The LLVM Target-Independent Code Generator

Warning
This is a work in progress.

Introduction

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target—either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler). The LLVM target-independent code generator consists of six main components:

1. Abstract target description interfaces which capture important properties about various aspects of the machine, independently of how they will be used. These interfaces are defined in `include/llvm/Target/`.
2. Classes used to represent the code being generated for a target. These classes are intended to be abstract enough to represent the machine code for *any* target machine. These classes are de-

fined in `include/llvm/CodeGen/`. At this level, concepts like "constant pool entries" and "jump tables" are explicitly exposed.
3. Classes and algorithms used to represent code at the object file level, the MC Layer. These classes represent assembly level constructs like labels, sections, and instructions. At this level, concepts like "constant pool entries" and "jump tables" don't exist.
4. Target-independent algorithms used to implement various phases of native code generation (register allocation, scheduling, stack frame representation, etc). This code lives in `lib/CodeGen/`.
5. Implementations of the abstract target description interfaces for particular targets. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target-specific passes, to build complete code generators for a specific target. Target descriptions live in `lib/Target/`.
6. The target-independent JIT components. The LLVM JIT is completely target independent (it uses the `TargetJITInfo` structure to interface for target-specific issues. The code for the target-independent JIT lives in `lib/ExecutionEngine/JIT`.

Depending on which part of the code generator you are interested in working on, different pieces of this will be useful to you. In any case, you should be familiar with the target description and machine code representation classes. If you want to add a backend for a new target, you will need to implement the target description classes for your new target and understand the LLVM code representation. If you are interested in implementing a new code generation algorithm, it should only depend on the target-description and machine code representation classes, ensuring that it is portable.

Required components in the code generator

The two pieces of the LLVM code generator are the high-level interface to the code generator and the set of reusable components that can be used to build target-specific backends. The two most important interfaces (`TargetMachine` and `DataLayout`) are the only ones that are required to be defined for a backend to fit into the LLVM system, but the others must be defined if the reusable code generator components are going to be used.

This design has two important implications. The first is that LLVM can support completely non-traditional code generation targets. For example, the C backend does not require register allocation, instruction selection, or any of the other standard components provided by the system. As such, it only implements these two interfaces, and does its own thing. Note that C backend was removed from the trunk since LLVM 3.1 release. Another example of a code generator like this is a (purely hypothetical) backend that converts LLVM to the GCC RTL form and uses GCC to emit machine code for a target.

This design also implies that it is possible to design and implement radically different code generators in the LLVM system that do not make use of any of the built-in components. Doing so is not recommended at all, but could be required for radically different targets that do not fit into the LLVM machine description model: FPGAs for example.

The high-level design of the code generator

The LLVM target-independent code generator is designed to support efficient and quality code generation for standard register-based microprocessors. Code generation in this model is divided into the following stages:

1. Instruction Selection — This phase determines an efficient way to express the input LLVM code in the target instruction set. This stage produces the initial code for the program in the target instruction set, then makes use of virtual registers in SSA form and physical registers that represent any required register assignments due to target constraints or calling conventions. This step turns the LLVM code into a DAG of target instructions.
2. Scheduling and Formation — This phase takes the DAG of target instructions produced by the instruction selection phase, determines an ordering of the instructions, then emits the instructions as `MachineInstr`s with that ordering. Note that we describe this in the instruction selection section because it operates on a SelectionDAG.
3. SSA-based Machine Code Optimizations — This optional stage consists of a series of machine-code optimizations that operate on the SSA-form produced by the instruction selector.

Optimizations like modulo-scheduling or peephole optimization work here.
4. Register Allocation — The target code is transformed from an infinite virtual register file in SSA form to the concrete register file used by the target. This phase introduces spill code and eliminates all virtual register references from the program.
5. Prolog/Epilog Code Insertion — Once the machine code has been generated for the function and the amount of stack space required is known (used for LLVM alloca's and spill slots), the prolog and epilog code for the function can be inserted and "abstract stack location references" can be eliminated. This stage is responsible for implementing optimizations like frame-pointer elimination and stack packing.
6. Late Machine Code Optimizations — Optimizations that operate on "final" machine code can go here, such as spill code scheduling and peephole optimizations.
7. Code Emission — The final stage actually puts out the code for the current function, either in the target assembler format or in machine code.

The code generator is based on the assumption that the instruction selector will use an optimal pattern matching selector to create high-quality sequences of native instructions. Alternative code generator designs based on pattern expansion and aggressive iterative peephole optimization are much slower. This design permits efficient compilation (important for JIT environments) and aggressive optimization (used when generating code offline) by allowing components of varying levels of sophistication to be used for any step of compilation.

In addition to these stages, target implementations can insert arbitrary target-specific passes into the flow. For example, the X86 target uses a special pass to handle the 80x87 floating point stack architecture. Other targets with unusual requirements can be supported with custom passes as needed.

Using TableGen for target description

The target description classes require a detailed description of the target architecture. These target descriptions often have a large amount of common information (e.g., an `add` instruction is almost identical to a `sub` instruction). In order to allow the maximum amount of commonality to be factored out, the LLVM code generator uses the TableGen Overview tool to describe big chunks of the target machine, which allows the use of domain-specific and target-specific abstractions to reduce the amount of repetition.

As LLVM continues to be developed and refined, we plan to move more and more of the target description to the `.td` form. Doing so gives us a number of advantages. The most important is that it makes it easier to port LLVM because it reduces the amount of C++ code that has to be written, and the surface area of the code generator that needs to be understood before someone can get something working. Second, it makes it easier to change things. In particular, if tables and other things are all emitted by `tblgen`, we only need a change in one place (`tblgen`) to update all of the targets to a new interface.

Target description classes

The LLVM target description classes (located in the `include/llvm/Target` directory) provide an abstract description of the target machine independent of any particular client. These classes are designed to capture the *abstract* properties of the target (such as the instructions and registers it has), and do not incorporate any particular pieces of code generation algorithms.

All of the target description classes (except the `DataLayout` class) are designed to be subclassed by the concrete target implementation, and have virtual methods implemented. To get to these implementations, the `TargetMachine` class provides accessors that should be implemented by the target.

The `TargetMachine` class

The `TargetMachine` class provides virtual methods that are used to access the target-specific implementations of the various target description classes via the `get*Info` methods (`getInstrInfo`, `getRegisterInfo`, `getFrameInfo`, etc.). This class is designed to be specialized by a concrete target implementation (e.g., `X86TargetMachine`) which implements the various virtual methods. The only re-

quired target description class is the `DataLayout` class, but if the code generator components are to be used, the other interfaces should be implemented as well.

The `DataLayout` class

The `DataLayout` class is the only required target description class, and it is the only class that is not extensible (you cannot derive a new class from it). `DataLayout` specifies information about how the target lays out memory for structures, the alignment requirements for various data types, the size of pointers in the target, and whether the target is little-endian or big-endian.

The `TargetLowering` class

The `TargetLowering` class is used by SelectionDAG based instruction selectors primarily to describe how LLVM code should be lowered to SelectionDAG operations. Among other things, this class indicates:

- an initial register class to use for various `ValueTypes`,
- which operations are natively supported by the target machine,
- the return type of `setcc` operations,
- the type to use for shift amounts, and
- various high-level characteristics, like whether it is profitable to turn division by a constant into a multiplication sequence.

The `TargetRegisterInfo` class

The `TargetRegisterInfo` class is used to describe the register file of the target and any interactions between the registers.

Registers are represented in the code generator by unsigned integers. Physical registers (those that actually exist in the target description) are unique small numbers, and virtual registers are generally large. Note that register `#0` is reserved as a flag value.

Each register in the processor description has an associated `TargetRegisterDesc` entry, which provides a textual name for the register (used for assembly output and debugging dumps) and a set of aliases (used to indicate whether one register overlaps with another).

In addition to the per-register description, the `TargetRegisterInfo` class exposes a set of processor specific register classes (instances of the `TargetRegisterClass` class). Each register class contains sets of registers that have the same properties (for example, they are all 32-bit integer registers). Each SSA virtual register created by the instruction selector has an associated register class. When the register allocator runs, it replaces virtual registers with a physical register in the set.

The target-specific implementations of these classes is auto-generated from a TableGen Overview description of the register file.

The `TargetInstrInfo` class

The `TargetInstrInfo` class is used to describe the machine instructions supported by the target. Descriptions define things like the mnemonic for the opcode, the number of operands, the list of implicit register uses and defs, whether the instruction has certain target-independent properties (accesses memory, is commutable, etc), and holds any target-specific flags.

The `TargetFrameLowering` class

The `TargetFrameLowering` class is used to provide information about the stack frame layout of the target. It holds the direction of stack growth, the known stack alignment on entry to each function, and the offset to the local area. The offset to the local area is the offset from the stack pointer on function entry to the first location where function data (local variables, spill locations) can be stored.

The `TargetSubtarget` class

The `TargetSubtarget` class is used to provide information about the specific chip set being targeted. A sub-target informs code generation of which instructions are supported, instruction latencies and instruction execution itinerary; i.e., which processing units are used, in what order, and for how long.

The `TargetJITInfo` class

The `TargetJITInfo` class exposes an abstract interface used by the Just-In-Time code generator to perform target-specific activities, such as emitting stubs. If a `TargetMachine` supports JIT code generation, it should provide one of these objects through the `getJITInfo` method.

Machine code description classes

At the high-level, LLVM code is translated to a machine specific representation formed out of `MachineFunction` , `MachineBasicBlock` , and `MachineInstr` instances (defined in `include/llvm/CodeGen`). This representation is completely target agnostic, representing instructions in their most abstract form: an opcode and a series of operands. This representation is designed to support both an SSA representation for machine code, as well as a register allocated, non-SSA form.

The `MachineInstr` class

Target machine instructions are represented as instances of the `MachineInstr` class. This class is an extremely abstract way of representing machine instructions. In particular, it only keeps track of an opcode number and a set of operands.

The opcode number is a simple unsigned integer that only has meaning to a specific backend. All of the instructions for a target should be defined in the `*InstrInfo.td` file for the target. The opcode enum values are auto-generated from this description. The `MachineInstr` class does not have any information about how to interpret the instruction (i.e., what the semantics of the instruction are); for that you must refer to the `TargetInstrInfo` class.

The operands of a machine instruction can be of several different types: a register reference, a constant integer, a basic block reference, etc. In addition, a machine operand should be marked as a def or a use of the value (though only registers are allowed to be defs).

By convention, the LLVM code generator orders instruction operands so that all register definitions come before the register uses, even on architectures that are normally printed in other orders. For example, the SPARC add instruction: "`add %i1, %i2, %i3`" adds the "%i1", and "%i2" registers and stores the result into the "%i3" register. In the LLVM code generator, the operands should be stored as "`%i3, %i1, %i2`": with the destination first.

