南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Compilers: Three Easy Pieces

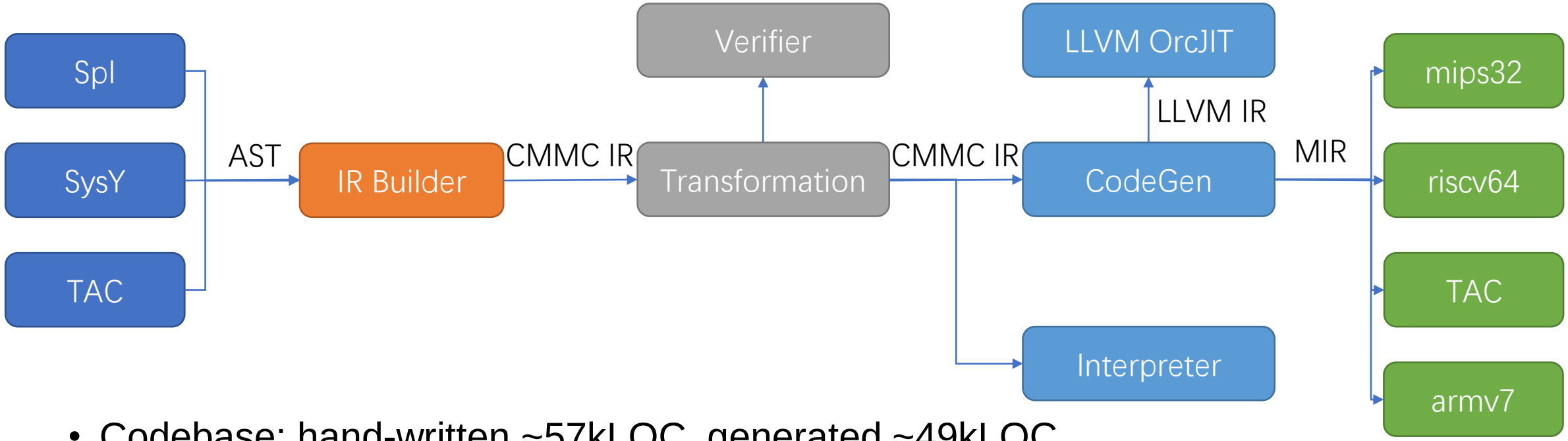
Yingwei Zheng

Bingzhen Wang

2023/9/21

A brief introduction to CMMC

CMMC (C Minus Minus Compiler)



- Codebase: hand-written ~57kLOC, generated ~49kLOC
- Source available on GitHub: https://github.com/dtcxzyw/cmmc/

Team members

- 郑英炜 @dtcxzyw
  - ISCAS PLCT Lab Intern, LLVM Committer (Transform/RISC-V backend)
  - Contribution: Infrastructure/Transform
- 王炳臻 @infiWang
  - ISCAS PLCT Lab Intern, luajit rv64 port contributor
  - Contribution: RISC-V backend
- 邬一帆 @GhostFrankWu
  - CTFer, COMPASS CTF Team Leader
  - Contribution: Regression testing/Fuzzing
- 严文谦 @YanWQ-monad
  - CTFer
  - Contribution: Transform/ARM-v7 backend/CI

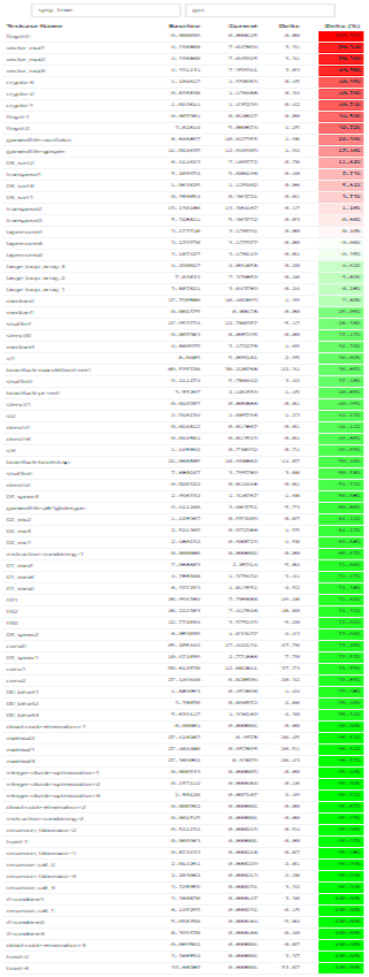
Highlights

2023全国大学生计算机系统能力大赛 编译系统设计赛（华为毕昇杯）编译系统设计赛道 决赛入围名单					
队伍编号	队名/学校	功能分	性能分	总分	赛道
202314325201374	CMMC/ 南方科技大学	100	99.8496	99.9248	RISC-V
202314325201374	CMMC/ 南方科技大学	100	90.7329	95.3664	ARM
202310006201934	那一年喵喵变成了光/ 北京航空航天大学	100	78.188	89.094	ARM
202310006201725	ATRI/ 北京航空航天大学	100	73.707	86.8535	ARM
202310055201427	没有op就不配拿奖吗/ 南开大学	100	65.8981	82.9491	ARM
202310532201184	卷窝鸣人/ 湖南大学	100	58.1337	79.0668	ARM
202310055201422	NKUF4/ 南开大学	100	57.3885	78.6943	ARM
202310006201725	ATRI/ 北京航空航天大学	98	50.6447	74.3223	RISC-V

Preliminary result

#	用户名	队伍	提交次数(ASC)	最后提交时间(ASC)	正确分	性能分	总分
1	202314325201374	CMMC/ 南方科技大学	8	2023-08-21 16:23:25	98	92.6925	93.8708
2	202310007201692	bit.newnewcc/ 北京理工大学	23	2023-08-21 14:18:13	100	49.3855	60.6219
3	202310558201558	Yat-CC/ 中山大学	17	2023-08-21 14:53:12	99	47.8161	59.1789
4	202310006201898	喵喵队引体向上/ 北京航空航天大学	38	2023-08-21 16:50:16	100	45.5433	57.6327
5	202310055201721	生成式智能人工队/ 南开大学	18	2023-08-21 17:18:56	100	45.4676	57.5738

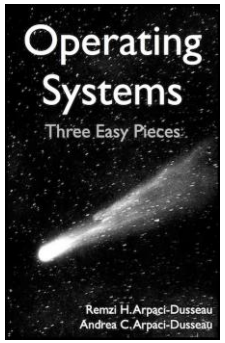
Final result



CMMC vs Baseline (GCC -O3)

Table of Contents

- Operating Systems: Three Easy Pieces
  - Virtualization
  - Concurrency
  - Persistence



- Compiler = pattern matching + heuristics algorithms + decision tree
- Compiler is a highly-tuned expert system!!!

