Writing an LLVM Backend

Introduction

This document describes techniques for writing compiler backends that convert the LLVM Intermediate Representation (IR) to code for a specified machine or other languages. Code intended for a specific machine can take the form of either assembly code or binary code (usable for a JIT compiler).

The backend of LLVM features a target-independent code generator that may create output for several types of target CPUs — including X86, PowerPC, ARM, and SPARC. The backend may also be used to generate code targeted at SPUs of the Cell processor or GPUs to support the execution of compute kernels.

The document focuses on existing examples found in subdirectories of `llvm/lib/Target` in a downloaded LLVM release. In particular, this document focuses on the example of creating a static compiler (one that emits text assembly) for a SPARC target, because SPARC has fairly standard characteristics, such as a RISC instruction set and straightforward calling conventions.

Audience

The audience for this document is anyone who needs to write an LLVM backend to generate code for a specific hardware or software target.

Prerequisite Reading

These essential documents must be read before reading this document:

- LLVM Language Reference Manual — a reference manual for the LLVM assembly language.
- The LLVM Target-Independent Code Generator — a guide to the components (classes and code generation algorithms) for translating the LLVM internal representation into machine code for a specified target. Pay particular attention to the descriptions of code generation stages: Instruction Selection, Scheduling and Formation, SSA-based Optimization, Register Allocation, Prolog/Epilog Code Insertion, Late Machine Code Optimizations, and Code Emission.
- TableGen Overview — a document that describes the TableGen (`tblgen`) application that manages domain-specific information to support LLVM code generation. TableGen processes input from a target description file (`.td` suffix) and generates C++ code that can be used for code generation.
- Writing an LLVM Pass — The assembly printer is a `FunctionPass`, as are several `SelectionDAG` processing steps.

To follow the SPARC examples in this document, have a copy of The SPARC Architecture Manual, Version 8 for reference. For details about the ARM instruction set, refer to the ARM Architecture Reference Manual. For more about the GNU Assembler format (`GAS`), see Using As, especially for the assembly printer. “Using As” contains a list of target machine dependent features.

Basic Steps

To write a compiler backend for LLVM that converts the LLVM IR to code for a specified target (machine or other language), follow these steps:

- Create a subclass of the `TargetMachine` class that describes characteristics of your target machine. Copy existing examples of specific `TargetMachine` class and header files; for example, start with `SparcTargetMachine.cpp` and `SparcTargetMachine.h`, but change the file names for your target. Similarly, change code that references “`Sparc`” to reference your target.
- Describe the register set of the target. Use TableGen to generate code for register definition, register aliases, and register classes from a target-specific `RegisterInfo.td` input file. You should also write additional code for a subclass of the `TargetRegisterInfo` class that represents the class register file data used for register allocation and also describes the interactions between registers.
- Describe the instruction set of the target. Use TableGen to generate code for target-specific instructions from target-specific versions of `TargetInstrFormats.td` and `TargetInstrInfo.td`. You should write additional code for a subclass of the `TargetInstrInfo` class to represent machine instructions supported by the target machine.
- Describe the selection and conversion of the LLVM IR from a Directed Acyclic Graph (DAG) representation of instructions to native target-specific instructions. Use TableGen to generate code that matches patterns and selects instructions based on additional information in a target-specific version of `TargetInstrInfo.td`. Write code for `XXXISelDAGToDAG.cpp`, where XXX identifies the specific target, to perform pattern matching and DAG-to-DAG instruction selection. Also write code in `XXXISelLowering.cpp` to replace or remove operations and data types that are not supported natively in a SelectionDAG.
- Write code for an assembly printer that converts LLVM IR to a GAS format for your target machine. You should add assembly strings to the instructions defined in your target-specific version of `TargetInstrInfo.td`. You should also write code for a subclass of `AsmPrinter` that performs the LLVM-to-assembly conversion and a trivial subclass of `TargetAsmInfo`.
- Optionally, add support for subtargets (i.e., variants with different capabilities). You should also write code for a subclass of the `TargetSubtarget` class, which allows you to use the `-mcpu=` and `-mattr=` command-line options.
- Optionally, add JIT support and create a machine code emitter (subclass of `TargetJITInfo`) that is used to emit binary code directly into memory.

In the `.cpp` and `.h` files, initially stub up these methods and then implement them later. Initially, you may not know which private members that the class will need and which components will need to be subclassed.

Preliminaries

To actually create your compiler backend, you need to create and modify a few files. The absolute minimum is discussed here. But to actually use the LLVM target-independent code generator, you must perform the steps described in the LLVM Target-Independent Code Generator document.

First, you should create a subdirectory under `lib/Target` to hold all the files related to your target. If your target is called “Dummy”, create the directory `lib/Target/Dummy`.

In this new directory, create a `CMakeLists.txt`. It is easiest to copy a `CMakeLists.txt` of another target and modify it. It should at least contain the `LLVM_TARGET_DEFINITIONS` variable. The library can be named `LLVMDummy` (for example, see the MIPS target). Alternatively, you can split the library into `LLVMDummyCodeGen` and `LLVMDummyAsmPrinter`, the latter of which should be implemented in a subdirectory below `lib/Target/Dummy` (for example, see the PowerPC target).

Note that these two naming schemes are hardcoded into `llvm-config`. Using any other naming scheme will confuse `llvm-config` and produce a lot of (seemingly unrelated) linker errors when linking `llc`.

To make your target actually do something, you need to implement a subclass of `TargetMachine`. This implementation should typically be in the file `lib/Target/DummyTargetMachine.cpp`, but any file in the `lib/Target` directory will be built and should work. To use LLVM’s target independent code generator, you should do what all current machine backends do: create a subclass of `LLVMTargetMachine`. (To create a target from scratch, create a subclass of `TargetMachine`.)

To get LLVM to actually build and link your target, you need to run `cmake` with `-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=Dummy`. This will build your target without needing to add it to the list of all the targets.

Once your target is stable, you can add it to the `LLVM_ALL_TARGETS` variable located in the main `CMakeLists.txt`.

Target Machine

`LLVMTargetMachine` is designed as a base class for targets implemented with the LLVM target-independent code generator. The `LLVMTargetMachine` class should be specialized by a concrete target class that implements the various virtual methods. `LLVMTargetMachine` is defined as a subclass of `TargetMachine` in `include/llvm/Target/TargetMachine.h`. The `TargetMachine` class implementation (`TargetMachine.cpp`) also processes numerous command-line options.

To create a concrete target-specific subclass of `LLVMTargetMachine`, start by copying an existing `TargetMachine` class and header. You should name the files that you create to reflect your specific target. For instance, for the SPARC target, name the files `SparcTargetMachine.h` and `SparcTargetMachine.cpp`.

For a target machine XXX, the implementation of `XXXTargetMachine` must have access methods to obtain objects that represent target components. These methods are named `get*Info`, and are intended to obtain the instruction set (`getInstrInfo`), register set (`getRegisterInfo`), stack frame layout (`getFrameInfo`), and similar information. `XXXTargetMachine` must also implement the `getDataLayout` method to access an object with target-specific data characteristics, such as data type size and alignment requirements.

For instance, for the SPARC target, the header file `SparcTargetMachine.h` declares prototypes for several `get*Info` and `getDataLayout` methods that simply return a class member.

```
namespace llvm {

class Module;

class SparcTargetMachine : public LLVMTargetMachine {
  const DataLayout DataLayout;       // Calculates type size & alignment
  SparcSubtarget Subtarget;
  SparcInstrInfo InstrInfo;
```

```
  TargetFrameInfo FrameInfo;

protected:
  virtual const TargetAsmInfo *createTargetAsmInfo() const;

public:
  SparcTargetMachine(const Module &M, const std::string &FS);

  virtual const SparcInstrInfo *getInstrInfo() const {return &InstrInfo; }
  virtual const TargetFrameInfo *getFrameInfo() const {return &FrameInfo; }
  virtual const TargetSubtarget *getSubtargetImpl() const{return &Subtarget; }
  virtual const TargetRegisterInfo *getRegisterInfo() const {
    return &InstrInfo.getRegisterInfo();
  }
  virtual const DataLayout *getDataLayout() const { return &DataLayout; }
  static unsigned getModuleMatchQuality(const Module &M);

  // Pass Pipeline Configuration
  virtual bool addInstSelector(PassManagerBase &PM, bool Fast);
  virtual bool addPreEmitPass(PassManagerBase &PM, bool Fast);
};

} // end namespace llvm
```

- `getInstrInfo()`
- `getRegisterInfo()`
- `getFrameInfo()`
- `getDataLayout()`
- `getSubtargetImpl()`

For some targets, you also need to support the following methods:

- `getTargetLowering()`
- `getJITInfo()`

Some architectures, such as GPUs, do not support jumping to an arbitrary program location and implement branching using masked execution and loop using special instructions around the loop body. In order to avoid CFG modifications that introduce irreducible control flow not handled by such hardware, a target must call *`setRequiresStructuredCFG(true)`* when being initialized.

In addition, the `XXXTargetMachine` constructor should specify a `TargetDescription` string that determines the data layout for the target machine, including characteristics such as pointer size, alignment, and endianness. For example, the constructor for `SparcTargetMachine` contains the following:

```
SparcTargetMachine::SparcTargetMachine(const Module &M, const std::string &FS)
  : DataLayout("E-p:32:32-f128:128:128"),
    Subtarget(M, FS), InstrInfo(Subtarget),
    FrameInfo(TargetFrameInfo::StackGrowsDown, 8, 0) {
}
```

Hyphens separate portions of the `TargetDescription` string.

