cranelift Documentation

Release 0.0

Cranelift Developers

Feb 07, 2020

Contents

1 Cranelift IR Reference 3
2 Cranelift Meta Language Reference 19
3 Testing Cranelift 25
4 Register Allocation in Cranelift 33
5 Cranelift compared to LLVM 39
6 Rust Crate Documentation 43
7 Indices and tables 45
8 Todo list 47
Index 49

Contents:

CHAPTER 1

Cranelift IR Reference

Todo: Update the IR reference

This document is likely to be outdated and missing some important information. It is recommended to look at the list of instructions as documented in the *InstBuilder* documentation: https://docs.rs/cranelift-codegen/latest/cranelift_codegen/ir/trait.InstBuilder.html

The Cranelift intermediate representation (*IR*) has two primary forms: an *in-memory data structure* that the code generator library is using, and a *text format* which is used for test cases and debug output. Files containing Cranelift textual IR have the `.clif` filename extension.

This reference uses the text format to describe IR semantics but glosses over the finer details of the lexical and syntactic structure of the format.

1.1 Overall structure

Cranelift compiles functions independently. A `.clif` IR file may contain multiple functions, and the programmatic API can create multiple function handles at the same time, but the functions don't share any data or reference each other directly.

This is a simple C function that computes the average of an array of floats:

```
float
average(const float *array, size_t count)
{
    double sum = 0;
    for (size_t i = 0; i < count; i++)
        sum += array[i];
    return sum / count;
}
```

Here is the same function compiled into Cranelift IR:

```
function %average(i32, i32) -> f32 system_v {
    ss0 = explicit_slot 8         ; Stack slot for ``sum``.

block1(v0: i32, v1: i32):
    v2 = f64const 0x0.0
    stack_store v2, ss0
    brz v1, block5                  ; Handle count == 0.
    jump block2

block2:
    v3 = iconst.i32 0
    jump block3(v3)

block3(v4: i32):
    v5 = imul_imm v4, 4
    v6 = iadd v0, v5
    v7 = load.f32 v6              ; array[i]
    v8 = fpromote.f64 v7
    v9 = stack_load.f64 ss0
    v10 = fadd v8, v9
    stack_store v10, ss0
    v11 = iadd_imm v4, 1
    v12 = icmp ult v11, v1
    brnz v12, block3(v11)           ; Loop backedge.
    jump block4

block4:
    v13 = stack_load.f64 ss0
    v14 = fcvt_from_uint.f64 v1
    v15 = fdiv v13, v14
    v16 = fdemote.f32 v15
    return v16

block5:
    v100 = f32const +NaN
    return v100
}
```

The first line of a function definition provides the function *name* and the *function signature* which declares the parameter and return types. Then follows the *function preamble* which declares a number of entities that can be referenced inside the function. In the example above, the preamble declares a single explicit stack slot, `ss0`.

After the preamble follows the *function body* which consists of *extended basic blocks* (EBBs), the first of which is the *entry block*. Every EBB ends with a *terminator instruction*, so execution can never fall through to the next EBB without an explicit branch.

A `.clif` file consists of a sequence of independent function definitions:

```
function_list  ::=  { function }
function       ::=  "function" function_name signature "{" preamble function_body "}"
preamble       ::=  { preamble_decl }
function_body  ::=  { extended_basic_block }
```

1.1.1 Static single assignment form

The instructions in the function body use and produce *values* in SSA form. This means that every value is defined exactly once, and every use of a value must be dominated by the definition.

Cranelift does not have phi instructions but uses *EBB parameter*s instead. An EBB can be defined with a list of typed parameters. Whenever control is transferred to the EBB, argument values for the parameters must be provided. When entering a function, the incoming function parameters are passed as arguments to the entry EBB's parameters.

Instructions define zero, one, or more result values. All SSA values are either EBB parameters or instruction results.

In the example above, the loop induction variable `i` is represented as three SSA values: In the entry block, `v4` is the initial value. In the loop block `ebb2`, the EBB parameter `v5` represents the value of the induction variable during each iteration. Finally, `v12` is computed as the induction variable value for the next iteration.

The *cranelift_frontend* crate contains utilities for translating from programs containing multiple assignments to the same variables into SSA form for Cranelift *IR*.

Such variables can also be presented to Cranelift as *stack slot*s. Stack slots are accessed with the *stack_store* and *stack_load* instructions, and can have their address taken with *stack_addr*, which supports C-like programming languages where local variables can have their address taken.

1.2 Value types

All SSA values have a type which determines the size and shape (for SIMD vectors) of the value. Many instructions are polymorphic – they can operate on different types.

1.2.1 Boolean types

Boolean values are either true or false.

The *b1* type represents an abstract boolean value. It can only exist as an SSA value, and can't be directly stored in memory. It can, however, be converted into an integer with value 0 or 1 by the *bint* instruction (and converted back with *icmp_imm* with 0).

Several larger boolean types are also defined, primarily to be used as SIMD element types. They can be stored in memory, and are represented as either all zero bits or all one bits.

- b1
- b8
- b16
- b32
- b64

1.2.2 Integer types

Integer values have a fixed size and can be interpreted as either signed or unsigned. Some instructions will interpret an operand as a signed or unsigned number, others don't care.

The support for i8 and i16 arithmetic is incomplete and use could lead to bugs.

- i8
- i16

- i32
- i64

1.2.3 Floating point types

The floating point types have the IEEE 754 semantics that are supported by most hardware, except that non-default rounding modes, unmasked exceptions, and exception flags are not currently supported.

There is currently no support for higher-precision types like quad-precision, double-double, or extended-precision, nor for narrower-precision types like half-precision.

NaNs are encoded following the IEEE 754-2008 recommendation, with quiet NaN being encoded with the MSB of the trailing significand set to 1, and signaling NaNs being indicated by the MSB of the trailing significand set to 0.

Except for bitwise and memory instructions, NaNs returned from arithmetic instructions are encoded as follows:

- If all NaN inputs to an instruction are quiet NaNs with all bits of the trailing significand other than the MSB set to 0, the result is a quiet NaN with a nondeterministic sign bit and all bits of the trailing significand other than the MSB set to 0.
- Otherwise the result is a quiet NaN with a nondeterministic sign bit and all bits of the trailing significand other than the MSB set to nondeterministic values.
- f32
- f64

1.2.4 CPU flags types

Some target ISAs use CPU flags to represent the result of a comparison. These CPU flags are represented as two value types depending on the type of values compared.

Since some ISAs don't have CPU flags, these value types should not be used until the legalization phase of compilation where the code is adapted to fit the target ISA. Use instructions like *icmp* instead.

The CPU flags types are also restricted such that two flags values can not be live at the same time. After legalization, some instruction encodings will clobber the flags, and flags values are not allowed to be live across such instructions either. The verifier enforces these rules.

- iflags
- fflags

1.2.5 SIMD vector types

A SIMD vector type represents a vector of values from one of the scalar types (boolean, integer, and floating point). Each scalar value in a SIMD type is called a *lane*. The number of lanes must be a power of two in the range 2-256.

i%Bx%N A SIMD vector of integers. The lane type *iB* is one of the integer types *i8* … *i64*.

Some concrete integer vector types are *i32x4*, *i64x8*, and *i16x4*.

The size of a SIMD integer vector in memory is $\frac{NB}{8}$ bytes.

f32x%N A SIMD vector of single precision floating point numbers.

Some concrete *f32* vector types are: *f32x2*, *f32x4*, and *f32x8*.

The size of a *f32* vector in memory is $4N$ bytes.

f64x%N A SIMD vector of double precision floating point numbers.

Some concrete *f64* vector types are: *f64x2*, *f64x4*, and *f64x8*.

The size of a *f64* vector in memory is $8N$ bytes.

b1x%N A boolean SIMD vector.

Boolean vectors are used when comparing SIMD vectors. For example, comparing two *i32x4* values would produce a *b1x4* result.

Like the *b1* type, a boolean vector cannot be stored in memory.

1.2.6 Pseudo-types and type classes

These are not concrete types, but convenient names used to refer to real types in this reference.

iAddr A Pointer-sized integer representing an address.

This is either *i32*, or *i64*, depending on whether the target platform has 32-bit or 64-bit pointers.

iB Any of the scalar integer types *i8* – *i64*.

Int Any scalar *or vector* integer type: *iB* or *iBxN*.

fB Either of the floating point scalar types: *f32* or *f64*.

Float Any scalar *or vector* floating point type: *fB* or *fBxN*.

%Tx%N Any SIMD vector type.

Mem Any type that can be stored in memory: *Int* or *Float*.

Testable Either *b1* or *iN*.

1.2.7 Immediate operand types

These types are not part of the normal SSA type system. They are used to indicate the different kinds of immediate operands on an instruction.

imm64 A 64-bit immediate integer. The value of this operand is interpreted as a signed two's complement integer. Instruction encodings may limit the valid range.

In the textual format, *imm64* immediates appear as decimal or hexadecimal literals using the same syntax as C.

offset32 A signed 32-bit immediate address offset.

In the textual format, *offset32* immediates always have an explicit sign, and a 0 offset may be omitted.

ieee32 A 32-bit immediate floating point number in the IEEE 754-2008 binary32 interchange format. All bit patterns are allowed.

ieee64 A 64-bit immediate floating point number in the IEEE 754-2008 binary64 interchange format. All bit patterns are allowed.

bool A boolean immediate value, either false or true.

In the textual format, *bool* immediates appear as 'false' and 'true'.

intcc An integer condition code. See the *icmp* instruction for details.

floatcc A floating point condition code. See the *fcmp* instruction for details.

