Howto: Implementing LLVM Integrated Assembler

A Simple Guide

Simon Cook
Embecosm

Application Note 10. Issue 1
Publication date October 2012

Table of Contents

List of Figures

Chapter 1. Introduction

The *LLVM* machine code project is part of the *LLVM* compiler suite and designed for aiding in the assembly and disassembly of native instructions as well as the handling of object files.

1.1. Target Audience

This guide is for developers who are already comfortable with the LLVM build system and the TableGen language/tool for the creation and manipulation of target-specific parameters.

This guide assumes that the basic LLVM toolchain has already been implemented and is functioning for the target architecture using a MC (*LLVM* Machine Code) based instruction printer. (i.e. LLVM is already capable of producing assembly files for the target architecture via **llc**.)

The detail of steps here have been tested using the (at time of writing) latest development version of LLVM for the OpenRISC 1000, the source of which can be found on GitHub at www.github.com/openrisc/llvm-or1k.

1.2. Examples

This application note includes examples from the LLVM backend for *OpenRISC 1000* , originally written by Stefan Kristiansson and extended by Simon Cook of Embecosm for the implementation of the integrated assembler.

Examples used are definitions of instructions, methods for their encoding and decoding, written by the same authors.

At the time of writing the OpenRISC 1000 implementation is not part of the main LLVM distribution, but the source can be downloaded and compiled from www.github.com/openrisc/llvm-or1k.

Instructions for building and testing can be found in the file **`README.or1k`**.

General examples of source code, file names, etc. will use the term ***`arch`*** in italics. An implementation of the LLVM assembler would replace this with the name of the target directory. For example, with the case of *OpenRISC 1000*, ***`arch`*** would be replaced with **`OR1K`**.

1.3. Further Information

The main source of information regarding LLVM is the LLVM website (www.llvm.org). This website hosts the main documentation (www.llvm.org/docs), including instructions on how to implement various parts of a LLVM backend.

In addition, the LLVM API documentation at www.llvm.org/doxygen contains useful information about available APIs and in many cases is the best source for information.

There is also a mailing list llvmdev@cs.uicu.edu and IRC channel #llvm on irc.oftc.net where questions about LLVM development can be asked.

1.4. About Embecosm Application Notes

Embecosm publishes a series of free and open source application notes, designed to help working engineers with practical problems.

Feedback is always welcome, which should be sent to `<info@embecosm.com>`.

Chapter 2. Integrated Assembler within LLVM

The integrated assembler when used with the rest of the LLVM system allows source to be compiled directly to a native object file without the need of outputting assembly instructions to a file and then parsing them back in order to encode them.

This provides the benefit of faster compiling, and when combined with the C language compiler *clang*, allows C/C++ to native object file compilation in one step ready for linking.

This application note focuses on the LLVM assembler in the case where its intended use is as part of the *clang* C compiler. As such, assembly parsing, encoding and decoding is restricted to the case where purely instructions are given to the **llvm-mc** tool described below. It is however possible to extend the assembler to accept and reconstruct assembly directives, allowing **llvm-mc** to behave more like a standalone compiler.

No special steps/changes are required to allow *clang* to use the integrated assembler, once it is set up and functioning, it can be invoked by using the **`-integrated-as`** as one of the arguments, alongside with the target architecture, as demonstrated below.

```
clang -target or1k -O3 -integrated-as helloworld.c
```

2.1. llvm-mc

The **llvm-mc** tool, described as the "*LLVM* machine code playground", is used for testing components of a target's MC implementation. The main task this tool fulfills is to assemble a .s file (via the **`-assemble`** command), disassemble strings of bytes (**`-disassemble`**), as well as show the encoding of instructions and their internal representations (**`-show-encoding`** and **`-show-inst`** respectively).

In order to determine the architecture, the parameter **`-arch=`*`arch`*** is used and **`-mattr=`*`a1,+a2,-a3`*** is used to enable and disable features of the processor required for instructions to be valid.

An example of the above for the *OpenRISC 1000* architecture is the optional support for multiplication and division instructions. To enable these, a command like the following would be used.

```
llvm-mc -assemble -show-encoding -arch=or1k -mattr=+mul,+div input.s
```

Chapter 3. Implementing Assembly Parser Support

The first component which needs to be implemented is support for parsing assembly files. This allows **llvm-mc** to correctly read in assembly instructions and provide an internal representation of these for encoding.

The assembly parser is configured as a separate library which has the initial target implementation library as its parent. This library is placed in the **lib/Target/*arch*/AsmParser** directory.

3.1. LLVM Build System

As with all libraries, the assembly parser library needs declaring within the build system so that when compiled, the functionality this library provides is added.

This consists of a **Makefile** for when **make** is used for the build, a **CMakeLists.txt** file for when **cmake** is used instead and a **LLVMBuild.txt** file for the rest of build system.

These files declare which libraries are required and need to be compiled first. In this case the TableGen output for the architecture needs to be generated first so that function generated can be used within the library.

For example, the build files for the *OpenRISC 1000* implementation are as follows.

```
;===- ./lib/Target/OR1K/AsmParser/LLVMBuild.txt ----------------*- Conf -*--===;
;
;                     The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;===------------------------------------------------------------------------===;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;===------------------------------------------------------------------------===;

[component_0]
type = Library
name = OR1KAsmParser
parent = OR1K
required_libraries = MC MCParser Support OR1KDesc OR1KInfo
add_to_library_groups = OR1K
```

Figure 3.1. **LLVMBuild.txt**

```
add_llvm_library(LLVMOR1KASMParser
  OR1KAsmParser.cpp
  )

add_dependencies(LLVMOR1KAsmParser OR1KCommonTableGen)
```

Figure 3.2. `CMakeLists.txt`

```
##===- lib/Target/OR1K/AsmParser/Makefile ------------------*- Makefile -*-===##
#
#                     The LLVM Compiler Infrastructure
#
# This file is distributed under the University of Illinois Open Source
# License. See LICENSE.TXT for details.
#
##===----------------------------------------------------------------------===##
LEVEL = ../../../..
LIBRARYNAME = LLVMOR1KAsmParser

# Hack: we need to include 'main' or1k target directory to grab private headers
CPP.Flags += -I$(PROJ_OBJ_DIR)/.. -I$(PROJ_SRC_DIR)/..

include $(LEVEL)/Makefile.common
```

Figure 3.3. `Makefile`

The final parts of the build system that need modifying is adding the new library to its parent.