- An upper-case “E” in the string indicates a big-endian target data model. A lower-case “e” indicates little-endian.
- “`p:`” is followed by pointer information: size, ABI alignment, and preferred alignment. If only two figures follow “`p:`”, then the first value is pointer size, and the second value is both ABI and preferred alignment.
- Then a letter for numeric type alignment: “`i`”, “`f`”, “`v`”, or “`a`” (corresponding to integer, floating point, vector, or aggregate). “`i`”, “`v`”, or “`a`” are followed by ABI alignment and preferred alignment. “`f`” is followed by three values: the first indicates the size of a long double, then ABI alignment, and then ABI preferred alignment.

翻译

Target Registration

You must also register your target with the `TargetRegistry`, which is what other LLVM tools use to be able to lookup and use your target at runtime. The `TargetRegistry` can be used directly, but for most targets there are helper templates which should take care of the work for you.

All targets should declare a global `Target` object which is used to represent the target during registration. Then, in the target's `TargetInfo` library, the target should define that object and use the `RegisterTarget` template to register the target. For example, the Sparc registration code looks like this:

```
Target llvm::getTheSparcTarget();

extern "C" void LLVMInitializeSparcTargetInfo() {
  RegisterTarget<Triple::sparc, /*HasJIT=*/false>
    X(getTheSparcTarget(), "sparc", "Sparc");
}
```

This allows the `TargetRegistry` to look up the target by name or by target triple. In addition, most targets will also register additional features which are available in separate libraries. These registration steps are separate, because some clients may wish to only link in some parts of the target — the JIT code generator does not require the use of the assembler printer, for example. Here is an example of registering the Sparc assembly printer:

```
extern "C" void LLVMInitializeSparcAsmPrinter() {
  RegisterAsmPrinter<SparcAsmPrinter> X(getTheSparcTarget());
}
```

For more information, see "llvm/Target/TargetRegistry.h".

Register Set and Register Classes

You should describe a concrete target-specific class that represents the register file of a target machine. This class is called `XXXRegisterInfo` (where `XXX` identifies the target) and represents the class register file data that is used for register allocation. It also describes the interactions between registers.

You also need to define register classes to categorize related registers. A register class should be added for groups of registers that are all treated the same way for some instruction. Typical examples are register classes for integer, floating-point, or vector registers. A register allocator allows an instruction to use any register in a specified register class to perform the instruction in a similar manner. Register classes allocate virtual registers to instructions from these sets, and register classes let the target-independent register allocator automatically choose the actual registers.

Much of the code for registers, including register definition, register aliases, and register classes, is generated by TableGen from `XXXRegisterInfo.td` input files and placed in `XXXGenRegisterInfo.h.inc` and `XXXGenRegisterInfo.inc` output files. Some of the code in the implementation of `XXXRegisterInfo` requires hand-coding.

Defining a Register

The `XXXRegisterInfo.td` file typically starts with register definitions for a target machine. The `Register` class (specified in `Target.td`) is used to define an object for each register. The specified string `n` becomes the `Name` of the register. The basic `Register` object does not have any subregisters and does not specify any aliases.

```
class Register<string n> {
  string Namespace = "";
  string AsmName = n;
  string Name = n;
  int SpillSize = 0;
  int SpillAlignment = 0;
  list<Register> Aliases = [];
  list<Register> SubRegs = [];
  list<int> DwarfNumbers = [];
}
```

For example, in the X86RegisterInfo.td file, there are register definitions that utilize the Register class, such as:

```
def AL : Register<"AL">, DwarfRegNum<[0, 0, 0]>;
```

This defines the register AL and assigns it values (with DwarfRegNum) that are used by gcc, gdb, or a debug information writer to identify a register. For register AL, DwarfRegNum takes an array of 3 values representing 3 different modes: the first element is for X86-64, the second for exception handling (EH) on X86-32, and the third is generic. -1 is a special Dwarf number that indicates the gcc number is undefined, and -2 indicates the register number is invalid for this mode.

From the previously described line in the X86RegisterInfo.td file, TableGen generates this code in the X86GenRegisterInfo.inc file:

```
static const unsigned GR8[] = { X86::AL, ... };

const unsigned AL_AliasSet[] = { X86::AX, X86::EAX, X86::RAX, 0 };

const TargetRegisterDesc RegisterDescriptors[] = {
  ...
{ "AL", "AL", AL_AliasSet, Empty_SubRegsSet, Empty_SubRegsSet, AL_SuperRegsSet }, ...
```

From the register info file, TableGen generates a TargetRegisterDesc object for each register. TargetRegisterDesc is defined in include/llvm/Target/TargetRegisterInfo.h with the following fields:

```
struct TargetRegisterDesc {
  const char     *AsmName;      // Assembly language name for the register
  const char     *Name;         // Printable name for the reg (for debugging)
  const unsigned *AliasSet;     // Register Alias Set
  const unsigned *SubRegs;      // Sub-register set
  const unsigned *ImmSubRegs;   // Immediate sub-register set
  const unsigned *SuperRegs;    // Super-register set
};
```

TableGen uses the entire target description file (.td) to determine text names for the register (in the AsmName and Name fields of TargetRegisterDesc) and the relationships of other registers to the defined register (in the other TargetRegisterDesc fields). In this example, other definitions establish the registers “AX”, “EAX”, and “RAX” as aliases for one another, so TableGen generates a null-terminated array (AL_AliasSet) for this register alias set.

The Register class is commonly used as a base class for more complex classes. In Target.td, the Register class is the base for the RegisterWithSubRegs class that is used to define registers that need to specify subregisters in the SubRegs list, as shown here:

```
class RegisterWithSubRegs<string n, list<Register> subregs> : Register<n> {
  let SubRegs = subregs;
}
```

In SparcRegisterInfo.td, additional register classes are defined for SPARC: a Register subclass, SparcReg, and further subclasses: Ri, Rf, and Rd. SPARC registers are identified by 5-bit ID numbers, which is a feature common to these subclasses. Note the use of “let” expressions to override values that are initially defined in a superclass (such as SubRegs field in the Rd class).

```
class SparcReg<string n> : Register<n> {
  field bits<5> Num;
  let Namespace = "SP";
}
// Ri - 32-bit integer registers
class Ri<bits<5> num, string n> :
SparcReg<n> {
  let Num = num;
}
// Rf - 32-bit floating-point registers
class Rf<bits<5> num, string n> :
```

```
SparcReg<n> {
  let Num = num;
}
// Rd - Slots in the FP register file for 64-bit floating-point values.
class Rd<bits<5> num, string n, list<Register> subregs> : SparcReg<n> {
  let Num = num;
  let SubRegs = subregs;
}
```

In the `SparcRegisterInfo.td` file, there are register definitions that utilize these subclasses of `Register`, such as:

```
def G0 : Ri< 0, "G0">, DwarfRegNum<[0]>;
def G1 : Ri< 1, "G1">, DwarfRegNum<[1]>;
...
def F0 : Rf< 0, "F0">, DwarfRegNum<[32]>;
def F1 : Rf< 1, "F1">, DwarfRegNum<[33]>;
...
def D0 : Rd< 0, "F0", [F0, F1]>, DwarfRegNum<[32]>;
def D1 : Rd< 2, "F2", [F2, F3]>, DwarfRegNum<[34]>;
```

The last two registers shown above (`D0` and `D1`) are double-precision floating-point registers that are aliases for pairs of single-precision floating-point sub-registers. In addition to aliases, the sub-register and super-register relationships of the defined register are in fields of a register's `TargetRegisterDesc`.

Defining a Register Class

The `RegisterClass` class (specified in `Target.td`) is used to define an object that represents a group of related registers and also defines the default allocation order of the registers. A target description file `XXXRegisterInfo.td` that uses `Target.td` can construct register classes using the following class:

```
class RegisterClass<string namespace,
list<ValueType> regTypes, int alignment, dag regList> {
  string Namespace = namespace;
  list<ValueType> RegTypes = regTypes;
  int Size = 0;  // spill size, in bits; zero lets tblgen pick the size
  int Alignment = alignment;

  // CopyCost is the cost of copying a value between two registers
  // default value 1 means a single instruction
  // A negative value means copying is extremely expensive or impossible
  int CopyCost = 1;
  dag MemberList = regList;

  // for register classes that are subregisters of this class
  list<RegisterClass> SubRegClassList = [];

  code MethodProtos = [{}];  // to insert arbitrary code
  code MethodBodies = [{}];
}
```

To define a `RegisterClass`, use the following 4 arguments:

- The first argument of the definition is the name of the namespace.
- The second argument is a list of `ValueType` register type values that are defined in `include/llvm/CodeGen/ValueTypes.td`. Defined values include integer types (such as `i16`, `i32`, and `i1` for Boolean), floating-point types (`f32`, `f64`), and vector types (for example, `v8i16` for an `8 x i16` vector). All registers in a `RegisterClass` must have the same `ValueType`, but some registers may store vector data in different configurations. For example a register that can process a 128-bit vector may be able to handle 16 8-bit integer elements, 8 16-bit integers, 4 32-bit integers, and so on.
- The third argument of the `RegisterClass` definition specifies the alignment required of the registers when they are stored or loaded to memory.
- The final argument, `regList`, specifies which registers are in this class. If an alternative allocation order method is not specified, then `regList` also defines the order of allocation used by the register allocator.

Besides simply listing registers with `(add R0, R1, ...)`, more advanced set operators are available. See `include/llvm/Target/Target.td` for more information.