Keeping destination (definition) operands at the beginning of the operand list has several advantages. In particular, the debugging printer will print the instruction like this:

```
%r3 = add %i1, %i2
```

Also if the first operand is a def, it is easier to create instructions whose only def is the first operand.

Using the `MachineInstrBuilder.h` functions

Machine instructions are created by using the `BuildMI` functions, located in the `include/llvm/CodeGen/MachineInstrBuilder.h` file. The `BuildMI` functions make it easy to build arbitrary machine instructions. Usage of the `BuildMI` functions look like this:

```
// Create a 'DestReg = mov 42' (rendered in X86 assembly as 'mov DestReg, 42')
// instruction and insert it at the end of the given MachineBasicBlock.
const TargetInstrInfo &TII = ...
MachineBasicBlock &MBB = ...
DebugLoc DL;
MachineInstr *MI = BuildMI(MBB, DL, TII.get(X86::MOV32ri), DestReg).addImm(42);
```

```
// Create the same instr, but insert it before a specified iterator point.
MachineBasicBlock::iterator MBBI = ...
BuildMI(MBB, MBBI, DL, TII.get(X86::MOV32ri), DestReg).addImm(42);

// Create a 'cmp Reg, 0' instruction, no destination reg.
MI = BuildMI(MBB, DL, TII.get(X86::CMP32ri8)).addReg(Reg).addImm(42);

// Create an 'sahf' instruction which takes no operands and stores nothing.
MI = BuildMI(MBB, DL, TII.get(X86::SAHF));

// Create a self looping branch instruction.
BuildMI(MBB, DL, TII.get(X86::JNE)).addMBB(&MBB);
```

If you need to add a definition operand (other than the optional destination register), you must explicitly mark it as such:

```
MI.addReg(Reg, RegState::Define);
```

Fixed (preassigned) registers

One important issue that the code generator needs to be aware of is the presence of fixed registers. In particular, there are often places in the instruction stream where the register allocator *must* arrange for a particular value to be in a particular register. This can occur due to limitations of the instruction set (e.g., the X86 can only do a 32-bit divide with the `EAX/EDX` registers), or external factors like calling conventions. In any case, the instruction selector should emit code that copies a virtual register into or out of a physical register when needed.

For example, consider this simple LLVM example:

```
define i32 @test(i32 %X, i32 %Y) {
  %Z = sdiv i32 %X, %Y
  ret i32 %Z
}
```

The X86 instruction selector might produce this machine code for the `div` and `ret`:

```
;; Start of div
%EAX = mov %reg1024           ;; Copy X (in reg1024) into EAX
%reg1027 = sar %reg1024, 31
%EDX = mov %reg1027           ;; Sign extend X into EDX
idiv %reg1025                 ;; Divide by Y (in reg1025)
%reg1026 = mov %EAX           ;; Read the result (Z) out of EAX

;; Start of ret
%EAX = mov %reg1026           ;; 32-bit return value goes in EAX
ret
```

By the end of code generation, the register allocator would coalesce the registers and delete the resultant identity moves producing the following code:

```
;; X is in EAX, Y is in ECX
mov %EAX, %EDX
sar %EDX, 31
idiv %ECX
ret
```

This approach is extremely general (if it can handle the X86 architecture, it can handle anything!) and allows all of the target specific knowledge about the instruction stream to be isolated in the instruction selector. Note that physical registers should have a short lifetime for good code generation, and all physical registers are assumed dead on entry to and exit from basic blocks (before register allocation). Thus, if you need a value to be live across basic block boundaries, it *must* live in a virtual register.

Call-clobbered registers

Some machine instructions, like calls, clobber a large number of physical registers. Rather than adding `<def,dead>` operands for all of them, it is possible to use an `MO_RegisterMask` operand instead. The register mask operand holds a bit mask of preserved registers, and everything else is considered to be clobbered by the instruction.

Machine code in SSA form

`MachineInstr`'s are initially selected in SSA-form, and are maintained in SSA-form until register allocation happens. For the most part, this is trivially simple since LLVM is already in SSA form; LLVM PHI nodes become machine code PHI nodes, and virtual registers are only allowed to have a single definition.

After register allocation, machine code is no longer in SSA-form because there are no virtual registers left in the code.

The `MachineBasicBlock` class

The `MachineBasicBlock` class contains a list of machine instructions (`MachineInstr` instances). It roughly corresponds to the LLVM code input to the instruction selector, but there can be a one-to-many mapping (i.e. one LLVM basic block can map to multiple machine basic blocks). The `MachineBasicBlock` class has a "`getBasicBlock`" method, which returns the LLVM basic block that it comes from.

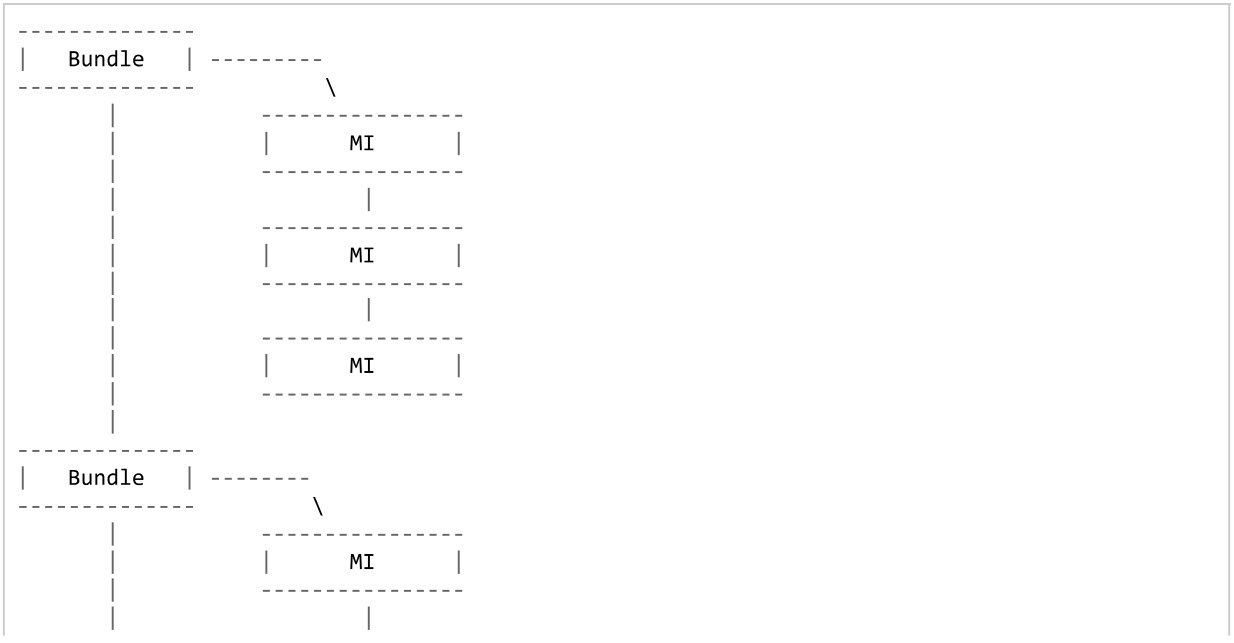
The `MachineFunction` class

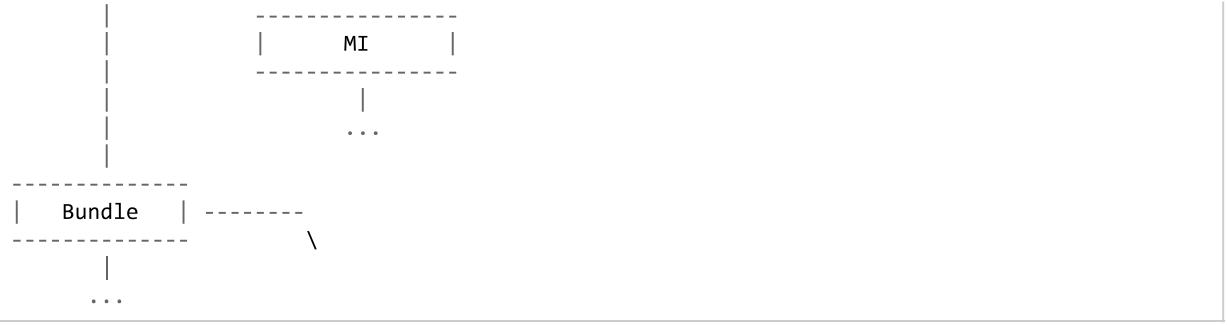
The `MachineFunction` class contains a list of machine basic blocks (`MachineBasicBlock` instances). It corresponds one-to-one with the LLVM function input to the instruction selector. In addition to a list of basic blocks, the `MachineFunction` contains a `MachineConstantPool`, a `MachineFrameInfo`, a `MachineFunctionInfo`, and a `MachineRegisterInfo`. See `include/llvm/CodeGen/MachineFunction.h` for more information.

`MachineInstr` Bundles

LLVM code generator can model sequences of instructions as MachineInstr bundles. A MI bundle can model a VLIW group / pack which contains an arbitrary number of parallel instructions. It can also be used to model a sequential list of instructions (potentially with data dependencies) that cannot be legally separated (e.g. ARM Thumb2 IT blocks).

Conceptually a MI bundle is a MI with a number of other MIs nested within:





MI bundle support does not change the physical representations of MachineBasicBlock and MachineInstr. All the MIs (including top level and nested ones) are stored as sequential list of MIs. The "bundled" MIs are marked with the 'InsideBundle' flag. A top level MI with the special BUNDLE opcode is used to represent the start of a bundle. It's legal to mix BUNDLE MIs with individual MIs that are not inside bundles nor represent bundles.

MachineInstr passes should operate on a MI bundle as a single unit. Member methods have been taught to correctly handle bundles and MIs inside bundles. The MachineBasicBlock iterator has been modified to skip over bundled MIs to enforce the bundle-as-a-single-unit concept. An alternative iterator instr_iterator has been added to MachineBasicBlock to allow passes to iterate over all of the MIs in a MachineBasicBlock, including those which are nested inside bundles. The top level BUNDLE instruction must have the correct set of register MachineOperand's that represent the cumulative inputs and outputs of the bundled MIs.

Packing / bundling of MachineInstrs for VLIW architectures should generally be done as part of the register allocation super-pass. More specifically, the pass which determines what MIs should be bundled together should be done after code generator exits SSA form (i.e. after two-address pass, PHI elimination, and copy coalescing). Such bundles should be finalized (i.e. adding BUNDLE MIs and input and output register MachineOperands) after virtual registers have been rewritten into physical registers. This eliminates the need to add virtual register operands to BUNDLE instructions which would effectively double the virtual register def and use lists. Bundles may use virtual registers and be formed in SSA form, but may not be appropriate for all use cases.

