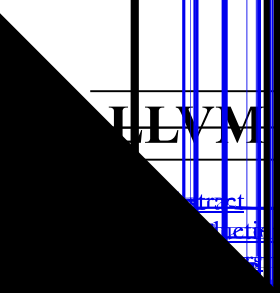
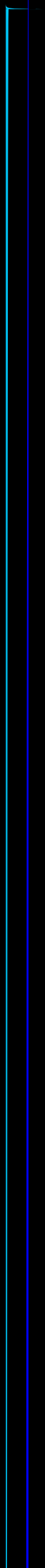




Introduction

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. The th

a
t

low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IR's", allowing many source languages to be mapped to them). By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function... allowing it to be promoted to a simple SSA value instead of a memory location.

Well-Formedness

It i

`linkonce`:
: Globals with "`linkonce`" linkage are merged with other globals of the same name when linkage occurs. This is typically used to implement inline functions, templates, or other code which must be generated in each translation unit that uses it. Unreferenced `linkonce` globals are allowed to be discarded.

`weak`:
: "`weak`" linkage has the same merging semantics as `linkonce` linkage, except that unreferenced

LLVM allows an ex

by t e target to whatever it
alig ment. All alignments mus

For ample, the following defines a

```
 = addrspace(5) constant float 1.
```

Functi s

LLV function definitions consist of the "`define`" keyord, a
convention, a return type, an optional parameter attribute for th
with tional para n tes), optional function attributes, a

Data Layout

A module may specify a target specific data layout string tha

e addresses associated with the first operand of

iable's storage.
ge of the allocated storage.

f all pointer values that contribute (directly or

he operand of the `bitcast`.
n not defined within LLVM may be associated

Overview:

The void type does not represent any value and has no size.

Syntax:

```
void
```

Label Type

Overview:

The label type represents code labels.

Syntax:

```
label
```

Metadata Type

Overview:

The metadata type represents embedded metadata. The only derived type that may contain metadata is `metadata*` or a fu

The structure type is used to represent a collection of data members together in memory. The packing of the field types is defined to match the ABI of the underlying processor. The elements of a structure may be any type that has a size.

Structures are accessed using '`load` and '`store`' by getting a pointer to a field with the '`getelementptr`' instruction.

Syntax:

```
{ <type list> }
```

Examples:

`{ i32, i32, i32 }`	A triple of three `i32` values
`{ float, i32 (i32) * }`	A pair, where the first element is a `float` and g

regate types to be used as function return types. The specific limit on how large an aggregate return type the code generator can currently handle is target-dependent, and also dependent on the aggregate element types.

Packed Structure Type

Overview:

The packed structure type is used to represent a collection of data members together in memory. There is no padding between fields. Further, the alignment of a packed structure is 1 byte. The elements of a packed structure may be any type that has a size.

Structures are accessed using '`load` and '`store`' by getting a pointer to a field with the '`getelementptr`' instruction.

Syntax:

```
< { <type list> } >
```

Examples:

`< { i32, i32, i32 } >`	A triple of three `i32` values
`< { float, i32 (i32)* } >`	A pair, where the first element is a `float` and the second element is a pointer to a function that takes an `i32`, retur

Overview:

A vector type is a simple derived type that represents a vector of elements. Vector types are used when multiple primitive data are operated in parallel using a single instruction (SIMD). A vector type requires a size (number of elements) and an underlying primitive data type. Vectors must have a power of two length (1, 2, 4, 8, 16 ...). Vector types are considered first class.

Syntax:

```
< <# elements> x <elementtype> >
```

The number of elements is a constant integer value; elementtype may be any integer or floating point type.

Examples:

`<4 x i32>`	Vector of 4 32-bit integer values.
`<8 x float>`	Vector of 8 32-bit floating-point values.
`<2 x i64>`	Vector of 2 64-bit integer values.

Note that the code generator does not yet support large vector types to be used as function return types. The specific limit on how large a vector return type codegen can currently handle is target-dependent; currently it's often a few times longer than a hardware vector register.

Opaque Type

Overview:

Opaque types are used to represent unknown types in the system. This correspon
es can eventually be resolved to any type (not just a structure type).

Syntax:

```
opaque
```

Examples:

`opaque`	An opaque type.

