LLVM - Another Toolchain Platform

杨勇勇 (yongyong.yang@ia.ac.cn)

自动化所 · 集成电路中心

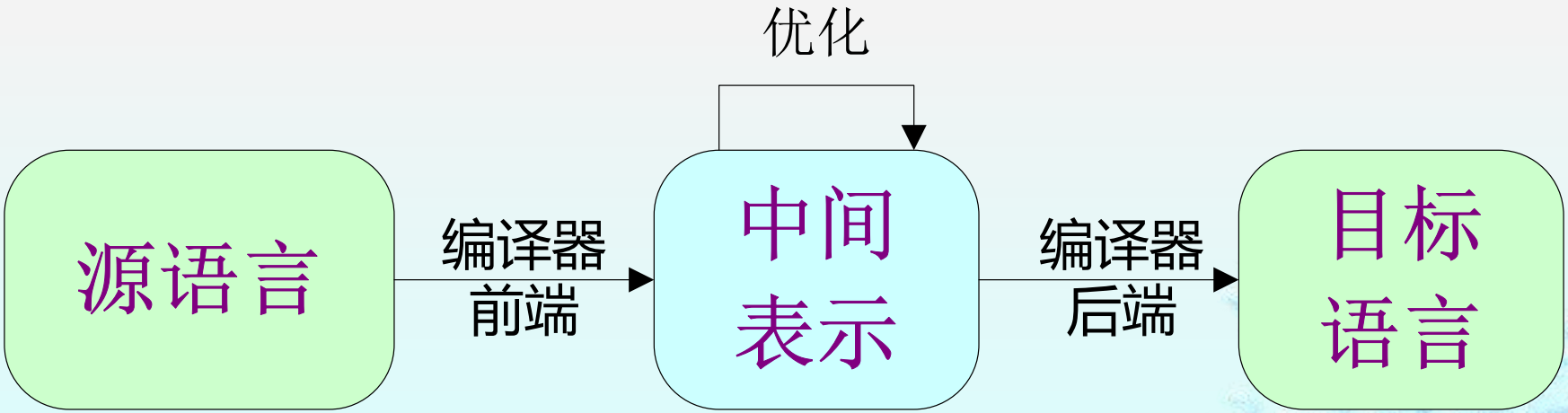
内容简介

- LLVM的后端框架及代码结构
- 如何实现一个后端
- 整合汇编器和反汇编器
- 链接器和调试器

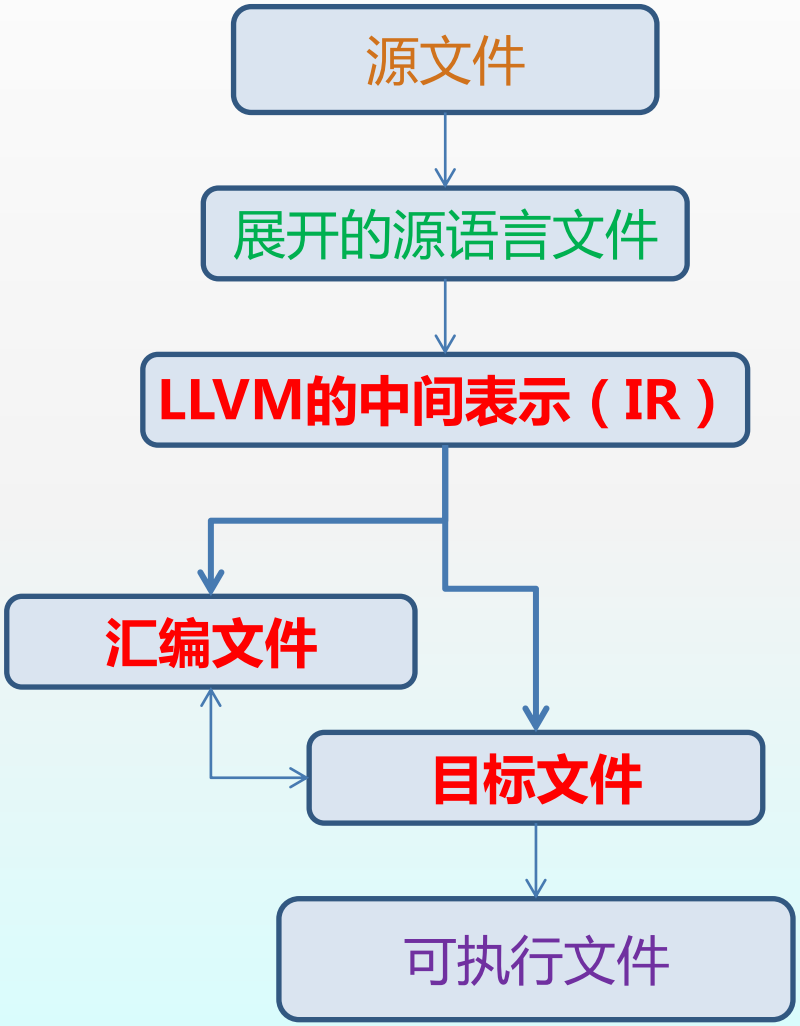
- LLVM的后端框架及代码结构
- 如何实现一个后端
- 整合汇编器和反汇编器
- 链接器和调试器

基本编译结构

- 源语言：C, C++, Object-C, …
- 中间表示（Immediate Representation）
- 目标语言：汇编文本，二进制目标文件（elf, …）



基本编译流程 (1)



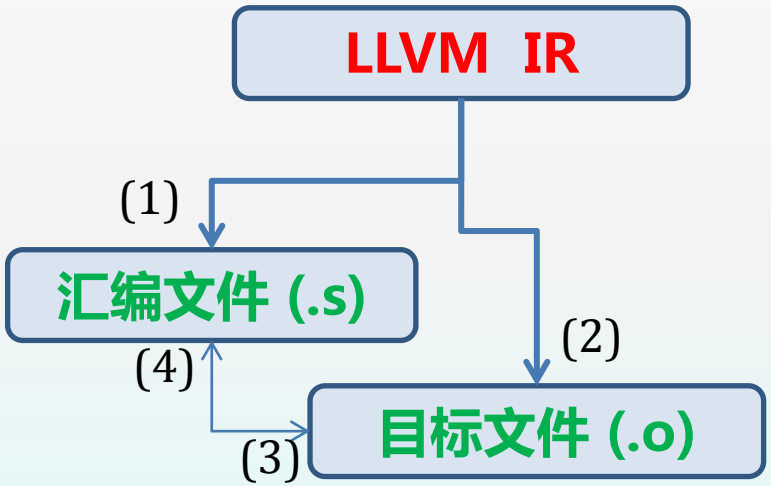
基本编译流程 (2)

操作名称	对应选项 (与gcc一致)	输出结果
预处理	-E	展开的C文件
编译为汇编文本	-S (大写)	汇编文件
编译为目标文件（未链接）	-c (小写)	二进制文件，elf, coff等
编译为可执行文件（链接）	无选项	可执行文件，elf, coff等

