Icewind

Fast Full-System Emulators from Formal Specifications of Instruction Set Architectures

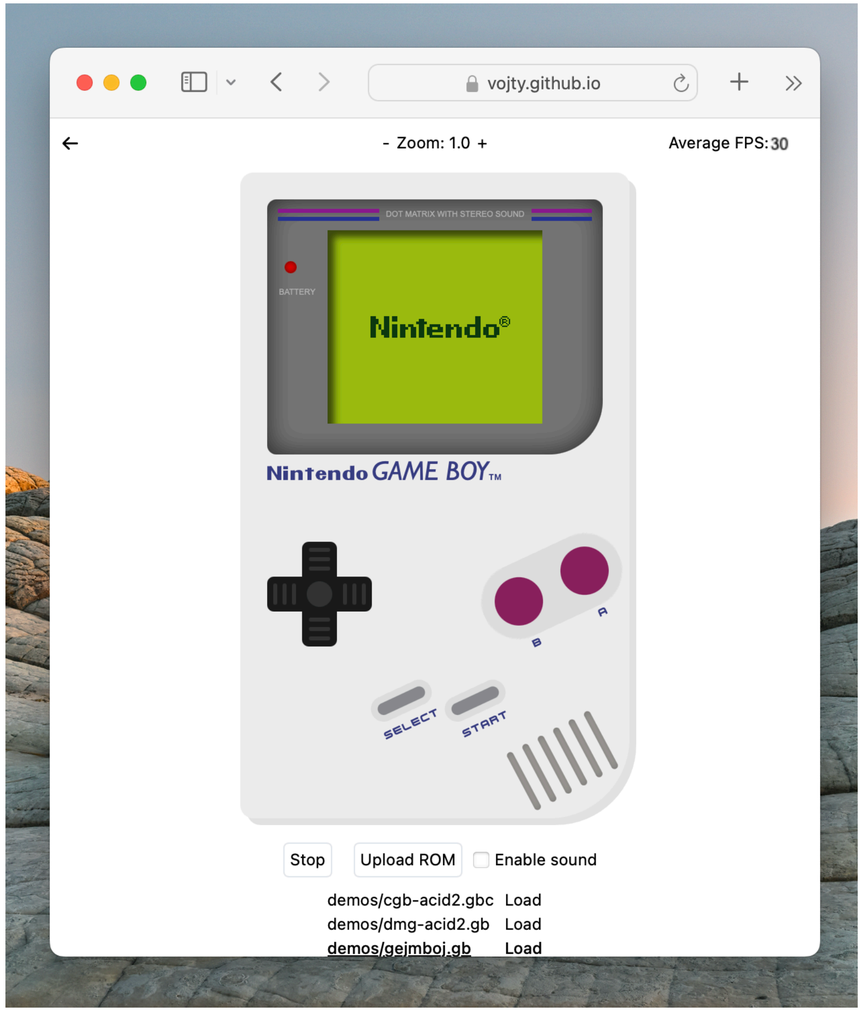
Ferdia McKeogh, University of St Andrews

Supervised by Dr Tom Spink and Prof. Al Dearle

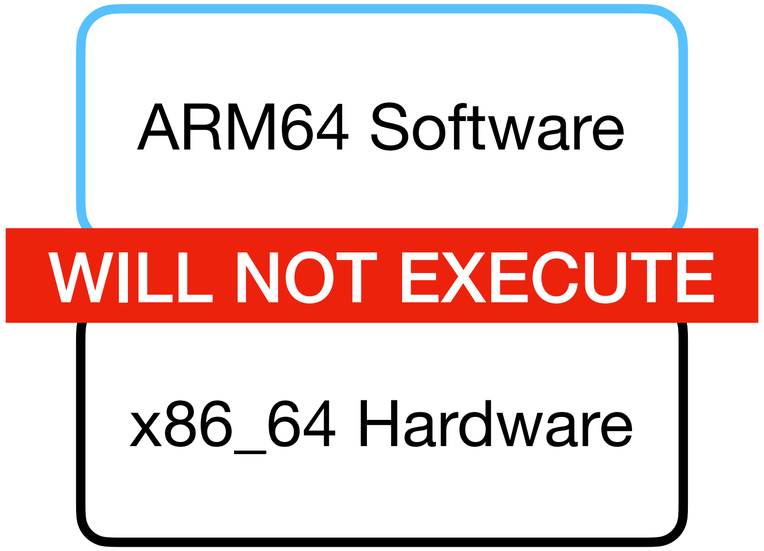
Funded by Adobe, Inc.

Emulators

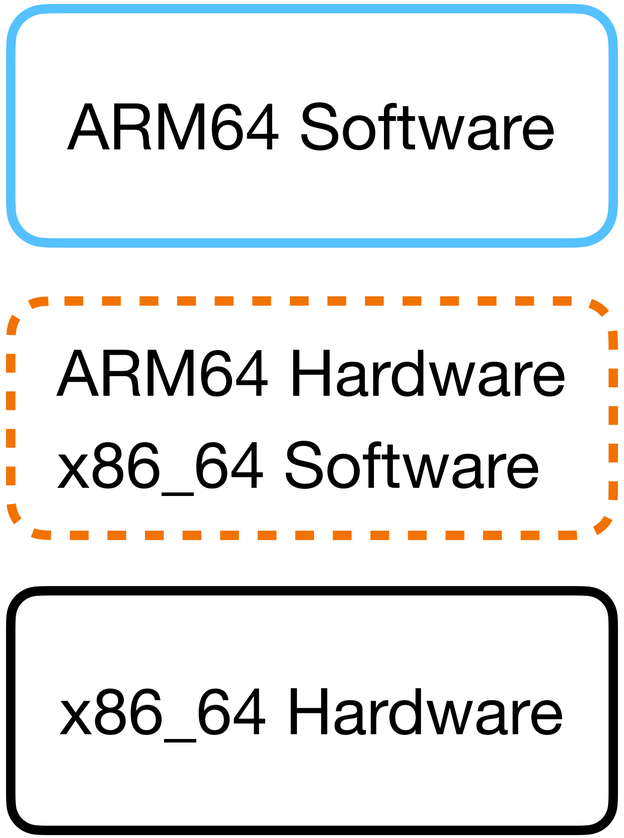
- “Run this ARM64 binary on an x86_64 machine”
- “Play this GameBoy ROM in a browser”
- Useful:
  - Testing
  - Debugging
  - Legacy + Future



Emulators



Emulators


ARM64 Software
ARM64 Hardware
x86_64 Software
x86_64 Hardware

Instruction Set Architecture

- Definition of interface between hardware and software
  - Instructions
    - How do you decode them?
    - How do you execute them?
  - Registers
  - Memory
  - Other misc. state and behaviours (exceptions, etc)

Existing Emulators

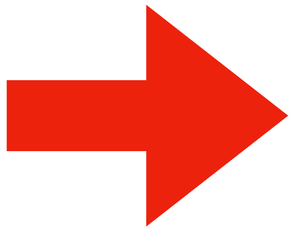
QEMU

- Very popular emulation tool
- Full-system or userspace
- Fast!

Captive

- New research tool
- Full-system emulation
- Faster than QEMU!
  - Hardware-accelerated guest memory translation

Manual Architecture Implementations

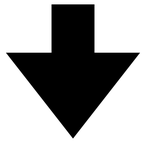


Sail

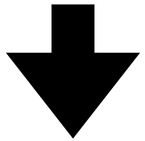
- Language for formally specifying an ISA
- Imperative, functional elements, advanced type system
- Formal verification research
- ARM->Sail
- RISC-V
- Many existing models
- Correct

Sail ISAs

```
decode(opcode: u32) { … }
```



```
decode_add(opcode: u32) { … }
```



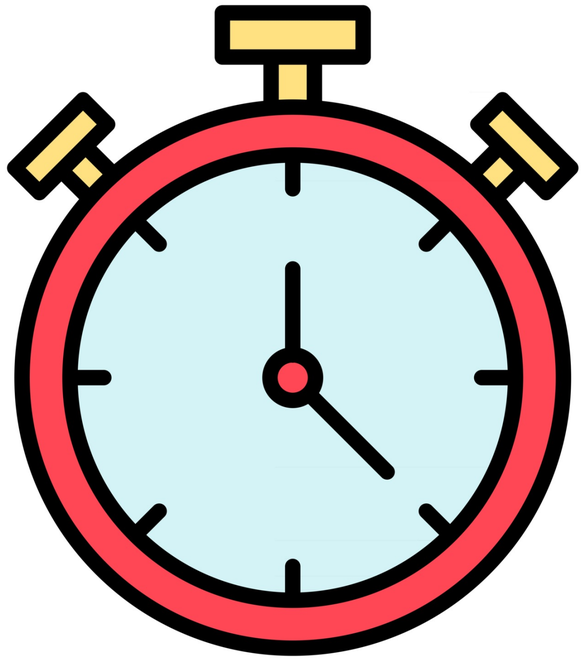
```
execute_add(src: u4, dst: u4, imm: u8) { … }
```

Sail Emulators

- Sail compiler C backend
  - Slow interpreter
  - 2000x slower than QEMU
- Pydrofoil
  - Faster, uses Python JIT
  - 300x slower than QEMU

Why the speed difference?