The two IEEE floating point immediate types *ieee32* and *ieee64* are displayed as hexadecimal floating point literals in the textual *IR* format. Decimal floating point literals are not allowed because some computer systems can round differently when converting to binary. The hexadecimal floating point format is mostly the same as the one used by C99, but extended to represent all NaN bit patterns:

Normal numbers Compatible with C99: `-0x1.Tpe` where `T` are the trailing significand bits encoded as hexadecimal, and `e` is the unbiased exponent as a decimal number. *ieee32* has 23 trailing significand bits. They are padded with an extra LSB to produce 6 hexadecimal digits. This is not necessary for *ieee64* which has 52 trailing significand bits forming 13 hexadecimal digits with no padding.

Zeros Positive and negative zero are displayed as `0.0` and `-0.0` respectively.

Subnormal numbers Compatible with C99: `-0x0.Tpemin` where `T` are the trailing significand bits encoded as hexadecimal, and `emin` is the minimum exponent as a decimal number.

Infinities Either `-Inf` or `Inf`.

Quiet NaNs Quiet NaNs have the MSB of the trailing significand set. If the remaining bits of the trailing significand are all zero, the value is displayed as `-NaN` or `NaN`. Otherwise, `-NaN:0xT` where `T` are the trailing significand bits encoded as hexadecimal.

Signaling NaNs Displayed as `-sNaN:0xT`.

1.3 Control flow

Branches transfer control to a new EBB and provide values for the target EBB's arguments, if it has any. Conditional branches only take the branch if their condition is satisfied, otherwise execution continues at the following instruction in the EBB.

JT = jump_table [EBB0, EBB1, . . . , EBBn] Declare a jump table in the *function preamble*.

This declares a jump table for use by the *br_table* indirect branch instruction. Entries in the table are EBB names.

The EBBs listed must belong to the current function, and they can't have any arguments.

arg EBB0 Target EBB when `x = 0`.

arg EBB1 Target EBB when `x = 1`.

arg EBBn Target EBB when `x = n`.

result A jump table identifier. (Not an SSA value).

Traps stop the program because something went wrong. The exact behavior depends on the target instruction set architecture and operating system. There are explicit trap instructions defined below, but some instructions may also cause traps for certain input value. For example, *udiv* traps when the divisor is zero.

1.4 Function calls

A function call needs a target function and a *function signature*. The target function may be determined dynamically at runtime, but the signature must be known when the function call is compiled. The function signature describes how to call the function, including parameters, return values, and the calling convention:

```
signature     ::=  "(" [paramlist] ")" ["->" retlist] [call_conv]
paramlist     ::=  param { "," param }
```

```
retlist       ::=  paramlist
param         ::=  type [paramext] [paramspecial]
paramext      ::=  "uext" | "sext"
paramspecial  ::=  "sret" | "link" | "fp" | "csr" | "vmctx" | "sigid" | "stack_limit"
callconv      ::=  "fast" | "cold" | "system_v" | "fastcall" | "baldrdash_system_v" | "bal
```

A function's calling convention determines exactly how arguments and return values are passed, and how stack frames are managed. Since all of these details depend on both the instruction set /// architecture and possibly the operating system, a function's calling convention is only fully determined by a *(TargetIsa, CallConv)* tuple.

Name	Description
sret	pointer to a return value in memory
link	return address
fp	the initial value of the frame pointer
csr	callee-saved register
vmctx	VM context pointer, which may contain pointers to heaps etc.
sigid	signature id, for checking caller/callee signature compatibility
stack_limit	limit value for the size of the stack

The "not-ABI-stable" conventions do not follow an external specification and may change between versions of Cranelift.

The "fastcall" convention is not yet implemented.

Parameters and return values have flags whose meaning is mostly target dependent. These flags support interfacing with code produced by other compilers.

Functions that are called directly must be declared in the *function preamble*:

FN = [colocated] NAME signature Declare a function so it can be called directly.

If the colocated keyword is present, the symbol's definition will be defined along with the current function, such that it can use more efficient addressing.

arg NAME Name of the function, passed to the linker for resolution.

arg signature Function signature. See below.

result FN A function identifier that can be used with *call*.

This simple example illustrates direct function calls and signatures:

```
function %gcd(i32 uext, i32 uext) -> i32 uext system_v {
    fn0 = %divmod(i32 uext, i32 uext) -> i32 uext, i32 uext

block1(v0: i32, v1: i32):
    brz v1, block3
    jump block2

block2:
    v2, v3 = call fn0(v0, v1)
    return v2

block3:
    return v0
}
```

Indirect function calls use a signature declared in the preamble.

1.5 Memory

Cranelift provides fully general *load* and *store* instructions for accessing memory, as well as *extending loads and truncating stores*.

If the memory at the given address is not *addressable*, the behavior of these instructions is undefined. If it is addressable but not *accessible*, they *trap*.

There are also more restricted operations for accessing specific types of memory objects.

Additionally, instructions are provided for handling multi-register addressing.

1.5.1 Memory operation flags

Loads and stores can have flags that loosen their semantics in order to enable optimizations.

Flag	Description
notrap	Memory is assumed to be *accessible*.
aligned	Trapping allowed for misaligned accesses.
readonly	The data at the specified address will not modified between when this function is called and exited.

When the `accessible` flag is set, the behavior is undefined if the memory is not *accessible*.

Loads and stores are *misaligned* if the resultant address is not a multiple of the expected alignment. By default, misaligned loads and stores are allowed, but when the `aligned` flag is set, a misaligned memory access is allowed to *trap*.

1.5.2 Explicit Stack Slots

One set of restricted memory operations access the current function's stack frame. The stack frame is divided into fixed-size stack slots that are allocated in the *function preamble*. Stack slots are not typed, they simply represent a contiguous sequence of *accessible* bytes in the stack frame.

SS = explicit_slot Bytes, Flags... Allocate a stack slot in the preamble.

If no alignment is specified, Cranelift will pick an appropriate alignment for the stack slot based on its size and access patterns.

arg Bytes Stack slot size on bytes.

flag align(N) Request at least N bytes alignment.

result SS Stack slot index.

The dedicated stack access instructions are easy for the compiler to reason about because stack slots and offsets are fixed at compile time. For example, the alignment of these stack memory accesses can be inferred from the offsets and stack slot alignments.

It's also possible to obtain the address of a stack slot, which can be used in *unrestricted loads and stores*.

The *stack_addr* instruction can be used to macro-expand the stack access instructions before instruction selection:

```
v0 = stack_load.f64 ss3, 16
; Expands to:
v1 = stack_addr ss3, 16
v0 = load.f64 v1
```

When Cranelift code is running in a sandbox, it can also be necessary to include stack overflow checks in the prologue.

1.5.3 Global values

A *global value* is an object whose value is not known at compile time. The value is computed at runtime by *global_value*, possibly using information provided by the linker via relocations. There are multiple kinds of global values using different methods for determining their value. Cranelift does not track the type of a global value, for they are just values stored in non-stack memory.

When Cranelift is generating code for a virtual machine environment, globals can be used to access data structures in the VM's runtime. This requires functions to have access to a *VM context pointer* which is used as the base address. Typically, the VM context pointer is passed as a hidden function argument to Cranelift functions.

Chains of global value expressions are possible, but cycles are not allowed. They will be caught by the IR verifier.

GV = vmctx Declare a global value of the address of the VM context struct.

This declares a global value which is the VM context pointer which may be passed as a hidden argument to functions JIT-compiled for a VM.

Typically, the VM context is a *#[repr(C, packed)]* struct.

result GV Global value.

A global value can also be derived by treating another global variable as a struct pointer and loading from one of its fields. This makes it possible to chase pointers into VM runtime data structures.

GV = load.Type BaseGV [Offset] Declare a global value pointed to by BaseGV plus Offset, with type Type.

It is assumed the BaseGV plus Offset resides in accessible memory with the appropriate alignment for storing a value with type Type.

arg BaseGV Global value providing the base pointer.

arg Offset Offset added to the base before loading.

result GV Global value.

GV = iadd_imm BaseGV, Offset Declare a global value which has the value of BaseGV offset by Offset.

arg BaseGV Global value providing the base value.

arg Offset Offset added to the base value.

GV = [colocated] symbol Name Declare a symbolic address global value.

The value of GV is symbolic and will be assigned a relocation, so that it can be resolved by a later linking phase.

If the colocated keyword is present, the symbol's definition will be defined along with the current function, such that it can use more efficient addressing.

arg Name External name.

result GV Global value.

1.5.4 Heaps

Code compiled from WebAssembly or asm.js runs in a sandbox where it can't access all process memory. Instead, it is given a small set of memory areas to work in, and all accesses are bounds checked. Cranelift models this through the concept of *heaps*.

A heap is declared in the function preamble and can be accessed with the *heap_addr* instruction that *traps* on out-of-bounds accesses or returns a pointer that is guaranteed to trap. Heap addresses can be smaller than the native pointer size, for example unsigned *i32* offsets on a 64-bit architecture.

A heap appears as three consecutive ranges of address space:

Fig. 1: Heap address space layout

1. The *mapped pages* are the *accessible* memory range in the heap. A heap may have a minimum guaranteed size which means that some mapped pages are always present.
2. The *unmapped pages* is a possibly empty range of address space that may be mapped in the future when the heap is grown. They are *addressable* but not *accessible*.
3. The *offset-guard pages* is a range of address space that is guaranteed to always cause a trap when accessed. It is used to optimize bounds checking for heap accesses with a shared base pointer. They are *addressable* but not *accessible*.

