人間でもわかる LLVMバックエンド入門

風薬(@kazegusuri)

x86/x64最適化勉強会#5

アジェンダ

- 自己紹介
- 小ネタ
- LLVMについて
- LLVMバックエンド
- 最適化のポイント

自己紹介

- 風薬(@kazegusuri)
- サークル MotiPizzaで活動
  - http://motipizza.com/
  - LLVM本
  - 冬はClang本(予定)
- 仕事はWeb系の開発運用
  - LLVMは全く関係無し

- 勉強会での発表はこれが初めて

宣伝！

- LLVM本出してます！
  - きつねさんでもわかるLLVM
  - 達人出版会様より販売
  - http://tatsu-zine.com/books/llvm

小ネタ

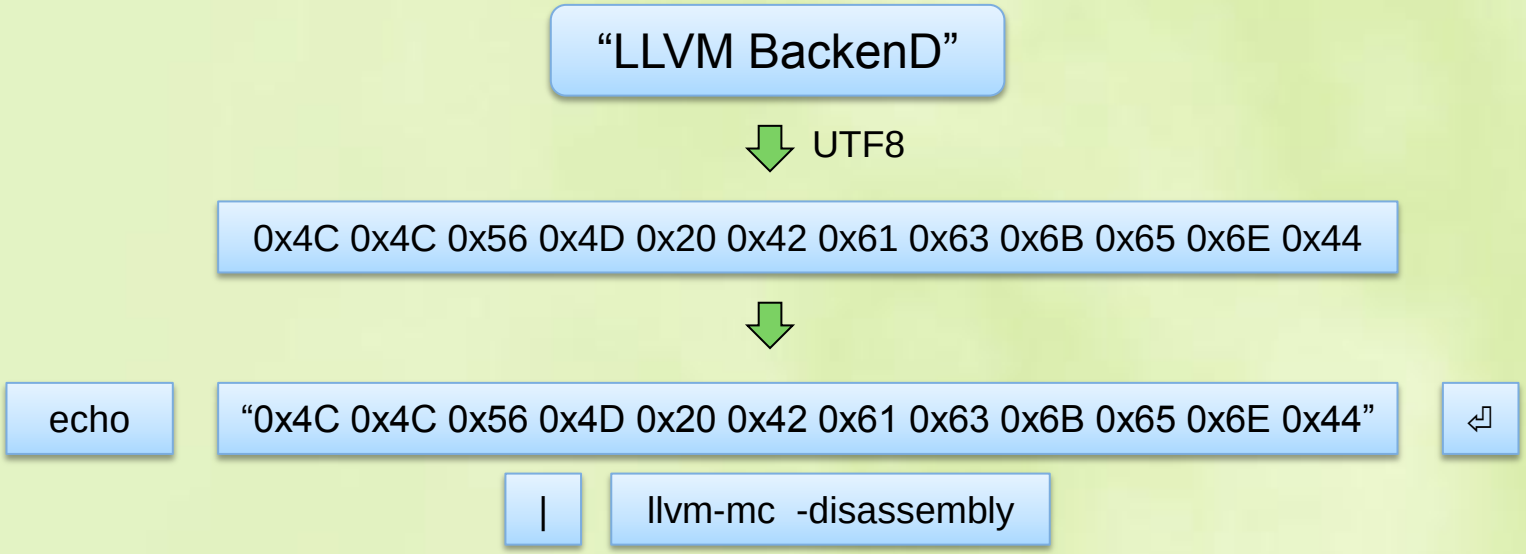
“LLVM BackenD”

小ネタ

“LLVM BackenD”

0x4C 0x4C 0x56 0x4D 0x20 0x42 0x61 0x63 0x6B 0x65 0x6E 0x44

小ネタ


“LLVM BackenD”
UTF8
0x4C 0x4C 0x56 0x4D 0x20 0x42 0x61 0x63 0x6B 0x65 0x6E 0x44
echo
“0x4C 0x4C 0x56 0x4D 0x20 0x42 0x61 0x63 0x6B 0x65 0x6E 0x44”
⏎
|
llvm-mc -disassembly