- **Interpreters (slow)**
  - Fetch, decode, execute loop
  - 1 emulated instruction -> 1000-10000s real instructions
  - Repeated for each instruction
- **Dynamic binary translators (fast)**
  - Fetch, decode block of instructions
  - Translate block using a Just-In-Time compiler to native machine code
  - Execute native code every time we see emulated block



Why the speed difference?

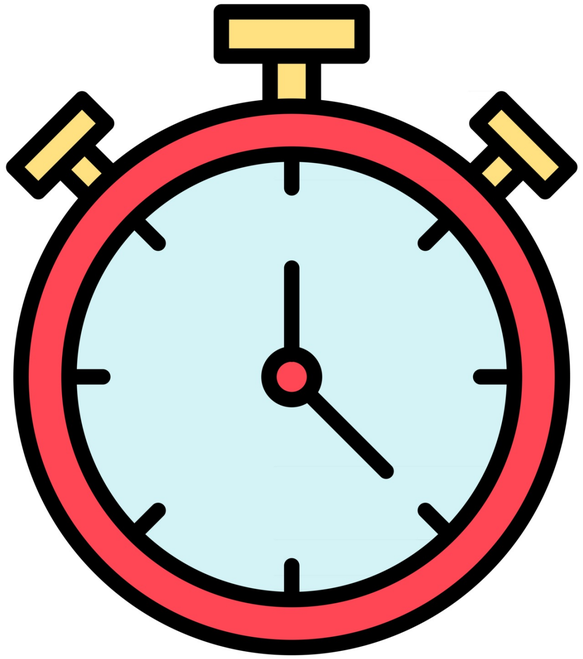
- **Interpreters (slow)**
  - Has to be generic over every variant
- **Dynamic binary translators (fast)**
  - Know constant information

```
fn add(a: u64, b: u64) -> u64 {
  a + b
}
```

```
fn add(a: u64, b: u64 = 5) -> u64 {
  a + 5
}
```

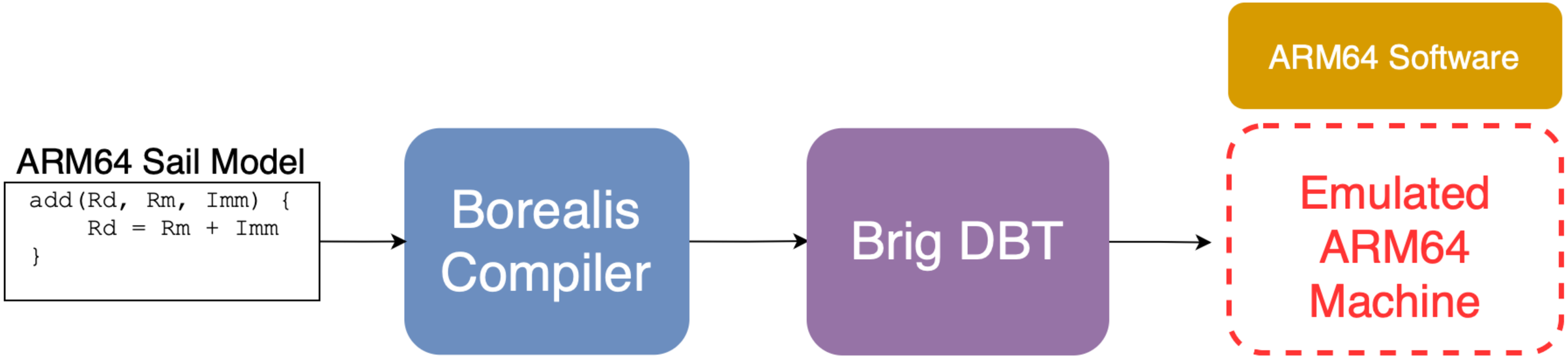
How can we extract functional elements from a formal model for fast emulation?