The "MC" Layer

The MC Layer is used to represent and process code at the raw machine code level, devoid of "high level" information like "constant pools", "jump tables", "global variables" or anything like that. At this level, LLVM handles things like label names, machine instructions, and sections in the object file. The code in this layer is used for a number of important purposes: the tail end of the code generator uses it to write a .s or .o file, and it is also used by the llvm-mc tool to implement standalone machine code assemblers and disassemblers.

This section describes some of the important classes. There are also a number of important subsystems that interact at this layer, they are described later in this manual.

The `MCStreamer` API

MCStreamer is best thought of as an assembler API. It is an abstract API which is *implemented* in different ways (e.g. to output a .s file, output an ELF .o file, etc) but whose API correspond directly to what you see in a .s file. MCStreamer has one method per directive, such as EmitLabel, EmitSymbolAttribute, switchSection, emitValue (for .byte, .word), etc, which directly correspond to assembly level directives. It also has an EmitInstruction method, which is used to output an MCInst to the streamer.

This API is most important for two clients: the llvm-mc stand-alone assembler is effectively a parser that parses a line, then invokes a method on MCStreamer. In the code generator, the Code Emission phase of the code generator lowers higher level LLVM IR and Machine* constructs down to the MC layer, emitting directives through MCStreamer.

On the implementation side of MCStreamer, there are two major implementations: one for writing out a .s file (MCAsmStreamer), and one for writing out a .o file (MCObjectStreamer). MCAsmStreamer is a straightforward implementation that prints out a directive for each method (e.g. `EmitValue -> .byte`), but MCObjectStreamer implements a full assembler.

For target specific directives, the MCStreamer has a MCTargetStreamer instance. Each target that needs it defines a class that inherits from it and is a lot like MCStreamer itself: It has one method per directive and two classes that inherit from it, a target object streamer and a target asm streamer. The target asm streamer just prints it (`emitFnStart -> .fnstart`), and the object streamer implement the assembler logic for it.

To make llvm use these classes, the target initialization must call TargetRegistry::RegisterAsmStreamer and TargetRegistry::RegisterMCObjectStreamer passing call-backs that allocate the corresponding target streamer and pass it to createAsmStreamer or to the appropriate object streamer constructor.

The `MCContext` class

The MCContext class is the owner of a variety of uniqued data structures at the MC layer, including symbols, sections, etc. As such, this is the class that you interact with to create symbols and sections. This class can not be subclassed.

The `MCSymbol` class

The MCSymbol class represents a symbol (aka label) in the assembly file. There are two interesting kinds of symbols: assembler temporary symbols, and normal symbols. Assembler temporary symbols are used and processed by the assembler but are discarded when the object file is produced. The distinction is usually represented by adding a prefix to the label, for example "L" labels are assembler temporary labels in MachO.

MCSymbols are created by MCContext and uniqued there. This means that MCSymbols can be compared for pointer equivalence to find out if they are the same symbol. Note that pointer inequality does not guarantee the labels will end up at different addresses though. It's perfectly legal to output something like this to the .s file:

```
foo:
bar:
  .byte 4
```

In this case, both the foo and bar symbols will have the same address.

The `MCSection` class

The `MCSection` class represents an object-file specific section. It is subclassed by object file specific implementations (e.g. `MCSectionMachO`, `MCSectionCOFF`, `MCSectionELF`) and these are created and uniqued by MCContext. The MCStreamer has a notion of the current section, which can be changed with the SwitchToSection method (which corresponds to a ".section" directive in a .s file).

The `MCInst` class

The `MCInst` class is a target-independent representation of an instruction. It is a simple class (much more so than MachineInstr) that holds a target-specific opcode and a vector of MCOperands. MCOperand, in turn, is a simple discriminated union of three cases: 1) a simple immediate, 2) a target register ID, 3) a symbolic expression (e.g. "`Lfoo-Lbar+42`") as an MCExpr.

MCInst is the common currency used to represent machine instructions at the MC layer. It is the type used by the instruction encoder, the instruction printer, and the type generated by the assembly parser and disassembler.

Object File Format

The MC layer's object writers support a variety of object formats. Because of target-specific aspects of object formats each target only supports a subset of the formats supported by the MC layer. Most targets support emitting ELF objects. Other vendor-specific objects are generally supported only on targets that are supported by that vendor (i.e. MachO is only supported on targets supported by Darwin, and XCOFF is only supported on targets that support AIX). Additionally some targets have their own object formats (i.e. DirectX, SPIR-V and WebAssembly).

The table below captures a snapshot of object file support in LLVM:

Table 99 Object File Formats

Format	Supported Targets
`COFF`	AArch64, ARM, X86
`DXContainer`	DirectX
`ELF`	AArch64, AMDGPU, ARM, AVR, BPF, CSKY, Hexagon, Lanai, LoongArch, M86k, MSP430, MIPS, PowerPC, RISCV, SPARC, SystemZ, VE, X86
`GOFF`	SystemZ
`MachO`	AArch64, ARM, X86
`SPIR-V`	SPIRV
`WASM`	WebAssembly
`XCOFF`	PowerPC

Target-independent code generation algorithms

This section documents the phases described in the high-level design of the code generator. It explains how they work and some of the rationale behind their design.

Instruction Selection

Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions. There are several well-known ways to do this in the literature. LLVM uses a SelectionDAG based instruction selector.

Portions of the DAG instruction selector are generated from the target description (`*.td`) files. Our goal is for the entire instruction selector to be generated from these `.td` files, though currently there are still things that require custom C++ code.

GlobalISel is another instruction selection framework.

Introduction to SelectionDAGs

The SelectionDAG provides an abstraction for code representation in a way that is amenable to instruction selection using automatic techniques (e.g. dynamic-programming based optimal pattern matching selectors). It is also well-suited to other phases of code generation; in particular, instruction scheduling (SelectionDAG's are very close to scheduling DAGs post-selection). Additionally, the SelectionDAG provides a host representation where a large variety of very-low-level (but target-independent) optimizations may be performed; ones which require extensive information about the instructions efficiently supported by the target.

The SelectionDAG is a Directed-Acyclic-Graph whose nodes are instances of the `SDNode` class. The primary payload of the `SDNode` is its operation code (Opcode) that indicates what operation the node performs and the operands to the operation. The various operation node types are described at the top of the `include/llvm/CodeGen/ISDOpcodes.h` file.

Although most operations define a single value, each node in the graph may define multiple values. For example, a combined div/rem operation will define both the dividend and the remainder. Many other situations require multiple values as well. Each node also has some number of operands, which are edges to the node defining the used value. Because nodes may define multiple values, edges are represented by instances of the `SDValue` class, which is a `<SDNode, unsigned>` pair, indicating the node

and result value being used, respectively. Each value produced by an `SDNode` has an associated `MVT` (Machine Value Type) indicating what the type of the value is.

SelectionDAGs contain two different kinds of values: those that represent data flow and those that represent control flow dependencies. Data values are simple edges with an integer or floating point value type. Control edges are represented as "chain" edges which are of type `MVT::Other`. These edges provide an ordering between nodes that have side effects (such as loads, stores, calls, returns, etc). All nodes that have side effects should take a token chain as input and produce a new one as output. By convention, token chain inputs are always operand #0, and chain results are always the last value produced by an operation. However, after instruction selection, the machine nodes have their chain after the instruction's operands, and may be followed by glue nodes.

A SelectionDAG has designated "Entry" and "Root" nodes. The Entry node is always a marker node with an Opcode of `ISD::EntryToken`. The Root node is the final side-effecting node in the token chain. For example, in a single basic block function it would be the return node.

One important concept for SelectionDAGs is the notion of a "legal" vs. "illegal" DAG. A legal DAG for a target is one that only uses supported operations and supported types. On a 32-bit PowerPC, for example, a DAG with a value of type i1, i8, i16, or i64 would be illegal, as would a DAG that uses a SREM or UREM operation. The legalize types and legalize operations phases are responsible for turning an illegal DAG into a legal DAG.

SelectionDAG Instruction Selection Process

SelectionDAG-based instruction selection consists of the following steps:

1. Build initial DAG — This stage performs a simple translation from the input LLVM code to an illegal SelectionDAG.
2. Optimize SelectionDAG — This stage performs simple optimizations on the SelectionDAG to simplify it, and recognize meta instructions (like rotates and `div`/`rem` pairs) for targets that support these meta operations. This makes the resultant code more efficient and the select instructions from DAG phase (below) simpler.
3. Legalize SelectionDAG Types — This stage transforms SelectionDAG nodes to eliminate any types that are unsupported on the target.
4. Optimize SelectionDAG — The SelectionDAG optimizer is run to clean up redundancies exposed by type legalization.
5. Legalize SelectionDAG Ops — This stage transforms SelectionDAG nodes to eliminate any operations that are unsupported on the target.
6. Optimize SelectionDAG — The SelectionDAG optimizer is run to eliminate inefficiencies introduced by operation legalization.
7. Select instructions from DAG — Finally, the target instruction selector matches the DAG operations to target instructions. This process translates the target-independent input DAG into another DAG of target instructions.
8. SelectionDAG Scheduling and Formation — The last phase assigns a linear order to the instructions in the target-instruction DAG and emits them into the MachineFunction being compiled. This step uses traditional prepass scheduling techniques.

After all of these steps are complete, the SelectionDAG is destroyed and the rest of the code generation passes are run.

One of the most common ways to debug these steps is using `-debug-only=isel`, which prints out the DAG, along with other information like debug info, after each of these steps. Alternatively, `-debug-only=isel-dump` shows only the DAG dumps, but the results can be filtered by function names using `-filter-print-funcs=<function names>`.

One great way to visualize what is going on here is to take advantage of a few LLC command line options. The following options pop up a window displaying the SelectionDAG at specific times (if you only get errors printed to the console while using this, you probably need to configure your system to add support for it).

- `-view-dag-combine1-dags` displays the DAG after being built, before the first optimization pass.
- `-view-legalize-dags` displays the DAG before Legalization.
- `-view-dag-combine2-dags` displays the DAG before the second optimization pass.
- `-view-isel-dags` displays the DAG before the Select phase.
- `-view-sched-dags` displays the DAG before Scheduling.

The `-view-sunit-dags` displays the Scheduler's dependency graph. This graph is based on the final SelectionDAG, with nodes that must be scheduled together bundled into a single scheduling-unit node, and with immediate operands and other nodes that aren't relevant for scheduling omitted.

