中国科学院软件研究所
Institute of Software Chinese Academy of Sciences

智能软件研究中心
Intelligent Software Research Center

软件所智能软件中心PLCT实验室 王鹏 实习生

2020//04/22

目录

01 rvv-llvm中加入vstart等指令

02 llvm intrinsics函数结合源代码介绍

03 llvm intrinsics函数结合llvm IR介绍

- 01 rvv-llvm中加入vstart等指令

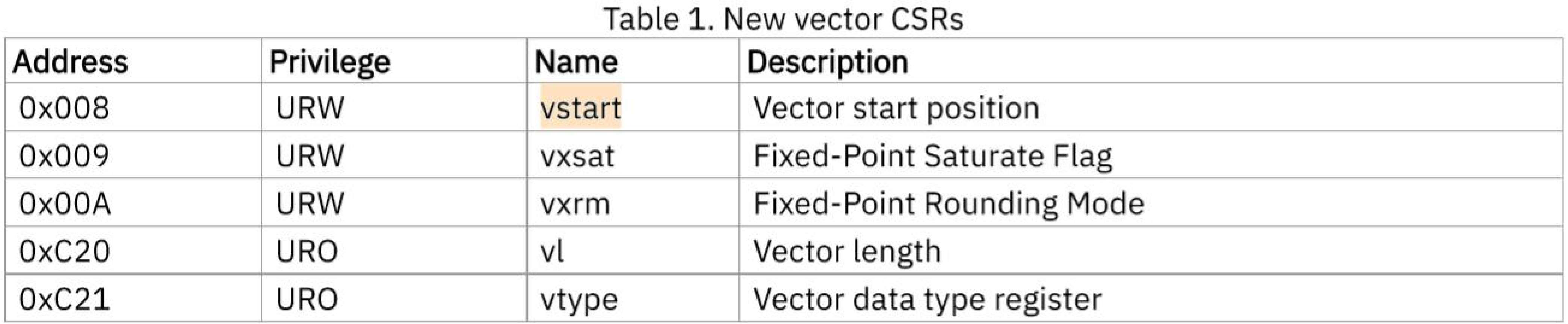
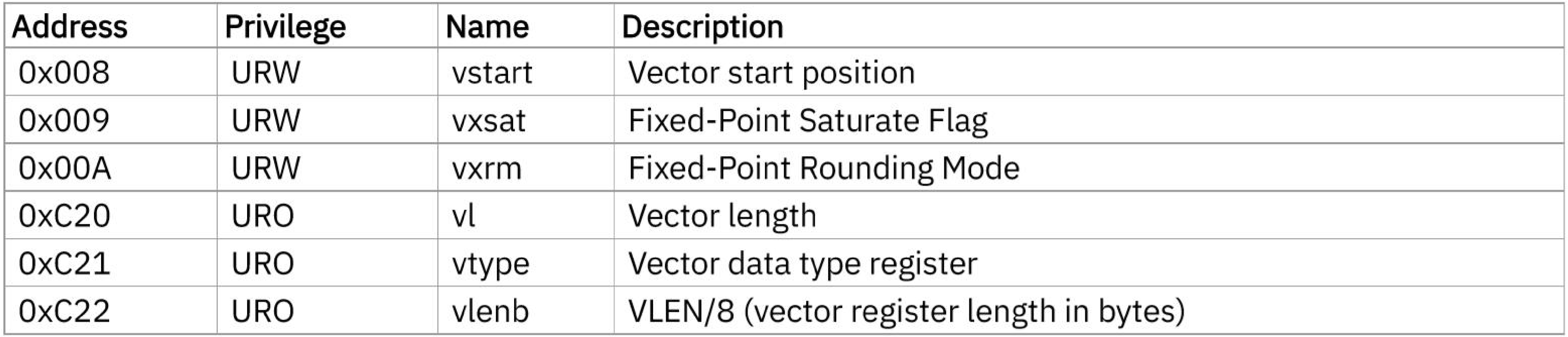


