LLC: Walkthrough

Table of Contents

1 LLVM target-independent 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.

1.1 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**.

1.2 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**.

1.3 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.

1.4 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.

1.5 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 allocas 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.

1.6 Late machine code optimizations

Optimizations that operate on “final” machine code can go here, such as spill code scheduling and peephole optimizations.

1.7 Code emission

The final stage actually puts out the code for the current function, either in the target assembler format or in machine code.

2 LLVM target description classes

The LLVM target, described by six different classes (located in `include/llvm/Target`), provides an abstract description of the target machine. These classes are designed to represent abstract target properties such as the instructions, registers it has, etc.

2.1 TargetMachine class

The **TargetMachine** class provides virtual methods used to accecss the target-specific implementations of the various target description classes via the `get*Info` methods (`getInstrInfo`, `getRegisterInfo`, `getFrameInfo`, etc.). This class is to be specialized by a concrete target implementatino (e.g. `X86TargetMachine`).

2.2 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.

2.3 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 **ValueType**s,
- 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.

2.4 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 autogenerated from a **TableGen** description of the register file.

2.5 TargetInstrInfo class

The **TargetInstrInfo** class is used to describe the machine instructions supported by the target. It is essentially an array of TargetInstrDescriptor objects, each of which describes one instruction the target supports. Descriptors 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.

2.6 TargetFrameInfo class

The **TargetFrameInfo** 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.

2.7 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.

2.8 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.

3 LLVM 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 (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.

3.1 MachineInstr class

3.2 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

3.3 MachineFunction class

The **MachineFunction** 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.

3.4 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:

4 LLVM "MC" layer

The **MC layer** is 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.

4.1 MCStreamer API

4.2 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.

4.3 MCSymbol class

The **MCSymbol** class represents a **symbol** (a.k.a. **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. Its 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.

4.4 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).

4.5 MCInst class

5 Initialization

5.1 sys::printStackTraceOnErrorSignal()

- register **printStackTrace** as a signal handler

5.2 PrettyStackTraceProgram

- add **CrashHandler** as another signal handler

5.3 getGlobalContext

- `LLVMContext &context = getGlobalContext()`
- **LLVMContext**: container of "global" state in LLVM (e.g. global type and constant uniquing tables) (**ManagedStatic** object)

5.4 llvm_shutdown_obj

- `llvm_shutdown_obj Y`: allocate object on stack (so that it will be deallocated on exit from this stack frame)
- dtor of this object contains the `llvm_shutdown()` function to be called when exit
  - deallocate and destroy all **ManagedStatic** objects
- **ManagedStatic**: lazily constructed on demand (goof for reducing startup times of dynamic libraries that link to LLVM components) and makes destruction to be explicit through the `llvm_shutdown()` function call

5.5 Initialize Targets, TargetMCs, AsmPrinters, AsmParsers

- **Target**:
  - e.g. Sparc, X86, PowerPC, MBlaze, MSP430
- **TargetMC**:
- **AsmPrinter**:
- **AsmParser**:

5.6 cl::ParseCommandLineOptions

5.7 SMDiagnostic Err

5.8 std::auto_ptr Module M

5.9 M.reset(ParseIRFile(inputfilename, Err, Context))

- **MemoryBuffer**: simple read-only access to block of memory and provides methods for reading FILES or STDIN into a memory buffer
- **ParseIRFile** returns **Module***, which is set to `M`
- `OwningPtr<MemoryBuffer> File;`
- `MemoryBuffer::getFileOrSTDIN(filename.c_str(), File)`

5.10 Triple TheTriple(mod.getTargetTriple())

5.11 GetOutputStram

5.12 PassManager PM

- `PM = new PassManagerImpl();`
- `PM->setTopLevelManager(PM);`
- **PassManager** manages a tree of **Pass**es, where its implemenation is delegated to **PassManagerImpl**
- `PassManagerImpl::add(Pass *P)`
- **Pass**: interface implemented by all "passes"; there are multiple subclasses of passes (e.g. **BasicBlockPass**, **FunctionPass**, **ModulePass**, etc.)
- see **PassManagers.h** for comments

5.13 PM.add(TargetData(*Target.getTargetData()))

5.14 PM.Run or PassManager::run(Module &M)

- **dumpArguments()**
- **dumpPasses()**
- **initializeAllAnalysisInfo()**
  - **PassManagers** contains a vector of **PMDataManager** which manages the analysis data used by pass managers
- `getContainedmanager(idx)->runOnModule(M);`