Type Up-references

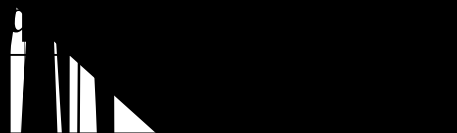
Overview:

An "up reference" allows you to refer to a lexically enclosing type without requiring it to have a name. For instance, a structure declaration may contain a pointer to any of the types it is lexically a member

i1

floating point

pointer type



```
ret <type> <value>       ; Return a value from a non-void function
ret void                 ; Return from void function
```

Overview:

The '`ret`' instruction is used to return control flow (and optionally a value) from a function back to the caller.

There are two forms of the '`ret`' instruction: one that returns a value and then causes control flow, and one that just causes control flow to occur.

Arguments:

The '`ret`' instruction optionally accepts a single argument, the return value. The type of the return value must be a '[first class](#)' type.

A function is not [well formed](#) if it it has a non-void return type and contains a '`ret`' instruction with no return value or a return value with a type that does not match its type, or if it has a void return type and contains a '`ret`' instructi

ret

unwind

calling convention

There are s

'add' Instruction

Syntax:

```
<result> = add <ty> <op1>,
<result> = add nuw <ty> <op1
<result> = add nsw <ty> <op1>,
<result> = add nuw nsw <ty> <op1
```

Overview:

The '`add`' instruction returns the sum of its two ope

Arguments:

The two arguments to the '`add`' instruction must be integer or ve

Semantics:

The value produced is the integer sum of the two operands.

If the sum has unsigned overflow, the result returned is the mathematical result m

Because LLVM integers use a two's complement representation, this instruction is appl

`nuw` and `nsw` stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` a
result value of the `add` is undefined if unsigned and/or signed overflow, respectively, occurs.