In `SparcRegisterInfo.td`, three `RegisterClass` objects are defined: `FPRegs`, `DFPRegs`, and `IntRegs`. For all three register classes, the first argument defines the namespace with the string "`SP`". `FPRegs` defines a group of 32 single-precision floating-point registers (`F0` to `F31`); `DFPRegs` defines a group of 16 double-precision registers (`D0-D15`).

```
// F0, F1, F2, ..., F31
def FPRegs : RegisterClass<"SP", [f32], 32, (sequence "F%u", 0, 31)>;

def DFPRegs : RegisterClass<"SP", [f64], 64,
                            (add D0, D1, D2, D3, D4, D5, D6, D7, D8,
                                 D9, D10, D11, D12, D13, D14, D15)>;

def IntRegs : RegisterClass<"SP", [i32], 32,
    (add L0, L1, L2, L3, L4, L5, L6, L7,
         I0, I1, I2, I3, I4, I5,
         O0, O1, O2, O3, O4, O5, O7,
         G1,
         // Non-allocatable regs:
         G2, G3, G4,
         O6,        // stack ptr
         I6,        // frame ptr
         I7,        // return address
         G0,        // constant zero
         G5, G6, G7 // reserved for kernel
    )>;
```

Using `SparcRegisterInfo.td` with TableGen generates several output files that are intended for inclusion in other source code that you write. `SparcRegisterInfo.td` generates `SparcGenRegisterInfo.h.inc`, which should be included in the header file for the implementation of the SPARC register implementation that you write (`SparcRegisterInfo.h`). In `SparcGenRegisterInfo.h.inc` a new structure is defined called `SparcGenRegisterInfo` that uses `TargetRegisterInfo` as its base. It also specifies types, based upon the defined register classes: `DFPRegsClass`, `FPRegsClass`, and `IntRegsClass`.

`SparcRegisterInfo.td` also generates `SparcGenRegisterInfo.inc`, which is included at the bottom of `SparcRegisterInfo.cpp`, the SPARC register implementation. The code below shows only the generated integer registers and associated register classes. The order of registers in `IntRegs` reflects the order in the definition of `IntRegs` in the target description file.

```
// IntRegs Register Class...
static const unsigned IntRegs[] = {
  SP::L0, SP::L1, SP::L2, SP::L3, SP::L4, SP::L5,
  SP::L6, SP::L7, SP::I0, SP::I1, SP::I2, SP::I3,
  SP::I4, SP::I5, SP::O0, SP::O1, SP::O2, SP::O3,
  SP::O4, SP::O5, SP::O7, SP::G1, SP::G2, SP::G3,
  SP::G4, SP::O6, SP::I6, SP::I7, SP::G0, SP::G5,
  SP::G6, SP::G7,
};

// IntRegsVTs Register Class Value Types...
static const MVT::ValueType IntRegsVTs[] = {
  MVT::i32, MVT::Other
};

namespace SP {   // Register class instances
  DFPRegsClass    DFPRegsRegClass;
  FPRegsClass     FPRegsRegClass;
  IntRegsClass    IntRegsRegClass;
...
  // IntRegs Sub-register Classes...
  static const TargetRegisterClass* const IntRegsSubRegClasses [] = {
    NULL
  };
...
  // IntRegs Super-register Classes..
```

```
  static const TargetRegisterClass* const IntRegsSuperRegClasses [] = {
    NULL
  };
...
  // IntRegs Register Class sub-classes...
  static const TargetRegisterClass* const IntRegsSubclasses [] = {
    NULL
  };
...
  // IntRegs Register Class super-classes...
  static const TargetRegisterClass* const IntRegsSuperclasses [] = {
    NULL
  };

  IntRegsClass::IntRegsClass() : TargetRegisterClass(IntRegsRegClassID,
    IntRegsVTs, IntRegsSubclasses, IntRegsSuperclasses, IntRegsSubRegClasses,
    IntRegsSuperRegClasses, 4, 4, 1, IntRegs, IntRegs + 32) {}
}
```

The register allocators will avoid using reserved registers, and callee saved registers are not used until all the volatile registers have been used. That is usually good enough, but in some cases it may be necessary to provide custom allocation orders.

Implement a subclass of **`TargetRegisterInfo`**

The final step is to hand code portions of `XXXRegisterInfo`, which implements the interface described in `TargetRegisterInfo.h` (see The TargetRegisterInfo class). These functions return `0`, `NULL`, or `false`, unless overridden. Here is a list of functions that are overridden for the SPARC implementation in `SparcRegisterInfo.cpp`:

- `getCalleeSavedRegs` — Returns a list of callee-saved registers in the order of the desired callee-save stack frame offset.
- `getReservedRegs` — Returns a bitset indexed by physical register numbers, indicating if a particular register is unavailable.
- `hasFP` — Return a Boolean indicating if a function should have a dedicated frame pointer register.
- `eliminateCallFramePseudoInstr` — If call frame setup or destroy pseudo instructions are used, this can be called to eliminate them.
- `eliminateFrameIndex` — Eliminate abstract frame indices from instructions that may use them.
- `emitPrologue` — Insert prologue code into the function.
- `emitEpilogue` — Insert epilogue code into the function.

Instruction Set

During the early stages of code generation, the LLVM IR code is converted to a `SelectionDAG` with nodes that are instances of the `SDNode` class containing target instructions. An `SDNode` has an opcode, operands, type requirements, and operation properties. For example, is an operation commutative, does an operation load from memory. The various operation node types are described in the `include/llvm/CodeGen/SelectionDAGNodes.h` file (values of the `NodeType` enum in the `ISD` namespace).

TableGen uses the following target description (`.td`) input files to generate much of the code for instruction definition:

- `Target.td` — Where the `Instruction`, `Operand`, `InstrInfo`, and other fundamental classes are defined.
- `TargetSelectionDAG.td` — Used by `SelectionDAG` instruction selection generators, contains `SDTC*` classes (selection DAG type constraint), definitions of `SelectionDAG` nodes (such as `imm`, `cond`, `bb`, `add`, `fadd`, `sub`), and pattern support (`Pattern`, `Pat`, `PatFrag`, `PatLeaf`, `ComplexPattern`.
- `XXXInstrFormats.td` — Patterns for definitions of target-specific instructions.
- `XXXInstrInfo.td` — Target-specific definitions of instruction templates, condition codes, and instructions of an instruction set. For architecture modifications, a different file name may be used. For example, for Pentium with SSE instruction, this file is `X86InstrSSE.td`, and for Pentium with MMX, this file is `X86InstrMMX.td`.

There is also a target-specific `XXX.td` file, where XXX is the name of the target. The `XXX.td` file includes the other `.td` input files, but its contents are only directly important for subtargets.

You should describe a concrete target-specific class `XXXInstrInfo` that represents machine instructions supported by a target machine. `XXXInstrInfo` contains an array of `XXXInstrDescriptor` objects, each of which describes one instruction. An instruction descriptor defines:

- Opcode mnemonic
- Number of operands
- List of implicit register definitions and uses
- Target-independent properties (such as memory access, is commutable)
- Target-specific flags

The Instruction class (defined in `Target.td`) is mostly used as a base for more complex instruction classes.

```
class Instruction {
  string Namespace = "";
  dag OutOperandList;    // A dag containing the MI def operand list.
  dag InOperandList;     // A dag containing the MI use operand list.
  string AsmString = ""; // The .s format to print the instruction with.
  list<dag> Pattern;     // Set to the DAG pattern for this instruction.
  list<Register> Uses = [];
  list<Register> Defs = [];
  list<Predicate> Predicates = [];  // predicates turned into isel match code
  ... remainder not shown for space ...
}
```

A `SelectionDAG` node (`SDNode`) should contain an object representing a target-specific instruction that is defined in `XXXInstrInfo.td`. The instruction objects should represent instructions from the architecture manual of the target machine (such as the SPARC Architecture Manual for the SPARC target).

A single instruction from the architecture manual is often modeled as multiple target instructions, depending upon its operands. For example, a manual might describe an add instruction that takes a register or an immediate operand. An LLVM target could model this with two instructions named `ADDri` and `ADDrr`.

You should define a class for each instruction category and define each opcode as a subclass of the category with appropriate parameters such as the fixed binary encoding of opcodes and extended opcodes. You should map the register bits to the bits of the instruction in which they are encoded (for the JIT). Also you should specify how the instruction should be printed when the automatic assembly printer is used.

As is described in the SPARC Architecture Manual, Version 8, there are three major 32-bit formats for instructions. Format 1 is only for the `CALL` instruction. Format 2 is for branch on condition codes and `SETHI` (set high bits of a register) instructions. Format 3 is for other instructions.

Each of these formats has corresponding classes in `SparcInstrFormat.td`. `InstSP` is a base class for other instruction classes. Additional base classes are specified for more precise formats: for example in `SparcInstrFormat.td`, `F2_1` is for `SETHI`, and `F2_2` is for branches. There are three other base classes: `F3_1` for register/register operations, `F3_2` for register/immediate operations, and `F3_3` for floating-point operations. `SparcInstrInfo.td` also adds the base class `Pseudo` for synthetic SPARC instructions.

`SparcInstrInfo.td` largely consists of operand and instruction definitions for the SPARC target. In `SparcInstrInfo.td`, the following target description file entry, `LDrr`, defines the Load Integer instruction for a Word (the LD SPARC opcode) from a memory address to a register. The first parameter, the value 3 (11_2), is the operation value for this category of operation. The second parameter (000000_2) is the specific operation value for LD/Load Word. The third parameter is the output destination, which is a register operand and defined in the `Register` target description file (`IntRegs`).

```
def LDrr : F3_1 <3, 0b000000, (outs IntRegs:$rd), (ins (MEMrr $rs1, $rs2):$addr),
                 "ld [$addr], $dst",
                 [(set i32:$dst, (load ADDRrr:$addr))]>;
```

The fourth parameter is the input source, which uses the address operand `MEMrr` that is defined earlier in `SparcInstrInfo.td`:

```
def MEMrr : Operand<i32> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops IntRegs, IntRegs);
}
```

The fifth parameter is a string that is used by the assembly printer and can be left as an empty string until the assembly printer interface is implemented. The sixth and final parameter is the pattern used to match the instruction during the SelectionDAG Select Phase described in The LLVM Target-Independent Code Generator. This parameter is detailed in the next section, Instruction Selector.