Table 1. New vector CSRs

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register

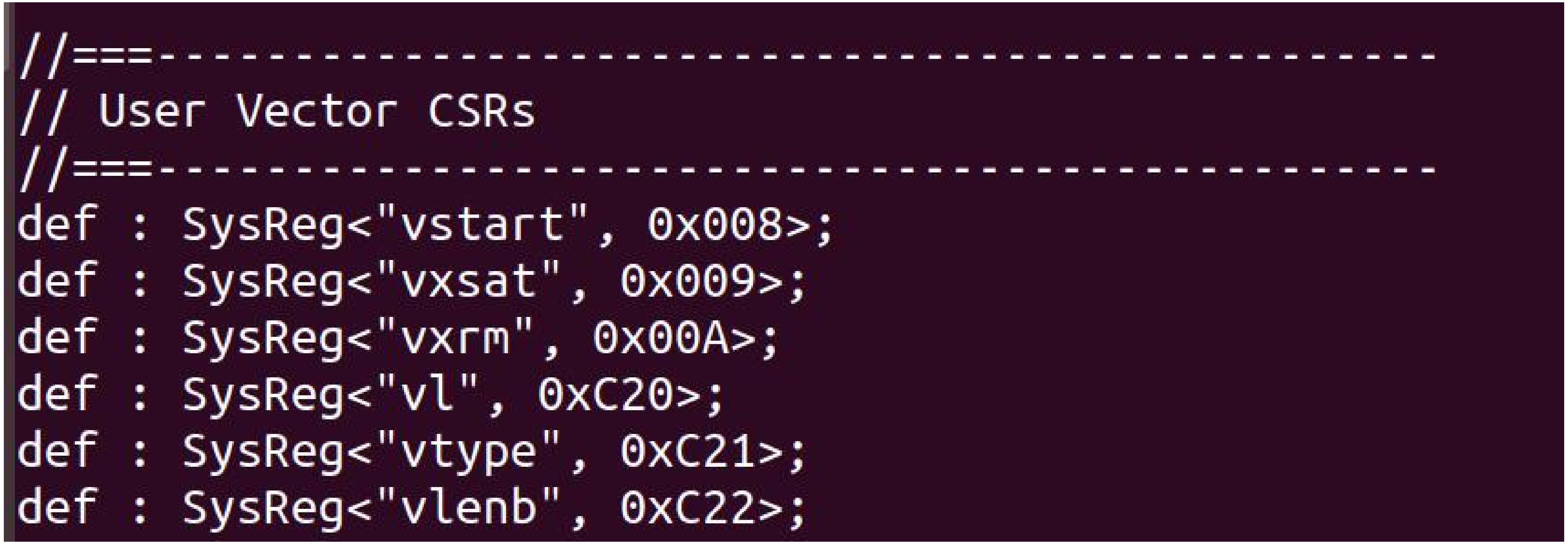
riscv-v-spec 0.7.1

Table 1. New vector CSRs



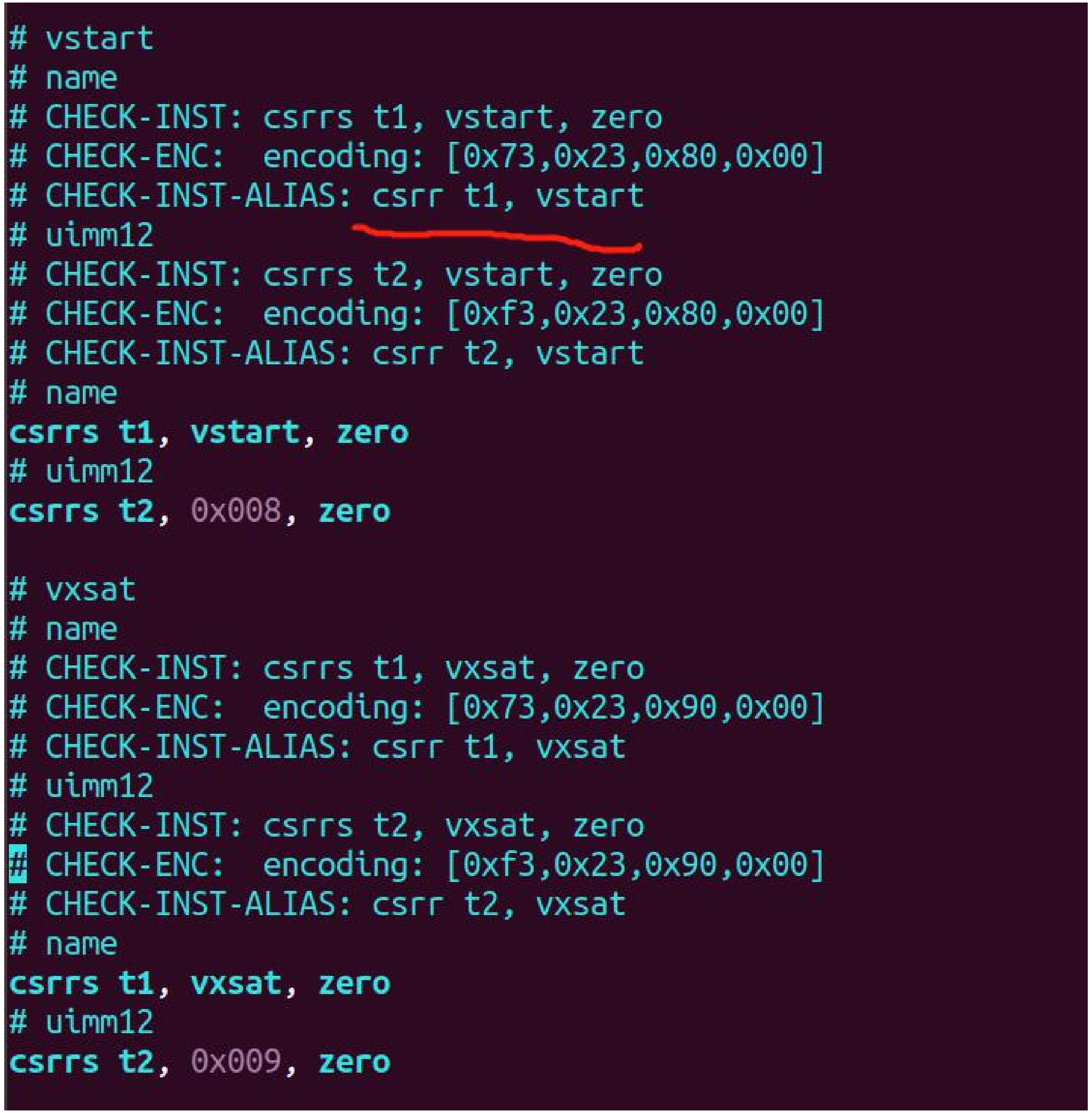
Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

riscv-v-spec 0.8 新增vlenb



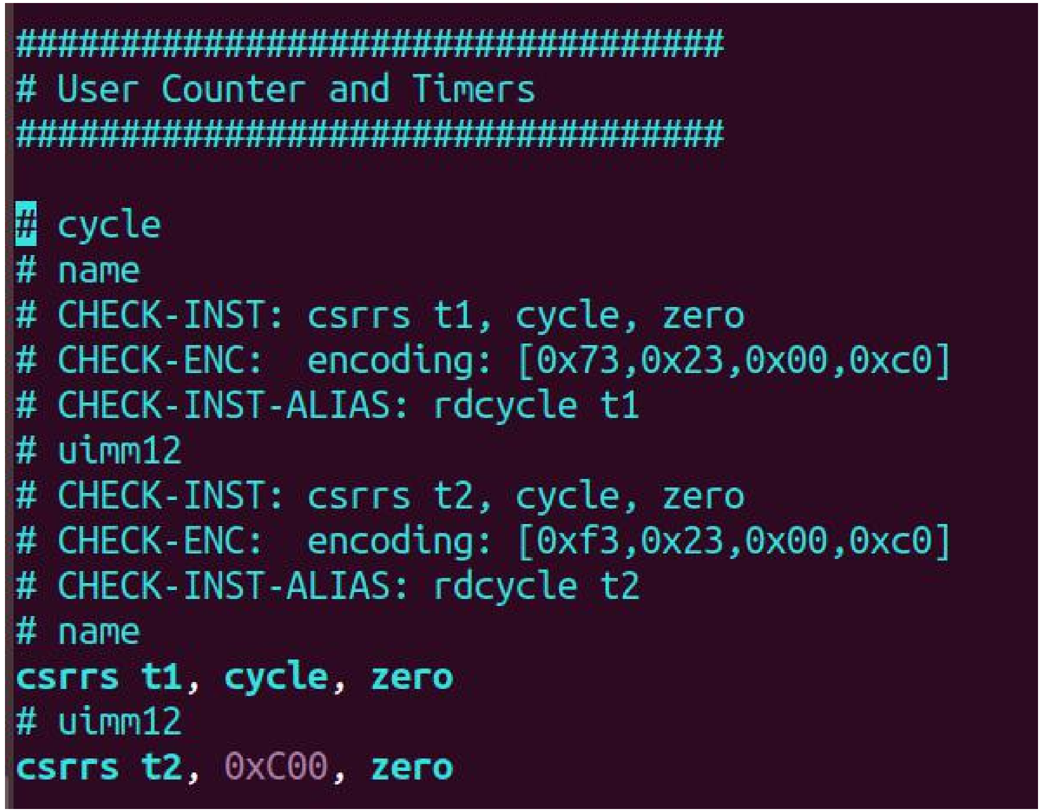
```
//===-----------------------------------------------
// User Vector CSRs
//===-----------------------------------------------
def : SysReg<"vstart", 0x008>;
def : SysReg<"vxsat", 0x009>;
def : SysReg<"vxrm", 0x00A>;
def : SysReg<"vl", 0xC20>;
def : SysReg<"vtype", 0xC21>;
def : SysReg<"vlenb", 0xC22>;
```

RISCVSystemOperands.td



```
# vstart
# name
# CHECK-INST: csrrs t1, vstart, zero
# CHECK-ENC:  encoding: [0x73,0x23,0x80,0x00]
# CHECK-INST-ALIAS: csrr t1, vstart
# uimm12
# CHECK-INST: csrrs t2, vstart, zero
# CHECK-ENC:  encoding: [0xf3,0x23,0x80,0x00]
# CHECK-INST-ALIAS: csrr t2, vstart
# name
csrrs t1, vstart, zero
# uimm12
csrrs t2, 0x008, zero

# vxsat
# name
# CHECK-INST: csrrs t1, vxsat, zero
# CHECK-ENC:  encoding: [0x73,0x23,0x90,0x00]
# CHECK-INST-ALIAS: csrr t1, vxsat
# uimm12
# CHECK-INST: csrrs t2, vxsat, zero
# CHECK-ENC:  encoding: [0xf3,0x23,0x90,0x00]
# CHECK-INST-ALIAS: csrr t2, vxsat
# name
csrrs t1, vxsat, zero
# uimm12
csrrs t2, 0x009, zero
```

