Getting started with llvm for Iluvatar BI archtecure

by Yunxin.Yang

2021.12.21

目录

[Getting started with llvm for Iluvatar BI archtecure 1](#_Toc2062525644)

[目录 2](#_Toc1581632105)

[Getting started with llvm for Iluvatar BI archtecure 4](#_Toc238915560)

[1. LLVM 相关组建与库 4](#_Toc822685122)

[1.1 LLVM基础库 4](#_Toc974640138)

[2. 后端backend 6](#_Toc1300845836)

[2.1 后端编译优化流程介绍 6](#_Toc1445071479)

[2.1.1指令选择阶段 6](#_Toc1173518961)

[2.1.2 指令调度阶段 23](#_Toc305537343)

[2.1.3 寄存器分配 23](#_Toc927400610)

[2.1.4 后寄存器指令调度阶段 24](#_Toc1870987221)

[2.1.5 代码输出(Code Emission) 24](#_Toc146766510)

[2.2 后端代码结构介绍 24](#_Toc468008765)

[2.3 后端库介绍 25](#_Toc548650566)

[2.4 后端实现函数及类介绍 26](#_Toc418383173)

[2.5 TableGen工具语法及td文件介绍 26](#_Toc419060655)

[2.6 添加自定义流程(pass)介绍 34](#_Toc1902833463)

[3. 编译驱动器ClangDriver研究介绍 34](#_Toc1303929385)

[4. 基于Loongson和Iluvatar平台进行编译器适配 34](#_Toc706983610)

[4.1 Loognson 平台适配相关改动 34](#_Toc433680499)

[5. Iluvatar平台编译器pass 41](#_Toc146774634)

Getting started with llvm for Iluvatar BI archtecure

It is the road you construct a new LLVM compiler for platform archtecture,including a new CPU chip arcgtecture and GPU. for starting the journey of Iluvatar BI archtecture, Learning and reconstructing the compiler pass for the LLVM backend, being define specify BI platform infomations, such as instructions, register and hardware structure.

当重新构建新架构的LLVM编译器时，最重要的几项适配包括：平台相关机器指令的定义，平台相关机器指令的匹配，平台相关寄存器数量与结构的定义，平台相关寄存器分配，MachineInstruction到Machineodeinstruction的转换，调度器的特殊策略等。

1. LLVM 相关组建与库

LLVM包括如下项目：前后端，优化器，汇编器，链接器，标准库，JIT等不同的组建构成，在构建新平台的编译器时需要对上述组建进行重构或者定义。

Clang作为C/C++语言的前端语言解释器，用于解析C/C++、Object-O或者其他自定义语言的前端，而在LLVM和Clang中，编译器驱动程序(compiler driver)是clang工具，它可以隐式的从前端到IR、优化器、后端、链接器到最后的汇编器、目标程序调用所有工具来驱动整个编译过程。

同时Clang作为LLVM项目中重要的独立开源项目，既可以作为编译器前端(compiler frontend)解析C/C++、Object-O等语言成LLVM IR，又可以通过libclangdriver库驱动整个编译器的前后端，类似与GNU GCC或G++，来请求整个编译过程的流程(pass)。

如Clang驱动程序调用的第一个第一个工具是带有-ccl参数的clang自身程序，实现阶段性输出中间代码，类似的参数还有更多。LLVM的各个组件是库，因此clang -ccl会与IR生成器，目标机器代码生成器，以及汇编器库进行链接，clang -ccl能够调用其他库，并监视内存中的编译过程，直到目标文件完成。然后Clang驱动程序调用链接程序生成可执行程序。

其他独立工具如：opt(optimizer)，旨在IR级对程序进行优化,输入前端解析的bitcode IR中间代码，然后对LLVM IR代码进行优化生成优化后的IR代码。llc，将通过特定后端将bitcode中间码转换为目标机器汇编 文件或者目标文件。llvm-mc，该工具用于汇编指令并生成对应平台的目标文件。lli，是LLVM IP的解释器和JIT编译器。llvm-link，在LLVM IR优化阶段，用于将LLVM bitcode链接起来产生完成的LLVM bitcode位码。

1.1 LLVM基础库

libLLVMCore是LLVM的核心库，包括与LLVM IR相关的所有逻辑与功能：IR构造生成如数据、指令、函数，IR校验器，编译过程的各种编译流程。

libLLVMCodeGen实现与目标无关的代码生成和机器级别的的分析转换(lowering,目标无关的代码生存，分配配置通用寄存器指令)。

libLLVMTarget通过通用目标抽象来提供对目标机器信息的访问接口，LLVM Target库包含通用类接口。这些高级抽象接口为在libLLVMCodeen中实现的通用后端算法与为下一个库保留的特定与目标的逻辑之间进行通信的网关。libLLVMTarget接口抽象出通用编译接口，将LLVM IR转换成通用的机器码(Machine Instrution)。特定的目标平台通过继承、实例化通用接口实现特定目标机器码的编译。

libLLVMIluvatarCodeGen，该库是libLLVMCodeGen的具体目标的实现，具有特定于Iluvatar目标的代码生成信息、转换和分析过程，该库构成了Iluvatar的后端。

libclangDriver，clang的驱动程序库，包含编译器驱动程序工具需要的相关类，类似 GCC的命令行参数。

继承和多态性通过将通用的代码生成算法封装到基类抽象出后端的公共任务，而每个特定的后端通过重写基类范型操作或者虚函数从而实现特定目标的特殊任务。

在LLVM设计中，整个编译过程通过流程(pass)来控制，LLVM API允许在程序编译生命周期的不同部分注册任何流程。流程管理器用于注册、调度和声明流程之间的依赖关系。Passanager实例在编译的不同阶段都是可用的。

|  |
| --- |
| void BIPassConfig::addPreEmitPass() {  addPass(createBIMemoryLegalizerPass());  addPass(createBICachePolicyPass());  addPass(&BIInsertSkipsPassID);  addPass(createBIInsertWaitcntsPass());  addPass(createBIFlatAccessLocalPass());  // The hazard recognizer that runs as part of the post-ra scheduler does not  // guarantee to be able handle all hazards correctly. This is because if there  // are multiple scheduling regions in a basic block, the regions are scheduled  // bottom up, so when we begin to schedule a region we don't know what  // instructions were emitted directly before it.  //  // Here we add a stand-alone hazard recognizer pass which can handle all  // cases.  //  // FIXME: This stand-alone pass will emit indiv. S\_NOP 0, as needed. It would  // be better for it to emit S\_NOP <N> when possible.  //addPass(&PostRAHazardRecognizerID);  addPass(createBIHazardFixPass());  addPass(createBIDebuggerInsertNopsPass());  addPass(&BranchRelaxationPassID);  } |

1. 后端backend

LLVM后端由一组代码生分析成器和变换流程(pass)组成，这些流程将LLVM IR转换成目标代码或者汇编代码。

前端(Frontend)负责解析编译新的高级语言，将语言解析成LLVM IR中间码，而后端(Backend)负责将LLVM IR编译成汇编代码或者可执行份文件等机器码，LLVM 中添加一个新的目标需要构建目标相关的信息定义和流程(pass)定义。

所有的平台的后端共享一个通用接口(libLLVMTarget)该接口使用通用API抽象出后端任务，是独立于目标平台的代码生成器的组成部分，目标平台通过实例化通用接口为特殊目标定义专门的信息和流程，目标平台在代码生成器通用类的基础上进行特化实例化，以实现该目标平台的所需的特有功能。

|  |
| --- |
| **LLVM IR**  **流程**  **指令选择**  **指令调度**  **寄存器分配**  **流程**  **代码输出**  **指令调度**  **流程**  **汇编代码**  **目标代码**  **流程** |

* 1. 后端编译优化流程介绍

2.1.1指令选择阶段

指令选择阶段将内存中的LLVM IR中间码转换成制定目标的SelectionDAG(Directed Acyclic Graph),LLVM IR阶段经过Optimizer优化后的中间码为bitcode的三地址格式，指令选择阶段首先将三地址格式的IR转换成SelectionDAG形式，为每条指令添加对应的Node和Edge，用来表示该指令以及与其他指令的依赖关系。Backend通过DAG图，由LLVM代码生成器库(libLLVMCodeGen/IluvatarCodeGen),基于对应的模式匹配的指令选择代码，实现后端目标代码的导出。指令选择阶段会将DAG图中的所有LLVM IR节点转换成目标机器节点，由目标机器指令代替LLVM IR指令。

指令选择是将LLVM IR转化为代表目标指令的SelectionDAG(SDNode)节点，第一步是从LLVM IR指令构建DAG，从而创建一个其节点执行IR操作的SelectionDAG对象；第二步对这些节点执行降级(Lowering)、DAG组合器、优化器阶段，使其能更容易与目标指令向匹配；第三步指令选择算法使用节点模式匹配进行DAG到DAG的转换，并将SelectionDAG节点转换成代表目标指令的节点。

指令选择阶段是后端代码执行流程中执行时间最长的阶段之一，指令选择阶段基本占据llc工具中-O2级优化生成代码时间的一半。

1.SelectionDAG

SelectionDAG类使用DAG来表示每一个基本块(basic block)，每个SDNode对应一个指令或者操作数。SelectionDAG中包括各个节点以及各个节点之间的依赖关系，如节点之间的数据依赖关系、执行先后顺序、非数据流链(强制规定两条不相关的指令之间的执行顺序)。SelectionDAG对象有一个表示基本块入口的特殊EntryToken节点，该节点对应一个类型为other 的值。

==

2.降级Lowering

|  |
| --- |
| 类型合法化2  DAG合并  DAG合法化  DAG合并2  指令选择  DAG合并  矢量合法化  类型合法化1  DAG合并1  LLVM IR  0SelectionDAG节点  IR指令映射到SelectionDAG节点  目标降级 |

图 降级过程

在指令选择之前，SelectionDAG类中已经有一些特定于目标的节点，上图为指令选择阶段在指令选择之前的所有操作。

从SelectionDAGBuilder实例中产生的输出SelectionDAGd对象还需要经过上图降级过程的一些列流程，才能进行指令选择，包括DAG合，类型合法化，矢量合法化，DAG合法化。

首先SelectionDAGBuilder实例(SelectionDAGISel.cpp)访问每一个函数，并为每一个函数基础块创建一个SelectionDAG对象，在这个过程中，某些特殊的IR指令(如 call,ret)已经遵循特定于目标机器的习惯用法，以便转换成SelectionDAG。上述问题的处理算法在TargetLowering类中进行实现(class TargetLowering : public TargetLoweringBase)，TargetLowering类对抽象接口，每个编译目标需要根据目标特性实现该接口，同时该接口依然有很多对所有编译目标通用的功能。

特定目标为了实现该功能接口中目标相关的功能，每个编译目标都需要实现一个具体的<Target>TargetLowering的TargeLowering的子类，如IluvatarTargetLowering子类(class IluvatarTargetLowering : public TargetLowering , llvm/lib/Target/Iluvatar/IluvatarISelLowering.h)。每个目标都需要重载这些与将高层次的目标节点降低到更低层次、更接近目标机器的功能相关的函数。在指令选择阶段的降级(Lowering)流程中，只有以小部分的节点需要以这种目标相关的方式降低，大部分的其他节点则在指令选择时被匹配和替换。如：llvm/lib/Target/Iluvatar/BIISelLowering.cpp中:

BITargetLowering::BITargetLowering(const TargetMachine &TM,const BISubtarget &STI): IluvatarTargetLowering(TM, STI), Subtarget(&STI)，BITargetLowering是Iluvatar平台中针对BI目标的特殊节点的降级(Lowering)实现函数的实现类，其中实现在指令选择前降级(Lowering)目标相关节点转换的函数接口有：bool BITargetLowering::CanLowerReturn，SDValue BITargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG)。

|  |
| --- |
| SDValue BITargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) const {  switch (Op.getOpcode()) {  default:  return IluvatarTargetLowering::LowerOperation(Op, DAG);  case ISD::BRCOND:  return LowerBRCOND(Op, DAG);  case ISD::LOAD: {  SDValue Result = LowerLOAD(Op, DAG);  assert((!Result.getNode() || Result.getNode()->getNumValues() == 2) &&  "Load should return a value and a chain");  return Result;  }  case ISD::FSIN:  case ISD::FCOS:  return LowerTrig(Op, DAG);  case ISD::SELECT:  return LowerSELECT(Op, DAG);  case ISD::ATOMIC\_CMP\_SWAP:  return LowerATOMIC\_CMP\_SWAP(Op, DAG);  case ISD::STORE:  return LowerSTORE(Op, DAG);  case ISD::GlobalAddress: {  MachineFunction &MF = DAG.getMachineFunction();  BIMachineFunctionInfo \*MFI = MF.getInfo<BIMachineFunctionInfo>();  return LowerGlobalAddress(MFI, Op, DAG);  }  case ISD::INTRINSIC\_WO\_CHAIN:  return LowerINTRINSIC\_WO\_CHAIN(Op, DAG);  case ISD::INTRINSIC\_W\_CHAIN:  return LowerINTRINSIC\_W\_CHAIN(Op, DAG);  case ISD::INTRINSIC\_VOID:  return LowerINTRINSIC\_VOID(Op, DAG);  case ISD::ADDRSPACECAST:  return lowerADDRSPACECAST(Op, DAG);  case ISD::INSERT\_VECTOR\_ELT:  return lowerINSERT\_VECTOR\_ELT(Op, DAG);  case ISD::EXTRACT\_VECTOR\_ELT:  return lowerEXTRACT\_VECTOR\_ELT(Op, DAG);  case ISD::EXTRACT\_VECTOR\_ELT:  return lowerEXTRACT\_VECTOR\_ELT(Op, DAG);  case ISD::SETCC:  return lowerSETCC(Op, DAG);  case ISD::BUILD\_VECTOR:  return lowerBUILD\_VECTOR(Op, DAG);  case ISD::FP\_ROUND:  return lowerFP\_ROUND(Op, DAG);  case ISD::FP\_TO\_SINT:  return LowerFP\_TO\_SINT(Op, DAG);  case ISD::TRAP:  return lowerTRAP(Op, DAG);  case ISD::DEBUGTRAP:  return lowerDEBUGTRAP(Op, DAG);  case ISD::FABS:  case ISD::FNEG:  return splitUnaryVectorOp(Op, DAG);  case ISD::FDIV:  return LowerFDIV(Op, DAG);  case ISD::SHL:  case ISD::SRA:  case ISD::SRL:  case ISD::ADD: {  if (Op.getValueType() == MVT::i1)  return IluvatarTargetLowering::LowerOperation(Op, DAG);  }  case ISD::SUB:  case ISD::MUL:  case ISD::SMIN:  case ISD::SMAX:  case ISD::UMIN:  case ISD::UMAX:  case ISD::FMINNUM:  case ISD::FMAXNUM:  case ISD::FADD:  case ISD::FSUB:  case ISD::FMUL:  return splitBinaryVectorOp(Op, DAG);  case ISD::MULHS:  case ISD::MULHU:  return LowerMULH(Op, DAG);  }  return SDValue();  } |

3. DAG合并与以及合法化

DAG合并流程对次优的SelectionDAG图结构进行优化，为此，首先继续宁节点的匹配，并尽可能使用更加简单的结构替换当前节点。特定于目标的合并过程可以识别节点的模式并通过合并或折叠等方式提高其对于指定目标的指令选择质量。