小ネタ

“LLVM BackenD”

⇩ UTF8

0x4C 0x4C 0x56 0x4D 0x20 0x42 0x61 0x63 0x6B 0x65 0x6E 0x44

⇩

echo “0x4C 0x4C 0x56 0x4D 0x20 0x42 0x61 0x63 0x6B 0x65 0x6E 0x44” ⏎

| llvm-mc -disassembly

⇩

```
decl    %esp
decl    %esp
pushl   %esi
decl    %ebp
andb    %al, 97(%edx)
arpl    %bp, 101(%ebx)
outsb
incl    %esp
```

＿人人人人人人人人人＿
＞　特に意味は無い　＜
￣Y^Y^Y^Y^Y^Y^Y^Y￣

こんなこともできます…

アジェンダ

- 自己紹介
- 小ネタ
- **LLVMについて**
- LLVMバックエンド
- 最適化のポイント

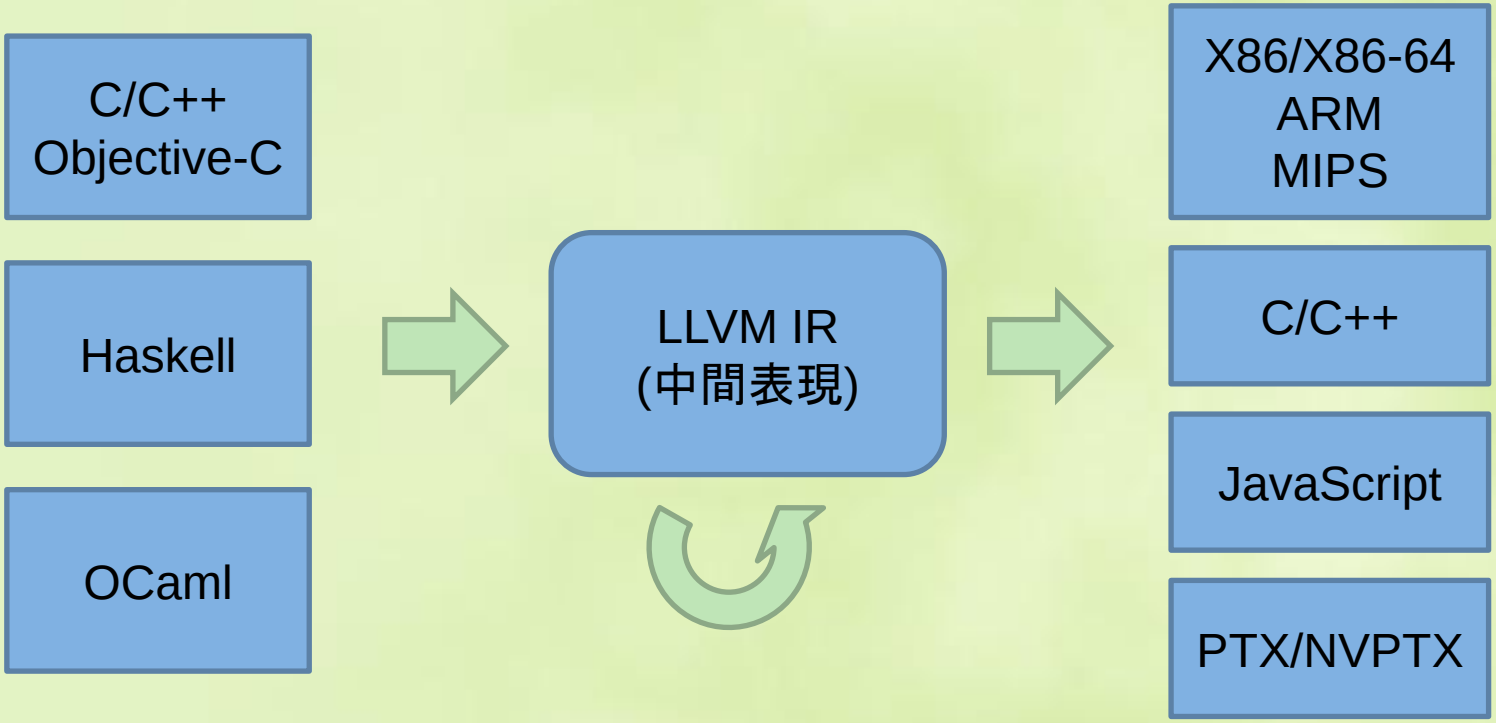
LLVMとは

- コンパイラ基盤
  - オプティマイザとコード生成
  - 中間表現(LLVM IR)を入力とする
- LLVMプロジェクトの1つ
  - LLVM Core
  - 単にLLVMというとLLVM Coreを指すことが多い
- サブプロジェクト
  - Clang, LLDB, compiler-rt, libc++, vmkit, polly…