The option `-filter-view-dags` allows to select the name of the basic block that you are interested to visualize and filters all the previous `view-*-dags` options.

Initial SelectionDAG Construction

The initial SelectionDAG is naïvely peephole expanded from the LLVM input by the `SelectionDAGBuilder` class. The intent of this pass is to expose as much low-level, target-specific details to the SelectionDAG as possible. This pass is mostly hard-coded (e.g. an LLVM `add` turns into an `SDNode add` while a `getelementptr` is expanded into the obvious arithmetic). This pass requires target-specific hooks to lower calls, returns, varargs, etc. For these features, the `TargetLowering` interface is used.

SelectionDAG LegalizeTypes Phase

The Legalize phase is in charge of converting a DAG to only use the types that are natively supported by the target.

There are two main ways of converting values of unsupported scalar types to values of supported types: converting small types to larger types ("promoting"), and breaking up large integer types into smaller ones ("expanding"). For example, a target might require that all f32 values are promoted to f64 and that all i1/i8/i16 values are promoted to i32. The same target might require that all i64 values be expanded into pairs of i32 values. These changes can insert sign and zero extensions as needed to make sure that the final code has the same behavior as the input.

There are two main ways of converting values of unsupported vector types to value of supported types: splitting vector types, multiple times if necessary, until a legal type is found, and extending vector types by adding elements to the end to round them out to legal types ("widening"). If a vector gets split all the way down to single-element parts with no supported vector type being found, the elements are converted to scalars ("scalarizing").

A target implementation tells the legalizer which types are supported (and which register class to use for them) by calling the `addRegisterClass` method in its `TargetLowering` constructor.

SelectionDAG Legalize Phase

The Legalize phase is in charge of converting a DAG to only use the operations that are natively supported by the target.

Targets often have weird constraints, such as not supporting every operation on every supported datatype (e.g. X86 does not support byte conditional moves and PowerPC does not support sign-extending loads from a 16-bit memory location). Legalize takes care of this by open-coding another sequence of operations to emulate the operation ("expansion"), by promoting one type to a larger type that supports the operation ("promotion"), or by using a target-specific hook to implement the legalization ("custom").

A target implementation tells the legalizer which operations are not supported (and which of the above three actions to take) by calling the `setOperationAction` method in its `TargetLowering` constructor.

If a target has legal vector types, it is expected to produce efficient machine code for common forms of the shufflevector IR instruction using those types. This may require custom legalization for

SelectionDAG vector operations that are created from the shufflevector IR. The shufflevector forms that should be handled include:

- Vector select — Each element of the vector is chosen from either of the corresponding elements of the 2 input vectors. This operation may also be known as a "blend" or "bitwise select" in target assembly. This type of shuffle maps directly to the `shuffle_vector` SelectionDAG node.
- Insert subvector — A vector is placed into a longer vector type starting at index 0. This type of shuffle maps directly to the `insert_subvector` SelectionDAG node with the `index` operand set to 0.
- Extract subvector — A vector is pulled from a longer vector type starting at index 0. This type of shuffle maps directly to the `extract_subvector` SelectionDAG node with the `index` operand set to 0.
- Splat — All elements of the vector have identical scalar elements. This operation may also be known as a "broadcast" or "duplicate" in target assembly. The shufflevector IR instruction may change the vector length, so this operation may map to multiple SelectionDAG nodes including `shuffle_vector`, `concat_vectors`, `insert_subvector`, and `extract_subvector`.

Prior to the existence of the Legalize passes, we required that every target selector supported and handled every operator and type even if they are not natively supported. The introduction of the Legalize phases allows all of the canonicalization patterns to be shared across targets, and makes it very easy to optimize the canonicalized code because it is still in the form of a DAG.

SelectionDAG Optimization Phase: the DAG Combiner

The SelectionDAG optimization phase is run multiple times for code generation, immediately after the DAG is built and once after each legalization. The first run of the pass allows the initial code to be cleaned up (e.g. performing optimizations that depend on knowing that the operators have restricted type inputs). Subsequent runs of the pass clean up the messy code generated by the Legalize passes, which allows Legalize to be very simple (it can focus on making code legal instead of focusing on generating *good* and legal code).

One important class of optimizations performed is optimizing inserted sign and zero extension instructions. We currently use ad-hoc techniques, but could move to more rigorous techniques in the future. Here are some good papers on the subject:

"Widening integer arithmetic"
Kevin Redwine and Norman Ramsey
International Conference on Compiler Construction (CC) 2004

"Effective sign extension elimination"
Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani
Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation.

SelectionDAG Select Phase

The Select phase is the bulk of the target-specific code for instruction selection. This phase takes a legal SelectionDAG as input, pattern matches the instructions supported by the target to this DAG, and produces a new DAG of target code. For example, consider the following LLVM fragment:

```
%t1 = fadd float %W, %X
%t2 = fmul float %t1, %Y
%t3 = fadd float %t2, %Z
```

This LLVM code corresponds to a SelectionDAG that looks basically like this:

```
(fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z)
```

If a target supports floating point multiply-and-add (FMA) operations, one of the adds can be merged with the multiply. On the PowerPC, for example, the output of the instruction selector might look like

this DAG:

```
(FMADDS (FADDS W, X), Y, Z)
```

The FMADDS instruction is a ternary instruction that multiplies its first two operands and adds the third (as single-precision floating-point numbers). The FADDS instruction is a simple binary single-precision add instruction. To perform this pattern match, the PowerPC backend includes the following instruction definitions:

```
def FMADDS : AForm_1<59, 29,
                     (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
                     "fmadds $FRT, $FRA, $FRC, $FRB",
                     [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),
                                            F4RC:$FRB))]>;
def FADDS : AForm_2<59, 21,
                     (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRB),
                     "fadds $FRT, $FRA, $FRB",
                     [(set F4RC:$FRT, (fadd F4RC:$FRA, F4RC:$FRB))]>;
```

The highlighted portion of the instruction definitions indicates the pattern used to match the instructions. The DAG operators (like fmul/fadd) are defined in the include/llvm/Target/TargetSelectionDAG.td file. "F4RC" is the register class of the input and result values.

The TableGen DAG instruction selector generator reads the instruction patterns in the .td file and automatically builds parts of the pattern matching code for your target. It has the following strengths:

- At compiler-compile time, it analyzes your instruction patterns and tells you if your patterns make sense or not.
- It can handle arbitrary constraints on operands for the pattern match. In particular, it is straightforward to say things like "match any immediate that is a 13-bit sign-extended value". For examples, see the immSExt16 and related tblgen classes in the PowerPC backend.
- It knows several important identities for the patterns defined. For example, it knows that addition is commutative, so it allows the FMADDS pattern above to match "(fadd X, (fmul Y, Z))" as well as "(fadd (fmul X, Y), Z)", without the target author having to specially handle this case.
- It has a full-featured type-inferencing system. In particular, you should rarely have to explicitly tell the system what type parts of your patterns are. In the FMADDS case above, we didn't have to tell tblgen that all of the nodes in the pattern are of type 'f32'. It was able to infer and propagate this knowledge from the fact that F4RC has type 'f32'.
- Targets can define their own (and rely on built-in) "pattern fragments". Pattern fragments are chunks of reusable patterns that get inlined into your patterns during compiler-compile time. For example, the integer "(not x)" operation is actually defined as a pattern fragment that expands as "(xor x, -1)", since the SelectionDAG does not have a native 'not' operation. Targets can define their own short-hand fragments as they see fit. See the definition of 'not' and 'ineg' for examples.
- In addition to instructions, targets can specify arbitrary patterns that map to one or more instructions using the 'Pat' class. For example, the PowerPC has no way to load an arbitrary integer immediate into a register in one instruction. To tell tblgen how to do this, it defines:

  ```
  // Arbitrary immediate support.  Implement in terms of LIS/ORI.
  def : Pat<(i32 imm:$imm),
            (ORI (LIS (HI16 imm:$imm)), (LO16 imm:$imm))>;
  ```

  If none of the single-instruction patterns for loading an immediate into a register match, this will be used. This rule says "match an arbitrary i32 immediate, turning it into an ORI ('or a 16-bit immediate') and an LIS ('load 16-bit immediate, where the immediate is shifted to the left 16 bits') instruction". To make this work, the LO16/HI16 node transformations are used to manipulate the input immediate (in this case, take the high or low 16-bits of the immediate).

- When using the 'Pat' class to map a pattern to an instruction that has one or more complex operands (like e.g. X86 addressing mode), the pattern may either specify the operand as a whole using a `ComplexPattern`, or else it may specify the components of the complex operand separately. The latter is done e.g. for pre-increment instructions by the PowerPC back end:

```
def STWU  : DForm_1<37, (outs ptr_rc:$ea_res), (ins GPRC:$rS, memri:$dst),
                "stwu $rS, $dst", LdStStoreUpd, []>,
                RegConstraint<"$dst.reg = $ea_res">, NoEncode<"$ea_res">;

def : Pat<(pre_store GPRC:$rS, ptr_rc:$ptrreg, iaddroff:$ptroff),
          (STWU GPRC:$rS, iaddroff:$ptroff, ptr_rc:$ptrreg)>;
```

  Here, the pair of `ptroff` and `ptrreg` operands is matched onto the complex operand `dst` of class `memri` in the `STWU` instruction.

- While the system does automate a lot, it still allows you to write custom C++ code to match special cases if there is something that is hard to express.

While it has many strengths, the system currently has some limitations, primarily because it is a work in progress and is not yet finished:

- Overall, there is no way to define or match SelectionDAG nodes that define multiple values (e.g. `SMUL_LOHI`, `LOAD`, `CALL`, etc). This is the biggest reason that you currently still *have to* write custom C++ code for your instruction selector.
- There is no great way to support matching complex addressing modes yet. In the future, we will extend pattern fragments to allow them to define multiple values (e.g. the four operands of the X86 addressing mode, which are currently matched with custom C++ code). In addition, we'll extend fragments so that a fragment can match multiple different patterns.
- We don't automatically infer flags like `isStore/isLoad` yet.
- We don't automatically generate the set of supported registers and operations for the Legalizer yet.
- We don't have a way of tying in custom legalized nodes yet.

Despite these limitations, the instruction selector generator is still quite useful for most of the binary and logical operations in typical instruction sets. If you run into any problems or can't figure out how to do something, please let Chris know!

SelectionDAG Scheduling and Formation Phase

The scheduling phase takes the DAG of target instructions from the selection phase and assigns an order. The scheduler can pick an order depending on various constraints of the machines (i.e. order for minimal register pressure or try to cover instruction latencies). Once an order is established, the DAG is converted to a list of `MachineInstrs` and the SelectionDAG is destroyed.