The *heap bound* is the total size of the mapped and unmapped pages. This is the bound that *heap_addr* checks against. Memory accesses inside the heap bounds can trap if they hit an unmapped page (which is not *accessible*).

Two styles of heaps are supported, *static* and *dynamic*. They behave differently when resized.

Static heaps

A *static heap* starts out with all the address space it will ever need, so it never moves to a different address. At the base address is a number of mapped pages corresponding to the heap's current size. Then follows a number of unmapped pages where the heap can grow up to its maximum size. After the unmapped pages follow the offset-guard pages which are also guaranteed to generate a trap when accessed.

H = static Base, min MinBytes, bound BoundBytes, offset_guard OffsetGuardBytes Declare a static heap in the preamble.

arg Base Global value holding the heap's base address.

arg MinBytes Guaranteed minimum heap size in bytes. Accesses below this size will never trap.

arg BoundBytes Fixed heap bound in bytes. This defines the amount of address space reserved for the heap, not including the offset-guard pages.

arg OffsetGuardBytes Size of the offset-guard pages in bytes.

Dynamic heaps

A *dynamic heap* can be relocated to a different base address when it is resized, and its bound can move dynamically. The offset-guard pages move when the heap is resized. The bound of a dynamic heap is stored in a global value.

H = dynamic Base, min MinBytes, bound BoundGV, offset_guard OffsetGuardBytes Declare a dynamic heap in the preamble.

arg Base Global value holding the heap's base address.

arg MinBytes Guaranteed minimum heap size in bytes. Accesses below this size will never trap.

arg BoundGV Global value containing the current heap bound in bytes.

arg OffsetGuardBytes Size of the offset-guard pages in bytes.

Heap examples

The SpiderMonkey VM prefers to use fixed heaps with a 4 GB bound and 2 GB of offset-guard pages when running WebAssembly code on 64-bit CPUs. The combination of a 4 GB fixed bound and 1-byte bounds checks means that no code needs to be generated for bounds checks at all:

```
function %add_members(i32, i64 vmctx) -> f32 baldrdash_system_v {
    gv0 = vmctx
    gv1 = load.i64 notrap aligned gv0+64
    heap0 = static gv1, min 0x1000, bound 0x1_0000_0000, offset_guard 0x8000_0000

block0(v0: i32, v5: i64):
    v1 = heap_addr.i64 heap0, v0, 1
    v2 = load.f32 v1+16
    v3 = load.f32 v1+20
    v4 = fadd v2, v3
    return v4
}
```

A static heap can also be used for 32-bit code when the WebAssembly module declares a small upper bound on its memory. A 1 MB static bound with a single 4 KB offset-guard page still has opportunities for sharing bounds checking code:

```
function %add_members(i32, i32 vmctx) -> f32 baldrdash_system_v {
    gv0 = vmctx
    gv1 = load.i32 notrap aligned gv0+64
    heap0 = static gv1, min 0x1000, bound 0x10_0000, offset_guard 0x1000

block0(v0: i32, v5: i32):
    v1 = heap_addr.i32 heap0, v0, 1
    v2 = load.f32 v1+16
    v3 = load.f32 v1+20
    v4 = fadd v2, v3
    return v4
}
```

If the upper bound on the heap size is too large, a dynamic heap is required instead.

Finally, a runtime environment that simply allocates a heap with `malloc()` may not have any offset-guard pages at all. In that case, full bounds checking is required for each access:

```
function %add_members(i32, i64 vmctx) -> f32 baldrdash_system_v {
    gv0 = vmctx
    gv1 = load.i64 notrap aligned gv0+64
    gv2 = load.i32 notrap aligned gv0+72
    heap0 = dynamic gv1, min 0x1000, bound gv2, offset_guard 0

block0(v0: i32, v6: i64):
    v1 = heap_addr.i64 heap0, v0, 20
    v2 = load.f32 v1+16
    v3 = heap_addr.i64 heap0, v0, 24
    v4 = load.f32 v3+20
    v5 = fadd v2, v4
```

(continues on next page)

(continued from previous page)

```
    return v5
}
```

1.5.5 Tables

Code compiled from WebAssembly often needs access to objects outside of its linear memory. WebAssembly uses *tables* to allow programs to refer to opaque values through integer indices.

A table is declared in the function preamble and can be accessed with the *table_addr* instruction that *traps* on out-of-bounds accesses. Table addresses can be smaller than the native pointer size, for example unsigned *i32* offsets on a 64-bit architecture.

A table appears as a consecutive range of address space, conceptually divided into elements of fixed sizes, which are identified by their index. The memory is *accessible*.

The *table bound* is the number of elements currently in the table. This is the bound that *table_addr* checks against.

A table can be relocated to a different base address when it is resized, and its bound can move dynamically. The bound of a table is stored in a global value.

T = dynamic Base, min MinElements, bound BoundGV, element_size ElementSize Declare a table in the preamble.

arg Base Global value holding the table's base address.

arg MinElements Guaranteed minimum table size in elements.

arg BoundGV Global value containing the current heap bound in elements.

arg ElementSize Size of each element.

1.5.6 Constant materialization

A few instructions have variants that take immediate operands, but in general an instruction is required to load a constant into an SSA value: *iconst*, *f32const*, *f64const* and *bconst* serve this purpose.

1.5.7 Bitwise operations

The bitwise operations and operate on any value type: Integers, floating point numbers, and booleans. When operating on integer or floating point types, the bitwise operations are working on the binary representation of the values. When operating on boolean values, the bitwise operations work as logical operators.

The shift and rotate operations only work on integer types (scalar and vector). The shift amount does not have to be the same type as the value being shifted. Only the low *B* bits of the shift amount is significant.

When operating on an integer vector type, the shift amount is still a scalar type, and all the lanes are shifted the same amount. The shift amount is masked to the number of bits in a *lane*, not the full size of the vector type.

The bit-counting instructions are scalar only.

1.5.8 Floating point operations

These operations generally follow IEEE 754-2008 semantics.

Sign bit manipulations

The sign manipulating instructions work as bitwise operations, so they don't have special behavior for signaling NaN operands. The exponent and trailing significand bits are always preserved.

Minimum and maximum

These instructions return the larger or smaller of their operands. Note that unlike the IEEE 754-2008 *minNum* and *maxNum* operations, these instructions return NaN when either input is NaN.

When comparing zeroes, these instructions behave as if $-0.0 < 0.0$.

Rounding

These instructions round their argument to a nearby integral value, still represented as a floating point number.

1.5.9 Conversion operations

1.5.10 Extending loads and truncating stores

Most ISAs provide instructions that load an integer value smaller than a register and extends it to the width of the register. Similarly, store instructions that only write the low bits of an integer register are common.

In addition to the normal *load* and *store* instructions, Cranelift provides extending loads and truncation stores for 8, 16, and 32-bit memory accesses.

These instructions succeed, trap, or have undefined behavior, under the same conditions as *normal loads and stores*.

1.6 ISA-specific instructions

Target ISAs can define supplemental instructions that do not make sense to support generally.

1.6.1 x86

Instructions that can only be used by the x86 target ISA.

1.7 Codegen implementation instructions

Frontends don't need to emit the instructions in this section themselves; Cranelift will generate them automatically as needed.

1.7.1 Legalization operations

These instructions are used as helpers when legalizing types and operations for the target ISA.

1.7.2 Special register operations

The prologue and epilogue of a function needs to manipulate special registers like the stack pointer and the frame pointer. These instructions should not be used in regular code.

1.7.3 CPU flag operations

These operations are for working with the “flags” registers of some CPU architectures.

1.7.4 Live range splitting

Cranelift’s register allocator assigns each SSA value to a register or a spill slot on the stack for its entire live range. Since the live range of an SSA value can be quite large, it is sometimes beneficial to split the live range into smaller parts.

A live range is split by creating new SSA values that are copies or the original value or each other. The copies are created by inserting *copy*, *spill*, or *fill* instructions, depending on whether the values are assigned to registers or stack slots.

This approach permits SSA form to be preserved throughout the register allocation pass and beyond.

Register values can be temporarily diverted to other registers by the *regmove* instruction, and to and from stack slots by *regspill* and *regfill*.

1.8 Instruction groups

All of the shared instructions are part of the *base* instruction group.

Target ISAs may define further instructions in their own instruction groups.

1.9 Implementation limits

Cranelift’s intermediate representation imposes some limits on the size of functions and the number of entities allowed. If these limits are exceeded, the implementation will panic.

Number of instructions in a function At most $2^{31} - 1$.

Number of EBBs in a function At most $2^{31} - 1$.

Every EBB needs at least a terminator instruction anyway.

Number of secondary values in a function At most $2^{31} - 1$.

Secondary values are any SSA values that are not the first result of an instruction.

Other entities declared in the preamble At most $2^{32} - 1$.

This covers things like stack slots, jump tables, external functions, and function signatures, etc.

Number of arguments to an EBB At most 2^{16}.

Number of arguments to a function At most 2^{16}.

This follows from the limit on arguments to the entry EBB. Note that Cranelift may add a handful of ABI register arguments as function signatures are lowered. This is for representing things like the link register, the incoming frame pointer, and callee-saved registers that are saved in the prologue.

Size of function call arguments on the stack At most $2^{32} - 1$ bytes.

This is probably not possible to achieve given the limit on the number of arguments, except by requiring extremely large offsets for stack arguments.