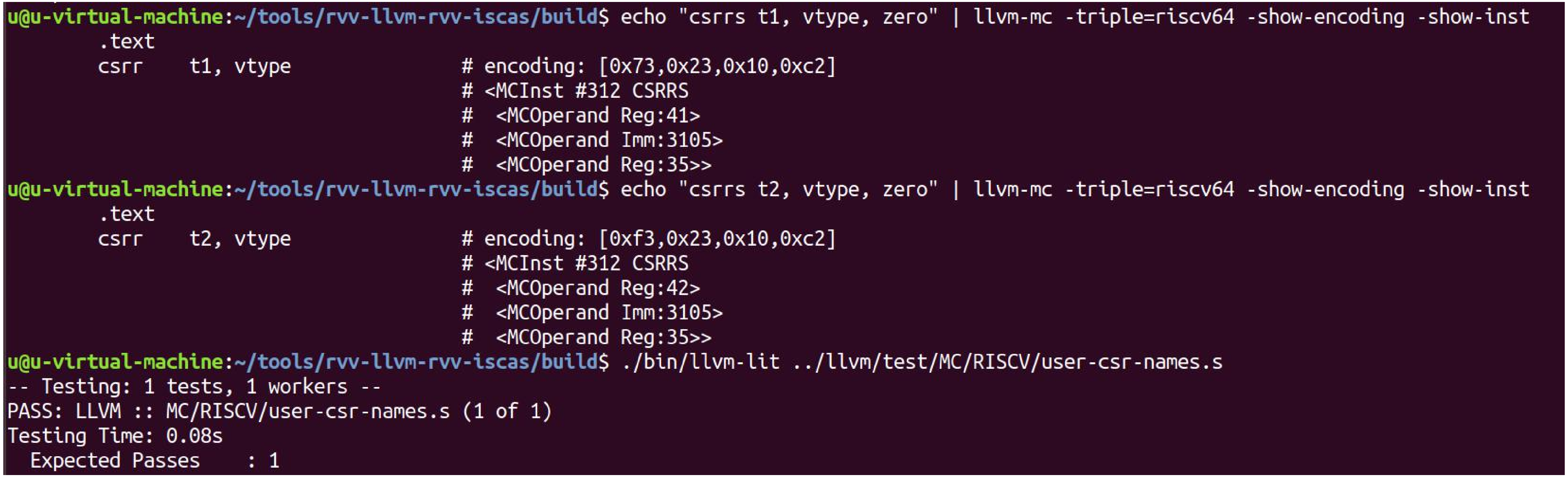
在user-csr-names.s



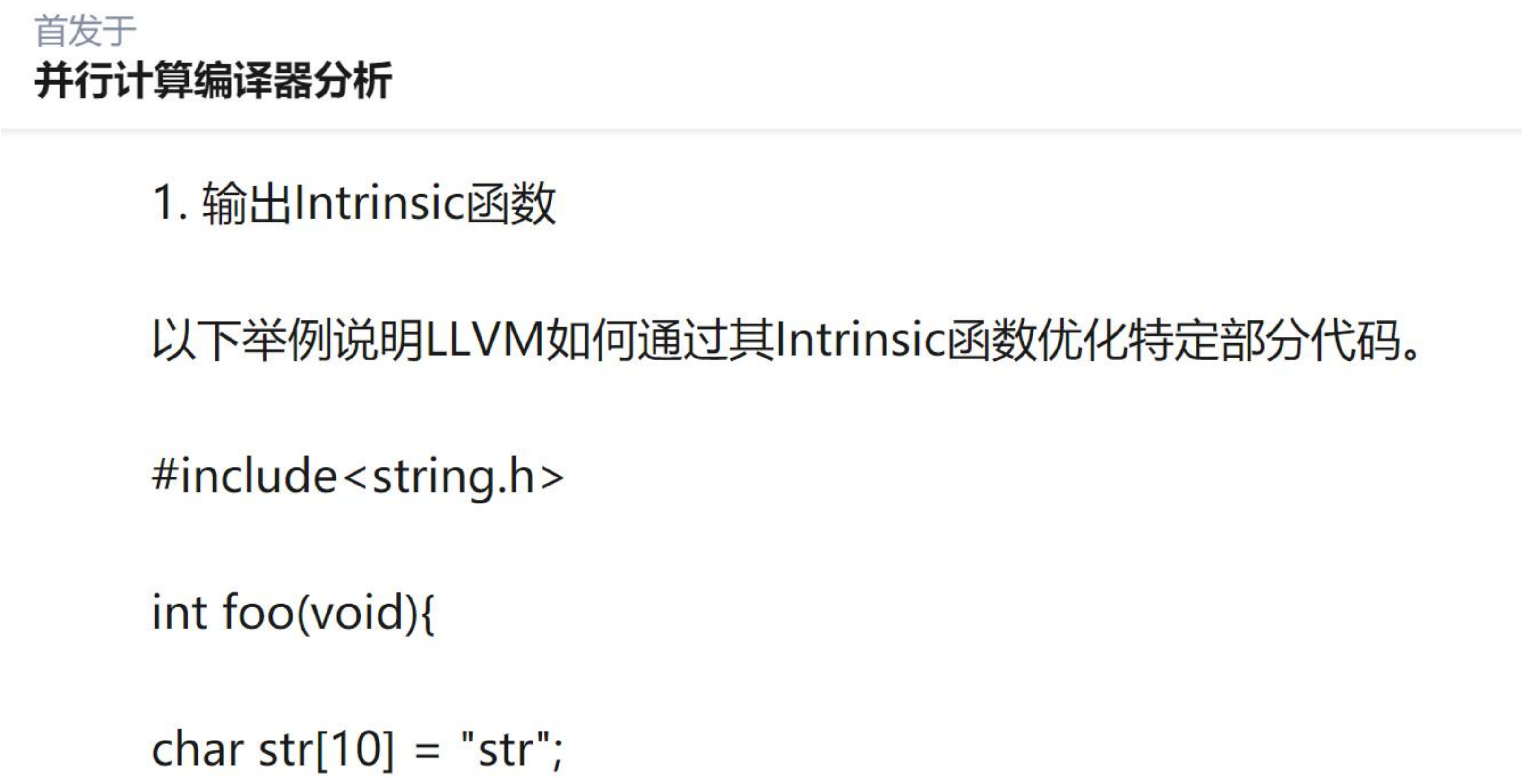
```
###################################
# User Counter and Timers
###################################

# cycle
# name
# CHECK-INST: csrrs t1, cycle, zero
# CHECK-ENC:  encoding: [0x73,0x23,0x00,0xc0]
# CHECK-INST-ALIAS: rdcycle t1
# uimm12
# CHECK-INST: csrrs t2, cycle, zero
# CHECK-ENC:  encoding: [0xf3,0x23,0x00,0xc0]
# CHECK-INST-ALIAS: rdcycle t2
# name
csrrs t1, cycle, zero
# uimm12
csrrs t2, 0xC00, zero
```

在user-csr-names.s中 time,cycle,instret的CHECK-INST-ALIAS和hpccounter3-31的不同格式?

硬件性能计数器(Hardware Performance counter Monitor,HPM



```
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/build$ echo "csrrs t1, vtype, zero" | llvm-mc -triple=riscv64 -show-encoding -show-inst
        .text
        csrr    t1, vtype               # encoding: [0x73,0x23,0x10,0xc2]
                                        # <MCInst #312 CSRRS
                                        #  <MCOperand Reg:41>
                                        #  <MCOperand Imm:3105>
                                        #  <MCOperand Reg:35>>
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/build$ echo "csrrs t2, vtype, zero" | llvm-mc -triple=riscv64 -show-encoding -show-inst
        .text
        csrr    t2, vtype               # encoding: [0xf3,0x23,0x10,0xc2]
                                        # <MCInst #312 CSRRS
                                        #  <MCOperand Reg:42>
                                        #  <MCOperand Imm:3105>
                                        #  <MCOperand Reg:35>>
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/build$ ./bin/llvm-lit ../llvm/test/MC/RISCV/user-csr-names.s
-- Testing: 1 tests, 1 workers --
PASS: LLVM :: MC/RISCV/user-csr-names.s (1 of 1)
Testing Time: 0.08s
  Expected Passes    : 1
```



首发于

并行计算编译器分析

1. 输出Intrinsic函数

以下举例说明LLVM如何通过其Intrinsic函数优化特定部分代码。

```
#include<string.h>

int foo(void){

char str[10] = "str";
```