DAG合并的程序入口为：下面是常见的目标无关的DAG合并函数

|  |
| --- |
| class DAGCombiner {  SelectionDAG &DAG;  const TargetLowering &TLI;  const SelectionDAGTargetInfo \*STI;  CombineLevel Level;  CodeGenOpt::Level OptLevel;  bool LegalDAG = false;  bool LegalOperations = false;  bool LegalTypes = false;  bool ForCodeSize;  bool DisableGenericCombines;  /// Worklist of all of the nodes that need to be simplified.  ///  /// This must behave as a stack -- new nodes to process are pushed onto the  /// back and when processing we pop off of the back.  ///  /// The worklist will not contain duplicates but may contain null entries  /// due to nodes being deleted from the underlying DAG.  SmallVector<SDNode \*, 64> Worklist;  ####################################################################################  /// This is the entry point for the file.  void SelectionDAG::Combine(CombineLevel Level, AliasAnalysis \*AA,  CodeGenOpt::Level OptLevel) {  /// This is the main entry point to this class.  DAGCombiner(\*this, AA, OptLevel).Run(Level);  }  ####################################################################################  //===----------------------------------------------------------------------===//  // Main DAG Combiner implementation  //===----------------------------------------------------------------------===//  void DAGCombiner::Run(CombineLevel AtLevel) {  // set the instance variables, so that the various visit routines may use it.  Level = AtLevel;  LegalDAG = Level >= AfterLegalizeDAG;  LegalOperations = Level >= AfterLegalizeVectorOps;  LegalTypes = Level >= AfterLegalizeTypes;  WorklistInserter AddNodes(\*this);  // Add all the dag nodes to the worklist.  for (SDNode &Node : DAG.allnodes())  AddToWorklist(&Node);  // Create a dummy node (which is not added to allnodes), that adds a reference  // to the root node, preventing it from being deleted, and tracking any  // changes of the root.  HandleSDNode Dummy(DAG.getRoot());  // While we have a valid worklist entry node, try to combine it.  while (SDNode \*N = getNextWorklistEntry()) {  // If N has no uses, it is dead. Make sure to revisit all N's operands once  // N is deleted from the DAG, since they too may now be dead or may have a  // reduced number of uses, allowing other xforms.  if (recursivelyDeleteUnusedNodes(N))  continue;  WorklistRemover DeadNodes(\*this);  // If this combine is running after legalizing the DAG, re-legalize any  // nodes pulled off the worklist.  if (LegalDAG) {  SmallSetVector<SDNode \*, 16> UpdatedNodes;  bool NIsValid = DAG.LegalizeOp(N, UpdatedNodes);  for (SDNode \*LN : UpdatedNodes)  AddToWorklistWithUsers(LN);  if (!NIsValid)  continue;  }  LLVM\_DEBUG(dbgs() << "\nCombining: "; N->dump(&DAG));  // Add any operands of the new node which have not yet been combined to the  // worklist as well. Because the worklist uniques things already, this  // won't repeatedly process the same operand.  CombinedNodes.insert(N);  for (const SDValue &ChildN : N->op\_values())  if (!CombinedNodes.count(ChildN.getNode()))  AddToWorklist(ChildN.getNode());  SDValue RV = combine(N);  if (!RV.getNode())  continue;  ++NodesCombined;  // If we get back the same node we passed in, rather than a new node or  // zero, we know that the node must have defined multiple values and  // CombineTo was used. Since CombineTo takes care of the worklist  // mechanics for us, we have no work to do in this case.  if (RV.getNode() == N)  continue;  assert(N->getOpcode() != ISD::DELETED\_NODE &&  RV.getOpcode() != ISD::DELETED\_NODE &&  "Node was deleted but visit returned new node!");  LLVM\_DEBUG(dbgs() << " ... into: "; RV.getNode()->dump(&DAG));  if (N->getNumValues() == RV.getNode()->getNumValues())  DAG.ReplaceAllUsesWith(N, RV.getNode());  else {  assert(N->getValueType(0) == RV.getValueType() &&  N->getNumValues() == 1 && "Type mismatch");  DAG.ReplaceAllUsesWith(N, &RV);  }  // Push the new node and any users onto the worklist. Omit this if the  // new node is the EntryToken (e.g. if a store managed to get optimized  // out), because re-visiting the EntryToken and its users will not uncover  // any additional opportunities, but there may be a large number of such  // users, potentially causing compile time explosion.  if (RV.getOpcode() != ISD::EntryToken) {  AddToWorklist(RV.getNode());  AddUsersToWorklist(RV.getNode());  }  // Finally, if the node is now dead, remove it from the graph. The node  // may not be dead if the replacement process recursively simplified to  // something else needing this node. This will also take care of adding any  // operands which have lost a user to the worklist.  recursivelyDeleteUnusedNodes(N);  }  // If the root changed (e.g. it was a dead load, update the root).  DAG.setRoot(Dummy.getValue());  DAG.RemoveDeadNodes();  } |

在Iluvatar平台以及BI芯片中，目标相关的Lowering在Iluvatar目录下接口函数重载实现，llvm/lib/Target/Iluvatar/BIISelLowering.cpp。

|  |
| --- |
| IluvatarTargetLowering::LowerReturn(Chain, CallConv, isVarArg, Outs, OutVals, DL, DAG);  IluvatarTargetLowering::LowerOperation(Op, DAG);//针对BI芯片的特殊操作指令进行合并  IluvatarTargetLowering::LowerGlobalAddress(MFI, Op, DAG); |

另外在llvm/lib/Target/Iluvatar/IluvatarISelLowering.cpp中定义Iluvatar平台通用指令降级函数，这部分指令在指令选择和匹配时被降级。

|  |
| --- |
| IluvatarTargetLowering::LowerReturn(SDValue Chain, CallingConv::ID CallConv,  IluvatarTargetLowering::LowerCall(CallLoweringInfo &CLI,  IluvatarTargetLowering::LowerDYNAMIC\_STACKALLOC(SDValue Op,  SDValue IluvatarTargetLowering::LowerOperation(SDValue Op,  SDValue IluvatarTargetLowering::LowerGlobalAddress(IluvatarMachineFunction \*MFI,  SDValue IluvatarTargetLowering::LowerCONCAT\_VECTORS(SDValue Op,  IluvatarTargetLowering::LowerEXTRACT\_SUBVECTOR(SDValue Op,  SDValue IluvatarTargetLowering::LowerDIVREM24(SDValue Op, SelectionDAG &DAG,  void IluvatarTargetLowering::LowerUDIVREM64(SDValue Op,  SDValue IluvatarTargetLowering::LowerUDIVREM(SDValue Op,  SDValue IluvatarTargetLowering::LowerSDIVREM(SDValue Op,  SDValue IluvatarTargetLowering::LowerFREM(SDValue Op, SelectionDAG &DAG) const {  SDValue IluvatarTargetLowering::LowerFCEIL(SDValue Op,  SDValue IluvatarTargetLowering::LowerFTRUNC(SDValue Op,  SDValue IluvatarTargetLowering::LowerFRINT(SDValue Op,  SDValue IluvatarTargetLowering::LowerFNEARBYINT(SDValue Op,  SDValue IluvatarTargetLowering::LowerFROUND32\_16(SDValue Op,  SDValue IluvatarTargetLowering::LowerFROUND64(SDValue Op,  SDValue IluvatarTargetLowering::LowerFROUND(SDValue Op,  SDValue IluvatarTargetLowering::LowerFFLOOR(SDValue Op,  SDValue IluvatarTargetLowering::LowerFLOG(SDValue Op, SelectionDAG &DAG,  SDValue IluvatarTargetLowering::LowerCTLZ\_CTTZ(SDValue Op,  SDValue IluvatarTargetLowering::LowerMUL\_LOHI(SDValue Op,  SDValue IluvatarTargetLowering::LowerINT\_TO\_FP32(SDValue Op, SelectionDAG &DAG,  SDValue IluvatarTargetLowering::LowerINT\_TO\_FP64(SDValue Op, SelectionDAG &DAG,  SDValue IluvatarTargetLowering::LowerUINT\_TO\_FP(SDValue Op,  SDValue IluvatarTargetLowering::LowerSINT\_TO\_FP(SDValue Op,  SDValue IluvatarTargetLowering::LowerFP64\_TO\_INT(SDValue Op, SelectionDAG &DAG,  SDValue IluvatarTargetLowering::LowerFP\_TO\_FP16(SDValue Op,  SDValue IluvatarTargetLowering::LowerFP\_TO\_SINT(SDValue Op,  SDValue IluvatarTargetLowering::LowerFP\_TO\_UINT(SDValue Op,  SDValue IluvatarTargetLowering::LowerADD(SDValue Op, |

setTargetDAGCombine()负责标记当前编译目标下应该合并的的节点，SDValue BITargetLowering::PerformDAGCombine(SDNode \*N, DAGCombinerInfo &DCI), 根据节点的指令类型调用相关的函数 performXXXCombine()用来执行该指令的合并。如下例子：

|  |
| --- |
| /// Targets should invoke this method for each target independent node that  /// they want to provide a custom DAG combiner for by implementing the  /// PerformDAGCombine virtual method.  void setTargetDAGCombine(ISD::NodeType NT) {  assert(unsigned(NT >> 3) < array\_lengthof(TargetDAGCombineArray));  TargetDAGCombineArray[NT >> 3] |= 1 << (NT&7);  }  setTargetDAGCombine(ISD::FADD);  SDValue BITargetLowering::performFAddCombine(SDNode \*N, DAGCombinerInfo &DCI) const {  if (DCI.getDAGCombineLevel() < AfterLegalizeDAG)  return SDValue();  SelectionDAG &DAG = DCI.DAG;  EVT VT = N->getValueType(0);  if (VT.isVector())  return SDValue();  SDLoc SL(N);  SDValue LHS = N->getOperand(0);  SDValue RHS = N->getOperand(1);  // These should really be instruction patterns, but writing patterns with  // source modiifiers is a pain.  // fadd (fadd (a, a), b) -> mad 2.0, a, b  if (LHS.getOpcode() == ISD::FADD) {  SDValue A = LHS.getOperand(0);  if (A == LHS.getOperand(1)) {  unsigned FusedOp = getFusedOpcode(DAG, N, LHS.getNode());  if (FusedOp != 0) {  const SDValue Two = DAG.getConstantFP(2.0, SL, VT);  return DAG.getNode(FusedOp, SL, VT, A, Two, RHS);  }  }  }  // fadd (b, fadd (a, a)) -> mad 2.0, a, b  if (RHS.getOpcode() == ISD::FADD) {  SDValue A = RHS.getOperand(0);  if (A == RHS.getOperand(1)) {  unsigned FusedOp = getFusedOpcode(DAG, N, RHS.getNode());  if (FusedOp != 0) {  const SDValue Two = DAG.getConstantFP(2.0, SL, VT);  return DAG.getNode(FusedOp, SL, VT, A, Two, LHS);  }  }  }  return SDValue();  }  //下面是ISD::FADD 指令的定义  def fadd : SDNode<"ISD::FADD" , SDTFPBinOp, [SDNPCommutative]>;  SDTFPBinOp //节点类型  SDNPCommutative //节点优先级  def SDTFPBinOp : SDTypeProfile<1, 2, [ // fadd, fmul, etc.  SDTCisSameAs<0, 1>, SDTCisSameAs<0, 2>, SDTCisFP<0>  ]>; |

4.类型合法化

类型合法化流程保证指令选择阶段只需要处理合法类型，“合法类型”是指编译目标原生支持的类型。例如支持i32类型的目标，操作数i64的加法操作是非法的，对于这种情况，类型合法化工具会将i64的操作数拆分成两个i32操作数，同时插入适当的处理节点。

类型合法化流程运行两次，一次是在第一次DAG合并后，一次是在矢量合法化以后。

编译目标会事先定义好寄存器类与每个类型(操作数，数据类型)的对应关系(在Target下对应的目录下定义td文件中定义寄存器类型和指令类型)，从而明确声明其所支持的类型。在此基础上，编译器必须检测和处理非法类型：

1.标量类型可以被提升或者扩展，矢量类型可以被分割、标量化或者填充。

|  |
| --- |
| //定义 Target Lowering 类，llvm/include/llvm/CodeGen/TargetLowering.h。  class TargetLoweringBase  //vector矢量类型分割，抽象类中的实现  virtual unsigned getVectorTypeBreakdownForCallingConv(  LLVMContext &Context, CallingConv::ID CC, EVT VT, EVT &IntermediateVT,  unsigned &NumIntermediates, MVT &RegisterVT) const {  return getVectorTypeBreakdown(Context, VT, IntermediateVT, NumIntermediates,  RegisterVT);  }  //特定目标如BI芯片的具体实现，特殊情况下采用BI特定的处理方法，最后默认采用基类通用的实现。llvm/lib/Target/Iluvatar/BIISelLowering.cpp  unsigned BITargetLowering::getVectorTypeBreakdownForCallingConv(  LLVMContext &Context, CallingConv::ID CC, EVT VT, EVT &IntermediateVT,  unsigned &NumIntermediates, MVT &RegisterVT) const {  if (CC != CallingConv::ILUVATAR\_KERNEL && VT.isVector()) {  unsigned NumElts = VT.getVectorNumElements();  EVT ScalarVT = VT.getScalarType();  unsigned Size = ScalarVT.getSizeInBits();  if (Size == 32) {  RegisterVT = ScalarVT.getSimpleVT();  IntermediateVT = RegisterVT;  NumIntermediates = NumElts;  return NumIntermediates;  }  if (Size == 64) {  RegisterVT = MVT::i32;  IntermediateVT = RegisterVT;  NumIntermediates = 2 \* NumElts;  return NumIntermediates;  }  if (Size == 16 && Subtarget->has16BitInsts()) {  RegisterVT = VT.isInteger() ? MVT::v2i16 : MVT::v2f16;  IntermediateVT = RegisterVT;  NumIntermediates = (NumElts + 1) / 2;  return NumIntermediates;  }  }  return TargetLowering::getVectorTypeBreakdownForCallingConv(  Context, CC, VT, IntermediateVT, NumIntermediates, RegisterVT);  }  //降级流程的调用过程，llvm/lib/CodeGen/SelectionDAG/SelectionDAGBuilder.cpp。  const TargetLowering &TLI = DAG.getTargetLoweringInfo();  ... ...  if (IsABIRegCopy) {  NumRegs = TLI.getVectorTypeBreakdownForCallingConv(  \*DAG.getContext(), CallConv.getValue(), ValueVT, IntermediateVT,  NumIntermediates, RegisterVT);  } else {  NumRegs =  TLI.getVectorTypeBreakdown(\*DAG.getContext(), ValueVT, IntermediateVT,  NumIntermediates, RegisterVT);  } |

2.自定义方法合法化类型

可以通过自定义的方法对某些类型进行合法化处理。

5.矢量合法化

另外有些情况下，目标平台有相应的寄存器类支持向量类型，但可能不支持某些针对给定向量类型的特定操作，比如向量类型的ADD、OR、AND等，比如v4i32类型。矢量合法化工具需要处理这些情况，使用合法的类型和指令提升或扩展操作。编译目标可以痛苦自定义的方式处理合法化问题。

Iluvatar平台对矢量合法化操作作了定义，IluvatarISelLowering.cpp定义了Iluvatar平台一些通用的矢量合法化的函数功能，BIISelLowring.cpp定义了BI目标在Iluvatar平台上特有的矢量合法化的功能函数。

|  |
| --- |
| // Lower floating point store/load to integer store/load to reduce the number  // of patterns in tablegen.  setOperationAction(ISD::LOAD, MVT::f32, Promote);  AddPromotedToType(ISD::LOAD, MVT::f32, MVT::i32);  setOperationAction(ISD::LOAD, MVT::v2f32, Promote);  AddPromotedToType(ISD::LOAD, MVT::v2f32, MVT::v2i32);  setOperationAction(ISD::LOAD, MVT::v3f32, Promote);  AddPromotedToType(ISD::LOAD, MVT::v3f32, MVT::v3i32);  setOperationAction(ISD::LOAD, MVT::v4f32, Promote);  AddPromotedToType(ISD::LOAD, MVT::v4f32, MVT::v4i32);  setOperationAction(ISD::LOAD, MVT::v5f32, Promote);  AddPromotedToType(ISD::LOAD, MVT::v5f32, MVT::v5i32); |
| setLoadExtAction(ISD::EXTLOAD, VT, MVT::v2i8, Expand);  setLoadExtAction(ISD::SEXTLOAD, VT, MVT::v2i8, Expand);  setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::v2i8, Expand);  setLoadExtAction(ISD::EXTLOAD, VT, MVT::v4i8, Expand);  setLoadExtAction(ISD::SEXTLOAD, VT, MVT::v4i8, Expand);  setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::v4i8, Expand);  setLoadExtAction(ISD::EXTLOAD, VT, MVT::v2i16, Expand);  setLoadExtAction(ISD::SEXTLOAD, VT, MVT::v2i16, Expand);  setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::v2i16, Expand);  setLoadExtAction(ISD::EXTLOAD, VT, MVT::v3i16, Expand);  setLoadExtAction(ISD::SEXTLOAD, VT, MVT::v3i16, Expand);  setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::v3i16, Expand);  setLoadExtAction(ISD::EXTLOAD, VT, MVT::v4i16, Expand);  setLoadExtAction(ISD::SEXTLOAD, VT, MVT::v4i16, Expand);  setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::v4i16, Expand); |
| // We need to custom lower vector stores from local memory  setOperationAction(ISD::LOAD, MVT::v2i32, Custom);  setOperationAction(ISD::LOAD, MVT::v4i32, Custom);  setOperationAction(ISD::LOAD, MVT::v8i32, Custom);  setOperationAction(ISD::LOAD, MVT::v16i32, Custom);  setOperationAction(ISD::LOAD, MVT::i1, Custom);  setOperationAction(ISD::STORE, MVT::v2i32, Custom);  setOperationAction(ISD::STORE, MVT::v4i32, Custom);  setOperationAction(ISD::STORE, MVT::v8i32, Custom);  setOperationAction(ISD::STORE, MVT::v16i32, Custom);  setOperationAction(ISD::STORE, MVT::i1, Custom); |