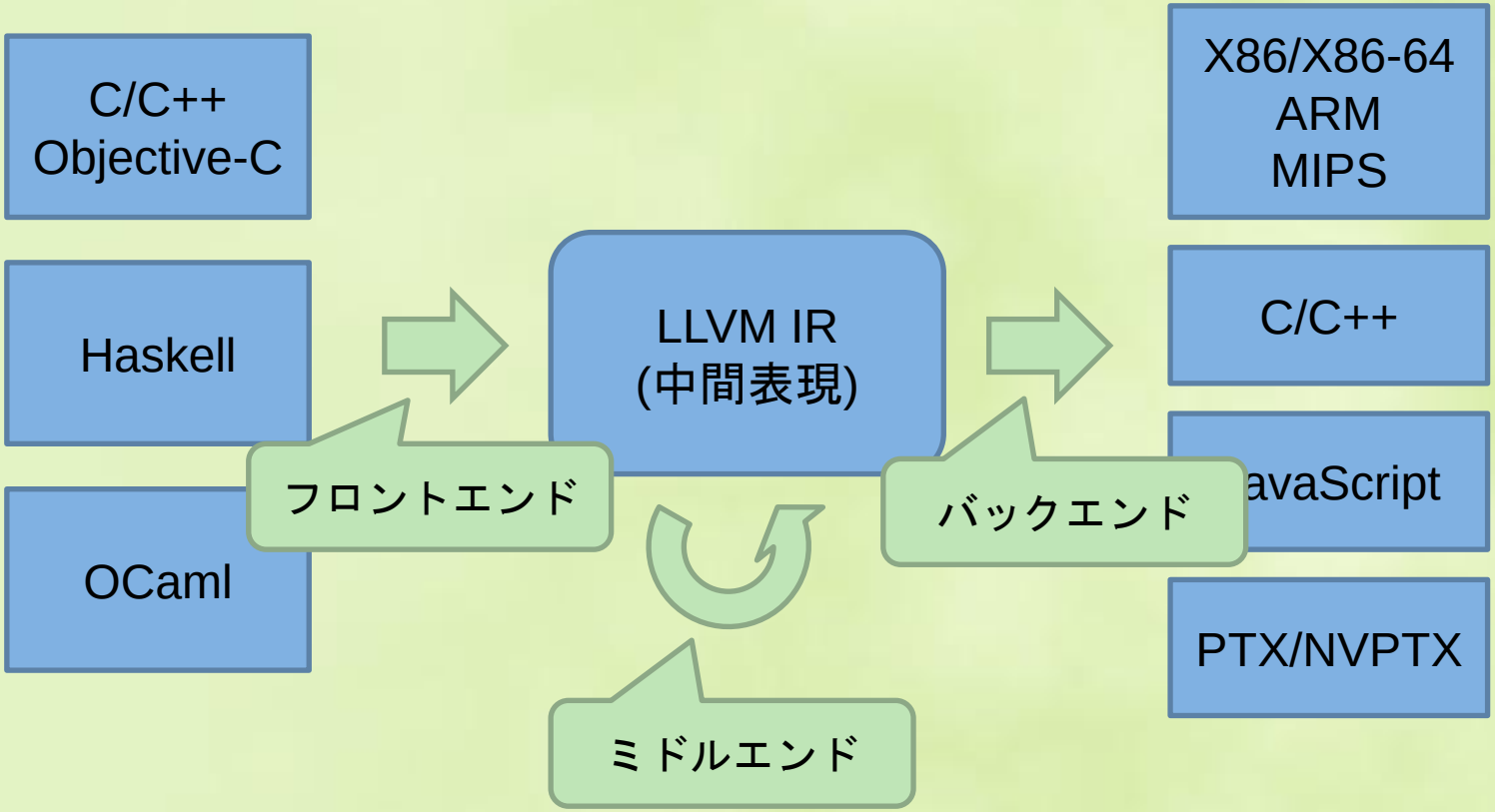
なぜ注目されているのか

- BSDライクの制限の緩いライセンス
  - GPLが使えない企業など
  - FreeBSDのデフォルトコンパイラ
- モジュール化による再利用性
  - 一部分にフォーカスできて再発明が不要
  - ライブラリのように外からも叩ける
  - 実装がわかりやすい(GCCと比較して)