Note that this phase is logically separate from the instruction selection phase, but is tied to it closely in the code because it operates on SelectionDAGs.

Future directions for the SelectionDAG

1. Optional function-at-a-time selection.
2. Auto-generate entire selector from `.td` file.

SSA-based Machine Code Optimizations

To Be Written

Live Intervals

Live Intervals are the ranges (intervals) where a variable is *live*. They are used by some register allocator passes to determine if two or more virtual registers which require the same physical register are

live at the same point in the program (i.e., they conflict). When this situation occurs, one virtual register must be *spilled*.

Live Variable Analysis

The first step in determining the live intervals of variables is to calculate the set of registers that are immediately dead after the instruction (i.e., the instruction calculates the value, but it is never used) and the set of registers that are used by the instruction, but are never used after the instruction (i.e., they are killed). Live variable information is computed for each *virtual* register and *register allocatable* physical register in the function. This is done in a very efficient manner because it uses SSA to sparsely compute lifetime information for virtual registers (which are in SSA form) and only has to track physical registers within a block. Before register allocation, LLVM can assume that physical registers are only live within a single basic block. This allows it to do a single, local analysis to resolve physical register lifetimes within each basic block. If a physical register is not register allocatable (e.g., a stack pointer or condition codes), it is not tracked.

Physical registers may be live in to or out of a function. Live in values are typically arguments in registers. Live out values are typically return values in registers. Live in values are marked as such, and are given a dummy "defining" instruction during live intervals analysis. If the last basic block of a function is a `return`, then it's marked as using all live out values in the function.

`PHI` nodes need to be handled specially, because the calculation of the live variable information from a depth first traversal of the CFG of the function won't guarantee that a virtual register used by the `PHI` node is defined before it's used. When a `PHI` node is encountered, only the definition is handled, because the uses will be handled in other basic blocks.

For each `PHI` node of the current basic block, we simulate an assignment at the end of the current basic block and traverse the successor basic blocks. If a successor basic block has a `PHI` node and one of the `PHI` node's operands is coming from the current basic block, then the variable is marked as *alive* within the current basic block and all of its predecessor basic blocks, until the basic block with the defining instruction is encountered.

Live Intervals Analysis

We now have the information available to perform the live intervals analysis and build the live intervals themselves. We start off by numbering the basic blocks and machine instructions. We then handle the "live-in" values. These are in physical registers, so the physical register is assumed to be killed by the end of the basic block. Live intervals for virtual registers are computed for some ordering of the machine instructions `[1, N]`. A live interval is an interval `[i, j)`, where `1 >= i >= j > N`, for which a variable is live.

> **Note**
>
> More to come...

Register Allocation

The *Register Allocation problem* consists in mapping a program P_v, that can use an unbounded number of virtual registers, to a program P_p that contains a finite (possibly small) number of physical registers. Each target architecture has a different number of physical registers. If the number of physical registers is not enough to accommodate all the virtual registers, some of them will have to be mapped into memory. These virtuals are called *spilled virtuals*.

How registers are represented in LLVM

In LLVM, physical registers are denoted by integer numbers that normally range from 1 to 1023. To see how this numbering is defined for a particular architecture, you can read the `GenRegisterNames.inc` file for that architecture. For instance, by inspecting `lib/Target/X86/X86GenRegisterInfo.inc` we see that the 32-bit register `EAX` is denoted by 43, and the MMX register `MM0` is mapped to 65.

Some architectures contain registers that share the same physical location. A notable example is the X86 platform. For instance, in the X86 architecture, the registers `EAX`, `AX` and `AL` share the first eight bits. These physical registers are marked as *aliased* in LLVM. Given a particular architecture, you can check which registers are aliased by inspecting its `RegisterInfo.td` file. Moreover, the class `MCRegAliasIterator` enumerates all the physical registers aliased to a register.

Physical registers, in LLVM, are grouped in *Register Classes*. Elements in the same register class are functionally equivalent, and can be interchangeably used. Each virtual register can only be mapped to physical registers of a particular class. For instance, in the X86 architecture, some virtuals can only be allocated to 8 bit registers. A register class is described by `TargetRegisterClass` objects. To discover if a virtual register is compatible with a given physical, this code can be used:

```
bool RegMapping_Fer::compatible_class(MachineFunction &mf,
                                      unsigned v_reg,
                                      unsigned p_reg) {
  assert(TargetRegisterInfo::isPhysicalRegister(p_reg) &&
         "Target register must be physical");
  const TargetRegisterClass *trc = mf.getRegInfo().getRegClass(v_reg);
  return trc->contains(p_reg);
}
```

Sometimes, mostly for debugging purposes, it is useful to change the number of physical registers available in the target architecture. This must be done statically, inside the `TargetRegisterInfo.td` file. Just `grep` for `RegisterClass`, the last parameter of which is a list of registers. Just commenting some out is one simple way to avoid them being used. A more polite way is to explicitly exclude some registers from the *allocation order*. See the definition of the `GR8` register class in `lib/Target/X86/X86RegisterInfo.td` for an example of this.

Virtual registers are also denoted by integer numbers. Contrary to physical registers, different virtual registers never share the same number. Whereas physical registers are statically defined in a `TargetRegisterInfo.td` file and cannot be created by the application developer, that is not the case with virtual registers. In order to create new virtual registers, use the method `MachineRegisterInfo::createVirtualRegister()`. This method will return a new virtual register. Use an `IndexedMap<Foo, VirtReg2IndexFunctor>` to hold information per virtual register. If you need to enumerate all virtual registers, use the function `TargetRegisterInfo::index2VirtReg()` to find the virtual register numbers:

```
for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {
  unsigned VirtReg = TargetRegisterInfo::index2VirtReg(i);
  stuff(VirtReg);
}
```

Before register allocation, the operands of an instruction are mostly virtual registers, although physical registers may also be used. In order to check if a given machine operand is a register, use the boolean function `MachineOperand::isRegister()`. To obtain the integer code of a register, use `MachineOperand::getReg()`. An instruction may define or use a register. For instance, `ADD reg:1026 := reg:1025 reg:1024` defines the registers 1024, and uses registers 1025 and 1026. Given a register operand, the method `MachineOperand::isUse()` informs if that register is being used by the instruction. The method `MachineOperand::isDef()` informs if that registers is being defined.

We will call physical registers present in the LLVM bitcode before register allocation *pre-colored registers*. Pre-colored registers are used in many different situations, for instance, to pass parameters of functions calls, and to store results of particular instructions. There are two types of pre-colored registers: the ones *implicitly* defined, and those *explicitly* defined. Explicitly defined registers are normal operands, and can be accessed with `MachineInstr::getOperand(int)::getReg()`. In order to check which registers are implicitly defined by an instruction, use the `TargetInstrInfo::get(opcode)::ImplicitDefs`, where `opcode` is the opcode of the target instruction. One important difference between explicit and implicit physical registers is that the latter are defined statically for each instruction, whereas the former may vary depending on the program being compiled. For example, an instruction that represents a function call will always implicitly define or use the same set of physical registers. To read the registers implicitly used by an instruction, use

`TargetInstrInfo::get(opcode)::ImplicitUses`. Pre-colored registers impose constraints on any register allocation algorithm. The register allocator must make sure that none of them are overwritten by the values of virtual registers while still alive.

Mapping virtual registers to physical registers

There are two ways to map virtual registers to physical registers (or to memory slots). The first way, that we will call *direct mapping*, is based on the use of methods of the classes `TargetRegisterInfo`, and `MachineOperand`. The second way, that we will call *indirect mapping*, relies on the `VirtRegMap` class in order to insert loads and stores sending and getting values to and from memory.

The direct mapping provides more flexibility to the developer of the register allocator; however, it is more error prone, and demands more implementation work. Basically, the programmer will have to specify where load and store instructions should be inserted in the target function being compiled in order to get and store values in memory. To assign a physical register to a virtual register present in a given operand, use `MachineOperand::setReg(p_reg)`. To insert a store instruction, use `TargetInstrInfo::storeRegToStackSlot(...)`, and to insert a load instruction, use `TargetInstrInfo::loadRegFromStackSlot`.

The indirect mapping shields the application developer from the complexities of inserting load and store instructions. In order to map a virtual register to a physical one, use `VirtRegMap::assignVirt2Phys(vreg, preg)`. In order to map a certain virtual register to memory, use `VirtRegMap::assignVirt2StackSlot(vreg)`. This method will return the stack slot where `vreg`'s value will be located. If it is necessary to map another virtual register to the same stack slot, use `VirtRegMap::assignVirt2StackSlot(vreg, stack_location)`. One important point to consider when using the indirect mapping, is that even if a virtual register is mapped to memory, it still needs to be mapped to a physical register. This physical register is the location where the virtual register is supposed to be found before being stored or after being reloaded.

If the indirect strategy is used, after all the virtual registers have been mapped to physical registers or stack slots, it is necessary to use a spiller object to place load and store instructions in the code. Every virtual that has been mapped to a stack slot will be stored to memory after being defined and will be loaded before being used. The implementation of the spiller tries to recycle load/store instructions, avoiding unnecessary instructions. For an example of how to invoke the spiller, see `RegAllocLinearScan::runOnMachineFunction` in `lib/CodeGen/RegAllocLinearScan.cpp`.

Handling two address instructions

With very rare exceptions (e.g., function calls), the LLVM machine code instructions are three address instructions. That is, each instruction is expected to define at most one register, and to use at most two registers. However, some architectures use two address instructions. In this case, the defined register is also one of the used registers. For instance, an instruction such as `ADD %EAX, %EBX`, in X86 is actually equivalent to `%EAX = %EAX + %EBX`.

In order to produce correct code, LLVM must convert three address instructions that represent two address instructions into true two address instructions. LLVM provides the pass `TwoAddressInstructionPass` for this specific purpose. It must be run before register allocation takes place. After its execution, the resulting code may no longer be in SSA form. This happens, for instance, in situations where an instruction such as `%a = ADD %b %c` is converted to two instructions such as:

```
%a = MOVE %b
%a = ADD %a %c
```

Notice that, internally, the second instruction is represented as `ADD %a[def/use] %c`. I.e., the register operand `%a` is both used and defined by the instruction.

The SSA deconstruction phase

An important transformation that happens during register allocation is called the *SSA Deconstruction Phase*. The SSA form simplifies many analyses that are performed on the control flow graph of programs. However, traditional instruction sets do not implement PHI instructions. Thus, in order to generate executable code, compilers must replace PHI instructions with other instructions that preserve their semantics.

There are many ways in which PHI instructions can safely be removed from the target code. The most traditional PHI deconstruction algorithm replaces PHI instructions with copy instructions. That is the strategy adopted by LLVM. The SSA deconstruction algorithm is implemented in `lib/CodeGen/PHIElimination.cpp`. In order to invoke this pass, the identifier `PHIEliminationID` must be marked as required in the code of the register allocator.

