TransISA: A Static Transpiler for Migrating Legacy x86 Assembly to ARM

Improving Scientific Software Conference 2026

Felix Hirwa Nshuti

NCAR, Boulder, CO

April 9, 2026

`github.com/fnhirwa/TransISA`

Outline

The Portability Wall

The Shift

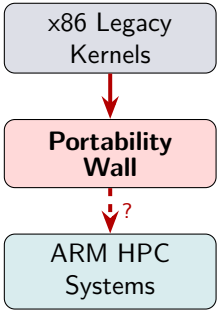
HPC is moving toward energy-efficient ARM:

- Fugaku (A64FX), #1 supercomputer (2020–22)
- AWS Graviton, Ampere Altra in cloud HPC
- Apple Silicon dominating desktop/laptop

The Problem

Legacy scientific kernels **hand-optimized in x86 assembly** are incompatible with ARM.

Manual rewriting is **error-prone**, **costly**, and **does not scale**.



Existing Approaches & Their Gaps

Dynamic Binary Translation

Rosetta 2, QEMU

- ✓ Works on binaries directly
- × Black-box, no inspection
- × Runtime overhead
- × Platform-locked (Rosetta 2)

LLM-Based Transpilation

CRT (Ahmed et al.)

- ✓ Up to 79% accuracy on ARM
- × Probabilistic, no guarantees
- × Requires A100 GPUs
- × No intermediate verification

Manual Rewriting

- ✓ Full control
- × Months of expert effort
- × Error-prone at scale
- × Does not scale

What's Missing?

A **static**, **transparent**, **inspectable** tool:

- Translates assembly *source* (not binaries)
- Verifiable intermediate stages
- Integrates with compiler infrastructure
- Modular can be etended to new instructions/optimizations

TransISA: Design Philosophy

Core Principles

1. **Static translation:** ahead-of-time, deterministic
2. **Source-level input:** operates on `.s` files, not binaries
3. **Full transparency:** every stage is inspectable
4. **Compiler architecture:** follows the classic frontend $\rightarrow$ middle-end $\rightarrow$ backend pattern
5. **LLVM-powered:** leverages mature optimization and code generation infrastructure

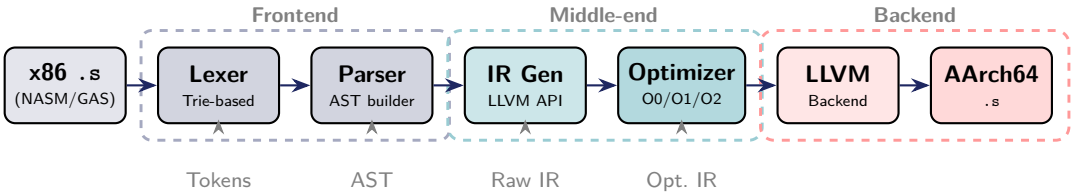
Key Advantage

Developers can **inspect**, **verify**, and **tune** every intermediate representation:

- Token stream
- Abstract Syntax Tree
- LLVM IR (pre/post optimization)
- Final ARM assembly

Implementation: $\sim$4,300 lines of C++17

Pipeline Architecture



Each dashed output is accessible via CLI flags (`--verbose`, `--emit-ir`)

Frontend: Lexer & Parser

Trie-Based Lexer

Classifies tokens via a trie over instruction mnemonics, register names, and directives.

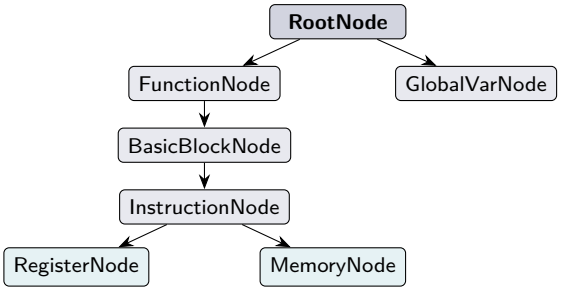
Handles: decimal/hex/binary/octal immediates, string literals, memory operands (`[rbp-8]`), labels.

