PCFC11: a credible path to a modern C11 compiler

Assessment of 0.2.13, its current architecture, and the upgrades with the strongest practical return. Prepared for CC.

PCFC11 can become a respected compiler for a clearly defined platform and workload range. The immediate priorities are a dependable language contract, broader independent validation, a reusable optimizer representation, and proven numerical-library interoperability.

Its present evidence is meaningful, but it does not establish general C11 conformance or high-performance matrix code generation. A small compiler can earn trust through predictable behavior, useful diagnostics, reproducible results and maintained interoperability. It does not need every GCC or Clang feature before it becomes useful.

Evidence reviewed	What it supports
Native Windows 0.2.13 [E1]	24/24 phases pass; 149 language cases pass with two documented skips; all 49 bundled PyCForge C fixtures pass.
Source-package Linux evidence [E2]	25/25 feedback phases pass; separate bootstrap, reference-oracle and compiler-sanitizer gate reports pass. These are archived results reviewed here, not a fresh full rerun.
Current optimizer [E2]	Typed AST transformations, bounded integer helper expansion, constant flow, liveness/dead stores, four dedicated integer registers and direct expression operands.
Limits [E1, E2]	No general register allocator, loop transformations or SIMD vectorizer; incomplete hosted-library/floating-point audit; live conversion and application validation remain separate.

Recommended investment order

First: consolidate the new Windows evidence; improve diagnostics and the C11/ABI contract. **Next:** introduce a small typed intermediate representation and general register allocation. **In parallel as project work:** qualify CBLAS/LAPACKE interoperability. **Then:** range analysis, loop transformations and SIMD. Profile-guided and link-time optimization follow a stable middle end.

Scope: source and evidence inspection, assembly review and primary-source research. The compiler was rebuilt successfully with the local GCC during review. No new matrix benchmark, full regression rerun, production certification or compiler upgrade is claimed. The source package still says Windows 0.2.13 is pending; the supplied results supersede that status.

What skeptical engineers should be able to verify

A precise C11 contract

Publish a requirement-by-requirement ledger covering mandatory diagnostics, declarations and compatible types, integer constant expressions, conversions, initialization, preprocessing phases, implementation-defined choices and the compiler/header/runtime combination. Keep positive execution tests, required rejection tests and accepted extensions distinct. A passing test count is not a conformance percentage.

C11 makes VLAs, complex arithmetic, atomics and the threads library conditional facilities. Extended alignment is implementation-defined. Therefore the two Windows skips are documented capability limits, not automatically two mandatory-C11 failures. Windows thread-local object emission needs its own treatment: omitting the threads library does not by itself resolve the core storage-duration implementation. [R1]

PCFC11 currently advertises no complex support, and on Windows no VLAs or threads library. Its atomics are a narrower supported profile: naturally aligned integer/pointer objects through eight bytes, using conservative ordering. Audit the claimed atomic types, operations, memory-order constraints and header contract before broadening any conformance claim. [E2]

Diagnostics that mean what the command line says

Concrete issue: main.c:371-380 accepts -Wall and -Wextra without enabling warning groups. Implement useful warning categories, or explicitly report their unsupported status. Add uninitialized-use, incompatible-call, suspicious-conversion and format diagnostics incrementally; avoid presenting warning-free compilation as analysis that never ran. Retain source ranges, macro-expansion context, stable diagnostic identifiers and JSON output. [E2]

Evidence that survives independent examination

Gate	Evidence to retain
Correctness	Native Linux and Windows execution, GCC and Clang comparisons on defined-behavior cases, minimized reproducers, expected diagnostics, ABI checks in both call directions.
Stress and fuzzing	Seeded Csmith/YARPGen campaigns; generated inputs spanning optimizer interactions; failure reduction; compiler ASan/UBSan; time/memory budgets for malformed input. [R6, R7]
Real software	Build and execute selected unmodified third-party C libraries/applications; document required extensions. Retain the 49 fixtures and add larger live PyCForge conversions and application workflows.
Release provenance	Source/compiler hashes, exact commands, dependencies and target profiles; reproducible packages where feasible; published known issues and a repeatable bug-report path.

Self-compilation is useful evidence of exercised language coverage, not a proof of compiler correctness. Compiler sanitizers test the compiler process; they do not mean PCFC11-generated applications have sanitizer instrumentation. Existing archived sanitizer passes should be preserved, then rerun for the release being qualified. [E2, R13]

The design change that unlocks larger gains

The current conservative design is a useful baseline: it declines transformations when effects, aliasing or budgets are unclear. Its limitation is that reusable global facts are scarce. A temporary CFG exists for liveness, but optimization still transforms the typed AST; register placement assigns each selected integer object one of R12-R15 for its entire invocation. Floating-point and pointer objects remain outside that placement pass. [E2]

Recommended architecture: retain the C implementation and independent backend; add a compact typed, three-address intermediate representation with explicit control flow and memory effects. SSA is a useful design choice, not a requirement to adopt LLVM.