Instruction class definitions are not overloaded for different operand types, so separate versions of instructions are needed for register, memory, or immediate value operands. For example, to perform a Load Integer instruction for a Word from an immediate operand to a register, the following instruction class is defined:

```
def LDri : F3_2 <3, 0b000000, (outs IntRegs:$rd), (ins (MEMri $rs1, $simm13):$addr),
                 "ld [$addr], $dst",
                 [(set i32:$rd, (load ADDRri:$addr))]>;
```

Writing these definitions for so many similar instructions can involve a lot of cut and paste. In `.td` files, the `multiclass` directive enables the creation of templates to define several instruction classes at once (using the `defm` directive). For example in `SparcInstrInfo.td`, the `multiclass` pattern `F3_12` is defined to create 2 instruction classes each time `F3_12` is invoked:

```
multiclass F3_12 <string OpcStr, bits<6> Op3Val, SDNode OpNode> {
  def rr  : F3_1 <2, Op3Val,
                 (outs IntRegs:$rd), (ins IntRegs:$rs1, IntRegs:$rs1),
                 !strconcat(OpcStr, " $rs1, $rs2, $rd"),
                 [(set i32:$rd, (OpNode i32:$rs1, i32:$rs2))]>;
  def ri  : F3_2 <2, Op3Val,
                 (outs IntRegs:$rd), (ins IntRegs:$rs1, i32imm:$simm13),
                 !strconcat(OpcStr, " $rs1, $simm13, $rd"),
                 [(set i32:$rd, (OpNode i32:$rs1, simm13:$simm13))]>;
}
```

So when the `defm` directive is used for the `XOR` and `ADD` instructions, as seen below, it creates four instruction objects: `XORrr`, `XORri`, `ADDrr`, and `ADDri`.

```
defm XOR   : F3_12<"xor", 0b000011, xor>;
defm ADD   : F3_12<"add", 0b000000, add>;
```

`SparcInstrInfo.td` also includes definitions for condition codes that are referenced by branch instructions. The following definitions in `SparcInstrInfo.td` indicate the bit location of the SPARC condition code. For example, the 10th bit represents the “greater than” condition for integers, and the 22nd bit represents the “greater than” condition for floats.

```
def ICC_NE  : ICC_VAL< 9>;  // Not Equal
def ICC_E   : ICC_VAL< 1>;  // Equal
def ICC_G   : ICC_VAL<10>;  // Greater
...
def FCC_U   : FCC_VAL<23>;  // Unordered
def FCC_G   : FCC_VAL<22>;  // Greater
def FCC_UG  : FCC_VAL<21>;  // Unordered or Greater
...
```

(Note that `Sparc.h` also defines enums that correspond to the same SPARC condition codes. Care must be taken to ensure the values in `Sparc.h` correspond to the values in `SparcInstrInfo.td`. I.e., `SPCC::ICC_NE = 9`, `SPCC::FCC_U = 23` and so on.)

Instruction Operand Mapping

The code generator backend maps instruction operands to fields in the instruction. Whenever a bit in the instruction encoding `Inst` is assigned to field without a concrete value, an operand from the `outs` or `ins` list is

expected to have a matching name. This operand then populates that undefined field. For example, the Sparc target defines the `XNORrr` instruction as a `F3_1` format instruction having three operands: the output `$rd`, and the inputs `$rs1`, and `$rs2`.

```
def XNORrr  : F3_1<2, 0b000111,
                   (outs IntRegs:$rd), (ins IntRegs:$rs1, IntRegs:$rs2),
                   "xnor $rs1, $rs2, $rd",
                   [(set i32:$rd, (not (xor i32:$rs1, i32:$rs2)))]>;
```

The instruction templates in `SparcInstrFormats.td` show the base class for `F3_1` is `InstSP`.

```
class InstSP<dag outs, dag ins, string asmstr, list<dag> pattern> : Instruction {
  field bits<32> Inst;
  let Namespace = "SP";
  bits<2> op;
  let Inst{31-30} = op;
  dag OutOperandList = outs;
  dag InOperandList = ins;
  let AsmString   = asmstr;
  let Pattern = pattern;
}
```

`InstSP` defines the `op` field, and uses it to define bits 30 and 31 of the instruction, but does not assign a value to it.

```
class F3<dag outs, dag ins, string asmstr, list<dag> pattern>
    : InstSP<outs, ins, asmstr, pattern> {
  bits<5> rd;
  bits<6> op3;
  bits<5> rs1;
  let op{1} = 1;   // Op = 2 or 3
  let Inst{29-25} = rd;
  let Inst{24-19} = op3;
  let Inst{18-14} = rs1;
}
```

`F3` defines the `rd`, `op3`, and `rs1` fields, and uses them in the instruction, and again does not assign values.

```
class F3_1<bits<2> opVal, bits<6> op3val, dag outs, dag ins,
           string asmstr, list<dag> pattern> : F3<outs, ins, asmstr, pattern> {
  bits<8> asi = 0; // asi not currently used
  bits<5> rs2;
  let op         = opVal;
  let op3        = op3val;
  let Inst{13}   = 0;     // i field = 0
  let Inst{12-5} = asi;   // address space identifier
  let Inst{4-0}  = rs2;
}
```

`F3_1` assigns a value to `op` and `op3` fields, and defines the `rs2` field. Therefore, a `F3_1` format instruction will require a definition for `rd`, `rs1`, and `rs2` in order to fully specify the instruction encoding.

The `XNORrr` instruction then provides those three operands in its OutOperandList and InOperandList, which bind to the corresponding fields, and thus complete the instruction encoding.

For some instructions, a single operand may contain sub-operands. As shown earlier, the instruction `LDrr` uses an input operand of type `MEMrr`. This operand type contains two register sub-operands, defined by the `MIOperandInfo` value to be `(ops IntRegs, IntRegs)`.

```
def LDrr : F3_1 <3, 0b000000, (outs IntRegs:$rd), (ins (MEMrr $rs1, $rs2):$addr),
                 "ld [$addr], $dst",
                 [(set i32:$dst, (load ADDRrr:$addr))]>;
```

As this instruction is also the `F3_1` format, it will expect operands named `rd`, `rs1`, and `rs2` as well. In order to allow this, a complex operand can optionally give names to each of its sub-operands. In this example `MEMrr`'s

first sub-operand is named `$rs1`, the second `$rs2`, and the operand as a whole is also given the name `$addr`.

When a particular instruction doesn't use all the operands that the instruction format defines, a constant value may instead be bound to one or all. For example, the RDASR instruction only takes a single register operand, so we assign a constant zero to `rs2`:

```
let rs2 = 0 in
  def RDASR : F3_1<2, 0b101000,
                   (outs IntRegs:$rd), (ins ASRRegs:$rs1),
                   "rd $rs1, $rd", []>;
```

Instruction Operand Name Mapping

TableGen will also generate a function called getNamedOperandIdx() which can be used to look up an operand's index in a MachineInstr based on its TableGen name. Setting the UseNamedOperandTable bit in an instruction's TableGen definition will add all of its operands to an enumeration in the llvm::XXX:OpName namespace and also add an entry for it into the OperandMap table, which can be queried using getNamedOperandIdx()

```
int DstIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::dst); // => 0
int BIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::b);     // => 1
int CIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::c);     // => 2
int DIndex = SP::getNamedOperandIdx(SP::XNORrr, SP::OpName::d);     // => -1

...
```

The entries in the OpName enum are taken verbatim from the TableGen definitions, so operands with lowercase names will have lower case entries in the enum.

To include the getNamedOperandIdx() function in your backend, you will need to define a few preprocessor macros in XXXInstrInfo.cpp and XXXInstrInfo.h. For example:

XXXInstrInfo.cpp:

```
#define GET_INSTRINFO_NAMED_OPS // For getNamedOperandIdx() function
#include "XXXGenInstrInfo.inc"
```

XXXInstrInfo.h:

```
#define GET_INSTRINFO_OPERAND_ENUM // For OpName enum
#include "XXXGenInstrInfo.inc"

namespace XXX {
  int16_t getNamedOperandIdx(uint16_t Opcode, uint16_t NamedIndex);
} // End namespace XXX
```

Instruction Operand Types

TableGen will also generate an enumeration consisting of all named Operand types defined in the backend, in the llvm::XXX::OpTypes namespace. Some common immediate Operand types (for instance i8, i32, i64, f32, f64) are defined for all targets in `include/llvm/Target/Target.td`, and are available in each Target's OpTypes enum. Also, only named Operand types appear in the enumeration: anonymous types are ignored. For example, the X86 backend defines `brtarget` and `brtarget8`, both instances of the TableGen `Operand` class, which represent branch target operands:

```
def brtarget : Operand<OtherVT>;
def brtarget8 : Operand<OtherVT>;
```

This results in:

```
namespace X86 {
namespace OpTypes {
enum OperandType {
  ...
```

```
  brtarget,
  brtarget8,
  ...
  i32imm,
  i64imm,
  ...
  OPERAND_TYPE_LIST_END
} // End namespace OpTypes
} // End namespace X86
```

In typical TableGen fashion, to use the enum, you will need to define a preprocessor macro:

```
#define GET_INSTRINFO_OPERAND_TYPES_ENUM // For OpTypes enum
#include "XXXGenInstrInfo.inc"
```

Instruction Scheduling

Instruction itineraries can be queried using MCDesc::getSchedClass(). The value can be named by an enumeration in llvm::XXX::Sched namespace generated by TableGen in XXXGenInstrInfo.inc. The name of the schedule classes are the same as provided in XXXSchedule.td plus a default NoItinerary class.