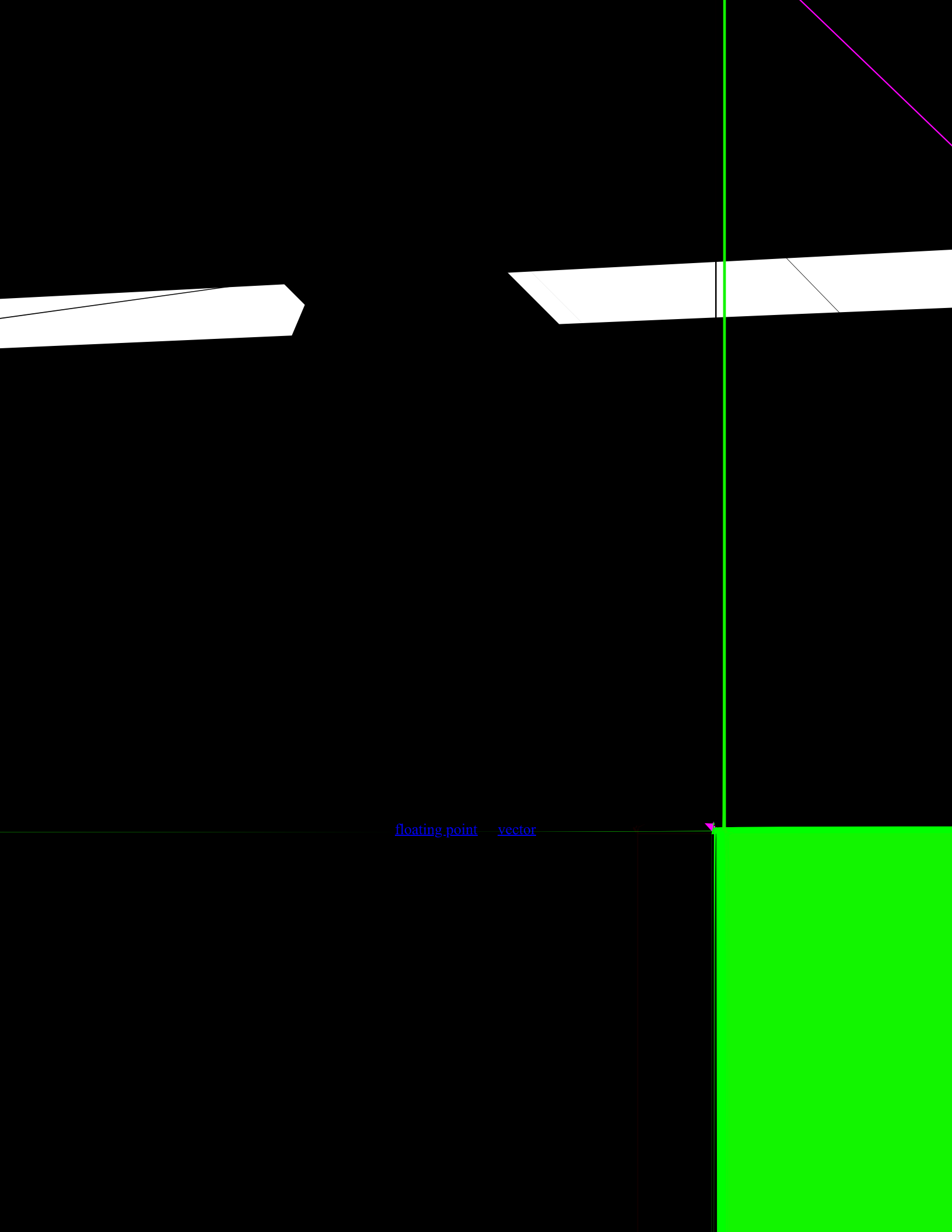
E

ields {ty}:result

operands.

'neg' instruction present in most other inter entations.

integer or vector of integer values. Both arg


floating point
vector

The 'udiv' instruction returns the quotient of its two operands.

Arguments:

The two arguments to the 'udiv' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the unsigned integer quotient of the two operands.