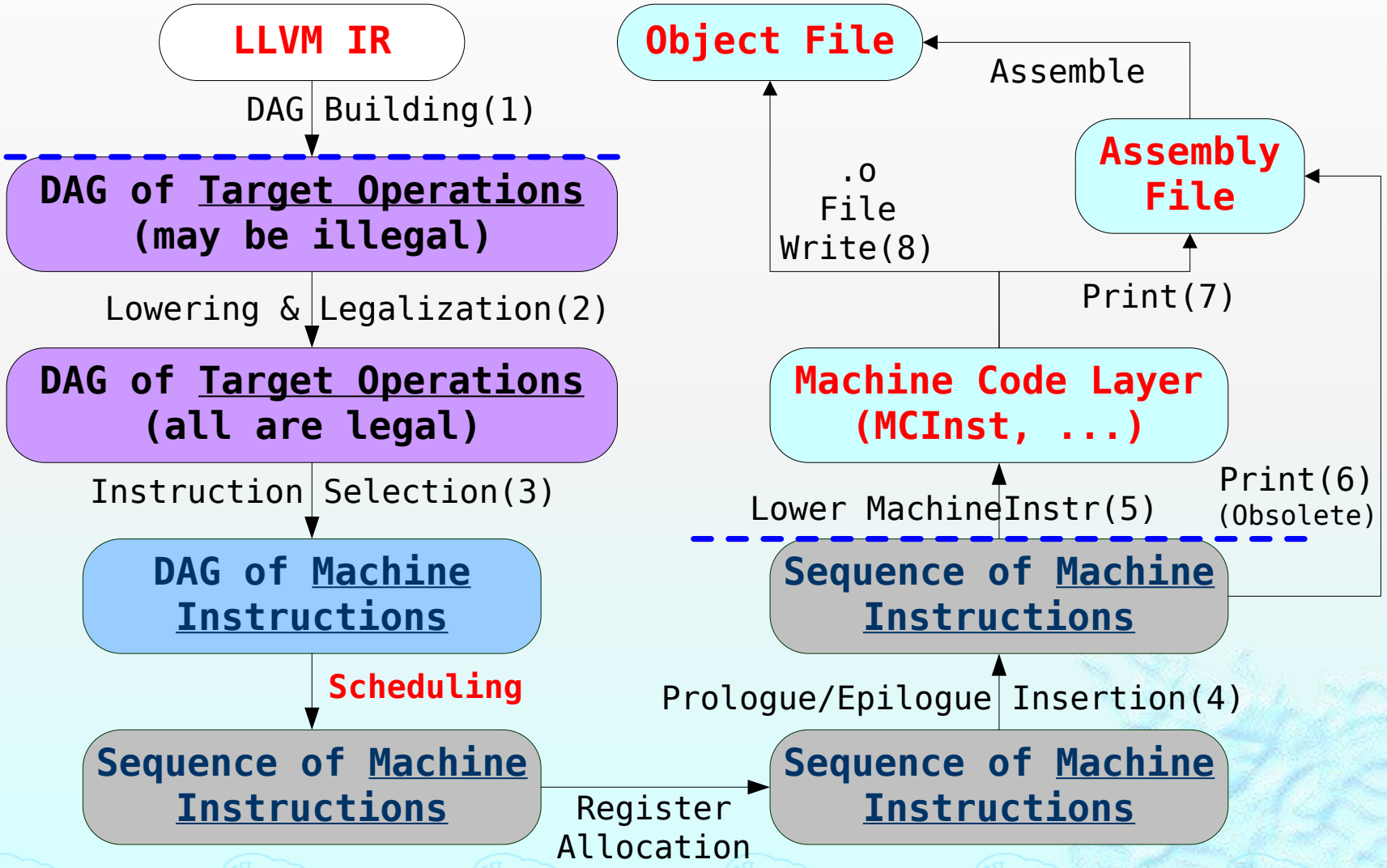
LLVM的命令序列：

```
clang -S -emit-llvm hello.c -o hello.ll    // 前端；hell.ll : LLVM的中间表示
llc -march=XX -mcpu=XY hello.ll -o hello.s // 后端；hello.s : 目标汇编语言
```

LLVM的后端部分



后端代码生成流程


LLVM IR
DAG Building(1)
DAG of Target Operations (may be illegal)
Lowering & Legalization(2)
DAG of Target Operations (all are legal)
Instruction Selection(3)
DAG of Machine Instructions
Scheduling
Sequence of Machine Instructions
Register Allocation
Sequence of Machine Instructions
Prologue/Epilogue Insertion(4)
Sequence of Machine Instructions
Lower MachineInstr(5)
Machine Code Layer (MCInst, ...)
Print(6) (Obsolete)
Print(7)
Assembly File
Assemble
Object File
.o File Write(8)

划分代码生成流程

根据上页的流程图可以将后端划为**两大部分**:

- **第一部分**：以蓝虚线分隔。
  将IR编译成MachineInstr，由TargetMachine类中的接口控制
- **第二部分**：第2条蓝虚线分隔以后的流程。
  以Machine Code Layer为核心，整合汇编与反汇编等与二进制文件相关的功能

描述后端第一部分 (1)

操作	描述
DAG Building (1) (或称为Initialization)	从LLVM的IR表示构建SelectionDAG，用于后续的代码选择
Lowering & Legalization (2)	在构建的SelectionDAG中，将target不支持的操作/数据类型转化为支持的操作/数据类型
SelectionDAG-based Instruction Selection (3)	Pattern-matching instruction selection
Prologue/Epilogue Insertion (4)	插入建立/撤销函数调用栈的代码

描述后端第一部分(2)

操作	类	所在的文件
DAG Building (1) (或称Initialization)	SelectionDAGBuilder (platform built-in)	lib/CodeGen/SelectionDAG/SelectionDAGBuilder.[h|cpp]
Lowering & Legalization (2)	XXTargetLowering (基类TargetLowering)	lib/Target/XX/XXISelLowering.[h|cpp] (基类include/llvm/Target/TargetLowering.h)
SelectionDAG-based Instruction Selection (3)	XXDAGToDAGISel (基类SelectionDAGISel)	lib/Target/XX/XXISelDAGToDAG.[h|cpp] (基类 include/llvm/CodeGen/SelectionDAGISel.h)
Prologue/Epilogue Insertion (4)	XXFrameLowering (基类 TargetFrameLowering)	lib/Target/XX/XXFrameLowering.[h|cpp] (基类 include/llvm/Target/TargetFrameLowering.h)

第一部分的基本数据结构（1）

LLVM IR

- 一种high-level的中间语言
- 包含类型信息
- 丰富完善的接口支持
- 多种优化技术实现
- clang前端对C/C++/ObjC的良好支持
- 学术研究、编译器构建的良好平台