介绍intrinsic函数

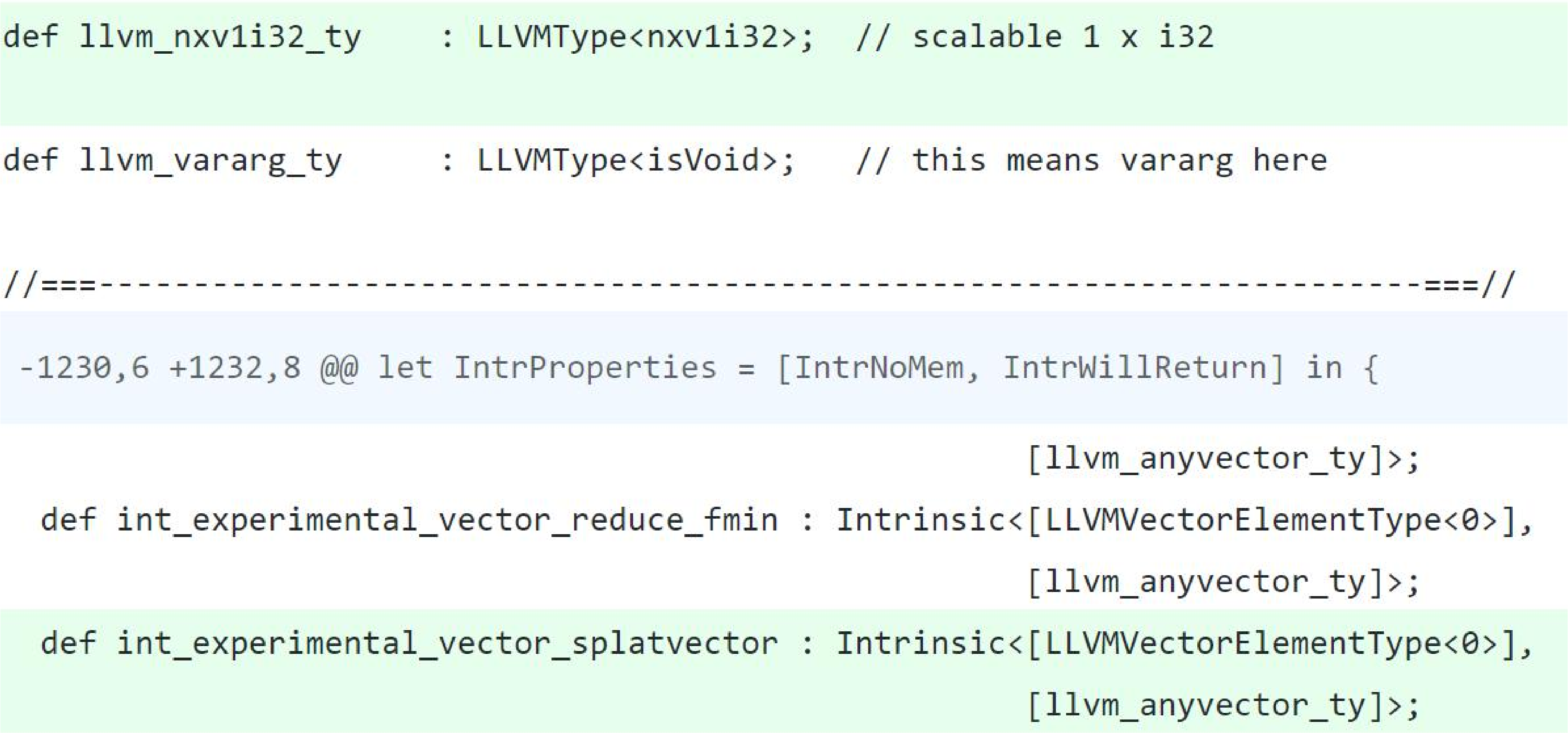
https://zhuanlan.zhihu.com/p/53659330

• 02 llvm intrinsics函数介绍

llvm/include/llvm/IR/Intrinsics*.td

为intinsic函数添加一个条目。描述其内存访问特性以进行优化（这将控制是否进行DCE，CSE等）。如果有任何参数需要立即数，则必须使用ImmArg属性指示它们。请注意，任何将其中一种llvm_any*_ty类型用作参数或返回类型的intinsic函数都将被视为tblgen重载，并且在内部函数名称上将需要相应的后缀。

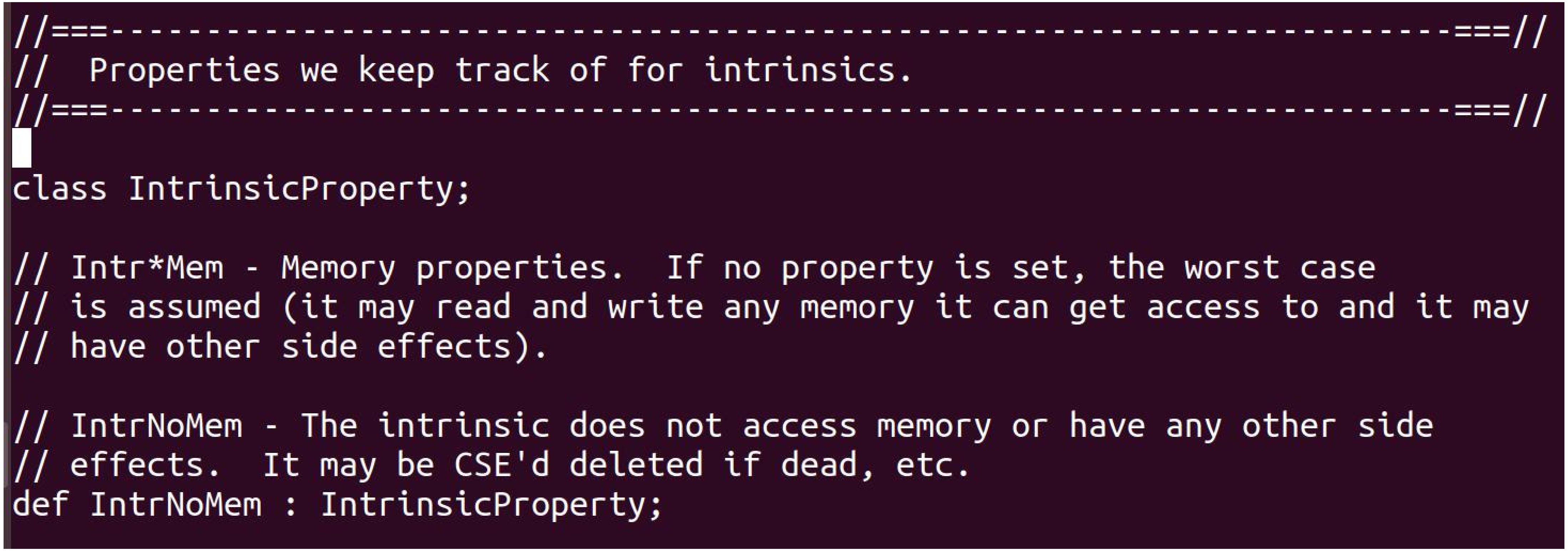
llvm/include/llvm/IR/Intrinsics.td



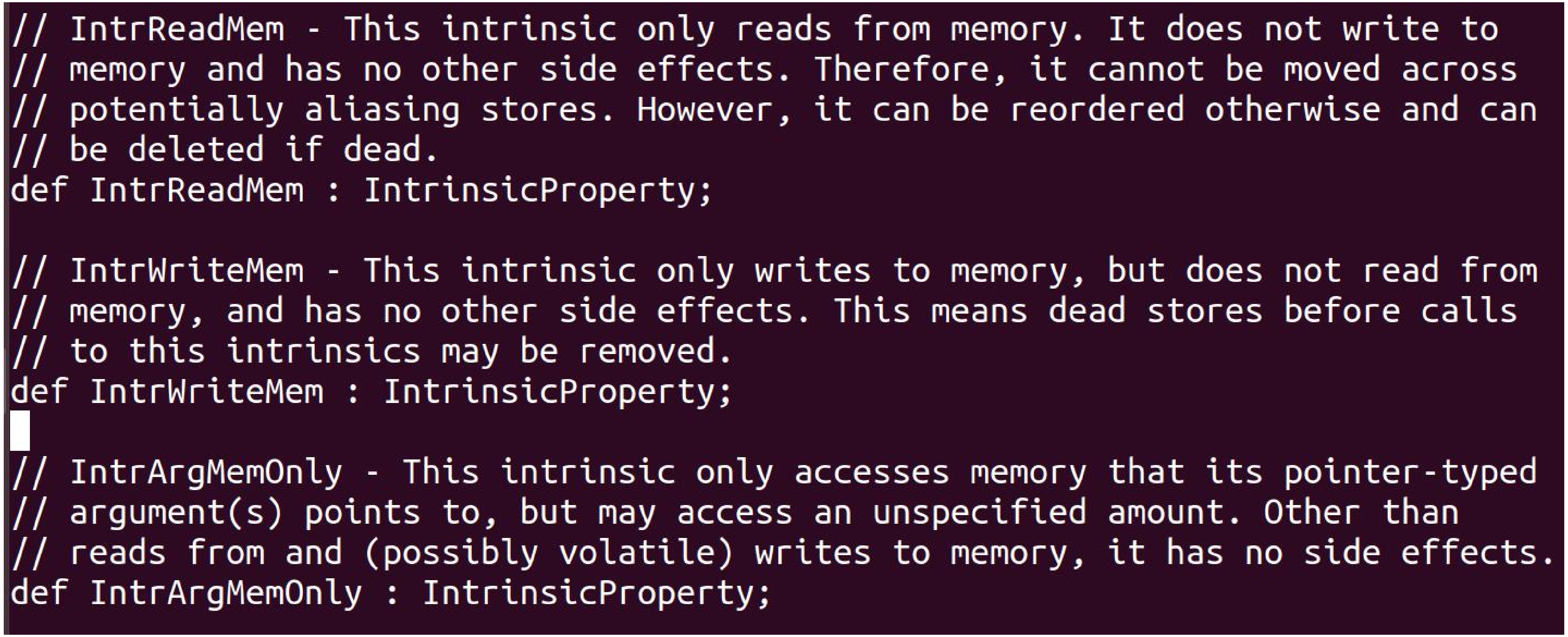
```
def llvm_nxv1i32_ty    : LLVMType<nxv1i32>;  // scalable 1 x i32

def llvm_vararg_ty     : LLVMType<isVoid>;   // this means vararg here

//===----------------------------------------------------------------------===//
 -1230,6 +1232,8 @@ let IntrProperties = [IntrNoMem, IntrWillReturn] in {
                                                      [llvm_anyvector_ty]>;
  def int_experimental_vector_reduce_fmin : Intrinsic<[LLVMVectorElementType<0>],
                                                      [llvm_anyvector_ty]>;
  def int_experimental_vector_splatvector : Intrinsic<[LLVMVectorElementType<0>],
                                                      [llvm_anyvector_ty]>;
```