Note that unsigned integer division and signed integer division are distinct operations; for signed integer division, use 'sdiv'.

Division by zero leads to undefined behavior.

Example:

```
  <result> = udiv i32 4, %var          ; yields {i32}:result = 4 / %var
```

'sdiv' Instruction

Syntax:

```
  <result> = sdiv <ty> <op1>, <op2>         ; yields {ty}:result
  <result> = sdiv exact <ty> <op1>, <op2>   ; yields {ty}:result
```

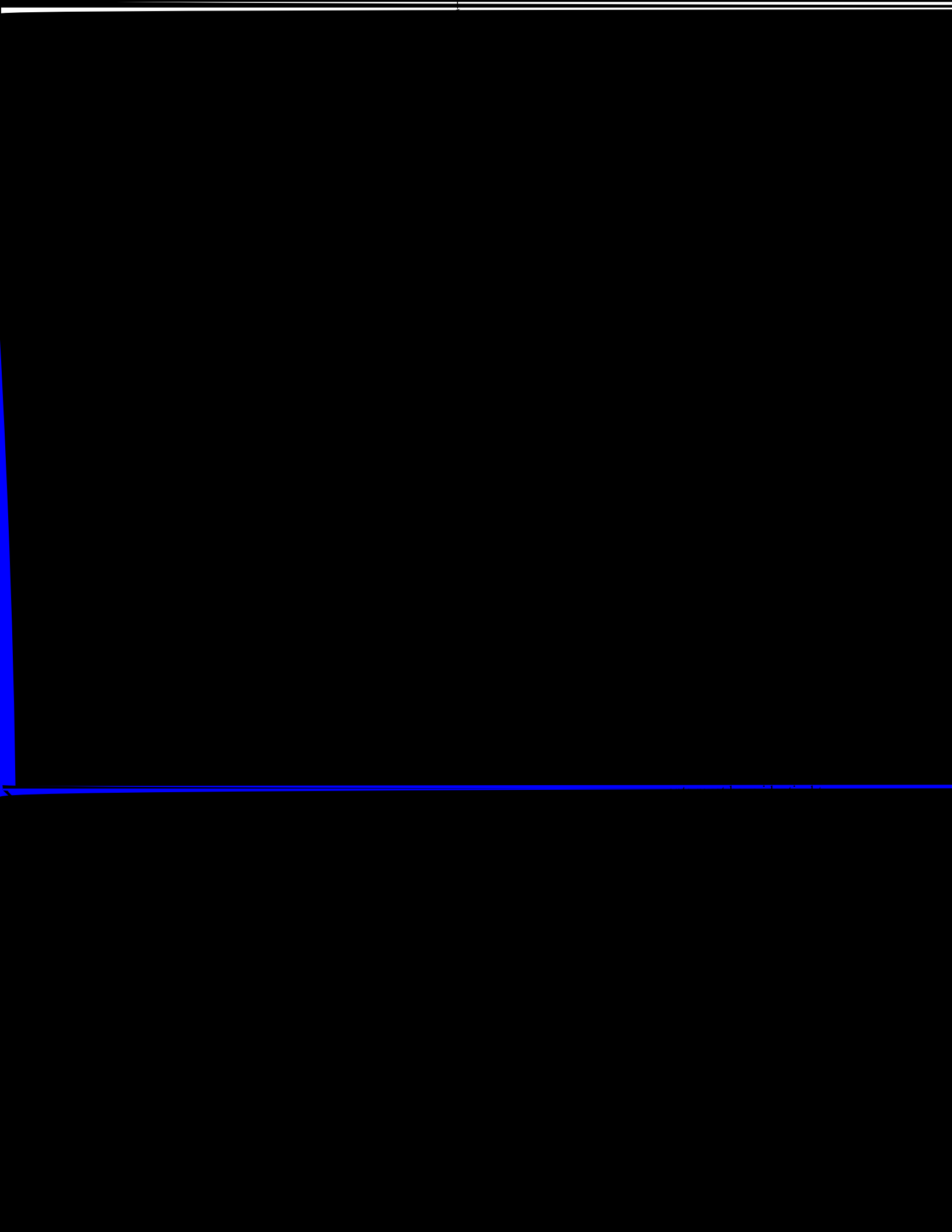
Overview:

The 'sdiv' instruction returns the quotient of its two o

Example:

```
<result> = srem i32 4, %var          ; yields {i32}:result = 4 % %var
```

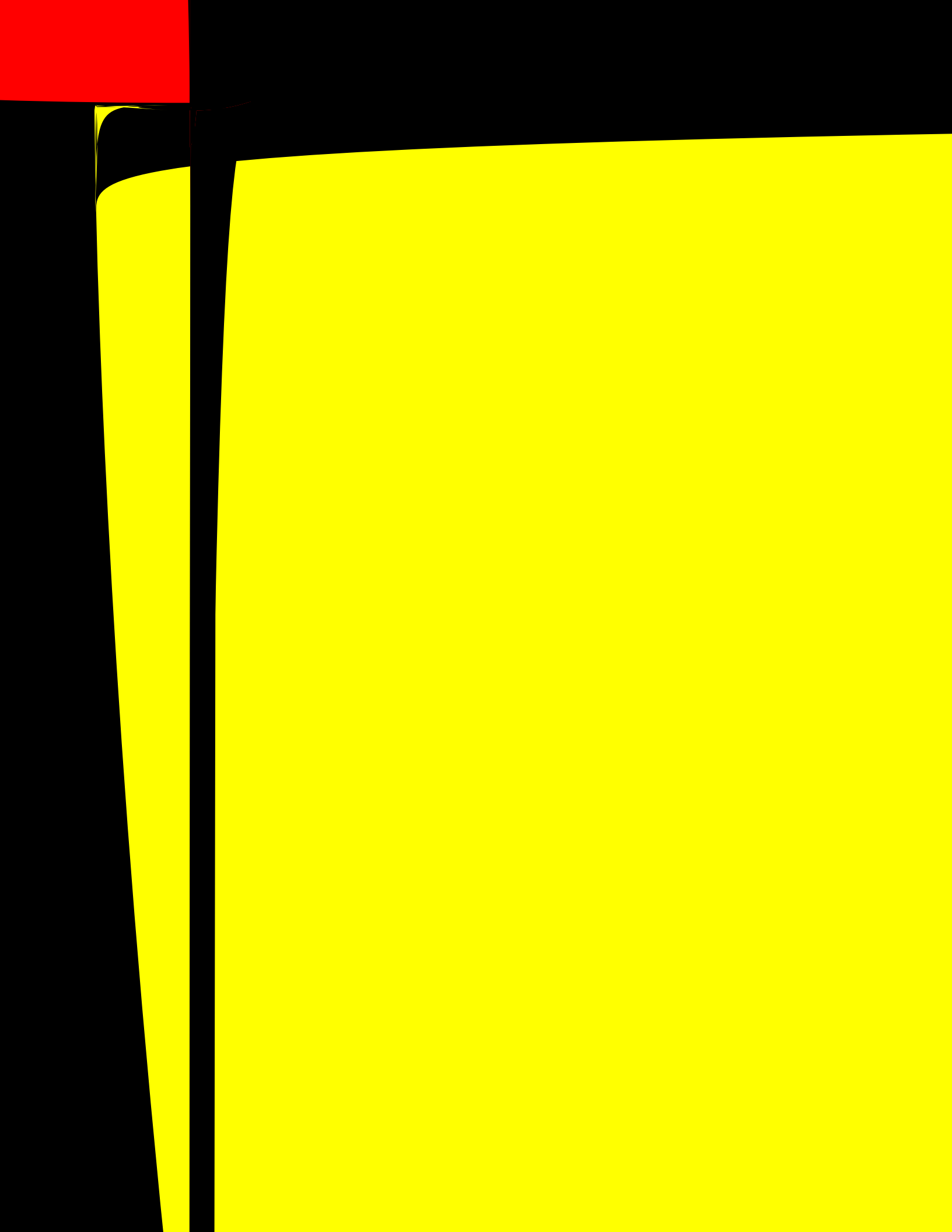
'frem' Instruction

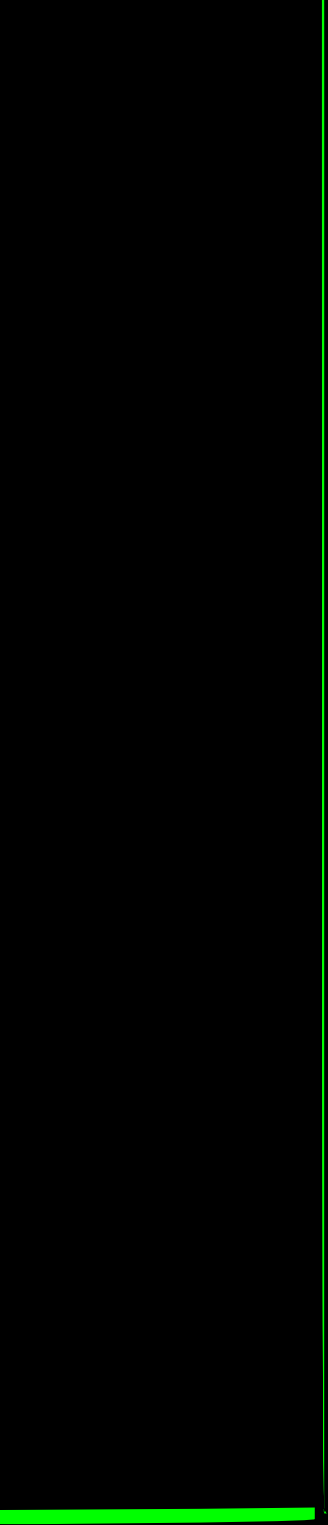


yields {ty}:result

al exclusive or of its two operands. The `xor` is used to invert

[vector]





Arguments:

The '`sext`' instruction takes a value to cast, which must be of integer type, and a type to cast it to, which must also be of integer type. The bit size of the `value` must be smaller than the bit size of the destination type, `ty2`.

Semantics:

The '`sext`' ins

[integer]

[floating point]

ype to cast it to `ty2`, which
number of elements as

he value

ds
ds
ds

int

13. `ule`: yields `true` if either operand is a QNAN or `op1` is less than or equal to `op2`.
14. `une`: yields `true` if either operand is a QNAN or `op1` is not equal to `op2`.
15. `uno`: yields `true` if eith

```
                  ne float 4.0, 5.0    ; yields: result=true
<result> = fcmp olt float 4.0, 5.0    ; yields: result=true
<result> = fcmp ueq double 1.0, 2.0   ; yields: result=false
```

Note that the code generator does not yet support vector types with the `fcmp` instruction.

'phi' Instruction

Syntax:

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

Overview:

The '`phi`' instruction is used to implement the φ node in the SSA graph representing the function.

Arguments:

The type of the incoming values is specified with the first type field. After this, the '`phi`' instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Only values of first class type may be used as the value arguments to the PHI node. Only labels may be used as the label arguments.

There must be no non-phi instructions between the start of a basic block and the PHI instructions: i.e. PHI instructions must be first in a basic block.

For the p

'va_arg' Instruction

Syntax:

```
<resultval> = va_arg <va_list*> <arglist>, <argty>
```

Overview:

The '`va_arg`' instruction is used to access arguments passed through the "variable argument" area of a function call. It is used to implement the `va_arg` macro in C.

Arguments:

This instruction takes a `va_list*` value and the type of the argument. It returns a value of the specified argument type and increments the `va_list` to point to the next argument. The actual type of `va_list` is target specific.

Semantics:

The '`va_arg`' instruction lo
ent. For more i
nation, see the variable argument handling Intrinsic Functions.

It is legal for this instru

Variable argument support is defined in LLVM with the va_arg instruction and these three intrinsic functions. These functions are related to the similarly named macros defined in the <stdarg.h> header file.

All of these functions operate on arguments that use a target-specific value type "va_list". The LLVM assembly language reference manual does not define what this type is, so all transformations should be prepared to handle these functions regardless of the type used.