The schedule models are generated by TableGen by the SubtargetEmitter, using the `CodeGenSchedModels` class. This is distinct from the itinerary method of specifying machine resource use. The tool `utils/schedcover.py` can be used to determine which instructions have been covered by the schedule model description and which haven't. The first step is to use the instructions below to create an output file. Then run `schedcover.py` on the output file:

```
$ <src>/utils/schedcover.py <build>/lib/Target/AArch64/tblGenSubtarget.with
instruction, default, CortexA53Model, CortexA57Model, CycloneModel, ExynosM3Model, Falk
ABSv16i8, WriteV, , , CyWriteV3, M3WriteNMISC1, FalkorWr_2VXVY_2cyc, KryoWrite_2cyc_XY_
ABSv1i64, WriteV, , , CyWriteV3, M3WriteNMISC1, FalkorWr_1VXVY_2cyc, KryoWrite_2cyc_XY_
...
```

To capture the debug output from generating a schedule model, change to the appropriate target directory and use the following command: command with the `subtarget-emitter` debug option:

```
$ <build>/bin/llvm-tblgen -debug-only=subtarget-emitter -gen-subtarget \
  -I <src>/lib/Target/<target> -I <src>/include \
  -I <src>/lib/Target <src>/lib/Target/<target>/<target>.td \
  -o <build>/lib/Target/<target>/<target>GenSubtargetInfo.inc.tmp \
  > tblGenSubtarget.dbg 2>&1
```

Where `<build>` is the build directory, `src` is the source directory, and `<target>` is the name of the target. To double check that the above command is what is needed, one can capture the exact TableGen command from a build by using:

```
$ VERBOSE=1 make ...
```

and search for `llvm-tblgen` commands in the output.

Instruction Relation Mapping

This TableGen feature is used to relate instructions with each other. It is particularly useful when you have multiple instruction formats and need to switch between them after instruction selection. This entire feature is driven by relation models which can be defined in `XXXInstrInfo.td` files according to the target-specific instruction set. Relation models are defined using `InstrMapping` class as a base. TableGen parses all the models and generates instruction relation maps using the specified information. Relation maps are emitted as tables in the `XXXGenInstrInfo.inc` file along with the functions to query them. For the detailed information on how to use this feature, please refer to How To Use Instruction Mappings.

Implement a subclass of **`TargetInstrInfo`**

The final step is to hand code portions of `XXXInstrInfo`, which implements the interface described in `TargetInstrInfo.h` (see The TargetInstrInfo class). These functions return `0` or a Boolean or they assert, unless overridden. Here's a list of functions that are overridden for the SPARC implementation in `SparcInstrInfo.cpp`:

- `isLoadFromStackSlot` — If the specified machine instruction is a direct load from a stack slot, return the register number of the destination and the `FrameIndex` of the stack slot.
- `isStoreToStackSlot` — If the specified machine instruction is a direct store to a stack slot, return the register number of the destination and the `FrameIndex` of the stack slot.
- `copyPhysReg` — Copy values between a pair of physical registers.
- `storeRegToStackSlot` — Store a register value to a stack slot.
- `loadRegFromStackSlot` — Load a register value from a stack slot.
- `storeRegToAddr` — Store a register value to memory.
- `loadRegFromAddr` — Load a register value from memory.
- `foldMemoryOperand` — Attempt to combine instructions of any load or store instruction for the specified operand(s).

Branch Folding and If Conversion

Performance can be improved by combining instructions or by eliminating instructions that are never reached. The `analyzeBranch` method in `XXXInstrInfo` may be implemented to examine conditional instructions and remove unnecessary instructions. `analyzeBranch` looks at the end of a machine basic block (MBB) for opportunities for improvement, such as branch folding and if conversion. The `BranchFolder` and `IfConverter` machine function passes (see the source files `BranchFolding.cpp` and `IfConversion.cpp` in the `lib/CodeGen` directory) call `analyzeBranch` to improve the control flow graph that represents the instructions.

Several implementations of `analyzeBranch` (for ARM, Alpha, and X86) can be examined as models for your own `analyzeBranch` implementation. Since SPARC does not implement a useful `analyzeBranch`, the ARM target implementation is shown below.

`analyzeBranch` returns a Boolean value and takes four parameters:

- `MachineBasicBlock &MBB` — The incoming block to be examined.
- `MachineBasicBlock *&TBB` — A destination block that is returned. For a conditional branch that evaluates to true, `TBB` is the destination.
- `MachineBasicBlock *&FBB` — For a conditional branch that evaluates to false, `FBB` is returned as the destination.
- `std::vector<MachineOperand> &Cond` — List of operands to evaluate a condition for a conditional branch.

In the simplest case, if a block ends without a branch, then it falls through to the successor block. No destination blocks are specified for either `TBB` or `FBB`, so both parameters return `NULL`. The start of the `analyzeBranch` (see code below for the ARM target) shows the function parameters and the code for the simplest case.

```
bool ARMInstrInfo::analyzeBranch(MachineBasicBlock &MBB,
                                 MachineBasicBlock *&TBB,
                                 MachineBasicBlock *&FBB,
                                 std::vector<MachineOperand> &Cond) const
{
  MachineBasicBlock::iterator I = MBB.end();
  if (I == MBB.begin() || !isUnpredicatedTerminator(--I))
    return false;
```

If a block ends with a single unconditional branch instruction, then `analyzeBranch` (shown below) should return the destination of that branch in the `TBB` parameter.

```
if (LastOpc == ARM::B || LastOpc == ARM::tB) {
  TBB = LastInst->getOperand(0).getMBB();
  return false;
}
```

If a block ends with two unconditional branches, then the second branch is never reached. In that situation, as shown below, remove the last branch instruction and return the penultimate branch in the TBB parameter.

```
if ((SecondLastOpc == ARM::B || SecondLastOpc == ARM::tB) &&
    (LastOpc == ARM::B || LastOpc == ARM::tB)) {
  TBB = SecondLastInst->getOperand(0).getMBB();
  I = LastInst;
  I->eraseFromParent();
  return false;
}
```

A block may end with a single conditional branch instruction that falls through to successor block if the condition evaluates to false. In that case, `analyzeBranch` (shown below) should return the destination of that conditional branch in the TBB parameter and a list of operands in the `Cond` parameter to evaluate the condition.

```
if (LastOpc == ARM::Bcc || LastOpc == ARM::tBcc) {
  // Block ends with fall-through condbranch.
  TBB = LastInst->getOperand(0).getMBB();
  Cond.push_back(LastInst->getOperand(1));
  Cond.push_back(LastInst->getOperand(2));
  return false;
}
```

If a block ends with both a conditional branch and an ensuing unconditional branch, then `analyzeBranch` (shown below) should return the conditional branch destination (assuming it corresponds to a conditional evaluation of "`true`") in the TBB parameter and the unconditional branch destination in the FBB (corresponding to a conditional evaluation of "`false`"). A list of operands to evaluate the condition should be returned in the `Cond` parameter.

```
unsigned SecondLastOpc = SecondLastInst->getOpcode();

if ((SecondLastOpc == ARM::Bcc && LastOpc == ARM::B) ||
    (SecondLastOpc == ARM::tBcc && LastOpc == ARM::tB)) {
  TBB =  SecondLastInst->getOperand(0).getMBB();
  Cond.push_back(SecondLastInst->getOperand(1));
  Cond.push_back(SecondLastInst->getOperand(2));
  FBB = LastInst->getOperand(0).getMBB();
  return false;
}
```

For the last two cases (ending with a single conditional branch or ending with one conditional and one unconditional branch), the operands returned in the `Cond` parameter can be passed to methods of other instructions to create new branches or perform other operations. An implementation of `analyzeBranch` requires the helper methods `removeBranch` and `insertBranch` to manage subsequent operations.

`analyzeBranch` should return false indicating success in most circumstances. `analyzeBranch` should only return true when the method is stumped about what to do, for example, if a block has three terminating branches. `analyzeBranch` may return true if it encounters a terminator it cannot handle, such as an indirect branch.

Instruction Selector

LLVM uses a `SelectionDAG` to represent LLVM IR instructions, and nodes of the `SelectionDAG` ideally represent native target instructions. During code generation, instruction selection passes are performed to convert non-native DAG instructions into native target-specific instructions. The pass described in `XXXISelDAGToDAG.cpp` is used to match patterns and perform DAG-to-DAG instruction selection. Optionally, a pass may be defined (in `XXXBranchSelector.cpp`) to perform similar DAG-to-DAG operations for branch instructions. Later, the code in `XXXISelLowering.cpp` replaces or removes operations and data types not supported natively (legalizes) in a `SelectionDAG`.

TableGen generates code for instruction selection using the following target description input files:

- `XXXInstrInfo.td` — Contains definitions of instructions in a target-specific instruction set, generates `XXXGenDAGISel.inc`, which is included in `XXXISelDAGToDAG.cpp`.

- `XXXCallingConv.td` — Contains the calling and return value conventions for the target architecture, and it generates `XXXGenCallingConv.inc`, which is included in `XXXISelLowering.cpp`.