- Compilers: Three Easy Pieces
  - Abstraction
  - Canonicalization
  - Legalization

Part Ⅰ Abstraction

We need abstraction

- Target-independent optimizations
  - IR -> X86, IR -> RISC-V
  - How to describe instructions/registers/side effects?
- Provide heuristics algorithms with a cost model
  - mul takes 3 cycles, div takes 6-68 cycles on sifive-u74
- Reuse logic to simplify similar patterns
  - a in [0, 16), b in [-15, 0] → b <= a
  - a = x & 0xf ? a = b + 15 ? a = abs(x % 16) ?

Part Ⅰ Abstraction

- Abstraction for values
  - Constant range + known bits
  - SCEV
- Abstraction for memory
  - Memory access
  - Address generation
  - Alias analysis
- Abstraction for control flow
  - Block freq estimation
  - Constraint elimination
- Abstraction for ISA
  - Instructions
  - Registers
- Abstraction for microarchitecture
  - Instruction scheduling
  - Macro fusion

Part Ⅰ Abstraction for scalar values

- Constant range
  - [signed min, signed max], [unsigned min, unsigned max]
  - transfer function: a in [0, 1], b in [1, 4] → (a + b) in [1, 5]
  - contextual range: if (0 <= a && a < 16) f(a /* a in [0, 16) */);

- Known bits
  - (known zeros, known ones)
  - int4 a; a & (1 << 2) → known zeros = 0b0100 known ones = 0

Part Ⅰ Abstraction for induction variables

SCEV (Scalar Evolution)

- v = {initial, op, step } → v0 = initial, vk = initial + k * step
- Basic Induction Variable (BIV): i = {0, +, 1}
- Generalized Induction Variable (GIV):
  - sum = {0, +, {0, +, 1}} = {0, +, 0, +, 1}
- SaaS (SCEV as a Service)
  - SCEV-based Alias Analysis
  - IndVar simplification
  - Loop trip count

```
int test() {
    int sum = 0;
    for (int i = 0; i < 32; ++i)
        sum += i;
    return sum;
}
```

- [Scalar Evolution - Demystified (llvm.org)]

Part Ⅰ Abstraction for memory access

- Source
  - global variables
  - stack allocations
  - arguments
- Action: Load/Store/~~Atomic ops/CAS~~
- Address generation (later)
- Alias analysis
  - Color-based alias analysis (later)
  - Type-based alias analysis (TBAA) (float* != int*)
  - Interprocedural alias analysis (backtrace along ptr args/escape analysis of stack address)
  - SCEV-based Alias Analysis (&A[i] != &A[i+1] in loops)
- ~~Volatile~~

Part Ⅰ Abstraction for address generation

- ~~ptrtoint/inttoptr/ptrcast~~
- getelementptr (LLVM/CMMC)
  - int A[4][4] → [4 x [4 x i32]]* A
  - A[i] → [4 x i32]* ptr1 = getelementptr (*A)[0][i]
  - A[i][j] → i32* ptr2 = getelementptr (*ptr1)[0][j]
  - A[i + 1] → [4 x i32]* ptr3 = getelementptr (*ptr1)[1]
- ptradd (CMMC only)
  - A[1][2] → ptradd [4 x [4 x i32]]* A, 24
  - Motivation by nikic: [RFC] Replacing getelementptr with ptradd - IR & Optimizations - LLVM Discussion Forums
  - Implemented in the CGP(CodeGenPrepare) stage of CMMC

Part Ⅰ Abstraction for address generation

ptradd in stencil computation

```
int commonbase(int a[500][500], int i, int j) {
    return a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] +
      a[i][j-1] + a[i][j] + a[i][j+1] +
      a[i+1][j-1] + a[i+1][j] + a[i+1][j+1];
}
```