Supported instructions:

Category	Instructions
Data movement	`mov`, `lea`, `push`, `pop`
Arithmetic	`add`, `sub`, `inc`, `dec`, `neg`
Multiply/Divide	`imul`, `idiv`, `cdq/cqo`
Bitwise	`and`, `or`, `xor`, `shl`, `shr`
Compare	`cmp`, `test`
Control flow	`jmp`, `je/jne`, `jg/jl`, ...
System	`syscall`, `call`, `ret`

Parser & AST

Produces a hierarchical AST from the token stream:



Sections: `.text`, `.data`, `.bss` parsed independently

IR Generation: Register Model & Flag Simulation

x86 Register → LLVM IR

Each x86 register → `alloca` in the entry block.

Aliases resolved: `rax`→`eax`, `rdi`→`edi`.

LLVMIRGenerator.cpp

```
// Alloca in entry block for mem2reg
IRBuilder<> entryB(
    &func->getEntryBlock(),
    func->getEntryBlock()
        .getFirstInsertionPt());
regPtr = entryB.CreateAlloca(
    Type::getInt32Ty(ctx),
    nullptr, resolved);
```

Key: Entry-block allocas are promotable by `mem2reg` at O1+, eliminating load/store overhead.

CPU Flag Simulation

x86 flags modeled as explicit values:

LLVMIRGenerator.h

```
struct FlagState {
  Value* zeroFlag;      // ZF
  Value* signFlag;      // SF
  Value* carryFlag;     // CF
  Value* overflowFlag; // OF
};
```

Syscall Translation

Platform-aware inline assembly:

`x86 syscall` → ARM `svc #0x80`

macOS: `x16 = num | 0x2000000`

Walkthrough: x86 Input (GCD (Euclidean Algorithm))

benchmarking/x86/gcd.s; Computes GCD(48, 18) = 6

```
1  _start:
2      movq $48, %rax          # a = 48
3      movq $18, %rbx          # b = 18
4  gcd_loop:
5      cmpq %rbx, %rax         # compare a and b
6      je gcd_done             # if a == b, done
7      cmpq %rbx, %rax         # a < b?
8      jl b_greater
9      subq %rbx, %rax         # a > b: a = a - b
10     jmp gcd_loop
11 b_greater:
12     subq %rax, %rbx         # b > a: b = b - a
13     jmp gcd_loop
14 gcd_done:
15     movq %rax, %rbx         # save result
16     movq $0x2000001, %rax   # syscall: exit
17     movq %rbx, %rdi         # status = 6
18     syscall
```

Tests: cmp, je, jl, sub, jmp, data-dependent branching, classic numeric kernel.

Walkthrough: LLVM IR at O0; GCD Register Simulation

Entry block: registers → allocas

```
define void @_start() {
entry:
  ; register simulation
  %eax = alloca i32, align 4
  %ebx = alloca i32, align 4
  %edi = alloca i32, align 4
  %edx = alloca i32, align 4
  ; push/pop stack buffer
  %stackmem__start =
      alloca [1024 x i32], align 4  ; 4 KB
  %rsp__start = alloca i32, align 4
  store i32 1024, ptr %rsp__start
  %stackmem_ptr__start =
      alloca [1024 x ptr], align 8  ; 8 KB
  ; ... 2 more shadow tracking vars
  store i32 48, ptr %eax  ; movq $48,%rax
  store i32 18, ptr %ebx  ; movq $18,%rbx
  br label %gcd_loop
```

Loop body: load every use, store every write

```
gcd_loop:
  %eax_load = load i32, ptr %eax
  %ebx_load = load i32, ptr %ebx
  %cmp_tmp = sub i32 %eax_load,
                    %ebx_load
  %zeroFlag = icmp eq i32 %cmp_tmp, 0
  %signFlag = icmp slt i32 %cmp_tmp, 0
  %carryFlag = icmp ult i32
                %eax_load, %ebx_load
  ; ... 4 more flag computations (OF)
  br i1 %zeroFlag, label %gcd_done,
                    label %
                        branch_fallback
  ; ... branch_fallback, b_greater,
  ;     gcd_done blocks
```

Walkthrough: LLVM IR at O0 (cont.)