Instruction folding

Instruction folding is an optimization performed during register allocation that removes unnecessary copy instructions. For instance, a sequence of instructions such as:

```
%EBX = LOAD %mem_address
%EAX = COPY %EBX
```

can be safely substituted by the single instruction:

```
%EAX = LOAD %mem_address
```

Instructions can be folded with the `TargetRegisterInfo::foldMemoryOperand(...)` method. Care must be taken when folding instructions; a folded instruction can be quite different from the original instruction. See `LiveIntervals::addIntervalsForSpills` in `lib/CodeGen/LiveIntervalAnalysis.cpp` for an example of its use.

Built in register allocators

The LLVM infrastructure provides the application developer with three different register allocators:

- *Fast* — This register allocator is the default for debug builds. It allocates registers on a basic block level, attempting to keep values in registers and reusing registers as appropriate.
- *Basic* — This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics. Since code can be rewritten on-the-fly during allocation, this framework allows interesting allocators to be developed as extensions. It is not itself a production register allocator but is a potentially useful stand-alone mode for triaging bugs and as a performance baseline.
- *Greedy — The default allocator*. This is a highly tuned implementation of the *Basic* allocator that incorporates global live range splitting. This allocator works hard to minimize the cost of spill code.
- *PBQP* — A Partitioned Boolean Quadratic Programming (PBQP) based register allocator. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment.

The type of register allocator used in `llc` can be chosen with the command line option `-regalloc=...`:

```
$ llc -regalloc=linearscan file.bc -o ln.s
$ llc -regalloc=fast file.bc -o fa.s
$ llc -regalloc=pbqp file.bc -o pbqp.s
```

Prolog/Epilog Code Insertion

Note
To Be Written

Compact Unwind

Throwing an exception requires *unwinding* out of a function. The information on how to unwind a given function is traditionally expressed in DWARF unwind (a.k.a. frame) info. But that format was originally developed for debuggers to backtrace, and each Frame Description Entry (FDE) requires ~20-30 bytes per function. There is also the cost of mapping from an address in a function to the corresponding FDE at runtime. An alternative unwind encoding is called *compact unwind* and requires just 4-bytes per function.

The compact unwind encoding is a 32-bit value, which is encoded in an architecture-specific way. It specifies which registers to restore and from where, and how to unwind out of the function. When the linker creates a final linked image, it will create a `__TEXT,__unwind_info` section. This section is a small and fast way for the runtime to access unwind info for any given function. If we emit compact unwind info for the function, that compact unwind info will be encoded in the `__TEXT,__unwind_info` section. If we emit DWARF unwind info, the `__TEXT,__unwind_info` section will contain the offset of the FDE in the `__TEXT,__eh_frame` section in the final linked image.

For X86, there are three modes for the compact unwind encoding:

Function with a Frame Pointer (``EBP`` or ``RBP``)

`EBP/RBP`-based frame, where `EBP/RBP` is pushed onto the stack immediately after the return address, then `ESP/RSP` is moved to `EBP/RBP`. Thus to unwind, `ESP/RSP` is restored with the current `EBP/RBP` value, then `EBP/RBP` is restored by popping the stack, and the return is done by popping the stack once more into the PC. All non-volatile registers that need to be restored must have been saved in a small range on the stack that starts `EBP-4` to `EBP-1020` (`RBP-8` to `RBP-1020`). The offset (divided by 4 in 32-bit mode and 8 in 64-bit mode) is encoded in bits 16-23 (mask: `0x00FF0000`). The registers saved are encoded in bits 0-14 (mask: `0x00007FFF`) as five 3-bit entries from the following table:

Compact Number	i386 Register	x86-64 Register
1	EBX	RBX
2	ECX	R12
3	EDX	R13
4	EDI	R14
5	ESI	R15
6	EBP	RBP

Frameless with a Small Constant Stack Size (``EBP`` or ``RBP`` is not used as a frame pointer)

To return, a constant (encoded in the compact unwind encoding) is added to the `ESP/RSP`. Then the return is done by popping the stack into the PC. All non-volatile registers that need to be restored must have been saved on the stack immediately after the return address. The stack size (divided by 4 in 32-bit mode and 8 in 64-bit mode) is encoded in bits 16-23 (mask: `0x00FF0000`). There is a maximum stack size of 1024 bytes in 32-bit mode and 2048 in 64-bit mode. The number of registers saved is encoded in bits 9-12 (mask: `0x00001C00`). Bits 0-9 (mask: `0x000003FF`) contain which registers were saved and their order. (See the `encodeCompactUnwindRegistersWithoutFrame()` function in `lib/Target/X86FrameLowering.cpp` for the encoding algorithm.)

Frameless with a Large Constant Stack Size (``EBP`` or ``RBP`` is not used as a frame pointer)

This case is like the "Frameless with a Small Constant Stack Size" case, but the stack size is too large to encode in the compact unwind encoding. Instead it requires that the function contains "`subl $nnnnnn, %esp`" in its prolog. The compact encoding contains the offset to the `$nnnnnn` value in the function in bits 9-12 (mask: `0x00001C00`).

Late Machine Code Optimizations

Note

To Be Written

Code Emission

The code emission step of code generation is responsible for lowering from the code generator abstractions (like MachineFunction, MachineInstr, etc) down to the abstractions used by the MC layer (MCInst, MCStreamer, etc). This is done with a combination of several different classes: the (misnamed) target-independent AsmPrinter class, target-specific subclasses of AsmPrinter (such as SparcAsmPrinter), and the TargetLoweringObjectFile class.

Since the MC layer works at the level of abstraction of object files, it doesn't have a notion of functions, global variables etc. Instead, it thinks about labels, directives, and instructions. A key class used at this time is the MCStreamer class. This is an abstract API that is implemented in different ways (e.g. to output a .s file, output an ELF .o file, etc) that is effectively an "assembler API". MCStreamer has one method per directive, such as EmitLabel, EmitSymbolAttribute, switchSection, etc, which directly correspond to assembly level directives.

If you are interested in implementing a code generator for a target, there are three important things that you have to implement for your target:

1. First, you need a subclass of AsmPrinter for your target. This class implements the general lowering process converting MachineFunction's into MC label constructs. The AsmPrinter base class provides a number of useful methods and routines, and also allows you to override the lowering process in some important ways. You should get much of the lowering for free if you are implementing an ELF, COFF, or MachO target, because the TargetLoweringObjectFile class implements much of the common logic.
2. Second, you need to implement an instruction printer for your target. The instruction printer takes an MCInst and renders it to a raw_ostream as text. Most of this is automatically generated from the .td file (when you specify something like "`add $dst, $src1, $src2`" in the instructions), but you need to implement routines to print operands.
3. Third, you need to implement code that lowers a MachineInstr to an MCInst, usually implemented in "<target>MCInstLower.cpp". This lowering process is often target specific, and is responsible for turning jump table entries, constant pool indices, global variable addresses, etc into MCLabels as appropriate. This translation layer is also responsible for expanding pseudo ops used by the code generator into the actual machine instructions they correspond to. The MCInsts that are generated by this are fed into the instruction printer or the encoder.

Finally, at your choosing, you can also implement a subclass of MCCodeEmitter which lowers MCInst's into machine code bytes and relocations. This is important if you want to support direct .o file emission, or would like to implement an assembler for your target.

Emitting function stack size information

A section containing metadata on function stack sizes will be emitted when `TargetLoweringObjectFile::StackSizesSection` is not null, and `TargetOptions::EmitStackSizeSection` is set (-stack-size-section). The section will contain an array of pairs of function symbol values (pointer size) and stack sizes (unsigned LEB128). The stack size values only include the space allocated in the function prologue. Functions with dynamic stack allocations are not included.

VLIW Packetizer

In a Very Long Instruction Word (VLIW) architecture, the compiler is responsible for mapping instructions to functional-units available on the architecture. To that end, the compiler creates groups of instructions called *packets* or *bundles*. The VLIW packetizer in LLVM is a target-independent mechanism to enable the packetization of machine instructions.

Mapping from instructions to functional units

Instructions in a VLIW target can typically be mapped to multiple functional units. During the process of packetizing, the compiler must be able to reason about whether an instruction can be added to a packet. This decision can be complex since the compiler has to examine all possible mappings of instructions to functional units. Therefore to alleviate compilation-time complexity, the VLIW packetizer parses the instruction classes of a target and generates tables at compiler build time. These tables can then be queried by the provided machine-independent API to determine if an instruction can be accommodated in a packet.

How the packetization tables are generated and used

The packetizer reads instruction classes from a target's itineraries and creates a deterministic finite automaton (DFA) to represent the state of a packet. A DFA consists of three major elements: inputs, states, and transitions. The set of inputs for the generated DFA represents the instruction being added to a packet. The states represent the possible consumption of functional units by instructions in a packet. In the DFA, transitions from one state to another occur on the addition of an instruction to an existing packet. If there is a legal mapping of functional units to instructions, then the DFA contains a corresponding transition. The absence of a transition indicates that a legal mapping does not exist and that the instruction cannot be added to the packet.

To generate tables for a VLIW target, add *Target*GenDFAPacketizer.inc as a target to the Makefile in the target directory. The exported API provides three functions: `DFAPacketizer::clearResources()`, `DFAPacketizer::reserveResources(MachineInstr *MI)`, and `DFAPacketizer::canReserveResources(MachineInstr *MI)`. These functions allow a target packetizer to add an instruction to an existing packet and to check whether an instruction can be added to a packet. See `llvm/CodeGen/DFAPacketizer.h` for more information.

Implementing a Native Assembler

Though you're probably reading this because you want to write or maintain a compiler backend, LLVM also fully supports building a native assembler. We've tried hard to automate the generation of the assembler from the .td files (in particular the instruction syntax and encodings), which means that a large part of the manual and repetitive data entry can be factored and shared with the compiler.

Instruction Parsing

Note
To Be Written

Instruction Alias Processing

Once the instruction is parsed, it enters the MatchInstructionImpl function. The MatchInstructionImpl function performs alias processing and then does actual matching.

Alias processing is the phase that canonicalizes different lexical forms of the same instructions down to one representation. There are several different kinds of alias that are possible to implement and they are listed below in the order that they are processed (which is in order from simplest/weakest to most complex/powerful). Generally you want to use the first alias mechanism that meets the needs of your instruction, because it will allow a more concise description.