This example shows how the va_arg instruction and the variable argument handling intrinsic functions are used.

```
define i32 @test(i32 %X, ...) {
  ; Initialize variable argument processing
  %ap = alloca i8*
  %ap2 = bitcast i8** %ap to i8*
  call void @llvm.va_start(i8* %ap2)

  ; Read a single integer argument
  %tmp = va_arg i8** %ap, i32

  ; Demonstrate usage of llvm.va_copy a
```

Syntax:

```
declare i8* @llvm.gcread(i8* %ObjPtr, i8** %Ptr)
```

Overview:

The '`llvm.gcread`' intrinsic identifies reads of references from heap locations, allowing garbage collector implementations that require read barriers.

Arguments:

The second argument is a

a pointer to the start of the referenced object, if needed by the allocat

Semantics:

algorithm

The argument to this intrinsic indicates which function to return the address for. Zero indicates the calling function, one indicates its caller, etc. The argument is **required** to be a constant integer value.

Semantics:

The '`llvm.returnaddress`' intrinsic either returns a pointer indicating the return address of the specified call frame, or zero if it cannot be identified. The value returned by this intrinsic is likely to be incorrect or 0 for arguments other than zero, so it should only be used for debugging purposes.

Note that calling this intrinsic does not prevent function inlining or other aggressive transformations, so the value returned may not be that of the obvious source-language caller.

'`llvm.frameaddress`' Intrinsic

Syntax:

```
declare i8 *@llvm.frameaddress(i32 <level>)
```

Overview:

The '`llvm.frameaddress`' intrinsic attempts to return the target-specific frame pointer value for the specified stack frame.

Arguments:

The argument to this intrinsic indicates which function to return the frame pointer for. Zero indicates the calling function, one indicates its caller, etc. The argument is **required** to be a constant integer value.

Semantics:

The '`llvm.frameaddress`' intrinsic either returns a pointer indicating the frame address of the specified call frame, or zero if it cannot be identified. The value returned by this intrinsic is likely to be incorrect or 0 for arguments other than zero, so it should only be used for debugging purposes.

Note that calling this intrinsic does not prevent function inlining or other aggressive transformations, so the value returned may not be that of the obvious source-language caller.

'`llvm.stacksave`' Intrinsic

Syntax:

```
declare i8 *@llvm.stacksave()
```

Overview:

The '`llvm.stacksave`' intrinsic is used to remember the current state of the function stack, for use with `llvm.stackrestore`. This is useful for ~~implementing language features like scoped automatic variable sized~~ arrays in C99.

Semantics:

This intrinsic returns a opaque pointer value that can be passed to `llvm.stackrestore`. When an `llvm.stackrestore` intrinsic is executed with a value saved from `llvm.stacksave`, it effectively restores the state of the stack to the state it was in when the `llvm.stacksave` intrinsic executed. In practice, this pops any alloca blocks from the stack that were allocated after the `llvm.stacksave` was executed.

'`llvm.stackrestore`' Intrinsic

Syntax:

```
declare void @llvm.stackrestore(i8 * %ptr)
```

Overview:

The '`llvm.readcyclecounter`' intrinsic provides access to the cycle counter register (or similar low latency, high accuracy clocks) on those targets that support it. On X86, it should map to RDTSC. On Alpha, it should map to RPCC. As the name

```
declare float     @llvm.sqrt.f32(float %Val)
declare double    @llvm.sqrt.
```

```
llvm.cos.ppcf128(ppc_fp128  %Val)
```