Structural Cost Summary

Allocas	10 (4 regs + 6 stack simulation)
IR blocks	5 (entry, loop, 2 fallback, done)
Loads	2 per loop iteration
Stores	1 per register write
Flag ops	7 per `cmp`

Key Insight

Every register access requires a load or store in O0. `mem2reg` at O1 removes this register-stack overhead by promoting allocas to SSA values.

Walkthrough: LLVM IR at O2; Allocas Eliminated

O2 IR: entire GCD loop collapses to **one block**

```
define void @_start() local_unnamed_addr {
entry:
  br label %branch_fallback

gcd_done:
  ; constants folded; no register loads
  %addtmp = add i32 %spec.select, 48
  store i32 %addtmp, ptr @result
  %r = tail call i64 asm sideeffect
    "...svc #0x80...", "=r,r,r,r"
    (i64 33554436, i64 1, ...)
  ret void
```

Key: O2 has no register allocas and no stack-resident register model.

Walkthrough: LLVM IR at O2 (cont.)

phi + **select** encode loop state and branch logic

```
branch_fallback:
  %ebx.043 = phi i32 [18, %entry],
                     [%spec.select41, %
                         branch_fallback]
  %eax.042 = phi i32 [48, %entry],
                     [%spec.select, %
                         branch_fallback]
  %cmp_tmp = sub i32 %eax.042, %ebx.043
  %jl_cond = icmp slt i32 %cmp_tmp, 0
  %spec.select = select i1 %jl_cond,
                 i32 %eax.042, i32 %cmp_tmp
  %spec.select41 = sub i32 %ebx.043, ...
  %zeroFlag = icmp eq i32 %spec.select,
                          %spec.select41
  br i1 %zeroFlag, label %gcd_done,
                   label %branch_fallback
}
```

Insights

- `phi` nodes carry loop-carried values (old eax/ebx to next iteration).
- `select` replaces one control-flow split with data-flow.
- Fewer branches improves instruction scheduling and backend code quality.
- This is the core reason O2 drops from heavy stack traffic to register-resident code.

Walkthrough: AArch64 Assembly at O0; Stack-Spill Overhead

Prologue (1/2): stack frame setup

```
1  __start:
2    stp x20,x19, [sp,#-32]!
3    stp x29,x30, [sp,#16]
4    add x29, sp, #16
5    sub sp, sp, #3, lsl #12
6    sub sp, sp, #48
7    mov x19, sp
8    ; init stack simulation
9    mov w8, #1024
10   str wzr,  [x19,#12]
11   stp w8, wzr, [x19,#16]
```

Prologue (2/2): initial register state

```
1    str w8,  [x19,#8220]
2    mov w8, #48
3    stur w8, [x29,#-32]
4    mov w8, #18
5    b   LBB0_2
6  LBB0_1:
7    ldp w9,w8, [x29,#-32]
8    sub w8, w8, w9
9  LBB0_2:
10   stur w8, [x29,#-28]
```

Key: O0 establishes stack-backed register state before entering the loop.

Walkthrough: AArch64 Assembly at O0 (cont.)