In the **`CMakeLists.txt`** file for the target, the TableGen assembly matcher needs to be generated and the AsmParser directory added. The equivalent variables also need changing in the **`Makefile`**. For *OpenRISC 1000* this would be as follows.

```
tablegen(LLVM OR1KGenAsmMatcher.inc -gen-asm-matcher)
add_subdirectory(AsmParser)
```

Figure 3.4. `CMakeLists.txt`

```
 # Make sure that tblgen is run, first thing.
 BUILT_SOURCES = OR1KGenRegisterInfo.inc OR1KGenInstrInfo.inc \
-                OR1KGenAsmWriter.inc OR1KGenDAGISel.inc \
+                OR1KGenAsmWriter.inc OR1KGenAsmMatcher.inc OR1KGenDAGISel.inc \
                 OR1KGenSubtargetInfo.inc OR1KGenCallingConv.inc

-DIRS = InstPrinter TargetInfo MCTargetDesc
+DIRS = AsmParser InstPrinter TargetInfo MCTargetDesc
```

Figure 3.5. `Makefile`

Finally, in the **LLVMBuild.txt** file for the target library, the parameter **has_parser** is defined as 1.

3.2. *arch*ASMParser Class

The assembly parser consists of one source file, ***arch*ASMParser.cpp** which contains the ***arch*ASMParser** class, inheriting from **MCTargetAsmParser** in addition to a second class for holding target-specific operand information.

Note

Information about this class can be found in *LLVM*'s documentation at llvm.org/docs/doxygen/html/classllvm_1_1MCTargetAsmParser.html

The primary class consists of the **MatchAndEmitInstruction** function which is called for each instruction to be parsed, emitting out an internal representation of each instruction as well as supporting functions which help it parse instruction operands.

It should be noted that the following two lines appear in the class declaration to import functions generated by TableGen which will do most of the heavy lifting in the system.

```
#define GET_ASSEMBLER_HEADER
#include "archGenAsmMatcher.inc"
```

3.3. *arch*Operand Structure

Before the instruction matcher can be created the *arch*Operand structure must be defined. This represents a parsed machine instruction, storing information about the types and contents of its operands.

Note

This class inherits from **MCParsedAsmOperand**, whose API can be found at llvm.org/docs/doxygen/html/classllvm_1_1MCParsedAsmOperand.html.

Within this structure is an enum and union which store the operand along with its type.

In the case for *OpenRISC 1000*, these operand types are tokens, registers and immediate. Similarly, **SMLoc**s (source code locations) are stored for the start and stop locations of the operand.

```
    enum KindTy {
      Token,
      Register,
      Immediate
    } Kind;

    SMLoc StartLoc, EndLoc;

    union {
      struct {
        const char *Data;
        unsigned Length;
      } Tok;
      struct {
        unsigned RegNum;
      } Reg;
      struct {
        const MCExpr *Val;
      } Imm;
    };
```

The majority of functions within this struct are simply getters and setters for the different operands stored in the object. For the getters, asserts are added to check that the operand is of the correct type before returning its value. An example is shown below.

```
    StringRef getToken() const {
      assert (Kind == Token && "Invalid type access!");
      return StringRef(Tok.Data, Tok.Length);
    }
```

Generator functions called **CreateToken**, **CreateReg** etc. are defined which take the data type for the operand as well as the start and end locations of the operand (with the exception of tokens which only take a start location). These functions then create a new object for the provided operand details, set its contents and then returns it.

The final functions in this structure add operands to a particular instruction. These use the **addOperand** function of a **MCInst** to add the relevant operand. For registers, the **getReg()** is simply used. Immediates however use a more complex method where if it is possible to add the instruction as an immediate it is done so, otherwise it is added as an expression, as demonstrated below.

```
    void addExpr(MCInst &Inst, const MCExpr *Expr) const {
      // Add as immediates where possible. Null MCExpr = 0
      if (Expr == 0)
        Inst.addOperand(MCOperand::CreateImm(0));
      else if (const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(Expr))
        Inst.addOperand(MCOperand::CreateImm(CE->getValue()));
      else
        Inst.addOperand(MCOperand::CreateExpr(Expr));
    }
```

3.4. Matching and Emitting Instructions

With the OR1KOperand structure defined, the main **`MatchAndEmitInstruction`** can be implemented, calling on the **`MatchInstructionImpl`** generated by TableGen to do the hard work.

When called, **`MatchInstructionImpl`** will use a given **`MCInst`** to store details on the instruction if possible, returning an error value. If the instruction was successfully parsed, the value **`Match_Success`** is returned and the instruction can be simply emitted via the provided **`MCStreamer`**.

If however there is a problem parsing the instruction, the return code will be different and set depending on where the problem occurred. For example **`Match_MissingFeature`** is returned if a required target feature is not enabled and **`Match_MnemonicFail`** is returned if the instruction mnemonic is not recognized.

These return codes can be used to generate a useful error message, though the simplest case would be just to state than an error occurred and then return. This case is demonstrated below.

```
bool OR1KAsmParser::
MatchAndEmitInstruction(SMLoc IDLoc,
                        SmallVectorImpl<MCParsedAsmOperand*> &Operands,
                        MCStreamer &Out) {
  MCInst Inst;
  SMLoc ErrorLoc;
  unsigned ErrorInfo;

  if (MatchInstructionImpl(Operands, Inst, ErrorInfo)) {
    Out.EmitInstruction(Inst);
    return false;
  }
  else
    return Error(IDLoc, "Error parsing instruction");
}
```

3.5. Parsing Registers and Immediates

The rest of the functions within ***arch*`ASMParser`** handle parsing particular operand types.

To parse the name of a register, a **`ParseRegister`** function is created. The lexer reading the file is asked for which type of token the function has been given. A register is an identifier (i.e. short string, e.g. **`r0`**). To firstly filter out incorrect token types, if it is not an identifier, the function simply returns 0 to indicate that it was unable to parse it.

If the token is an identifier, it is then given to the TableGen generated **`MatchRegisterName`** function. This returns a value corresponding to the register if it is valid, otherwise 0 is returned.

For example if **`MatchRegisterName`** is given an instruction mnemonic, which is obviously incorrect but also an identifier, 0 will be returned, so checking for an error is key.

Note

Register values start with a value of 1, so in the case of an architecture like *OpenRISC 1000* which uses r0, r1, etc. to name registers, r0 will be represented by 1, r1 by 2, etc.

In addition, this numbering convention is specified by TableGen so it is not guaranteed that register numbers are the same between different compiled versions of *LLVM*.

If a match was made, the lexer then consumes the token that was used in the match, preparing it for the next operand or instruction. Finally an ***arch*****Operand** is created for the register using the **CreateReg** function defined above and then returned.

```
OR1KOperand *OR1KAsmParser::ParseRegister(unsigned &RegNo) {
  SMLoc S = Parser.getTok().getLoc();
  SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() -1);

  switch(getLexer().getKind()) {
    default: return 0;
    case AsmToken::Identifier:
      RegNo = MatchRegisterName(getLexer().getTok().getIdentifier());
      if (RegNo == 0)
        return 0;
      getLexer().Lex();
      return OR1KOperand::CreateReg(RegNo, S, E);
  }
  return 0;
}
```

The same method is applied when parsing immediates. In this case any integer, plus or minus is accepted as a valid token type for the operand. If the type matches, then a **MCParser** is used to calculate the immediate via the **ParseExpression** function.