6.DAG合法化

DAG合法化与矢量合法化程序具有相同的作用，但DAG合法化负责处理对不支持类型(标量和矢量)的所有剩余操作。它也包含提升、扩展、自定义及诶的那处理等之类的操作。但是对于一些不支持的操作，例如i8类型带符号整数与浮点数转换(ISD::SINT\_TO\_FP)，这就需要DAG合法器进行提升、扩展、自定义处理相应的操作。

如对上述ISD::SINT\_TO\_FP的提升、扩展操作：

|  |
| --- |
| bool DAGTypeLegalizer::PromoteIntegerOperand(SDNode \*N, unsigned OpNo) :  case ISD::SINT\_TO\_FP: Res = PromoteIntOp\_SINT\_TO\_FP(N);  SDValue DAGTypeLegalizer::PromoteIntOp\_SINT\_TO\_FP(SDNode \*N) {  return SDValue(DAG.UpdateNodeOperands(N,  SExtPromotedInteger(N->getOperand(0))), 0);  }  #############################################################=  DAGTypeLegalizer::ExpandIntegerOperand(SDNode \*N, unsigned OpNo) : //类型合法化  case ISD::SINT\_TO\_FP: Res = ExpandIntOp\_SINT\_TO\_FP(N);  SDValue DAGTypeLegalizer::ExpandIntOp\_SINT\_TO\_FP(SDNode \*N) {  bool IsStrict = N->isStrictFPOpcode();  SDValue Chain = IsStrict ? N->getOperand(0) : SDValue();  SDValue Op = N->getOperand(IsStrict ? 1 : 0);  EVT DstVT = N->getValueType(0);  RTLIB::Libcall LC = RTLIB::getSINTTOFP(Op.getValueType(), DstVT);  assert(LC != RTLIB::UNKNOWN\_LIBCALL &&  "Don't know how to expand this SINT\_TO\_FP!");  TargetLowering::MakeLibCallOptions CallOptions;  CallOptions.setSExt(true);  std::pair<SDValue, SDValue> Tmp =  TLI.makeLibCall(DAG, LC, DstVT, Op, CallOptions, SDLoc(N), Chain);  if (!IsStrict)  return Tmp.first;  ReplaceValueWith(SDValue(N, 1), Tmp.second);  ReplaceValueWith(SDValue(N, 0), Tmp.first);  return SDValue();  }  //Convert Float Results to Integer  void DAGTypeLegalizer::SoftenFloatResult(SDNode \*N, unsigned ResNo)  case ISD::SINT\_TO\_FP:  case ISD::UINT\_TO\_FP: R = SoftenFloatRes\_XINT\_TO\_FP(N); break;  ##################################################################=  /// Return a legal replacement for the given operation, with all legal operands.  void SelectionDAGLegalize::LegalizeOp(SDNode \*Node) //DAG合法化  case ISD::FP\_TO\_FP16:  case ISD::SINT\_TO\_FP:  case ISD::UINT\_TO\_FP:  case ISD::EXTRACT\_VECTOR\_ELT:  case ISD::LROUND:  case ISD::LLROUND:  case ISD::LRINT:  case ISD::LLRINT:  Action = TLI.getOperationAction(Node->getOpcode(),  Node->getOperand(0).getValueType());  break; |

7.DAG到DAG指令选择

DAG到DAG指令选择的目的是通过使用模式匹配将目标无关节点转化目标相关节点，指令选择算法是一个局部算法，每次在SelectionDAG(基础块，BasicBlock)实例上执行。

|  |  |
| --- | --- |
| WeChat Image_20211221114111 | WeChat Image_20211221114120 |
| 指令选择前SelectionDAG节点图 | 指令选择前SelectionDAG节点图 |

SelectionDAG节点图经过DAG合并、类型合法化、矢量合法化、DAG合法化等一些列操作后SelectionDAG节点图被逐步优化，如下图，左图中为指令选择前的SelectionDAG节点图，右图为指令选择后的SelectionDAG节点图，作图中部分节点被转换成了目标相关的指令，另外一些寄存器相关的节点保持不变，等待下一步寄存器分配继续优化。

目标编译的模式匹配方法定义在目标的Target目录里，如Iluvatar平台，编译目标通过IluvatarDAGToDAGISel 类，作为SelectionDAGISel的子类继承SelectionDAGISel类并实现Select方法来进行指令选择。

如Iluvatar平台实现Select()方法，路径如下：llvm/lib/Target/Iluvatar/IluvatarISelDAGToDAG.cpp，Iluvatar中的 IluvatarDAGToDAGISel::Select(SDNode \*N)，该方法传入一个待匹配的SDNode作为参数，并返回表示实际指令的SDNode。

Select()方法允许以两种方式来匹配实际指令，最直接的方法是通过调用从TableGen模式生成的匹配代码。但是这种生成的匹配模式的方式不可能完全支持所有的所有的指令匹配行为，特别是一些不常见的匹配方式。在这种情况下，则需要编写定制化的C++算法函数实现匹配逻辑。两种匹配方法如下：

1. Select()方法调用SelectCode()。TableGen工具为每一个编译器目标生成SelectCode()方法，在此代码中，包含由将ISD和<Target>ISD节点映射到实际指令节点的MatcherTable匹配表。该匹配表是从.td文件(IluvatarInstrInfo.td)中的指令定义生成。SelectCode()方法最后调用SelectCodeCommo()方法。SelectCodeCommo()方法是一个与目标相关的函数，以便通过使用之前生成的匹配表来匹配节点。TableGen工具有一个专门的指令选择后端来生成这些匹配表和函数，如：

llvm-tblgen -gen-dag-isel IluvatarXXX.td -I xxx/iluvatar/include

上述工具接口生成的输出代码保存在<build>/lib/Target/Iluvatar/IluvatarGenISel.inc中，该文件可以被C++源代码include进代码进行调用。

1. 在调用SelectCode()之前在Select()方法中通过函数重载提供自定义的匹配代码。在下面的代码中可以看到，N是要进行匹配的SDNode参数，在不同的case下，N等同于不同的待匹配的自定义指，令，如：ISD::SUBE、IluvatarISD::FMUL\_W\_CHAIN、IluvatarISD::FMUL\_W\_CHAIN、IluvatarISD::FMUL\_W\_CHAIN、ISD::BUILD\_VECTOR等。为匹配该指令，首先调用CurDAG->getMachineNode()来创建一个带有实际指令的节点，有时候为了匹配一个节点还需要创建多个节点如高位和低位节点SDNode \*Hi、SDNode \*Lo，该高位和低位节点也分别带有不同的实际指令。接下来，通过调用CurDAG->SelectNodeTo()函数创建一个特殊的指令节点，并将所有使用的原指令的结果改为指向新指令的结果。

Select()函数的具体实现如下所示：

|  |
| --- |
| void IluvatarDAGToDAGISel::Select(SDNode \*N) {  unsigned int Opc = N->getOpcode();  if (N->isMachineOpcode()) {  N->setNodeId(-1);  return; // Already selected.  }  switch (Opc) {  default:  break;  // We are selecting i64 ADD here instead of custom lower it during  // DAG legalization, so we can fold some i64 ADDs used for address  // calculation into the LOAD and STORE instructions.  case ISD::ADDC:  case ISD::ADDE:  case ISD::SUBC:  case ISD::SUBE: {  if (N->getValueType(0) != MVT::i64)  break;  SelectADD\_SUB\_I64(N);  return;  }  case IluvatarISD::FMUL\_W\_CHAIN: {  SelectFMUL\_W\_CHAIN(N);  return;  }  case IluvatarISD::FMA\_W\_CHAIN: {  SelectFMA\_W\_CHAIN(N);  return;  }  case ISD::SCALAR\_TO\_VECTOR:  case ISD::BUILD\_VECTOR: {  EVT VT = N->getValueType(0);  unsigned NumVectorElts = VT.getVectorNumElements();  if (VT.getScalarSizeInBits() == 16) {  if (Opc == ISD::BUILD\_VECTOR && NumVectorElts == 2) {  uint32\_t LHSVal, RHSVal;  if (getConstantValue(N->getOperand(0), LHSVal) &&  getConstantValue(N->getOperand(1), RHSVal)) {  uint32\_t K = LHSVal | (RHSVal << 16);  CurDAG->SelectNodeTo(  N, Iluvatar::SL\_MOV\_B32, VT,  CurDAG->getTargetConstant(K, SDLoc(N), MVT::i32));  return;  }  }  break;  } else if (VT.getScalarSizeInBits() == 8) {  if (Opc == ISD::BUILD\_VECTOR && NumVectorElts == 4) {  uint32\_t SVal0, SVal1, SVal2, SVal3;  if (getConstantValue(N->getOperand(0), SVal0) &&  getConstantValue(N->getOperand(1), SVal1) &&  getConstantValue(N->getOperand(2), SVal2) &&  getConstantValue(N->getOperand(3), SVal3)) {  uint32\_t K = SVal0 | (SVal1 << 8) | (SVal2 << 16) | (SVal3 << 24);  CurDAG->SelectNodeTo(  N, Iluvatar::SL\_MOV\_B32, VT,  CurDAG->getTargetConstant(K, SDLoc(N), MVT::i32));  return;  }  }  break;  }  assert(VT.getVectorElementType().bitsEq(MVT::i32));  unsigned RegClassID = selectSGPRVectorRegClassID(NumVectorElts);  SelectBuildVector(N, RegClassID);  return;  }  case ISD::BUILD\_PAIR: {  case ISD::BUILD\_PAIR: {  SDValue RC, SubReg0, SubReg1;  SDLoc DL(N);  if (N->getValueType(0) == MVT::i128) {  RC =  CurDAG->getTargetConstant(Iluvatar::SReg\_128RegClassID, DL, MVT::i32);  SubReg0 = CurDAG->getTargetConstant(Iluvatar::sub0\_sub1, DL, MVT::i32);  SubReg1 = CurDAG->getTargetConstant(Iluvatar::sub2\_sub3, DL, MVT::i32);  } else if (N->getValueType(0) == MVT::i64) {  RC = CurDAG->getTargetConstant(Iluvatar::SReg\_64RegClassID, DL, MVT::i32);  SubReg0 = CurDAG->getTargetConstant(Iluvatar::sub0, DL, MVT::i32);  SubReg1 = CurDAG->getTargetConstant(Iluvatar::sub1, DL, MVT::i32);  } else {  llvm\_unreachable("Unhandled value type for BUILD\_PAIR");  }  const SDValue Ops[] = {RC, N->getOperand(0), SubReg0, N->getOperand(1),  SubReg1};  ReplaceNode(N, CurDAG->getMachineNode(TargetOpcode::REG\_SEQUENCE, DL,  N->getValueType(0), Ops));  return;  }  case ISD::Constant:  case ISD::ConstantFP: {  if (N->getValueType(0).getSizeInBits() != 64 || isInlineImmediate(N))  break;  uint64\_t Imm;  if (ConstantFPSDNode \*FP = dyn\_cast<ConstantFPSDNode>(N))  Imm = FP->getValueAPF().bitcastToAPInt().getZExtValue();  else {  ConstantSDNode \*C = cast<ConstantSDNode>(N);  Imm = C->getZExtValue();  }  SDLoc DL(N);  SDNode \*Lo = CurDAG->getMachineNode(  Iluvatar::SL\_MOV\_B32, DL, MVT::i32,  CurDAG->getConstant(Imm & 0xFFFFFFFF, DL, MVT::i32));  SDNode \*Hi =  CurDAG->getMachineNode(Iluvatar::SL\_MOV\_B32, DL, MVT::i32,  CurDAG->getConstant(Imm >> 32, DL, MVT::i32));  const SDValue Ops[] = {  CurDAG->getTargetConstant(Iluvatar::SReg\_64RegClassID, DL, MVT::i32),  SDValue(Lo, 0), CurDAG->getTargetConstant(Iluvatar::sub0, DL, MVT::i32),  SDValue(Hi, 0),  CurDAG->getTargetConstant(Iluvatar::sub1, DL, MVT::i32)};  ReplaceNode(N, CurDAG->getMachineNode(TargetOpcode::REG\_SEQUENCE, DL,  N->getValueType(0), Ops));  return;  }  case ISD::LOAD: {  // SDValue Op = N->getOperand(0);  // EVT VT = Op.getValueType();  LoadSDNode \*Load = cast<LoadSDNode>(N);  unsigned Alignment = Load->getAlignment();  LLVM\_DEBUG(dbgs() << "Select Load alignment: " << Alignment << '\n');  if (Alignment < 4) {  MachineFunction &MF = CurDAG->getMachineFunction();  MachineMemOperand \*MMO = Load->getMemOperand();  unsigned MMOSize = MMO->getSize();  if ((MMO->getFlags() & MachineMemOperand::MOLoad) == 0) {  LLVM\_DEBUG(dbgs() << "load flag mismatch ");  }  const MachinePointerInfo &BasePtrInfo = MMO->getPointerInfo();  MachineMemOperand \*NewMMO = MF.getMachineMemOperand(BasePtrInfo,  MMO->getFlags(), MMOSize, Align(4));  Load->getMemOperand()->refineAlignment(NewMMO);  }  LLVM\_DEBUG(dbgs() << "New alignment: " << Load->getAlignment() << '\n');  break;  }  case ISD::STORE: {  // SDValue Op = N->getOperand(0);  // EVT VT = Op.getValueType();  StoreSDNode \*Store = cast<StoreSDNode>(N);  unsigned Alignment = Store->getAlignment();  LLVM\_DEBUG(dbgs() << "Select Store alignment: " << Alignment << '\n');  if (Alignment < 4) {  MachineFunction &MF = CurDAG->getMachineFunction();  MachineMemOperand \*MMO = Store->getMemOperand();  unsigned MMOSize = MMO->getSize();  if ((MMO->getFlags() & MachineMemOperand::MOStore) == 0) {  LLVM\_DEBUG(dbgs() << "store flag mismatch ");  }  const MachinePointerInfo &BasePtrInfo = MMO->getPointerInfo();  MachineMemOperand \*NewMMO = MF.getMachineMemOperand(BasePtrInfo,  MMO->getFlags(), MMOSize, Align(4));  Store->getMemOperand()->refineAlignment(NewMMO);  }  LLVM\_DEBUG(dbgs() << "New alignment: " << Store->getAlignment() << '\n');  break;  }  // case IluvatarISD::BFE\_I32:  case IluvatarISD::BFE\_U32: {  // There is a scalar version available, but unlike the vector version which  // has a separate operand for the offset and width, the scalar version packs  // the width and offset into a single operand. Try to move to the scalar  // version if the offsets are constant, so that we can try to keep extended  // loads of kernel arguments in SGPRs.  // TODO: Technically we could try to pattern match scalar bitshifts of  // dynamic values, but it's probably not useful.  // ConstantSDNode \*Offset = dyn\_cast<ConstantSDNode>(N->getOperand(1)); // if (!Offset)  // break;  // ConstantSDNode \*Width = dyn\_cast<ConstantSDNode>(N->getOperand(2));  // if (!Width)  // break;  // bool Signed = Opc == IluvatarISD::BFE\_I32;  // uint32\_t OffsetVal = Offset->getZExtValue();  // uint32\_t WidthVal = Width->getZExtValue();  // ReplaceNode(N, getS\_BFE(Signed ? Iluvatar::S\_BFE\_I32 : Iluvatar::S\_BFE\_U32,  // SDLoc(N), N->getOperand(0), OffsetVal, WidthVal));  // return;  ReplaceNode(N, CurDAG->getMachineNode(Iluvatar::ML\_LGC\_BFE\_B32, SDLoc(N), MVT::i32,  +------ 2 lines: N->getOperand(0), N->getOperand(1),  return;  }  // case IluvatarISD::MAD\_I64\_I32:  // case IluvatarISD::MAD\_U64\_U32: {  // SelectMAD\_64\_32(N);  // return;  // }  case ISD::CopyToReg: {  const BITargetLowering &Lowering =  \*static\_cast<const BITargetLowering \*>(getTargetLowering());  N = Lowering.legalizeTargetIndependentNode(N, \*CurDAG);  break;  }  // case ISD::AND:  // case ISD::SRL:  // case ISD::SRA:  // case ISD::SIGN\_EXTEND\_INREG:  // if (N->getValueType(0) != MVT::i32)  // break;  // SelectS\_BFE(N);  // return;  case ISD::BRCOND:  SelectBRCOND(N);  return;  case IluvatarISD::ATOMIC\_CMP\_SWAP:  SelectATOMIC\_CMP\_SWAP(N);  return;  case IluvatarISD::F2I:  SelectUnaaryOpRN(N, Iluvatar::ML\_CVT\_I32\_F32);  return;  case IluvatarISD::F2UI:  SelectUnaaryOpRN(N, Iluvatar::ML\_CVT\_U32\_F32);  return;  case IluvatarISD::RCPRP:  SelectUnaaryOpRN(N, Iluvatar::ML\_RCP\_F32);  return;  case IluvatarISD::SETMODEREG:  const SDValue Ops[] = { N->getOperand(0) };  ReplaceNode(N, CurDAG->getMachineNode(Iluvatar::SL\_REG\_WRITE\_MODE, SDLoc(N),  MVT::Other, Ops));  LLVM\_DEBUG(dbgs() << "SETMODEREG ");  // SDLoc SL(N);  // if (N->getGluedNode()) {  // SmallVector<SDValue, 3> RetOps;  //  // RetOps.push\_back(N->getOperand(1));  // RetOps.push\_back(N->getOperand(0));  // RetOps.push\_back(N->getOperand(2));  // CurDAG->SelectNodeTo(N, Iluvatar::ML\_CVT\_I32\_F32, N->getVTList(), RetOps);  // } else {  // CurDAG->SelectNodeTo(N, Iluvatar::ML\_CVT\_I32\_F32, MVT::f32, N->getOperand(0));  // }  return;  }  SelectCode(N);  } |