The implementation of an instruction selection pass must include a header that declares the `FunctionPass` class or a subclass of `FunctionPass`. In `XXXTargetMachine.cpp`, a Pass Manager (PM) should add each instruction selection pass into the queue of passes to run.

The LLVM static compiler (`llc`) is an excellent tool for visualizing the contents of DAGs. To display the `SelectionDAG` before or after specific processing phases, use the command line options for `llc`, described at SelectionDAG Instruction Selection Process.

To describe instruction selector behavior, you should add patterns for lowering LLVM code into a `SelectionDAG` as the last parameter of the instruction definitions in `XXXInstrInfo.td`. For example, in `SparcInstrInfo.td`, this entry defines a register store operation, and the last parameter describes a pattern with the store DAG operator.

```
def STrr  : F3_1< 3, 0b000100, (outs), (ins MEMrr:$addr, IntRegs:$src),
                "st $src, [$addr]", [(store i32:$src, ADDRrr:$addr)]>;
```

`ADDRrr` is a memory mode that is also defined in `SparcInstrInfo.td`:

```
def ADDRrr : ComplexPattern<i32, 2, "SelectADDRrr", [], []>;
```

The definition of `ADDRrr` refers to `SelectADDRrr`, which is a function defined in an implementation of the Instructor Selector (such as `SparcISelDAGToDAG.cpp`).

In `lib/Target/TargetSelectionDAG.td`, the DAG operator for store is defined below:

```
def store : PatFrag<(ops node:$val, node:$ptr),
                    (unindexedstore node:$val, node:$ptr)> {
  let IsStore = true;
  let IsTruncStore = false;
}
```

`XXXInstrInfo.td` also generates (in `XXXGenDAGISel.inc`) the `SelectCode` method that is used to call the appropriate processing method for an instruction. In this example, `SelectCode` calls `Select_ISD_STORE` for the `ISD::STORE` opcode.

```
SDNode *SelectCode(SDValue N) {
  ...
  MVT::ValueType NVT = N.getNode()->getValueType(0);
  switch (N.getOpcode()) {
  case ISD::STORE: {
    switch (NVT) {
    default:
      return Select_ISD_STORE(N);
      break;
    }
    break;
  }
  ...
```

The pattern for `STrr` is matched, so elsewhere in `XXXGenDAGISel.inc`, code for `STrr` is created for `Select_ISD_STORE`. The `Emit_22` method is also generated in `XXXGenDAGISel.inc` to complete the processing of this instruction.

```
SDNode *Select_ISD_STORE(const SDValue &N) {
  SDValue Chain = N.getOperand(0);
  if (Predicate_store(N.getNode())) {
    SDValue N1 = N.getOperand(1);
    SDValue N2 = N.getOperand(2);
    SDValue CPTmp0;
    SDValue CPTmp1;

    // Pattern: (st:void i32:i32:$src,
```

```
    //           ADDRrr:i32:$addr)<<P:Predicate_store>>
    // Emits: (STrr:void ADDRrr:i32:$addr, IntRegs:i32:$src)
    // Pattern complexity = 13  cost = 1  size = 0
    if (SelectADDRrr(N, N2, CPTmp0, CPTmp1) &&
        N1.getNode()->getValueType(0) == MVT::i32 &&
        N2.getNode()->getValueType(0) == MVT::i32) {
      return Emit_22(N, SP::STrr, CPTmp0, CPTmp1);
    }
...
```

The SelectionDAG Legalize Phase

The Legalize phase converts a DAG to use types and operations that are natively supported by the target. For natively unsupported types and operations, you need to add code to the target-specific `XXXTargetLowering` implementation to convert unsupported types and operations to supported ones.

In the constructor for the `XXXTargetLowering` class, first use the `addRegisterClass` method to specify which types are supported and which register classes are associated with them. The code for the register classes are generated by TableGen from `XXXRegisterInfo.td` and placed in `XXXGenRegisterInfo.h.inc`. For example, the implementation of the constructor for the SparcTargetLowering class (in `SparcISelLowering.cpp`) starts with the following code:

```
addRegisterClass(MVT::i32, SP::IntRegsRegisterClass);
addRegisterClass(MVT::f32, SP::FPRegsRegisterClass);
addRegisterClass(MVT::f64, SP::DFPRegsRegisterClass);
```

You should examine the node types in the `ISD` namespace (`include/llvm/CodeGen/SelectionDAGNodes.h`) and determine which operations the target natively supports. For operations that do **not** have native support, add a callback to the constructor for the `XXXTargetLowering` class, so the instruction selection process knows what to do. The `TargetLowering` class callback methods (declared in `llvm/Target/TargetLowering.h`) are:

- `setOperationAction` — General operation.
- `setLoadExtAction` — Load with extension.
- `setTruncStoreAction` — Truncating store.
- `setIndexedLoadAction` — Indexed load.
- `setIndexedStoreAction` — Indexed store.
- `setConvertAction` — Type conversion.
- `setCondCodeAction` — Support for a given condition code.

Note: on older releases, `setLoadXAction` is used instead of `setLoadExtAction`. Also, on older releases, `setCondCodeAction` may not be supported. Examine your release to see what methods are specifically supported.

These callbacks are used to determine that an operation does or does not work with a specified type (or types). And in all cases, the third parameter is a `LegalAction` type enum value: `Promote`, `Expand`, `Custom`, or `Legal`. `SparcISelLowering.cpp` contains examples of all four `LegalAction` values.

Promote

For an operation without native support for a given type, the specified type may be promoted to a larger type that is supported. For example, SPARC does not support a sign-extending load for Boolean values (`i1` type), so in `SparcISelLowering.cpp` the third parameter below, `Promote`, changes `i1` type values to a large type before loading.

```
setLoadExtAction(ISD::SEXTLOAD, MVT::i1, Promote);
```

Expand

For a type without native support, a value may need to be broken down further, rather than promoted. For an operation without native support, a combination of other operations may be used to similar effect. In SPARC, the

floating-point sine and cosine trig operations are supported by expansion to other operations, as indicated by the third parameter, `Expand`, to `setOperationAction`:

```
setOperationAction(ISD::FSIN, MVT::f32, Expand);
setOperationAction(ISD::FCOS, MVT::f32, Expand);
```

Custom

For some operations, simple type promotion or operation expansion may be insufficient. In some cases, a special intrinsic function must be implemented.

For example, a constant value may require special treatment, or an operation may require spilling and restoring registers in the stack and working with register allocators.

As seen in `SparcISelLowering.cpp` code below, to perform a type conversion from a floating point value to a signed integer, first the `setOperationAction` should be called with `Custom` as the third parameter:

```
setOperationAction(ISD::FP_TO_SINT, MVT::i32, Custom);
```

In the `LowerOperation` method, for each `Custom` operation, a case statement should be added to indicate what function to call. In the following code, an `FP_TO_SINT` opcode will call the `LowerFP_TO_SINT` method:

```
SDValue SparcTargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) {
  switch (Op.getOpcode()) {
  case ISD::FP_TO_SINT: return LowerFP_TO_SINT(Op, DAG);
  ...
  }
}
```

Finally, the `LowerFP_TO_SINT` method is implemented, using an FP register to convert the floating-point value to an integer.

```
static SDValue LowerFP_TO_SINT(SDValue Op, SelectionDAG &DAG) {
  assert(Op.getValueType() == MVT::i32);
  Op = DAG.getNode(SPISD::FTOI, MVT::f32, Op.getOperand(0));
  return DAG.getNode(ISD::BITCAST, MVT::i32, Op);
}
```

Legal

The `Legal` `LegalizeAction` enum value simply indicates that an operation **is** natively supported. `Legal` represents the default condition, so it is rarely used. In `SparcISelLowering.cpp`, the action for `CTPOP` (an operation to count the bits set in an integer) is natively supported only for SPARC v9. The following code enables the `Expand` conversion technique for non-v9 SPARC implementations.

```
setOperationAction(ISD::CTPOP, MVT::i32, Expand);
...
if (TM.getSubtarget<SparcSubtarget>().isV9())
  setOperationAction(ISD::CTPOP, MVT::i32, Legal);
```

Calling Conventions

To support target-specific calling conventions, `XXXGenCallingConv.td` uses interfaces (such as `CCIfType` and `CCAssignToReg`) that are defined in `lib/Target/TargetCallingConv.td`. TableGen can take the target descriptor file `XXXGenCallingConv.td` and generate the header file `XXXGenCallingConv.inc`, which is typically included in `XXXISelLowering.cpp`. You can use the interfaces in `TargetCallingConv.td` to specify:

- The order of parameter allocation.
- Where parameters and return values are placed (that is, on the stack or in registers).
- Which registers may be used.
- Whether the caller or callee unwinds the stack.