基本数据结构（2）：DAG

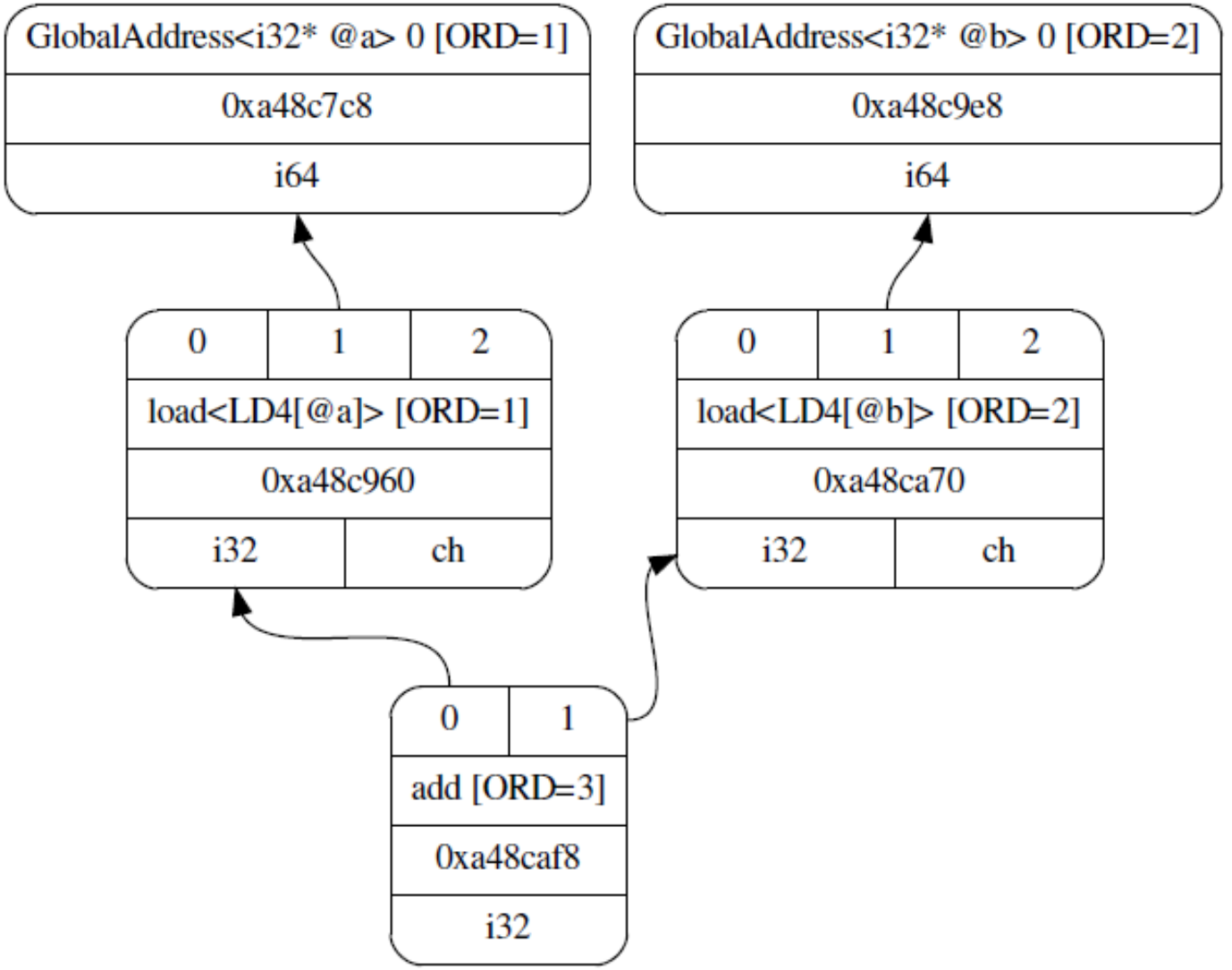
- DAG的承载体：**SelectionDAG**类
  参看文件include/llvm/CodeGen/SelectionDAG.h

- 构成DAG的元素：**SDValue**和**SDNode**
  参看文件include/llvm/CodeGen/SelectionDAGNodes.h

  SDNode: DAG中的节点，表示target operation
  如ISD::ADD, 参看文件include/llvm/CodeGen/ISDOpcodes.h中原生支持的操作。用户可以在此基础上自定义target operation。
  SDValue: 从SDNode到SDNode的一条单向边，表示数据的流动

DAG示例

```
{
    ……
    extern int *a,
    extern int *b;
    *a + *b;
    ……
}
```



操作SelectionDAG的接口

- SelectionDAG的若干基本方法：

```
SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT, SDValue N);
SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT, SDValue N1,
                SDValue N2);
SDValue getRegister(unsigned Reg, EVT VT);
SDValue getConstant(const ConstantInt &Val, EVT VT, bool
                    isTarget = false);
……
```

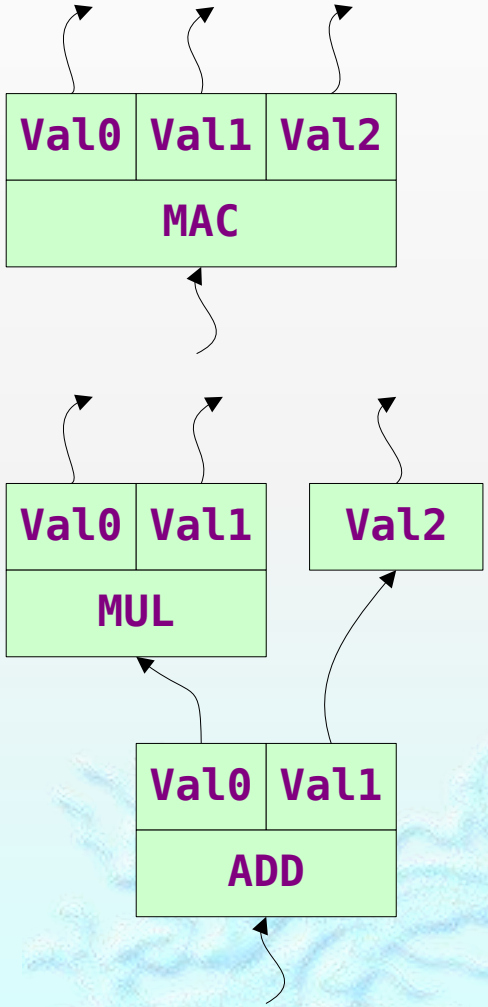
一般形式：

```
SDValue getXXXX(……); // 根据指定的条件在DAG中查找或者创建节点
```

改写DAG：拆分乘累加指令

```
SDValue LowerMAC(SDValue Op,
                 SelectionDAG &DAG)
{
  DebugLoc dl = Op.getDebugLoc();
  SDValue mul = DAG.getNode(ISD::MUL,
                    dl, MVT::i32,
                    Op.getOperand(0),
                    Op.getOperand(1));

  SDValue value = DAG.getNode(ISD::ADD,
                    dl, MVT::i32,
                    mul, Op.getOperand(2));
  return value;
}
```