- common base opt
- generate addresses from a[i][j]

```
func @commonbase([500 * [500 * i32]]* %a, i32 %i, i32 %j) -> i32 { NoMemoryWrite NoSideEffect NoRecurse } {
  ^entry:
    [500 * i32]* %0 = getelementptr &([500 * [500 * i32]]* %a)[i64 0][i32 %i];
    [500 * i32]* %1 = getelementptr &([500 * i32]* %0)[i64 -1];
    i32* %2 = getelementptr &([500 * i32]* %1)[i64 0][i32 %j];
    i32* %3 = getelementptr &(i32* %2)[i64 -1];
    i32 %4 = load i32* %3;
    i32 %5 = load i32* %2;
    i32 %6 = add i32 %4, i32 %5;
    i32* %7 = getelementptr &(i32* %2)[i64 1];
    i32 %8 = load i32* %7;
    i32 %9 = add i32 %6, i32 %8;
    i32* %10 = getelementptr &([500 * i32]* %0)[i64 0][i32 %j];
    i32* %11 = getelementptr &(i32* %10)[i64 -1];
    i32 %12 = load i32* %11;
    i32 %13 = add i32 %9, i32 %12;
    i32 %14 = load i32* %10;
    i32 %15 = add i32 %13, i32 %14;
    i32* %16 = getelementptr &(i32* %10)[i64 1];
    i32 %17 = load i32* %16;
    i32 %18 = add i32 %15, i32 %17;
    [500 * i32]* %19 = getelementptr &([500 * i32]* %0)[i64 1];
    i32* %20 = getelementptr &([500 * i32]* %19)[i64 0][i32 %j];
    i32* %21 = getelementptr &(i32* %20)[i64 -1];
    i32 %22 = load i32* %21;
    i32 %23 = add i32 %18, i32 %22;
    i32 %24 = load i32* %20;
    i32 %25 = add i32 %23, i32 %24;
    i32* %26 = getelementptr &(i32* %20)[i64 1];
    i32 %27 = load i32* %26;
    i32 %28 = add i32 %25, i32 %27;
    ret i32 %28;
}
```

```
func @commonbase([500 * [500 * i32]]* %a, i32 %i, i32 %j) -> i32 { NoMemoryWrite NoSideEffect NoRecurse } {
  ^entry:
    [500 * i32]* %0 = getelementptr &([500 * [500 * i32]]* %a)[i64 0][i32 %i];
    i32* %1 = getelementptr &([500 * i32]* %0)[i64 0][i32 %j];
    i32* %2 = ptradd i32* %1, i32 -2004;
    i32 %3 = load i32* %2;
    i32* %4 = ptradd i32* %1, i32 -2000;
    i32 %5 = load i32* %4;
    i32 %6 = add i32 %3, i32 %5;
    i32* %7 = ptradd i32* %1, i32 -1996;
    i32 %8 = load i32* %7;
    i32 %9 = add i32 %6, i32 %8;
    i32* %10 = ptradd i32* %1, i32 -4;
    i32 %11 = load i32* %10;
    i32 %12 = add i32 %9, i32 %11;
    i32 %13 = load i32* %1;
    i32 %14 = add i32 %12, i32 %13;
    i32* %15 = ptradd i32* %1, i32 4;
    i32 %16 = load i32* %15;
    i32 %17 = add i32 %14, i32 %16;
    i32* %18 = ptradd i32* %1, i32 1996;
    i32 %19 = load i32* %18;
    i32 %20 = add i32 %17, i32 %19;
    i32* %21 = ptradd i32* %1, i32 2000;
    i32 %22 = load i32* %21;
    i32 %23 = add i32 %20, i32 %22;
    i32* %24 = ptradd i32* %1, i32 2004;
    i32 %25 = load i32* %24;
    i32 %26 = add i32 %23, i32 %25;
    ret i32 %26;
}
```

- [D150862 [RISCV][CodeGenPrepare] Select the optimal base offset for GEPs with large offset (llvm.org)]

Part Ⅰ Abstraction for address generation

results from llvm-mca (sifive-u74):

- CMMC: **17 cycles**, IPC = 1.23
- Clang: 22 cycles, IPC = 1.72
- GCC: 20 cycles, IPC = 1.4

- https://godbolt.org/z/4asTPT45T

```
commonbase:
        li t0, 2000
        mul a4, a1, t0
        add a5, a0, a4
        sh2add a3, a2, a5
        lw a1, -2004(a3)
        lw a0, -2000(a3)
        lw t0, -1996(a3)
        addw a2, a1, a0
        lw a5, -4(a3)
        addw a4, a2, t0
        lw a2, 0(a3)
        addw a0, a4, a5
        lw a5, 4(a3)
        addw a1, a0, a2
        lw a0, 1996(a3)
        addw a4, a1, a5
        lw a5, 2000(a3)
        addw a2, a4, a0
        lw a4, 2004(a3)
        addw a1, a2, a5
        addw a0, a1, a4
```

CMMC O3

```
commonbase(int (*) [500], int, int):
        addiw   a3, a1, -1
        li      a6, 2000
        slli    a2, a2, 2
        mul     a3, a3, a6
        addi    a5, a2, -4
        add     a3, a3, a0
        add     t0, a3, a5
        add     a7, a3, a2
        addi    t1, a2, 4
        mul     a4, a1, a6
        add     t2, a3, t1
        add     a4, a4, a0
        lw      t3, 0(t0)
        add     t0, a4, a5
        lw      a7, 0(a7)
        addiw   a1, a1, 1
        lw      t2, 0(t2)
        add     a7, a7, t3
        lw      a3, 0(t0)
        mul     a1, a1, a6
        add     a6, a4, a2
        add     a4, a4, t1
        add     a0, a0, a1
        add     a3, a3, t2
        add     a5, a5, a0
        lw      a1, 0(a6)
        add     a2, a2, a0
        lw      a4, 0(a4)
        add     a0, a0, t1
        add     a3, a3, a7
        lw      a5, 0(a5)
        add     a1, a1, a4
        lw      a2, 0(a2)
        add     a1, a1, a5
        lw      a0, 0(a0)
        add     a1, a1, a3
        add     a0, a0, a2
        addw    a0, a0, a1
```

Clang O3

```
commonbase(int (*) [500], int, int):
        li      a5,2000
        slli    a2,a2,2
        mul     a1,a1,a5
        addi    a5,a1,-2000
        add     a4,a1,a2
        add     a5,a5,a2
        add     a4,a0,a4
        add     a5,a0,a5
        addi    a1,a1,2000
        lw      a6,-4(a5)
        lw      a3,0(a5)
        addw    a3,a3,a6
        lw      a5,4(a5)
        addw    a5,a5,a3
        lw      a3,-4(a4)
        addw    a3,a3,a5
        add     a1,a1,a2
        lw      a5,0(a4)
        add     a0,a0,a1
        addw    a5,a5,a3
        lw      a3,4(a4)
        addw    a3,a3,a5
        lw      a4,-4(a0)
        lw      a5,0(a0)
        addw    a4,a4,a3
        lw      a0,4(a0)
        addw    a5,a5,a4
        addw    a0,a0,a5
```

GCC O3

Part Ⅰ Abstraction for alias analysis

Color-based alias analysis

- ptr = set of colors, (c1, c2) in S → ptr1 with c1 and ptr2 with c2 are distinct.
- Rules:
  - global vars ←→ stack objects
  - stack object ←→ stack object
  - A[idx1] ←→ A[idx2] iff idx1 != idx2
- Transfer functions

Part Ⅰ Abstraction for block freq estimation

- Which blocks are “hot” ?
- Estimated branch probabilities (uniform/loop exit) → block freq
- E[entry] = 1
- E[block] = sum E[pred] * prob
- Ax = b, b = [1 0 0 0 0], A[i][j] = (i == j) + sum prob(j -> i)
- Solve:
  - LUP decomposition
  - (A – I) is a Markov matrix?