llvm/include/llvm/IR/Intrinsics.td

这个文件定义了所有LLVM Intrinsics的性质

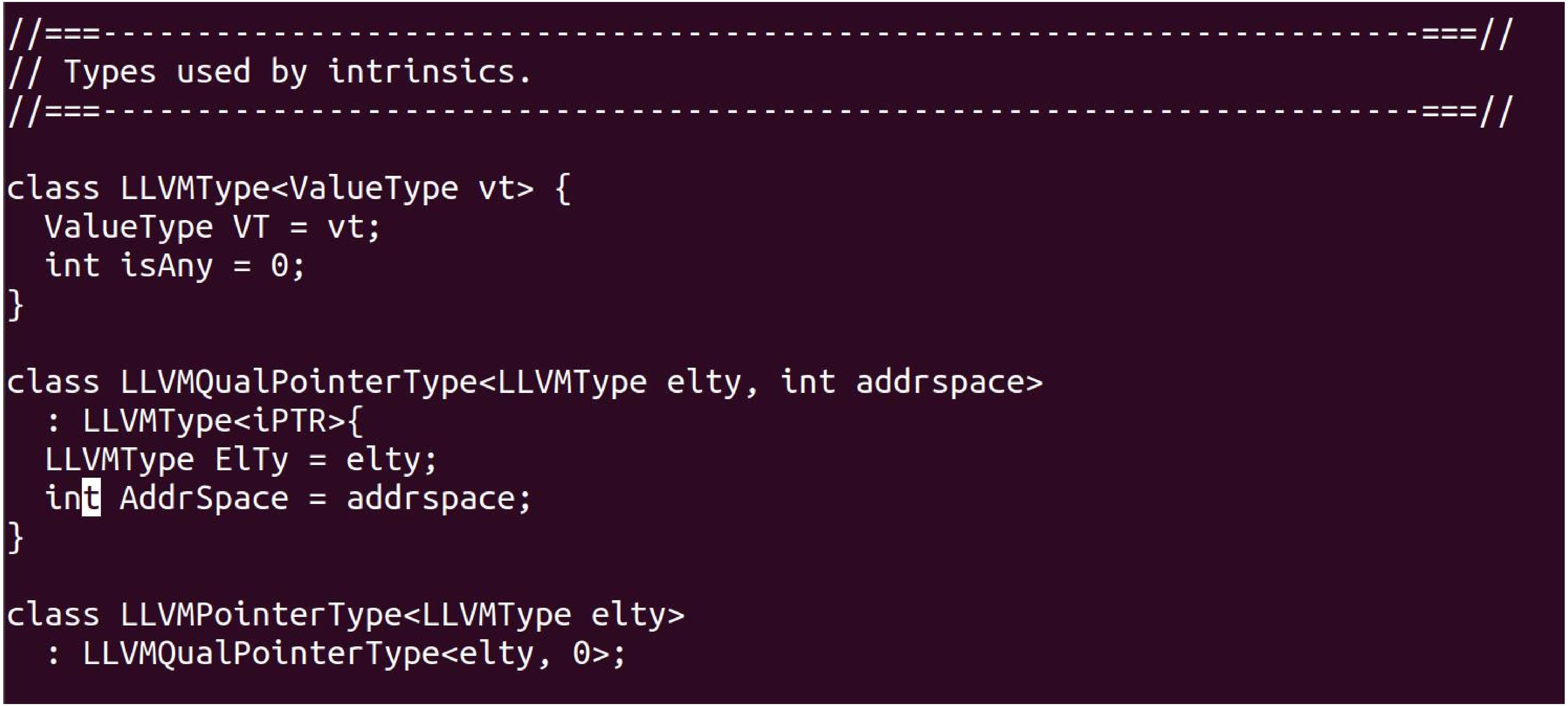


```
//===----------------------------------------------------------------------===//
//  Properties we keep track of for intrinsics.
//===----------------------------------------------------------------------===//

class IntrinsicProperty;

// Intr*Mem - Memory properties.  If no property is set, the worst case
// is assumed (it may read and write any memory it can get access to and it may
// have other side effects).

// IntrNoMem - The intrinsic does not access memory or have any other side
// effects.  It may be CSE'd deleted if dead, etc.
def IntrNoMem : IntrinsicProperty;
```

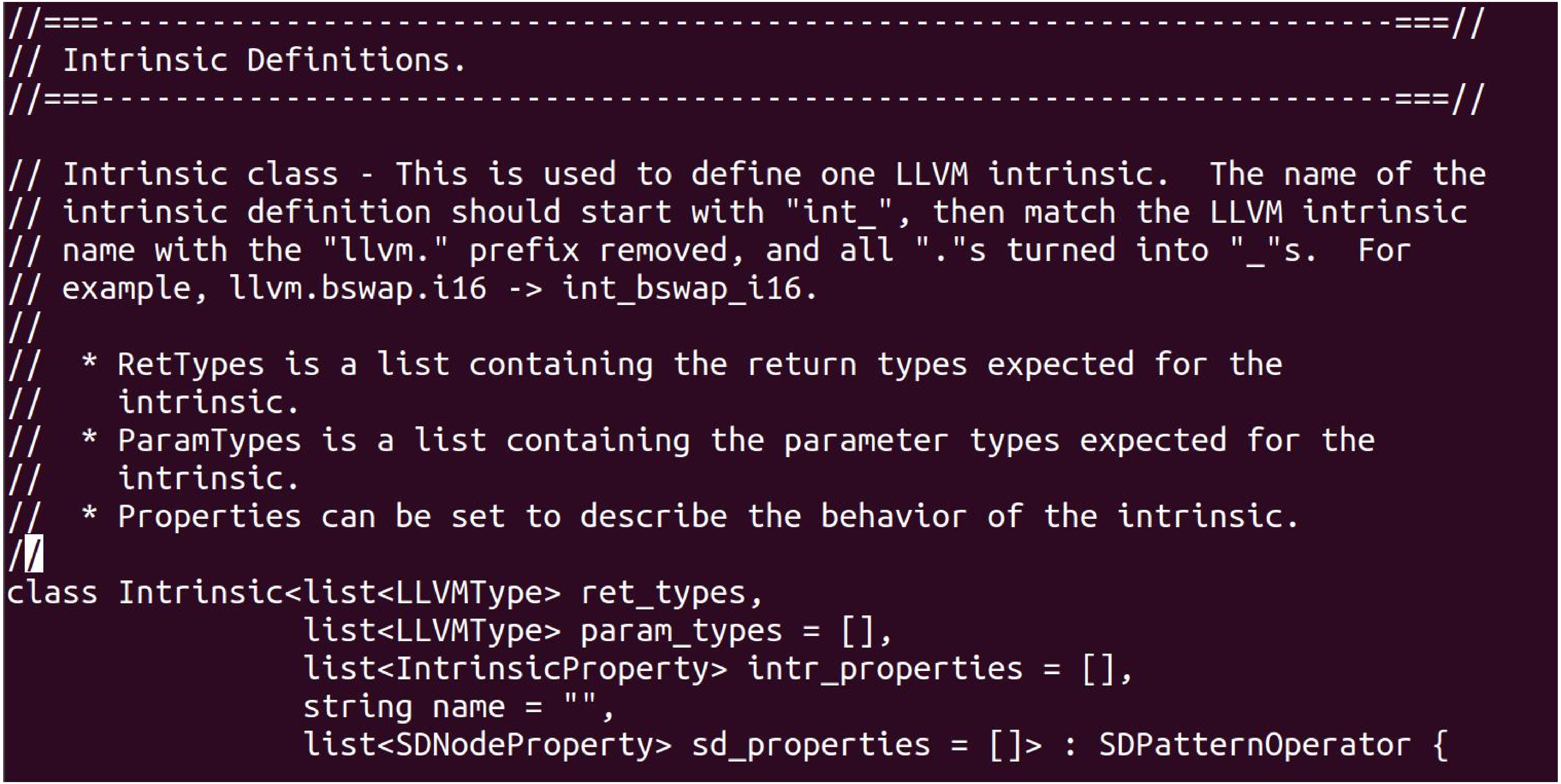


```
// IntrReadMem - This intrinsic only reads from memory. It does not write to
// memory and has no other side effects. Therefore, it cannot be moved across
// potentially aliasing stores. However, it can be reordered otherwise and can
// be deleted if dead.
def IntrReadMem : IntrinsicProperty;

// IntrWriteMem - This intrinsic only writes to memory, but does not read from
// memory, and has no other side effects. This means dead stores before calls
// to this intrinsics may be removed.
def IntrWriteMem : IntrinsicProperty;

// IntrArgMemOnly - This intrinsic only accesses memory that its pointer-typed
// argument(s) points to, but may access an unspecified amount. Other than
// reads from and (possibly volatile) writes to memory, it has no side effects.
def IntrArgMemOnly : IntrinsicProperty;
```