Layer	Required responsibilities
Frontend and lowering	Finish semantic diagnostics before optimization. Lower valid typed AST operations into basic blocks, explicit conversions, loads, stores, calls and branches. Preserve chosen sequencing and C-visible effects.
Reusable analyses	Dominators, loop structure, def-use chains, scalar SSA, value ranges/known bits, escape/alias facts and call memory effects. Keep volatile, atomic and floating-environment rules explicit.
Machine representation	Virtual registers, target instruction constraints, call-clobber sets, separate integer/FP/vector register classes, instruction selection, spills and stack-frame layout.
Verification and debugging	IR verifier after every pass; stable dumps and pass-disable/bisect controls; source locations through lowering; bounded analysis work and safe fallback for unsupported functions.

Migrate in small, testable steps

First lower a restricted set of ordinary integer functions without optimizing them and compare execution with the existing emitter. Add scalar promotion, copy propagation and simple constant propagation. Expand control-flow coverage before memory transformations. Preserve a whole-function fallback while migration is incomplete; do not partially mix incompatible frame or register assumptions.

Replace fixed register assignment with live-range allocation

A bounded linear-scan allocator is a practical first implementation. Add live intervals, register reuse, call-aware splitting, coalescing, spill/reload placement and rematerialization. Include pointer and floating-point values; later include vectors. Shrink frames only when address exposure, alignment, stack probing and unwind metadata remain correct. Judge the result on spills executed, code size, runtime and compilation cost, including short functions.

SSA/value analyses and loop optimizations are established approaches, but their exact scope and cost limits here are recommendations for PCFC11. A future memory-dependence representation must complement alias analysis; it cannot override volatile/atomic semantics. [R2, R3]

A practical optimization sequence

Priority	Upgrade	Practical return / dependency
1	Copy propagation, scalar promotion and live-range allocation	Remove helper temporaries and repeated frame traffic. Requires explicit values and control flow; preserve conversions and alias-visible storage.
1	Value ranges, known bits and branch simplification	Prove nonnegative values, impossible correction paths and bounded induction. Targets the current division workload without recognizing fixture names.
2	Global value numbering and memory-aware dead stores	Reuse repeated computations and proven redundant loads/stores. Requires dominance, escape/alias facts and call effects.
2	Loop-invariant motion and induction simplification	Move proven invariant work and simplify address calculations. Respect zero-trip loops, possible traps, writes and observable FP effects.
2	Broader inlining and scalar replacement of aggregates	Expose small helper/record computations across calls. Use code-size budgets; preserve object identity, layout and address escape rules.
3	Loop transforms and SIMD	Add selective unrolling, interchange, tiling, SLP and loop vectorization with legality checks, remainder handling and a target cost model.
4	PGO, LTO and target scheduling	Use actual hotness and cross-file knowledge after IR, profiling formats, linking contracts and diagnostics are stable. [R5]

The division benchmark identifies the next mechanism

The Windows run measures about 7.73 times GCC -O2 runtime for the PyCForge division kernel. Its O1 work loop has **no hardware divide and no helper call**, but retains temporary memory traffic, sign adjustments and correction/loop branches. GCC emits a much smaller induction-based loop. This points toward range analysis and temporary elimination; assembly alone does not isolate their timing contributions. [E1]

Additional constant-divisor strength reduction, including multiply-high recipes, remains useful later. Preserve signed rounding and boundary behavior. Remove overflow/error paths only with the necessary range proof; broaden div/mod reuse only for proven equal operands and compatible control-flow conditions.

Maintain optimization accountability

State pass preconditions and missed-optimization reasons; retain reduced reproducers. Test pass interactions and verify IR. Bounded SMT validation can strengthen selected integer rewrites; LLVM-oriented Alive2 is not a drop-in verifier for PCFC11's AST. [R8]

Yes, through two distinct capability paths

Near-term high-demand matrices: compile the application and call a validated optimized numerical library. Longer-term: generate competitive matrix kernels directly, after floating-point allocation, dependence analysis and SIMD are established.

Path A: qualify numerical-library interoperability first

Use the CBLAS interface to a selected OpenBLAS or BLIS build for dense BLAS operations, and LAPACKE for factorizations and solvers. This keeps PCFC11 responsible for reliable surrounding C code and ABI calls while specialized kernels perform the expensive work. OpenBLAS supplies optimized BLAS routines and optional runtime CPU dispatch; BLIS explicitly structures its implementation around optimized kernels. [R9, R10]

Validate actual public headers, import/static libraries, symbol names, calling conventions, float/double arguments, stack alignment and error paths on each host. CBLAS/LAPACKE integration was **not** established by the supplied 49 PyCForge fixtures. Being able to call a prebuilt library also does not establish that PCFC11 can compile that library from source.