1.10 Glossary

addressable Memory in which loads and stores have defined behavior. They either succeed or *trap*, depending on whether the memory is *accessible*.

accessible *Addressable* memory in which loads and stores always succeed without *trapping*, except where specified otherwise (eg. with the *aligned* flag). Heaps, globals, tables, and the stack may contain accessible, merely addressable, and outright unaddressable regions. There may also be additional regions of addressable and/or accessible memory not explicitly declared.

basic block A maximal sequence of instructions that can only be entered from the top, and that contains no branch or terminator instructions except for the last instruction.

entry block The *EBB* that is executed first in a function. Currently, a Cranelift function must have exactly one entry block which must be the first block in the function. The types of the entry block arguments must match the types of arguments in the function signature.

extended basic block

EBB A maximal sequence of instructions that can only be entered from the top, and that contains no *terminator instructions* except for the last one. An EBB can contain conditional branches that can fall through to the following instructions in the block, but only the first instruction in the EBB can be a branch target.

The last instruction in an EBB must be a *terminator instruction*, so execution cannot flow through to the next EBB in the function. (But there may be a branch to the next EBB.)

Note that some textbooks define an EBB as a maximal *subtree* in the control flow graph where only the root can be a join node. This definition is not equivalent to Cranelift EBBs.

EBB parameter A formal parameter for an EBB is an SSA value that dominates everything in the EBB. For each parameter declared by an EBB, a corresponding argument value must be passed when branching to the EBB. The function's entry EBB has parameters that correspond to the function's parameters.

EBB argument Similar to function arguments, EBB arguments must be provided when branching to an EBB that declares formal parameters. When execution begins at the top of an EBB, the formal parameters have the values of the arguments passed in the branch.

function signature A function signature describes how to call a function. It consists of:

- The calling convention.
- The number of arguments and return values. (Functions can return multiple values.)
- Type and flags of each argument.
- Type and flags of each return value.

Not all function attributes are part of the signature. For example, a function that never returns could be marked as `noreturn`, but that is not necessary to know when calling it, so it is just an attribute, and not part of the signature.

function preamble A list of declarations of entities that are used by the function body. Some of the entities that can be declared in the preamble are:

- Stack slots.
- Functions that are called directly.

- Function signatures for indirect function calls.
- Function flags and attributes that are not part of the signature.

function body The extended basic blocks which contain all the executable code in a function. The function body follows the function preamble.

intermediate representation

IR The language used to describe functions to Cranelift. This reference describes the syntax and semantics of Cranelift IR. The IR has two forms: Textual, and an in-memory data structure.

stack slot A fixed size memory allocation in the current function's activation frame. These include *explicit stack slot*s and *spill stack slot*s.

explicit stack slot A fixed size memory allocation in the current function's activation frame. These differ from *spill stack slot*s in that they can be created by frontends and they may have their addresses taken.

spill stack slot A fixed size memory allocation in the current function's activation frame. These differ from *explicit stack slot*s in that they are only created during register allocation, and they may not have their address taken.

terminator instruction A control flow instruction that unconditionally directs the flow of execution somewhere else. Execution never continues at the instruction following a terminator instruction.

The basic terminator instructions are *br*, *return*, and *trap*. Conditional branches and instructions that trap conditionally are not terminator instructions.

trap

traps

trapping Terminates execution of the current thread. The specific behavior after a trap depends on the underlying OS. For example, a common behavior is delivery of a signal, with the specific signal depending on the event that triggered it.

CHAPTER 2

Cranelift Meta Language Reference

The Cranelift meta language is used to define instructions for Cranelift. It is a domain specific language embedded in Rust.

Todo: Point to the Rust documentation of the meta crate here.

This document is very out-of-date. Instead, you can have a look at the work-in-progress documentation of the *meta* crate there: https://docs.rs/cranelift-codegen-meta/0.34.0/cranelift_codegen_meta/.

This document describes the Python modules that form the embedded DSL.

The meta language descriptions are Python modules under the *cranelift-codegen/meta-python* directory. The descriptions are processed in two steps:

1. The Python modules are imported. This has the effect of building static data structures in global values in the modules. These static data structures in the *base* and *isa* packages use the classes in the *cdsl* package to describe instruction sets and other properties.
2. The static data structures are processed to produce Rust source code and constant tables.

The main driver for this source code generation process is the *cranelift-codegen/meta-python/build.py* script which is invoked as part of the build process if anything in the *cranelift-codegen/meta-python* directory has changed since the last build.

2.1 Settings

Settings are used by the environment embedding Cranelift to control the details of code generation. Each setting is defined in the meta language so a compact and consistent Rust representation can be generated. Shared settings are defined in the *base.settings* module. Some settings are specific to a target ISA, and defined in a *settings.py* module under the appropriate *cranelift-codegen/meta-python/isa/** directory.

Settings can take boolean on/off values, small numbers, or explicitly enumerated symbolic values.

All settings must belong to a *group*, represented by a `SettingGroup` object.

Normally, a setting group corresponds to all settings defined in a module. Such a module looks like this:

```
group = SettingGroup('example')

foo = BoolSetting('use the foo')
bar = BoolSetting('enable bars', True)
opt = EnumSetting('optimization level', 'Debug', 'Release')

group.close(globals())
```

2.2 Instruction descriptions

New instructions are defined as instances of the `Instruction` class. As instruction instances are created, they are added to the currently open `InstructionGroup`.

The basic Cranelift instruction set described in *Cranelift IR Reference* is defined by the Python module *base.instructions*. This module has a global value *base.instructions.GROUP* which is an `InstructionGroup` instance containing all the base instructions.

An instruction is defined with a set of distinct input and output operands which must be instances of the `Operand` class.

Cranelift uses two separate type systems for operand kinds and SSA values.

2.2.1 Type variables

Instruction descriptions can be made polymorphic by using `cdsl.operands.Operand` instances that refer to a *type variable* instead of a concrete value type. Polymorphism only works for SSA value operands. Other operands have a fixed operand kind.

If multiple operands refer to the same type variable they will be required to have the same concrete type. For example, this defines an integer addition instruction:

```
Int = TypeVar('Int', 'A scalar or vector integer type', ints=True, simd=True)
a = Operand('a', Int)
x = Operand('x', Int)
y = Operand('y', Int)

iadd = Instruction('iadd', 'Integer addition', ins=(x, y), outs=a)
```

The type variable *Int* is allowed to vary over all scalar and vector integer value types, but in a given instance of the *iadd* instruction, the two operands must have the same type, and the result will be the same type as the inputs.

There are some practical restrictions on the use of type variables, see *Restricted polymorphism*.

2.2.2 Immediate operands

Immediate instruction operands don't correspond to SSA values, but have values that are encoded directly in the instruction. Immediate operands don't have types from the `cdsl.types.ValueType` type system; they often have enumerated values of a specific type. The type of an immediate operand is indicated with an instance of `ImmediateKind`.

2.2.3 Entity references

Instruction operands can also refer to other entities in the same function. This can be extended basic blocks, or entities declared in the function preamble.

2.2.4 Value types

Concrete value types are represented as instances of `ValueType`. There are subclasses to represent scalar and vector types.

There are no predefined vector types, but they can be created as needed with the `LaneType.by()` function.

2.3 Instruction representation

The Rust in-memory representation of instructions is derived from the instruction descriptions. Part of the representation is generated, and part is written as Rust code in the `cranelift.instructions` module. The instruction representation depends on the input operand kinds and whether the instruction can produce multiple results.

Since all SSA value operands are represented as a *Value* in Rust code, value types don't affect the representation.

When an instruction description is created, it is automatically assigned a predefined instruction format which is an instance of `InstructionFormat`.

2.3.1 Restricted polymorphism

The instruction format strictly controls the kinds of operands on an instruction, but it does not constrain value types at all. A given instruction description typically does constrain the allowed value types for its value operands. The type variables give a lot of freedom in describing the value type constraints, in practice more freedom than what is needed for normal instruction set architectures. In order to simplify the Rust representation of value type constraints, some restrictions are imposed on the use of type variables.

A polymorphic instruction has a single *controlling type variable*. For a given opcode, this type variable must be the type of the first result or the type of the input value operand designated by the *typevar_operand* argument to the `InstructionFormat` constructor. By default, this is the first value operand, which works most of the time.

The value types of instruction results must be one of the following:

1. A concrete value type.
2. The controlling type variable.
3. A type variable derived from the controlling type variable.

This means that all result types can be computed from the controlling type variable.

Input values to the instruction are allowed a bit more freedom. Input value types must be one of:

1. A concrete value type.
2. The controlling type variable.
3. A type variable derived from the controlling type variable.
4. A free type variable that is not used by any other operands.

This means that the type of an input operand can either be computed from the controlling type variable, or it can vary independently of the other operands.

2.4 Encodings

Encodings describe how Cranelift instructions are mapped to binary machine code for the target architecture. After the legalization pass, all remaining instructions are expected to map 1-1 to native instruction encodings. Cranelift instructions that can't be encoded for the current architecture are called *illegal instruction*s.