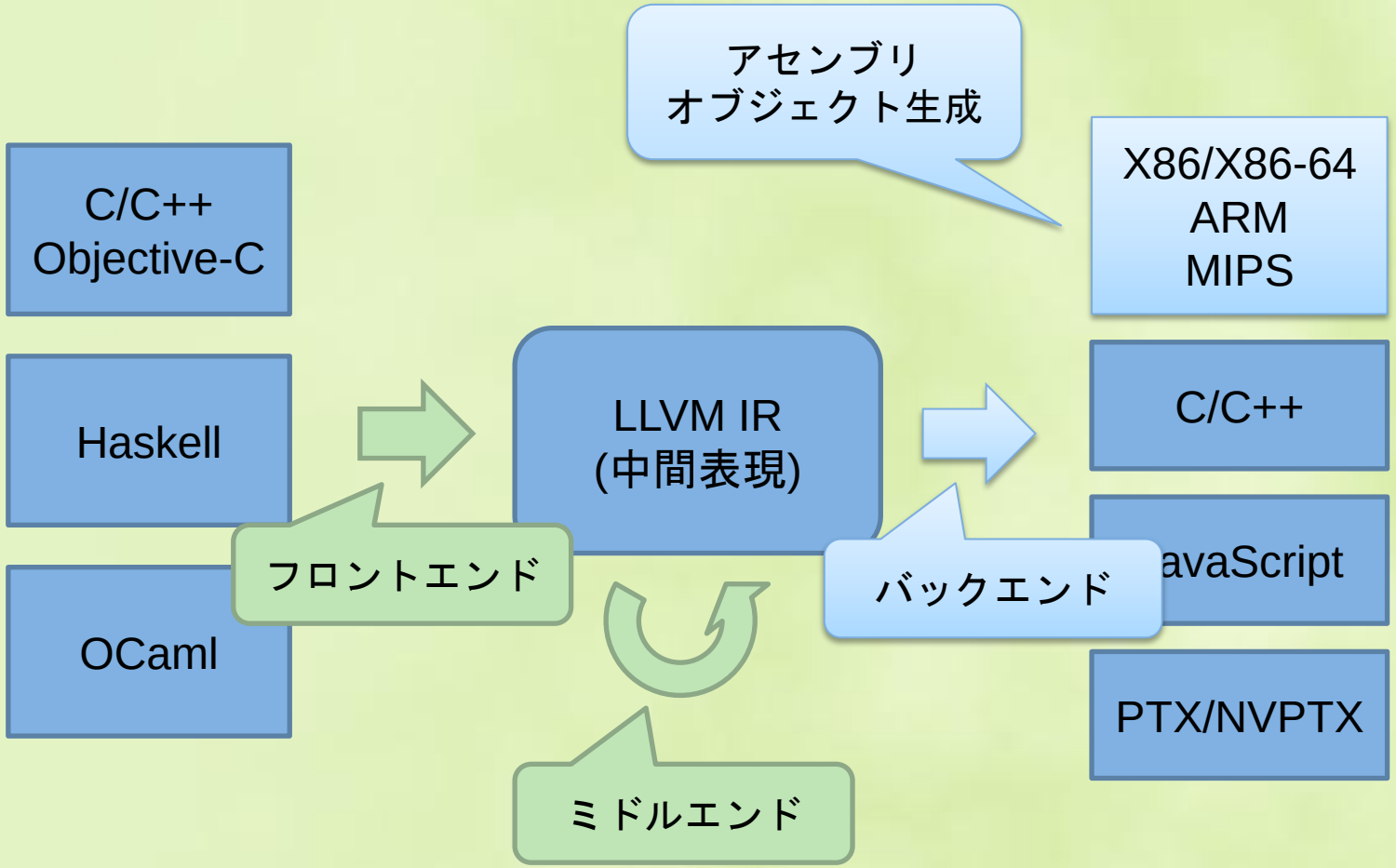
LLVMの流れ


C/C++
Objective-C
Haskell
OCaml
LLVM IR
(中間表現)
X86/X86-64
ARM
MIPS
C/C++
JavaScript
PTX/NVPTX

LLVMの流れ


C/C++
Objective-C
Haskell
OCaml
フロントエンド
LLVM IR
(中間表現)
ミドルエンド
バックエンド
X86/X86-64
ARM
MIPS
C/C++
avaScript
PTX/NVPTX

LLVMの流れ

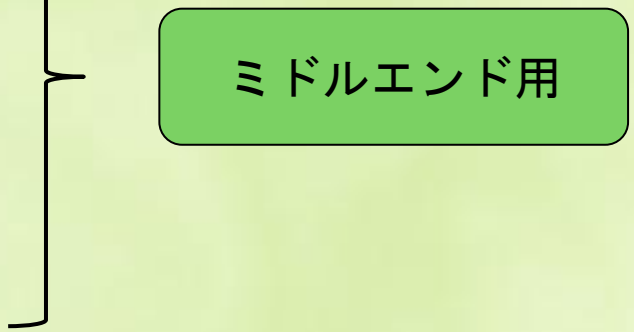

アセンブリ
オブジェクト生成
X86/X86-64
ARM
MIPS
C/C++
Objective-C
Haskell
OCaml