The following example demonstrates the use of the `CCIfType` and `CCAssignToReg` interfaces. If the `CCIfType` predicate is true (that is, if the current argument is of type `f32` or `f64`), then the action is performed. In this case, the `CCAssignToReg` action assigns the argument value to the first available register: either `R0` or `R1`.

```
CCIfType<[f32,f64], CCAssignToReg<[R0, R1]>>
```

`SparcCallingConv.td` contains definitions for a target-specific return-value calling convention (`RetCC_Sparc32`) and a basic 32-bit C calling convention (`CC_Sparc32`). The definition of `RetCC_Sparc32` (shown below) indicates which registers are used for specified scalar return types. A single-precision float is returned to register `F0`, and a double-precision float goes to register `D0`. A 32-bit integer is returned in register `I0` or `I1`.

```
def RetCC_Sparc32 : CallingConv<[
  CCIfType<[i32], CCAssignToReg<[I0, I1]>>,
  CCIfType<[f32], CCAssignToReg<[F0]>>,
  CCIfType<[f64], CCAssignToReg<[D0]>>
]>;
```

The definition of `CC_Sparc32` in `SparcCallingConv.td` introduces `CCAssignToStack`, which assigns the value to a stack slot with the specified size and alignment. In the example below, the first parameter, 4, indicates the size of the slot, and the second parameter, also 4, indicates the stack alignment along 4-byte units. (Special cases: if size is zero, then the ABI size is used; if alignment is zero, then the ABI alignment is used.)

```
def CC_Sparc32 : CallingConv<[
  // All arguments get passed in integer registers if there is space.
  CCIfType<[i32, f32, f64], CCAssignToReg<[I0, I1, I2, I3, I4, I5]>>,
  CCAssignToStack<4, 4>
]>;
```

`CCDelegateTo` is another commonly used interface, which tries to find a specified sub-calling convention, and, if a match is found, it is invoked. In the following example (in `X86CallingConv.td`), the definition of `RetCC_X86_32_C` ends with `CCDelegateTo`. After the current value is assigned to the register `ST0` or `ST1`, the `RetCC_X86Common` is invoked.

```
def RetCC_X86_32_C : CallingConv<[
  CCIfType<[f32], CCAssignToReg<[ST0, ST1]>>,
  CCIfType<[f64], CCAssignToReg<[ST0, ST1]>>,
  CCDelegateTo<RetCC_X86Common>
]>;
```

`CCIfCC` is an interface that attempts to match the given name to the current calling convention. If the name identifies the current calling convention, then a specified action is invoked. In the following example (in `X86CallingConv.td`), if the `Fast` calling convention is in use, then `RetCC_X86_32_Fast` is invoked. If the `SSECall` calling convention is in use, then `RetCC_X86_32_SSE` is invoked.

```
def RetCC_X86_32 : CallingConv<[
  CCIfCC<"CallingConv::Fast", CCDelegateTo<RetCC_X86_32_Fast>>,
  CCIfCC<"CallingConv::X86_SSECall", CCDelegateTo<RetCC_X86_32_SSE>>,
  CCDelegateTo<RetCC_X86_32_C>
]>;
```

`CCAssignToRegAndStack` is the same as `CCAssignToReg`, but also allocates a stack slot, when some register is used. Basically, it works like: `CCIf<CCAssignToReg<regList>, CCAssignToStack<size, align>>`.

```
class CCAssignToRegAndStack<list<Register> regList, int size, int align>
    : CCAssignToReg<regList> {
  int Size = size;
  int Align = align;
}
```

Other calling convention interfaces include:

- `CCIf <predicate, action>` — If the predicate matches, apply the action.
- `CCIfInReg <action>` — If the argument is marked with the "`inreg`" attribute, then apply the action.
- `CCIfNest <action>` — If the argument is marked with the "`nest`" attribute, then apply the action.
- `CCIfNotVarArg <action>` — If the current function does not take a variable number of arguments, apply the action.
- `CCAssignToRegWithShadow <registerList, shadowList>` — similar to `CCAssignToReg`, but with a shadow list of registers.
- `CCPassByVal <size, align>` — Assign value to a stack slot with the minimum specified size and alignment.
- `CCPromoteToType <type>` — Promote the current value to the specified type.
- `CallingConv <[actions]>` — Define each calling convention that is supported.

Assembly Printer

During the code emission stage, the code generator may utilize an LLVM pass to produce assembly output. To do this, you want to implement the code for a printer that converts LLVM IR to a GAS-format assembly language for your target machine, using the following steps:

- Define all the assembly strings for your target, adding them to the instructions defined in the `XXXInstrInfo.td` file. (See Instruction Set.) TableGen will produce an output file (`XXXGenAsmWriter.inc`) with an implementation of the `printInstruction` method for the `XXXAsmPrinter` class.
- Write `XXXTargetAsmInfo.h`, which contains the bare-bones declaration of the `XXXTargetAsmInfo` class (a subclass of `TargetAsmInfo`).
- Write `XXXTargetAsmInfo.cpp`, which contains target-specific values for `TargetAsmInfo` properties and sometimes new implementations for methods.
- Write `XXXAsmPrinter.cpp`, which implements the `AsmPrinter` class that performs the LLVM-to-assembly conversion.

The code in `XXXTargetAsmInfo.h` is usually a trivial declaration of the `XXXTargetAsmInfo` class for use in `XXXTargetAsmInfo.cpp`. Similarly, `XXXTargetAsmInfo.cpp` usually has a few declarations of `XXXTargetAsmInfo` replacement values that override the default values in `TargetAsmInfo.cpp`. For example in `SparcTargetAsmInfo.cpp`:

```
SparcTargetAsmInfo::SparcTargetAsmInfo(const SparcTargetMachine &TM) {
  Data16bitsDirective = "\t.half\t";
  Data32bitsDirective = "\t.word\t";
  Data64bitsDirective = 0;  // .xword is only supported by V9.
  ZeroDirective = "\t.skip\t";
  CommentString = "!";
  ConstantPoolSection = "\t.section \".rodata\",#alloc\n";
}
```

The X86 assembly printer implementation (`X86TargetAsmInfo`) is an example where the target specific `TargetAsmInfo` class uses an overridden methods: `ExpandInlineAsm`.

A target-specific implementation of `AsmPrinter` is written in `XXXAsmPrinter.cpp`, which implements the `AsmPrinter` class that converts the LLVM to printable assembly. The implementation must include the following headers that have declarations for the `AsmPrinter` and `MachineFunctionPass` classes. The `MachineFunctionPass` is a subclass of `FunctionPass`.

```
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
```

As a `FunctionPass`, `AsmPrinter` first calls `doInitialization` to set up the `AsmPrinter`. In `SparcAsmPrinter`, a `Mangler` object is instantiated to process variable names.

In `XXXAsmPrinter.cpp`, the `runOnMachineFunction` method (declared in `MachineFunctionPass`) must be implemented for `XXXAsmPrinter`. In `MachineFunctionPass`, the `runOnFunction` method invokes

`runOnMachineFunction`. Target-specific implementations of `runOnMachineFunction` differ, but generally do the following to process each machine function:

- Call `SetupMachineFunction` to perform initialization.
- Call `EmitConstantPool` to print out (to the output stream) constants which have been spilled to memory.
- Call `EmitJumpTableInfo` to print out jump tables used by the current function.
- Print out the label for the current function.
- Print out the code for the function, including basic block labels and the assembly for the instruction (using `printInstruction`)

The `XXXAsmPrinter` implementation must also include the code generated by TableGen that is output in the `XXXGenAsmWriter.inc` file. The code in `XXXGenAsmWriter.inc` contains an implementation of the `printInstruction` method that may call these methods:

- `printOperand`
- `printMemOperand`
- `printCCOperand` (for conditional statements)
- `printDataDirective`
- `printDeclare`
- `printImplicitDef`
- `printInlineAsm`

The implementations of `printDeclare`, `printImplicitDef`, `printInlineAsm`, and `printLabel` in `AsmPrinter.cpp` are generally adequate for printing assembly and do not need to be overridden.

The `printOperand` method is implemented with a long `switch`/`case` statement for the type of operand: register, immediate, basic block, external symbol, global address, constant pool index, or jump table index. For an instruction with a memory address operand, the `printMemOperand` method should be implemented to generate the proper output. Similarly, `printCCOperand` should be used to print a conditional operand.

`doFinalization` should be overridden in `XXXAsmPrinter`, and it should be called to shut down the assembly printer. During `doFinalization`, global variables and constants are printed to output.

Subtarget Support

Subtarget support is used to inform the code generation process of instruction set variations for a given chip set. For example, the LLVM SPARC implementation provided covers three major versions of the SPARC microprocessor architecture: Version 8 (V8, which is a 32-bit architecture), Version 9 (V9, a 64-bit architecture), and the UltraSPARC architecture. V8 has 16 double-precision floating-point registers that are also usable as either 32 single-precision or 8 quad-precision registers. V8 is also purely big-endian. V9 has 32 double-precision floating-point registers that are also usable as 16 quad-precision registers, but cannot be used as single-precision registers. The UltraSPARC architecture combines V9 with UltraSPARC Visual Instruction Set extensions.

If subtarget support is needed, you should implement a target-specific `XXXSubtarget` class for your architecture. This class should process the command-line options `-mcpu=` and `-mattr=`.