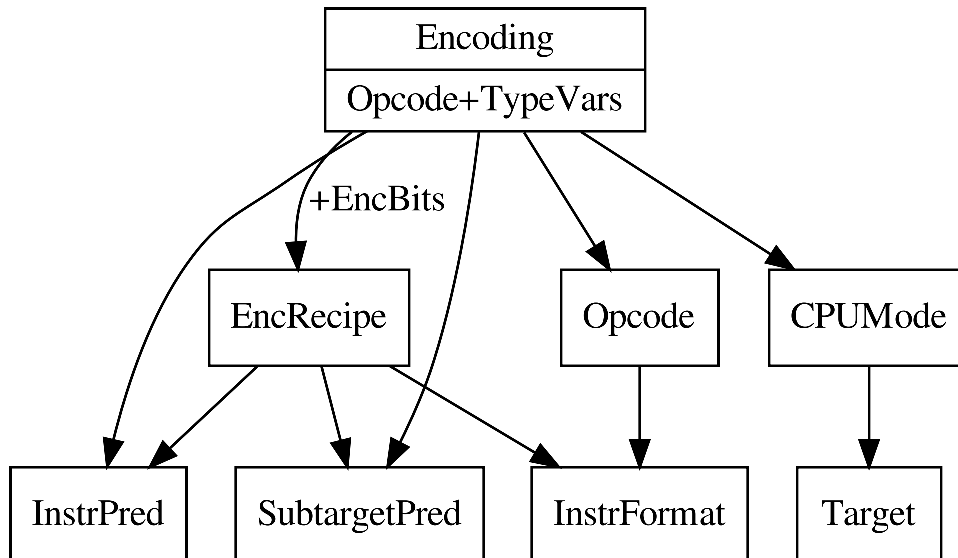
Some instruction set architectures have different *CPU mode*s with incompatible encodings. For example, a modern ARMv8 CPU might support three different CPU modes: *A64* where instructions are encoded in 32 bits, *A32* where all instructions are 32 bits, and *T32* which has a mix of 16-bit and 32-bit instruction encodings. These are incompatible encoding spaces, and while an *iadd* instruction can be encoded in 32 bits in each of them, it's not the same 32 bits. It's a judgement call if CPU modes should be modelled as separate targets, or as sub-modes of the same target. In the ARMv8 case, the different register banks means that it makes sense to model A64 as a separate target architecture, while A32 and T32 are CPU modes of the 32-bit ARM target.

In a given CPU mode, there may be multiple valid encodings of the same instruction. Both RISC-V and ARMv8's T32 mode have 32-bit encodings of all instructions with 16-bit encodings available for some opcodes if certain constraints are satisfied.

Encodings are guarded by *sub-target predicate*s. For example, the RISC-V "C" extension which specifies the compressed encodings may not be supported, and a predicate would be used to disable all of the 16-bit encodings in that case. This can also affect whether an instruction is legal. For example, x86 has a predicate that controls the SSE 4.1 instruction encodings. When that predicate is false, the SSE 4.1 instructions are not available.

Encodings also have a *instruction predicate* which depends on the specific values of the instruction's immediate fields. This is used to ensure that immediate address offsets are within range, for example. The instructions in the base Cranelift instruction set can often represent a wider range of immediates than any specific encoding. The fixed-size RISC-style encodings tend to have more range limitations than CISC-style variable length encodings like x86.

The diagram below shows the relationship between the classes involved in specifying instruction encodings:



An `Encoding` instance specifies the encoding of a concrete instruction. The following properties are used to select

instructions to be encoded:

- An opcode, i.e. *iadd_imm*, that must match the instruction's opcode.
- Values for any type variables if the opcode represents a polymorphic instruction.
- An *instruction predicate* that must be satisfied by the instruction's immediate operands.
- The CPU mode that must be active.
- A *sub-target predicate* that must be satisfied by the currently active sub-target.

An encoding specifies an *encoding recipe* along with some *encoding bits* that the recipe can use for native opcode fields etc. The encoding recipe has additional constraints that must be satisfied:

- An `InstructionFormat` that must match the format required by the opcodes of any encodings that use this recipe.
- An additional *instruction predicate*.
- An additional *sub-target predicate*.

The additional predicates in the `EncRecipe` are merged with the per-encoding predicates when generating the encoding matcher code. Often encodings only need the recipe predicates.

2.5 Register constraints

After an encoding recipe has been chosen for an instruction, it is the register allocator's job to make sure that the recipe's *Register constraint*s are satisfied. Most ISAs have separate integer and floating point registers, and instructions can usually only use registers from one of the banks. Some instruction encodings are even more constrained and can only use a subset of the registers in a bank. These constraints are expressed in terms of register classes.

Sometimes the result of an instruction is placed in a register that must be the same as one of the input registers. Some instructions even use a fixed register for inputs or results.

Each encoding recipe specifies separate constraints for its value operands and result. These constraints are separate from the instruction predicate which can only evaluate the instruction's immediate operands.

2.5.1 Register class constraints

The most common type of register constraint is the register class. It specifies that an operand or result must be allocated one of the registers from the given register class:

```
IntRegs = RegBank('IntRegs', ISA, 'General purpose registers', units=16, prefix='r')
GPR = RegClass(IntRegs)
R = EncRecipe('R', Binary, ins=(GPR, GPR), outs=GPR)
```

This defines an encoding recipe for the `Binary` instruction format where both input operands must be allocated from the `GPR` register class.

2.5.2 Tied register operands

In more compact machine code encodings, it is common to require that the result register is the same as one of the inputs. This is represented with tied operands:

```
CR = EncRecipe('CR', Binary, ins=(GPR, GPR), outs=0)
```

This indicates that the result value must be allocated to the same register as the first input value. Tied operand constraints can only be used for result values, so the number always refers to one of the input values.

2.5.3 Fixed register operands

Some instructions use hard-coded input and output registers for some value operands. An example is the `pblendvb` x86 SSE instruction which takes one of its three value operands in the hard-coded `%xmm0` register:

```
XMM0 = FPR[0]
SSE66_XMM0 = EncRecipe('SSE66_XMM0', Ternary, ins=(FPR, FPR, XMM0), outs=0)
```

The syntax `FPR[0]` selects the first register from the `FPR` register class which consists of all the XMM registers.

2.5.4 Stack operands

Cranelift's register allocator can assign an SSA value to a stack slot if there isn't enough registers. It will insert *spill* and *fill* instructions as needed to satisfy instruction operand constraints, but it is also possible to have instructions that can access stack slots directly:

```
CSS = EncRecipe('CSS', Unary, ins=GPR, outs=Stack(GPR))
```

An output stack value implies a store to the stack, an input value implies a load.

2.6 Targets

Cranelift can be compiled with support for multiple target instruction set architectures. Each ISA is represented by a `cdsl.isa.TargetISA` instance.

The definitions for each supported target live in a package under *cranelift-codegen/meta-python/isa*.

2.7 Glossary

Illegal instruction An instruction is considered illegal if there is no encoding available for the current CPU mode. The legality of an instruction depends on the value of *sub-target predicate*s, so it can't always be determined ahead of time.

CPU mode Every target defines one or more CPU modes that determine how the CPU decodes binary instructions. Some CPUs can switch modes dynamically with a branch instruction (like ARM/Thumb), while other modes are process-wide (like x86 32/64-bit).

Sub-target predicate A predicate that depends on the current sub-target configuration. Examples are "Use SSE 4.1 instructions", "Use RISC-V compressed encodings". Sub-target predicates can depend on both detected CPU features and configuration settings.

Instruction predicate A predicate that depends on the immediate fields of an instruction. An example is "the load address offset must be a 10-bit signed integer". Instruction predicates do not depend on the registers selected for value operands.

Register constraint Value operands and results correspond to machine registers. Encodings may constrain operands to either a fixed register or a register class. There may also be register constraints between operands, for example some encodings require that the result register is one of the input registers.

CHAPTER 3

Testing Cranelift

Cranelift is tested at multiple levels of abstraction and integration. When possible, Rust unit tests are used to verify single functions and types. When testing the interaction between compiler passes, file-level tests are appropriate.

The top-level shell script `test-all.sh` runs all of the tests in the Cranelift repository.

3.1 Rust tests

Rust and Cargo have good support for testing. Cranelift uses unit tests, doc tests, and integration tests where appropriate.

3.1.1 Unit tests

Unit test live in a `tests` sub-module of the code they are testing:

```
pub fn add(x: u32, y: u32) -> u32 {
    x + y
}

#[cfg(test)]
mod tests {
    use super::add;

    #[test]
    check_add() {
        assert_eq!(add(2, 2), 4);
    }
}
```

Since sub-modules have access to non-public items in a Rust module, unit tests can be used to test module-internal functions and types too.

3.1.2 Doc tests

Documentation comments can contain code snippets which are also compiled and tested:

```
//! The `Flags` struct is immutable once it has been created. A `Builder` instance is␣
↪used to
//! create it.
//!
//! # Example
//! ```
//! use cranelift_codegen::settings::{self, Configurable};
//!
//! let mut b = settings::builder();
//! b.set("opt_level", "fastest");
//!
//! let f = settings::Flags::new(&b);
//! assert_eq!(f.opt_level(), settings::OptLevel::Fastest);
//! ```
```

These tests are useful for demonstrating how to use an API, and running them regularly makes sure that they stay up to date. Documentation tests are not appropriate for lots of assertions; use unit tests for that.

3.1.3 Integration tests

Integration tests are Rust source files that are compiled and linked individually. They are used to exercise the external API of the crates under test.

These tests are usually found in the `tests` top-level directory where they have access to all the crates in the Cranelift repository. The `cranelift-codegen` and `cranelift-reader` crates have no external dependencies, which can make testing tedious. Integration tests that don't need to depend on other crates can be placed in `cranelift-codegen/tests` and `cranelift-reader/tests`.

3.2 File tests

Compilers work with large data structures representing programs, and it quickly gets unwieldy to generate test data programmatically. File-level tests make it easier to provide substantial input functions for the compiler tests.

File tests are `*.clif` files in the `filetests/` directory hierarchy. Each file has a header describing what to test followed by a number of input functions in the *Cranelift textual intermediate representation*:

```
test_file      ::=  test_header function_list
test_header    ::=  test_commands (isa_specs | settings)
test_commands  ::=  test_command { test_command }
test_command   ::=  "test" test_name { option } "\n"
```

The available test commands are described below.