Match the library's integer ABI: BLAS LP64 commonly uses 32-bit dimensions, ILP64 uses 64-bit dimensions. These labels do not change Windows' LLP64 C data model. Use the selected library's declared integer types and symbols, check dimension/byte-count overflow, and test row/column layout, leading dimensions and transposition. LAPACKE documents both layouts and notes possible row-major conversion costs. [R9, R11]

Path B: improve native numerical code generation

Workload	What PCFC11 would need
Small fixed / batched matrices	FP registers, inlining, scalar replacement, SLP and controlled unrolling. Measure call/packing overhead; a library call need not win for tiny kernels.
Large dense GEMM	Cache blocking, packing strategy, vector loads/stores, multiple accumulators, ISA-aware FMA policy and register-pressure control. Threaded libraries are the practical first route.
GEMV / sparse matrices	Memory-bandwidth and irregular-access measurements, layout/stride work, pointer analysis and controlled threading. Wider vectors alone are insufficient.
Complex / mixed precision / GPU	Document complex-type limits; validate a compatible library representation where available. GPU libraries are a separate C-ABI/runtime integration project, not automatic C11 offload. [R11, R14]

Prefer heap allocation for large matrices. Windows' missing VLA support does not prevent pointer-based matrices. SIMD can start with safe unaligned operations; extended local alignment is useful but is not an absolute prerequisite. Keep numerical dependencies optional and explicitly installed/configured to preserve the compact compiler distribution.

Numerical correctness before speed

Three policies to design and test

Policy	Meaning
Default semantics	Preserve the documented C/target FP behavior. No blanket reassociation, reciprocal approximation or silent flush-to-zero. Define FMA contraction and supported rounding/exception behavior.
Explicit fast numeric mode	Opt-in, documented permissions for contraction/reassociation and other approximations. Make each permission inspectable; a speed setting must not silently alter expected error behavior.
Reproducibility policy	Specify reduction order, thread count, dispatch and library settings for the supported configuration. Bitwise reproducibility across all CPUs, libraries and thread schedules is a separate substantial goal.

Vector reductions can reorder additions; floating-point addition is not associative. FMA also changes rounding relative to separate multiply and add. Preserve strict paths where required and expose relaxed choices explicitly. LLVM's vectorizer and constrained-FP documentation illustrate these legality boundaries. [R4, R12]

Finish the current fenv/rounding/conversion audit before making scientific-accuracy claims. Test NaNs, infinities, signed zero, subnormals, cancellation and supported exception modes. Windows uses binary64 long double in PCFC11; Linux uses x87 extended precision. Cross-compiler objects and runtime formatting must agree with the chosen profile. [E2]

A useful matrix qualification suite

Functional coverage: float/double dot products, AXPY, GEMV and GEMM; square and rectangular shapes; tiny, cache-resident and large data; transpose variants; padded strides; odd dimensions and SIMD tails; aligned/unaligned inputs; legal alias cases; scalar alpha/beta edge cases. Add LU/QR or linear-system residual checks for the solver path.

Numerical oracles: use a scalar reference or higher-precision reference where justified, with operation- and conditioning-aware tolerances. Require exact integer results, appropriate floating-error measures and solver residual/backward-error checks. Agreement with one other compiler is not a universal numerical-accuracy proof. Record the library, precision and FP policy alongside every result.

Separate three performance questions

Compiler kernels: PCFC11 versus GCC/Clang compiling the same plain C and FP policy. **Library integration:** both compilers call the same optimized BLAS build with matching settings; this measures integration overhead, not PCFC11's own GEMM code generation. **End-to-end:** include allocation, layout conversion, packing/transfers and application work where users actually pay those costs.

Use warmups, alternating repeated samples, explicit thread counts, fixed inputs/checksums and host/load records. Retain medians and dispersion. For large GEMM report elapsed time and approximately 2mnk/t operations per second; also report memory and numerical error. Split O0/O1 and O1/GCC comparisons into separately calibrated pairs if the slow O0 baseline prevents useful timing duration. Never compare raw durations from unequal work.