LLVM IR
(中間表現)
C/C++
avaScript
PTX/NVPTX
フロントエンド
バックエンド
ミドルエンド

Passの概念

- LLVMでの処理は全てPassで行われる
  - 解析・最適化・コード生成
- 利用者は任意のPassを組み合わせて使う
  - llc などはそれらのデフォルトの組み合わせ
  - opt で特定のPassを適用することもできる

Passの種類

- ImmutablePass
- ModulePass
- FunctionPass
- LoopPass
- RegionPass
- BasicBlockPass



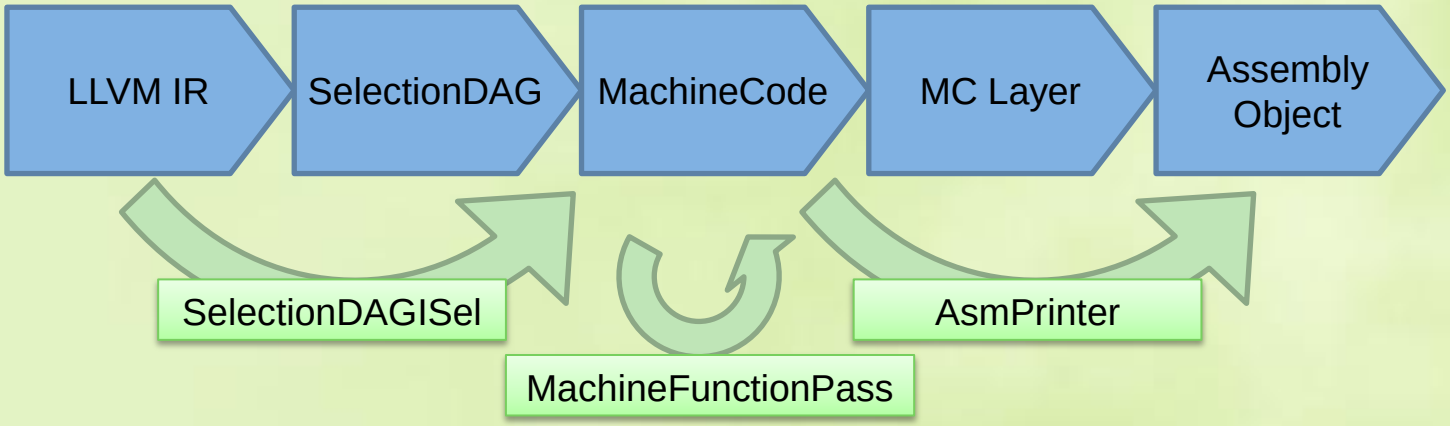
- **MachineFunctionPass**
  - バックエンド用Pass
  - 実際はFunctionPass