Mnemonic Aliases

The first phase of alias processing is simple instruction mnemonic remapping for classes of instructions which are allowed with two different mnemonics. This phase is a simple and unconditionally remapping from one input mnemonic to one output mnemonic. It isn't possible for this form of alias to look at the operands at all, so the remapping must apply for all forms of a given mnemonic. Mnemonic aliases are defined simply, for example X86 has:

```
def : MnemonicAlias<"cbw",     "cbtw">;
def : MnemonicAlias<"smovq",   "movsq">;
def : MnemonicAlias<"fldcww",  "fldcw">;
def : MnemonicAlias<"fucompi", "fucomip">;
def : MnemonicAlias<"ud2a",    "ud2">;
```

... and many others. With a MnemonicAlias definition, the mnemonic is remapped simply and directly. Though MnemonicAlias's can't look at any aspect of the instruction (such as the operands) they can depend on global modes (the same ones supported by the matcher), through a Requires clause:

```
def : MnemonicAlias<"pushf", "pushfq">, Requires<[In64BitMode]>;
def : MnemonicAlias<"pushf", "pushfl">, Requires<[In32BitMode]>;
```

In this example, the mnemonic gets mapped into a different one depending on the current instruction set.

Instruction Aliases

The most general phase of alias processing occurs while matching is happening: it provides new forms for the matcher to match along with a specific instruction to generate. An instruction alias has two parts: the string to match and the instruction to generate. For example:

```
def : InstAlias<"movsx $src, $dst", (MOVSX16rr8W GR16:$dst, GR8  :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX16rm8W GR16:$dst, i8mem:$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX32rr8  GR32:$dst, GR8  :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX32rr16 GR32:$dst, GR16 :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX64rr8  GR64:$dst, GR8  :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX64rr16 GR64:$dst, GR16 :$src)>;
def : InstAlias<"movsx $src, $dst", (MOVSX64rr32 GR64:$dst, GR32 :$src)>;
```

This shows a powerful example of the instruction aliases, matching the same mnemonic in multiple different ways depending on what operands are present in the assembly. The result of instruction aliases can include operands in a different order than the destination instruction, and can use an input multiple times, for example:

```
def : InstAlias<"clrb $reg", (XOR8rr  GR8 :$reg, GR8 :$reg)>;
def : InstAlias<"clrw $reg", (XOR16rr GR16:$reg, GR16:$reg)>;
def : InstAlias<"clrl $reg", (XOR32rr GR32:$reg, GR32:$reg)>;
def : InstAlias<"clrq $reg", (XOR64rr GR64:$reg, GR64:$reg)>;
```

This example also shows that tied operands are only listed once. In the X86 backend, XOR8rr has two input GR8's and one output GR8 (where an input is tied to the output). InstAliases take a flattened operand list without duplicates for tied operands. The result of an instruction alias can also use immediates and fixed physical registers which are added as simple immediate operands in the result, for example:

```
// Fixed Immediate operand.
def : InstAlias<"aad", (AAD8i8 10)>;

// Fixed register operand.
def : InstAlias<"fcomi", (COM_FIr ST1)>;

// Simple alias.
def : InstAlias<"fcomi $reg", (COM_FIr RST:$reg)>;
```

Instruction aliases can also have a Requires clause to make them subtarget specific.

If the back-end supports it, the instruction printer can automatically emit the alias rather than what's being aliased. It typically leads to better, more readable code. If it's better to print out what's being aliased, then pass a '0' as the third parameter to the InstAlias definition.

Instruction Matching

Note
To Be Written

Target-specific Implementation Notes

This section of the document explains features or design decisions that are specific to the code generator for a particular target.

Tail call optimization

Tail call optimization, callee reusing the stack of the caller, is currently supported on x86/x86-64, PowerPC, AArch64, and WebAssembly. It is performed on x86/x86-64, PowerPC, and AArch64 if:

- Caller and callee have the calling convention `fastcc`, `cc 10` (GHC calling convention), `cc 11` (HiPE calling convention), `tailcc`, or `swifttailcc`.
- The call is a tail call - in tail position (ret immediately follows call and ret uses value of call or is void).
- Option `-tailcallopt` is enabled or the calling convention is `tailcc`.
- Platform-specific constraints are met.

x86/x86-64 constraints:

- No variable argument lists are used.
- On x86-64 when generating GOT/PIC code only module-local calls (visibility = hidden or protected) are supported.

PowerPC constraints:

- No variable argument lists are used.
- No byval parameters are used.
- On ppc32/64 GOT/PIC only module-local calls (visibility = hidden or protected) are supported.

WebAssembly constraints:

- No variable argument lists are used
- The 'tail-call' target attribute is enabled.
- The caller and callee's return types must match. The caller cannot be void unless the callee is, too.

AArch64 constraints:

- No variable argument lists are used.

Example:

Call as `llc -tailcallopt test.ll`.

```
declare fastcc i32 @tailcallee(i32 inreg %a1, i32 inreg %a2, i32 %a3, i32 %a4)

define fastcc i32 @tailcaller(i32 %in1, i32 %in2) {
  %l1 = add i32 %in1, %in2
  %tmp = tail call fastcc i32 @tailcallee(i32 inreg %in1, i32 inreg %in2, i32 %in1, i32 %l1)
  ret i32 %tmp
}
```

Implications of `-tailcallopt`:

To support tail call optimization in situations where the callee has more arguments than the caller a 'callee pops arguments' convention is used. This currently causes each `fastcc` call that is not tail call optimized (because one or more of above constraints are not met) to be followed by a readjustment of the stack. So performance might be worse in such cases.

Sibling call optimization

Sibling call optimization is a restricted form of tail call optimization. Unlike tail call optimization described in the previous section, it can be performed automatically on any tail calls when `-tailcallopt` option is not specified.

Sibling call optimization is currently performed on x86/x86-64 when the following constraints are met:

- Caller and callee have the same calling convention. It can be either `c` or `fastcc`.
- The call is a tail call - in tail position (ret immediately follows call and ret uses value of call or is void).
- Caller and callee have matching return type or the callee result is not used.
- If any of the callee arguments are being passed in stack, they must be available in caller's own incoming argument stack and the frame offsets must be the same.

Example:

```
declare i32 @bar(i32, i32)

define i32 @foo(i32 %a, i32 %b, i32 %c) {
entry:
  %0 = tail call i32 @bar(i32 %a, i32 %b)
  ret i32 %0
}
```

The X86 backend

The X86 code generator lives in the `lib/Target/X86` directory. This code generator is capable of targeting a variety of x86-32 and x86-64 processors, and includes support for ISA extensions such as MMX and SSE.

X86 Target Triples supported

The following are the known target triples that are supported by the X86 backend. This is not an exhaustive list, and it would be useful to add those that people test.

- **i686-pc-linux-gnu** — Linux
- **i386-unknown-freebsd5.3** — FreeBSD 5.3
- **i686-pc-cygwin** — Cygwin on Win32
- **i686-pc-mingw32** — MingW on Win32
- **i386-pc-mingw32msvc** — MingW crosscompiler on Linux
- **i686-apple-darwin*** — Apple Darwin on X86
- **x86_64-unknown-linux-gnu** — Linux

X86 Calling Conventions supported

The following target-specific calling conventions are known to backend:

- **x86_StdCall** — stdcall calling convention seen on Microsoft Windows platform (CC ID = 64).
- **x86_FastCall** — fastcall calling convention seen on Microsoft Windows platform (CC ID = 65).
- **x86_ThisCall** — Similar to X86_StdCall. Passes first argument in ECX, others via stack. Callee is responsible for stack cleaning. This convention is used by MSVC by default for methods in its ABI (CC ID = 70).

Representing X86 addressing modes in MachineInstrs

The x86 has a very flexible way of accessing memory. It is capable of forming memory addresses of the following expression directly in integer instructions (which use ModR/M addressing):

```
SegmentReg: Base + [1,2,4,8] * IndexReg + Disp32
```

In order to represent this, LLVM tracks no less than 5 operands for each memory operand of this form. This means that the "load" form of '`mov`' has the following `MachineOperands` in this order:

```
Index:        0     |    1       2       3           4         5
Meaning:   DestReg, | BaseReg,  Scale, IndexReg, Displacement Segment
OperandTy: VirtReg, | VirtReg, UnsImm, VirtReg,   SignExtImm  PhysReg
```

Stores, and all other instructions, treat the four memory operands in the same way and in the same order. If the segment register is unspecified (regno = 0), then no segment override is generated. "Lea" operations do not have a segment register specified, so they only have 4 operands for their memory reference.

X86 address spaces supported

x86 has a feature which provides the ability to perform loads and stores to different address spaces via the x86 segment registers. A segment override prefix byte on an instruction causes the instruction's memory access to go to the specified segment. LLVM address space 0 is the default address space, which includes the stack, and any unqualified memory accesses in a program. Address spaces 1-255 are currently reserved for user-defined code. The GS-segment is represented by address space 256, the FS-segment is represented by address space 257, and the SS-segment is represented by address space 258. Other x86 segments have yet to be allocated address space numbers.

While these address spaces may seem similar to TLS via the `thread_local` keyword, and often use the same underlying hardware, there are some fundamental differences.

The `thread_local` keyword applies to global variables and specifies that they are to be allocated in thread-local memory. There are no type qualifiers involved, and these variables can be pointed to with normal pointers and accessed with normal loads and stores. The `thread_local` keyword is target-independent at the LLVM IR level (though LLVM doesn't yet have implementations of it for some configurations)

Special address spaces, in contrast, apply to static types. Every load and store has a particular address space in its address operand type, and this is what determines which address space is accessed. LLVM ignores these special address space qualifiers on global variables, and does not provide a way to directly allocate storage in them. At the LLVM IR level, the behavior of these special address spaces depends in part on the underlying OS or runtime environment, and they are specific to x86 (and LLVM doesn't yet handle them correctly in some cases).

Some operating systems and runtime environments use (or may in the future use) the FS/GS-segment registers for various low-level purposes, so care should be taken when considering them.

Instruction naming

An instruction name consists of the base name, a default operand size, and a character per operand with an optional special size. For example:

```
ADD8rr      -> add, 8-bit register, 8-bit register
IMUL16rmi   -> imul, 16-bit register, 16-bit memory, 16-bit immediate
IMUL16rmi8  -> imul, 16-bit register, 16-bit memory, 8-bit immediate
MOVSX32rm16 -> movsx, 32-bit register, 16-bit memory
```

The PowerPC backend

The PowerPC code generator lives in the lib/Target/PowerPC directory. The code generation is retargetable to several variations or *subtargets* of the PowerPC ISA; including ppc32, ppc64 and altivec.