Part Ⅰ Abstraction for constraint elimination

Simplify compares implied by dom conditions:

```
if (i > j)
    f(i == j); // always false
```

- CMMC: Relations + Transitive Closure + Brute-force SAT
  - a < b, b <= c → a < c
  - (a < b) & (b < a) → false
- LLVM: Linear constraints c >= c_i * v_i
  - -1 >= a - b, 0 >= b – c → -1 >= a – c → a < c

Part Ⅰ Abstraction for instructions

- Operands
  - Register (VReg/GPR/FPR/CSR Def/Use)
  - Immediate
  - Relocatable (b target/call target)
  - Expression (.label1 - .label2)
- Side effects
  - Control flow (return, call, branch, fall through)
  - Memory (load, store)

Part Ⅰ Abstraction for instructions

How to represent a class of instructions?

CMMC:

```
## 2.6 Load and Store Instructions
Load:
  Format: "$Mnemonic:Template $Rd:GPR[Def], $Imm:Imm12[Metadata]($Rs1:BaseLike[Use]) # $Alignment:Align[Metadata]"
  Flag: [Load]
InstanceLoad:
  Template: Load
  LB:
    Mnemonic: lb
  LH:
    Mnemonic: lh
```

→ Jinja2 → implementations of the base class InstInfo

LLVM:

```
// Pseudo load instructions.
class PseudoLoad<string opcodestr, RegisterClass rdty = GPR>        Kito Cheng, 5 years
    : Pseudo<(outs rdty:$rd), (ins bare_symbol:$addr), [], opcodestr, "$rd, $addr"> {
  let hasSideEffects = 0;
  let mayLoad = 1;
  let mayStore = 0;
  let isCodeGenOnly = 0;
  let isAsmParserOnly = 1;
}
```

```
def PseudoLB  : PseudoLoad<"lb">;
def PseudoLBU : PseudoLoad<"lbu">;
def PseudoLH  : PseudoLoad<"lh">;
def PseudoLHU : PseudoLoad<"lhu">;
def PseudoLW  : PseudoLoad<"lw">;
```

→ TableGen

Part Ⅰ Abstraction for instructions

Generated code for RISCV.LB in CMMC:

```
class RISCVInstInfoLB final : public InstInfo {
public:
    RISCVInstInfoLB() = default;
    void print(std::ostream& out, const MIRInst& inst, bool printComment) const override {
        CMMC_UNUSED(inst);
        out << "lb " << ::cmmc::mir::RISCV::OperandDumper{ inst.getOperand(0) } << ", "
            << ::cmmc::mir::RISCV::OperandDumper{ inst.getOperand(1) } << "("
            << ::cmmc::mir::RISCV::OperandDumper{ inst.getOperand(2) } << ")";
        if(printComment)
            out << " # " << ::cmmc::mir::RISCV::OperandDumper{ inst.getOperand(3) };
    }
    [[nodiscard]] bool verify(const MIRInst& inst, const CodeGenContext& ctx) const override {
        return inst.checkOperandCount(4) && mir::checkISASpecificOperands(inst, ctx, 4) &&
            ::cmmc::mir::RISCV::isOperandGPR(inst.getOperand(0)) && ::cmmc::mir::RISCV::isOperandImm12(inst.getOperand(1)) &&
            ::cmmc::mir::RISCV::isOperandBaseLike(inst.getOperand(2)) && ::cmmc::mir::RISCV::isOperandAlign(inst.getOperand(3));
    }
    [[nodiscard]] uint32_t getOperandNum() const noexcept override {
        return 4;
    }
    [[nodiscard]] OperandFlag getOperandFlag(uint32_t idx) const noexcept override {
        switch(idx) {
            case 0:
                return OperandFlagDef;
            case 1:
                return OperandFlagMetadata;
            case 2:
                return OperandFlagUse;
            case 3:
                return OperandFlagMetadata;
            default:
                reportUnreachable(location: CMMC_LOCATION());
        }
    }
    [[nodiscard]] InstFlag getInstFlag() const noexcept override {
        return InstFlagNone | InstFlagLoad;
    }
    [[nodiscard]] std::string_view getUniqueName() const noexcept override {
        return "RISCV.LB";
    }
};
```

Part Ⅰ Abstraction for instructions

How to represent an instruction?

Opcode + Operand Array

- Registers: live variables analysis → dead/killed
- MIRModule
  - MIRGlobal
    - MIRFunction
      - MIRBasicBlock
        - **MIRInst**
    - MIRGlobalVariable
      - .data
      - .rodata
      - .bss

Part Ⅰ Abstraction for instructions

How to represent a two-address instruction?

- CMMC: Implicit Use

  ARMv7.MOVT:

```
MOVT:
  Format: "movt $Rd:GPR[Def], $Imm:UImm16[Metadata] @ Implicit Use: $Rs:GPR[Use]"
  CustomVerifier: true
```

- LLVM: tied-to-def/early-clobber

```
let Predicates = [HasStdExtZacas] in {
class PseudoAMOCAS<RegisterClass RC = GPR>
    : Pseudo<(outs RC:$res),
             (ins GPR:$addr, RC:$cmpval, RC:$newval, ixlenimm:$ordering), []> {
  let Constraints = "@earlyclobber $res, $res = $cmpval";
  let mayLoad = 1;
  let mayStore = 1;
  let hasSideEffects = 0;
}
```

Part Ⅰ Abstraction for instructions

How to identify/modify a (conditional) branch after RA?