アジェンダ

- 自己紹介
- 小ネタ
- LLVMについて
- **LLVMバックエンド**
- 最適化のポイント

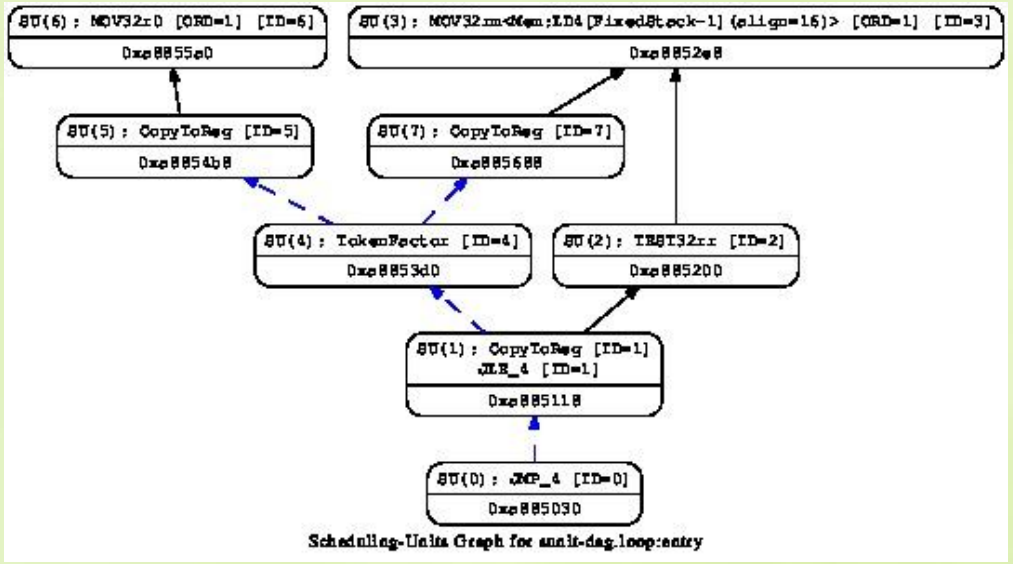
バックエンドの流れ

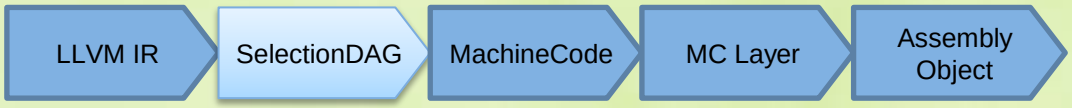
- LLVM IRを入力として何度か形式を変える
  - 形式の変更のことをLoweringと呼ぶ
  - 処理はMachineFunctionPassで行われる



SelectionDAGISelパス

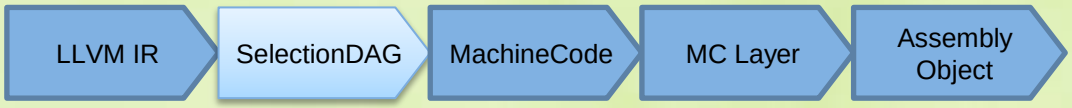
- LLVM IRをDAG(有向非巡回グラフ)に変換
- ノードの置き換えや共通部分削除など
- 最終的にMachineCodeを生成