LLVM PowerPC ABI

LLVM follows the AIX PowerPC ABI, with two deviations. LLVM uses a PC relative (PIC) or static addressing for accessing global values, so no TOC (r2) is used. Second, r31 is used as a frame pointer to allow dynamic growth of a stack frame. LLVM takes advantage of having no TOC to provide space to save the frame pointer in the PowerPC linkage area of the caller frame. Other details of PowerPC ABI can be found at PowerPC ABI. Note: This link describes the 32 bit ABI. The 64 bit ABI is similar except space for GPRs are 8 bytes wide (not 4) and r13 is reserved for system use.

Frame Layout

The size of a PowerPC frame is usually fixed for the duration of a function's invocation. Since the frame is fixed size, all references into the frame can be accessed via fixed offsets from the stack pointer. The exception to this is when dynamic alloca or variable sized arrays are present, then a base pointer (r31) is used as a proxy for the stack pointer and stack pointer is free to grow or shrink. A base pointer is also used if llvm-gcc is not passed the -fomit-frame-pointer flag. The stack pointer is always aligned to 16 bytes, so that space allocated for altivec vectors will be properly aligned.

An invocation frame is laid out as follows (low memory at top):

Linkage
Parameter area
Dynamic area
Locals area
Saved registers area

Previous Frame

The *linkage* area is used by a callee to save special registers prior to allocating its own frame. Only three entries are relevant to LLVM. The first entry is the previous stack pointer (sp), aka link. This allows probing tools like gdb or exception handlers to quickly scan the frames in the stack. A function epilog can also use the link to pop the frame from the stack. The third entry in the linkage area is used to save the return address from the lr register. Finally, as mentioned above, the last entry is used to save the previous frame pointer (r31.) The entries in the linkage area are the size of a GPR, thus the linkage area is 24 bytes long in 32 bit mode and 48 bytes in 64 bit mode.

32 bit linkage area:

0	Saved SP (r1)
4	Saved CR
8	Saved LR
12	Reserved
16	Reserved
20	Saved FP (r31)

64 bit linkage area:

0	Saved SP (r1)
8	Saved CR
16	Saved LR
24	Reserved

32	Reserved
40	Saved FP (r31)

The *parameter area* is used to store arguments being passed to a callee function. Following the PowerPC ABI, the first few arguments are actually passed in registers, with the space in the parameter area unused. However, if there are not enough registers or the callee is a thunk or vararg function, these register arguments can be spilled into the parameter area. Thus, the parameter area must be large enough to store all the parameters for the largest call sequence made by the caller. The size must also be minimally large enough to spill registers r3–r10. This allows callees blind to the call signature, such as thunks and vararg functions, enough space to cache the argument registers. Therefore, the parameter area is minimally 32 bytes (64 bytes in 64 bit mode.) Also note that since the parameter area is a fixed offset from the top of the frame, that a callee can access its split arguments using fixed offsets from the stack pointer (or base pointer.)

Combining the information about the linkage, parameter areas and alignment. A stack frame is minimally 64 bytes in 32 bit mode and 128 bytes in 64 bit mode.

The *dynamic area* starts out as size zero. If a function uses dynamic alloca then space is added to the stack, the linkage and parameter areas are shifted to top of stack, and the new space is available immediately below the linkage and parameter areas. The cost of shifting the linkage and parameter areas is minor since only the link value needs to be copied. The link value can be easily fetched by adding the original frame size to the base pointer. Note that allocations in the dynamic space need to observe 16 byte alignment.

The *locals area* is where the llvm compiler reserves space for local variables.

The *saved registers area* is where the llvm compiler spills callee saved registers on entry to the callee.

Prolog/Epilog

The llvm prolog and epilog are the same as described in the PowerPC ABI, with the following exceptions. Callee saved registers are spilled after the frame is created. This allows the llvm epilog/prolog support to be common with other targets. The base pointer callee saved register r31 is saved in the TOC slot of linkage area. This simplifies allocation of space for the base pointer and makes it convenient to locate programmatically and during debugging.

Dynamic Allocation

Note
TODO - More to come.

The NVPTX backend

The NVPTX code generator under lib/Target/NVPTX is an open-source version of the NVIDIA NVPTX code generator for LLVM. It is contributed by NVIDIA and is a port of the code generator used in the CUDA compiler (nvcc). It targets the PTX 3.0/3.1 ISA and can target any compute capability greater than or equal to 2.0 (Fermi).

This target is of production quality and should be completely compatible with the official NVIDIA toolchain.

Code Generator Options:

Option	Description
sm_20	Set shader model/compute capability to 2.0
sm_21	Set shader model/compute capability to 2.1
sm_30	Set shader model/compute capability to 3.0
sm_35	Set shader model/compute capability to 3.5

ptx30	Target PTX 3.0
ptx31	Target PTX 3.1

The extended Berkeley Packet Filter (eBPF) backend

Extended BPF (or eBPF) is similar to the original ("classic") BPF (cBPF) used to filter network packets. The bpf() system call performs a range of operations related to eBPF. For both cBPF and eBPF programs, the Linux kernel statically analyzes the programs before loading them, in order to ensure that they cannot harm the running system. eBPF is a 64-bit RISC instruction set designed for one to one mapping to 64-bit CPUs. Opcodes are 8-bit encoded, and 87 instructions are defined. There are 10 registers, grouped by function as outlined below.

```
R0        return value from in-kernel functions; exit value for eBPF program
R1 - R5   function call arguments to in-kernel functions
R6 - R9   callee-saved registers preserved by in-kernel functions
R10       stack frame pointer (read only)
```

Instruction encoding (arithmetic and jump)

eBPF is reusing most of the opcode encoding from classic to simplify conversion of classic BPF to eBPF. For arithmetic and jump instructions the 8-bit 'code' field is divided into three parts:

```
+----------------+--------+--------------------+
|   4 bits       |  1 bit |   3 bits           |
| operation code | source | instruction class  |
+----------------+--------+--------------------+
(MSB)                                  (LSB)
```

Three LSB bits store instruction class which is one of:

```
BPF_LD     0x0
BPF_LDX    0x1
BPF_ST     0x2
BPF_STX    0x3
BPF_ALU    0x4
BPF_JMP    0x5
(unused)   0x6
BPF_ALU64  0x7
```

When BPF_CLASS(code) == BPF_ALU or BPF_ALU64 or BPF_JMP, 4th bit encodes source operand

```
BPF_X     0x1  use src_reg register as source operand
BPF_K     0x0  use 32 bit immediate as source operand
```

and four MSB bits store operation code

```
BPF_ADD   0x0  add
BPF_SUB   0x1  subtract
BPF_MUL   0x2  multiply
BPF_DIV   0x3  divide
BPF_OR    0x4  bitwise logical OR
BPF_AND   0x5  bitwise logical AND
BPF_LSH   0x6  left shift
BPF_RSH   0x7  right shift (zero extended)
BPF_NEG   0x8  arithmetic negation
BPF_MOD   0x9  modulo
BPF_XOR   0xa  bitwise logical XOR
BPF_MOV   0xb  move register to register
BPF_ARSH  0xc  right shift (sign extended)
BPF_END   0xd  endianness conversion
```

If BPF_CLASS(code) == BPF_JMP, BPF_OP(code) is one of

```
BPF_JA    0x0  unconditional jump
BPF_JEQ   0x1  jump ==
BPF_JGT   0x2  jump >
BPF_JGE   0x3  jump >=
BPF_JSET  0x4  jump if (DST & SRC)
BPF_JNE   0x5  jump !=
BPF_JSGT  0x6  jump signed >
BPF_JSGE  0x7  jump signed >=
BPF_CALL  0x8  function call
BPF_EXIT  0x9  function return
```

Instruction encoding (load, store)

For load and store instructions the 8-bit 'code' field is divided as:

```
+--------+--------+-------------------+
| 3 bits | 2 bits |   3 bits          |
|  mode  |  size  | instruction class |
+--------+--------+-------------------+
(MSB)                            (LSB)
```

Size modifier is one of

```
BPF_W       0x0  word
BPF_H       0x1  half word
BPF_B       0x2  byte
BPF_DW      0x3  double word
```

Mode modifier is one of

```
BPF_IMM     0x0  immediate
BPF_ABS     0x1  used to access packet data
BPF_IND     0x2  used to access packet data
BPF_MEM     0x3  memory
(reserved)  0x4
(reserved)  0x5
BPF_XADD    0x6  exclusive add
```

Packet data access (BPF_ABS, BPF_IND)

Two non-generic instructions: (BPF_ABS | <size> | BPF_LD) and (BPF_IND | <size> | BPF_LD) which are used to access packet data. Register R6 is an implicit input that must contain pointer to sk_buff. Register R0 is an implicit output which contains the data fetched from the packet. Registers R1-R5 are scratch registers and must not be used to store the data across BPF_ABS | BPF_LD or BPF_IND | BPF_LD instructions. These instructions have implicit program exit condition as well. When eBPF program is trying to access the data beyond the packet boundary, the interpreter will abort the execution of the program.

BPF_IND | BPF_W | BPF_LD is equivalent to:

R0 = ntohl(*(u32 *) (((struct sk_buff *) R6)->data + src_reg + imm32))

eBPF maps

eBPF maps are provided for sharing data between kernel and user-space. Currently implemented types are hash and array, with potential extension to support bloom filters, radix trees, etc. A map is defined by its type, maximum number of elements, key size and value size in bytes. eBPF syscall supports create, update, find and delete functions on maps.

Function calls

Function call arguments are passed using up to five registers (R1 - R5). The return value is passed in a dedicated register (R0). Four additional registers (R6 - R9) are callee-saved, and the values in these

registers are preserved within kernel functions. R0 - R5 are scratch registers within kernel functions, and eBPF programs must therefor store/restore values in these registers if needed across function calls. The stack can be accessed using the read-only frame pointer R10. eBPF registers map 1:1 to hardware registers on x86_64 and other 64-bit architectures. For example, x86_64 in-kernel JIT maps them as

```
R0 - rax
R1 - rdi
R2 - rsi
R3 - rdx
R4 - rcx
R5 - r8
R6 - rbx
R7 - r13
R8 - r14
R9 - r15
R10 - rbp
```

since x86_64 ABI mandates rdi, rsi, rdx, rcx, r8, r9 for argument passing and rbx, r12 - r15 are callee saved.

Program start

An eBPF program receives a single argument and contains a single eBPF main routine; the program does not contain eBPF functions. Function calls are limited to a predefined set of kernel functions. The size of a program is limited to 4K instructions: this ensures fast termination and a limited number of kernel function calls. Prior to running an eBPF program, a verifier performs static analysis to prevent loops in the code and to ensure valid register usage and operand types.

The AMDGPU backend

The AMDGPU code generator lives in the `lib/Target/AMDGPU` directory. This code generator is capable of targeting a variety of AMD GPU processors. Refer to User Guide for AMDGPU Backend for more information.