Fast execution



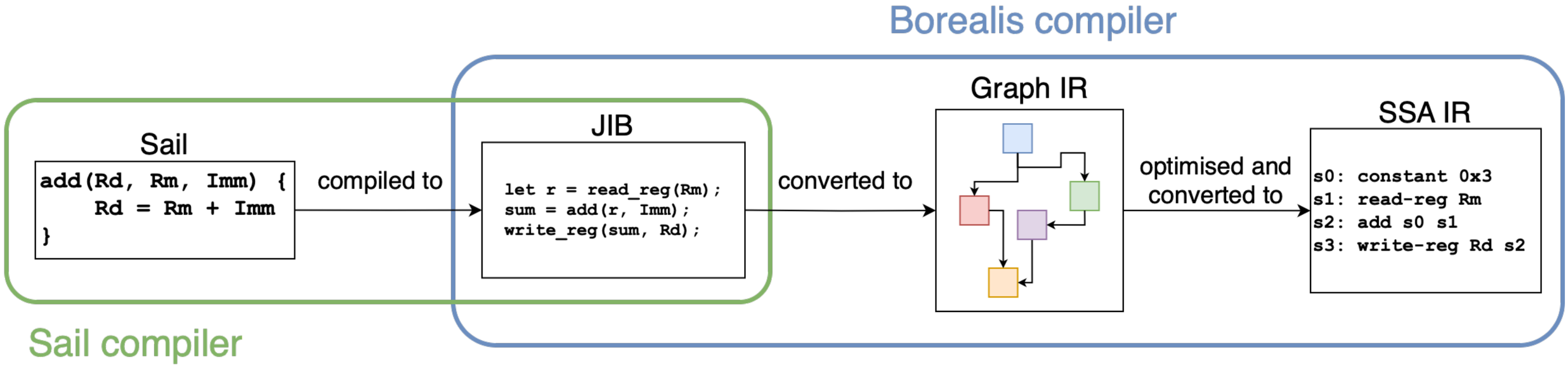
Automatic implementation

Icewind:
A DBT for Sail



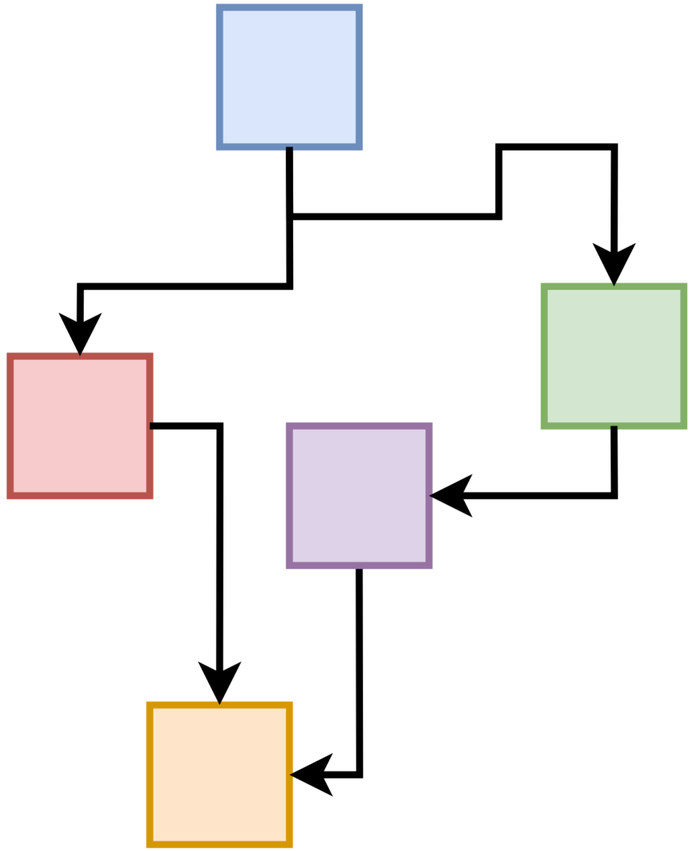
Borealis

Compilation Steps



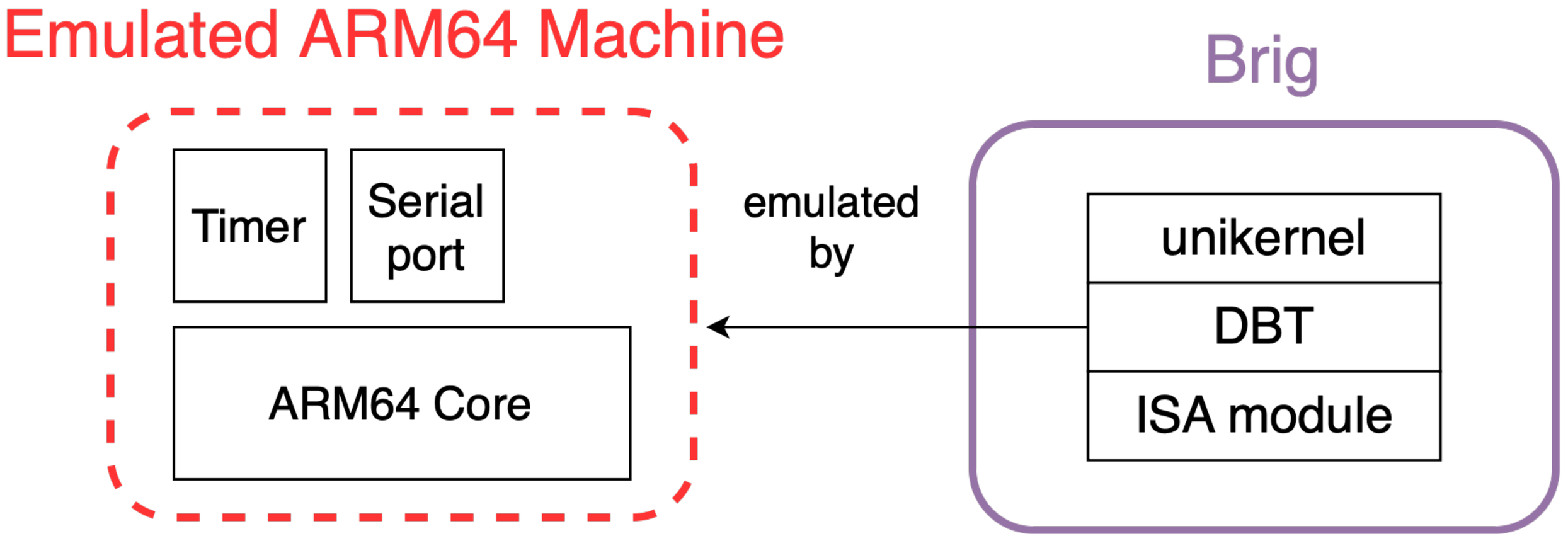
Why all the IRs?

- Can’t modify JIB data structures due to being encoded in OCaml
- Graph IR bridges gap between JIB and SSA IR
- Control flow optimizations
- Compiler built-ins
- Structs, unions, enums, reals, strings, casting, bitvector monomorphization



Brig

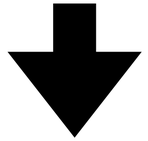
Brig Overview



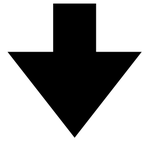
Execution

- Sequence of statements that represent how an instruction should be executed

```
decode(opcode: u32)
```



```
decode_add(opcode: u32)
```



```
execute_add(src: u4, dst: u4, imm: u8)

    s0: read-var imm
    s1: read-reg src
    s2: add s0 s1
    s3: write-reg dst s2
```

Interpret

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src          = 8
s2: add s0 s1
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src          = 8
s2: add s0 s1
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src          = 8
s2: add s0 s1             = 11
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src          = 8
s2: add s0 s1             = 11
s3: write-reg dst s2
```

Interpret

```
s0: read-var imm          = 3
s1: read-reg src          = 8
s2: add s0 s1             = 11
s3: write-reg dst s2
```

Simple, but slow

Translator

- Build tree of nodes
- Collapse tree lazily
- Only capture logic that has side-effects