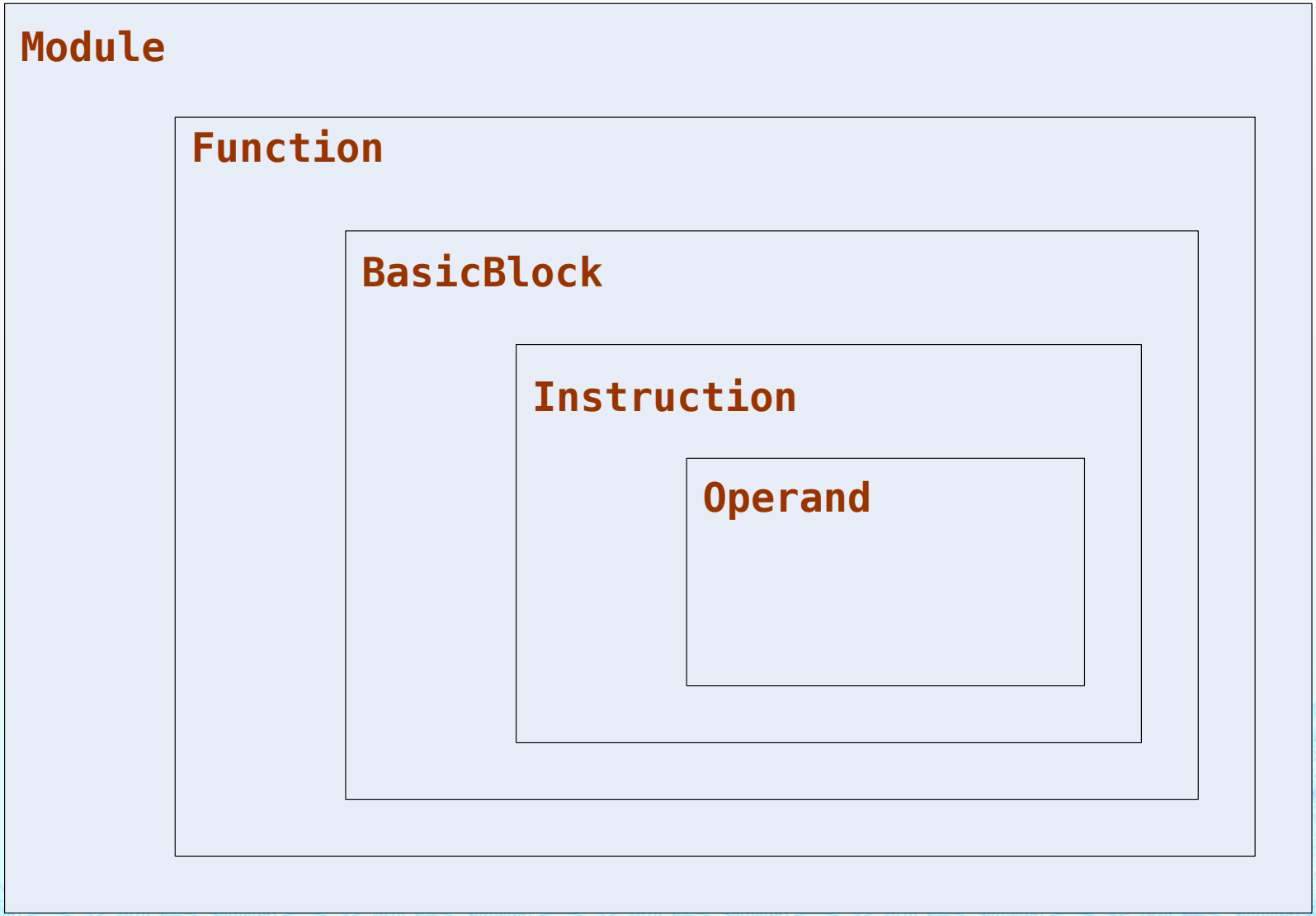


基本数据结构 (3)：MachiInstr

- **MachiInstr**类：Machine Instruction的载体，大体对应于tablegen中定义的指令，记录opcode + operand(s) + **上下文信息**：相邻的其它指令、所属基本块、所属函数、所属编译单元、……
- tablegen中的例子：

```
def AddI32 : Instruction {
    let OutOperandList = (outs I32Reg:$d);
    let InOperandList = (ins I32Reg:$s1, $s2);
    let AsmString = “add $d, $s1, $s2”;
    let Pattern = [(set I32Reg:$d, (add I32Reg:$s1,
    I32Reg:$s2))]
}
```

