Mini-Tutorial: How to implement an LLVM Assembler

Simon Cook

2013 European LLVM Conference, Paris

This presentation

- Inspired by previous tutorials.
- Covering some of the details easily tripped up on.
- Using the *OpenRISC 1000* backend as an example where needed.
- More detailed version of this available in *Embecosm Application Note 10: LLVM Integrated Assembler*
  - *http://www.embecosm.com/appnotes/ean10/ean10-howto-llvmas-1.0.pdf*
- Source used as demonstration in GitHub:
  - *https://github.com/simonpcook/llvm-or1k*

Motivation for MC Based Assembler

- General (Simplified) Compiler Workflow.
  - `clang -target=foo -c bar.c`
  - Front End converts C to IR
  - Back End lowers IR to foo's instruction set
  - Carefully format `.s` file
  - Assembler parses `.s`, generates object
- Key Idea: More efficient to directly generate the object file within the compiler.
- Additionally: We already defined our instruction set, why define it again?

4 Steps to Assembler Success

1. Parsing Instructions
2. Encoding Instructions
3. Decoding Instructions
4. Generating Object File (in our case ELF)

But First... *Foo*InstrInfo.td

- Instruction definitions need to link printable instruction to encoding.
  - `field bits<`*n*`> Inst;`
  - `Inst` field used with TableGen to get you 95% of the way by building instruction encoding/decoding tables.
- Set bits for instruction opcodes/etc. and fields filled in by backend.

Reduced *or1k* Example

```
class InstOR1K<dag outs, dag ins, string asmstr, list<dag> pattern> :
Instruction {  field bits<32> Inst; bits<2> optype;
  bits<4> opcode;
  let Inst{31-30} = optype;  let Inst{29-26} = opcode;
}

class InstRR<bits<4> op, dag outs, dag ins, string asmstr, list<dag>
pattern>
  : InstOR1K<outs, ins, asmstr, pattern> {
  let optype = 0b11;
  let opcode = op;
}
class ALU_RR<bits<4> subOp, string asmstr, list<dag> pattern>
  : InstRR<0x8, (outs GPR:$rD), (ins GPR:$rA, GPR:$rB),
           !strconcat(asmstr, "\t$rD, $rA, $rB"), pattern> {
  bits<5> rD;  bits<5> rA;  bits<5> rB;
  let Inst{25-21} = rD;  let Inst{20-16} = rA;  let Inst{15-11} = rB;
  let Inst{9-8} = op2;  let Inst{3-0} = op3;

def ADD  : ALU1_RR<0x0, "l.add", add>;
```

4 Steps to Assembler Success

1. Parsing Instructions
2. Encoding Instructions
3. Decoding Instructions
4. Generating Object File (in our case ELF)

Assembly Parsing

- Turns instruction strings into MC representations.
- Need to implement two classes:
  - *Foo*`Operand` – stores operand information and type
    - e.g. "register", "2"
  - *Foo*`AsmParser` – uses TableGen information to check validity, but need to write functions for parsing operands and creating *Foo*`Operands`.
    - `valid`*`OpType`* `? create`*`OpType`* `: return 0;`
    - In `ParseInstruction`: If your instruction mnemonics are of the form `l.add`, the string needs parsing to form `[l, .add]`.

4 Steps to Assembler Success

1. Parsing Instructions
2. Encoding Instructions
3. Decoding Instructions
4. Generating Object File (in our case ELF)

Instruction Encoding

- To encode instructions, the class *`Foo`*`MCCodeEmitter` needs implementing providing the following functionality:
  - Target operand encodings
    - `getMachineOpValue` for registers and immediates with no fixups.
  - Byte emitting (for current endianness)
    - Emit in `EncodeInstruction` after calling TableGen `getBinaryCodeForInstr`.
  - Custom register function (in some cases)

Encoding Custom Operands

- Custom operands need encoding manually.
- Specify `EncoderMethod` in operand definition.
- Encoding is done within l.s.*n* bits, regardless of final dest.

```
unsigned OR1KMCCodeEmitter::
getMemoryOpValue(const MCInst &MI, unsigned Op) const {
  unsigned encoding;
  const MCOperand op1 = MI.getOperand(1);
  assert(op1.isReg() && "First operand is not register.");
  encoding = (getOR1KRegisterNumbering(op1.getReg()) << 16);

  MCOperand op2 = MI.getOperand(2);
  assert(op2.isImm() && "Second operand is not immediate.");
  encoding |= (static_cast<short>(op2.getImm()) & 0xffff);
  return encoding;
}
```

4 Steps to Assembler Success

1. Parsing Instructions
2. Encoding Instructions
3. Decoding Instructions
4. Generating Object File (in our case ELF)

Instruction Decoding

- Whilst not needed to assemble, generally also useful.
- To decode, implement `*foo*Disassembler`, centered around `getInstruction`.
  - General flow of function:
    1. Read *N* bytes of memory.
    2. Call generated `decode*foo*Instruction*n*`.
    3. Return instruction.
  - In the case of variable length instructions, the approach is to loop the above, e.g. try 16-bit insns, then 32-bit.
- Operands are added with instructions `addOperand` function.

EMBECOSM®

Decoding Tables with TableGen

- For disassembling to succeed, each possible encoding must map to only one instruction.
- Otherwise, build fails:

```
Decoding Conflict:
        010001..........................
        ................................
    JR  010001__________________________
    RET 010001__________________________
```

- Conflicts can be solved by providing context as to when to use each instruction.
  - Simplest (when useful) is to declare instructions as `isPsuedo = 1` or `isCodeGen = 1`.

4 Steps to Assembler Success

1. Parsing Instructions
2. Encoding Instructions
3. Decoding Instructions
4. Generating Object File (in our case ELF)

EMBECOSM®

Writing ELF Objects

- To write ELF objects, `*foo*ELFObjectWriter` and `*foo*AsmBackend` need implementing.
  - AsmBackend responsible for applying fixups when information is available via `applyFixup`, `adjustFixupValue` and `writeNopData`.
  - ElfObjectWriter responsible for fixup to reloc conversion
- Other support definitions
  - Relocations in `include/llvm/Support/ELF.h`
  - Fixups in `*foo*FixupKinds.h`.
- `createObjectWriter/createFooMCStreamer` instantiates all of the above.

Done

- You should now be able to test your new assembler

☺

- To test your assembler with clang

  -
    ```
    clang -target or1k -integrated-as helloworld.c
    ```

Thank you

www.embecosm.com