- Generic/pseudo machine inst
  - Pros: target-independent
  - Cons: bad isel, invalid inst (see Legalization part)
- LLVM/CMMC: pattern match/in-place modification

```
virtual bool matchBranch(const MIRInst& inst, MIRBasicBlock*& target, double& prob) const;
bool matchConditionalBranch(const MIRInst& inst, MIRBasicBlock*& target, double& prob) const;
bool matchUnconditionalBranch(const MIRInst& inst, MIRBasicBlock*& target) const;
bool matchCopy(const MIRInst& inst, MIROperand& dst, MIROperand& src) const;
virtual void redirectBranch(MIRInst& inst, MIRBasicBlock* target) const;
virtual MIRInst emitGoto(MIRBasicBlock* target) const = 0;
virtual void inverseBranch(MIRInst& inst, MIRBasicBlock* newTarget) const = 0;
```

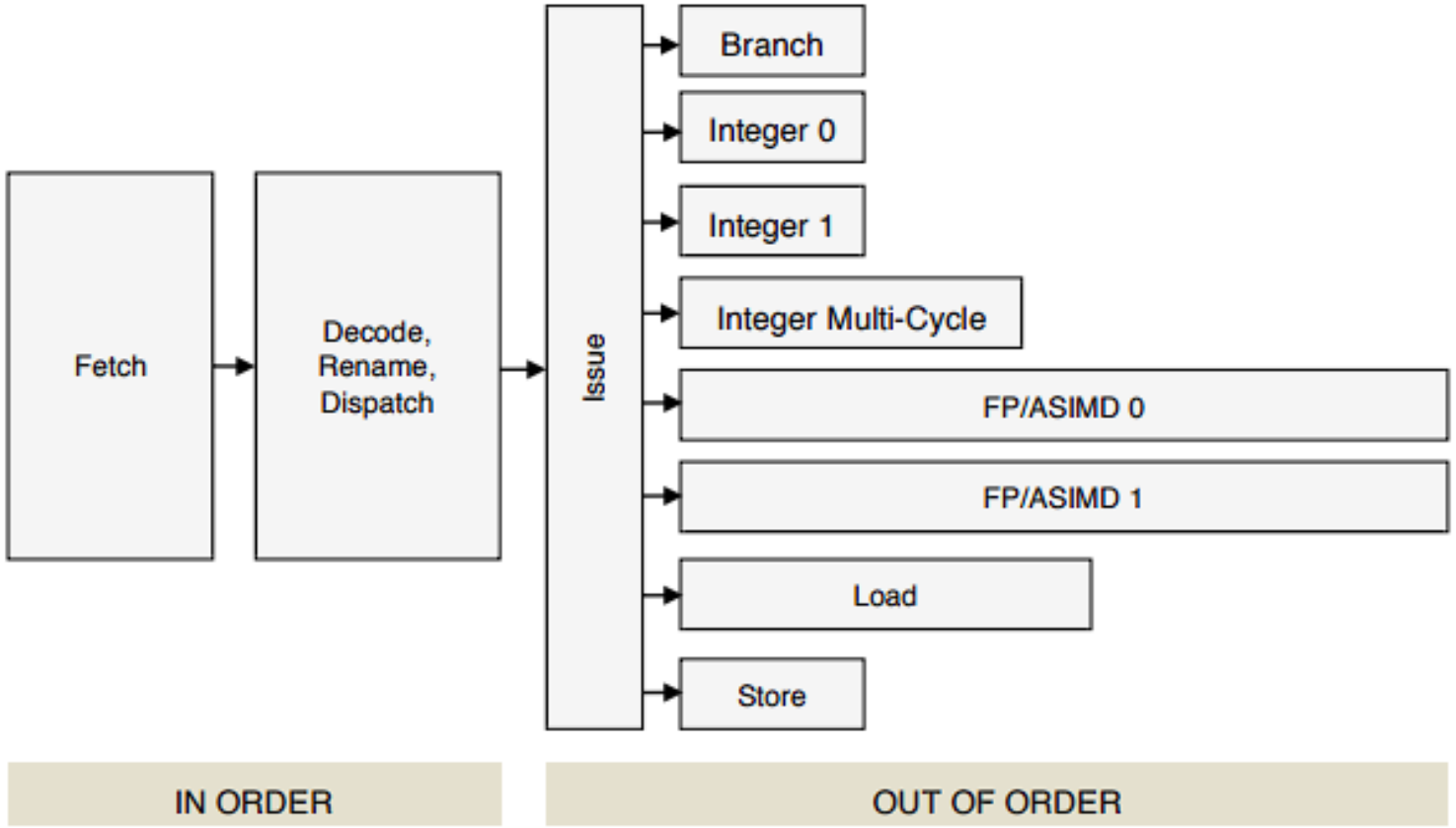
  - Pros: good isel
  - Cons: implemented by targets (generated code)

Part Ⅰ Abstraction for registers

- Virtual register (id + type)
- Stack slot/Frame index (id + type + offset + size + alignment)
- ISA register (id + register class)
- Sub-registers
  - X86: mmx xmm ymm zmm
  - MIPS: 64-bit register for double = a pair of 32-bit registers

Part Ⅰ Abstraction for schedules

- ***Backend***
  - Issue width
  - Pipelines/EUs
  - Instruction latency
  - Register bypassing
    - Cascade FMA
- Frontend
  - ~~Move elimination~~
  - Macro fusion (later)



ARM cortex-a72 pipeline

Part Ⅰ Abstraction for schedules

Schedule class in CMMC

- Instructions → schedule classes

```
// query
uint32_t queryRegisterLatency(const MIRInst& inst, uint32_t idx) const;
[[nodiscard]] bool isPipelineReady(uint32_t pipelineId) const;
[[nodiscard]] bool isAvailable(uint32_t mask) const;
// issue
void setIssued(uint32_t mask);
void resetPipeline(uint32_t pipelineId, uint32_t duration);
void makeRegisterReady(const MIRInst& inst, uint32_t idx, uint32_t latency);

uint32_t nextCycle();
```

bypassing
blocking pipeline
fully pipelined ports

called by top-down scheduler

Part Ⅰ Abstraction for schedules

Schedule class in CMMC

- ARM.MLA/MLS/SMMLA → MultiplyAccumulate

```
class ARMScheduleClassMultiplyAccumulate final : public ScheduleClass {
public:
    bool schedule(ScheduleState& state, const MIRInst& inst, const InstInfo& instInfo) const override {
        CMMC_UNUSED(instInfo);

        if(!state.isPipelineReady(pipelineId: ARMPipelineMultiCycle))
            return false;

        if(state.queryRegisterLatency(inst, idx: 1) > 0 || state.queryRegisterLatency(inst, idx: 2) > 0)
            return false;
        if(state.queryRegisterLatency(inst, idx: 3) > 3)
            return false;

        state.resetPipeline(pipelineId: ARMPipelineMultiCycle, duration: 1);
        state.makeRegisterReady(inst, idx: 0, latency: 4);
        return true;
    }
};
```

is pipeline available?

are operands ready?

issue!