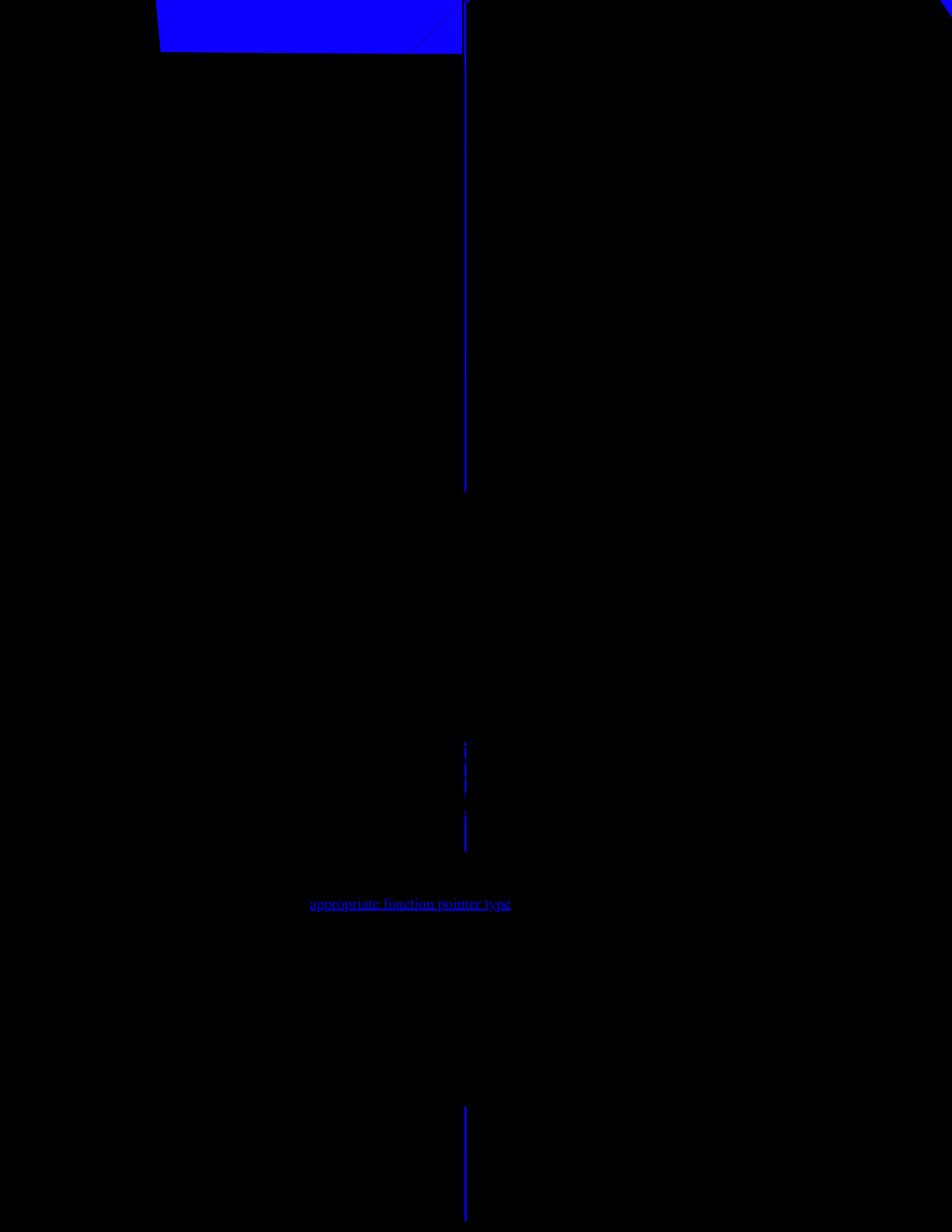
Overview:

The '`llvm.cos.*`' in

LLVM provides intrinsics for a few important bit manipulation operations. These allow efficient code generation for some

ion of the two arg whet

y bit width, ust ha


appropriate function pointer type

provided by LLVM should allow a clean implementation of all of these APIs and parallel programming models. No one model or paradigm should be selected above others unless the hardware itself ubiquitously does so.

'llvm.memory.barrier' Intrinsic

Syntax:

```
declare void @llvm.memory.barrier( i1 <ll>, i1 <ls>, i1 <sl>, i1 <ss>, i1 <device> )
```

Overview:

The `llvm.memory.barrier` intrinsic guarantees ordering between specif

This loads a value in memory and compares it to a given value. It [illegible]

```
                                        ; yields {
```

```
 val2     = add i32 1, 1
%result2  = call i32 @llvm.atomic.swap.i32.p0i32( i32* %ptr, i32 %val2 )
                                        ; yields {i32}:result2 = 8

%stored2  = icmp eq i32 %result2, 8     ; yields {i1}:stored2 = true
%memval2  = load i32* %ptr              ; yields {i32}:memval2 = 2
```

'`llvm.atomic.load.add.*`' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.atomic.load.add` on any integer bit width. Not all

The intrinsic takes two arguments, the first a pointer to an integer value and the second an integer value. The result is also an integer value. These integer types can have any bit width, but they must all have the same bit width. The targets may only lower integer representations they support.

Semantics:

This intrinsic does a series of operations atomically. It first loads the value stored at `ptr`. It then subtracts `delta`, stores the result to `ptr`. It yields the original value stored at `pt`

```
                                                    1 = 8
%result2  = call i32 @llvm.atomic.load.sub.i32.p0i32( i32*
```

```
  %ptr      = malloc i32
         store i32 0x0F0F, %ptr
  %result0  = call i32 @llvm.atomic.load.nand.i32.p0i32( i32* %ptr, i32 0xFF )
                                ; yields {i32}:result0 = 0x0F0F
  %result1  = call i32 @llvm.atomic.load.and.i32.p0i32( i32* %ptr, i32 0xFF )
                                ; yields {i32}:result1 = 0xFFFFFFF0
  %result2  = call i32 @llvm.atomic.load.or.i32.p0i32( i32* %


                                                      ult3 = FF
  %memval1  = load i32* %ptr      ; yields {i32}:memval1 = F0
```

