc --disassemble -triple=x86_64-apple-darwin9
 int $33
$ echo "0 0" | llvm-mc --disassemble -triple=x86_64-apple-darwin9
 addb %al, (%rax)
$ echo "0 0" | llvm-mc --disassemble -triple=i386-apple-darwin9
 addb %al, (%eax)
$ echo "0x0f 0x01 0xc2" | llvm-mc --disassemble -triple=i386-apple-darwin9
 vmlaunch
```

Stand-alone Assembler

If you combine the Assembly Parser, Instruction Parser, the Assembler Backend and the Instruction Encoder, you get a traditional stand-alone system assembler. The llvm-mc tool provides this capability with a command like:

```
$ llvm-mc foo.s -filetype=obj -o foo.o
$ echo "    invalid_inst _foo, %ebx" | llvm-mc -filetype=obj -o /dev/null
<stdin>:1:5: error: unrecognized instruction
```

```
    invalid_inst _foo, %ebx
    ^
```

The second example shows that the assembly parser gives nice caret diagnostics, not just "parse error on line 1" that many assemblers produce.

Really Complicated Assembly 'cat'

If you want to, you can take the stand-alone assembler and hook the MCAsmStreamer interface up to the MCStreamer the assembler is calling. This allows you to read in a .s file (validating it) and then immediately print it back out. You can also play with options like -show-encoding and -show-inst (which shows MCInsts). Assembly cat is the default mode of llvm-mc for testing's sake, and you can play with it like this:

```
$ llvm-mc foo.s -o bar.s
$ echo "addl %eax, %ebx" | llvm-mc -show-encoding -show-inst
 .section __TEXT,__text,regular,pure_instructions
 addl %eax, %ebx              ## encoding: [0x01,0xc3]
                                        ## <MCInst #66 ADD32rr
                                        ##  <MCOperand Reg:29>
                                        ##  <MCOperand Reg:29>
                                        ##  <MCOperand Reg:27>>
$ echo "xorl _foo, %ebx" | llvm-mc -show-encoding
 .section __TEXT,__text,regular,pure_instructions
 xorl _foo, %ebx              ## encoding: [0x33,0x1c,0x25,A,A,A,A]
                                        ##   fixup A - offset: 3, value: _foo, kind: FK_Data_4
```

The last example shows that the instruction encoder generates a fixup (aka relocation) as well as the data bytes for the instruction.

Assembly Generating Compiler

Now that the compiler is wired up to emit all of its code through the MCStreamer API, we can connect various streamers to it. To get a "classic compiler", we can hook up an MCAsmStreamer to the and get a normal .s file out of the compiler:

```
$ clang foo.c -S -o foo.s
```

This uses the MCAsmStreamer and the Instruction Printers for the current target. One interesting observation is that .s file printing is very performance critical in the old world (when you don't have direct .o file writing support). Generating that huge text file requires a lot of detailed formatting and the compiler isn't doing a lot else at "-O0" mode, so it is a significant percentage of compile time.

However, most build systems (e.g. makefiles) use the compiler in "-c" mode which asks for a .o file. If the compiler supports directly writing a .o file, the .s path isn't hot at all because it isn't being used. This means that we can reasonably make -S mode default to emitting a .s file with tons of helpful comments in it: this mode (e.g. gcc -fverbose-asm) is supported in some form by many compilers, but most people don't know it exists. The presence of direct .o file writing support means that we can turn this on by default! Also, the LLVM implementation of this is pretty nice, with information about spills, loop nests etc.

Compiler-Integrated Assembler

As you might have guessed by now, LLVM also supports an integrated assembler. While this will eventually become the default for targets that support it, right now clang requires you to pass the -integrated-as flag:

```
$ clang foo.c -c -o foo.o -integrated-as
```

To the user, this works just like the non-integrated-as support. If you pass -v though, you'll see that the compiler is directly emitting the .o file and the assembler isn't run by the driver.

This works very simply: the assembler backend is provided as the MCStreamer to the compiler, and it handles the various things the system assembler used to take care of. This means that the assembler never has to (for example) print out MCInsts to a textual representation and reparse them, it just passes the well-formed MCInsts the code generator is using directly to the assembler backend.

This is all well and good except for one thing: inline assembly. With inline assembly, arbitrary assembly language can be included in the source file (optionally munged by the compiler to insert operand constraint information etc), and we have no choice but to parse this with the assembly parser. Fortunately, we have a nice clean and well decomposed system, so we the assembly handling code just creates a new temporary assembly parser to handle the inline asm, and has it talk to the same MCStreamer instance that the compiler is using. This means that the current state of the streamer is the same when the inline assembly is parsed as it would be if the system assembler is used.

Using an integrated has other benefits as well: we can tightly integrate it with the clang diagnostics system instead of having the assembler detect the error late and report it with a foreign error. Beyond consistency, we can actually give really good information about where the error occurs, relaying it back to where in the *original source code* the problem is. Compare these examples with and without the integrated assembler:

```
$ cat t.c
int foo(int X) {
 __asm__ ("frob    %0" : "+r" (X));
 return X;
}
$ gcc t.c
/var/folders/51/51Qw875vFdGa9KojoIi7Zk+++TM/-Tmp-//ccyXfgfZ.s:11:no such instruction: `frob %eax'
$ clang t.c
/var/folders/51/51Qw875vFdGa9KojoIi7Zk+++TM/-Tmp-/cc-4zimMn.s:15:no such instruction: `frob %eax'
clang: error: assembler command failed with exit code 1 (use -v to see invocation)
$ clang t.c -integrated-as
t.c:2:11: error: unrecognized instruction
        frob    %eax
        ^
<inline asm>:1:2: note: instantated into assembly here
 __asm__ ("frob    %0" : "+r" (X));
          ^
1 error generated.
```

I don't know about you, but I find it much more useful to know that line 2 of my C file is the problem than to find out that line 15 of a deleted file is. :) It is worth pointing out that it is incredibly important to show the code being parsed by the assembler, because after operand substitution etc it is not always clear from the source code what the problem is.

Status and Open Projects!

As described above, the architecture of the MC subsystems is already quite advanced and we have a lot of code written. However, not all of it is production quality and we don't yet have people signed up to do all the interesting work! As of this writing here is the high level status:

1. All the clang-side work is done: the driver knows -integrated-as, the diagnostics subsystem can handle assembler errors, the code generator is fully switched over to use MCStreamer, etc.
2. The X86 and ARM disassemblers are in good shape and considered to be at or near production quality.
3. The X86 instruction printer is done for both AT&T and Intel Syntax.
4. The MC components have been designed to be object file independent (e.g. work for MachO, ELF, PE-COFF etc) but only have an implementation for MachO (used on Mac OS/X systems). Adding support for ELF and/or PE-COFF should be reasonably straight-forward.
5. The X86 AT&T-syntax assembly parser is missing some important features, e.g. forward/backward labels, support for inferring suffixes on instructions (e.g. add → addl), macros etc. However, it is sufficiently advanced to reliably parse everything generated by the LLVM X86 backend. Adding these missing features should be quite straight-forward.
6. The X86 Intel-syntax assembly parser doesn't exist.
7. The ARM assembly parser is not complete (probably 50% to useful), and the instruction printer is about 50% refactored from the old-style printer.
8. The assembler backend works quite well and currently tries to produces bitwise identical object files to the system assembler on the Mac (to allow validating through "/usr/bin/cmp"). That said, it's algorithms aren't all performance tuned, so it could probably be substantially faster.
9. The old JIT code path is still alive in LLVM, it should be killed off once it is replaced with a new MC implementation. One win of this is that the JIT will support inline assembly :)

If you're interested in this level of the tool chain, this is a great area to get involved in, because there are lots of small to mid-sized projects just waiting to be tackled. I believe that the long term impact of this work is huge: it allows building new interesting CPU-level tools and it means that we can add new instructions to one .td file instead of having to add them to the compiler, the assembler, the disassembler, etc.

-Chris