Part Ⅰ Abstraction for macro fusions

- XiangShan NanHu:
  - large imm: lui + addi
  - zext.w : srli (slli x, 32), 32
  - indexed load: lw (base + offset << {1, 2, 3})
- SiFive U74:
  - Short forward branch optimization

```
beq a1, zero, next
mv a2, a3
next:
```

→ a2 := (a1 == 0 ? a2 : a3)

- Cortex-a72:
  - large imm: movw + movt

The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V

Christopher Celio, Palmer Dabbelt, David Patterson, Krste Asanović
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
celio@eecs.berkeley.edu

https://arxiv.org/pdf/1607.02318.pdf

Part Ⅰ Abstraction for macro fusions

- LLVM: post-processing for Selection DAG
- CMMC: pseudo instructions + expansion
  - pre-RA expansion (SFB on sifive-u74, no schedule class)

```
# Rd = cond ? Rs1 op Rs2 : Rs1
Select_GPR_Arith:
  Format: "select_gpr_arith $Rd:GPR[Def], $Rs1:GPR[Use], $Rs2:GPR[Use], $Lhs:GPR[Use], $Rhs:GPR[Use], $CC:CC[Metadata], $Op:Imm[Metadata]"
  Flag: [LegalizePreRA]
```

    expanded in RISCVISelInfo::preRALegalizeInst

  - post expansion (movt + movw pairs on cortex-a72, with schedule class)

```
MOVT_MOVW_PAIR:
  Format: movt_movw_pair $Rd:GPR[Def], $Imm:UImm32OrReloc[Metadata]
  Flag: [LoadConstant]
```

    expanded in ARMISelInfo::postLegalizeInstSeq

Part Ⅱ Canonicalization

We need canonicalization

- Easy to code/test/debug
  - `icmp ne i32 0, %a` → `icmp ne i32 %a, 0`
  - `icmp uge i32 %a, 2` → `icmp ugt i32 %a, 1`
- Expose hidden optimization opportunities
  - `icmp ne i32 %a, 0` == `icmp ugt i32 %a, 0`
- Reuse logic to handle equivalent representations

non-canonical form

```
while (i < 10) {
    // body
    ++i;
}
```

canonical form

```
if (i < 10) {
    do {
        // body
        ++i;
    } while (i < 10);
}
```

Part Ⅱ Canonicalization

- Canonicalization for `icmp + select` → minmax
- Canonicalization for `sext/zext + and` → select
- Canonicalization for extension of non-negative values
- Canonicalization and inverse transform
- Further reading: https://sunfishcode.github.io/blog/2018/10/22/Canonicalization.html

Part Ⅱ Canonicalization for minmax

`select (a s< b), a, b` → `smin(a, b)`

- Reveal more facts
  - smin(a, b) s>= c → a s>= c && b s>= c
  - ⚙ D155412 [ConstraintElim] Add facts implied by MinMaxIntrinsic (llvm.org)
- Simpler patterns
  - Case1:
    - Before: c = (b < a ? b : a), (a < c ? a : c) → a < b ? a : b
    - After: smin(a, smin(b, a)) → smin(a, b)
  - Case2:
    - Before: (a < 5 ? a : 5) < 2 → a < 2
    - After: smin(a, 5) < 2 → a < 2
    - ⚙ D156238 [InstCombine] Generalize foldICmpWithMinMax (llvm.org)

Part Ⅱ Canonicalization for select

- `and(zext(i1 x), y)` → `select x, y & 1, 0`
- `and(sext(i1 x), y)` → `select x, y, 0`

It looks worse for backend. (later)

Example: `x & !x` = `and (zext (x == 0), x)` → select x == 0, x & 1, 0 → 0

Reuse logic in InstCombineSelect!

[InstCombine] Canonicalize `and(zext(A), B)` into `select A, B & 1, 0` by dtcxzyw · Pull Request #66740 · llvm/llvm-project (github.com)

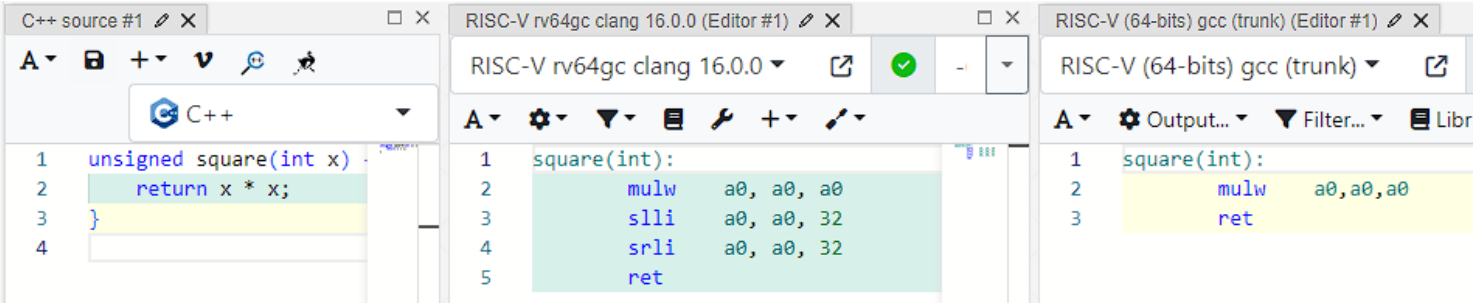
~~More aggressive:~~

- ~~and a, b → bitsel a, b, 0~~
- ~~or a, b → bitsel a, -1, b~~

Part Ⅱ Canonicalization for extension

Extends a non-negative i32 to i64:

- ZExt? SExt? SExt is free on riscv64!
- Example:



- https://godbolt.org/z/v88WsrfG6
- [DAGCombiner][RISCV] Prefer to sext i32 non-negative values by dtcxzyw · Pull Request #65984 · llvm/llvm-project (github.com)
- ~~More aggressive: zext → sext in mid-end (~0.1% improvement)~~

Part Ⅱ Canonicalization & inverse transform

- Problem: Canonicalization may cause regressions.
- Solution:
  - Step 1: Canonicalization in mid-end (“mostly” target-independent, maximize matches)
  - Step 2: Inverse transform them in backend (target-dependent, maximize performance).