```
execute_add(src: u4, dst: u4, imm: u8)
                    s0: read-var imm
                    s1: read-reg src
                    s2: add s0 s1
                    s3: write-reg dst s2
```

Translator

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

read-var
imm

Translator

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

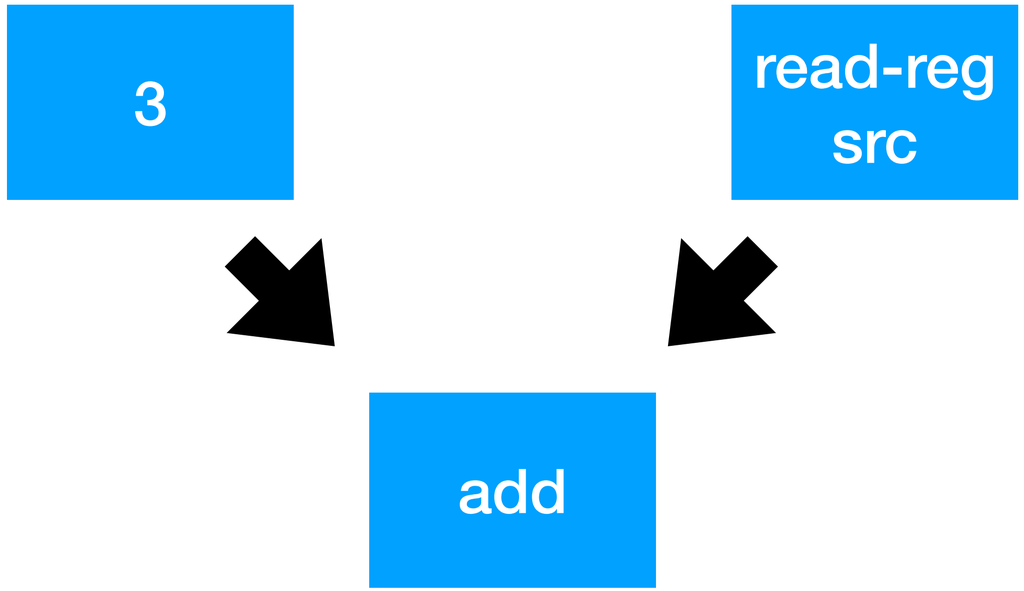
3

Translator

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

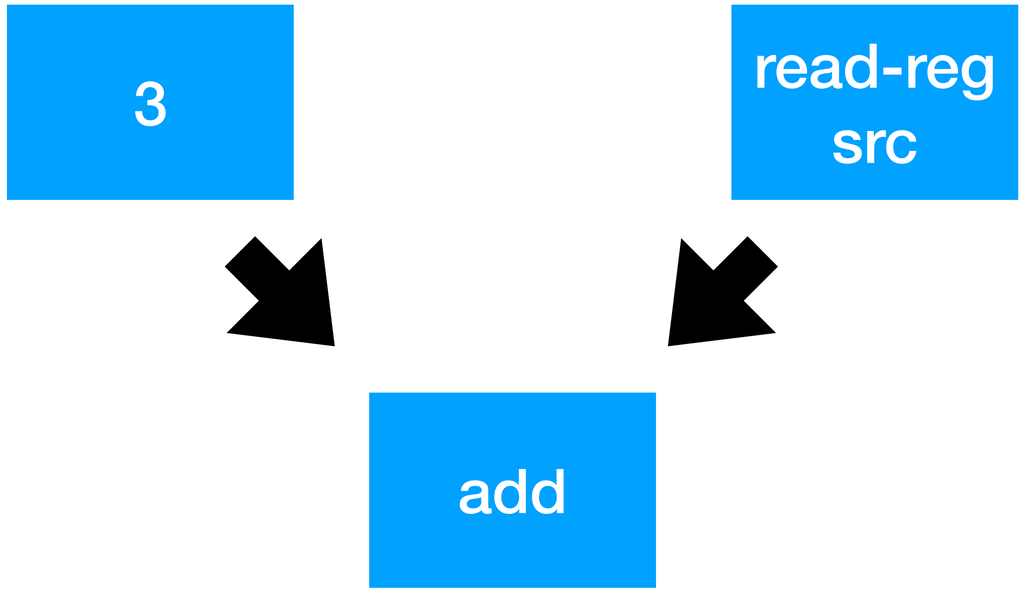
Translator

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```



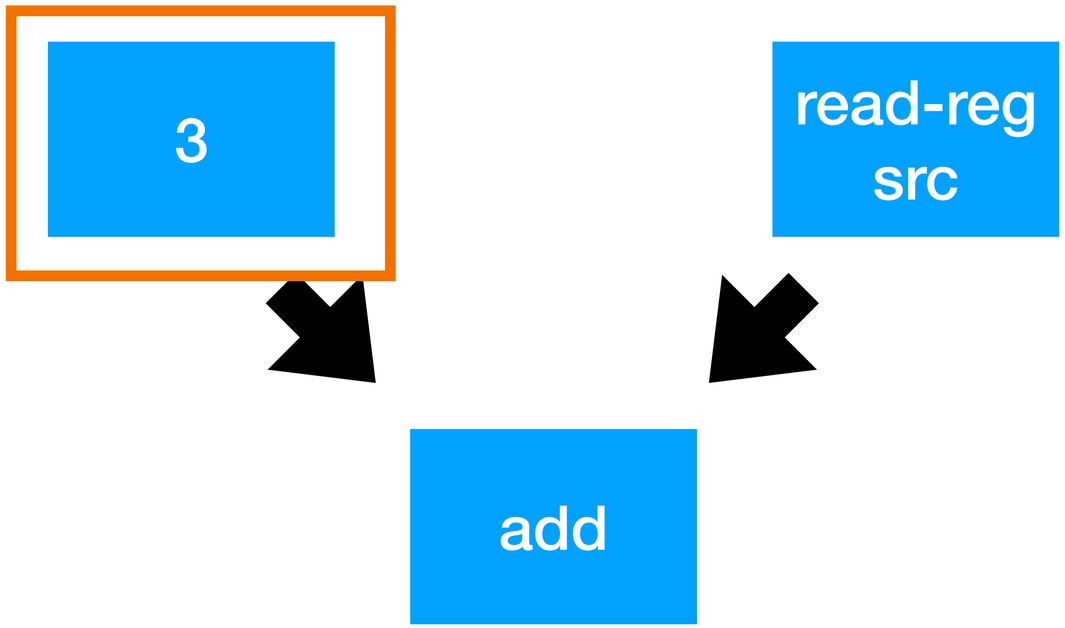
Translator

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```



Translator

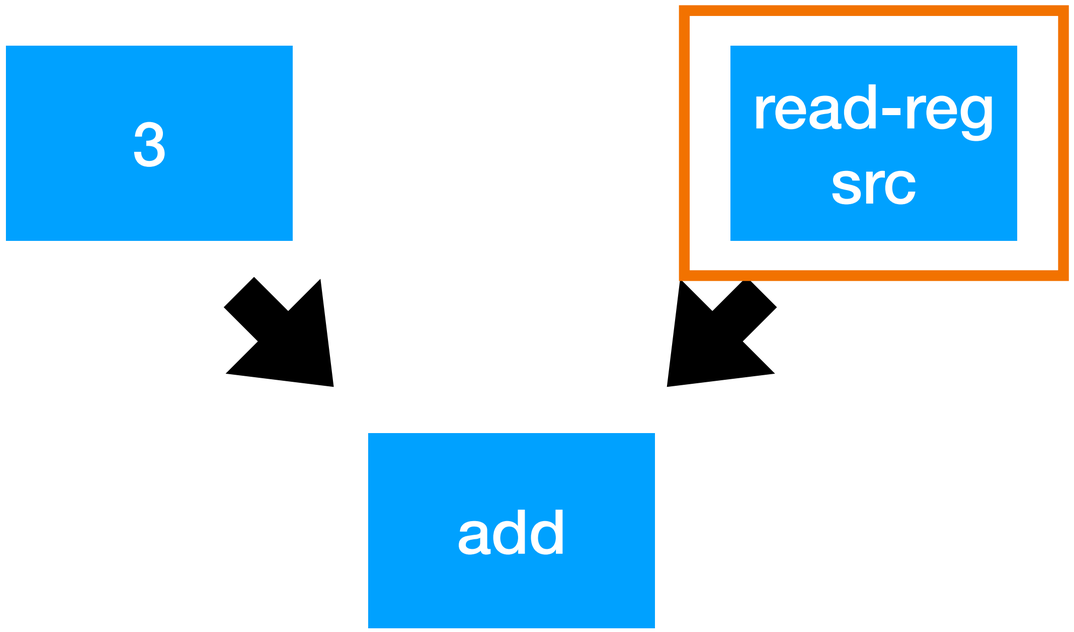
```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```




```
mov $3, v0
```

Translator

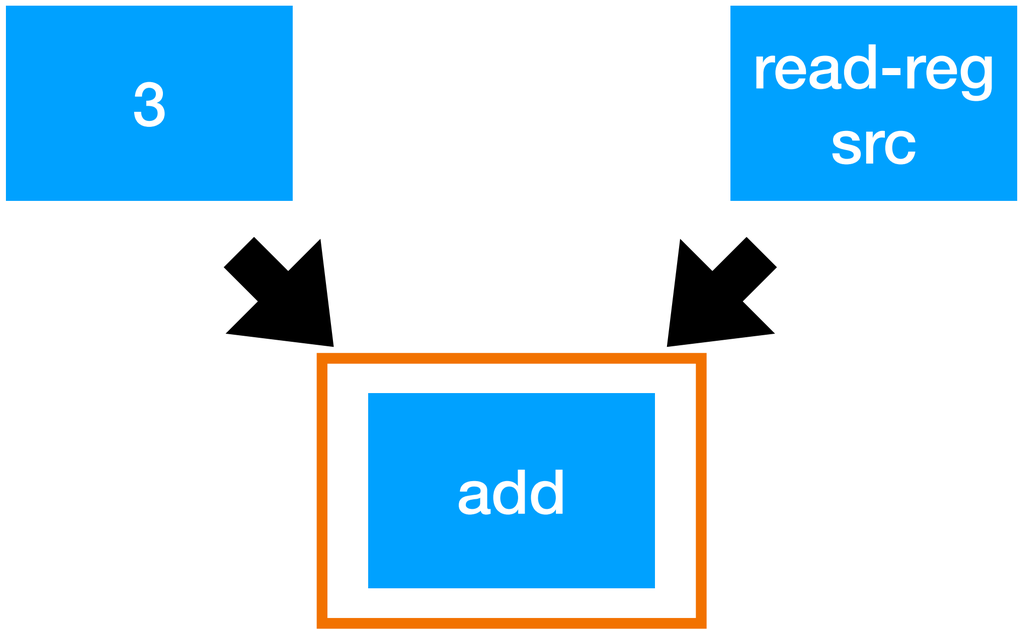
```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```