2.1.2 指令调度阶段

在完成指令选择后，编译器已将LLVM IR指令转换成了目标机器指令，编译器已经清除需要的所有目标机器指令。但是DAG图中并不包含没有相互依赖关系的指令间的顺序，所以还需要将DAG图指令转换成三地址指令形式，来确定基本快内的指令顺序。指令调度阶段负责尽可能多的优化指令级并行度的同时对指令进行排序，该阶段也称为前寄存器分配调度(Pre-register Allocation Scheduling)，将指令最终转换成了MachineInstr结构的三地址(三元地址)表示形式。每个三地址码指令，都可以被分解为四个[元组](https://baike.baidu.com/item/%E5%85%83%E7%BB%84" \t "/home/iluvatar/Documents\\x/_blank)（4-tuple）：（运算符，运算对象1，运算对象2，结果）。因为每个陈述都包含了三个变量，所以它被称为三地址码

该阶段称为前寄存器分配调度，因为该阶段是基于LLVM IR指令的无限寄存器假设对指令进行分配调度，该阶段没有考虑实际目标平台的寄存器资源。

在指令选择阶段之后，SelectionDAG结构将具有代表实际指令的节点，这些指令能直接运行在目标处理器上，然后，下一阶段包括对SelectionDAG节点(SDNodes)进行寄存器预分配调度。LLVM提供了几个不同的调度程序，它们都是ScheduleDAGSDNodes的子类，特定目标通过继承ScheduleDAGSDNodes类实现特定目标的指令调度类。指令调度器的实现代码为：llvm/lib/CodeGen/SelectionDAG/ScheduleDAGSDNodes.cpp。llc工具提供选项来选择编译时的调度器，-pre-RA-sched=<shceduler>选项可以设置编译时的调度器。LLVM常用的调度器schedule为：

1. list-ilp、list-hybrid、source和list-burr：这些选项引用由ScheduleDAGGRRList类实现的列表调度函数，相关源码路径为llvm/lib/CodeGen/SelectionDAG/ScheduleDAGRRList.cpp。
2. fast：ScheduleDAGFast类实现的次优但快速的调度器。
3. vliw-td：由ScheduleDAGVLIW实现的专门针对VLIW架构的调度程序。

default选项会自动为目标选择最佳的预定义调度程序，而linearize选项不执行任何调度。这些调度程序都可能根据详细的指令执行进程表和竞争关系等信息来更好的完成指令调度。

指令调度流程会在寄存器分配前指令一次，另外也会在寄存器分配完以后再执行一次指令调度，前一次指令调度对SelectionDAG或机器指令节点执行，后面一次是针对机器指令执行调度。而在代码生存器TableGen中,有三种不同的调度程序执行方式，两个在寄存器分配之前执行，一个在寄存器分配之后执行，第一个是对SelectionDAG节点执行，另外两个对机器指令执行。

1. 指令调度阶段

指令执行进程表是目标机器提供的，包括指令延迟和硬件流程信息的指令执行进程表，调度程序在调度决策期间使用这些属性来最大化吞吐量并避免性能损失。这些信息在每个目标的目录下的TableGen文件中继续描述，其文件名通常为<Target>Schedule.td。例如Iluvatar平台为：llvm/lib/Target/Iluvatar/IluvatarSchedule.td。

|  |
| --- |
| class HWWriteRes<SchedWrite write, list<ProcResourceKind> resources,  int latency, int rescycle> : WriteRes<write, resources> {  let Latency = latency;  let ResourceCycles = [rescycle];  }  class HWVALUWriteRes<SchedWrite write, int latency, int rlatency> :  HWWriteRes<write, [HWVALU], latency, rlatency>;  // The latency values are 1 / (operations / cycle) / 4.  multiclass BICommonWriteRes {  def : HWWriteRes<WriteBranch, [HWBranch], 12, 12>;  def : HWWriteRes<WriteLDS, [HWLMEM], 8, 2>; // Can be between 2 and 64  def : HWWriteRes<WriteSALU, [HWSALU], 1, 1>;  //def : HWWriteRes<WriteSMEM, [HWLGKM], 5>;  def : HWWriteRes<WriteSMEM, [HWSMEM], 10, 1>;  def : HWWriteRes<WriteVMEM, [HWVMEM], 24, 2>;  //def : HWWriteRes<WriteBarrier, [HWBranch], 500>; // XXX: Guessed ???  def : HWWriteRes<WriteBarrier, [HWBranch], 1, 1>; // XXX: Guessed ???  def : HWVALUWriteRes<Write32Bit, 1, 1>;  def : HWVALUWriteRes<Write64Bit, 2, 1>;  def : HWVALUWriteRes<WriteQuarterRate32, 4, 1>;  // def : HWMLALUWriteRes<WriteMLALU2Pass, 2, 1>;  def : HWVALUWriteRes<WriteVALU4Pass, 4, 1>;  def : HWVALUWriteRes<WriteVALU8Pass, 8, 1>;  def : HWVALUWriteRes<WriteVALU32Pass, 16, 1>;  }  def WriteSMEM : SchedWrite;  // Define a scheduler resource associated with a def operand.  class SchedWrite : SchedReadWrite; |

|  |
| --- |
| void DAGISEL\_CLASS\_COLONCOLON SelectCode(SDNode \*N)  52 {  53 // Some target values are emitted as 2 bytes, TARGET\_VAL handles  54 // this.  55 #define TARGET\_VAL(X) X & 255, unsigned(X) >> 8  56 static const unsigned char MatcherTable[] = { |

LLVM在llvm/include/llvm/Target/TargetItinerary.td中提供了ProcessorItineraries TableGen类，如下：

|  |
| --- |
| //===----------------------------------------------------------------------===//  // Processor itineraries - These values represent the set of all itinerary  // classes for a given chip set.  //  // Set property values to -1 to use the default.  // See InstrItineraryProps for comments and defaults.  class ProcessorItineraries<list<FuncUnit> fu, list<Bypass> bp,  list<InstrItinData> iid> {  list<FuncUnit> FU = fu;  list<Bypass> BP = bp;  list<InstrItinData> IID = iid;  // The packetizer automaton to use for this itinerary. By default all  // itineraries for a target are bundled up into the same automaton. This only  // works correctly when there are no conflicts in functional unit IDs between  // itineraries. For example, given two itineraries A<[SLOT\_A]>, B<[SLOT\_B]>,  // SLOT\_A and SLOT\_B will be assigned the same functional unit index, and  // the generated packetizer will confuse instructions referencing these slots.  //  // To avoid this, setting PacketizerNamespace to non-"" will cause this  // itinerary to be generated in a different automaton. The subtarget will need  // to declare a method "create##Namespace##DFAPacketizer()".  string PacketizerNamespace = "";  } |

编译目标可以为单一处理器架构或者处理器系列定义指令的执行进程表，所以，编译目标必须提供关于执行单元(FuncUnit)、管道旁路(Bypass)和指令执行进程表数据(InstrItinData)的列表。BI GPU没有定义芯片专用的执行进程表，ARMV6和AMD都有单独定义。

|  |
| --- |
| // Scheduling information derived from "ARM1176JZF-S Technical Reference Manual"  def ARMV6Itineraries : ProcessorItineraries<  [V6\_Pipe], [], [  // No operand cycles  InstrItinData<IIC\_iALUx , [InstrStage<1, [V6\_Pipe]>]>,  //Binary Instructions that produce a result  InstrItinData<IIC\_iALUi , [InstrStage<1, [V6\_Pipe]>], [2, 2]>,  InstrItinData<IIC\_iALUr , [InstrStage<1, [V6\_Pipe]>], [2, 2, 2]>,  InstrItinData<IIC\_iALUsi , [InstrStage<1, [V6\_Pipe]>], [2, 2, 1]>,  InstrItinData<IIC\_iALUsr , [InstrStage<2, [V6\_Pipe]>], [3, 3, 2, 1]>,  // Bitwise Instructions that produce a result  InstrItinData<IIC\_iBITi , [InstrStage<1, [V6\_Pipe]>], [2, 2]>,  InstrItinData<IIC\_iBITr , [InstrStage<1, [V6\_Pipe]>], [2, 2, 2]>,  InstrItinData<IIC\_iBITsi , [InstrStage<1, [V6\_Pipe]>], [2, 2, 1]>,  InstrItinData<IIC\_iBITsr , [InstrStage<2, [V6\_Pipe]>], [3, 3, 2, 1]>, |

SchedMachineModel由下面三类数据的子目标定义：

1. 粗粒度指令成本模型的基本属性。

2. 调度器读/写资源用于简单的每操作码成本模型。

3. 详细预订表的说明行程。

1.基本属性由SchedMachineModel类定义。Target hook允许子目标将操作码与这些属性关联。

2.pre-operand机器模型(machine model)可以通过以下方式的任意组合实现：

2.1.通过修改指令定义以从Sched继承，将per-oprand SchedReadWrite类型与指令关联。对于每个子目标，定义WriteRes和ReadAdvance，以将处理器资源和延迟与每个写入类型相关联。

2.2.在每个指令定义中，命名一个ItineraryClass类。对于每个子目标，定义ItinRW条目以将InvestrayClass类映射到per-oprand SchedReadWrite类型。与1中不同，这些类型可能通过定义SchedWrites和SchedAdvance，可以与子目标直接关联。

2.3.在子目标中，将SchederWrite类型映射到特定的操作码。这将覆盖指令定义的任何SchederWrite类或ItineraryClass类。与2中一样，子目标可以通过定义SchedWrites和SchedAdvance将资源与SchedWrite类型直接关联。

2.4.在目标或子目标中，定义SchedWriteVariant或SchederVariant以将一个SchederWrite类型映射到另一个SchederWrite类型序列(sequence)。这允许通过自定义C++代码动态选择指令的机器模型。它还允许独立于机器的SchederWrite类型映射到一系列依赖于机器的类型。

3.除了将指令映射到InvestrayClass类之外，还可以通过提供进程(Itineraries)来实现一个per-pipeline-stage机器模型。

2. 竞争检测

竞争识别器通过使用处理器的指令执行进度表中的信息来计算竞争关系。ScheduleHazardRecognizer类是实现竞争识别器的接口，而ScoreboardHazardRecognizer子类实现了基于记分板的竞争识别器(llvm/lib/CodeGen/ScoreboardHazardRecognizer.cpp)，这是LLVM默认的竞争识别器。(class ScoreboardHazardRecognizer : public ScheduleHazardRecognizer)

在TableGen无法表达特定约束的情况下，可以为编译目标提供自己的识别器，

3. 调度单元

调度程序在寄存器分配之前和之后运行。但是，SDNode指令表示形式仅在寄存器分配之前可用，而寄存器分配之后则使用MachineInstr类。为了处理SDNode节点和MachineInstrs，SUnit类在指令调度期间将底层指令抽象为调度单元。

llc可以通过使用选项-view-sunit-dags可以打印调度单元。

4.机器指令

寄存器分配器的处理对象是由MachineInstr类(MI,定义llvm/include/llvmCodeGen/MachineInstr.h)提供的指令表示形式。在指令调度之后运行的InstrEmitter流程将SDNode格式转换成MachineInstr格式。MachineInstr格式表示比LLVM IR指令更接近实际的目标指令，也更贴近底层。与SDNode格式和DAG形式不同，MI格式是程序的三地址(三元地址)表示，它是指令序列而不是DAG图，可以使编译器能够有效的进行调度，即决定每条指令的顺序。每个MI指令都包含一个操作码和一个操作数列表，其操作码是特定目标才能识别的指令。

MI类包含有关指令的重要信息元信息。它储存指令使用和定义的寄存器，并区分寄存器操作数和内存操作数，存储指令类型(分支，返回，调用和终止符等)，存储谓词。保留这些信息是非常重要的，即使在MI在较底层次也是如此，因为在InstrEmitter之后和代码输出之前运行的所有流程都依靠这些信息来执行分析。

2.1.3 寄存器分配

寄存器分配阶段就是将之前阶段基于无限寄存器集假设进行分配、调度指令转换成针对特定目标平台的有限寄存器集表示的指令，并且在需要是会产生溢出(spilling，即寄存器冲突，在前一个寄存器周期还未结束时再次调用该寄存器，此时会将寄存器的数据保存到memory，这样会影响运行的并行性和吞吐量)。

寄存器分配的基本任务是将数量不限的虚拟寄存器转换为物理有限的寄存器。由于编译目标的物理寄存器的数量有限，因此需要为一些虚拟寄存器分配对应的内存地址(即寄存器spilling)。

LLVM 寄存器分配器的另一个重要作用是解构IR的SSA(静态单赋值)形式，此时机器指令还包括原始LLVM IP复制的phi指令，SSA形式还原会正常形式需要复制指令(寄存器到内存的复制)代替phi指令。寄存器分配阶段进行正是分配寄存器和消除冗余复制操作等任务，因此解构SSA必须在寄存器分配之前。

LLVM有四种寄存器分配实现，使用llc的-regalloc=<regalloc\_name>选项选择他们，<regaaloc\_name>选项包括：pbqp、greedy、basic、fast。

pbqp：Partitioned Boolean Quadratic Problem被用于解决指令选择和寄存器分配问题。对于寄存器分配而言，PBQP算法的复杂度为VK^3，其中V是变量的数目，K是寄存器的数目。将寄存器分配问题转化成分区布尔二次方程问题。

greedy：是一个高效的全局优化寄存器分配算法，支持变量生存周期分割并最小化溢出次数，即对寄存器的利用保持greedy。

basic：是一个简单的分配器，并提供了扩展接口。

fast：是一种快速寄存器分配器，实现局部最优，该算法主要思想为将变量保存在寄存器中并尽可能地多次重用。

|  |
| --- |
| MachineInstr  机器指令类  (物理寄存器)  机器指令类  (虚拟寄存器)  流程  机器指令类  (虚拟寄存器)  机器指令类  (虚拟寄存器)  流程  机器指令类  (虚拟寄存器)  流程 |

1.寄存器合并

寄存器合并通过合并代码区间来删除多余的复制指令(copy)，该聚合是基于目标机器函数的流程，在RegisterCoalescer类中实现。基于机器函数的流程与基于每个函数的IR流程类似，但后者采取IR指令格式，前者采用基于机器指令的格式。在聚合过程中，函数joinAllIntervals()遍历复制指令的工作列表，函数joinCopy()通过复制机器指令创建CoalescerPair实例，尽可能的合并指令。

|  |
| --- |
| class RegisterCoalescer : public MachineFunctionPass,  private LiveRangeEdit::Delegate |

2.寄存器分配

3.虚拟寄存器重写

2.1.4 后寄存器指令调度阶段

该阶段实现与2.2指令调度阶段基本一直，前后两次指令调度的差异主要为，第二次指令调度阶段目标机器的寄存器信息已经在上述寄存器分配阶段分配到指令信息中，此时目标机器的寄存器信息已经可用，编译器需要根据硬件资源的竞争关系和不同寄存器的访问延迟差异性进一步提升生成代码的质量。

2.1.5 代码输出(Code Emission)

代码输出阶段将MachineInstr表示的指令转换成MCInst实例。MCInst实例更多的包含目标平台硬件资源信息，同时更适于汇编器和链接器输出汇编代码或者二进制表示的特定的目标代码格式。