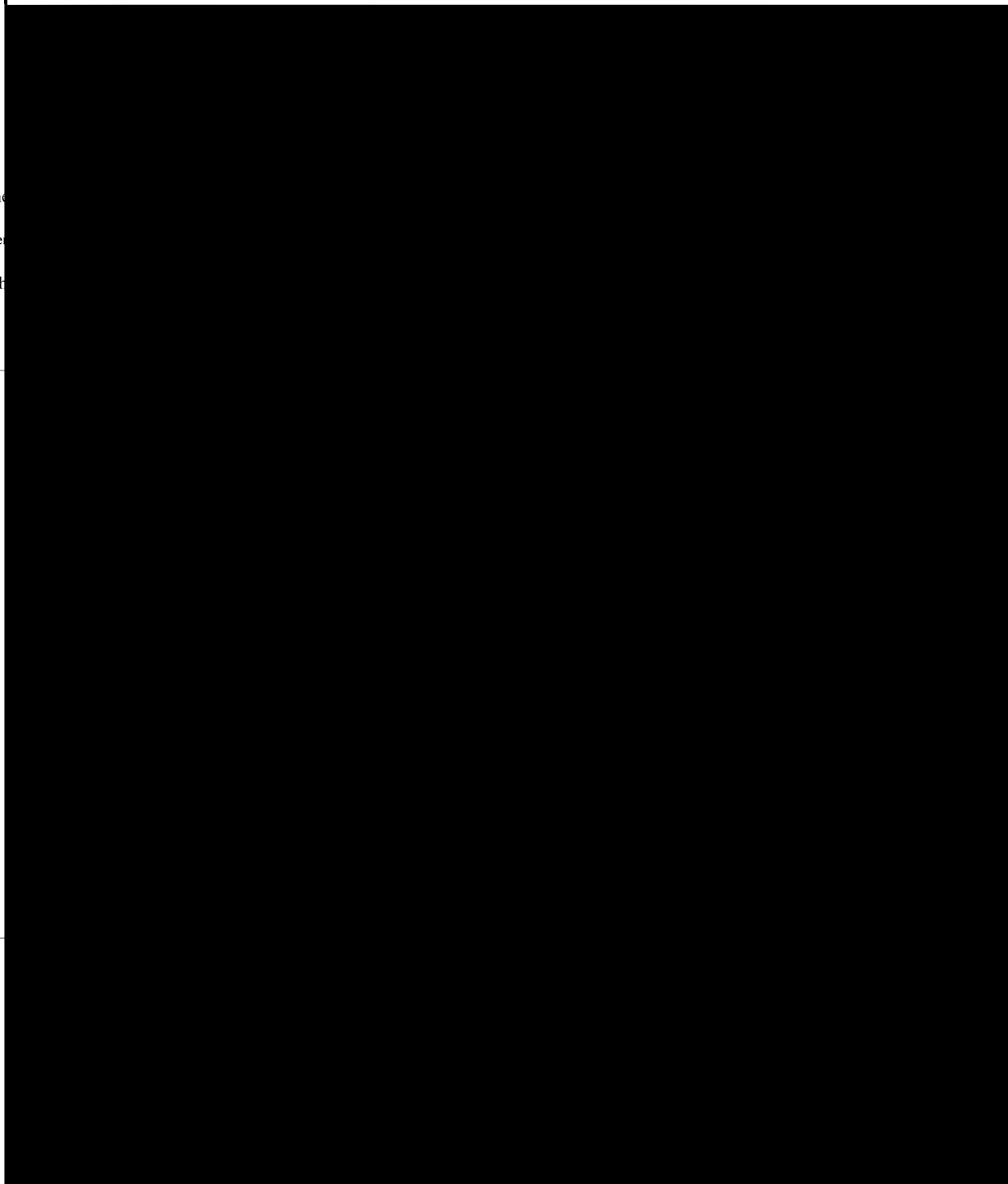
'llvm.atomic.load.max.*' Intrinsic
'llvm.atomic.load.min.*' Intrinsic
'llvm.atomic.load.umax.*

General Intrinsics

This class of intrinsics is designed to be generic and has no specific purpose.

'llvm.var.annotation' Intrinsic

Syntax:



This intrinsics is lowered to the target dependent trap instruction if the target does