Loop: reload, flag compute, branch-heavy path

```
LBB0_3:
  ldp w8,w9, [x29,#-32]
  cmp w8, w9
  b.eq LBB0_6
  ldur w9, [x29,#-28]
  sub  w10, w8, w9
  cmp  w10, #0
  cset w10, lt
  cmp  w8, #0
  cset w8, lt
  cmp  w9, #0
  cset w9, lt
  eor  w9, w8, w9
  eor  w8, w10, w8
  and  w8, w9, w8
  cmp  w10, w8
  b.ne LBB0_1
  ldp  w8,w9, [x29,#-32]
  sub  w8, w8, w9
  stur w8, [x29,#-32]
  b    LBB0_3
LBB0_6:
  ; ... convert/write/exit syscalls
```

Insights

- Repeated `ldp/ldur/stur` keeps virtual registers in memory.
- Flag emulation expands one compare into multiple dependent ops.
- O0 loop cost is about 20 instructions per iteration.

Walkthrough: AArch64 Assembly at O2; Register-Resident GCD

O2 loop: register-resident path

```
__start:
  mov w10, #48          ; eax = 48 (register!)
  mov w9,  #18          ; ebx = 18 (register!)
LBB0_1:                 ; entire GCD loop
  sub  w8, w10, w9      ; cmp_tmp = eax - ebx
  eor  w11, w9, w10     ; overflow detection
  bic  w12, w8, w11     ;   via bitmask ops
  and  w11, w10, w11    ;   (no branch needed)
  orr  w11, w11, w12
  and  w12, w10, w11, asr #31
  cmp  w11, #0
  csel w8, w10, w8, lt ; eax=select(jl,eax,sub)
  sub  w9, w9, w12      ; ebx -= (jl ? eax : 0)
  mov  w10, w8
  cmp  w8, w9
  b.ne LBB0_1           ; loop if eax != ebx
```

O2 exit: constants inlined

```
  add  w10, w8, #48     ; ASCII convert
  adrp x11, _result@PAGE
  add  x11, x11, _result@PAGEOFF
  mov  w12, #4          ; 0x2000004 lo
  mov  w14, #1
  mov  x9,  xzr
  movk w12, #512, lsl #16  ; = 0x2000004
  mov  w13, #1
  movk w14, #512, lsl #16  ; = 0x2000001
  str  w10, [x11]       ; store result
; write syscall
  mov x16, x12   ; x16 = 0x2000004
  mov x0, x13    ; fd = 1
  mov x1, x11    ; buf = &result
  mov x2, x13    ; len = 1
  svc #0x80
; exit syscall
  mov x16, x14   ; x16 = 0x2000001
  mov x0, x8     ; status = gcd result
  svc #0x80
  ret
```

Key: O2 keeps loop state in registers and avoids stack spills.

Walkthrough: AArch64 Assembly at O2 (cont.)

O0 vs O2 Summary

Metric	O0	O2
Stack frame	12,336 bytes	0 bytes
Instrs/GCD iteration	∼20	**7**
Total instruction count	83	**43**

O2 removes stack traffic and reduces branch/control overhead in the hot loop.

Optimization Passes

Three Optimization Levels

O0 No optimization raw lifted IR emitted as-is.

Useful for debugging; every x86 register maps 1:1 to a stack slot.

O1 LLVM default O1 pipeline.

mem2reg promotes allocas to SSA → eliminates most load/store overhead.

O2 Full LLVM O2 pipeline.

Dead code elimination, constant folding, loop optimizations, instruction combining.

Codegen.cpp, the optimization entry point

```
void Codegen::optimize(
    Module& module, OptLevel level) {
  if (level == OptLevel::O0) return;

  PassBuilder PB;
  // Register analysis managers
  PB.registerModuleAnalyses(MAM);
  PB.registerFunctionAnalyses(FAM);
  PB.crossRegisterProxies(...);

  auto MPM =
    PB.buildPerModuleDefaultPipeline(
      optLevel);
  MPM.run(module, MAM);
}
```

Benchmark Results Table (1/2): x86 Source vs arm64 Target

Instruction-count comparison across `arch=x86` source and transpiled `arch=arm64` output at O0/O1/O2.