Many test commands only make sense in the context of a target instruction set architecture. These tests require one or more ISA specifications in the test header:

```
isa_specs  ::=  { [settings] isa_spec }
isa_spec   ::=  "isa" isa_name { option } "\n"
```

The options given on the `isa` line modify the ISA-specific settings defined in `cranelift-codegen/meta-python/isa/*/settings.py`.

All types of tests allow shared Cranelift settings to be modified:

```
settings  ::=  { setting }
setting   ::=  "set" { option } "\n"
option    ::=  flag | setting "=" value
```

The shared settings available for all target ISAs are defined in `cranelift-codegen/meta-python/base/settings.py`.

The `set` lines apply settings cumulatively:

```
test legalizer
set opt_level=best
set is_pic=1
isa riscv64
set is_pic=0
isa supports_m=false

function %foo() {}
```

This example will run the legalizer test twice. Both runs will have `opt_level=best`, but they will have different `is_pic` settings. The 32-bit run will also have the RISC-V specific flag `supports_m` disabled.

The filetests are run automatically as part of *cargo test*, and they can also be run manually with the *clif-util test* command.

3.2.1 Filecheck

Many of the test commands described below use *filecheck* to verify their output. Filecheck is a Rust implementation of the LLVM tool of the same name. See the documentation for details of its syntax.

Comments in `.clif` files are associated with the entity they follow. This typically means an instruction or the whole function. Those tests that use filecheck will extract comments associated with each function (or its entities) and scan them for filecheck directives. The test output for each function is then matched against the filecheck directives for that function.

Comments appearing before the first function in a file apply to every function. This is useful for defining common regular expression variables with the `regex:` directive, for example.

Note that LLVM's file tests don't separate filecheck directives by their associated function. It verifies the concatenated output against all filecheck directives in the test file. LLVM's **`FileCheck`** command has a `CHECK-LABEL:` directive to help separate the output from different functions. Cranelift's tests don't need this.

3.2.2 *test cat*

This is one of the simplest file tests, used for testing the conversion to and from textual IR. The `test cat` command simply parses each function and converts it back to text again. The text of each function is then matched against the associated filecheck directives.

Example:

```
function %r1() -> i32, f32 {
ebb1:
    v10 = iconst.i32 3
    v20 = f32const 0.0
    return v10, v20
}
; sameln: function %r1() -> i32, f32 {
; nextln: ebb0:
; nextln:     v10 = iconst.i32 3
; nextln:     v20 = f32const 0.0
; nextln:     return v10, v20
; nextln: }
```

3.2.3 *test verifier*

Run each function through the IR verifier and check that it produces the expected error messages.

Expected error messages are indicated with an `error:` directive *on the instruction that produces the verifier error*. Both the error message and reported location of the error is verified:

```
test verifier

function %test(i32) {
    ebb0(v0: i32):
        jump ebb1       ; error: terminator
        return
}
```

This example test passes if the verifier fails with an error message containing the sub-string `"terminator"` *and* the error is reported for the `jump` instruction.

If a function contains no `error:` annotations, the test passes if the function verifies correctly.

3.2.4 *test print-cfg*

Print the control flow graph of each function as a Graphviz graph, and run filecheck over the result. See also the **clif-util print-cfg** command:

```
; For testing cfg generation. This code is nonsense.
test print-cfg
test verifier

function %nonsense(i32, i32) -> f32 {
; check: digraph %nonsense {
; regex: I=\binst\d+\b
; check: label="{ebb0 | <$(BRZ=$I)>brz ebb2 | <$(JUMP=$I)>jump ebb1}"]

ebb0(v0: i32, v1: i32):
    brz v1, ebb2            ; unordered: ebb0:$BRZ -> ebb2
    v2 = iconst.i32 0
    jump ebb1(v2)           ; unordered: ebb0:$JUMP -> ebb1

ebb1(v5: i32):
    return v0
```

(continues on next page)

(continued from previous page)

```
ebb2:
    v100 = f32const 0.0
    return v100
}
```

3.2.5 *test domtree*

Compute the dominator tree of each function and validate it against the `dominates:` annotations:

```
test domtree

function %test(i32) {
    ebb0(v0: i32):
        jump ebb1     ; dominates: ebb1
    ebb1:
        brz v0, ebb3  ; dominates: ebb3
        jump ebb2     ; dominates: ebb2
    ebb2:
        jump ebb3
    ebb3:
        return
}
```

Every reachable extended basic block except for the entry block has an *immediate dominator* which is a jump or branch instruction. This test passes if the `dominates:` annotations on the immediate dominator instructions are both correct and complete.

This test also sends the computed CFG post-order through filecheck.

3.2.6 *test legalizer*

Legalize each function for the specified target ISA and run the resulting function through filecheck. This test command can be used to validate the encodings selected for legal instructions as well as the instruction transformations performed by the legalizer.

3.2.7 *test regalloc*

Test the register allocator.

First, each function is legalized for the specified target ISA. This is required for register allocation since the instruction encodings provide register class constraints to the register allocator.

Second, the register allocator is run on the function, inserting spill code and assigning registers and stack slots to all values.

The resulting function is then run through filecheck.

3.2.8 *test binemit*

Test the emission of binary machine code.

The functions must contains instructions that are annotated with both encodings and value locations (registers or stack slots). For instructions that are annotated with a *bin:* directive, the emitted hexadecimal machine code for that instruction is compared to the directive:

```
test binemit
isa riscv

function %int32() {
ebb0:
    [-,%x5]             v0 = iconst.i32 1
    [-,%x6]             v1 = iconst.i32 2
    [R#0c,%x7]          v10 = iadd v0, v1        ; bin: 006283b3
    [R#200c,%x8]        v11 = isub v0, v1        ; bin: 40628433
    return
}
```

If any instructions are unencoded (indicated with a *[-]* encoding field), they will be encoded using the same mechanism as the legalizer uses. However, illegal instructions for the ISA won't be expanded into other instruction sequences. Instead the test will fail.

Value locations must be present if they are required to compute the binary bits. Missing value locations will cause the test to crash.

3.2.9 *test simple-gvn*

Test the simple GVN pass.

The simple GVN pass is run on each function, and then results are run through filecheck.

3.2.10 *test licm*

Test the LICM pass.

The LICM pass is run on each function, and then results are run through filecheck.

3.2.11 *test dce*

Test the DCE pass.

The DCE pass is run on each function, and then results are run through filecheck.

3.2.12 *test shrink*

Test the instruction shrinking pass.

The shrink pass is run on each function, and then results are run through filecheck.

3.2.13 *test preopt*

Test the preopt pass.

The preopt pass is run on each function, and then results are run through filecheck.

3.2.14 *test postopt*

Test the postopt pass.

The postopt pass is run on each function, and then results are run through filecheck.

3.2.15 *test compile*

Test the whole code generation pipeline.

Each function is passed through the full `Context::compile()` function which is normally used to compile code. This type of test often depends on assertions or verifier errors, but it is also possible to use filecheck directives which will be matched against the final form of the Cranelift IR right before binary machine code emission.

3.2.16 *test run*

Compile and execute a function.

Add a `; run` directive after each function that should be executed. These functions must have the signature `() -> bNN` where `bNN` is some sort of boolean, e.g. `b1` or `b32`. A `true` value is interpreted as a successful test execution, whereas a `false` value is interpreted as a failed test.

Example:

```
test run

function %trivial_test() -> b1 {
ebb0:
    v0 = bconst.b1 true
    return v0
}
; run
```

CHAPTER 4

Register Allocation in Cranelift

Cranelift uses a *decoupled, SSA-based* register allocator. Decoupled means that register allocation is split into two primary phases: *spilling* and *coloring*. SSA-based means that the code stays in SSA form throughout the register allocator, and in fact is still in SSA form after register allocation.

Before the register allocator is run, all instructions in the function must be *legalized*, which means that every instruction has an entry in the `encodings` table. The encoding entries also provide register class constraints on the instruction's operands that the register allocator must satisfy.

After the register allocator has run, the `locations` table provides a register or stack slot location for all SSA values used by the function. The register allocator may have inserted **:inst:'spill'**, **:inst:'fill'**, and **:inst:'copy'** instructions to make that possible.

4.1 SSA-based register allocation

The phases of the SSA-based register allocator are:

Liveness analysis For each SSA value, determine exactly where it is live.

Coalescing Form *virtual registers* which are sets of SSA values that should be assigned to the same location. Split live ranges such that values that belong to the same virtual register don't have interfering live ranges.

Spilling The process of deciding which SSA values go in a stack slot and which values go in a register. The spilling phase can also split live ranges by inserting **:inst:'copy'** instructions, or transform the code in other ways to reduce the number of values kept in registers.

After spilling, the number of live register values never exceeds the number of available registers.

Reload Insert **:inst:'spill'** and **:inst:'fill'** instructions as necessary such that instructions that expect their operands in registers won't see values that live on the stack and vice versa.

Reuse registers containing values loaded from the stack as much as possible without exceeding the maximum allowed register pressure.

Coloring The process of assigning specific registers to the live values. It's a property of SSA form that this can be done in a linear scan of the dominator tree without causing any additional spills.

Make sure that specific register operand constraints are satisfied.

The contract between the spilling and coloring phases is that the number of values in registers never exceeds the number of available registers. This sounds simple enough in theory, but in practice there are some complications.

4.1.1 Real-world complications to SSA coloring

In practice, instruction set architectures don't have "K interchangeable registers", and register pressure can't be measured with a single number. There are complications:

Different register banks Most ISAs separate integer registers from floating point registers, and instructions require their operands to come from a specific bank. This is a fairly simple problem to deal with since the register banks are completely disjoint. We simply count the number of integer and floating-point values that are live independently, and make sure that each number does not exceed the size of their respective register banks.