2.2 后端代码结构介绍

LLVM采用模块解耦的方式组织软件架构，编译功能的实现由哥哥功能函数变成的函数库组成，所以后端实现代码也分散在不同的目录，不同的代码库中。代码生成的主要库在llvm/lib目录及其子目录CodeGen、MC、TableGen和Target中。

CodeGen目录包含所有通用代码生成算法的实现和头文件，包括指令选择，指令调度，寄存器分配以及相应的辅助分析函数。

MC目录包含汇编器，反汇编(松弛算法)和具体对象文件(ELF,COFF)等地层功能的实现。

TableGen目录包含TableGen工具的完整实现，该工具用于根据.td文件中的高层目标描述来生成C++代码。

Target目录包含每个目标的具体实现，每个目标平台在Target目录下生成一个子目录包含目标平台的代码生成和硬件资源定义的具体实现，比如Target/Iluvatar。在针对新的目标平台创建后端程序时，平台后端代码仅存在于Target及其子目录下。

如下表为Target/Iluvatar目录下中的代码文件名，以及对应的实现功能描述。

|  |  |
| --- | --- |
| 源文件名 | 代码功能描述 |
| Iluvatar.td | 定义机器特征、架构、硬件信息、扩展等 |
| IluvatarRegisterInfo.td | 定义平台寄存器和指令及格式定义 |
| IluvatarSchedule.td | 调度器定义 |
| IluvatarInstrInfo.td | 指令集信息 |
| BIInstrInfo.td | 平台特定型号目标定义 |
| IluvatarISelDAGToDAG.cpp | 指令选择优化 |
| BIMachineScheduler.cpp | 指令调度功能实现 |
| IluvatarRegisterInfo.cpp | 寄存器分配实现 |
| IluvatarMCInstLower.cpp | MCInst机器码指令降级 |
| IluvatarTargetMachine.cpp | 特定目标的属性信息 |
| IluvatarInstPrinter.cpp | 汇编代码输出 |
| IluvatarISelLowering.cpp | SelectionDAG节点降级 |

2.3 后端库介绍

后端代码实现采用模块解耦的方式，主要实现功能的代码根据相应功能编译成不同的代码。

llc是LLVM后端编译工具，llc非共享代码(tools/llc/llc.cpp)非常少，与其他LLVM工具一样，其大部分功能都以可重用的库的形式实现。llc的功能由代码生成器库来提供，包括通用的目标无关部分和特有的目标相关部分，这两部分对应的库在不同的目录下，CodeGen.a和Target.a。

目标无关的代码生成器库包括：

1. libLLVMAsmParse.a库包含解析汇编语言和实现汇编程序的代码实现；
2. libLLVMAsmPrinter.a库包含打印汇编语言和实现能生成汇编文件的后端的代码实现；
3. libLLVMCodeGen.a库包含代码生存算法,LLVM的目标无关的平台代码生成函数实现；
4. libLLVMMC.a库包含MCInst类的相关代码，用于表示LLVM允许的最低层次程序；
5. libLLVMMCDisassembler.a库包含用于实现反编译器的代码；libLLVMMCJIT.a库包含实现JIT代码生存器的实现；
6. libLLVMMCParser.a库包含MCAsmParser类的接口用于实现一个组建，以解析汇编文本并执行汇编器的部分工作；
7. libLLVMSelectionDAG.a库包含SelectionDAG和相关类；
8. libLLVMTraget.a库包含的接口允许目标无关函数请求目标相关函数，尽管这个函数本身是在其他(目标相关)库中实现的，Target库的实现是平台无关的代码生成实现接口的抽象接口和通用接口，特有目标平台通过继承相应类，重写相应的通用接口实现特有目标代码的实现。

目标相关的代码生成器库包括：

1. IluvatarAsmParser.a
2. IluvatarAsmPrinter.a

1. IluvatarCodeGen.a

1. IluvatarDesc.a

1. IluvatarDisassembler.a

1. IluvatarInfo.a

1. Iluvatar

1. Iluvatar

2.4 后端实现函数及类介绍

2.5 TableGen工具语法及td文件介绍

TableGen工具的主要目的是想在一个单独的位置声明目标机器相关信息(如，Ilu关于机器指令的描述信息)然后哦使用一个TableGen后端利用该描述信息完成某些特定功能，比如生成给予模式匹配的的指令选择算法。

TableGen工具会进行下列几部分的修改： 更改寄存器分配器以显示寄存器支持哪些类型；更改汇编器打印机以反映汇编器如何打印寄存器；更改汇编器解析器以反映如何在汇编语言代码中执行解析；更改反汇编器以直到寄存器如何编码。

现在TableGen广泛应用于描述目标机器的各种信息，包括指令格式、指令、寄存器、模式匹配DAG图、指令选择匹配顺序、调用惯例和目标CPU属性(可支持的指令集架构(ISA)特性和处理器系列)。

TableGen语言用于生成记录的定义和类组成。定义语句def用于实例化来自关键字class和multicla，下面是IluvatarRegisterInfo.td里定义Target属性相关信息的具体代码。

|  |
| --- |
| //td文件定义BIReg类型，  class BIReg <string n, bits<16> regIdx = 0> : Register<n>,  DwarfRegNum<[!cast<int>(HWEncoding)]> {  let Namespace = "Iluvatar";  // This is the not yet the complete register encoding. An additional  // bit is set for VGPRs.  let HWEncoding = regIdx;  }  //定义BIReg类型实例  // Special Registers  def TCR\_LO : BIReg<"tcr\_lo", 102>;  def TCR\_HI : BIReg<"tcr\_hi", 103>;  // Pseudo-registers: Used as placeholders during isel and immediately  // replaced, never seeing the verifier.  def PRIVATE\_RSRC\_REG : BIReg<"", 0>;  def FP\_REG : BIReg<"", 0>;  def SP\_REG : BIReg<"", 0>;  def SCRATCH\_WAVE\_OFFSET\_REG : BIReg<"", 0>;  // TCR for 64-bit instructions  def TCR : RegisterWithSubRegs<"tcr", [TCR\_LO, TCR\_HI]>,  DwarfRegAlias<TCR\_LO> {  let Namespace = "Iluvatar";  let SubRegIndices = [sub0, sub1];  let HWEncoding = 102;  } |

寄存器和寄存器类被定义在IluvatarRegisterInfo.td文件中。在定义指令时，寄存器类被用来将指令的操作数与特定觉得寄存器组关联起来。

|  |
| --- |
| def SReg\_512 : RegisterClass<"Iluvatar", [v16i32, v16f32], 32,  (add SGPR\_512, TTMP\_512)> {  // Requires 8 s\_mov\_b64 to copy  let CopyCost = 8;  let AllocationPriority = 12;  }  // Register class for all vector registers (VGPRs + Interploation Registers)  def VReg\_32 : RegisterClass<"Iluvatar", [i16, f16, i32, f32, v2f16, v2i16, i8, v2i8, v4i8], 32, (add VGPR\_32)> {  let Size = 32;  // Requires 1 v\_mov\_b32 to copy  let CopyCost = 1;  let AllocationPriority = 1;  }  def VReg\_64 : RegisterClass<"Iluvatar", [i64, v2i32, v2f32, v4f16, v4i16], 32, (add VGPR\_64)> {  let Size = 64;  // Requires 2 v\_mov\_b32 to copy  let CopyCost = 2;  let AllocationPriority = 2;  }  def VReg\_128 : RegisterClass<"Iluvatar", [v4i32, v4f32, v2i64], 32, (add VGPR\_128)> {  let Size = 128;  // Requires 4 v\_mov\_b32 to copy  let CopyCost = 4;  let AllocationPriority = 4;  }  def VReg\_256 : RegisterClass<"Iluvatar", [v8i32, v8f32], 32, (add VGPR\_256)> {  let Size = 256;  let CopyCost = 8;  let AllocationPriority = 5;  }  def V107\_B64 : IluvatarReg<"v[107:108]">, DwarfRegNum<[591]> {  string AsmName = "v[107:108]";  list<Register> SubRegs = [V107\_B32, V108\_B32];  let HWEncoding{8-0} = { 1, 0, 1, 1, 0, 1, 0, 1, 1 };  list<SubRegIndex> SubRegIndices = [sub0, sub1];  }  def VRF\_B64 : RegisterClass<"Iluvatar", [i64, v2i32, v2f32, v4i16, v4f16], 32, (add V0\_B64, V1\_B64, V2\_B64, V3\_B64, V4\_B64, V5\_B64, V6\_B64, V7\_B64, V8\_B64, V9\_B64, V10\_B64, V11\_B64, V12\_B64, V13\_B64, V14\_B64, V15\_B64, V16\_B64, V17\_B64, V18\_B64, V19\_B64, V20\_B64, V21\_B64, V22\_B64, V23\_B64, V24\_B64, V25\_B64, V26\_B64, V27\_B64, V28\_B64, V29\_B64, V30\_B64, V31\_B64, V32\_B64, V33\_B64, V34\_B64, V35\_B64, V36\_B64, V37\_B64, V38\_B64, V39\_B64, V40\_B64, V41\_B64, V42\_B64, V43\_B64, V44\_B64, V45\_B64, V46\_B64, V47\_B64, V48\_B64, V49\_B64, V50\_B64, V51\_B64, V52\_B64, V53\_B64, V54\_B64, V55\_B64, V56\_B64, V57\_B64, V58\_B64, V59\_B64, V60\_B64, V61\_B64, V62\_B64, V63\_B64, V64\_B64, V65\_B64, V66\_B64, V67\_B64, V68\_B64, V69\_B64, V70\_B64, V71\_B64, V72\_B64, V73\_B64, V74\_B64, V75\_B64, V76\_B64, V77\_B64, V78\_B64, V79\_B64, V80\_B64, V81\_B64, V82\_B64, V83\_B64, V84\_B64, V85\_B64, V86\_B64, V87\_B64, V88\_B64, V89\_B64, V90\_B64, V91\_B64, V92\_B64, V93\_B64, V94\_B64, V95\_B64, V96\_B64, V97\_B64, V98\_B64, V99\_B64, V100\_B64, V101\_B64, V102\_B64, V103\_B64, V104\_B64, V105\_B64, V106\_B64, V107\_B64, V108\_B64, V109\_B64, V110\_B64, V111\_B64, V112\_B64, V113\_B64, V114\_B64, V115\_B64, V116\_B64, V117\_B64, V118\_B64, V119\_B64, V120\_B64, V121\_B64, V122\_B64, V123\_B64, V124\_B64, V125\_B64, V126\_B64, V127\_B64, V128\_B64, V129\_B64, V130\_B64, V131\_B64, V132\_B64, V133\_B64, V134\_B64, V135\_B64, V136\_B64, V137\_B64, V138\_B64, V139\_B64, V140\_B64, V141\_B64, V142\_B64, V143\_B64, V144\_B64, V145\_B64, V146\_B64, V147\_B64, V148\_B64, V149\_B64, V150\_B64, V151\_B64, V152\_B64, V153\_B64, V154\_B64, V155\_B64, V156\_B64, V157\_B64, V158\_B64, V159\_B64, V160\_B64, V161\_B64, V162\_B64, V163\_B64, V164\_B64, V165\_B64, V166\_B64, V167\_B64, V168\_B64, V169\_B64, V170\_B64, V171\_B64, V172\_B64, V173\_B64, V174\_B64, V175\_B64, V176\_B64, V177\_B64, V178\_B64, V179\_B64, V180\_B64, V181\_B64, V182\_B64, V183\_B64, V184\_B64, V185\_B64, V186\_B64, V187\_B64, V188\_B64, V189\_B64, V190\_B64, V191\_B64, V192\_B64, V193\_B64, V194\_B64, V195\_B64, V196\_B64, V197\_B64, V198\_B64, V199\_B64, V200\_B64, V201\_B64, V202\_B64, V203\_B64, V204\_B64, V205\_B64, V206\_B64, V207\_B64, V208\_B64, V209\_B64, V210\_B64, V211\_B64, V212\_B64, V213\_B64, V214\_B64, V215\_B64, V216\_B64, V217\_B64, V218\_B64, V219\_B64, V220\_B64, V221\_B64, V222\_B64, V223\_B64, V224\_B64, V225\_B64, V226\_B64, V227\_B64, V228\_B64, V229\_B64, V230\_B64, V231\_B64, V232\_B64, V233\_B64, V234\_B64, V235\_B64, V236\_B64, V237\_B64, V238\_B64, V239\_B64, V240\_B64, V241\_B64, V242\_B64, V243\_B64, V244\_B64, V245\_B64, V246\_B64, V247\_B64, V248\_B64, V249\_B64, V250\_B64, V251\_B64, V252\_B64, V253\_B64, V254\_B64)> ; |

let结构用于定定义一个额外的字段。由上述寄存器类型定义可知，寄存器的定义用于存放每个寄存器的名称/编号和子寄存器列表。可以通过查看TableGen工具通过解析IluvatarRegisterInfo.td生成的IluvatarGenRegisterInfo.inc，该文件会通过include的方式包含到IluvatarRegisterInfo.cpp中，将寄存器信息导入到CPP中。IluvatarGenRegisterInfo.inc文件还包括寄存器与数字的对应关系。

指令格式在IluvatarInstrFormats.td中定义， 而指令在IluvatarInstrInfo.td中定义，指令格式包含了二进制形式下指令中不同的编码字段，而每个指令记录都对应一条指令。可以将常见的特征(相似数据处理指令的常见编码)提取出来，创建用于派生出指令记录的TablrGen中间的指令。每个指令和指令格式都必须是定义在include/llvm/Target/Target.td中的Instruction TableGen类的直接或间接子类。