- Example1: `select x, y & 1, 0` → `and(zext(x), y)` in DAGCombiner
- goldstein/select and zext by goldsteinn · Pull Request #66793 · llvm/llvm-project (github.com)

- Example2: `br on x < c1` → `br on x – c1 < 0` in CodeGenPrepare
- ⚙ D147789 [CodeGenPrepare][RISCV] Reverse transform in CGP to use zero-compare branch (llvm.org)

Part Ⅲ Legalization

We need legalization

- Generate legal instructions
  - Legalize types (store an i64 into on rv32)
  - Expand `li $x, non-simm12` to `lui + addi` on risc-v
  - Expand `load_i32 $x, 2048($p)` to `addi + lw` on risc-v
- Legalization = partial lowering (MLIR)
- Minimize information loss (later)

Part Ⅲ Legalization in CMMC backend

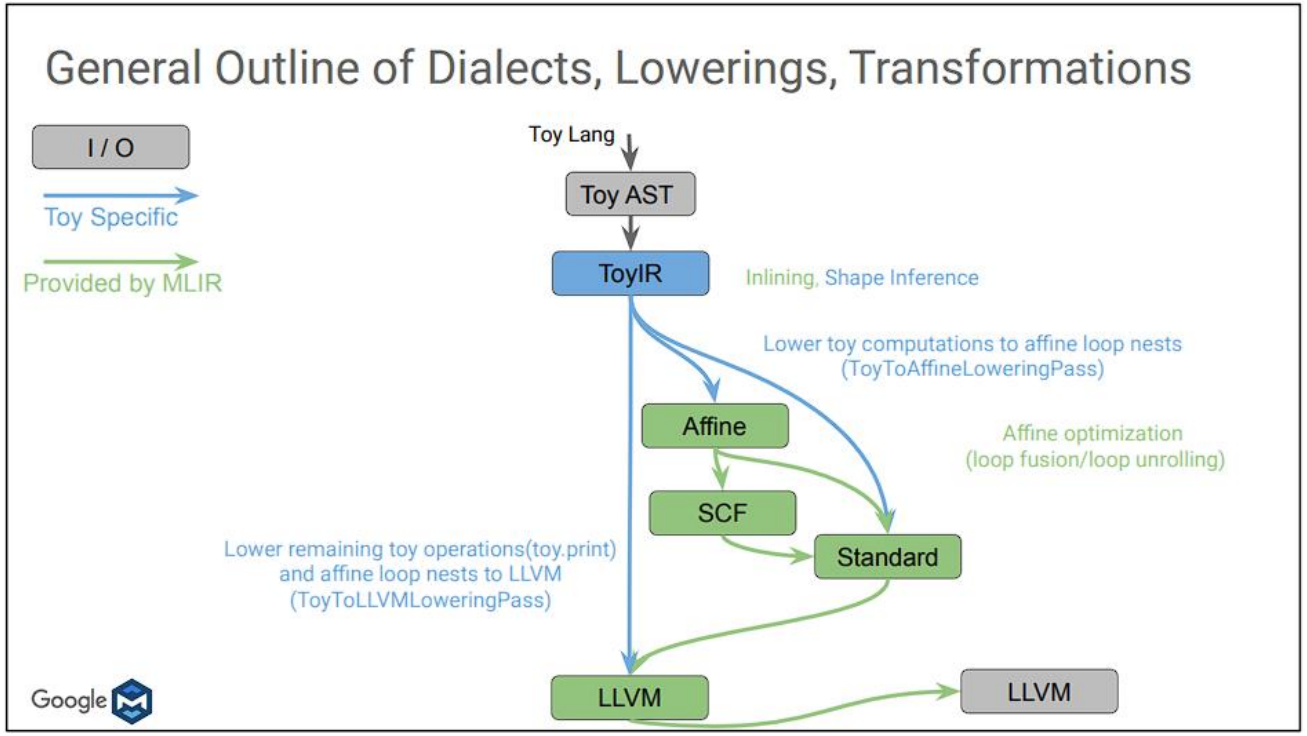
Stage/Assumptions	Generic insts	SSA form	Virtual regs	Unique terminator
Instruction selection	T	T	T	T
Pre-RA expansion	F	T	T	T
Register allocation	F	F	T	T
Simplify CFG	F	F	F	T
Post expansion	F	F	F	F

Less assumptions mean:

- information loss
- more optimization opportunities

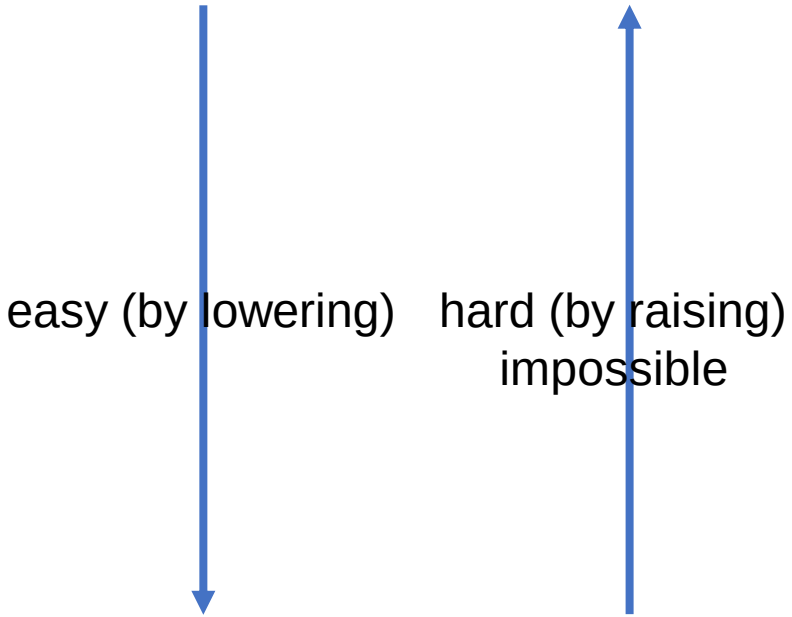
Part Ⅲ Legalization = partial lowering

- legalization = mark something as legal/illegal + iterative rewriting = partial lowering
- end to end → progressive
  - lower info loss per step
- MLIR (Multi-Level Intermediate Representation)
- [Source: https://mlir.llvm.org/talks/](https://mlir.llvm.org/talks/)