If the expression returned is valid it is then evaluated as an absolute value, with an ***arch*****Operand** being created and returned as before.

3.6. Parsing Operands

The **ParseOperand** function makes use of the two previously defined functions in order to parse an operand of which type is originally unknown.

Note

This section describes how this function is used in the *OpenRISC 1000* implementation, specifically for how it handles memory operands. Other architectures will need to modify this function to match the needs and types of operands expected to be found in an assembly output.

In the *OpenRISC 1000* architecture, there are three types of operand which need parsing by this method, registers, immediates and memory operands which are of the form **imm(reg)**.

Firstly attempts are made to parse an operand as a register, using the previously defined **ParseRegister** function. If this succeeds then the operand is added to the list of operands for the instruction and the function returns.

If this does not work (the operand is not a register), an attempt is then made to parse the operand as an immediate. Should the immediate operand successfully be parsed, then it should be considered as a memory operand first, before placing it into the list of operands.

As the form of a memory operand in *OpenRISC 1000* is of the form **imm(reg)**, then the next token is evaluated to determine if it is a the start of a memory operand. If this type matches (i.e. it is a left parenthesis, the next token is evaluated as a register. Finally should the last token be a right parenthesis, then a memory operand has instead been parsed.

Should parsing as a memory operand succeed, the two components are added to the operand list whereas if the first test failed (the first operand was not an left parenthesis) just the immediate is added.

If however, no valid operand was found (either not a valid type or memory parsing failed after identifying a left parenthesis), then an error is created and returned instead.

```
bool OR1KAsmParser::
ParseOperand(SmallVectorImpl<MCParsedAsmOperand*> &Operands) {
  OR1KOperand *Op, *Op2 = 0;

  // Attempt to parse token as register
  unsigned RegNo;
  Op = ParseRegister(RegNo);

  // Attempt to parse token as immediate
  if (!Op) {
    Op = ParseImmediate();

    // If next token is left parenthesis, then attempt to parse as memory
    // operand
    if (Op)
      if (getLexer().is(AsmToken::LParen)) {
        getLexer().Lex();
        // Swap tokens around so that they can be parsed
        Op2 = Op;
        Op = ParseRegister(RegNo);

        // Invalid memory operand, fail
        if (!Op || getLexer().isNot(AsmToken::RParen)) {
          Error(Parser.getTok().getLoc(), "unknown operand");
          return true;
        }
        getLexer().Lex();
      }
  }

  // If the token could not be parsed then fail
  if (!Op) {
    Error(Parser.getTok().getLoc(), "unknown operand");
    return true;
  }

  // Push back parsed operand(s) into list of operands
  Operands.push_back(Op);
  if (Op2)
    Operands.push_back(Op2);

  return false;
}
```

3.7. Parsing Instructions

The final function which needs defining is **`ParseInstruction`**, which parses the instruction mnemonic, followed by all the operands until the end of the statement is reached, called by **`MatchInstructionImpl`** to identify all operands for matching.

The instruction mnemonic is parsed by a TableGen generated function, but is first split into sections separated by the dots in the mnemonic, with each part being added to the instructions operand list.

For example the mnemonic `l.add` would become `[l, .add]`, `lf.add.s` would become `[lf, .add, .s]`, etc.

Once the mnemonic has been split up and added to the operand list, the **`ParseOperand`** function defined above is repeatedly called to parse the next operand until the end of the statement is reached (`AsmToken::EndOfStatement`), with commas being consumed between operands.

```
bool OR1KAsmParser::
ParseInstruction(StringRef Name, SMLoc NameLoc,
                 SmallVectorImpl<MCParsedAsmOperand*> &Operands) {
  // First operand is token for instruction
  size_t dotLoc = Name.find('.');
  Operands.push_back(OR1KOperand::CreateToken(Name.substr(0,dotLoc),NameLoc));
  if (dotLoc < Name.size()) {
    size_t dotLoc2 = Name.rfind('.');
    if (dotLoc == dotLoc2)
      Operands.push_back(OR1KOperand::CreateToken(Name.substr(dotLoc),NameLoc));
    else {
      Operands.push_back(OR1KOperand::CreateToken(Name.substr
                                  (dotLoc, dotLoc2-dotLoc), NameLoc));
      Operands.push_back(OR1KOperand::CreateToken(Name.substr
                                  (dotLoc2), NameLoc));
    }
  }

  // If there are no more operands, then finish
  if (getLexer().is(AsmToken::EndOfStatement))
    return false;

  // Parse first operand
  if (ParseOperand(Operands))
    return true;

  // Parse until end of statement, consuming commas between operands
  while (getLexer().isNot(AsmToken::EndOfStatement) &&
        getLexer().is(AsmToken::Comma)) {
    // Consume comma token
    getLexer().Lex();

    // Parse next operand
    if(ParseOperand(Operands))
      return true;
  }

  return false;
}
```

3.8. Testing

Note

These tests are not needed to fully test the integrated assembly implementation and will be superseded by future tests, but are explained here for testing the assembly parser if implemented separately from the other components.

Tests for assembly parsing should consist of an instance of each instruction in the target's instruction set, along with the expected output from **llvm-mc -show-inst**. An example is shown and explained below.

```
# RUN: llvm-mc -arch=or1k -show-inst %s | FileCheck %s

    l.add r1, r2, r3
# CHECK: <MCInst #{{[0-9]+}} ADD
# CHECK-NEXT: <MCOperand Reg:2>
# CHECK-NEXT: <MCOperand Reg:3>
# CHECK-NEXT: <MCOperand Reg:4>>
```

Each test consists of the instruction, which in this case is an add, followed by a **`CHECK`** line for the mnemonic and for each operand. The first **`CHECK`** line searches for the mnemonic, whose name was set in the TableGen declaration, in this case **`ADD`**. It does not matter about the internal value of this operand, so a search for **`{{[0-9]+}}`** matches any value as valid.

Next each operand is tested; **`CHECK-NEXT`** ensures that a test only passes if the line immediately follows the previous check line, i.e. that the operands specified belong with the instruction mnemonic specified in the previous line. In this case, registers 1, 2 and 3 were specified in the instruction then tests for register number 2, 3 and 4 are carried out, which are the internal representation of these registers.

The same is applied for immediate operands, except that the **`MCOperand`** type is **`Imm`** and it is expected that the immediate is the same as that in the instruction, not increased by one.

For *OpenRISC 1000*, tests for memory operands are carried out by checking for the register operand first and then the immediate on the line that follows (i.e. the same order that they were pushed to the operands when defining **`ParseInstruction`**).

Tests can be called in the usual way and should all pass if instructions are well defined in TableGen.

Chapter 4. Implementing Instruction Encoding

The next stage in implementing the LLVM assembler is to provide support for encoding instructions into their native bit patterns.

4.1. Build System

Unlike the assembly parser, no extra library is needed in order to encode instructions, as the instruction encoder lives in the target's **`MCTargetDesc`** directory.