Program	x86 (src)	arm64 O0	arm64 O1/2	PASS
fibonacci	22	91	41	✓
nested_loop	22	92	49	✓
gcd	21	83	43	✓
max_of_three	20	89	29	✓
sum_loop	17	70	38	✓
add	13	48	29	✓
hello	8	41	26	✓

Continued on next slide...

Benchmark Results Table (2/2): x86 Source vs arm64 Target

Program	x86 (src)	arm64 O0	arm64 O1/2	PASS
signed_div	7	52	14	✓
factorial	8	37	14	✓
abs_val	7	28	15	✓
imul3op	5	28	14	✓
swap	6	28	14	✓
bitwise_not	6	26	14	✓
overall mean	12.4	56.4	30.9	

Unified O0 overhead (mean): 4.6× vs x86
Unified O1 reduction (mean): 51% vs O0
Stack at O0: 12 323–12 339 B → 0 at O1/O2

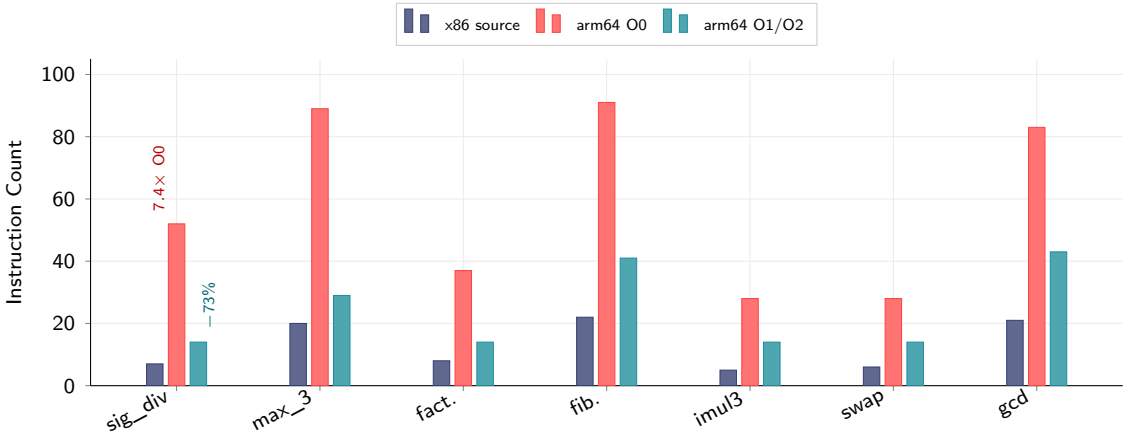
Optimization Takeaway

O1 = O2

for all 13 benchmarks

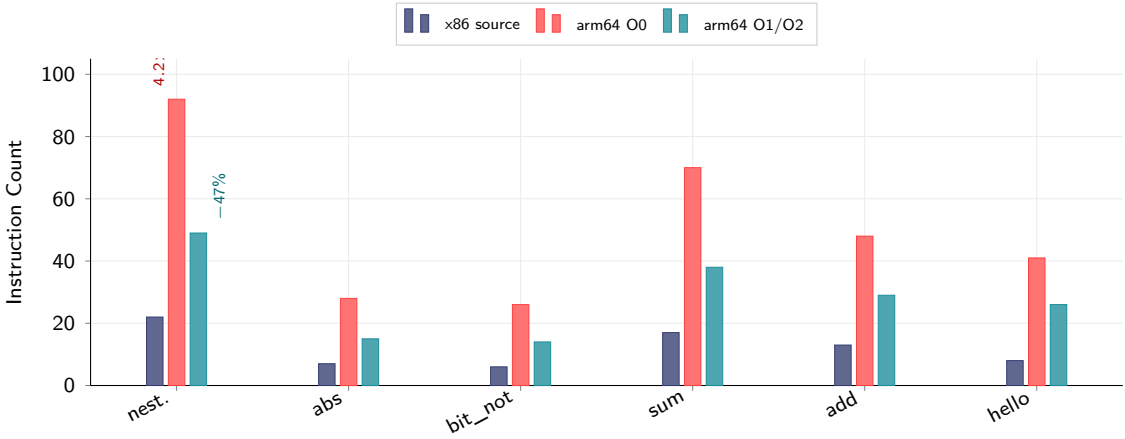
`mem2reg` captures the dominant optimization opportunity.