MachineInstr的从属关系



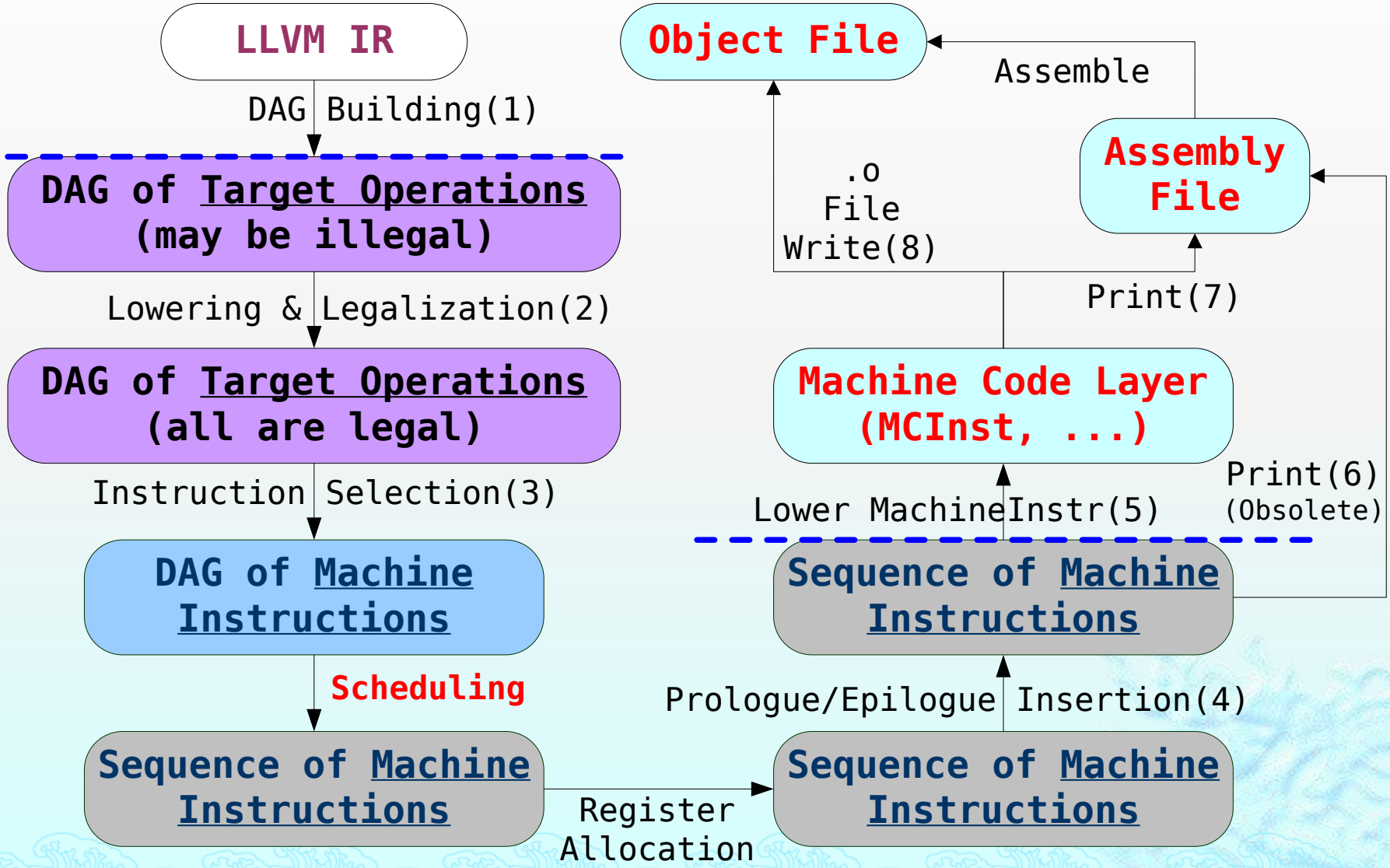
基本数据结构 (4)：MCInst

- **MCInst**类：MC层的承载体，MachiInstr类的极简化版，只记录opcode与operand(s)，其值可从MachiInstr的实例中抽取；不包含指令的上下文信息

- **将machine instruction转化至MC层的MCInst**

```
void XXLowerMachiInstrToMCInst(
          const MachineInstr& MI,
          MCInst& MCI,
          AsmPrinter& AP);
```

后端第二部分


LLVM IR
DAG Building(1)
DAG of Target Operations (may be illegal)
Lowering & Legalization(2)
DAG of Target Operations (all are legal)
Instruction Selection(3)
DAG of Machine Instructions
Scheduling
Sequence of Machine Instructions
Register Allocation
Sequence of Machine Instructions
Prologue/Epilogue Insertion(4)
Sequence of Machine Instructions
Lower MachineInstr(5)
Machine Code Layer (MCInst, ...)
Print(6) (Obsolete)
Print(7)
Assembly File
Assemble
.o File Write(8)
Object File

后端第二部分 (1)

操作	描述
Lower MachineInstr(5)	将MachineInstr实例转化为MCInst实例
Print(6)	将MachineInstr形式的编译结果输出为汇编文件。由于MC layer的加入，这种方式逐渐被舍弃，但实现代码仍然保留着
Print(7)	将MCInst形式的编译结果输出为汇编文件
.o File Write(8)	将MCInst形式的编译结果直接输出为二进制格式文件，如elf, coff, mach-o, ...

后端第二部分 (2)

操作	类	所在的文件
Lower MachineInstr (5)	XXMCInstLower (无需基类)	lib/Target/XX/XXMCInstLower.[h|cpp]
Print(6)	XXAsmPrinter (基类AsmPrinter)	lib/Target/XX/XXAsmPrinter.[h|cpp] (基类include/llvm/CodeGen/AsmPrinter.h)
Print(7)	XXInstPrinter (基类MCInstPrinter)	lib/Target/XX/InstPrinter/XXInstPrinter.[h|cpp] (基类include/llvm/MC/MCInstPrinter.h)
.o File Write(8)	XXMCCodeEmitter (基类MCCodeEmitter) XXAsmBackend (基类MCAsmBackend) XXELFObjectWriter (基类 MCELFObjectTargetWriter)	lib/Target/XX/MCTargetDesc/XXMCCodeEmitter.[h|cpp] (基类include/llvm/MC/MCCodeEmitter.h) lib/Target/XX/MCTargetDesc/XXAsmBackend.[h|cpp] (基类include/llvm/MC/MCAsmBackend.h) lib/Target/XX/MCTargetDesc/XXELFObjectWriter.[h|cpp] (基类include/llvm/MC/MCELFObjectWriter.h)

关于MC Layer

- 参考资料
  1. http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html
  2. http://www.llvm.org/devmtg/2010-11/Dunbar-MC.pdf
  3. http://llvm.org/devmtg/2011-11/Grosbach_Anderson_LLVMMC.pdf

- 用于instruction-set level工具中：
  assembly, disassembly, object file formats, …
- 主要数据结构：
  MCOperand, MCSymbol, MCExpr, MCInst, MCSection, …