As such, the only changes that need to be made to the build system are those that tell LLVM which files need to be compiled and generated by TableGen. In this case, the files are the ***`arch`*****`MCCodeEmitter`** C++ source file and the generated ***`arch`*****`GenCodeEmitter.inc`**.

4.2. Registration

To register the code emitter, the following code needs adding to the machine code target description ***`arch`*****`MCTargetDesc.cpp`**.

```
    // Register the MC code emitter
    TargetRegistry::RegisterMCCodeEmitter(ThearchTarget,
                                          llvm::createarchMCCodeEmitter);
```

The **`create`*****`arch`*****`MCCodeEmitter`** function is declared in the ***`arch`*****`MCTargetDesc.h`** header file and is defined in ***`arch`*****`MCCodeEmitter.cpp`**. This function simply creates and returns a new ***`arch`*****`MCCodeEmitter`**.

4.3. Register Support Function

Before the instruction encoding class can be implemented, it is useful to implement a function that will convert a register symbol to its register value.

A version of this function is generated by TableGen, though it is dependent on the order that registers are defined in a register class (assuming the first register defined is encoded as zero, etc.). Therefore if the register class does not match this (e.g. the class is defined in a different order for allocation purposes), then a custom function is required.

The register number support function is a simple switch statement which returns the encoding of a register to be used in an instruction. The default case should be a call to llvm_unreachable to warn of a problem where the encoding of an invalid register is requested.

The following example demonstrates how the function looks for the *OpenRISC 1000* implementation.

```
  static inline unsigned getOR1KRegisterNumbering(unsigned Reg) {
    switch(Reg) {
      case OR1K::R0  : return 0;
      case OR1K::R1  : return 1;
      ... other cases not shown ...
      case OR1K::R31 : return 31;
      default: llvm_unreachable("Unknown register number!");
    }
```

4.4. Encoding Instructions

Instructions are encoded through an implementation of the **`MCCodeEmitter`** class, which uses information about the target and encodes instructions through the **`EncodeInstruction`** function, streaming encoded bytes through a provided output stream.

Note

Information about this class can be found in *LLVM*'s documentation at llvm.org/docs/doxygen/html/classllvm_1_1MCCodeEmitter.html

This class requires only the **`EncodeInstruction`** function to be defined. However other functions should also be defined to assist in encoding instructions.

The key function here is **`getBinaryCodeForInstr`**, which is generated by TableGen. It takes a provided instruction and with the instruction encoding information defined with the instructions to generate the encoded instruction.

```
// getBinaryCodeForInstr - TableGen'erated function for getting the
// binary encoding for an instruction.
uint64_t getBinaryCodeForInstr(const MCInst &MI) const;
```

Other functions that are defined in this class (though not necessary), are support functions for outputting a number of bytes and constants (encoded instruction) with the correct endianness for the target. Examples of these functions are provided below.

In addition, functions are required to assist in encoding custom operand types, should their encoding not be known from their TableGen definition.

```
// Emit one byte through output stream (from MCBlazeMCCodeEmitter)
void EmitByte(unsigned char C, unsigned &CurByte, raw_ostream &OS) const {
  OS << (char)C;
  ++CurByte;
}

// Emit a series of bytes (from MCBlazeMCCodeEmitter)
void EmitConstant(uint64_t Val, unsigned Size, unsigned &CurByte,
                  raw_ostream &OS) const {
  assert(Size <= 8 && "size too big in emit constant");

  for (unsigned i = 0; i != Size; ++i) {
    EmitByte(Val & 255, CurByte, OS);
    Val >>= 8;
  }
}
```

The generated **`getBinaryCodeForInstr`** function requires one other function to be defined, **`getMachineOpValue`**, which provides the encoding for the default operand types (registers and immediates) where no relocation is required.

This function first checks the type of operand. If it is a register, then the custom function for retrieving register numbers defined above is called to get the encoding of this register. If instead it is an immediate, then it is cast to an unsigned value which is then returned.

Finally, if the operand is an expression (which is the case where relaxation is required), then information about this relocation is stored in a fixup, with 0 being returned as the encoding at this point.

```
unsigned OR1KMCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                  SmallVectorImpl<MCFixup> &Fixups) const {
  if (MO.isReg())
    return getOR1KRegisterNumbering(MO.getReg());
  if (MO.isImm())
    return static_cast<unsigned>(MO.getImm());

  // MO must be an expression
  assert(MO.isExpr());

  const MCExpr *Expr = MO.getExpr();
  MCExpr::ExprKind Kind = Expr->getKind();

  if (Kind == MCExpr::Binary) {
    Expr = static_cast<const MCBinaryExpr*>(Expr)->getLHS();
    Kind = Expr->getKind();
  }

  assert (Kind == MCExpr::SymbolRef);

  OR1K::Fixups FixupKind = OR1K::Fixups(0);

  switch(cast<MCSymbolRefExpr>(Expr)->getKind()) {
    default: llvm_unreachable("Unknown fixup kind!");
      break;
    case MCSymbolRefExpr::VK_OR1K_PLT:
      FixupKind = OR1K::fixup_OR1K_PLT26;
      break;
    ... other cases not shown ...
  }

  // Push fixup (all info is contained within)
  Fixups.push_back(MCFixup::Create(0, MO.getExpr(), MCFixupKind(FixupKind)));
  return 0;
}
```

The main **EncodeInstruction** function simply takes a provided instruction, passing it to the TableGen **getBinaryCodeForInstr** function, returning the encoded version of the instruction, which is then emitted via the support functions for outputting constants which were defined above.

Note

In the case of *OpenRISC 1000*, all instructions are 32 bits in length, therefore the same amount of data is outputted in all cases. Where an instruction set has variable length instructions, then testing the opcode would be required to determine the length of instruction to emit.

```
void OR1KMCCodeEmitter::
EncodeInstruction(const MCInst &MI, raw_ostream &OS,
                         SmallVectorImpl<MCFixup> &Fixups) const {
  // Keep track of the current byte being emitted
  unsigned CurByte = 0;
  // Get instruction encoding and emit it
  ++MCNumEmitted;       // Keep track of the number of emitted insns.
  unsigned Value = getBinaryCodeForInstr(MI);
  EmitConstant(Value, 4, CurByte, OS);
}
```

4.5. Encoding Custom Operands

Depending on the operand types defined in the architecture, custom encoding function may be required in order to encode these more complex types.

One example of an operand that may require custom encoding is the *OpenRISC 1000* memory operand which combines a register with an immediate offset. This is used for example with the **l.lwz** instruction, which loads a word from memory from a location specified by a register as a pointer plus some immediate offset stored in the instruction.

```
    l.lhz r1, 4(r2)
```

If an operand requires custom encoding, then **EncoderMethod** has to be specified in the operand TableGen definition, stating which function is used to encode the operand.

```
def MEMri : Operand<i32> {
  let PrintMethod = "printMemOperand";
  let EncoderMethod = "getMemoryOpValue";
  let MIOperandInfo = (ops GPR, i32imm);
}
```

Note

It does not matter where in an instruction an operand appears, encoding acts within the bit field of the size of the operand. The generated **getBinaryCodeForInstr** function takes care of mapping operand bits to their corresponding instruction bits.