6 Passes in -O3

6.1 Optimization Passes

- Target Data Layout
- No Alias Analysis (always returns 'may' alias)
- Type-Based Alias Analysis
- Basic Alias Analysis (stateless AA impl)
- Create Garbage Collector Module Metadata
- Machine Module Information
- ModulePass Manager
  - FunctionPass Manager
    - Preliminary module verification
    - Dominator Tree Construction
    - Module Verifier
    - Natural Loop Information
    - Loop Pass Manager
      1. Canonicalize natural loops
    - Scalar Evolution Analysis
    - Loop Pass Manager
      1. Canonicalize natural loops
      2. Induction Variable Users
      3. Loop Strength Reduction
    - Lower Garbage Collection Instructions
    - Remove unreachable blocks from the CFG
    - Exception handling preparation
    - Optimize for code generation
    - Insert stack protectors
    - Preliminary module verification
    - Module Verifier

6.2 Code Generation Passes

- Machine Function Analysis
- X86 DAG→DAG Instruction Selection
- X86 PIC Global Base Reg Initialization
- Expand ISel Pseudo-instructions
- Optimize machine instruction PHIs
- Local Stack Slot Allocation
- Remove dead machine instructions
- MachineDominator Tree Construction
- Machine Natural Loop Construction
- Machine Instruction LICM
- Machine Common Subexpression Elimination
- Machine code sinking
- Peephole Optimizations
- Tail Duplication
- X86 Maximal Stack Alignment Check
- Remove unreachable machine basic blocks

- Live Variable Analysis
- MachineDominator Tree Construction
- Machine Natural Loop Construction
- Eliminate ϕ-nodes for register allocation
- Two-Address instruction pass
- Process Implicit Definitions
- Slot index numbering
- Live Interval Analysis
- Debug Variable Analysis
- Simple Register Coalescing
- Calculate spill weights
- Live Stack Slot Analysis
- Virtual Register Map
- Linear Scan Register Allocator
- Stack Slot Coloring
- Machine Instruction LICM
- Bundle Machine CFG Edges
- X86 FP Stackifier
- Subregister lowering instruction pass
- Prolog/Epilog Insertion & Frame Finalization
- Post RA top-down list latency scheduler
- Control Flow Optimizer
- Tail Duplication
- Analyze Machine Code For Garbage Collection
- MachineDominator Tree Construction
- Machine Natural Loop Construction
- Code Placement Optimizer
- SSE execution domain fixup
- Machine Natural Loop Construction
- X86 AT&T-Style Assembly Printer
- Delete Garbage Collector Information

7 Analysis Passes

7.1 Alias Analysis

- used to determine if a storage location may be accessed in more than one way; two pointers are **aliased** if they point to the same memory location
- typically AA respond to a query with **must, may,** or **no** response
- many different variations:
  - **type-based alias analysis**: in *type-safe* languages where pointers to *local* variables are not allowed (e.g. ML, Haskell, or Java), some useful optimization can be made
  - **context-sensitive vs context-insensitive**
  - **flow-sensitive vs flow-insensitive**
  - **field-sensitive vs field-insensitive**
  - **unification-based vs subset-based**

7.1.1 Basic Alias Analysis (Stateless AA Impl)

- File: `/lib/Analysis/BasicAliasAnalysis.cpp`
- Option: `-basicaa`
- basic alias analysis pass that implements identities (two different globals cannot alias, etc.) but does not perform stateful analysis
- determines whether or not separate memory references point to the same location of the memory

7.1.2 ScalarEvolution-based Alias Analysis

- Option: `-scev-aa`
- simple alias analysis implemented in terms of ScalarEvolution queries
- this differs from traditional **loop dependence analysis** in that it tests for dependencies within a single iteration of a loop, rather than dependencies between different iterations
- ScalarEvolution has a more complete understanding of pointer arithmetic than Basic Alias Analysis' collection ad-hoc analyses

7.1.3 Exhaustive Alias Analysis Precision Evaluator

- Option: `-aa-eval`
- this is a simple $O(n^2)$ alias analysis accuracy evaluator
- for each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function

7.1.4 Count Alias Analysis Query Reponses

- Option: `-count-aa`
- a pass which can be used to count how many alias queries are being made and how the alias analysis implementation being used responds

7.1.5 Alias Analysis Use Debugger

- Option: `-debug-aa`
- this simple pass checks alias analysis users to ensure that if they create a new value, they do not qeury Alias Analysis (AA) without informing it of the value
- it acts as a shim over any other Alias Analysis pass you want
- keeping track of every value in the program is expensive but this is a debugging pass