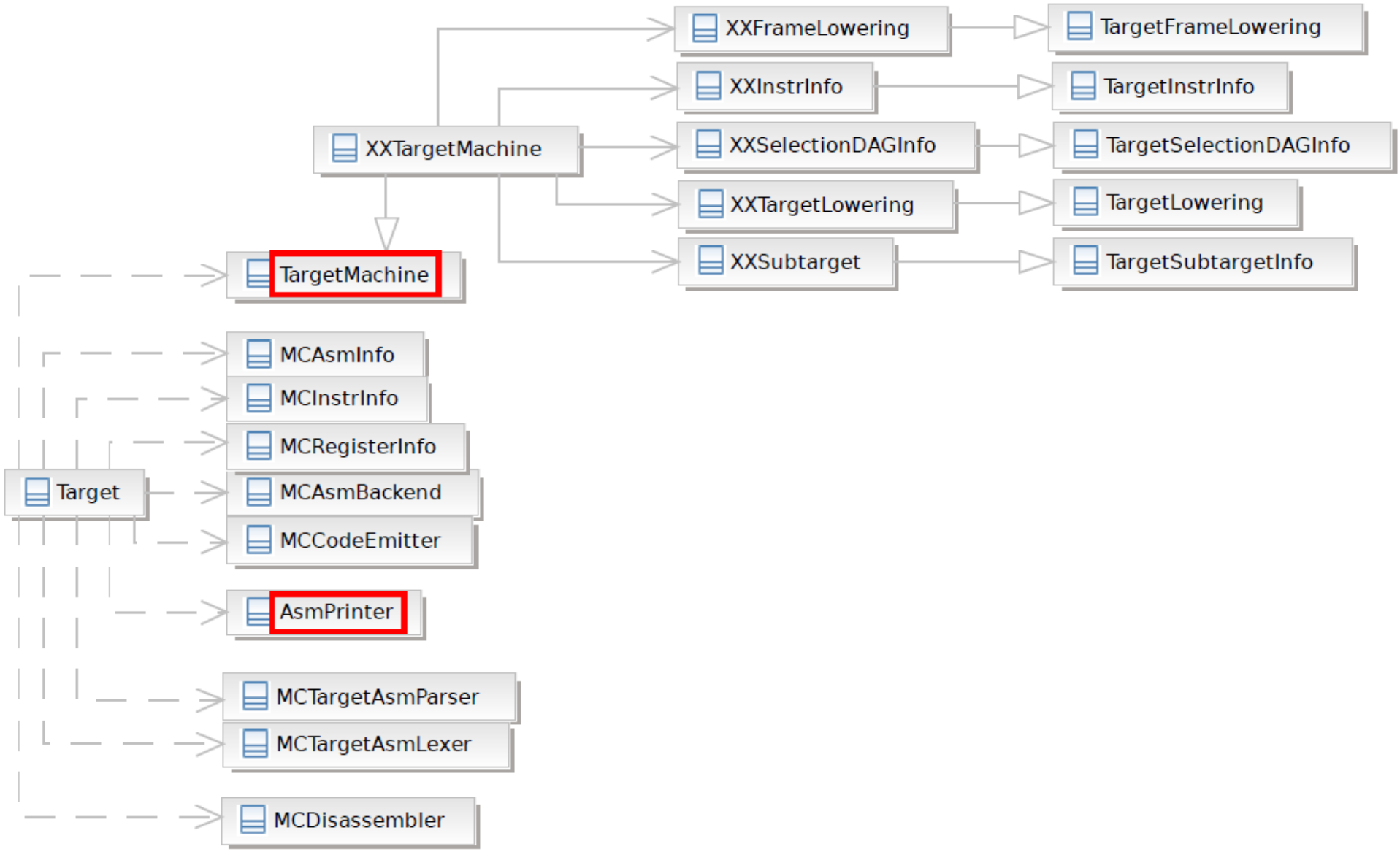
后端代码组织模式

工厂方法模式：

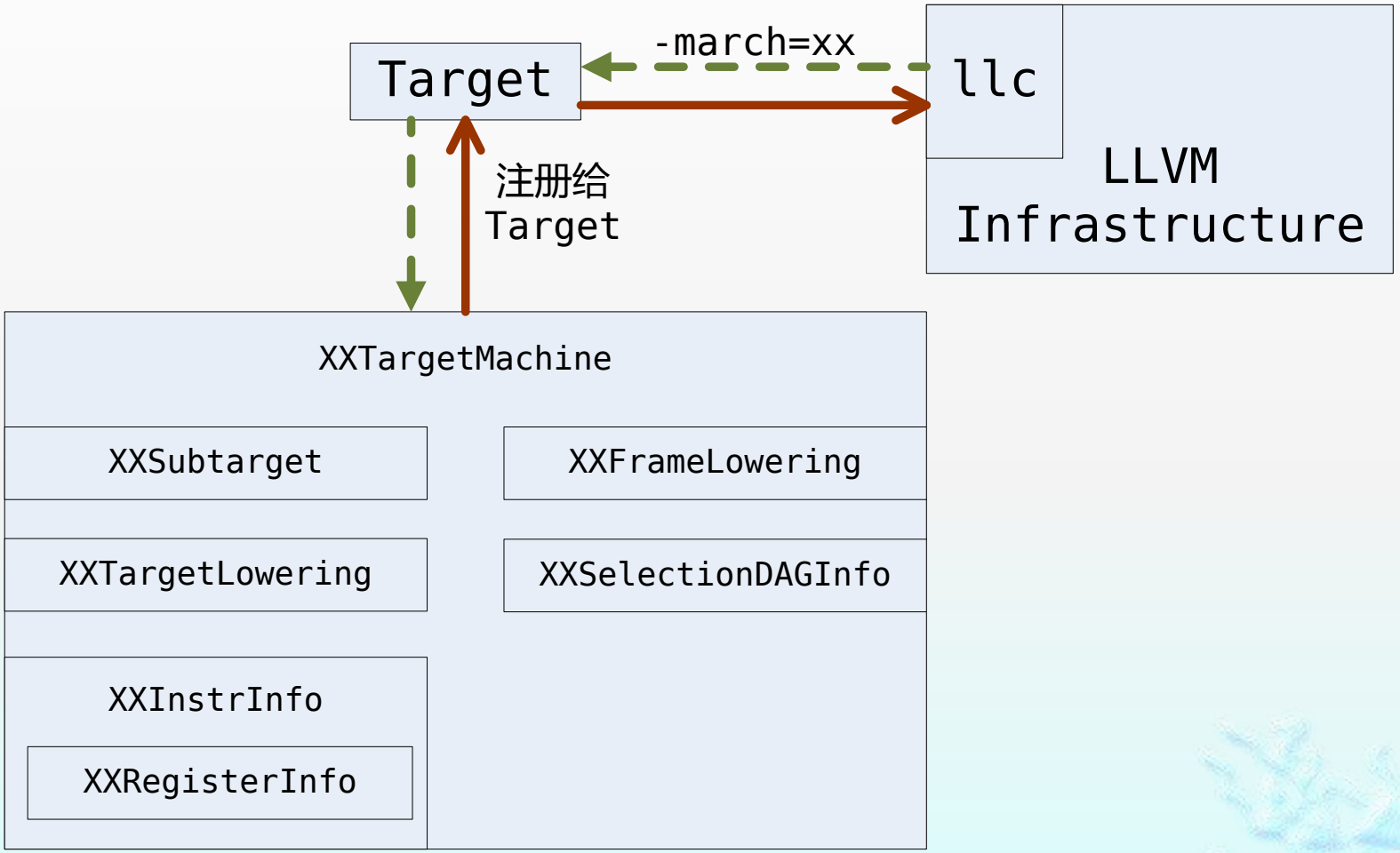
Target类中包含有多个**函数指针**成员，可调用它们创建某个模块的实例、然后将其返回，参见下页UML图示

- TargetMachine类扮演一个关键角色
- MC层是另一个关键角色
- AsmPrinter是二者的桥梁

LLVM后端的类结构图



TargetMachine的结构示意图



将LLVM IR编译至MachineInstr的功能实现
由TargetMachine类中的接口进行控制

LLVM的后端代码结构（1）

```
// in file lib/Target/Sparc/TargetInfo/SparcTargetInfo.cpp
Target llvm::TheSparcTarget;

extern "C" void LLVMInitializeSparcTargetInfo() {
  RegisterTarget<Triple::sparc> XYZ(TheSparcTarget,
                                    "sparc", "Sparc");
}
```

类Target、模板类RegisterTarget定义在
include/llvm/Support/TargetRegistry.h中

“sparc”: target name, 用在”llc -march=sparc …”
“Sparc“: target description, 执行命令”llc -version”显示的内容

LLVM的后端代码结构（2）

LLVMInitializeXXX**TargetInfo**如何被调用？

1. 定义在include/llvm/Support/TargetSelect.h中：

```
……
#define LLVM_TARGET(TargetName) \
        void LLVMInitialize##TargetName##TargetInfo();
#include "llvm/Config/Targets.def“
……
```

2. 在configure及build路径的include/llvm/Config/Targets.def中有

```
…
LLVM_TARGET(TargetName)
…
```

3. Targets.def由配置脚本在执行configure命令时根据指定的选项自动生成

其它类似的初始化函数

在include/llvm/Support/TargetSelect.h中
通过宏定义的其它初始化全局函数：

```
void LLVMInitialize##TargetName##Target();
void LLVMInitialize##TargetName##TargetMC();
void LLVMInitialize##TargetName##AsmPrinter();
void LLVMInitialize##TargetName##AsmParser();
void LLVMInitialize##TargetName##Disassembler();
```

比如

```
extern "C" void LLVMInitializeSparcTarget() {
  RegisterTargetMachine<SparcV8TargetMachine> X(TheSparcTarget);
}
```