Instructions with fixed operands Some instructions use a fixed register for an operand. This happens on the x86 ISAs:

- Dynamic shift and rotate instructions take the shift amount in CL.
- Division instructions use RAX and RDX for both input and output operands.
- Wide multiply instructions use fixed RAX and RDX registers for input and output operands.
- A few SSE variable blend instructions use a hardwired XMM0 input operand.

Operands constrained to register subclasses Some instructions can only use a subset of the registers for some operands. For example, the ARM NEON vmla (scalar) instruction requires the scalar operand to be located in D0-15 or even D0-7, depending on the data type. The other operands can be from the full D0-31 register set.

ABI boundaries Before making a function call, arguments must be placed in specific registers and stack locations determined by the ABI, and return values appear in fixed registers.

Some registers can be clobbered by the call and some are saved by the callee. In some cases, only the low bits of a register are saved by the callee. For example, ARM64 callees save only the low 64 bits of v8-15, and Win64 callees only save the low 128 bits of AVX registers.

ABI boundaries also affect the location of arguments to the entry block and return values passed to the **:inst:'return'** instruction.

Aliasing registers Different registers sometimes share the same bits in the register bank. This can make it difficult to measure register pressure. For example, the x86 registers RAX, EAX, AX, AL, and AH overlap.

If only one of the aliasing registers can be used at a time, the aliasing doesn't cause problems since the registers can simply be counted as one unit.

Early clobbers Sometimes an instruction requires that the register used for an output operand does not alias any of the input operands. This happens for inline assembly and in some other special cases.

4.2 Liveness Analysis

All the register allocator passes need to know exactly where SSA values are live. The liveness analysis computes this information.

The data structure representing the live range of a value uses the linear layout of the function. All instructions and EBB headers are assigned a *program position*. A starting point for a live range can be one of the following:

- The instruction where the value is defined.
- The EBB header where the value is an EBB parameter.

- An EBB header where the value is live-in because it was defined in a dominating block.

The ending point of a live range can be:

- The last instruction to use the value.
- A branch or jump to an EBB where the value is live-in.

When all the EBBs in a function are laid out linearly, the live range of a value doesn't have to be a contiguous interval, although it will be in a majority of cases. There can be holes in the linear live range.

The part of a value's live range that falls inside a single EBB will always be an interval without any holes. This follows from the dominance requirements of SSA. A live range is represented as:

- The interval inside the EBB where the value is defined.
- A set of intervals for EBBs where the value is live-in.

Any value that is only used inside a single EBB will have an empty set of live-in intervals. Some values are live across large parts of the function, and this can often be represented with coalesced live-in intervals covering many EBBs. It is important that the live range data structure doesn't have to grow linearly with the number of EBBs covered by a live range.

This representation is very similar to LLVM's `LiveInterval` data structure with a few important differences:

- The Cranelift `LiveRange` only covers a single SSA value, while LLVM's `LiveInterval` represents the union of multiple related SSA values in a virtual register. This makes Cranelift's representation smaller because individual segments don't have to annotated with a value number.
- Cranelift stores the def-interval separately from a list of coalesced live-in intervals, while LLVM stores an array of segments. The two representations are equivalent, but Cranelift optimizes for the common case of a value that is only used locally.
- It is simpler to check if two live ranges are overlapping. The dominance properties of SSA form means that it is only necessary to check the def-interval of each live range against the intervals of the other range. It is not necessary to check for overlap between the two sets of live-in intervals. This makes the overlap check logarithmic in the number of live-in intervals instead of linear.
- LLVM represents a program point as `SlotIndex` which holds a pointer to a 32-byte `IndexListEntry` struct. The entries are organized in a double linked list that mirrors the ordering of instructions in a basic block. This allows 'tombstone' program points corresponding to instructions that have been deleted.

  Cranelift uses a 32-bit program point representation that encodes an instruction or EBB number directly. There are no 'tombstones' for deleted instructions, and no mirrored linked list of instructions. Live ranges must be updated when instructions are deleted.

A consequence of Cranelift's more compact representation is that two program points can't be compared without the context of a function layout.

4.3 Coalescing algorithm

Unconstrained SSA form is not well suited to register allocation because of the problems that can arise around EBB parameters and arguments. Consider this simple example:

```
function %interference(i32, i32) -> i32 {
ebb0(v0: i32, v1: i32):
    brz v0, ebb1(v1)
    jump ebb1(v0)

ebb1(v2: i32):
```

(continues on next page)

(continued from previous page)

```
    v3 = iadd v1, v2
    return v3
}
```

Here, the value `v1` is both passed as an argument to `ebb1` *and* it is live in to the EBB because it is used by the **:inst:‘iadd‘** instruction. Since EBB arguments on the **:inst:‘brz‘** instruction need to be in the same register as the corresponding EBB parameter `v2`, there is going to be interference between `v1` and `v2` in the `ebb1` block.

The interference can be resolved by isolating the SSA values passed as EBB arguments:

```
function %coalesced(i32, i32) -> i32 {
ebb0(v0: i32, v1: i32):
    v5 = copy v1
    brz v0, ebb1(v5)
    v6 = copy v0
    jump ebb1(v6)

ebb1(v2: i32):
    v3 = iadd.i32 v1, v2
    return v3
}
```

Now the EBB argument is `v5` which is *not* itself live into `ebb1`, resolving the interference.

The coalescing pass groups the SSA values into sets called *virtual registers* and inserts copies such that:

1. Whenever a value is passed as an EBB argument, the corresponding EBB parameter value belongs to the same virtual register as the passed argument value.
2. The live ranges of values belonging to the same virtual register do not interfere, i.e. they don’t overlap anywhere.

Most virtual registers contains only a single isolated SSA value because most SSA values are never passed as EBB arguments. The `VirtRegs` data structure doesn’t store any information about these singleton virtual registers, it only tracks larger virtual registers and assumes that any value it doesn’t know about is its own singleton virtual register

Once the values have been partitioned into interference-free virtual registers, the code is said to be in conventional SSA form (CSSA). A program in CSSA form can be register allocated correctly by assigning all the values in a virtual register to the same stack or register location.

Conventional SSA form and the virtual registers are maintained through all the register allocator passes.

4.4 Spilling algorithm

The spilling pass is responsible for lowering the register pressure enough that the coloring pass is guaranteed to be able to find a coloring solution. It does this by assigning whole virtual registers to stack slots.

Besides just counting registers, the spiller also has to look at the instruction’s operand constraints because sometimes the constraints can require extra registers to solve, raising the register pressure:

- If a single value is used more than once by an instruction, and the operands have conflicting constraints, two registers must be used. The most common case is when a single value is passed as two separate arguments to a function call.
- If an instruction has a *tied operand constraint* where one of the input operands must use the same register as the output operand, the spiller makes sure that the tied input value doesn’t interfere with the output value by inserting a copy if needed.

The spilling heuristic used by Cranelift is very simple. Whenever the spiller determines that the register pressure is too high at some instruction, it picks the live SSA value whose definition is farthest away as the spill candidate. Then it spills all values in the corresponding virtual register to the same spill slot. It is important that all values in a virtual register get the same spill slot, otherwise we could need memory-to-memory copies when passing spilled arguments to a spilled EBB parameter.

This simple heuristic tends to spill values with long live ranges, and it depends on the reload pass to do a good job of reusing registers reloaded from spill slots if the spilled value gets used a lot. The idea is to minimize stack *write* traffic with the spilling heuristic and to minimize stack *read* traffic with the reload pass.

4.5 Coloring algorithm

The SSA coloring algorithm is based on a single observation: If two SSA values interfere, one of the values must be live where the other value is defined.

We visit the EBBs in a topological order such that all dominating EBBs are visited before the current EBB. The instructions in an EBB are visited in a top-down order, and each value define by the instruction is assigned an available register. With this iteration order, every value that is live at an instruction has already been assigned to a register.

This coloring algorithm works if the following condition holds:

> At every instruction, consider the values live through the instruction. No matter how the live values have been assigned to registers, there must be available registers of the right register classes available for the values defined by the instruction.

We'll need to modify this condition in order to deal with the real-world complications.

The coloring algorithm needs to keep track of the set of live values at each instruction. At the top of an EBB, this set can be computed as the union of:

- The set of live values before the immediately dominating branch or jump instruction. The topological iteration order guarantees that this set is available. Values whose live range indicate that they are not live-in to the current EBB should be filtered out.
- The set of parameters the EBB. These values should all be live-in, although it is possible that some are dead and never used anywhere.

For each live value, we also track its kill point in the current EBB. This is the last instruction to use the value in the EBB. Values that are live-out through the EBB terminator don't have a kill point. Note that the kill point can be a branch to another EBB that uses the value, so the kill instruction doesn't have to be a use of the value.

When advancing past an instruction, the live set is updated:

- Any values whose kill point is the current instruction are removed.
- Any values defined by the instruction are added, unless their kill point is the current instruction. This corresponds to a dead def which has no uses.

CHAPTER 5

Cranelift compared to LLVM

LLVM is a collection of compiler components implemented as a set of C++ libraries. It can be used to build both JIT compilers and static compilers like Clang, and it is deservedly very popular. Chris Lattner's chapter about LLVM in the Architecture of Open Source Applications book gives an excellent overview of the architecture and design of LLVM.