7.2 Call Graphs

7.2.1 Basic Call Graph Construction

- Option: `-basiccg`
-

7.3 Dominator Trees

7.3.1 Dominance Frontier Construction

- Option: `-domfrontier`
- domniator construction algorithm for finding forward dominator frontiers

7.3.2 Dominator Tree Construction

- File: `lib/VMCore/Dominator.cpp`
- Option: `-domtree`
- given an entry block S, block B_1 **dominates** block B_2 if every $S \rightarrow B_2$ path has to pass through B_1
  - entry block dominates all blocks
  - IDom (immediate dominator) of B :w

7.4 Loop Analysis

7.4.1 Natural Loop Information

- Option: `-loops`
- File: `Analysis/LoopInfo.h`
- identify natural loops and determine the loop depth of various nodes of the CFG
- calculates teh nesting structure of loops in a function: for each natural loop identified, this analysis identifies natural loops contained entirely within the loop and the basic blocks that make up the loop
- **natural loop**: has exactly one entry block (called **header**) and possible several back edges (called **latches**) leading to the header from inside the loop
- natural loops may be several loops that share the same header node

7.4.2 Scalar Evolution Analysis

- Option: `-scalar-evolution`
- used to analyze and categorize scalar expressions in loops
- specializes in recognizing generaal induction variables, representing them with the abstract and opaque `SCEV` class
- given this analysis, trip counts of loops and other important properties can be obtained
- Benefit: this analysis is primarily useful for *induction variable subtitution* and *strength reduction*

8 Transform Passes

8.1 Dead Code Elimination

8.1.1 Dead Argument Elimination

8.1.2 Dead Instruction Elimination

- Option: `-die`
- performs a single pass over the function, removing instructions that are obviously dead

8.1.3 Dead Store Elimination

- Option: `-dse`
- trival dead store elimination that only considers basic-block local redundant stores

8.1.4 Dead Global Elimination

- Option: `-globaldce`
- eliminates unreachable internal globals from the program
- uses an aggressive algorithm, searching out globals that are known to be alive; after it finds all of the globals which are needed, it deletes whatever is left over (this allows it to delete recursive chunks of program which are unreachable)

8.1.5 Aggressive Dead Code Elimination

8.2 Scalar Replacement of Aggregates (DT)

- File: `lib/Transform/Scalar/ScalarReplAggregates.cpp`
- Option: `scalarrepl`
- breaks up alloca instructions of aggregate type (structure or array) into individual instructions for each member (if possible); then, if possible, it transforms the individual alloca instructions into clean scalar SSA form; combines a simple SRoA algorithm with the Mem2Reg algorithm because often interact especially for C++ programs; as such iterating between SRoA, then Mem2Reg until we run of the things to promote works well
- Example: given a structure-type variable

```
typedef struct {
  int x;
  int y;
} point_t;
point_t p;
if (p->x > p->y) ...;
```

  we transform this into

```
int px = p->x;
int py = p->y;
if (px > py) ...;
```

- Caveat: applicable only if target component and the aggregate are not aliased
- Benefit: enables scalar-based optimizations and facilitates *register allocation* (since components are smaller than the whole and more fittable into registers) and *constant/copy propagation*

9 Utility Passes

9.1 Preliminary Module Verification

- File: `lib/VMCore/Verifier.cpp`
- Option: `-preverify`
- checks if `Function` object contains at least one `BasicBlock` object in the iterator
  - `Function` consists of `BasicBlocks`

9.2 Module Verification

- File: `lib/VMCore/Verifier.cpp`
- Option: `-verify`
- used for validity checking of input to the system; this does not provide full "Java style" security and verification; instead, it just tries to ensure that code is well-formed, like those listed below

1. both of a binary operator's operands are of the same type
2. verify that the indices of mem access instructions match other operands
3. verify that arithmetic and other things are only performed on first-class types
4. verify that shifts and logicals only happen on integrals f.e.
5. all of the constants in a switch statements are of the correct type
6. the code is in valid SSA form
7. it should be illgegal to put a lable into any other type (like a structure) or to return one (except constant arrays)
8. only ϕ-nodes can be self referential: `add i32 %0, %0 ; <int>:0` is bad
9. ϕ-nodes must have an entry for each predecessor, with no extras
10. ϕ-nodes must be the first thing in a basic block, all grouped together
11. ϕ-nodes must have at least one entry
12. all basic blocks should only end with terminator insts, not contain them
13. the entry node to a function must not have predecessors
14. all Instructions must be embedded into a basic block
15. functions cannot take a void-typed parameter
16. verify that a function's argument list agrees with it's declared type
17. it is illegal to specify a name for a void value
18. it is illegal to have a internal global value with no initializer
19. it is illegal to have a ret instruction that returns a value that does not agree with the function return value type
20. function call argument types match the function prototype
21. all other things that are tested by asserts spread about the code