类似的C函数接口实现散步在各个基类（TargetMachine, AsmPrinter, TargetMC, …）的派生类实现中。

- LLVM的后端框架及代码结构
- 如何实现一个后端
- 整合汇编器和反汇编器
- 链接器和调试器

添加一个新的target

- 复制一个已有例子。比如Sparc
- 修改配置文件。通过下属命令查找需要更改的文件

```
grep sparc -rni \
    --exclude-dir=Sparc  \
    --exclude-dir=test \
    --exclude-dir=unittests LLVM_SRC_DIR
```

- 改写已有例子。

  注意命名约定，会影响build system
- 参看官方文档

  1. http://www.llvm.org/docs/WritingAnLLVMBackend.html
  2. http://www.llvm.org/docs/CodeGenerator.html#code-generator
- 非官方的例子：TriCore Backend for LLVM

  http://www.opus.ub.uni-erlangen.de/opus/volltexte/2010/1659/pdf/tricore_llvm.pdf

TableGen介绍：文件组织关系

1. 通过include原语组织TableGen文件
比如：`include “llvm/Target/Target.td”`
进一步在Target.td中有：`include “llvm/Target/TargetSelectionDAG.td”`

2. 所有TableGen相关的文件汇集于XX.td中，比如Sparc.td

3. 在build时，通过makefile执行命令：
`llvm-tblgen --option1 --option2 … XX.td`
可用的命令选项通过`llvm-tblgen -help`查看

4. 生成相关C++代码，以.inc为文件后缀名，比如SparcGenInstrInfo.inc

5. 这些文件通过C/C++的预处理命令#include包含进源码中

6. tablegen的官方文档
http://www.llvm.org/docs/TableGenFundamentals.html#tablegen

TableGen文件组织结构示意图：

用TableGen定义指令

- **描述指令的构成**:
  操作数：输入、输出
  指令语法：助记符，...
- **描述指令匹配的语义模式**
  用于指令选择
- **允许在指令定义中不指定语义模式**
  这些指令不会参与指令选择，但仍可以用于代码生成及二进制功能的实现中

定义指令

继承TableGen类**Instruction** (在include/llvm/Target/Target.td中)

// 最直接的写法

```
def AddI32 : Instruction {
   let OutOperandList = (outs I32Reg:$d);
   let InOperandList = (ins I32Reg:$s1, $s2);
   let AsmString = “add $d, $s1, $s2”;
   let Pattern = [(set I32Reg:$d, (add I32Reg:$s1, I32Reg:$s2))]
}
```

// 推荐的、惯常的写法

... (请参考实际的后端例子)

理解指令定义中的语义模式

```
let Pattern = [(set I32Reg:$d, (add I32Reg:$s1, I32Reg:$s2))]
```

- **操作**：add

```
def add: SDNode<“ISD::ADD”, SDTIntBinOp,
                [SDNPCommutative, SDNPAssociative]>;
```

1. 指定操作码：ISD::ADD
2. 描述操作数：**SDTIntBinOp**
   属性：多少个输入/多少个输出
   数据类型列表：i8/i16/i32/i64/f32/f64/iPTR
3. 操作属性：交换性，结合性

- **操作数绑定**： I32Reg:$s1

I32Reg：寄存器类，其数据类型匹配上述的操作数描述
$s1：操作数名称

- **输出操作数的传递**：set

其它基本的tablegen类

```
def  R0: Register<“R0”>; // 描述物理寄存器的实体
…
def  F0: Register<“F0”>;
…

// 寄存器类描述一类物理寄存器的属性
// I32Reg contains R0 ~ R31
def I32Reg: RegisterClass<“XX”, [i32], 32, (sequence “R%u”, 0,
   31)>;
def F32Reg: RegisterClass<“XX”, [f32], 32, (sequence “F%u”, 0,
   31)>;

// 调用约定：函数的参数传递/值返回；函数调用
def XX_CC: CallingConv< [
   CCIfType<[i32], CCAssignToReg<[R0, R1, R2, R3]>>,
   CCIfType<[f32], CCAssignToReg<[F0, F1, F2, F3]>>
] >;
```

- LLVM的后端框架及代码结构
- 如何实现一个后端
- 整合汇编器和反汇编器
- 链接器和调试器

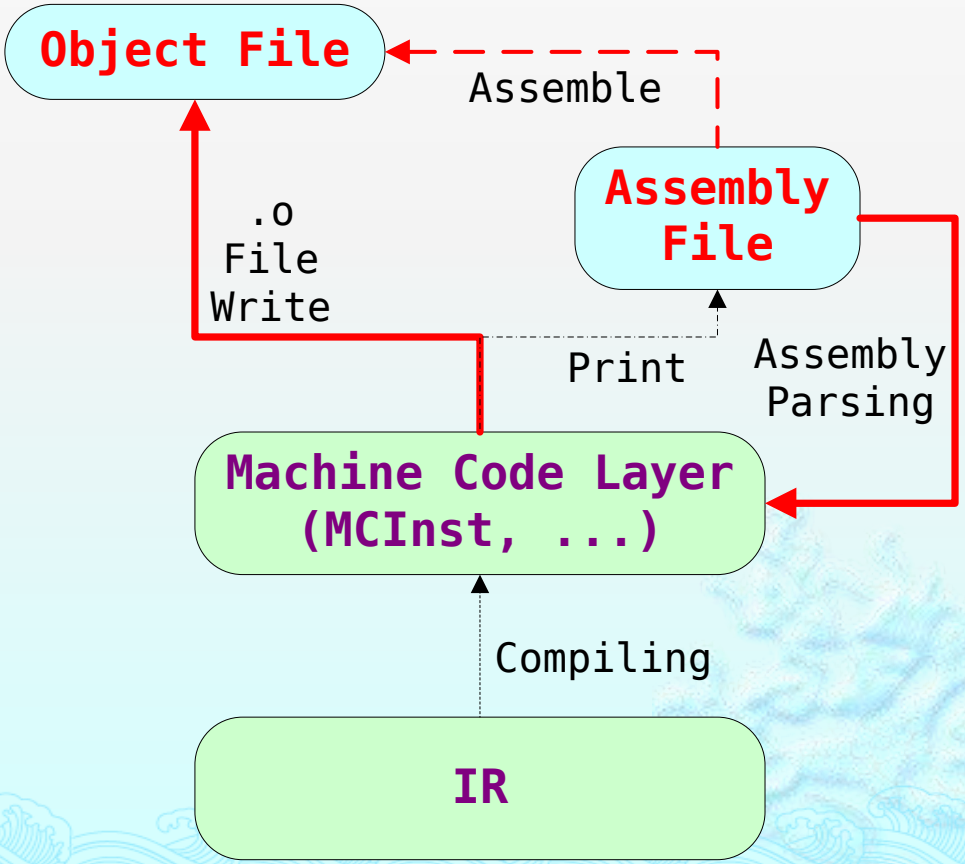
输出二进制目标文件 (1)

两种途径

1. 命令“llc … -filetype=obj …”将IR直接打印为二进制目标文件 (.o file writer)
2. 通过命令llvm-mc将汇编文本转化为二进制目标文件，