The following example covers the *OpenRISC 1000* memory operand, but the same method can be applied to any compound operand type.

```
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;
}
```

To create the encoding for this operand, the individual components (the immediate and the register) can be obtained in the same way as was done in **getMachineOpValue** and then be shifted to the relevant operand bits.

For this example the first operand (a register) is taken and its encoding taken and then shifted 16 bits left. (The *OpenRISC 1000* memory operand is a register followed by a 16 bit immediate). The second operand (the immediate offset) is then encoded and combined with the register value to give the full encoding of the operand.

Note
The operand locations are hard coded in this example as in the *OpenRISC 1000* implementation, memory operands are always at known locations and no instruction may have more than one memory operand. In a more generic case, it is best to use the provided Op value instead of hard coding operand placement.

With the functions defined, instruction encoding should now operate correctly.

4.6. Testing

Tests written for instruction encoding replace any tests written for assembly parsing with those that check for valid encoding. Assembly parsing is implicitly tested as the correct encoding is emitted only if the instruction was correctly parsed.

Each instruction in turn should have its instruction manually encoded based on the instruction set documentation and then tested with llvm-mc and the -show-encoding directive. An example of a test is shown below.

```
# RUN: llvm-mc -arch=or1k -mattr=mul,div,ror -show-encoding %s | FileCheck %s

    l.add r1, r2, r3
# CHECK: # encoding: [0xe0,0x22,0x18,0x00]
```

The **-show-encoding** flag will cause **llvm-mc** to output the encoded version of the provided instruction, which is in the format shown above. Tests can then be executed in the usual way.

To aid in writing these tests, a python function is specified below that takes an instruction encoding and prints out the **CHECK** line for a 32-bit big-endian processor.

```
def a(asm):
  asm=hex(asm)[2:].zfill(8)
  print '# CHECK: # encoding: [0x'+asm[-8:-6]+',0x'+asm[-6:-4]+',0x'+ \
  asm[-4:-2]+',0x'+asm[-2:]+']'
```

This function can then be simply used in the same way as this example for the *OpenRISC 1000* **l.add** instruction when in a big endian mode.

```
a(0x38<<26|1<<21<2<<16|3<<11)
```

Chapter 5. Implementing Instruction Decoding

Whilst strictly not necessary to assemble code, the ability to disassemble instructions may be of use, so this is next implemented.

5.1. TableGen Requirements

The instruction decoder uses a decode function built by TableGen in order to match instructions to their operands. In order for this function to be built correctly, it is important that only one instruction is mapped to any given bit pattern. If this is not the case then LLVM will fail to compile.

If two instructions do collide, a message such as the following will appear in the build log, identifying which instructions conflicted.

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

Conflicts can be solved by providing context as to when it is suitable to decode instructions as one type or another. One way of doing this (if suitable) is to disable an instruction for this stage by either marking it is pseudo (via `isPsuedo = 1`) or as for use in code generation only (`isCodeGen = 1`).

5.2. Build System

As with the assembler parser, the instruction decoder is a sub-library of the main target library and goes in the `Disassembler` directory. This library consists of one source file, *arch*`Disassembler.cpp`. Files for the build system should be created here in the same way as was done for the assembly parser.

In addition, the `Disassembler` directory is added to both the `Makefile`, `CMakeLists.txt` and `LLVMBuild.txt` for the target library, with `has_disassembler = 1` being set in the parents library definition.

This library makes use of two extra TableGen files' disassembler tables and enhanced disassembly info. These are in the files *arch*`GenDisassemblerTables.inc` and *arch*`GenEDInfo.inc` respectively. As such these need adding to the `Makefile` and `CMakeLists.txt` as usual.

5.3. Integration

Integration of the disassembler occurs in the same way as the instruction assembler, with a static `create`*arch*`Disassembler` function being defined, which creates and returns a new *arch*`Disassembler`.

```
static MCDisassembler *createarchDisassembler(const Target &T,
                                              const MCSubtargetInfo &STI) {
  return new archDisassembler(STI, T.createMCRegInfo(""));
}
```

The **LLVMInitialize*arch*Disassembler** function is also defined which registers the disassembler with the rest of the system.

```
extern "C" void LLVMInitializearchDisassembler() {
  // Register the disassembler
  TargetRegistry::RegisterMCDisassembler(ThearchTarget,
                                         createarchDisassembler);
}
```

5.4. Disassembler

The ***arch*Disassembler** extends the **MCDisassembler** class and is centered around the **getInstruction** function. This function uses a memory region and decodes an instruction along with its operands, storing this information in a provided **MCInst**.

Note

Information about the **MCDisassembler** class can be found in *LLVM*'s documentation at llvm.org/docs/doxygen/html/classllvm_1_1MCDisassembler.html

A support function can be defined which helps read data from the memory object, formatting it in a way that the decoder can use. In the case of *OpenRISC 1000*, this function reads in 4 bytes (the size of an instruction) and formats it as an unsigned big endian 32-bit word.

Should a processor support both big and small endian instructions or variable length instructions, this function would instead be configured to read a variable number of bytes or to create a word which matches the target's endianness.

It should be noted that the function returns Fail if memory could not be read as this is a required step before disassembly.

```
static DecodeStatus readInstruction32(const MemoryObject &region,
                                      uint64_t address,
                                      uint64_t &size,
                                      uint32_t &insn) {
  uint8_t Bytes[4];

  // We want to read exactly 4 bytes of data.
  if (region.readBytes(address, 4, (uint8_t*)Bytes, NULL) == -1) {
    size = 0;
    return MCDisassembler::Fail;
  }

  // Encoded as big-endian 32-bit word in the stream.
  insn = (Bytes[0] << 24) |
         (Bytes[1] << 16) |
         (Bytes[2] <<  8) |
         (Bytes[3] <<  0);

  return MCDisassembler::Success;
}
```

The **getInstruction** should first call the above function to read memory ready for decoding. Should this function return success, then it is passed to the TableGen generated function **decode*arch*, Instruction*Size*** which does the decoding.

This will return **Success** if the instruction was successfully decoded, otherwise it will return **Fail**. The *Size* parameter provided to the function is set to the size of instruction that was successfully decoded.

In the case of *OpenRISC 1000*, only 32-bit instructions are supported, so a valid decode will always set this value to 4.

5.5. Variable Length Instructions

For targets that support variable length instructions, then there will be multiple decode tables to parse through and memory may have to be read multiple times before a successful decode occurs.

One approach to this which is be suitable in most cases is to start with the smallest instructions, reading the smallest amount of memory possible and then using the appropriate decode table.

If that fails to match an instruction, then more memory should be read, trying the next smallest instruction table, repeating until all tables have been tested. If no match has then been made an error code should be returned.