Part Ⅲ Legalization and information loss

- Level 2: DSL compiler/DL compiler
  - Op fusion
- Level 2: frontend (clang++)
  - RVO (Return-value optimization)
  - EBO (Empty base optimization)
  - devirtualization
- Level 1: mid-end
  - TBAA
  - libcall to libc (routines for memcpy/memset/strlen)
- Level 0: backend

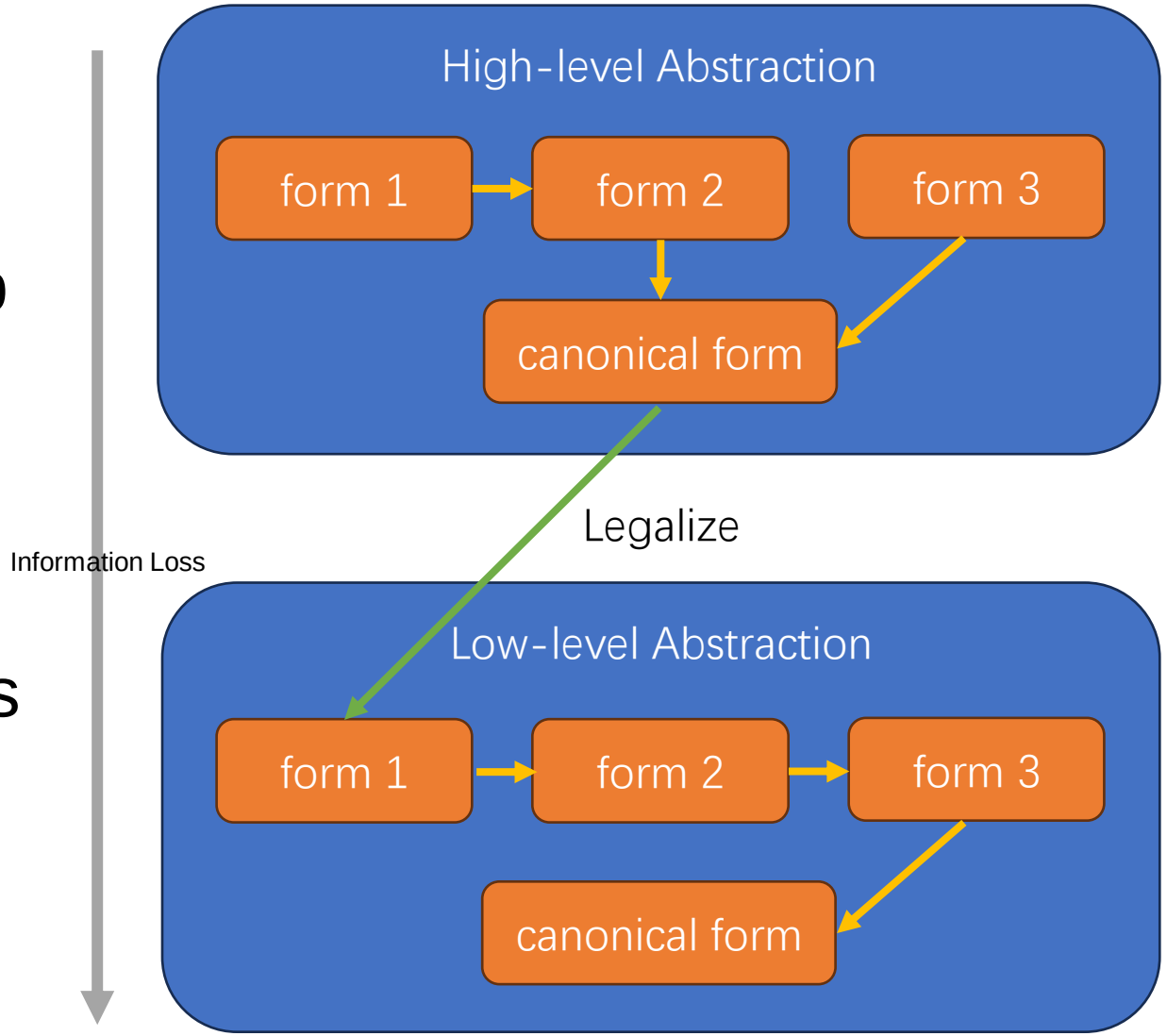


- Legalization is an irreversible process → “entropy increase”
  → information loss

Summary

Three easy pieces:

- ***Abstraction*** : exploit/keep information as much as possible
- ***Canonicalization*** : faster & easier to rewrite program into optimal forms
- ***Legalization*** : reduce information loss during rewriting



Future of compilers

MLIR: A Compiler Infrastructure for the End of Moore's Law

Chris Lattner * Google
Mehdi Amini Google
Uday Bondhugula IISc
Albert Cohen Google
Andy Davis Google
Jacques Pienaar Google
River Riddle Google
Tatiana Shpeisman Google
Nicolas Vasilache Google
Oleksandr Zinenko Google

- Domain-specific compilers
  - DSA needs DSL compilers!
  - AI / Scientific computing
- Compiler for security
  - Sanitizers/CFI
  - Mitigations for side-channel attacks (meltdown/spectre/downfall)
- Compiler for applications on WSC
  - AutoFDO (google)
  - BOLT (meta)

Compiler beginner’s guide

- Tests first (TDD)
  - Unit testing
  - Fuzzing (csmith)
  - Regression testing
  - Differential testing (longfruit for rv64gc)

- Develop compiler from scratch
  - Why we need these optimizations?

Q&A