Instruction Count: x86 Source vs arm64 O0/O1/O2 (1/2)



Left-to-right order still follows O1 reduction (highest first).

Instruction Count: x86 Source vs arm64 O0/O1/O2 (2/2)



Benchmarks are split for readability while preserving the same values. **Mean O0 overhead:** 4.6× **x86.**

Mean O1 reduction: 51% over O0.

Results: Key Insights (1/2)

Finding 1: O1 = O2 Universally

For all 13 benchmarks, O1 and O2 produce **identical instruction counts** and text sizes.

`mem2reg` (active at O1) eliminates all register `allocas` in a single pass. Loop and global value numbering passes added at O2 find no further savings on these programs.

Implication: the dominant optimization is structural (alloca promotion), not arithmetic.

Finding 2: Stack Disappears at O1

O0 stack: **12 323–12 339 bytes** per function (two simulation buffers: `i32[1024]` + `ptr[1024]`).

O1 stack: **0 bytes** for all benchmarks.

LLVM proves the buffers are unused for programs with no `push`/`pop` and eliminates them entirely.

Results: Key Insights (2/2)

Finding 3: Reduction Correlates with IR Complexity

Benchmark	O0	Red.%
signed_div (idiv guard)	52	**73.1%**
max_of_three (3 paths)	89	**67.4%**
factorial (imul loop)	37	**62.2%**
hello (syscall-heavy)	41	36.6%
add (minimal)	48	39.6%

Programs with more IR redundancy (flag computations, guard blocks, multi-path branches) benefit most from optimization. `hello`'s irreducible syscall inline-asm limits its gains.

Finding 4: Binary Size Anomaly

`signed_div` O0 binary: **49 968 B**
vs all others: ∼**16 792 B** at O0.

Cause: the divide-by-zero guard emits a `puts()` error-path that pulls in `libSystem` stubs, inflating the Mach-O binary. Eliminated at O1 via dead-code removal.

Correctness & Testing (1/2)

Correctness Validation

Each benchmark runs natively on Apple Silicon.

Exit code verification:

`fibonacci`	→ exit 55
`max_of_three`	→ exit 42
`gcd`	→ exit 6
`sum_loop`	→ exit 36
`nested_loop`	→ exit 20

All benchmarks produce **identical results** to their x86 originals.

Correctness & Testing (2/2)

Test Infrastructure

- **Google Test** suites for lexer, parser, and IR generation
- **LLVM** `verifyModule` active in CI, catches IR-level bugs early
- **GitHub Actions CI**: macOS + Ubuntu, both x86-64 and ARM64

Automated Benchmarking

`analyzefiles.py`:

- Transpiles all benchmarks at O0/O1/O2
- Compiles, links, executes on ARM
- Collects metrics, outputs CSV + JSON

Translation Analysis: Where the Bloat Comes From (1/2)

O0: Sources of Instruction Bloat

1. Register simulation

Each x86 register → stack slot. Every read is `ldr`, every write is `str`. A single `addq %rax, %rbx` becomes: load, load, add, store.

2. Flag materialization

x86 implicitly sets flags; ARM needs explicit `subs`/`cmp`. Each flag-setting instruction generates additional comparison code.

3. Syscall translation

x86 `syscall` (1 instr) → ARM inline asm block: argument marshalling + `svc` (5–8 instrs).

Translation Analysis: Where the Bloat Comes From (2/2)

O1/O2: What LLVM Recovers

1. `mem2reg` (O1+)

Promotes all register allocas to SSA values. Eliminates *all* load/store pairs for register simulation. Single biggest win.