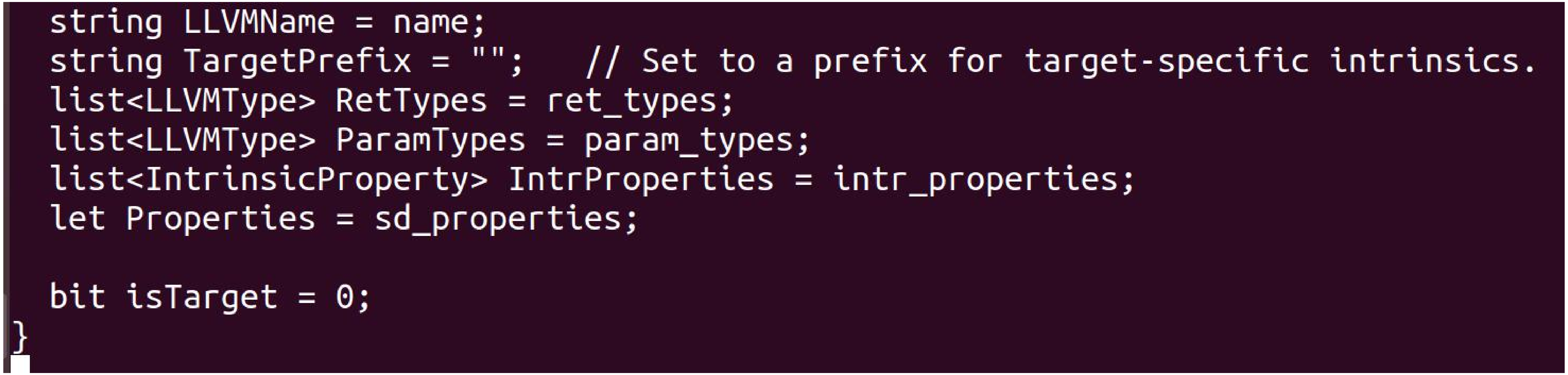
此外，还有IntrInaccessibleMemOnly，Commutative（X op Y == Y op X），Throws，class NoCapture，class NoAlias，class Returned等性质



```
//===----------------------------------------------------------------------===//
// Types used by intrinsics.
//===----------------------------------------------------------------------===//

class LLVMType<ValueType vt> {
  ValueType VT = vt;
  int isAny = 0;
}

class LLVMQualPointerType<LLVMType elty, int addrspace>
  : LLVMType<iPTR>{
  LLVMType ElTy = elty;
  int AddrSpace = addrspace;
}

class LLVMPointerType<LLVMType elty>
  : LLVMQualPointerType<elty, 0>;
```



```
//===----------------------------------------------------------------------===//
// Intrinsic Definitions.
//===----------------------------------------------------------------------===//

// Intrinsic class - This is used to define one LLVM intrinsic.  The name of the
// intrinsic definition should start with "int_", then match the LLVM intrinsic
// name with the "llvm." prefix removed, and all "."s turned into "_"s.  For
// example, llvm.bswap.i16 -> int_bswap_i16.
//
//  * RetTypes is a list containing the return types expected for the
//    intrinsic.
//  * ParamTypes is a list containing the parameter types expected for the
//    intrinsic.
//  * Properties can be set to describe the behavior of the intrinsic.
//
class Intrinsic<list<LLVMType> ret_types,
                list<LLVMType> param_types = [],
                list<IntrinsicProperty> intr_properties = [],
                string name = "",
                list<SDNodeProperty> sd_properties = []> : SDPatternOperator {
```



```
  string LLVMName = name;
  string TargetPrefix = "";   // Set to a prefix for target-specific intrinsics.
  list<LLVMType> RetTypes = ret_types;
  list<LLVMType> ParamTypes = param_types;
  list<IntrinsicProperty> IntrProperties = intr_properties;
  let Properties = sd_properties;

  bit isTarget = 0;
}
```

//===------ Variable Argument Handling Intrinsics --------===//

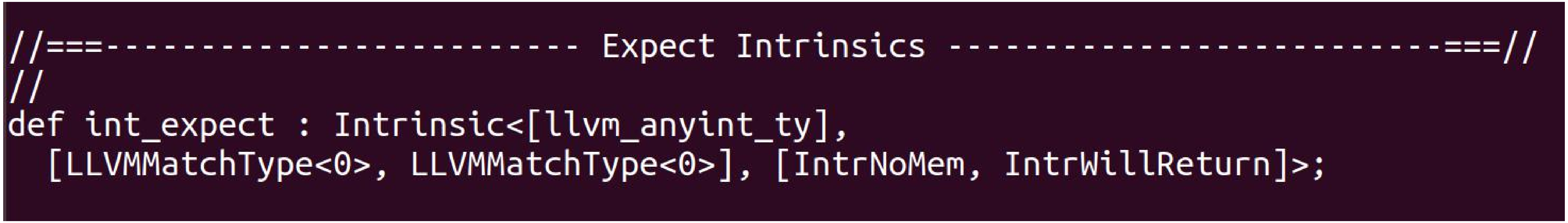
//===--------- Garbage Collection Intrinsics -------------===//

//===-------------- ObjC ARC runtime Intrinsics ---------===//

//===-------------- Code Generator Intrinsics -----------===//

//===--------- Standard C Library Intrinsics --------------===//

//===--------- Constrained Floating Point Intrinsics -------===//



```
//===------------------------- Expect Intrinsics --------------------------===//
//
def int_expect : Intrinsic<[llvm_anyint_ty],
  [LLVMMatchType<0>, LLVMMatchType<0>], [IntrNoMem, IntrWillReturn]>;
```

//===----------- Bit Manipulation Intrinsics ---------===//

//===------------ Debugger Intrinsics ------------===//

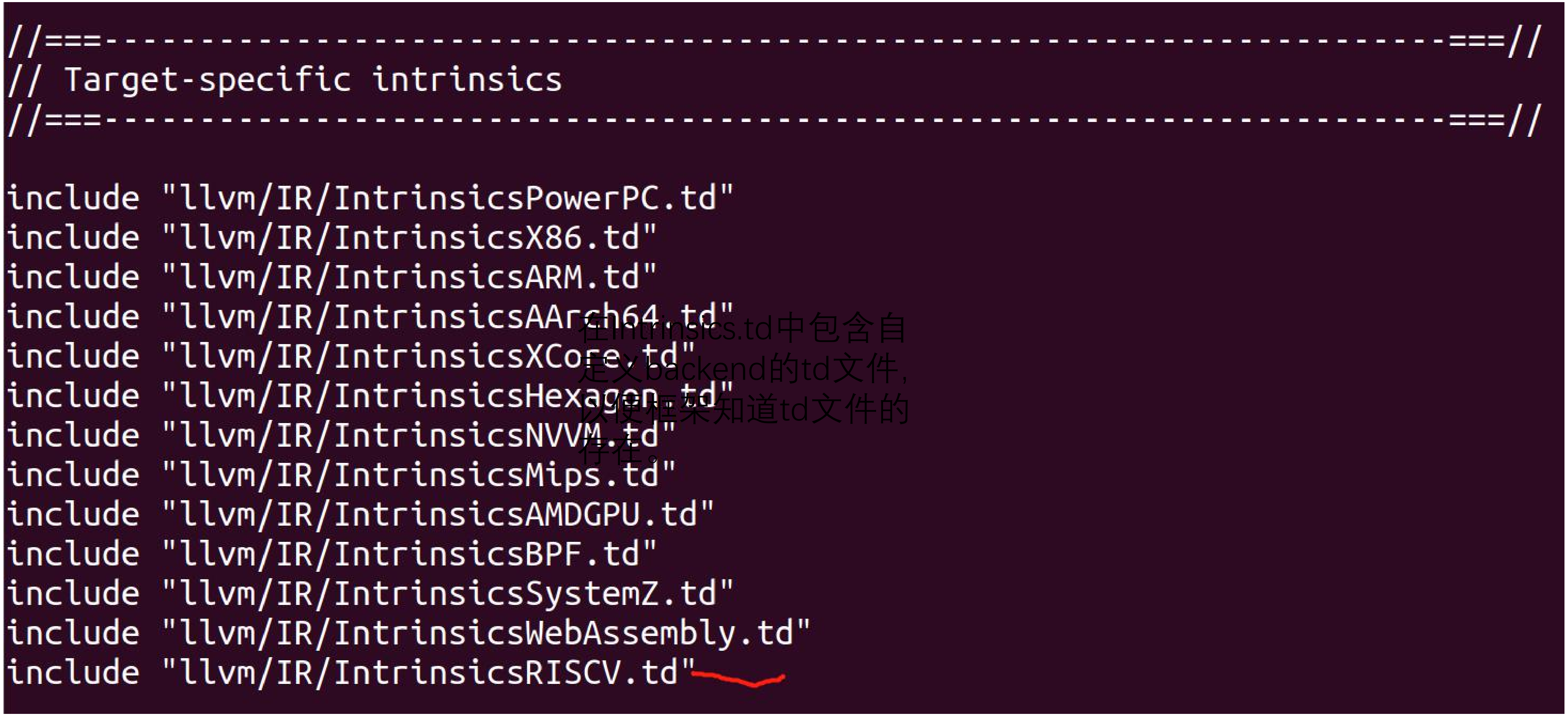
//===------------ Trampoline Intrinsics -----------===//