```
mov $3, v0
mov 0x8(%rbp), v1
```

Translator

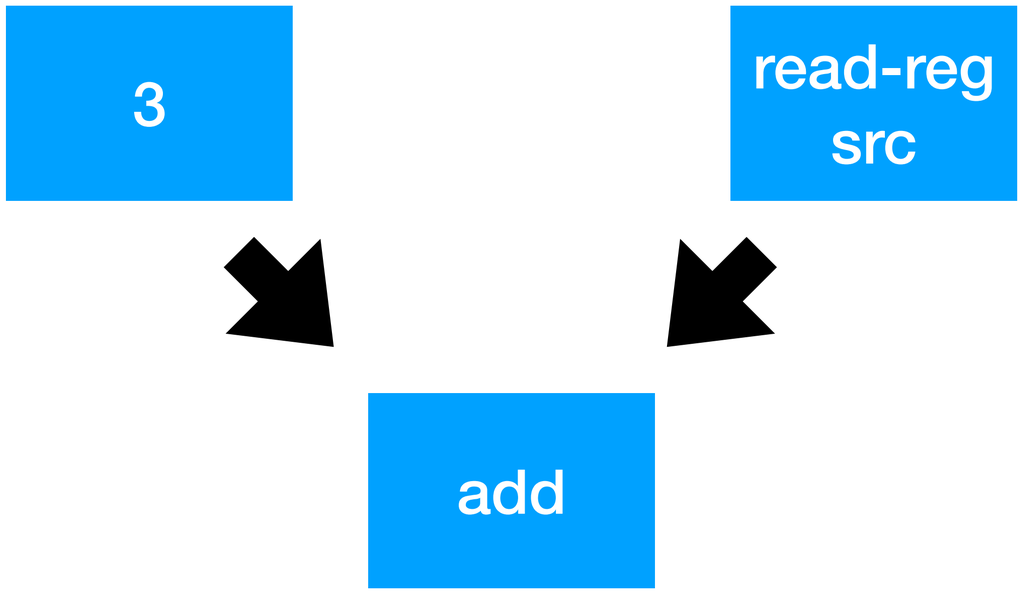
```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```




```
mov $3, v0
mov 0x8(%rbp), v1
add v0, v1
```

Translator

```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```

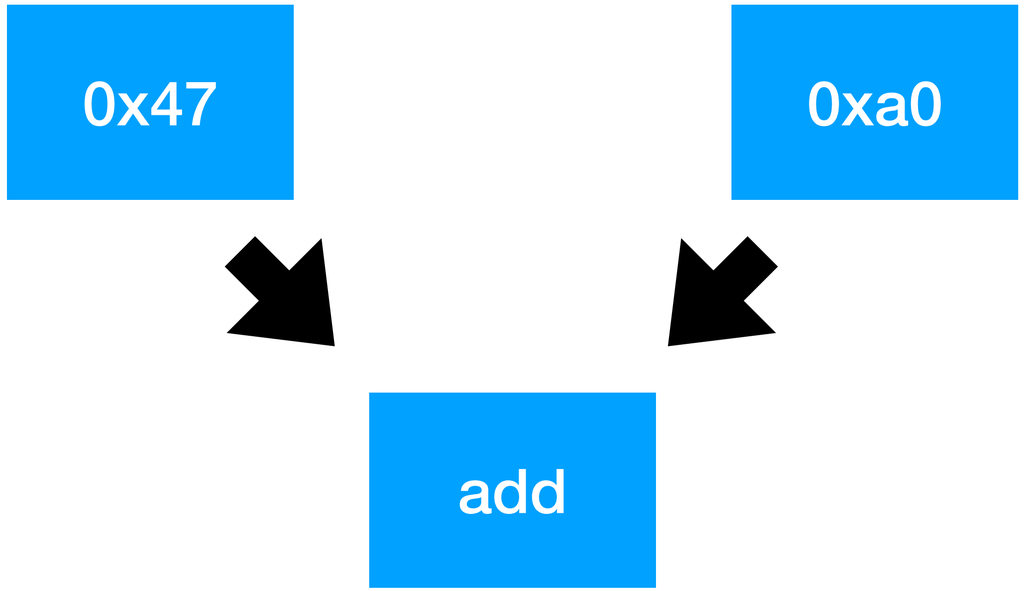