|  |
| --- |
| class InstructionEncoding {  // Size of encoded instruction.  int Size;  // The "namespace" in which this instruction exists, on targets like ARM  // which multiple ISA namespaces exist.  string DecoderNamespace = "";  // List of predicates which will be turned into isel matching code.  list<Predicate> Predicates = [];  string DecoderMethod = "";  // Is the instruction decoder method able to completely determine if the  // given instruction is valid or not. If the TableGen definition of the  // instruction specifies bitpattern A??B where A and B are static bits, the  // hasCompleteDecoder flag says whether the decoder method fully handles the  // ?? space, i.e. if it is a final arbiter for the instruction validity.  // If not then the decoder attempts to continue decoding when the decoder  // method fails.  //  // This allows to handle situations where the encoding is not fully  // orthogonal. Example:  // \* InstA with bitpattern 0b0000????,  // \* InstB with bitpattern 0b000000?? but the associated decoder method  // DecodeInstB() returns Fail when ?? is 0b00 or 0b11.  //  // The decoder tries to decode a bitpattern that matches both InstA and  // InstB bitpatterns first as InstB (because it is the most specific  // encoding). In the default case (hasCompleteDecoder = 1), when  // DecodeInstB() returns Fail the bitpattern gets rejected. By setting  // hasCompleteDecoder = 0 in InstB, the decoder is informed that  // DecodeInstB() is not able to determine if all possible values of ?? are  // valid or not. If DecodeInstB() returns Fail the decoder will attempt to  // decode the bitpattern as InstA too.  bit hasCompleteDecoder = true;  }  class Instruction : InstructionEncoding {  string Namespace = "";  dag OutOperandList; // An dag containing the MI def operand list.  dag InOperandList; // An dag containing the MI use operand list.  string AsmString = ""; // The .s format to print the instruction with.  // Allows specifying a canonical InstructionEncoding by HwMode. If non-empty,  // the Inst member of this Instruction is ignored.  EncodingByHwMode EncodingInfos;  // Pattern - Set to the DAG pattern for this instruction, if we know of one,  // otherwise, uninitialized.  list<dag> Pattern;  // The follow state will eventually be inferred automatically from the  // instruction pattern.  list<Register> Uses = []; // Default to using no non-operand registers  list<Register> Defs = []; // Default to modifying no non-operand registers  // Predicates - List of predicates which will be turned into isel matching  // code.  list<Predicate> Predicates = [];  // Size - Size of encoded instruction, or zero if the size cannot be determined  // from the opcode.  int Size = 0;  // Code size, for instruction selection.  // FIXME: What does this actually mean?  int CodeSize = 0;  // Added complexity passed onto matching pattern.  int AddedComplexity = 0;  // Indicates if this is a pre-isel opcode that should be  // legalized/regbankselected/selected.  bit isPreISelOpcode = false;  // These bits capture information about the high-level semantics of the  // instruction.  bit isReturn = false; // Is this instruction a return instruction?  bit isBranch = false; // Is this instruction a branch instruction?  bit isEHScopeReturn = false; // Does this instruction end an EH scope?  bit isIndirectBranch = false; // Is this instruction an indirect branch?  bit isCompare = false; // Is this instruction a comparison instruction?  bit isMoveImm = false; // Is this instruction a move immediate instruction?  bit isMoveReg = false; // Is this instruction a move register instruction?  bit isBitcast = false; // Is this instruction a bitcast instruction?  bit isSelect = false; // Is this instruction a select instruction?  bit isBarrier = false; // Can control flow fall through this instruction?  bit isCall = false; // Is this instruction a call instruction?  bit isAdd = false; // Is this instruction an add instruction?  bit isTrap = false; // Is this instruction a trap instruction?  bit canFoldAsLoad = false; // Can this be folded as a simple memory operand?  bit mayLoad = ?; // Is it possible for this inst to read memory?  bit mayStore = ?; // Is it possible for this inst to write memory?  bit mayRaiseFPException = false; // Can this raise a floating-point exception?  bit isConvertibleToThreeAddress = false; // Can this 2-addr instruction promote?  bit isCommutable = false; // Is this 3 operand instruction commutable?  bit isTerminator = false; // Is this part of the terminator for a basic block?  bit isReMaterializable = false; // Is this instruction re-materializable?  bit isPredicable = false; // 1 means this instruction is predicable  // even if it does not have any operand  // tablegen can identify as a predicate  bit isUnpredicable = false; // 1 means this instruction is not predicable  // even if it \_does\_ have a predicate operand  bit hasDelaySlot = false; // Does this instruction have an delay slot?  bit usesCustomInserter = false; // Pseudo instr needing special help.  bit hasPostISelHook = false; // To be \*adjusted\* after isel by target hook.  bit hasCtrlDep = false; // Does this instruction r/w ctrl-flow chains?  bit isNotDuplicable = false; // Is it unsafe to duplicate this instruction?  bit isConvergent = false; // Is this instruction convergent?  bit isAuthenticated = false; // Does this instruction authenticate a pointer?  bit isAsCheapAsAMove = false; // As cheap (or cheaper) than a move instruction.  bit hasExtraSrcRegAllocReq = false; // Sources have special regalloc requirement?  bit hasExtraDefRegAllocReq = false; // Defs have special regalloc requirement?  bit isRegSequence = false; // Is this instruction a kind of reg sequence?  // If so, make sure to override  // TargetInstrInfo::getRegSequenceLikeInputs.  bit isPseudo = false; // Is this instruction a pseudo-instruction?  // If so, won't have encoding information for  // the [MC]CodeEmitter stuff.  bit isExtractSubreg = false; // Is this instruction a kind of extract subreg?  // If so, make sure to override  // TargetInstrInfo::getExtractSubregLikeInputs.  bit isInsertSubreg = false; // Is this instruction a kind of insert subreg?  // If so, make sure to override  // TargetInstrInfo::getInsertSubregLikeInputs.  bit variadicOpsAreDefs = false; // Are variadic operands definitions?  // Does the instruction have side effects that are not captured by any  // operands of the instruction or other flags?  bit hasSideEffects = ?;  // Is this instruction a "real" instruction (with a distinct machine  // encoding), or is it a pseudo instruction used for codegen modeling  // purposes.  // FIXME: For now this is distinct from isPseudo, above, as code-gen-only  // instructions can (and often do) still have encoding information  // associated with them. Once we've migrated all of them over to true  // pseudo-instructions that are lowered to real instructions prior to  // the printer/emitter, we can remove this attribute and just use isPseudo.  //  // The intended use is:  // isPseudo: Does not have encoding information and should be expanded,  // at the latest, during lowering to MCInst.  //  // isCodeGenOnly: Does have encoding information and can go through to the  // CodeEmitter unchanged, but duplicates a canonical instruction  // definition's encoding and should be ignored when constructing the  // assembler match tables.  bit isCodeGenOnly = false;  // Is this instruction a pseudo instruction for use by the assembler parser.  bit isAsmParserOnly = false;  // This instruction is not expected to be queried for scheduling latencies  // and therefore needs no scheduling information even for a complete  // scheduling model.  bit hasNoSchedulingInfo = false;  InstrItinClass Itinerary = NoItinerary;// Execution steps used for scheduling.  // Scheduling information from TargetSchedule.td.  list<SchedReadWrite> SchedRW;  string Constraints = ""; // OperandConstraint, e.g. $src = $dst.  /// DisableEncoding - List of operand names (e.g. "$op1,$op2") that should not  /// be encoded into the output machineinstr.  string DisableEncoding = "";  string PostEncoderMethod = "";  /// Target-specific flags. This becomes the TSFlags field in TargetInstrDesc.  bits<64> TSFlags = 0;  ///@name Assembler Parser Support  ///@{    string AsmMatchConverter = "";  /// TwoOperandAliasConstraint - Enable TableGen to auto-generate a  /// TwoOperandAliasConstraint - Enable TableGen to auto-generate a  /// two-operand matcher inst-alias for a three operand instruction.  /// For example, the arm instruction "add r3, r3, r5" can be written  /// as "add r3, r5". The constraint is of the same form as a tied-operand  /// constraint. For example, "$Rn = $Rd".  string TwoOperandAliasConstraint = "";  /// Assembler variant name to use for this instruction. If specified then  /// instruction will be presented only in MatchTable for this variant. If  /// not specified then assembler variants will be determined based on  /// AsmString  string AsmVariantName = "";  ///@}  /// UseNamedOperandTable - If set, the operand indices of this instruction  /// can be queried via the getNamedOperandIdx() function which is generated  /// by TableGen.  bit UseNamedOperandTable = false;  /// Should generate helper functions that help you to map a logical operand's  /// index to the underlying MIOperand's index.  /// In most architectures logical operand indicies are equal to  /// MIOperand indicies, but for some CISC architectures, a logical operand  /// might be consist of multiple MIOperand (e.g. a logical operand that  /// uses complex address mode).  bit UseLogicalOperandMappings = false;  /// Should FastISel ignore this instruction. For certain ISAs, they have  /// instructions which map to the same ISD Opcode, value type operands and  /// instruction selection predicates. FastISel cannot handle such cases, but  /// SelectionDAG can.  bit FastISelShouldIgnore = false;  } |

目标平台继承Instruction类，定义自己的指令集，dag是一种特殊的TableGen类型，用于保存SelectionDAG节点，这些节点代表在指令选择阶段的操作码、寄存器或常数。上述主要字段的具体含义：

OutOperandList，储存结果节点，后端可以根据该字段识别代表指令结果的DAG节点。

InOperandList，保存输出节点。

AsmString，表示汇编指令的字符串。

Parttern，是指令选择期间将用于进行模式匹配的dag对象列表。如果模式匹配成功，则指令选择阶段用该指令替换匹配的节点。

Uses和Defs字段分别代表执行此指令隐式使用和定义的寄存器的列表。比如，返回指令隐式的使用返回寄存器，调用指令隐式的定义返回地质寄存器。

Predicates，储存在指令选择阶段尝试匹配指令之前被检查的先决条件的列表。

TableGen后端利用指令记录的信息来完成其功能，从相同的指令记录生成不同的.inc文件，IluvatarGenDAGISel.inc文件使用指令记录中patterns字段的信息来输出相应的代码，以选择SelectionDAG数据结构的指令，包含在IluvatarISelDAGtoDAG.cpp文件中。

IluvatarGenInstrInfo.inc是众多指令描述表的一员，该文件包含目标机器的所有指令的枚举类型，IluvatarGenAsmWriter.inc用于打印每个指令汇编码的字符串。IluvatarAsmPrinter.cpp包含该inc文件。

IluvatarGenCodeEmitter.inc包含用于将每个指令输出为二进制码的代码最终被用于生成目标文件爱你中的机器码，包含在IluvatarCodeEmitter.cpp。

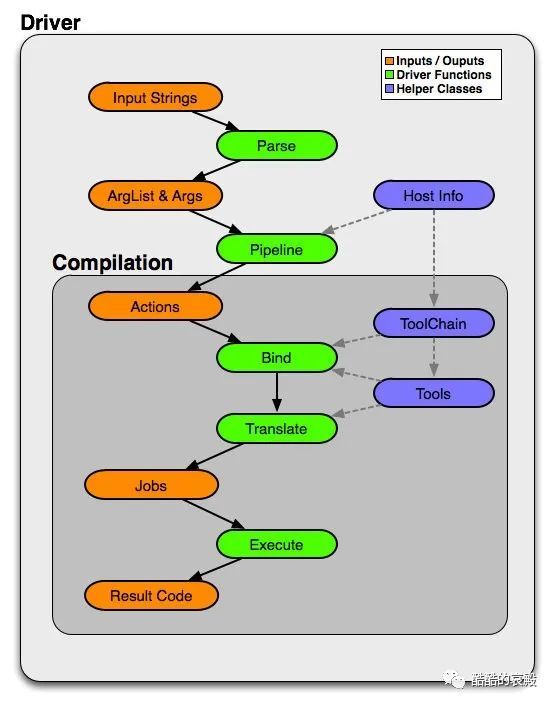
IluvatarGenAsmMatcher.inc文件实现目标指令汇编器所谓解析器。

2.6 添加自定义流程(pass)介绍

1. 编译驱动器ClangDriver及编译流程研究介绍

在不同的场景下，clang 可能具有不同的含义：

1. clang driver：编译器驱动程序，负责根据简单的参数生成更加负杂的参数
2. 编译器前端(在 clang 库实现)：专指编译器生成中间代码的过程（preprocessor 和 compiler）
3. 实际的编译器(在 clang -cc1 中实现):clang -cc1 通过 LLVM 库和其它工具实现了编译前端、后端、汇编等工作



3.1 ClangDriver 介绍

clang 先会以 clang driver 模式运行，随后组装 clang -cc1 和 ld 两个命令行，并进行下一步处理，clang driver 自身不负责对源码进行编译，负责拼接编译器命令和 ld 命令。clang Driver 的处理逻辑分为以下几步：

1.Parse: Option Parsing：解析传入的参数

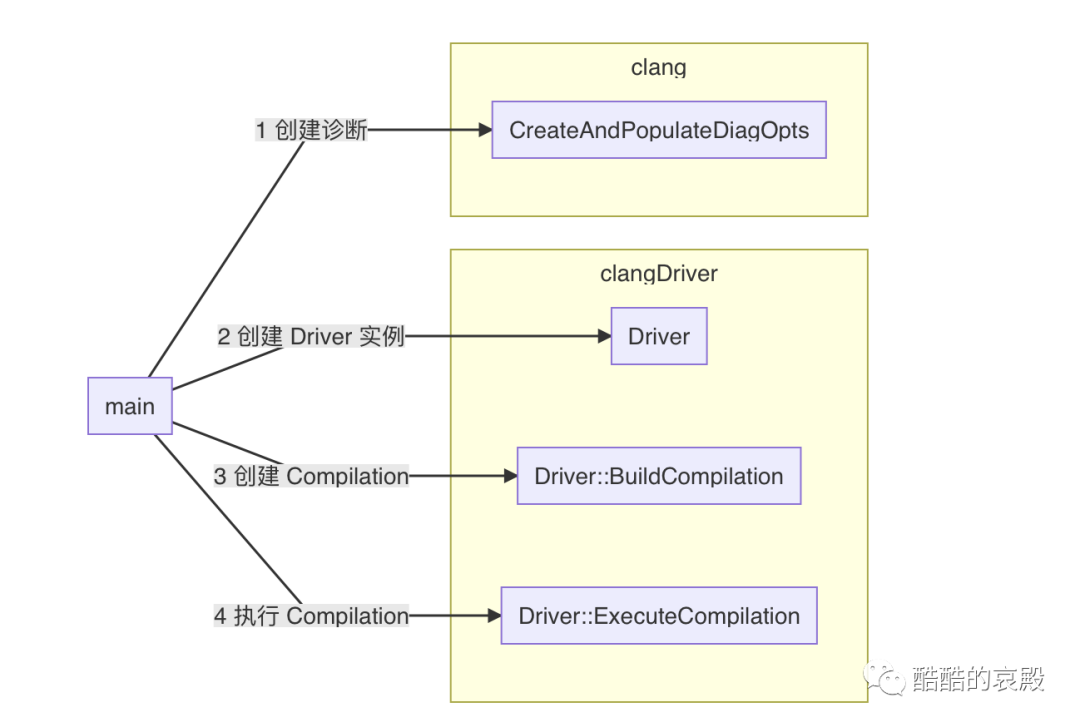
2.Pipeline: Compilation Action Construction：根据每个输入的文件和类型，组建 action（比如 PreprocessJobAction）,可以通过 clang -ccc-print-phases 可以查看需要处理的 action。

3.Bind: Tool & Filename Selection：根据 action 选择对应的工具和文件名信息，通过 clang -ccc-print-bindings 可以查看对应的工具和文件名信息。

4.Translate: Tool Specific Argument Translation：根据输入的参数转为不同 tool 的参数

5.Execute：调用不同的 tool 执行任务，该步骤会通过创建子进程方式调用tool。

首先，我们以 对整个流程做一个简单的介绍：第一步：clang 会以 driver 模式被调用；第二步，clang driver 会根据传入的 main.m 参数构建为两个 Job，第一个任务是编译任务，clang 接收 -cc1 参数后会以编译器的身份执行编译任务，输入文件是 xxx.m，输出文件是 xxx.o 对象文件，第二个任务是链接任务，ld 会将 xxx.o 链接为 可执行文件；最后，会根据上面的两个 Job 创建新的进程执行上面的两个 Job。



main 函数会先创建诊断 (DiagnosticsEngine)实例 诊断是编译器与开发者进行交互的重要部分。编译器通过诊断可以提供错误、警告或建议。入口在clang/tools/driver/driver.cpp。

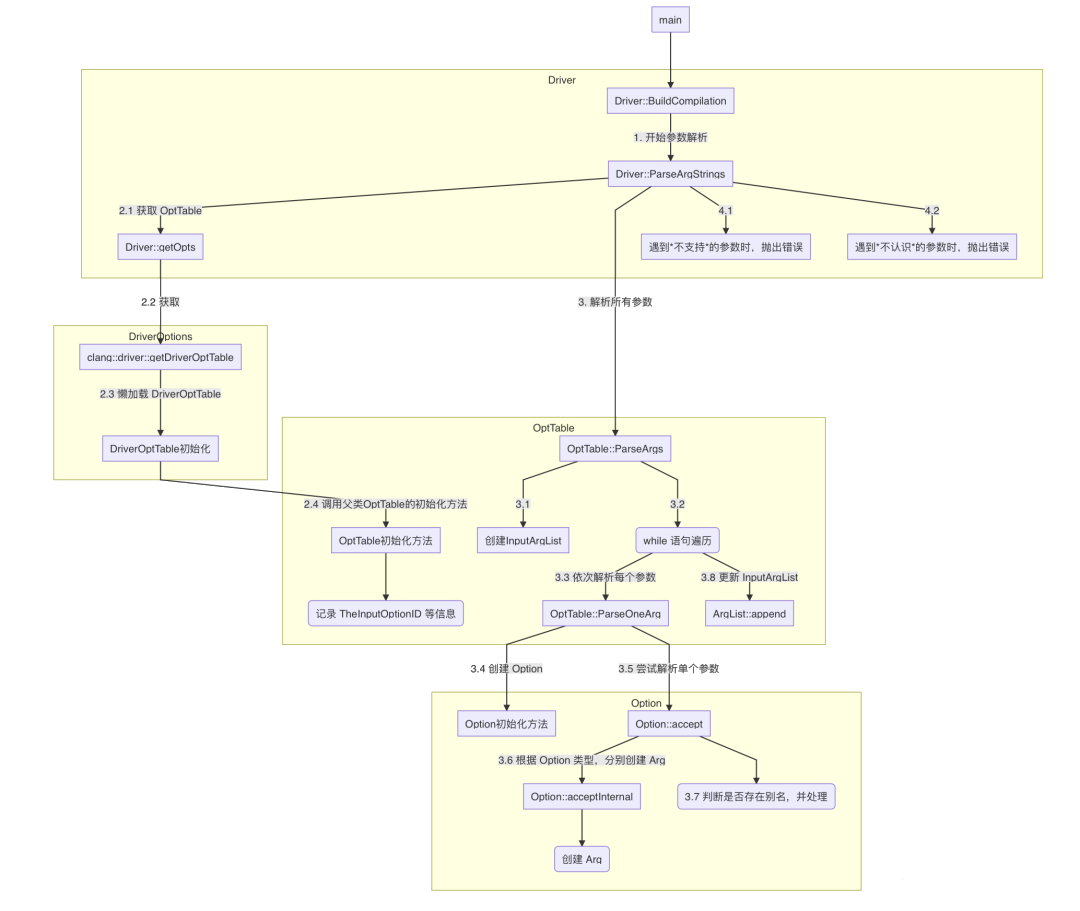
随后，开始创建 Driver (clang::driver::Driver) 的实例：TheDriver ，TheDriver 负责后续的 clang driver 相关任务。