Cranelift and LLVM are superficially similar projects, so it is worth highlighting some of the differences and similarities. Both projects:

- Use an ISA-agnostic input language in order to mostly abstract away the differences between target instruction set architectures.
- Depend extensively on SSA form.
- Have both textual and in-memory forms of their primary intermediate representation. (LLVM also has a binary bitcode format; Cranelift doesn't.)
- Can target multiple ISAs.
- Can cross-compile by default without rebuilding the code generator.

However, there are also some major differences, described in the following sections.

5.1 Intermediate representations

LLVM uses multiple intermediate representations as it translates a program to binary machine code:

LLVM IR This is the primary intermediate representation which has textual, binary, and in-memory forms. It serves two main purposes:

- An ISA-agnostic, stable(ish) input language that front ends can generate easily.
- Intermediate representation for common mid-level optimizations. A large library of code analysis and transformation passes operate on LLVM IR.

SelectionDAG A graph-based representation of the code in a single basic block is used by the instruction selector. It has both ISA-agnostic and ISA-specific opcodes. These main passes are run on the SelectionDAG representation:

- Type legalization eliminates all value types that don't have a representation in the target ISA registers.
- Operation legalization eliminates all opcodes that can't be mapped to target ISA instructions.
- DAG-combine cleans up redundant code after the legalization passes.
- Instruction selection translates ISA-agnostic expressions to ISA-specific instructions.

The SelectionDAG representation automatically eliminates common subexpressions and dead code.

MachineInstr A linear representation of ISA-specific instructions that initially is in SSA form, but it can also represent non-SSA form during and after register allocation. Many low-level optimizations run on MI code. The most important passes are:

- Scheduling.
- Register allocation.

MC MC serves as the output abstraction layer and is the basis for LLVM's integrated assembler. It is used for:

- Branch relaxation.
- Emitting assembly or binary object code.
- Assemblers.
- Disassemblers.

There is an ongoing "global instruction selection" project to replace the SelectionDAG representation with ISA-agnostic opcodes on the MachineInstr representation. Some target ISAs have a fast instruction selector that can translate simple code directly to MachineInstrs, bypassing SelectionDAG when possible.

Cranelift uses a single intermediate representation to cover these levels of abstraction. This is possible in part because of Cranelift's smaller scope.

- Cranelift does not provide assemblers and disassemblers, so it is not necessary to be able to represent every weird instruction in an ISA. Only those instructions that the code generator emits have a representation.
- Cranelift's opcodes are ISA-agnostic, but after legalization / instruction selection, each instruction is annotated with an ISA-specific encoding which represents a native instruction.
- SSA form is preserved throughout. After register allocation, each SSA value is annotated with an assigned ISA register or stack slot.

The Cranelift intermediate representation is similar to LLVM IR, but at a slightly lower level of abstraction, to allow it to be used all the way through the codegen process.

This design tradeoff does mean that Cranelift IR is less friendly for mid-level optimizations. Cranelift doesn't currently perform mid-level optimizations, however if it should grow to where this becomes important, the vision is that Cranelift would add a separate IR layer, or possibly an separate IR, to support this. Instead of frontends producing optimizer IR which is then translated to codegen IR, Cranelift would have frontends producing codegen IR, which can be translated to optimizer IR and back.

This biases the overall system towards fast compilation when mid-level optimization is not needed, such as when emitting unoptimized code for or when low-level optimizations are sufficient.

And, it removes some constraints in the mid-level optimize IR design space, making it more feasible to consider ideas such as using a VSDG-based IR.

5.1.1 Program structure

In LLVM IR, the largest representable unit is the *module* which corresponds more or less to a C translation unit. It is a collection of functions and global variables that may contain references to external symbols too.

In Cranelift's IR, used by the cranelift-codegen crate, functions are self-contained, allowing them to be compiled independently. At this level, there is no explicit module that contains the functions.

Module functionality in Cranelift is provided as an optional library layer, in the cranelift-module crate. It provides facilities for working with modules, which can contain multiple functions as well as data objects, and it links them together.

An LLVM IR function is a graph of *basic blocks*. A Cranelift IR function is a graph of *extended basic blocks* that may contain internal branch instructions. The main difference is that an LLVM conditional branch instruction has two target basic blocks—a true and a false edge. A Cranelift branch instruction only has a single target and falls through to the next instruction when its condition is false. The Cranelift representation is closer to how machine code works; LLVM's representation is more abstract.

LLVM uses phi instructions in its SSA representation. Cranelift passes arguments to EBBs instead. The two representations are equivalent, but the EBB arguments are better suited to handle EBBs that may contain multiple branches to the same destination block with different arguments. Passing arguments to an EBB looks a lot like passing arguments to a function call, and the register allocator treats them very similarly. Arguments are assigned to registers or stack locations.

5.1.2 Value types

Cranelift's type system is mostly a subset of LLVM's type system. It is less abstract and closer to the types that common ISA registers can hold.

- Integer types are limited to powers of two from **:clif:type:‘i8‘** to **:clif:type:‘i64‘**. LLVM can represent integer types of arbitrary bit width.
- Floating point types are limited to **:clif:type:‘f32‘** and **:clif:type:‘f64‘** which is what WebAssembly provides. It is possible that 16-bit and 128-bit types will be added in the future.
- Addresses are represented as integers—There are no Cranelift pointer types. LLVM currently has rich pointer types that include the pointee type. It may move to a simpler 'address' type in the future. Cranelift may add a single address type too.
- SIMD vector types are limited to a power-of-two number of vector lanes up to 256. LLVM allows an arbitrary number of SIMD lanes.
- Cranelift has no aggregate types. LLVM has named and anonymous struct types as well as array types.

Cranelift has multiple boolean types, whereas LLVM simply uses *i1*. The sized Cranelift boolean types are used to represent SIMD vector masks like `b32x4` where each lane is either all 0 or all 1 bits.

Cranelift instructions and function calls can return multiple result values. LLVM instead models this by returning a single value of an aggregate type.

5.1.3 Instruction set

LLVM has a small well-defined basic instruction set and a large number of intrinsics, some of which are ISA-specific. Cranelift has a larger instruction set and no intrinsics. Some Cranelift instructions are ISA-specific.

Since Cranelift instructions are used all the way until the binary machine code is emitted, there are opcodes for every native instruction that can be generated. There is a lot of overlap between different ISAs, so for example the **:clif:inst:‘iadd_imm‘** instruction is used by every ISA that can add an immediate integer to a register. A simple RISC

ISA like RISC-V can be defined with only shared instructions, while x86 needs a number of specific instructions to model addressing modes.

5.2 Undefined behavior

Cranelift does not generally exploit undefined behavior in its optimizations. LLVM's mid-level optimizations do, but it should be noted that LLVM's low-level code generator rarely needs to make use of undefined behavior either.

LLVM provides `nsw` and `nuw` flags for its arithmetic that invoke undefined behavior on overflow. Cranelift does not provide this functionality. Its arithmetic instructions either produce a value or a trap.

LLVM has an `unreachable` instruction which is used to indicate impossible code paths. Cranelift only has an explicit **:clif:inst:'trap'** instruction.

Cranelift does make assumptions about aliasing. For example, it assumes that it has full control of the stack objects in a function, and that they can only be modified by function calls if their address have escaped. It is quite likely that Cranelift will admit more detailed aliasing annotations on load/store instructions in the future. When these annotations are incorrect, undefined behavior ensues.

CHAPTER 6

Rust Crate Documentation

cranelift This is an umbrella crate that re-exports the codegen and frontend crates, to make them easier to use.

cranelift-codegen This is the core code generator crate. It takes Cranelift IR as input and emits encoded machine instructions, along with symbolic relocations, as output.

cranelift-codegen-meta This crate contains the meta-language utilities and descriptions used by the code generator.

cranelift-wasm This crate translates WebAssembly code into Cranelift IR.

cranelift-frontend This crate provides utilities for translating code into Cranelift IR.

cranelift-native This crate performs auto-detection of the host, allowing Cranelift to generate code optimized for the machine it's running on.

cranelift-reader This crate translates from Cranelift IR's text format into Cranelift IR in in-memory data structures.

cranelift-module This crate manages compiling multiple functions and data objects together.

cranelift-object This crate provides a object-based backend for *cranelift-module*, which emits native object files using the object library.

cranelift-faerie This crate provides a faerie-based backend for *cranelift-module*, which emits native object files using the faerie library.

cranelift-simplejit This crate provides a simple JIT backend for *cranelift-module*, which emits code and data into memory.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

CHAPTER 8

Todo list

Todo: Update the IR reference

This document is likely to be outdated and missing some important information. It is recommended to look at the list of instructions as documented in the *InstBuilder* documentation: https://docs.rs/cranelift-codegen/latest/cranelift_codegen/ir/trait.InstBuilder.html

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/cranelift/checkouts/stable/docs/ir.rst, line 8.)

Todo: Point to the Rust documentation of the meta crate here.

This document is very out-of-date. Instead, you can have a look at the work-in-progress documentation of the *meta* crate there: https://docs.rs/cranelift-codegen-meta/0.34.0/cranelift_codegen_meta/.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/cranelift/checkouts/stable/docs/meta.rst, line 11.)

Index

A

accessible, **17**
addressable, **17**

B

basic block, **17**

C

CPU mode, **24**

E

EBB, **17**
EBB argument, **17**
EBB parameter, **17**
entry block, **17**
explicit stack slot, **18**
extended basic block, **17**

F

function body, **18**
function preamble, **17**
function signature, **17**

I

Illegal instruction, **24**
Instruction predicate, **24**
intermediate representation, **18**
IR, **18**

R

Register constraint, **24**

S

spill stack slot, **18**
stack slot, **18**
Sub-target predicate, **24**

T

terminator instruction, **18**
trap, **18**
trapping, **18**
traps, **18**