```
mov $3, v0
mov 0x8(%rbp), v1
add v0, v1
mov v1, 0x16(%rbp)
```

Translator

```
mov $3, %rax
mov 0x8(%rbp), %rbx
add %rax, %rbx
mov %rbx, 0x16(%rbp)
```

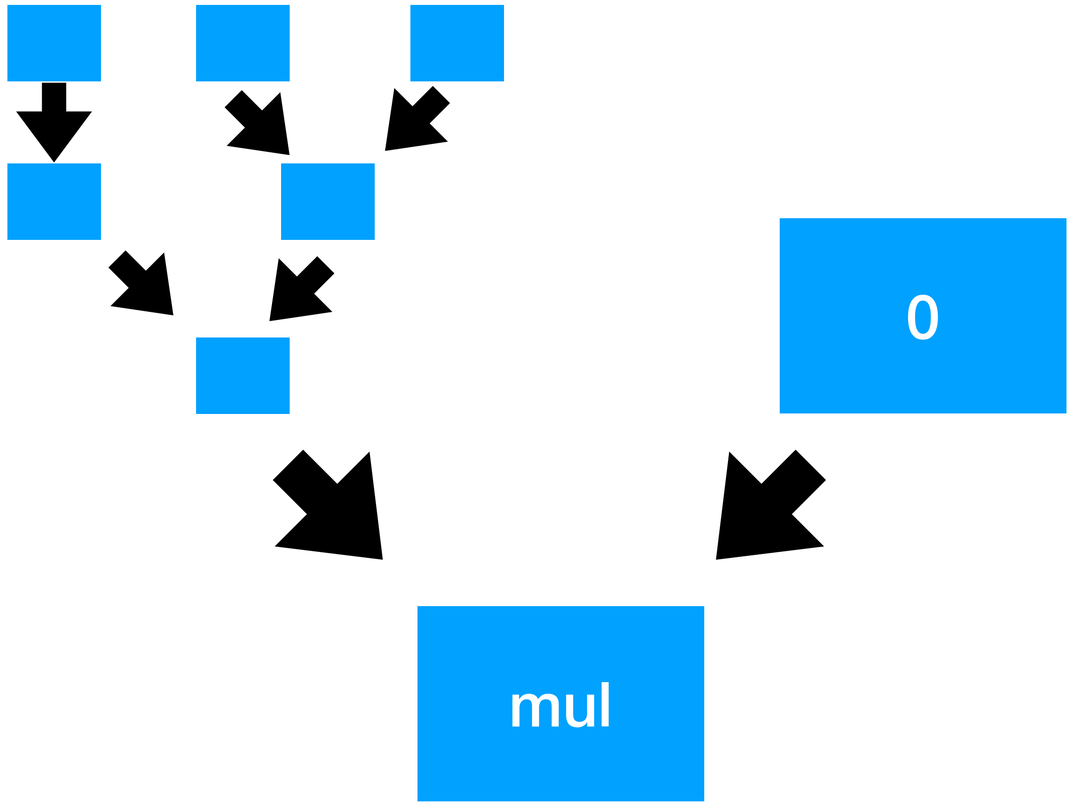
Constant Folding



Constant Folding

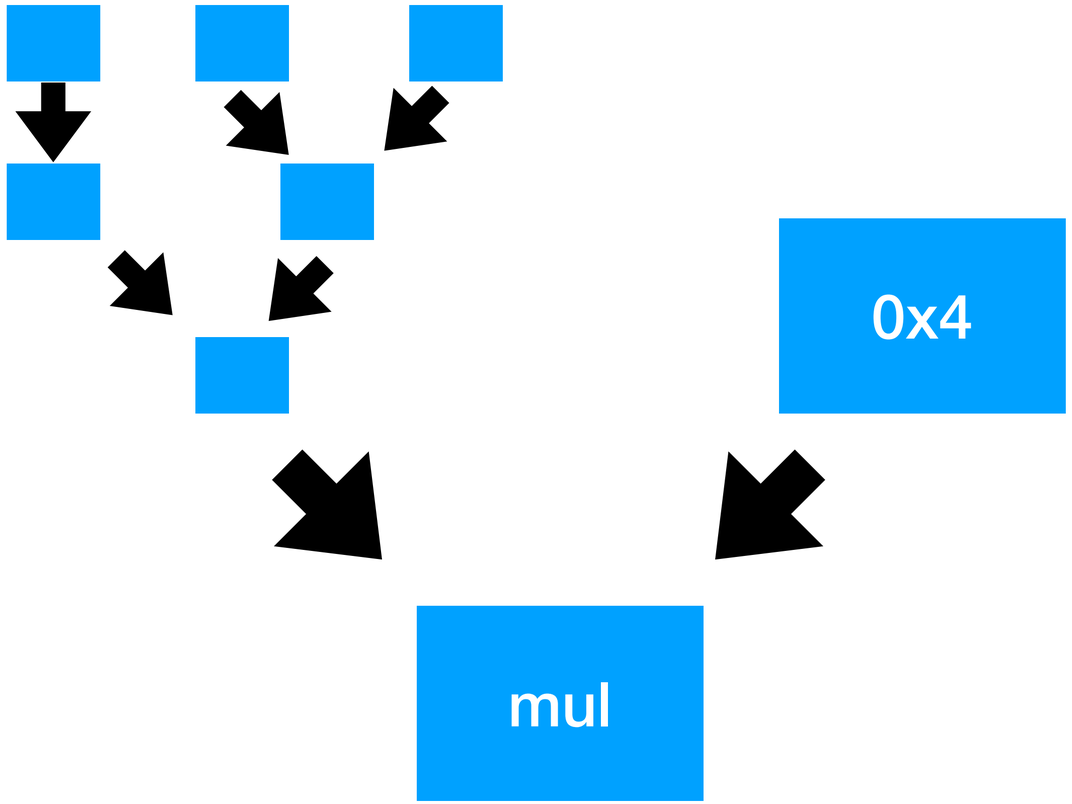
0xe7

Short-circuiting

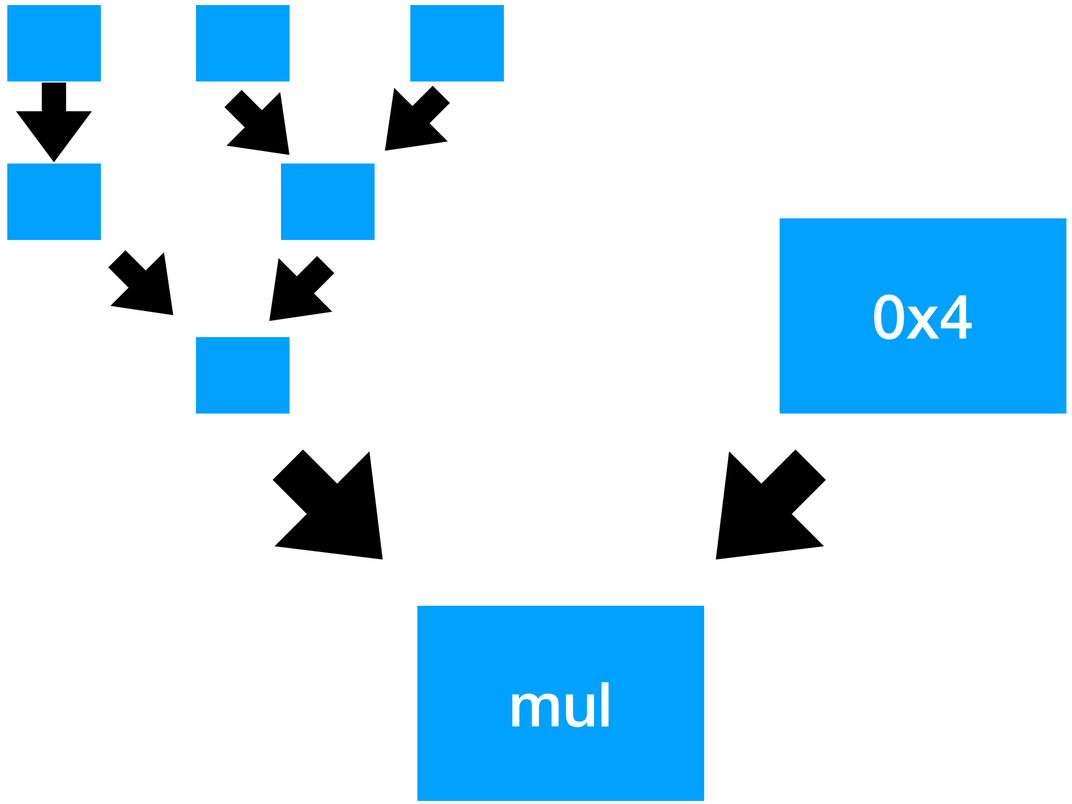


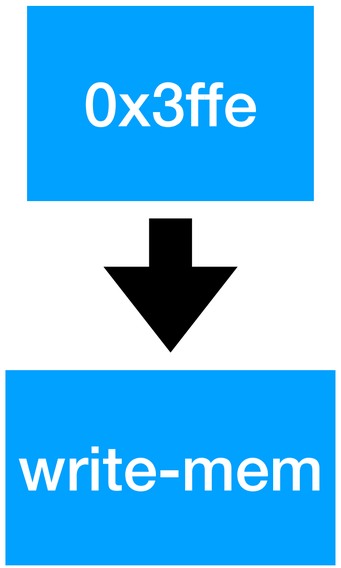
Short-circuiting

Side-effects

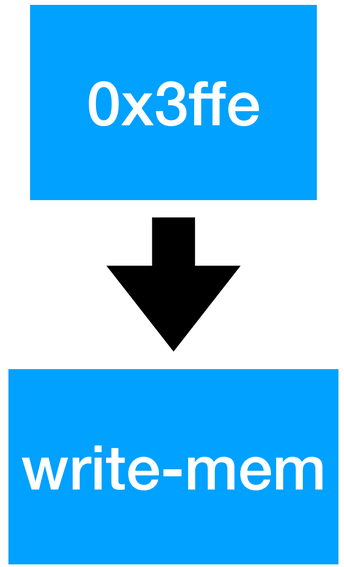


Side-effects



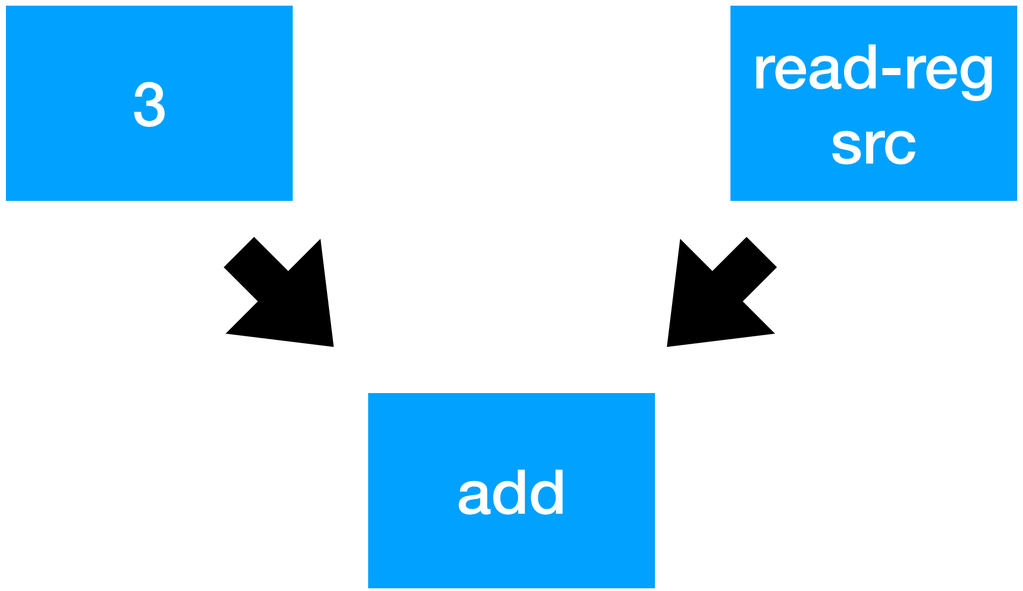


Side-effects

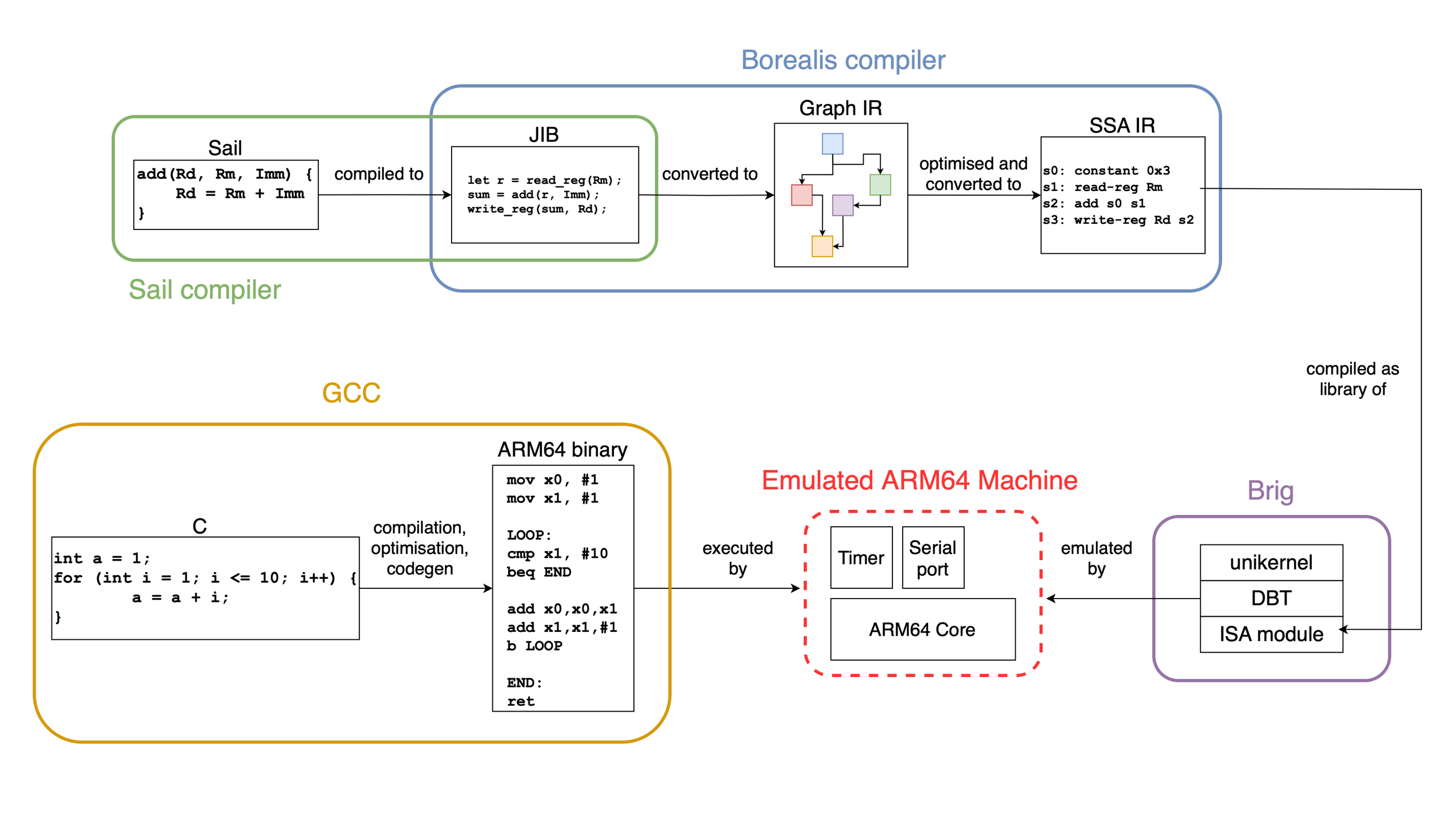


Translator

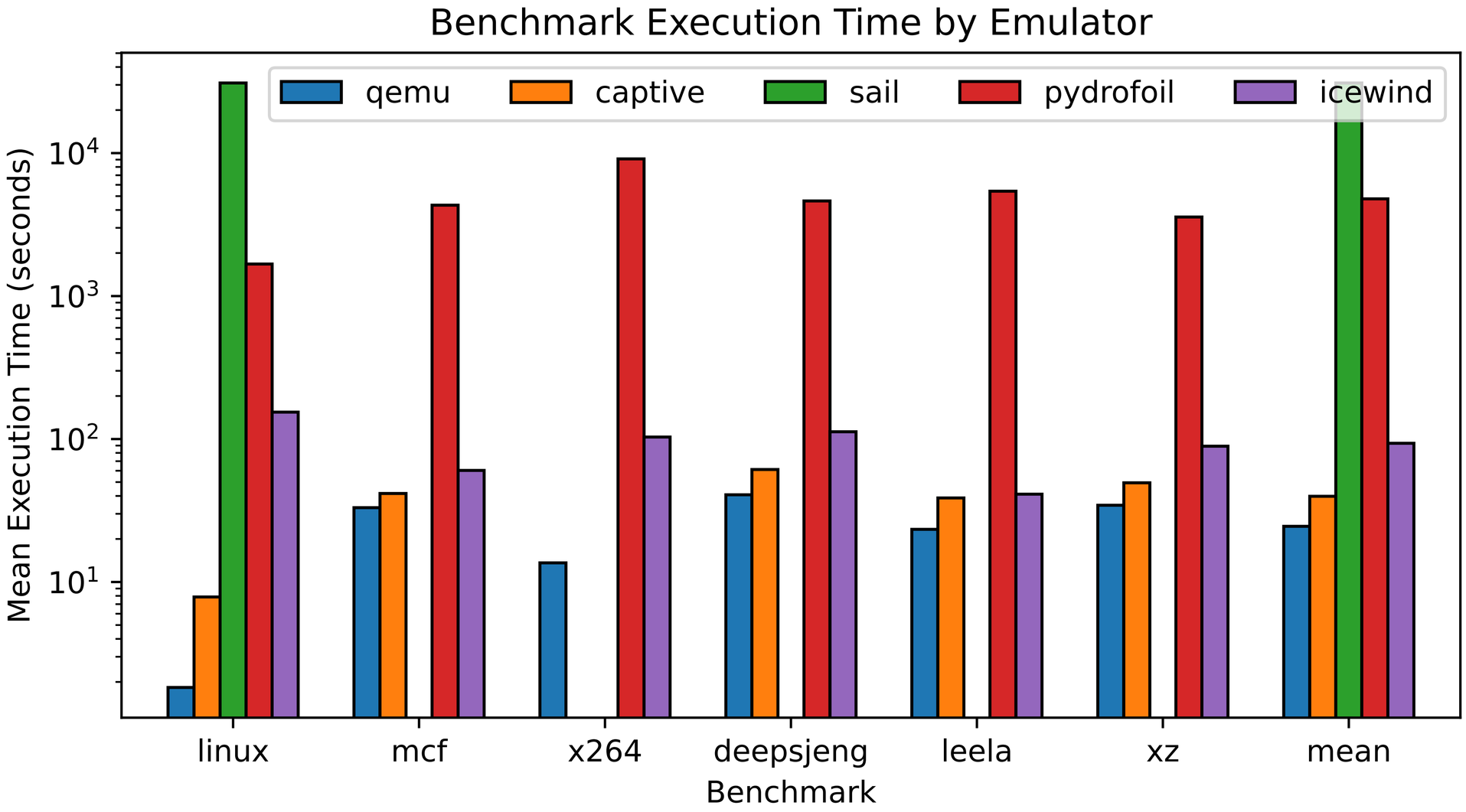
```
s0: read-var imm
s1: read-reg src
s2: add s0 s1
s3: write-reg dst s2
```