5.6. Decoding Register Classes

As with encoding register values when building the instruction encoder, when decoding instructions support for taking an encoded register value and creating an Operand object which appropriately describes it is required.

The names of these functions are defined by TableGen when building the decoding tables, but follow the form **Decode*RegClass*RegisterClass**. These functions are given the instruction to add an operand to, the encoded register value as well as the address in memory where the instruction can be found at.

In the case of the *OpenRISC 1000* architecture, the only register class which can be in an instruction is the a general register, meaning only one function needs defining, **DecodeGPRRegisterClass**. When called, this function checks that the given register number is within the valid range (0-31). If it is, a new register operand is added to the instruction.

```
DecodeStatus DecodeGPRRegisterClass(MCInst &Inst,
                                    unsigned RegNo,
                                    uint64_t Address,
                                    const void *Decoder) {

  if (RegNo > 31)
    return MCDisassembler::Fail;

  // The internal representation of the registers counts r0: 1, r1: 2, etc.
  Inst.addOperand(MCOperand::CreateReg(RegNo+1));
  return MCDisassembler::Success;
}
```

Setting the register value in this case is a case of incrementing the given number by 1 as the internal representation of registers is r0 is 1, r1 is 2, etc.

For targets with a more complex register definition, it is better to use a switch statement similar to the **getOR1KRegisterNumbering** function used in the instruction encoder, but checking the encoding and returning a register object if valid.

5.7. Decoding Custom Operands

For operands that required custom encoding methods in the instruction encoding, similar functions are also required to decode them.

Firstly in the operand's TableGen definition, the variable **DecoderMethod** needs to be defined as the name of the function in the Disassembler class that will handle them.

This function is then defined, taking the bits for each part of the operand, parsing them and then adding them to the operand in a similar way as was done in the decoding register class example.

For *OpenRISC 1000*, the only operand which requires a custom decoder is the memory operand, which combines a 5-bit register value with a signed 16-bit offset.

Firstly the bit values of the two operands are obtained from the given operand. These are in turn converted to **MCOperand**s and added to the instruction.

The register operand is handled in the same way as the register decoder, with the offset being sign extended to a 32-bit value before the immediate operand type is created.

```
static DecodeStatus DecodeMemoryValue(MCInst &Inst,
                                      unsigned Insn,
                                      uint64_t Address,
                                      const void *Decoder) {
  unsigned Register = (Insn >> 16) & 0b11111;
  Inst.addOperand(MCOperand::CreateReg(Register+1));
  unsigned Offset = (Insn & 0xffff);
  Inst.addOperand(MCOperand::CreateImm(SignExtend32<16>(Offset)));
  return MCDisassembler::Success;
}
```

5.8. Testing

Writing tests for the instruction decoder is simple once tests have been written for instruction encoding. Using **llvm-mc -disassemble** as the command for the test, tests can be generated by swapping the input and check lines from the encoding tests around, such that the encoding is the input and the printed instruction is the output.

For example, using the test from the previous examples.

```
    l.add r1, r2, r3
# CHECK: # encoding: [0x00,0x18,0x22,0xe0]
```

becomes

```
    0x00 0x18 0x22 0xe0
# CHECK: l.add r1, r2, r3
```

As an aside, both the instruction encoder and decoder can be tested at the same time outside of the test suite by piping them together, reformatting the output from the encoder before feeding it to the decoder. The following one-liner will test the encoding for a 32-bit instruction. Should the encoder and decoder be working, the outputted instruction (minus warnings), should be the same as the instruction specified at the start of the command.

```
echo "l.add r1, r20, r31" | llvm-mc -arch=or1k -show-encoding | \
sed 's/.*\(0x[0-9a-f]*\),\(0x[0-9a-f]*\),\(0x[0-9a-f]*\),\(0x[0-9a-f]*\).*'\
'/\1 \2 \3 \4/' | llvm-mc -arch=or1k -disassemble
```

Chapter 6. Implementing ELF Object Writing

Now that individual instructions can be encoded and decoded, the final stage is to enable the writing of ELF Objects which can then be linked by an external linker such as GNU ld.

This stage involves defining the relocations that are used with the architecture, LLVM fixups used as well as the algorithms used to manipulate these relocations and fixups.

6.1. Build System

Classes and source files for writing ELF files exist within the target's **`MCTargetDesc`** library, so setting up a new library is not needed. If the file has not been previously created, a **`AsmBackend`** needs adding to the CMakeLists.txt file in this library.

In addition the source file for the ELF object writer needs adding to the same build list. This file has the name ***`arch`*****`ELFObjectWriter.cpp`**.

6.2. Defining Fixups and Relocations

Within LLVM, fixups are used to represent information in instructions which is currently unknown. During instruction encoding, if some information is unknown (such as a memory location of an external symbol), it is encoded as if the value is equal to 0 and a fixup is emitted which contains information on how to rewrite the value when information is known.

The assembler goes through a stage of relaxation, applying fixups and modifying instruction values when they become known to the system. Once complete, any remaining fixups are converted to relocations and stored in the object file.

ELF Relocation types for a target are defined as an enum in the LLVM support header **`include/llvm/Support/ELF.h`** and are referred to as **`llvm::ELF::`*****`RELOCNAME`***.

Note

It is vital that these relocations have the same enumerated values as in the linker, otherwise the linker will not be able to understand and handle the object file correctly.

An example from the *OpenRISC 1000* implementation is given below.

```
enum {
  R_OR1K_NONE           =  0,
  R_OR1K_32             =  1,
...
  R_OR1K_RELATIVE       = 21
};
```

Fixups are defined in **`lib/Target/`*****`arch`*****`/MCTargetDesc/`** ***`arch`*****`FixupKinds.h`**, with (in the general case) one fixup being created for each relocation type defined above, with the exception of the no relocation required reloc.

These go into an enum called **`Fixups`**. The enum has its first item set to the value of **`FirstTargetFixupKind`** and ends with a marker for **`LastTargetFixupKind`**. The total number of fixups is then defined as **NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind**. An example of the fixups used in the *OpenRISC 1000* implementation is shown below.

```
enum Fixups {
  // Results in R_OR1K_32
  fixup_OR1K_32 = FirstTargetFixupKind,

  // Results in R_OR1K_16
  fixup_OR1K_16,

  // Results in R_OR1K_8
  fixup_OR1K_8,

  // Results in R_OR1K_LO_16_IN_INSN
  fixup_OR1K_LO16_INSN,
  ...
  // Marker
  LastTargetFixupKind,
  NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind
}
```

6.3. Assembly Backend

The assembly backend is responsible for manipulating fixup values, replacing them with values where information is available. The class for this backend inherits from the **MCAsmBackend**.