//===----------- Overflow Intrinsics -------------===//

//===---------- Saturation Arithmetic Intrinsics ------===//

//===-------- Fixed Point Arithmetic Intrinsic---------===//

llvm/include/llvm/IR/Intrinsics.td



```
//===-------------------------------------------------------------------===//
// Target-specific intrinsics
//===-------------------------------------------------------------------===//

include "llvm/IR/IntrinsicsPowerPC.td"
include "llvm/IR/IntrinsicsX86.td"
include "llvm/IR/IntrinsicsARM.td"
include "llvm/IR/IntrinsicsAArch64.td"
include "llvm/IR/IntrinsicsXCore.td"
include "llvm/IR/IntrinsicsHexagon.td"
include "llvm/IR/IntrinsicsNVVM.td"
include "llvm/IR/IntrinsicsMips.td"
include "llvm/IR/IntrinsicsAMDGPU.td"
include "llvm/IR/IntrinsicsBPF.td"
include "llvm/IR/IntrinsicsSystemZ.td"
include "llvm/IR/IntrinsicsWebAssembly.td"
include "llvm/IR/IntrinsicsRISCV.td"
```

在Intrinsics.td中包含自定义backend的td文件，以使框架知道td文件的存在。

在Intrinsics.td中包含定义的backend.td文件，框架可以知道td文件的存在。

llvm/include/llvm/IR/Intrinsics.td

llvm/include/llvm/IR/IntrinsicsRISCV.td

在include/llvm/IR中定义了llvm IR的函数。其中Intrinsics.td中描述了全部的指令，先定义各个类型函数的class，然后def 调用class进行函数的定义。

IntrinsicsRISCV.td定义了RISCV专有函数的定义。比如，let TargetPrefix = "riscv" 是规定平台，def后面的是指令，Intrinsic是函数的参数，第一个参数返回值，第二个参数，第三个是参数类型。

llvm/include/llvm/IR/Intrinsics.td

rvv-llvm在Intrinsic中增加了两个指令：

```
def llvm_nxv1i32_ty    : LLVMType<nxv1i32>;  // scalable 1 x i32

def int_experimental_vector_splatvector : Intrinsic<[LLVMVectorElementType<0>], [llvm_anyvector_ty]>;
```

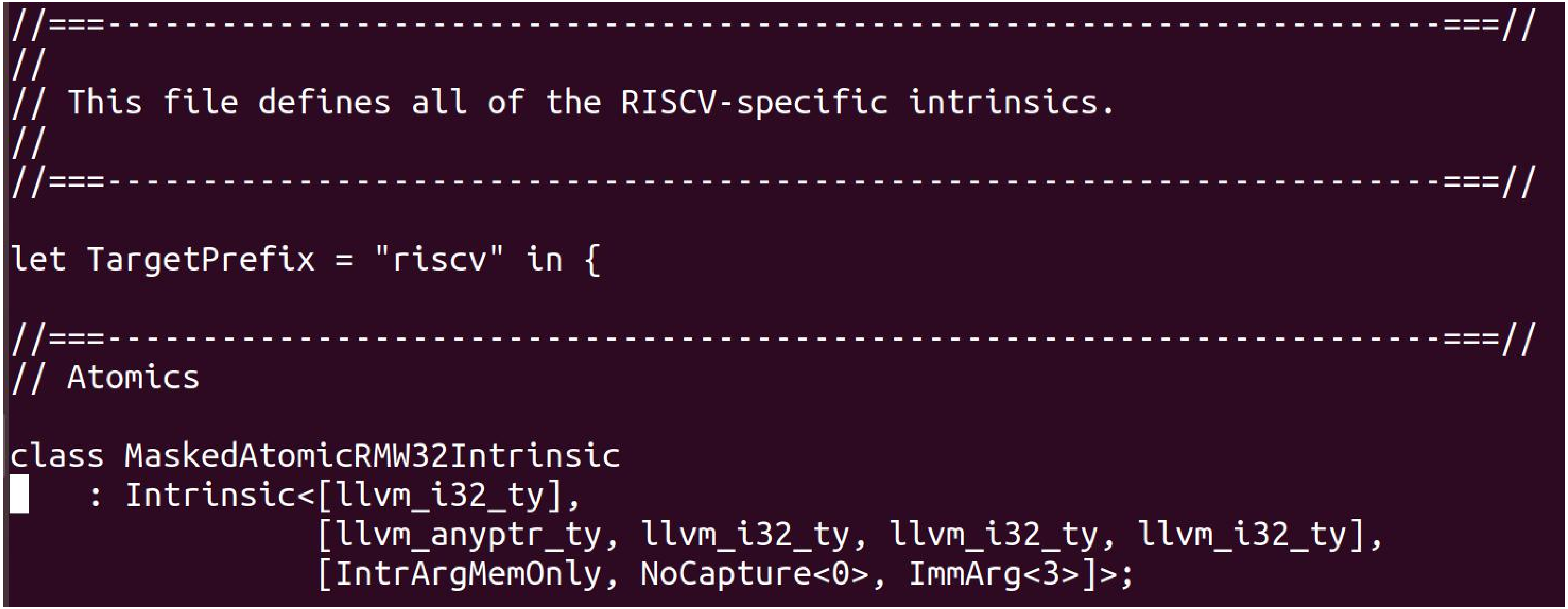
下周我会继续讲解，增加RISCV的SDNode，在RISCVISelLowering.h中添加enum NodeType和Handling of specific intrinsics。并且添加配合Intrinsic函数的定义到RISCVISelLowering.cpp，以及一系列相关定义。

/tools/clang/include/clang/Basic/BuiltinsRISCV.def没有

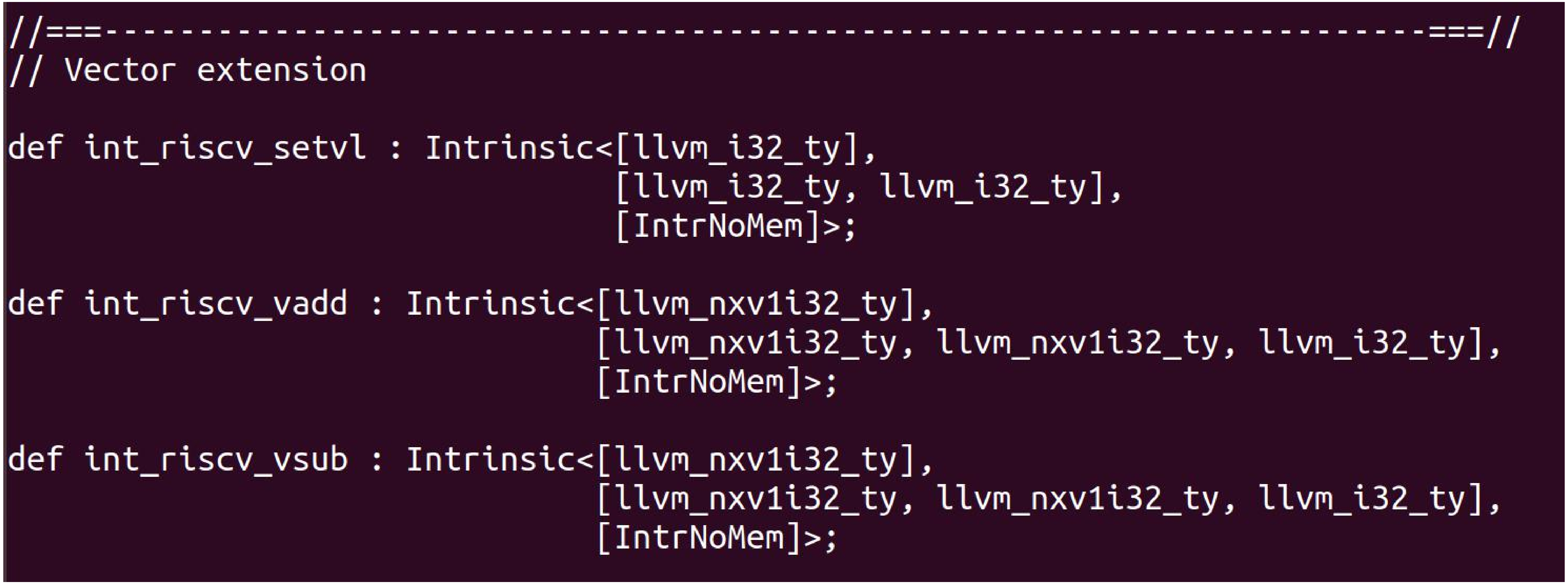
例如，BuiltinsX86.def文件定义了X86特定的内置函数数据库。该文件的用户必须定义 BUILTIN宏才能使用此信息。