Make each milestone independently reviewable

Milestone	Acceptance evidence	Relative effort
A. Reliable baseline	Integrate the supplied Windows status; strict/extension diagnostics and warning-flag behavior; expanded C11/ABI ledger; native release gates and reproducible bug reports.	Medium; immediate
B. Numerical interoperability	Real CBLAS/LAPACKE headers and libraries work on Linux/Windows; layouts, dimensions and precision checked; same-library comparisons; thread control and deployment documented.	Medium; early
C. Reusable optimizer	Typed IR and verifier; scalar SSA, copy/range passes, live-range allocation; existing workloads preserve behavior; expanded independent differential tests.	High; foundational
D. Native numerical kernels	FP allocation, alias/dependence analysis, loop transforms, SIMD and ISA fallback; measured matrix suite with documented accuracy/performance tradeoffs.	Very high
E. Broader production tooling	PGO/LTO, application instrumentation, stronger platform coverage and live PyCForge integration after their prerequisite interfaces stabilize.	High to very high

Other marks of a modern compiler

Debugging: extend existing file/line emission to types, scopes, variables and optimized location tracking; verify debugger workflows on each platform. Native Windows unwind checks already exist; expand large-frame/guard-page and prologue/epilogue coverage. A usable -g experience matters before exotic optimization.

Build/toolchain behavior: stable command-line/process APIs, dependency files and response-file semantics, shared/static library contracts, deterministic diagnostics, installation discovery and regression tests for failed assembly/linking. Continue using external assemblers/linkers; replacing them is not a prerequisite for credibility.

Safety and concurrency: bound preprocessing and compiler resource use, preserve output on failure, audit atomics/TLS, and plan generated-program stack protection and optional instrumentation. OpenMP needs frontend lowering plus a runtime; linking threaded BLAS does not require PCFC11 to implement OpenMP. Parallel compilation of independent files is distinct from parallel execution of generated programs. [R13, R15]

Maintainability: modular target/ABI descriptions, versioned optimization reports, testable ownership and allocation lifetimes, clean provenance and retained upstream attribution. Keep process isolation for PyCForge until an in-process compiler has explicit context and lifetime rules. Validate fresh conversion, build-service, GUI and installer behavior before bundling. [E2]

Defer a new object writer/linker, GPU kernel language, universal auto-parallelization and additional CPU backends until core correctness and the first target contracts are stable. These are separate major projects. The milestones above are a proposed engineering sequence, not promised release dates or performance multipliers.

Basis, limits and references

Observed findings come from the two project archives below. External references establish standards and engineering techniques; they do not certify PCFC11. The prioritization, architecture migration and acceptance criteria are recommendations derived from the inspected design. Primary sources were checked for this assessment.

Project evidence

[E1] windows-20261026T074530Z-2090bkcy.zip. Native Windows PCFC11 0.2.13: summary.json; compiler-info.log; check_*.json; benchmark.json; division C/assembly. Compiler SHA-256 begins 2c69628e6dbfd0aa. The archive timestamp is retained verbatim; assessment date follows the conversation date.

[E2] PCFC11-0.2.13-source.zip, current source package version 8. Inspected README.md; docs/ARCHITECTURE.md, CONFORMANCE.md, C11_AUDIT.md, ROADMAP.md, VALIDATION.md; optimize.c, registers.c, pcfc11.h, main.c, preprocess.c and selected code-generation paths. Reviewed evidence/0.2.13/linux-feedback.zip and gates-legacy-bootstrap.json, gates-oracles.json, gates-sanitizers.json. Existing project instructions and the no-LLVM architecture were respected.

Primary technical references

[R1] WG14 N1570, public C11 committee draft. Conformance, alignment, conditional features and language/library requirements.

[R2] LLVM analysis and transform passes. Established scalar, SSA and loop transformation techniques.

[R3] LLVM MemorySSA and alias analysis. Memory dependence, aliasing and limitations involving observable effects.

[R4] LLVM auto-vectorization. Loop/SLP vectorization, runtime alias checks, costs and reductions.

[R5] GCC optimization options. PGO, LTO and optimization controls.

[R6] Finding and Understanding Bugs in C Compilers (PLDI 2011). Csmith and defined-behavior differential testing.

[R7] Random Testing for C and C++ Compilers with YARPGen. Generated test diversity and optimizer bug discovery.

[R8] Alive2 project. LLVM-oriented transformation validation and its scope limits.

[R9] OpenBLAS documentation and distribution guide. CBLAS/LAPACKE packaging, integer interfaces, threading and CPU dispatch.

[R10] BLIS project. Portable BLAS framework with architecture-specific kernels.

[R11] Netlib LAPACKE C interface. Matrix layouts, dimensions, complex representations and C interfaces.

[R12] LLVM language reference. Strict/constrained floating-point effects and representation.

[R13] GCC instrumentation options. Application instrumentation differs from testing a sanitized compiler.

[R14] NVIDIA cuBLAS documentation. GPU numerical libraries require runtime, layout and transfer integration.

[R15] OpenMP specifications. A separate parallel programming implementation scope.