Note

Information about the **MCAsmBackend** class can be found in *LLVM*'s documentation at llvm.org/docs/doxygen/html/classllvm_1_1MCAsmBackend.html

The **getNumFixupKinds** function returns the number of fixups which the backend supports. This was defined as part of the **Fixups** enum, so this function simply returns this value.

```
unsigned getNumFixupKinds() const { return OR1K::NumTargetFixupKinds; }
```

The **applyFixup** function takes a fixup and provided data value and applies it to a given instruction.

To aid in this, a support function **adjustFixupValue** is created and called which manipulates the fixup's value before it is applied. This is done for example where a branch instruction does not store the exact location to branch to but that value without the first two bits. In this case, the value would be bitshifted by two before being applied.

With the fixup value adjusted appropriately, the instruction it is to be applied to is then reconstructed as a 64-bit unsigned integer. The fixup value is then shifted and masked into the correct location in the instruction before being applied. Once done, the now modified instruction is written back to the original data field.

The following example is from the *OpenRISC 1000* implementation, with the data being loaded and manipulated in a big endian fashion.

```
void OR1KAsmBackend::applyFixup(const MCFixup &Fixup, char *Data,
                                unsigned DataSize, uint64_t Value) const {
  MCFixupKind Kind = Fixup.getKind();
  Value = adjustFixupValue((unsigned)Kind, Value);

  if (!Value)
    return; // This value doesn't change the encoding

  // Where in the object and where the number of bytes that need
  // fixing up
  unsigned Offset = Fixup.getOffset();
  unsigned NumBytes = (getFixupKindInfo(Kind).TargetSize + 7) / 8;
  unsigned FullSize;

  switch((unsigned)Kind) {
    default:
      FullSize = 4;
      break;
  }

  // Grab current value, if any, from bits.
  uint64_t CurVal = 0;

  // Load instruction and apply value
  for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = (FullSize - 1 - i);
    CurVal |= (uint64_t)((uint8_t)Data[Offset + Idx]) << (i*8);
  }

  uint64_t Mask = ((uint64_t)(-1) >>
                  (64 - getFixupKindInfo(Kind).TargetSize));
  CurVal |= Value & Mask;

  // Write out the fixed up bytes back to the code/data bits.
  for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = (FullSize - 1 - i);
    Data[Offset + Idx] = (uint8_t)((CurVal >> (i*8)) & 0xff);
  }
}
```

Where there are spaces in an instruction stream that need filling with executable instructions, a series of **NOP**s should be inserted. This is done via the **writeNopData** function, which specifies the size of memory that needs filling. If valid instructions can be placed into the instruction stream they are then created and emitted via the provided **MCObjectWriter**.

In the case of the *OpenRISC 1000* implementation, the only **NOP** instruction is 32-bits long. Therefore if the space to fill is not a multiple of 4 bytes then the function returns false to indicate that it can't be filled. Otherwise for each set of four bytes, the encoding of a **NOP** is emitted via the **ObjectWriters** **Write32** function.

```
bool OR1KAsmBackend::writeNopData(uint64_t Count, MCObjectWriter *OW) const {
  if ((Count % 4) != 0)
    return false;

  for (uint64_t i = 0; i < Count; i += 4)
    OW->Write32(0x15000000);

  return true;
}
```

Another function which needs implementing is **`relaxInstruction`**, which takes an instruction and relaxes it to a longer instruction with the same effects.

For targets where no instruction ever needs relaxation (e.g. all instructions are the same size), this function simply returns. Otherwise the longer instruction is created, copying and formatting operands as appropriate.

Likewise the **`mayNeedRelaxation`** function returns true/false depending on if the instruction may need to be relaxed. In the case of the *OpenRISC 1000*, no instruction ever needs relaxing, therefore the function always returns false.

The **`fixupNeedsRelaxation`** returns whether an instruction needs to be relaxed based on the given fixup. As the case with the previous two functions, if no instruction needs to be relaxed this function will also always return false.

Finally, the **`getFixupKindInfo`** function needs overriding to provide information about target specific fixups, including their offset, size and flags. This function starts with a table containing details on each fixup.

If the fixup type provided is not target specific, the overridden function is called to get the result. Otherwise the entry is looked up in the table specified above and the relevant entry returned. If no entry exists in either table, then an error is raised.

Note

The entries in this table must be in the same order as specified in ***`arch`*****`FixupKinds.h`**.

```
const MCFixupKindInfo &OR1KAsmBackend::getFixupKindInfo(MCFixupKind Kind) const{
  const static MCFixupKindInfo Infos[OR1K::NumTargetFixupKinds] = {
    // This table *must* be in same the order of fixup_* kinds in
    // OR1KFixupKinds.h.
    //
    // name                    offset  bits  flags
    { "fixup_OR1K_32",          0,      32,   0 },
    { "fixup_OR1K_16",          0,      16,   0 },
    ... other fixups not shown ...
  };

  if (Kind < FirstTargetFixupKind)
     return MCAsmBackend::getFixupKindInfo(Kind);

  assert(unsigned(Kind - FirstTargetFixupKind) < getNumFixupKinds() &&
         "Invalid kind!");
  return Infos[Kind - FirstTargetFixupKind];
}
```

To enable and create the assembly backend, **create*arch*AsmBackend** is defined and returns a new ***arch*AsmBackend** object, based on a given target and triple.

```
MCAsmBackend *llvm::createarchAsmBackend(const Target &T, StringRef TT) {
  Triple TheTriple(TT);
  return new archAsmBackend(T, Triple(TT).getOS());
}
```

Finally this function is set up with the MC target registry, associating the assembly backend with the target.

In ***arch*MCTargetDesc.cpp**, the assembly backend is added in the same way as other components.

```
  // Register the ASM Backend
  TargetRegistry::RegisterMCAsmBackend(ThearchTarget,
                                       createarchAsmBackend);
```

6.4. ELF Object Writer

The ELF Object Writer class is based on the class **MCELFObjectTargetWriter** and handles the manipulations of fixups and conversion of fixups to relocs.

Note

Information about the **MCELFObjectTargetWriter** class can be found in *LLVM*'s documentation at llvm.org/docs/doxygen/html/classllvm_1_1MCELFObjectTargetWriter.html

The constructor for this class simply sets up the parent class, passing it the OS ABI version and the value used to identify the ELF file as belonging to that architecture.

Known machines can be found in an enum in **include/llvm/Support/ELF.h**. Should the machine architecture not be known, a new value should be used that does not conflict with any other architecture.

For the *OpenRISC 1000* architecture, this value is **ELF::EM_OPENRISC** and has the value 92. The constructor is therefore as follows.

```
OR1KELFObjectWriter::OR1KELFObjectWriter(uint8_t OSABI)
  : MCELFObjectTargetWriter(/*Is64Bit*/ false, OSABI, ELF::EM_OPENRISC,
                            /*HasRelocationAddend*/ true) {}
```

The **GetRelocType** function takes a fixup type, mapping it to a relocation type. As in many cases each fixup represents a single representation, the basic structure of this function is a switch statement and setting a variable (**Type**) to the relocation for the given fixup.