9.3 Loop Optimization

10 Code Generation

10.1 Optimize for Code Generation

10.2 Insert Stack Protectors

- If stack smashing protection required (based on heuristic which considers the size of buffer)
- only when stackprotec function attribute is given
- NOTE: wiki **Buffer overflow protection** for details

10.3 Machine Function Analysis

10.4 X86 DAG→DAG Instruction Selection

10.5 X86 PIC Global Base Reg Initialization

10.6 Expand ISel Pseudo-instructions

10.7 Optimize machine instruction PHIs

10.8 Local Stack Slot Allocation

10.9 Remove dead machine instructions

10.10 MachineDominator Tree Construction

10.11 Machine Natural Loop Construction

10.12 Machine Instruction LICM

10.13 Machine Common Subexpression Elimination

10.14 Machine code sinking

10.15 Peephole Optimizations

10.16 Tail Duplication

10.17 X86 Maximal Stack Alignment Check

10.18 Remove unreachable machine basic blocks

10.19 Live Variable Analysis

10.20 MachineDominator Tree Construction

10.21 Machine Natural Loop Construction

10.22 Eliminate ϕ-nodes for register allocation

10.23 Two-Address instruction pass

10.24 Process Implicit Definitions

10.25 Slot index numbering

10.26 Live Interval Analysis

10.27 Debug Variable Analysis

10.28 Simple Register Coalescing

10.29 Calculate spill weights

10.30 Live Stack Slot Analysis

10.31 Virtual Register Map

10.32 Linear Scan Register Allocator

10.33 Stack Slot Coloring

10.34 Machine Instruction LICM

10.35 Bundle Machine CFG Edges

10.36 X86 FP Stackifier

10.37 Subregister lowering instruction pass

10.38 Prolog/Epilog Insertion & Frame Finalization

10.39 Post RA top-down list latency scheduler

10.40 Control Flow Optimizer

10.41 Tail Duplication

10.42 Analyze Machine Code For Garbage Collection

10.43 MachineDominator Tree Construction

10.44 Machine Natural Loop Construction

- superclass of **Instruction** and **Function** classes
- each **Value** has a **Type**
- each **Value** has a "Use List" which keeps track of which other Values are using this Value

11.1.2 Use

- **Use** represents an operand of **Instruction** or some other **User** instance which refers to a **Value**

11.1.3 User

11.1.4 LLVMContext

- Owns and manages the core "global" data of LLVM (e.g. type and constant uniquing tables)
- No locking guarantees; so, be careful to have one context per thread

11.1.5 Module

- a Module instance is used to store all the information related to an LLVM module
- an LLVM module is a top-level container of all other LLVM IR objects
- each module directly contains
  - a list of **global varaibles**
  - a list of **funtions**
  - a list of **libraries** that this module depends on
  - **symbol table**
  - various data about target's characteristics

11.2 LLVM Support Classes

11.2.1 ManagedStatic

- Allows lazy on-demand creation of global statics.
- good for reducing startup time
- when user uses the given static object `C &operator*()` and `C *operator->()`, we construct the object at that time.

11.3 MemoryBuffer

- this interface provides simple read-only access to a block of memory, and provides simple methods for reading files and standard input into a memory buffer. In addition to basic access to the characters in the file, this interface guarantees you can read one character past the end of the file, and that this character will read as NULL
- The NULL guarantee is needed to support an optimization – it's intended to be more efficient for clients which are reading all the data to stop reading when they encounter a NULL than to continually check the file position to see if it has reached the end of the file

11.3.1 Mutex, SmartMutex

- supports methods: **acquire**, **release**, **tryacquire**
- **SmartMutex** is a mutex with a compile-time constant parameter that indicates whether this mutex should become a NO-OP when we're not running in multithreaded mode

11.4 MemoryFence

-

11.5 ThreadLocal

-

11.6 OwingPtr

- **OwningPtr** mimics a builtin pointer except that it guarantees deletion of the object pointed to, either on destruction of the **OwningPtr** or via an explicit `reset()`. Once created, ownership of the pointee object can be taken away from OwningPtr by using the `take()` method.

11.7 STL Utility Classes

11.7.1 auto_ptr

- provides some basic **RAII** (Resource Acquisition is Initialization) for C++ raw pointers.
  - RAII is essential for writing exception-safe C++ programs