```
mov $3, v0
mov 0x8(%rbp), v1
add v0, v1
mov v1, 0x16(%rbp)
```


Borealis compiler
Sail
add(Rd, Rm, Imm) {
Rd = Rm + Imm
}
compiled to
JIB
let r = read_reg(Rm);
sum = add(r, Imm);
write_reg(sum, Rd);
converted to
Graph IR
optimised and converted to
SSA IR
s0: constant 0x3
s1: read-reg Rm
s2: add s0 s1
s3: write-reg Rd s2
Sail compiler
compiled as library of
GCC
C
int a = 1;
for (int i = 1; i <= 10; i++) {
a = a + i;
}
compilation, optimisation, codegen
ARM64 binary
mov x0, #1
mov x1, #1
LOOP:
cmp x1, #10
beq END
add x0,x0,x1
add x1,x1,#1
b LOOP
END:
ret
executed by
Emulated ARM64 Machine
Timer
Serial port
ARM64 Core
emulated by
Brig
unikernel
DBT
ISA module

Results


Benchmark Execution Time by Emulator
qemu
captive
sail
pydrofoil
icewind
Mean Execution Time (seconds)
10^4
10^3
10^2
10^1
linux
mcf
x264
deepsjeng
leela
xz
mean
Benchmark

Results

Relative performance of Icewind to QEMU and Pydrofoil

Benchmark	QEMU	Pydrofoil
Linux boot	0.01	10.86
mcf	0.55	71.67
x264	0.13	88.18
deepsjeng	0.36	41.13
leela	0.57	131.57
xz	0.39	40.08
Geometric Mean	0.20	49.60

Results

Icewind is an automatic emulator that is:

- ~200x faster than the built-in Sail emulator
- ~50x faster than the fastest automatic emulator
- ~5x slower than the fastest manual emulator

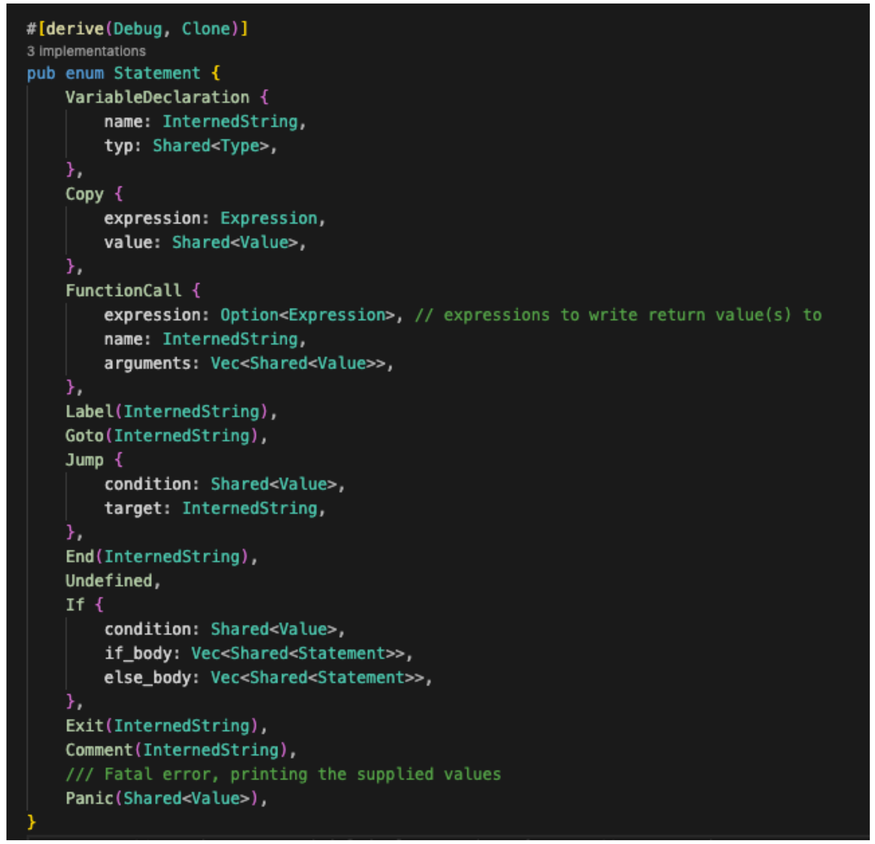
Icewind

Fast Full-System Emulators from Formal Specifications of Instruction Set Architectures

Ferdia McKeogh, University of St Andrews

Supervised by Dr Tom Spink and Prof. Al Dearle

Funded by Adobe, Inc.