rvv-llvm没有BuiltinsRISCV.def，这意味着用户没有必要在llvm/include/llvm/IR/Intrinsics.td中定义 BUILTIN宏，也能使用Intrinsic函数。

llvm/include/llvm/IR/IntrinsicsRISCV.td



```
//===----------------------------------------------------------------------===//
//
// This file defines all of the RISCV-specific intrinsics.
//
//===----------------------------------------------------------------------===//

let TargetPrefix = "riscv" in {

//===----------------------------------------------------------------------===//
// Atomics

class MaskedAtomicRMW32Intrinsic
    : Intrinsic<[llvm_i32_ty],
                [llvm_anyptr_ty, llvm_i32_ty, llvm_i32_ty, llvm_i32_ty],
                [IntrArgMemOnly, NoCapture<0>, ImmArg<3>]>;
```



```
//===----------------------------------------------------------------------===//
// Vector extension

def int_riscv_setvl : Intrinsic<[llvm_i32_ty],
                                [llvm_i32_ty, llvm_i32_ty],
                                [IntrNoMem]>;

def int_riscv_vadd : Intrinsic<[llvm_nxv1i32_ty],
                               [llvm_nxv1i32_ty, llvm_nxv1i32_ty, llvm_i32_ty],
                               [IntrNoMem]>;

def int_riscv_vsub : Intrinsic<[llvm_nxv1i32_ty],
                               [llvm_nxv1i32_ty, llvm_nxv1i32_ty, llvm_i32_ty],
                               [IntrNoMem]>;
```

int_riscv_vmul   int_riscv_vand   int_riscv_vor   int_riscv_vxor
int_riscv_vlw   int_riscv_vsw   int_riscv_vmpopcnt
int_riscv_vmfirst

• 03 llvm intrinsics函数结合llvm IR介绍

1. 什么是Intrinsic函数

Intrinsic函数是编译器内建的函数，由编译器提供，类似于内联函数。但与内联函数不同的是，因为Intrinsic函数是编译器提供，而编译器与硬件架构联系紧密，因此编译器知道如何利用硬件能力以最优的方式实现这些功能。通常函数的代码是inline插入，避免函数调用开销。LLVM支持Intrinsic函数的概念。这些函数的名称和语义可以是预先定义，也可以自定义，要求遵守特定的约定。在有些情况下，可能会调用库函数。例如，在参考文献[1]中列出的函数，都是调用libc。总的来说，这些Intrinsic函数代表了LLVM语言的一种扩展机制，当添加到语言中时，不要求改变LLVM的任何转化过程。对其它编译器，Intrinsic函数也称为内建函数。

[1] https://llvm.org/docs/ExtendingLLVM.html#intrinsic-function

在LLVM中，Intrinsic函数一般是在IR级代码优化时引入的，也就是由前端产生。也可以在程序代码中写Intrinsic函数，并通过前端直接发射。这些函数名的前缀一般是保留字“llvm.”。LLVM后端选择用最高效的形式将Intrinsic函数转换给硬件执行，可以将Intrinsic函数拆分为一系列机器指令，也可以映射为单独一条机器指令，并直接调用相应的硬件功能。下文中会针对这两种情况给出实例。

Intrinsic函数一般是外部函数，开发者不能在自己的代码中实现函数体，而只能调用这些Intrinsic函数。获得Intrinsic函数的地址是非法的。

1. 输出Intrinsic函数

以下举例说明LLVM如何通过其Intrinsic函数优化特定部分代码。

```
#include<string.h>
int foo(void){
char str[10] = "str";
return 0;
}
```

由Clang生产的LLVM IR如下：

```
define i32 @foo() #0 {
entry:
%str = alloca [10 x i8], align 1
%0 = bitcast [10 x i8]* %str to i8*
call void @llvm.memcpy.p0i8.p0i8.i64(i8* %0, i8*
getelementptr inbounds ([10 x i8]* @foo.str, i32 0, i32 0), i64
10, i32 1, i1 false)
ret i32 0
}
```

其中，llvm.memcpy就是clang输出的Intrinsic函数。如果LLVM没有定义llvm.memcpy，相应的内存操作LLVM IR代码就应该是一系列"store constant into str[0..3]"内存访问指令，而这些指令通常都是极耗时的。LLVM后端可将llvm.memcpy拆分为一系列高效机器指令，也可以映射为一条特定的机器指令，直接调用硬件的内存操作功能。

再举一例。

```
int func()
{
int a[5];
for (int i = 0; i != 5; ++i)
a[i] = 0;
return a[0];
}
```

使用Clang生成未经优化的IR代码，其中不包括任何Intrinsic函数。

```
define dso_local i32 @_Z4funcv() #0 {
entry:
%a = alloca [5 x i32], align 16
%i = alloca i32, align 4
store i32 0, i32* %i, align 4
br label %for.cond
for.cond: ; preds = %for.inc, %entry
%0 = load i32, i32* %i, align 4
%cmp = icmp ne i32 %0, 5
br i1 %cmp, label %for.body, label %for.end
for.body: ; preds = %for.cond
```

```
%1 = load i32, i32* %i, align 4
%idxprom = sext i32 %1 to i64
%arrayidx = getelementptr inbounds [5 x i32], [5 x i32]* %a, i64 0, i64 %idxprom
store i32 0, i32* %arrayidx, align 4
br label %for.inc
for.inc: ; preds = %for.body
%2 = load i32, i32* %i, align 4
%inc = add nsw i32 %2, 1
store i32 %inc, i32* %i, align 4
br label %for.cond
```

```
for.end: ; preds = %for.cond
%arrayidx1 = getelementptr inbounds [5 x i32], [5 x i32]* %a, i64 0, i64 0
%3 = load i32, i32* %arrayidx1, align 16
ret i32 %3
}
```

然后使用opt工具对IR做O1级别优化，得到IR如下：

```
define i32 @_Z4funcv() #0 {…
call void @llvm.memset.p0i8.i64(i8* %a2, i8 0, i64 20, i32 16, i1 false)
```

其中重要的优化是调用Intrinsic函数 llvm.memset.p0i8.i64为数组填0。Intrinsic函数也能用来实现代码的向量化和并行化，从而生成更优化的代码。比如，可以调用libc中最优化版本的memset。

有些Intrinsic函数可以重载，比如表示相同操作，但数据类型不同的一族函数。重载通常用来使Intrinsic函数可以在任何整数类型上操作。一个或多个参数类型或结果类型可以被重载以接受任何整数类型。

在LLVM中，被重载的Intrinsic函数名中会包括重载的参数类型，函数名中的每一个参数类型前会有一个句点。只有被重载的类型才会有名称后缀。例如，llvm.ctpop函数参数是任意宽度的整数，并且返回相同整型宽度的整数。这会引出一族函数，例如i8 @llvm.ctpop.i8(i8 %val) and i29 @llvm.ctpop.i29(i29 %val). 其中都只有一种类型被重载，函数名中也只有一种类型后缀，如.i8和.i29。以为参数类型和返回值类型匹配，二者在函数名中共用一个名称后缀。

2. 如何定义新Intrinsic函数

我会在下周，结合rvv-llvm在llvm/include/llvm/IR/Intrinsics.td新添加的两个变量进行讲解，包括整体的构建代码和相关说明。

04 参考资料

并行计算编译器分析
https://zhuanlan.zhihu.com/p/53659330

https://llvm.org/docs/ExtendingLLVM.html#intrinsic-function

PLCT实验室的rvv-llvm
https://github.com/isrc-cas/rvv-llvm/tree/rvv-iscas/ llvm

《玄铁C910指令集手册》

谢 谢

欢迎交流合作

2019/02/25