TableGen uses definitions in the `Target.td` and `Sparc.td` files to generate code in `SparcGenSubtarget.inc`. In `Target.td`, shown below, the `SubtargetFeature` interface is defined. The first 4 string parameters of the `SubtargetFeature` interface are a feature name, an attribute set by the feature, the value of the attribute, and a description of the feature. (The fifth parameter is a list of features whose presence is implied, and its default value is an empty array.)

```
class SubtargetFeature<string n, string a, string v, string d,
                       list<SubtargetFeature> i = []> {
  string Name = n;
  string Attribute = a;
  string Value = v;
  string Desc = d;
  list<SubtargetFeature> Implies = i;
}
```

In the `Sparc.td` file, the `SubtargetFeature` is used to define the following features.

```
def FeatureV9 : SubtargetFeature<"v9", "IsV9", "true",
                    "Enable SPARC-V9 instructions">;
def FeatureV8Deprecated : SubtargetFeature<"deprecated-v8",
                    "V8DeprecatedInsts", "true",
                    "Enable deprecated V8 instructions in V9 mode">;
def FeatureVIS : SubtargetFeature<"vis", "IsVIS", "true",
                    "Enable UltraSPARC Visual Instruction Set extensions">;
```

Elsewhere in `Sparc.td`, the `Proc` class is defined and then is used to define particular SPARC processor subtypes that may have the previously described features.

```
class Proc<string Name, list<SubtargetFeature> Features>
  : Processor<Name, NoItineraries, Features>;

def : Proc<"generic",         []>;
def : Proc<"v8",              []>;
def : Proc<"supersparc",      []>;
def : Proc<"sparclite",       []>;
def : Proc<"f934",            []>;
def : Proc<"hypersparc",      []>;
def : Proc<"sparclite86x",    []>;
def : Proc<"sparclet",        []>;
def : Proc<"tsc701",          []>;
def : Proc<"v9",              [FeatureV9]>;
def : Proc<"ultrasparc",      [FeatureV9, FeatureV8Deprecated]>;
def : Proc<"ultrasparc3",     [FeatureV9, FeatureV8Deprecated]>;
def : Proc<"ultrasparc3-vis", [FeatureV9, FeatureV8Deprecated, FeatureVIS]>;
```

From `Target.td` and `Sparc.td` files, the resulting `SparcGenSubtarget.inc` specifies enum values to identify the features, arrays of constants to represent the CPU features and CPU subtypes, and the `ParseSubtargetFeatures` method that parses the features string that sets specified subtarget options. The generated `SparcGenSubtarget.inc` file should be included in the `SparcSubtarget.cpp`. The target-specific implementation of the `XXXSubtarget` method should follow this pseudocode:

```
XXXSubtarget::XXXSubtarget(const Module &M, const std::string &FS) {
  // Set the default features
  // Determine default and user specified characteristics of the CPU
  // Call ParseSubtargetFeatures(FS, CPU) to parse the features string
  // Perform any additional operations
}
```

JIT Support

The implementation of a target machine optionally includes a Just-In-Time (JIT) code generator that emits machine code and auxiliary structures as binary output that can be written directly to memory. To do this, implement JIT code generation by performing the following steps:

- Write an `XXXCodeEmitter.cpp` file that contains a machine function pass that transforms target-machine instructions into relocatable machine code.
- Write an `XXXJITInfo.cpp` file that implements the JIT interfaces for target-specific code-generation activities, such as emitting machine code and stubs.
- Modify `XXXTargetMachine` so that it provides a `TargetJITInfo` object through its `getJITInfo` method.

There are several different approaches to writing the JIT support code. For instance, TableGen and target descriptor files may be used for creating a JIT code generator, but are not mandatory. For the Alpha and PowerPC target machines, TableGen is used to generate `XXXGenCodeEmitter.inc`, which contains the binary coding of machine instructions and the `getBinaryCodeForInstr` method to access those codes. Other JIT implementations do not.

Both `XXXJITInfo.cpp` and `XXXCodeEmitter.cpp` must include the `llvm/CodeGen/MachineCodeEmitter.h` header file that defines the `MachineCodeEmitter` class containing code for several callback functions that write data (in bytes, words, strings, etc.) to the output stream.

Machine Code Emitter

In `XXXCodeEmitter.cpp`, a target-specific of the `Emitter` class is implemented as a function pass (subclass of `MachineFunctionPass`). The target-specific implementation of `runOnMachineFunction` (invoked by `runOnFunction` in `MachineFunctionPass`) iterates through the `MachineBasicBlock` calls `emitInstruction` to process each instruction and emit binary code. `emitInstruction` is largely implemented with case statements on the instruction types defined in `XXXInstrInfo.h`. For example, in `X86CodeEmitter.cpp`, the `emitInstruction` method is built around the following `switch`/`case` statements:

```
switch (Desc->TSFlags & X86::FormMask) {
case X86II::Pseudo:  // for not yet implemented instructions
   ...               // or pseudo-instructions
   break;
case X86II::RawFrm:  // for instructions with a fixed opcode value
   ...
   break;
case X86II::AddRegFrm: // for instructions that have one register operand
   ...                 // added to their opcode
   break;
case X86II::MRMDestReg:// for instructions that use the Mod/RM byte
   ...                 // to specify a destination (register)
   break;
case X86II::MRMDestMem:// for instructions that use the Mod/RM byte
   ...                 // to specify a destination (memory)
   break;
case X86II::MRMSrcReg: // for instructions that use the Mod/RM byte
   ...                 // to specify a source (register)
   break;
case X86II::MRMSrcMem: // for instructions that use the Mod/RM byte
   ...                 // to specify a source (memory)
   break;
case X86II::MRM0r: case X86II::MRM1r:  // for instructions that operate on
case X86II::MRM2r: case X86II::MRM3r:  // a REGISTER r/m operand and
case X86II::MRM4r: case X86II::MRM5r:  // use the Mod/RM byte and a field
case X86II::MRM6r: case X86II::MRM7r:  // to hold extended opcode data
   ...
   break;
case X86II::MRM0m: case X86II::MRM1m:  // for instructions that operate on
case X86II::MRM2m: case X86II::MRM3m:  // a MEMORY r/m operand and
case X86II::MRM4m: case X86II::MRM5m:  // use the Mod/RM byte and a field
case X86II::MRM6m: case X86II::MRM7m:  // to hold extended opcode data
   ...
   break;
case X86II::MRMInitReg: // for instructions whose source and
   ...                  // destination are the same register
   break;
}
```

The implementations of these case statements often first emit the opcode and then get the operand(s). Then depending upon the operand, helper methods may be called to process the operand(s). For example, in `X86CodeEmitter.cpp`, for the `X86II::AddRegFrm` case, the first data emitted (by `emitByte`) is the opcode added to the register operand. Then an object representing the machine operand, `MO1`, is extracted. The helper methods such as `isImmediate`, `isGlobalAddress`, `isExternalSymbol`, `isConstantPoolIndex`, and `isJumpTableIndex` determine the operand type. (`X86CodeEmitter.cpp` also has private methods such as `emitConstant`, `emitGlobalAddress`, `emitExternalSymbolAddress`, `emitConstPoolAddress`, and `emitJumpTableAddress` that emit the data into the output stream.)

```
case X86II::AddRegFrm:
  MCE.emitByte(BaseOpcode + getX86RegNum(MI.getOperand(CurOp++).getReg()));

  if (CurOp != NumOps) {
    const MachineOperand &MO1 = MI.getOperand(CurOp++);
    unsigned Size = X86InstrInfo::sizeOfImm(Desc);
    if (MO1.isImmediate())
      emitConstant(MO1.getImm(), Size);
    else {
      unsigned rt = Is64BitMode ? X86::reloc_pcrel_word
```

```
        : (IsPIC ? X86::reloc_picrel_word : X86::reloc_absolute_word);
      if (Opcode == X86::MOV64ri)
        rt = X86::reloc_absolute_dword;  // FIXME: add X86II flag?
      if (MO1.isGlobalAddress()) {
        bool NeedStub = isa<Function>(MO1.getGlobal());
        bool isLazy = gvNeedsLazyPtr(MO1.getGlobal());
        emitGlobalAddress(MO1.getGlobal(), rt, MO1.getOffset(), 0,
                          NeedStub, isLazy);
      } else if (MO1.isExternalSymbol())
        emitExternalSymbolAddress(MO1.getSymbolName(), rt);
      else if (MO1.isConstantPoolIndex())
        emitConstPoolAddress(MO1.getIndex(), rt);
      else if (MO1.isJumpTableIndex())
        emitJumpTableAddress(MO1.getIndex(), rt);
    }
  }
  break;
```

In the previous example, `XXXCodeEmitter.cpp` uses the variable `rt`, which is a `RelocationType` enum that may be used to relocate addresses (for example, a global address with a PIC base offset). The `RelocationType` enum for that target is defined in the short target-specific `XXXRelocations.h` file. The `RelocationType` is used by the `relocate` method defined in `XXXJITInfo.cpp` to rewrite addresses for referenced global symbols.

For example, `X86Relocations.h` specifies the following relocation types for the X86 addresses. In all four cases, the relocated value is added to the value already in memory. For `reloc_pcrel_word` and `reloc_picrel_word`, there is an additional initial adjustment.

```
enum RelocationType {
  reloc_pcrel_word = 0,    // add reloc value after adjusting for the PC loc
  reloc_picrel_word = 1,   // add reloc value after adjusting for the PIC base
  reloc_absolute_word = 2, // absolute relocation; no additional adjustment
  reloc_absolute_dword = 3 // absolute relocation; no additional adjustment
};
```

Target JIT Info

`XXXJITInfo.cpp` implements the JIT interfaces for target-specific code-generation activities, such as emitting machine code and stubs. At minimum, a target-specific version of `XXXJITInfo` implements the following:

- `getLazyResolverFunction` — Initializes the JIT, gives the target a function that is used for compilation.
- `emitFunctionStub` — Returns a native function with a specified address for a callback function.
- `relocate` — Changes the addresses of referenced globals, based on relocation types.
- Callback function that are wrappers to a function stub that is used when the real target is not initially known.

`getLazyResolverFunction` is generally trivial to implement. It makes the incoming parameter as the global `JITCompilerFunction` and returns the callback function that will be used a function wrapper. For the Alpha target (in `AlphaJITInfo.cpp`), the `getLazyResolverFunction` implementation is simply:

```
TargetJITInfo::LazyResolverFn AlphaJITInfo::getLazyResolverFunction(
                                            JITCompilerFn F) {
  JITCompilerFunction = F;
  return AlphaCompilationCallback;
}
```

For the X86 target, the `getLazyResolverFunction` implementation is a little more complicated, because it returns a different callback function for processors with SSE instructions and XMM registers.

The callback function initially saves and later restores the callee register values, incoming arguments, and frame and return address. The callback function needs low-level access to the registers or stack, so it is typically implemented with assembler.