```rust
#[derive(Debug, Clone)]
3 implementations
pub enum Statement {
    VariableDeclaration {
        name: InternedString,
        typ: Shared<Type>,
    },
    Copy {
        expression: Expression,
        value: Shared<Value>,
    },
    FunctionCall {
        expression: Option<Expression>, // expressions to write return value(s) to
        name: InternedString,
        arguments: Vec<Shared<Value>>,
    },
    Label(InternedString),
    Goto(InternedString),
    Jump {
        condition: Shared<Value>,
        target: InternedString,
    },
    End(InternedString),
    Undefined,
    If {
        condition: Shared<Value>,
        if_body: Vec<Shared<Statement>>,
        else_body: Vec<Shared<Statement>>,
    },
    Exit(InternedString),
    Comment(InternedString),
    /// Fatal error, printing the supplied values
    Panic(Shared<Value>),
}
```

```rust
#[derive(Debug, Clone)]
4 implementations
pub enum Value {
    Identifier(InternedString),
    Literal(Shared<Literal>),
    Operation(Operation),
    Struct {
        name: InternedString,
        fields: Vec<NamedValue>,
    },
    Field {
        value: Shared<Self>,
        field_name: InternedString,
    },
    CtorKind {
        value: Shared<Self>,
        identifier: InternedString,
        types: Vec<Shared<Type>>,
    },
    CtorUnwrap {
        value: Shared<Self>,
        identifier: InternedString,
        types: Vec<Shared<Type>>,
    },
    Tuple(Vec<Shared<Self>>),
    VectorAccess {
        value: Shared<Self>,
        index: Shared<Self>,
    },
    VectorMutate {
        vector: Shared<Self>,
        element: Shared<Self>,
        index: Shared<Self>,
    },
}
```

```
86BF24897CE77016:
    goto finish_match_108146;

7D7C21A9B9598844:
    goto end_block_exception_108154;

CDF16C86B495CD65:
    goto case_108150;

case_108150:
    i64 u#6926;
    u#6926 = VL;
    bool gs#175668;
    gs#175668 = eq_int(u#6926, 2048);
    if (!gs#175668) {
        goto A634375DDD967C47;
    } else {
        goto 91B55C1B85644B17;
    }

91B55C1B85644B17:
    goto C86B32AA9BAC7DEE;

C86B32AA9BAC7DEE:
    i64 gs#92495;
    gs#92495 = 2048;
    gs#175663 = execute_LDNT1B_MZ_P_BR_4(gs#92495, esize, g, r
    if (have_exception) {
        goto AA1692C7F8B7BF32;
    } else {
        goto AE49A8E66F23B572;
    }
```

```rust
#[derive(Debug, Clone, serde::Serialize, serde::Deserialize)]
5 implementations
pub enum Statement {
    Constant {
        typ: Type,
        value: ConstantValue,
    },

    ReadVariable {
        symbol: Symbol,
    },

    WriteVariable {
        symbol: Symbol,
        value: Ref<Statement>,
    },

    ReadRegister {
        typ: Type,
        /// offset into register state
        ///
        /// During building, this should just be the `next_register_offset`
        /// value, not accessing any elements or fields
        offset: Ref<Statement>,
    },

    WriteRegister {
        /// offset into register state
        ///
        /// During building, this should just be the `next_register_offset`
        /// value, not accessing any elements or fields
        offset: Ref<Statement>,
        value: Ref<Statement>,
    },

    ReadMemory {
        offset: Ref<Statement>,
        size: Ref<Statement>,
    },
    WriteMemory {
        offset: Ref<Statement>,
        value: Ref<Statement>,
    },
```

```
function decode_aarch32_instrs_VCMP_A1enc_A_txt(cond: u4, D: u1, Vd: u4, size: u2, E: u1, M: u1, Vm: u4) -> () :
  block 0x1:
    s0: const #() : ()
    s1: call ConditionPassed(s0)
    s4: branch s1 ? block 0x2 : block 0x11
  block 0x11:
    s0: const #() : ()
    s1: write-var return:() <= s0:()
    s3: return s0
  block 0x2:
    s0: read-var cond:u4
    s1: const #15u : u4
    s2: cmp-ne s0 s1
    s6: assert s2
    s8: read-var size:u2
    s9: const #0u : u2
    s10: cmp-eq s8 s9
    s13: branch s10 ? block 0x5 : block 0x16
  block 0x16:
    s0: read-var size:u2
    s1: const #1u : u2
    s2: cmp-eq s0 s1
    s5: branch s2 ? block 0x5 : block 0x19
  block 0x19:
    s10: read-var size:u2
    s11: const #1u : u2
    s12: cmp-eq s10 s11
    s14: branch s12 ? block 0x8 : block 0xb
  block 0xb:
    s2: read-var E:u1
    s3: const #1u : u1
    s4: cmp-eq s2 s3
    s5: write-var quiet_nan_exc:u1 <= s4:u1
    s8: const #16s : i64
    s9: write-var esize:i64 <= s8:i64
    s10: const #0s : i64
    s18: write-var d:i64 <= s10:i64
    s19: const #0s : i64
    s27: write-var m:i64 <= s19:i64
    s28: read-var size:u2
    s31: const #1u : u2
    s32: cmp-eq s28 s31
    s35: not s32
    s36: branch s35 ? block 0xc : block 0x14
```