SelectionDAGISelパス

- Lowering
  - LLVM IRからSDNode(illegal)への変換
- Combine
  - パターンマッチによる最適化
- Legalize
  - SDNode(illegal)からSDNode(legal)
- Select
  - SDNodeからMachineCodeへの変換
- Schedule
  - 命令のスケジューリング



MachineCode

- より機械語に近い形式
  - LLVM IRは機械語と比較すると抽象度が高い
  - 実際の命令や物理レジスタを持つ
- フェーズによって形式が変わる
  - 仮想レジスタ、PHIノード有、SSA形式
  - 物理レジスタ、PHIノード無、Non-SSA形式
- 構造はLLVM IRと似ている
  - BasicBlock, Function, Instruction, Operand

MachineSSAOptimization

- SSA形式でのターゲット依存の最適化
  - Stack Slot Coloring
  - Local Stack Slot Allocation
  - Peephole Optimization
  - 他にも…

Non-SSA形式でのパス

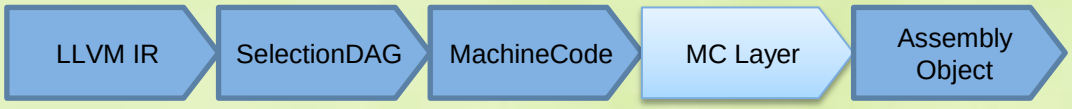
- Eliminate PHI nodes
  - Φノードをここでやっと削除
  - Non-SSA形式になる
- Register Allocation
  - 仮想レジスタから物理レジスタに
- Prologue/Epilogue Insertion
  - 関数呼び出しに関するターゲット依存の処理

AsmPrinterパス

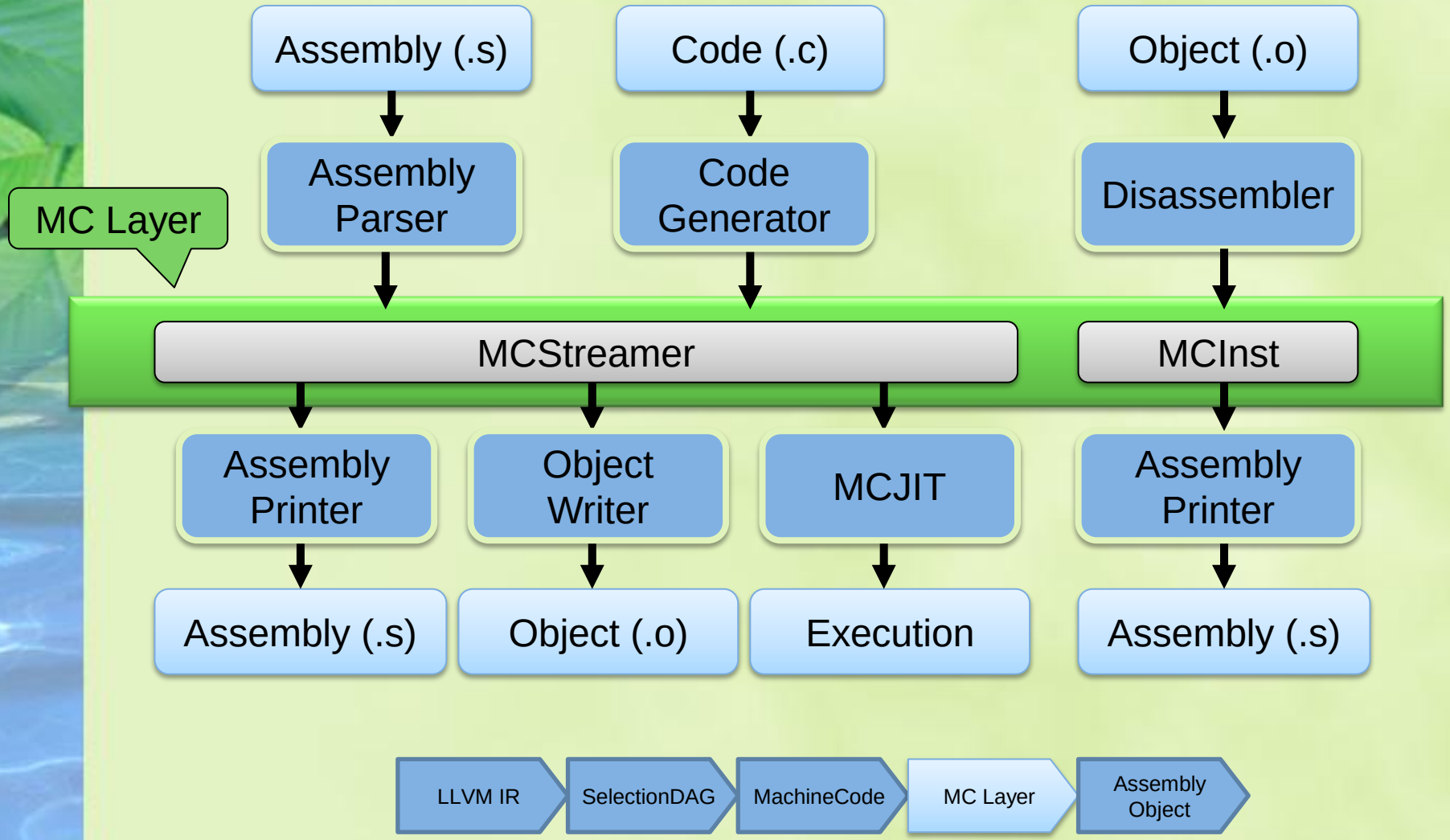
- コード生成
  - アセンブリもオブジェクトも共通処理
    - MC Layerで抽象化されている
- AsmPrinterの役割
  - MachineCodeからMCInstへのLowering
  - MC Layerの操作