In addition to the custom fixups, there are also some built-in fixups (for 32bit absolute relocation etc.) that also need mapping (these are called **FK_Data_4**, etc. and can be found in **include/llvm/MC/MCFixup.h**.

```
unsigned OR1KELFObjectWriter::GetRelocType(const MCValue &Target,
                                           const MCFixup &Fixup,
                                           bool IsPCRel,
                                           bool IsRelocWithSymbol,
                                           int64_t Addend) const {
  unsigned Type;
  unsigned Kind = (unsigned)Fixup.getKind();
  switch (Kind) {
    default: llvm_unreachable("Invalid fixup kind!");
    case OR1K::fixup_OR1K_PCREL32:
    case FK_PCRel_4:
      Type = ELF::R_OR1K_32_PCREL;
      break;
    ... other fixups note shown ..
  return Type;
}
```

The object writer is enabled in a similar fashion to the other MC components, a function **create*arch*ELFObjectWriter** is used to create and return a new object writer. This function can be modified for example to provide support for different object writers depending on word size and endianness. A simple example from *OpenRISC 1000* is shown below.

```
MCObjectWriter *llvm::createOR1KELFObjectWriter(raw_ostream &OS,
                                                uint8_t OSABI) {
  MCELFObjectTargetWriter *MOTW = new OR1KELFObjectWriter(OSABI);
  return createELFObjectWriter(MOTW, OS, /*IsLittleEndian=*/ false);
}
```

This function is then used in the previously defined Assembly Backend to set up the object writer when the backend needs it.

```
MCObjectWriter *OR1KAsmBackend::createObjectWriter(raw_ostream &OS) const {
  return createOR1KELFObjectWriter(OS,
                                   MCELFObjectTargetWriter::getOSABI(OSType));
}
```

Should the architecture require support for multiple object file types, then the function would be modified so that a different object writer is created depending on the OS requested.

Finally support for streaming out object files is added in the ***arch*MCTargetDesc.cpp** file, by registering a **create*arch*MCStreamer** function with the target registry.

In the below example from *OpenRISC 1000*, this function checks for if the requested target format is MACH-O or COFF. Neither of these are supported by this implementation, so an error is raised if this is requested.

```
static MCStreamer *createOR1KMCStreamer(const Target &T, StringRef TT,
                                        MCContext &Ctx, MCAsmBackend &MAB,
                                        raw_ostream &_OS,
                                        MCCodeEmitter *_Emitter,
                                        bool RelaxAll,
                                        bool NoExecStack) {
  Triple TheTriple(TT);
  if (TheTriple.isOSDarwin()) {
    llvm_unreachable("OR1K does not support Darwin MACH-O format");
  }
  if (TheTriple.isOSWindows()) {
    llvm_unreachable("OR1K does not support Windows COFF format");
  }
  return createELFStreamer(Ctx, MAB, _OS, _Emitter, RelaxAll, NoExecStack);
}
```

```
  // Register the object streamer
  TargetRegistry::RegisterMCObjectStreamer(TheOR1KTarget,
                                           createOR1KMCStreamer);
```

6.5. Testing

Tests for object writing can be written in one of two forms. The first consists of using llvm assembly files which are then compiled with **llc** and the file examined by **elf-dump** or native instructions are interpreted by **llvm-mc** and again examined by **elf-dump**.

elf-dump is a command that is part of the LLVM test infrastructure and outputs the instructions and relocations in an object file for the purposes of testing that instruction encoding, sections, relocations and other ELF-based functionality work correctly.

To provide a simple demonstration of the tests that can be written, the following has been taken from the tests for the x86_64 Linux target. The test consists of two instructions which when encoded should have relocations associated with them.

The **CHECK** lines of the test look for the section where the encoded instructions will appear and then check that the section's location, flags, etc. are correct. Finally it checks the relocations on the two instructions and ensures that their types, values and addends are correct.

If needed, only the relocation sections can be checked. This is useful where section address may not be known but the relocation symbol identifiers and types are still known.

```
// RUN: llvm-mc -filetype=obj -triple x86_64-pc-linux-gnu %s -o - | \
// RUN: elf-dump --dump-section-data | FileCheck  %s

// Test that we produce the correct relocation.

  loope 0                 # R_X86_64_PC8
  jmp -256                # R_X86_64_PC32

// CHECK:      # Section 2
// CHECK-NEXT: (('sh_name', 0x00000001) # '.rela.text'
// CHECK-NEXT:  ('sh_type', 0x00000004)
// CHECK-NEXT:  ('sh_flags', 0x0000000000000000)
// CHECK-NEXT:  ('sh_addr', 0x0000000000000000)
// CHECK-NEXT:  ('sh_offset', 0x00000000000002e8)
// CHECK-NEXT:  ('sh_size', 0x0000000000000030)
// CHECK-NEXT:  ('sh_link', 0x00000006)
// CHECK-NEXT:  ('sh_info', 0x00000001)
// CHECK-NEXT:  ('sh_addralign', 0x0000000000000008)
// CHECK-NEXT:  ('sh_entsize', 0x0000000000000018)
// CHECK-NEXT:  ('_relocations', [
// CHECK-NEXT:   # Relocation 0
// CHECK-NEXT:   (('r_offset', 0x0000000000000001)
// CHECK-NEXT:    ('r_sym', 0x00000000)
// CHECK-NEXT:    ('r_type', 0x0000000f)
// CHECK-NEXT:    ('r_addend', 0x0000000000000000)
// CHECK-NEXT:   ),
// CHECK-NEXT:   # Relocation 1
// CHECK-NEXT:   (('r_offset', 0x0000000000000003)
// CHECK-NEXT:    ('r_sym', 0x00000000)
// CHECK-NEXT:    ('r_type', 0x00000002)
// CHECK-NEXT:    ('r_addend', 0x0000000000000000)
// CHECK-NEXT:   ),
// CHECK-NEXT:  ])
// CHECK-NEXT: ),
```

Glossary

Application Binary Interface (ABI)
The definition of how registers are used during function call and return for a particular architecture.

big endian
A multi-byte number representation, in which the most significant byte is placed first (i.e. at the lowest address) in memory.
See also: little endian

little endian
A multi-byte number representation, in which the least significant byte is placed first (i.e. at the lowest address) in memory.
See also: big endian

Machine Code (MC)
LLVM Library designed for handling low level target-specific instruction constructs for creating assemblers, disassemblers, etc.

TableGen
LLVM language and tool for generating classes to aid in instruction assembly, printing, etc. while keeping the need to define the instruction architecture only once.

References

[1] LLVM Doxygen Documentation LLVM API documentation, available at http://llvm.org/doxygen.

[2] *OpenRISC 1000* Architecture Manual Available from the OpenCores SVN repository at http://opencores.org/svnget,or1k?file=/trunk/docs/openrisc_arch.pdf

[3] *OpenRISC 1000* Relocation Information Available in source code form at https://github.com/skristiansson/or1k-src/blob/or1k/include/elf/or1k.h.