通过 Driver 的 BuildCompilation 方法生成需要执行的命令。

当 Driver 构造完 Jobs 后，会通过 Driver 的 ExecuteCompilation 方法执行命令。

3.2 clang driver 参数Parse

为了控制 clang 的运行，clang 必须支持不同的参数对各种行为进行控制，所以，clang driver 启动后的第一个主要任务就是参数解析。



常用结构体或函数：

1. Info[] 是保存了预定义的各种 Option 信息的结构体。比如 -v 参数的帮助信息是 Show commands to run and use verbose output。
2. Option[] 是持有 Info 和 OptTable，提供了一些封装好的方法，比如通过 OptionClass getKind() 方法暴露 Info 的类型。
3. Arg[] 持有了 Option 和其它命令行参数信息，比如 -arch armv64 的 arm64 会被保存到 Arg
4. OptTable[] 提供解析参数，并懒加载创建 Option 的相关方法
5. InputArgList[] 持有了输入的原始参数和解析后的参数列表
6. DriverOptTable[] 记录了 clang driver 相关的 Info 信息，是 OptTable 的子类

3.2.1 DriverOptTable

DriverOptTable 记录了 clang driver 相关的 Info 信息，是 OptTable 的子类。

DriverOptions 模块提供了函数 const llvm::opt::OptTable &clang::driver::getDriverOptTable() 可以获取 clang driver 支持的所有参数信息 DriverOptTable 初始化时依赖的 InfoTable 参数是通过 clang/Driver/Options.inc 生成的。

因为 DriverOptTable 继承自 OptTable，所以，这里会触发 OptTable 的初始化方法。

OptTable 的初始化时，会记录一些关键的 ID，用于后续使用，比如 TheInputOptionID。

同时，会通过 PrefixChars 和 PrefixesUnion 记录合法的参数前缀，用于后续的快速参数合法性判断，比如 -v 参数的前缀是 -。

3.2.2 Driver::ParseArgStrings

Driver::ParseArgStrings 方法的作用是将字符串数组解析为 ArgList，并做相关的校验具体流程如下：

1. 调用 Driver::getOpts 获取 clang driver 支持的所有参数 Info
2. 调用 ParseArgs 解析命令行参数
3. 对解析到的命令行参数进行判断，检测到 不支持 或者 未知 的参数时，会抛出异常

如何区分不支持或者不认识的参数：

1. clang driver 不支持 的参数，都可以通过 Options.td 文件查到 以 -pass-exit-codes 为例，gcc支持该参数，但是clang不支持此参数。
2. 不认识的参数就是类似于-test这种，开发者随意拼写的参数。

3.2.3 ParseArgs

OptTable::ParseArgs 方法负责将字符串数组解析为 ArgList具体流程如下：

1. 先初始化 InputArgList 的实例，并存储原始的入参信息
2. 通过 while 对原始参数字符串进行遍历，并通过 OptTable::ParseOneArg 方法将所有的原始参数字符串解析为 Arg 的实例
3. 最后 Args 会持有所有的解析后的参数

3.2.4 ParseOneArg

OptTable::ParseOneArg 方法负责解析单个参数。

具体流程如下：

1. 先移除参数的前缀，并通过 std::lower\_bound 查找第一个前缀匹配的 Info 比如，-arch 会变成 arch。
2. 根据 Info 初始化 Option 持有参数信息。
3. 通过Option::accept方法校验参数是否正常，参数正常时直接返回。
4. 如果没有找到合适的参数，再判断参数是否以/开头，如果开始，会把参数当做源码文件进行处理，其它情况下，会当做参数当做未知参数进行下一步处理。
5. std::lower\_bound 会依赖下面两个方法查找第一个前缀匹配的参数。
6. Option::accept 方法会依次进行以下处理，比如，-fembed-bitcode-marker 就是 -fembed-bitcode=marker 参数的别名，两个参数的意义完全相同。先转发到 Option::acceptInternal 方法进行参数校验，判断解析到的参数是否属于别名，如果别名，会进行特殊处理。
7. Option::acceptInternal 方法会根据 Option 的类型进行处理并生成 Arg 实例。 因为 -arch 的类型是 SeparateClass ，所以，会将下一个原始参数字符串（arm64）当做 value 进行处理 类型示例Separate-arch arm64Flag-vJoined-fembed-bitcode=marker。

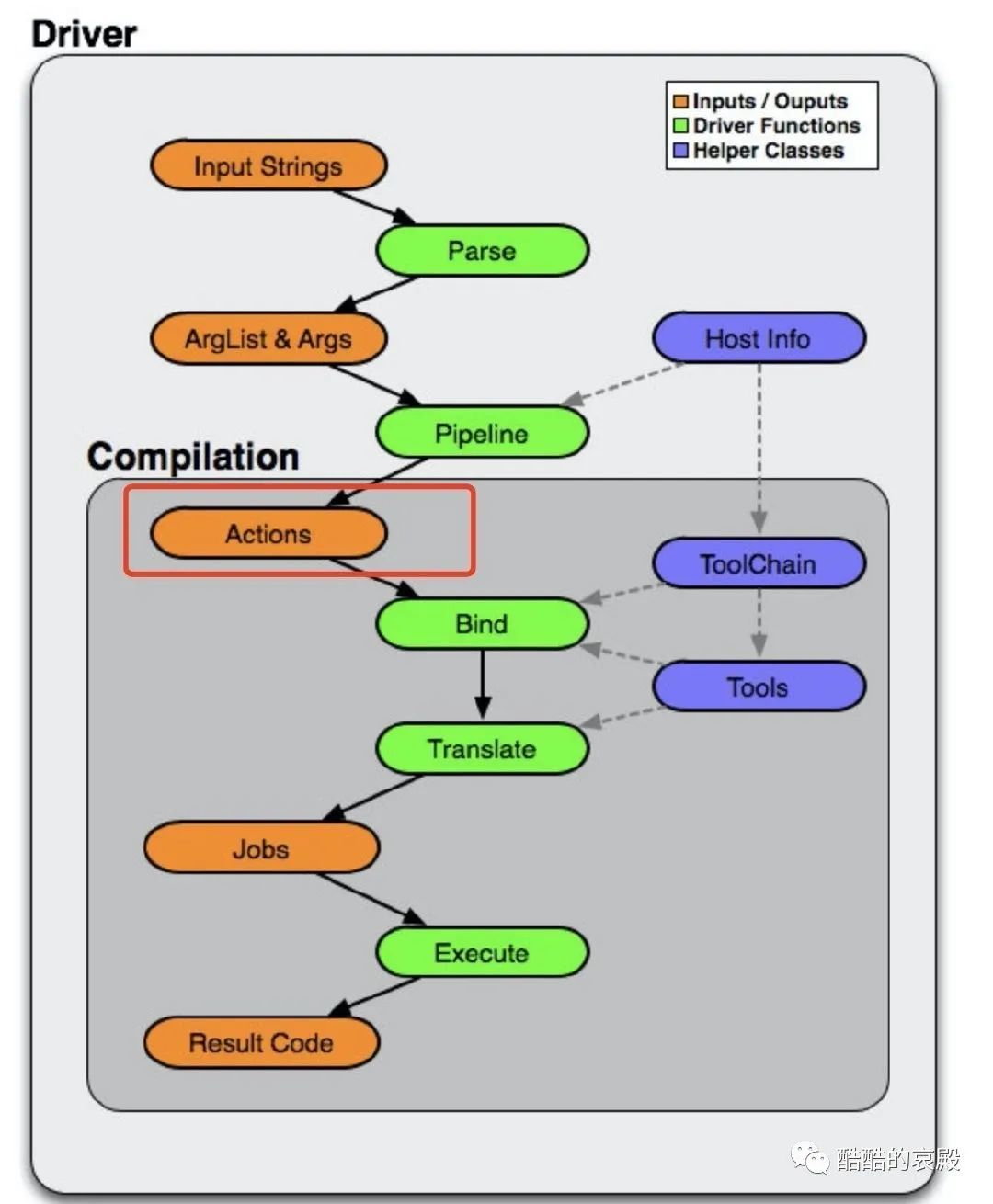
3.3构建Compilation

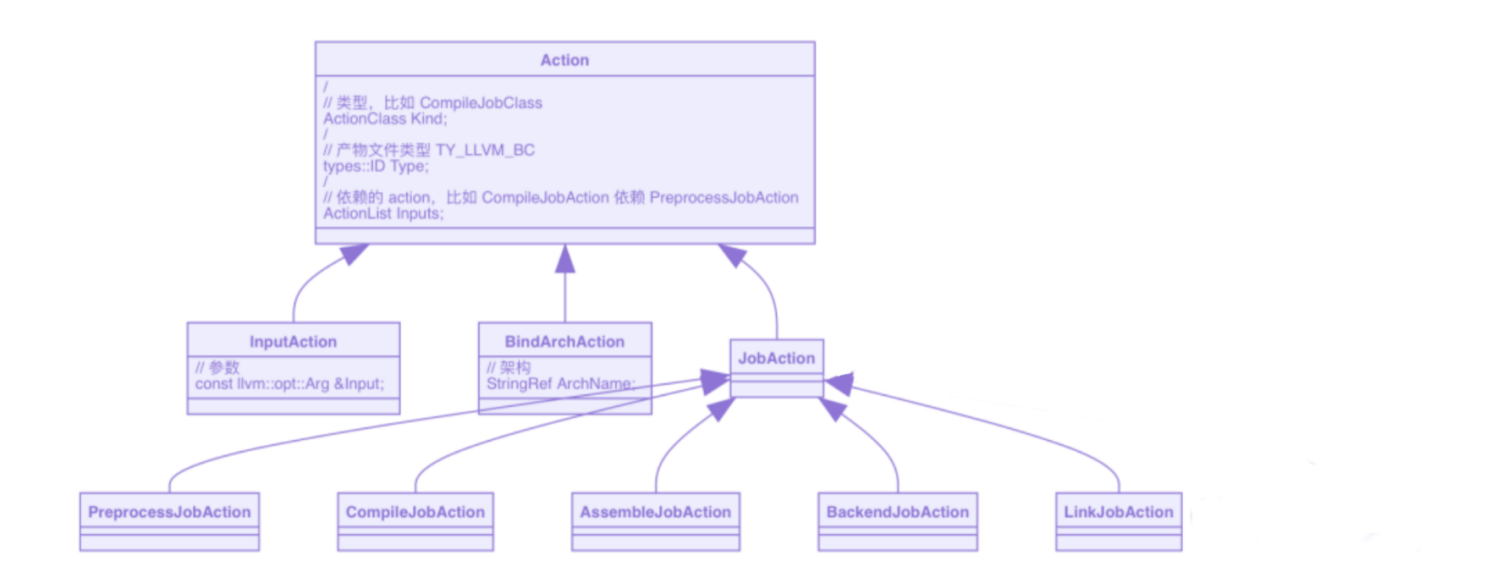
下面，我们再对 BuildCompilation 的流程进一步拆解。

BuildCompilation 方法主要包含以下步骤：

1. 调用 ParseArgStrings 函数 处理程序接收的参数 和 对配置文件解析
2. 通过 computeTargetTriple 函数获取 triple 并通过 getToolChain 函数获取对应的 ToolChain。
3. getToolChain 函数会返回根据 triple 的系统信息返回 Clang 的实例
4. 创建 Commpilation 持有实例
5. 通过 BuildInputs 函数获取输入文件，clang driver 支持一次性编译多个文件，比如下面的命令可以同时编译 main.m 和 test.m 两个源码文件。BuildInputs 方法会遍历所有的参数，并筛选 Option::InputClass 类型的参数，最后会调用函数 types::ID types::lookupTypeForExtension(llvm::StringRef Ext) 获取对应的 types::ID，types::ID types::lookupTypeForExtension(llvm::StringRef Ext) 函数会根据输入文件 main.m 的扩展名 m 获取该文件的类型 TY\_OBJC。
6. 输入文件处理完成后，会通过 BuildUniversalActions 函数构建 Action
7. 随后再通过 BuildJobs 函数构建 Jobs

3.4 clang driver构建Actions





当需要处理的源码文件列表构建完成后，我们就可以根据参数和源文件类型计算需要的 Action 了。

1. Action[1] 是执行的编译步骤基类，持有Input、Action 类型，产物类型等信息；可以理解为将某种输入转为输出文件的操作步骤，比如，PreprocessJobAction 可以将源码 main.m 转为 main.im
2. InputAction[2] 是特例，只代表原始的输入文件/参数
3. PreprocessJobAction[3] 是将源码进行预处理的过程
4. CompileJobAction[4] 是将上一步的结果转为 bitcode 的过程
5. BackendJobAction[5] 是将 bitcode 转为 .s 文件的过程
6. AssembleJobAction[6] 是将.s 文件转为 .o 二进制文件的过程
7. LinkJobAction[7] 是将 .o 文件合并为静态库/动态库/可执行文件的过程
8. BindArchAction[8] 是特例，将 .o 文件与特定的架构做绑定
9. LipoJobAction[9] 是用于将多个 BindArchAction 输入合并为单一的 fat mach-o 文件
10. JobAction[10] 可以理解能够通过单独的程序执行的过程，注意：Input 和 BindArchAction 没有对应任何的过程

构建 Actions 的目的是为了满足以下目的：

1. clang driver 需要根据 参数 计算需要进行的步骤 比如，当 -emit-llvm 参数传入时，编译器只需要 预处理、编译器前端 两步，不再需要进行 编译器后端 和 汇编 -emit-llvm 的含义是将输入文件编译为 bitcode 文件。
2. clang driver 需要根据 输入文件类型 计算需要进行的步骤 比如，当输入的源码文件是汇编类(扩展名是 .s )型时，只需要最后 汇编 阶段

BuildUniversalActions 方法负责构建 Actions：

1. 根据 -arch 参数生成需要处理的 Archs ，留待后续的处理使用。
2. 如果没有传入 -arch 参数，可以通过 ToolChain::getDefaultUniversalArchName() 方法获取 triple 对应的架构。
3. 调用 BuildActions 计算每个输入文件对应的 SingleActions (可能包含预处理、编译、后端、汇编等) 注意：BuildUniversalActions 的 SingleActions 参数传到 BuildActions 方法后，名字会变为 Actions
4. BuildActions 会先调用 Driver::handleArguments 方法对参数进行一些处理
5. 随后，会遍历输入源码文件Inputs，并通过 llvm::SmallVector<phases::ID, phases::MaxNumberOfPhases> types::getCompilationPhases(const clang::driver::Driver &Driver,llvm::opt::DerivedArgList &DAL, ID Id) 获取需要对输入文件进行处理的 phase 数组。
6. types::getCompilationPhases 内部会根据传入的参数获取需要执行的最后一个 phase。通过 -ccc-print-phases 参数可以对比两种场景的差异，比如，当 -emit-llvm 参数传入时，就会将移除 Backend 后面的 Assemble。
7. 随后，types::getCompilationPhases 会通过函数 llvm::SmallVector<phases::ID, phases::MaxNumberOfPhases> types::getCompilationPhases(ID Id, phases::ID LastPhase) 根据文件类型 TY\_ObjC 和 LastPhase 获取后续的 phase 列表 两个函数名相同，参数不一样。Types.def 文件维护了不同文件类型默认情况下需要经历的 phase。
8. 下一步，Driver::BuildActions 方法会先组装一个 InputAction (每个 Job 都包含一个 Kind 属性，代表该 Job 的类型，InputAction 的 Kind是 InputClass )，InputAction 就相当于输入文件的占位符。
9. 随后，会依次遍历 phase，并根据 phase 创建 Action（通过调用 ConstructPhaseAction 函数实现），每个 NewCurrent 都会持有 Current。

3.5 Action 创建流程介绍

本节会介绍通过除 InputAction 以外的 Action 创建流程

3.5.1 Preprocess

ConstructPhaseAction 方法检测到 phases::Preprocess 时会依次进行以下处理：

1. 根据文件类型获取 TT\_ObjC
2. 根据 TT\_ObjC 获取输出文件的类型 TY\_PP\_ObjC
3. 通过 Input 和 OutputTy 构建 PreprocessJobAction，.m 文件支持的第一个 phase 是 phases::Preprocess，.m 的预处理类型同样由 Types.def 文件维护。

3.5.2 Compile

phases::Compile 代表编译器的 前端 流程，phases::Compile 同样会根据传入的参数判断需要组装的类型，比如是否存在 -rewrite-objc 、-emit-ast 等参数