MC Layer(MCInst)

- コード生成などを抽象化するレイヤ
  - アセンブリ, オブジェクト, JIT
  - 処理が共通化
- MCInst
  - MC Layerで扱う命令形式
  - 関数などの構造が無くフラット

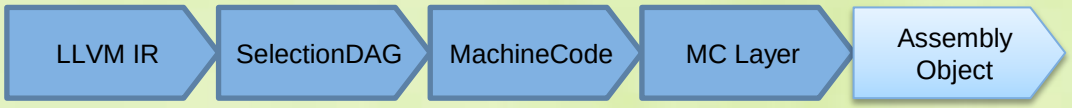


MC Layer


Assembly (.s)
Code (.c)
Object (.o)
Assembly Parser
Code Generator
Disassembler
MC Layer
MCStreamer
MCInst
Assembly Printer
Object Writer
MCJIT
Assembly Printer
Assembly (.s)
Object (.o)
Execution
Assembly (.s)
LLVM IR
SelectionDAG
MachineCode
MC Layer
Assembly Object

コード生成後

- 各種ツールもある
  - llvm-objdump (.o => .s)
  - clang (.o => a.out)
    - リンカ代わり
  - llvm-linker (.ll => .ll)
  - llc (.ll => .s or .o)



アジェンダ

- 自己紹介
- 小ネタ
- LLVMについて
- LLVMバックエンド
- **最適化のポイント**

最適化のポイント

- 最適化可能な場所が多い
  - 粒度が異なる
  - どこでやるか？
    - フロントエンド
    - ミドルエンド
    - バックエンド

最適化のポイント

- フロントエンド
  - LLVM IRに落とすところも重要
  - 元のソースコードの意味を活かせる
- ミドルエンド
  - 多くの情報を失っているがまだ大幅な最適化ができる
    - メタデータで情報を残すこともできる
  - 自動並列化(祖粒度,細粒度)

バックエンドでの最適化

- 機械命令レベルの最適化
  - 1命令を減らす最適化はバックエンドでのみ
- LLVM IRと同じ意味になる命令へ置き換え
  - パターンマッチによる置き換え(SelectionDAG)
  - 大規模な最適化はできない(?)
    - 関数を超えた最適化はできないかも
- 置き換え以上のことをやるなら独自Pass
  - 好きなタイミングでPass実行もできる
    - SSA or Non-SSA
  - 自分はやったことないですが…

LLVMとの戦いはまだまだ続く…

fin.