相当于一个独立完整的汇编器

参见图中红色的实心箭头



输出二进制目标文件 (2)

在XXInstrInfo.td中定义指令时加入编码信息：

opcode编码 + operand1编码 + operand2编码 + ...

```
def AddI32 : Instruction {
   let OutOperandList = (outs I32Reg:$d);
   let InOperandList = (ins I32Reg:$s1, $s2);
   let AsmString = “add $d, $s1, $s2”;
   let Pattern = [(set I32Reg:$d, (add I32Reg:$s1, I32Reg:$s2))]

   field bits<32> Inst;
   Inst{0-9} = 0b0000000000;
   Inst{10-15} = d;
   Inst{16-21} = s1;
   Inst{22-27} = s2;
}
```

实现汇编功能

实现 .o file writer

- 基类MCCodeEmitter: 描述指令编码，
  `EncodeInstruction();`
- 基类MCELFObjectTargetWriter: 描述重定向信息，...
  `GetRelocType();`
- 基类MCAsmBackend: assembler backend
  `relaxInstruction();`
  `applyFixup();`

实现汇编功能

实现asm parser，

将汇编文本描述转化至MC层的描述

- 基类MCTargetAsmParser

```
virtual bool ParseInstruction(); // 分析一条指令
virtual bool ParseDirective();   // 分析一条directive

// 将分析所得的指令进行匹配识别，如果是合法指令，则将其发射至MC层
virtual bool MatchAndEmitInstruction();
```

- 基类MCTargetAsmLexer

```
virtual AsmToken LexToken(); // 分析一个token
```

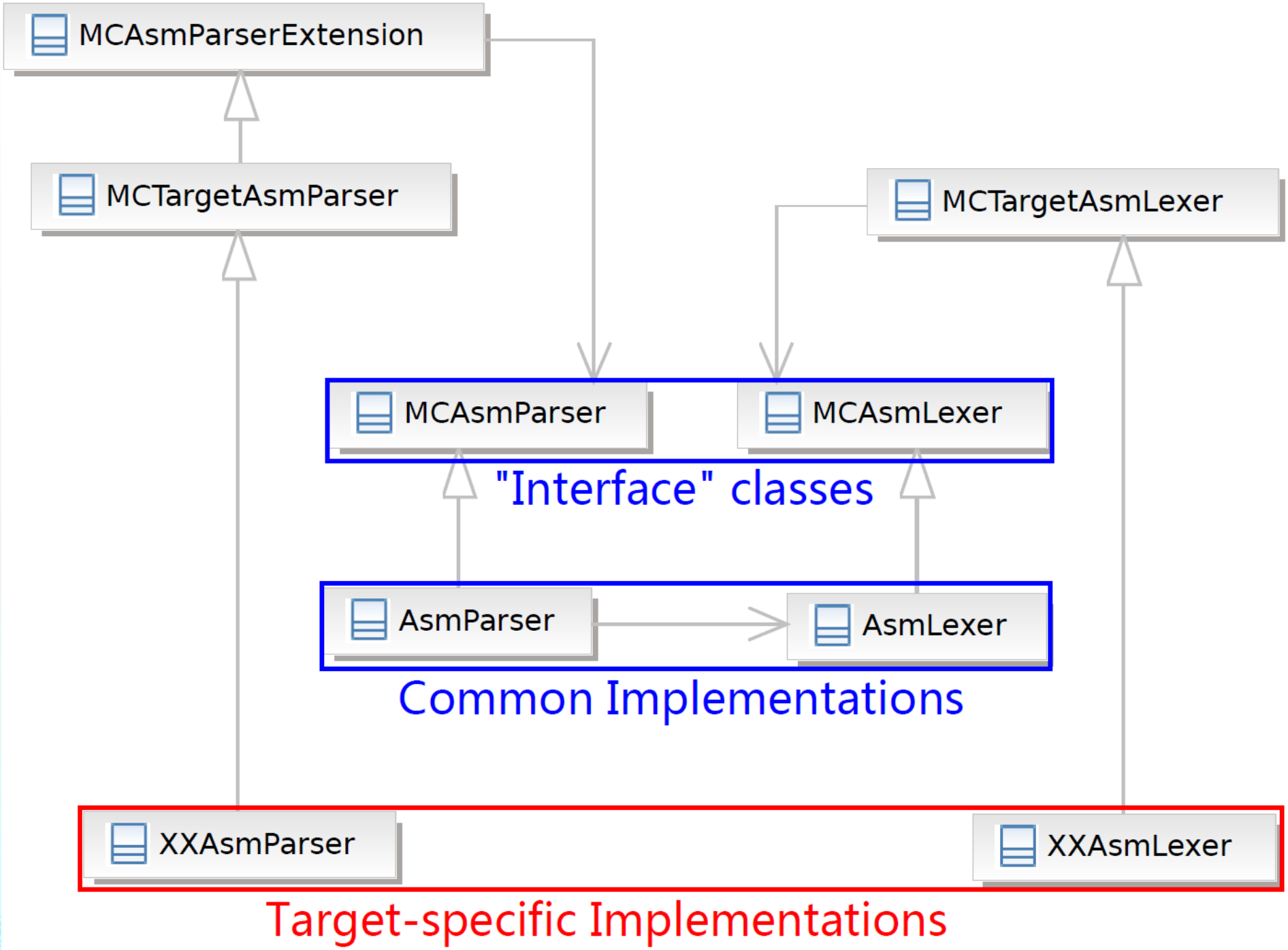
小贴士：类命名习惯

- 对派生类的命名无限制

  比如上表中的XXMCCodeEmitter允许改为XXCodeEmitter或者其它任何名称

- 名称中包含有“Target”字符串的类，一般是与目标相关的实现需要继承的**基类**，不包含“Target”字符串的类，一般是与目标无关的、通用的实现。

  比如MCAsmLexer、AsmLexer、MCTargetAsmLexer三者的关系，见下页图。


MCAsmParserExtension
MCTargetAsmParser
MCTargetAsmLexer
MCAsmParser
MCAsmLexer
"Interface" classes
AsmParser
AsmLexer
Common Implementations
XXAsmParser
XXAsmLexer
Target-specific Implementations

实现反汇编功能

- 基类MCDisassembler

```
const EDInstInfo *getEDInfo() const;
DecodeStatus getInstruction(); // 解码一条指令
```

一个关于汇编/反汇编的非官方文档

http://www.embecosm.com/download/ean10.html

包含MC层全部基本方面:

1. 指令编码、.o file writer
2. 汇编文法分析
3. 解码
4. build system的相应修改

描述**清晰、全面**，强烈推荐!

- LLVM的后端框架及代码结构
- 如何实现一个后端
- 整合汇编器和反汇编器
- 链接器和调试器

工具链平台的其它内容

- 链接器lld
  1. http://llvm.org/devmtg/2012-04-12/Slides/Michael_Spencer.pdf
  2. http://lld.llvm.org/

- 符号调试器lldb
  1. http://www.llvm.org/devmtg/2010-11/Clayton-LLDB.pdf
  2. http://lldb.llvm.org/

End.

Thanks!

Questions?