3.5.3 Backend

phases::Backend 就是我们通常所说的 编译器后端，phases::Backend 负责组装 BackendJobAction，本例中，该 JobAction 的输出文件类型是 TY\_PP\_Asm （文件扩展名是 .s）

3.5.4 Assemble

phases::Assemble 会组装 AssembleJobAction ，该 JobAction 的输出文件类型为 TY\_Object （文件扩展名是 .o）

3.5.5 Link

因为 link 是可以将一个或多个源码文件产出的 .o 文件进行链接，所以，LinkAction 会稍微复杂一些：

Driver::BuildActions 方法会维护一个 LinkerInputs 数组，负责记录需要进行 link 操作的 JobAction 当某个源码文件需要进行 link 操作时，就会先临时保存到 LinkerInputs 数组

当所有源码文件循环完毕后，会判断 LinkerInputs 是否为空；如果非空，会增加一个 LinkJobAction 进行下一步处理。

截止到这一步， 所有的 Action 就会构造为一个类似于链表的构造。

3.5.6 bind & Lipo

link action 创建完毕后，会根据 BuildUniversalActions 生成的 Archs 数组创建对应数量的 BindArchAction，该JobAction 记录需要产出文件的架构，比如 arm64 或者 armv7。

如果 Arch 数量大于 1，会新增一个 LipoJobAction ，LipoJobAction 会将不同的架构的二进制合并为一个 fat mach-o 文件。

1. 基于Loongson和Iluvatar平台进行编译器适配
   1. Loognson 平台适配相关改动

|  |  |
| --- | --- |
| Source | Diff |
| fat\_context.file\_header.magic\_word = 0x20160329;  fat\_context.file\_header.version = 0x00010000;  fat\_context.file\_header.fatbin\_size = 0;  memset(&(fat\_context.fatbin\_header), 0, sizeof(fatbin\_header\_t));  #ifdef \_WINDOWS  fat\_context.fatbin\_header.os\_type = OS\_TYPE\_WINDOWS;  #else  fat\_context.fatbin\_header.os\_type = OS\_TYPE\_LINUX;  #endif | fat\_context.file\_header.magic\_word = 0x20160329;  fat\_context.file\_header.version = 0x00010000;  fat\_context.file\_header.fatbin\_size = 0;  fat\_context.fatbin\_header.flags = 0;  #ifdef \_WINDOWS  fat\_context.fatbin\_header.os\_type = OS\_TYPE\_WINDOWS;  #else  fat\_context.fatbin\_header.os\_type = OS\_TYPE\_LINUX;  #endif  fat\_context.fatbin\_header.arch\_type = ARCH\_TYPE\_NONE;  fat\_context.fatbin\_header.compile\_type = COMPILE\_TYPE\_NONE; |
| llvm/tools/fatbinary/fatbinary.cpp | |

该部分改动是为了解决编译fatbinary文件是初始化问题，对fat\_context.fatbin\_header.arch\_type和fat\_context.fatbin\_header.compile\_type进行了初始化。

|  |  |
| --- | --- |
| Source | Diff |
| if (!RoundingFPMath)  CmdArgs.push\_back(Args.MakeArgString("-fno-rounding-math")); | if (!RoundingFPMath)  #ifdef \_\_loongarch\_\_  if (TC.getArch() != llvm::Triple::mips64) CmdArgs.push\_back(Args.MakeArgString("-fno-rounding-math"));  #else  CmdArgs.push\_back(Args.MakeArgString("-fno-rounding-math"));  #endif |
| CmdArgs.push\_back(Args.MakeArgString(TripleStr)); | #ifdef \_\_loongarch\_\_  CmdArgs.push\_back((IsCuda || IsHIP) && CudaDeviceInput ? "loongarch64-unknown-linux-gnu" : Args.MakeArgString(TripleStr));  #else  CmdArgs.push\_back(Args.MakeArgString(TripleStr));  #endif |
| if (D.CC1Main && !D.CCGenDiagnostics) {  // Invoke the CC1 directly in this process | // Finally add the compile command to the compilation.  #ifdef \_\_loongarch\_\_  SmallString<256> IncPath(D.ResourceDir);  llvm::sys::path::append(IncPath, "include");  CudaInstallationDetector CudaInstallation(D, getToolChain().getTriple(), Args);  if (TC.getArch() == llvm::Triple::bi) {  if (CudaInstallation.isValid())  CmdArgs.push\_back(Args.MakeArgString("-I" + CudaInstallation.getIncludePath()));  CmdArgs.push\_back(Args.MakeArgString("-I" + IncPath));  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/c++/8");  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/loongarch64-linux-gnu");  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/loongarch64-linux-gnu/c++/8");  CmdArgs.push\_back("-D\_GNU\_SOURCE");  }  if (TC.getArch() == llvm::Triple::mips64) {  bool IncAdded = false;  ArgStringList::iterator it = CmdArgs.begin();  while (it != CmdArgs.end()) {  StringRef s\_it(\*it);  if (s\_it == "-mframe-pointer=all" ||  s\_it == "-D\_\_ILCC\_\_" ||  s\_it == "-D\_\_ILUVATAR\_\_" ||  s\_it.startswith("-fgnuc-version=") ||  s\_it.startswith("-resource-dir=")) {  it = CmdArgs.erase(it);  continue;  } else if (s\_it == "-resource-dir" ||  s\_it == "-internal-isystem" ||  s\_it == "-internal-externc-isystem" ||  (s\_it == "-include" && StringRef(\*(it+1)) == "\_\_clang\_cuda\_runtime\_wrapper.h")) {  it = CmdArgs.erase(it);  it = CmdArgs.erase(it);  continue;  } else if (s\_it == "-aux-triple")  \*(++it) = "nvptx64-nvidia-cuda";  else if (s\_it == "-triple")  \*(++it) = "loongarch64-unknown-linux-gnu";  else if (s\_it == "-target-cpu")  \*(++it) = "gs464v";  else if (s\_it == "-target-abi")  \*(++it) = "lp64";  else if (!IncAdded && s\_it.startswith("-I")) {  if (CudaInstallation.isValid())  it = CmdArgs.insert(it, Args.MakeArgString("-I" + CudaInstallation.getIncludePath())) + 1;  IncAdded = true;  }  it++;  }  CmdArgs.push\_back("-resource-dir");  CmdArgs.push\_back("/usr/lib/llvm-8/lib/clang/8.0.1");  CmdArgs.push\_back(Args.MakeArgString("-I" + IncPath));  if (IsCuda && CudaInstallation.isValid())  CudaInstallation.AddCudaIncludeArgs(Args, CmdArgs);  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/c++/8");  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/loongarch64-linux-gnu");  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/loongarch64-linux-gnu/c++/8");  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/include/c++/8/backward");  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back(Args.MakeArgString(IncPath));  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back("/usr/local/include");  CmdArgs.push\_back("-internal-externc-isystem");  CmdArgs.push\_back("/usr/include/loongarch64-linux-gnu");  CmdArgs.push\_back("-internal-externc-isystem");  CmdArgs.push\_back("/include");  CmdArgs.push\_back("-internal-externc-isystem");  CmdArgs.push\_back("/usr/include");  if (CudaInstallation.isValid()) {  CmdArgs.push\_back("-internal-isystem");  CmdArgs.push\_back(Args.MakeArgString(CudaInstallation.getIncludePath()));  }  C.addCommand(std::make\_unique<Command>(JA, \*this, "/usr/bin/clang++", CmdArgs, Inputs));  } else  #endif  以上为loongson增加特有参数支持，下面是对参数判断增加命令行支持  if (Args.hasArg(options::OPT\_\_SLASH\_fallback) &&  Output.getType() == types::TY\_Object &&  (InputType == types::TY\_C || InputType == types::TY\_CXX)) {  auto CLCommand =  getCLFallback()->GetCommand(C, JA, Output, Inputs, Args, LinkingOutput);  C.addCommand(std::make\_unique<FallbackCommand>(  JA, \*this, Exec, CmdArgs, Inputs, std::move(CLCommand)));  } else if (Args.hasArg(options::OPT\_\_SLASH\_fallback) &&  isa<PrecompileJobAction>(JA)) {  // In /fallback builds, run the main compilation even if the pch generation  // fails, so that the main compilation's fallback to cl.exe runs.  C.addCommand(std::make\_unique<ForceSuccessCommand>(JA, \*this, Exec,  CmdArgs, Inputs));  } else if (D.CC1Main && !D.CCGenDiagnostics) { |
| clang/lib/Driver/ToolChains/Clang.cpp | |

该部分改动增加loongarch64-unknown-linux-gnu编译器的支持，该部分涉及工具链ToolChain,针对loongarch64架构在编译器工具链构建新的任务的时候增加对其的其他参数的支持，loongson增加特有参数支持，并对参数判断增加命令行支持。

|  |  |
| --- | --- |
| Source | Diff |
| if (!RoundingFPMath)  CmdArgs.push\_back(Args.MakeArgString("-fno-rounding-math")); | #ifdef \_\_loongarch\_\_  StringRef clangDir = llvm::sys::path::parent\_path(D.ClangExecutable);  if (llvm::sys::path::filename(clangDir) == "bin") {  SmallString<256> clangRoot(llvm::sys::path::parent\_path(clangDir));  SmallString<256> libPath(clangRoot);  llvm::sys::path::append(libPath, "lib");  SmallString<256> lib64Path(clangRoot);  llvm::sys::path::append(lib64Path, "lib64");  CmdArgs.push\_back(Args.MakeArgString("-L" + lib64Path));  CmdArgs.push\_back(Args.MakeArgString("-L" + libPath));  }  bool NeedCudaLibs = false;  ArgStringList::iterator it = CmdArgs.begin();  while (it != CmdArgs.end()) {  if (StringRef(\*it).startswith("--cuda-path=")) {  it = CmdArgs.erase(it);  NeedCudaLibs = true;  } else  it++;  }  if (NeedCudaLibs) {  CudaInstallationDetector CudaInstallation(D, getToolChain().getTriple(), Args);  CmdArgs.push\_back(Args.MakeArgString("-L" + CudaInstallation.getLibPath()));  CmdArgs.push\_back("-lcudart");  }  #endif |
| clang/lib/Driver/ToolChains/Gnu.cpp | |

该部分增加支持Loongson平台编译时对cuda库的链接，包括函数库路径，32位/64位函数库路径区分，当前目录，根目录等。

|  |  |
| --- | --- |
| Source | Diff |
| #ifdef \_\_loongarch\_\_  StringRef clangDir = llvm::sys::path::parent\_path(D.ClangExecutable);  if (llvm::sys::path::filename(clangDir) == "bin") {  SmallString<256> clangRoot(llvm::sys::path::parent\_path(clangDir));  SmallString<256> libPath(clangRoot);  llvm::sys::path::append(libPath, "lib");  SmallString<256> lib64Path(clangRoot);  llvm::sys::path::append(lib64Path, "lib64");  CmdArgs.push\_back(Args.MakeArgString("-L" + lib64Path));  CmdArgs.push\_back(Args.MakeArgString("-L" + libPath));  }  bool NeedCudaLibs = false;  ArgStringList::iterator it = CmdArgs.begin();  while (it != CmdArgs.end()) {  if (StringRef(\*it).startswith("--cuda-path=")) {  it = CmdArgs.erase(it);  NeedCudaLibs = true;  } else  it++;  }  if (NeedCudaLibs) {  CudaInstallationDetector CudaInstallation(D, getToolChain().getTriple(), Args);  CmdArgs.push\_back(Args.MakeArgString("-L" + CudaInstallation.getLibPath()));  CmdArgs.push\_back("-lcudart");  }  #endif | #ifdef \_\_loongarch\_\_  // Remove -mllvm options because they are not supported by g++.  {  ArgStringList::iterator it = CmdArgs.begin();  while (it != CmdArgs.end()) {  StringRef s\_it(\*it);  if (s\_it == "-mllvm") {  it = CmdArgs.erase(it);  it = CmdArgs.erase(it);  continue;  }  it++;  }  }  // Add the paths to ilcc libraries.  {  StringRef clangDir = llvm::sys::path::parent\_path(D.ClangExecutable);  if (llvm::sys::path::filename(clangDir) == "bin") {  SmallString<256> clangRoot(llvm::sys::path::parent\_path(clangDir));  SmallString<256> libPath(clangRoot);  llvm::sys::path::append(libPath, "lib");  SmallString<256> lib64Path(clangRoot);  llvm::sys::path::append(lib64Path, "lib64");  CmdArgs.push\_back(Args.MakeArgString("-L" + lib64Path));  CmdArgs.push\_back(Args.MakeArgString("-L" + libPath));  }  }  // If --cuda-path is provided, which is not recognized by g++, replace it  // with -L<cuda-path> -lcudart.  {  ArgStringList::iterator it = CmdArgs.begin();  while (it != CmdArgs.end()) {  if (StringRef(\*it).startswith("--cuda-path=")) {  it = CmdArgs.erase(it);  CudaInstallationDetector CudaInstallation(D, getToolChain().getTriple(), Args);  CmdArgs.push\_back(Args.MakeArgString("-L" + CudaInstallation.getLibPath()));  CmdArgs.push\_back("-lcudart");  break;  } else  it++;  }  }  #endif |
| clang/lib/Driver/ToolChains/Gnu.cpp | |

上面是对之前Gnu.cpp的第一版修改的继续改动，删除了-mllvm参数的支持，因为g++不支持该参数。下面对cuda相应的函数库路径进行支持，如果g++没有识别出cuda path 这用—L<cuda path> 来替换。

|  |  |
| --- | --- |
| Source | Diff |
| #elif defined(\_\_mips\_\_) || defined(\_\_m68k\_\_) | #elif defined(\_\_mips\_\_) || defined(\_\_loongarch\_\_) |
| llvm/utils/benchmark/src/cycleclock.h | |

|  |  |
| --- | --- |
| Source | Diff |
| #elif defined(\_\_mips\_\_) || defined(\_\_m68k\_\_) | elseif (LLVM\_NATIVE\_ARCH MATCHES "loongarch64")  set(LLVM\_NATIVE\_ARCH Mips) |
| llvm/cmake/config-ix.cmake | |

|  |  |
| --- | --- |
| Source | Diff |
|  | ARCH := $(shell uname -m | sed \  -e 's/i.86/i386/' \  -e 's/armv[0-7]\w\+/arm/' \  -e 's/aarch64/arm64/' \  -e 's/ppc64le/powerpc/') |
|  | ARCH = $(ARCH) |
|  | ifeq ($(ARCH), loongarch64)  CONFIG\_ARGS := \  $(CONFIG\_ARGS) \  -DLLVM\_DEFAULT\_TARGET\_TRIPLE:STRING=mips64  endif |
| Makefile | |

|  |  |
| --- | --- |
| Source | Diff |
| set(config\_guess ${LLVM\_MAIN\_SRC\_DIR}/cmake/config.guess) | if( CMAKE\_HOST\_SYSTEM\_PROCESSOR STREQUAL "loongarch64" )  set(config\_guess "/usr/share/libtool/build-aux/config.guess")  else()  set(config\_guess ${LLVM\_MAIN\_SRC\_DIR}/cmake/config.guess)  endif() |
| llvm/cmake/config-ix.cmake | |

|  |  |
| --- | --- |
| Source | Diff |
| if (!FPExceptionBehavior.empty())  CmdArgs.push\_back(Args.MakeArgString("-ffp-exception-behavior=" +  FPExceptionBehavior)); | #ifndef \_\_loongarch\_\_  if (!FPExceptionBehavior.empty())  CmdArgs.push\_back(Args.MakeArgString("-ffp-exception-behavior=" +  FPExceptionBehavior));  #endif |
| if (D.getDiags().getDiagnosticOptions().ShowColors)  CmdArgs.push\_back("-fcolor-diagnostics"); | #ifndef \_\_loongarch\_\_  if (D.getDiags().getDiagnosticOptions().ShowColors)  CmdArgs.push\_back("-fcolor-diagnostics");  #endif |
| llvm/cmake/config-ix.cmake | |

条件编译，在loongson平台去掉-ffp-exception-behavior=，-fcolor-diagnostics参数。

|  |  |
| --- | --- |
| Source | Diff |
|  | } else if (s\_it.startswith("-fcolor-diagnostics")) {  it = CmdArgs.erase(it);  continue; |
| llvm/cmake/config-ix.cmake | |

|  |  |
| --- | --- |
| Source | Diff |
| if (!UseAbiCalls)  Features.push\_back("+noabicalls");  else  Features.push\_back("-noabicalls"); | #ifndef \_\_loongarch\_\_  if (!UseAbiCalls)  Features.push\_back("+noabicalls");  else  Features.push\_back("-noabicalls");  #endif |
| llvm/cmake/config-ix.cmake | |

1. Iluvatar平台编译器pass