2. Dead code elimination (O2)

Removes unused flag computations and unreachable blocks from conservative translation.

3. Instruction combining (O2)

Fuses compare-and-branch patterns, simplifies constant expressions, reduces redundant moves.

Residual gap: Syscall overhead is irreducible architectural difference, not translation artifact.

Challenges & Lessons Learned (1/2)

Architectural Challenges

Register model is the root constraint

All registers map to `i32` regardless of actual width. Must be resolved before float support (e.g., `incq %rcx`) incorrectly operates on a 32-bit alloca.

Cross-compilation workflow

Build on x86 Linux/WSL, but benchmarks run natively on macOS ARM. Target triple hardcoded to `aarch64-apple-macosx11.0.0` regardless of host.

Challenges & Lessons Learned (2/2)

Key Lessons

Memory operand parsing

Indirect addressing (`[rbp-8]`) was the original blocker, required proper `MemoryNode` with `isIndirect=true`.

Entry-block allocas matter

Non-entry-block allocas are *not* promoted by `mem2reg`, causing the optimizer to treat loads as UB and eliminate function bodies.

Keep `verifyModule` active

Catches IR bugs (unterminated blocks, type mismatches) before they cascade into cryptic backend errors.

Future Work (1/4) Instruction Coverage

Instruction Coverage

- **Floating-point**: Refactor register model from `i32` to type-parameterized allocation; add `movss`, `addss`, `mulss`, `cvtsi2ss`.
- **Scientific kernels**: dot product, matrix transpose, FFT, the primary HPC migration targets.
- **SPEC CPU**: standard benchmarks for rigorous cross-ISA evaluation.

Future Work (2/4) Modular Target Support

Modular Target Support *(in progress)*

New `--target=` flag; `TargetConfig` struct bundles triple + ABI.

Currently supported: `macos-arm64`, `linux-arm64`, `linux-x86_64`.

Adding RISC-V requires only implementing the `svc` ABI pipeline structure unchanged.

Future Work (3/4) Custom LLVM Passes

Custom LLVM Passes

- **Stack trimming**: analyze actual `push`/`pop` depth and resize the fixed 1024-slot buffer accordingly, eliminates the 12 KB O0 frame for shallow-stack programs.
- **Flag elimination**: detect flag values that are computed but never read; prune the 7-op OF/SF/ZF/CF sequences for those `cmp` sites.
- **x86 idiom recognition**: map common idioms (`xor eax,eax`, `lea rax,[rip+0]`) to idiomatic ARM sequences at the IR level, before the LLVM backend sees them.

Future Work (4/4) Runtime & Formal Methods

Runtime & Formal Methods

- **NEON/SVE**: leverage ARM SIMD for translated SSE/AVX code once the float register model is in place.
- **Dynamic profiling**: cycle-accurate comparison via `perf`/Instruments, current evaluation is static only.
- **Formal equivalence**: translation validation or property-based testing to certify semantic equivalence beyond exit-code checking.

Conclusion

Summary

TransISA demonstrates that a **static, source-level transpiler** built on traditional compiler principles can provide a viable pathway for migrating legacy x86 assembly to ARM.

Key Contributions

1. Full pipeline from x86 source to AArch64 assembly via LLVM IR
2. Configurable optimization (O0/O1/O2) with inspectable intermediate stages
3. Open-source, extensible architecture for research and legacy migration

Takeaway

By targeting assembly *source* rather than binaries, TransISA gives developers full control:

Inspect $\rightarrow$ **Verify** $\rightarrow$ **Tune**

A sustainable tool for long-term software maintenance in the ARM era.

Questions?

Felix Hirwa Nshuti

`github.com/fnhirwa/TransISA`

Paper	IEEE ISS26 Proceedings (Upcoming)
Code	`github.com/fnhirwa/TransISA`
License	MIT