ChatGPT

LLVM360 Project Comprehensive Report

Project Overview

LLVM360 is an experimental tool aiming to statically recompile Xbox 360 software (both the console's OS and games) into native Windows executables. In essence, it takes PowerPC machine code from Xbox 360 binaries and translates it into an equivalent program that can run on a PC. The project leverages the LLVM compiler framework (hence the name "LLVM360" combining *LLVM + Xbox 360*) as the backbone for code translation [1] [2]. The scope of LLVM360 includes handling low-level Xbox 360 executables (XEX files), converting the PowerPC instructions into an intermediate representation, and then using LLVM to generate optimized x86-64 code. Ultimately, the goal is to run Xbox 360 games or system software on a PC **without** runtime interpretation (unlike traditional emulators), by producing a new compiled executable.

This project is described by its author as a "personal experiment" with LLVM [3]. The initial codebase was bootstrapped with components from prior Xbox 360 recompiler efforts – for example, the XEX file loader and some instruction decoding logic were adopted from an older project (often referred to as "Redex's recompiler") [3]. This gave LLVM360 a starting point in understanding the Xbox 360's executable format and basic instructions. The broader vision is to statically recompile *full* Xbox 360 games/OS binaries ahead-of-time, which is an ambitious undertaking given the complexity of the console's hardware and system libraries.

Current Progress

Implemented Features: A significant portion of the fundamental infrastructure is already in place. Notably, the project can **load and parse Xbox 360 executables** (XEX files) – the code includes a working **XEX loader** that was derived from earlier tools [3]. In terms of code translation, a **PowerPC instruction decoder** and an **IR (Intermediate Representation) generator** have been developed under the `Naive+` module. This means many of the Xbox 360's CPU instructions are recognized and translated into LLVM IR or an intermediate form (the author mentions "some of the instructions" are already decoded in the current codebase [3]). The integration with LLVM is functional: LLVM360 uses the LLVM libraries to turn the translated code into machine code for x86-64. In fact, the build system demonstrates the full **pipeline from PowerPC code to a new Windows executable** – generating an LLVM IR file, optimizing it, compiling to an object file, and then linking it into an `.exe` along with necessary runtime support libraries [4]. This confirms that the project is already capable of producing a native Windows binary from an Xbox 360 program (albeit with many features stubbed or incomplete, as discussed later).

Runtime and Emulation Support: Alongside pure code translation, LLVM360 implements parts of the Xbox 360 environment. A basic **memory manager** is in place to simulate the Xbox 360's memory address space for the program at runtime (e.g. allocating virtual memory with appropriate alignment) [5] [6]. There is also an **emulation layer for system calls and OS routines**. The code includes stubs or partial implementations of Xbox 360 kernel functions in modules like `XboxKrnl_Impl.h` (for low-level kernel calls) and `Xam_Impl.h` (for higher-level OS services). For example, functions such as

`RtlInitAnsiString`, `NtAllocateVirtualMemory`, and `KeGetCurrentProcessType` are implemented to the extent needed to allow the recompiled program to proceed [7] [5] [8]. Many other functions are currently **stubbed** – they log a "not implemented" warning and return dummy values [9] [10] – indicating where future work is needed (e.g. `DbgPrint_X` simply logs that it's unimplemented, and `RtlEnterCriticalSection_X` returns a placeholder result [9] [10]). This partial implementation strategy is common at the current stage: it allows the recompiled code to run through some initialization even if full functionality (like actual thread synchronization or debug printing) isn't there yet.

Development Activity: The project is under active development as of 2025. The original repository has accumulated on the order of **70+ commits** so far [11], and updates are being pushed regularly (the repository had commits as recently as mid-2025, indicating ongoing progress). For instance, recent commits have involved refining the compilation pipeline and runtime integration – the inclusion of a batch script to invoke LLVM's assembler, optimizer, and the MSVC linker shows a polished workflow for producing output binaries [4]. There is also evidence of new features being worked on, such as a **GUI interface** for the recompiler: the project includes **Dear ImGui** as a submodule [12] and even a `Graphics/ImGuiTest.h` file, suggesting that a graphical tool (perhaps for visualization or an editor) is being prototyped. In summary, LLVM360 has made substantial headway in loading Xbox 360 programs and generating runnable PC code. The core translation and linking framework is in place, and basic support systems are implemented at least in skeleton form.

Major Components & Implementation Status: The table below summarizes the major components of LLVM360 and the current state of their implementation:

Component / Feature	Implementation Status
XEX Loader (Executable Parser)	**Implemented:** Capable of loading and parsing Xbox 360 XEX binaries (borrowed from an older recompiler project) [3]. This provides the raw code and data for translation.
PowerPC Instruction Decoder	**Partially Implemented:** Many CPU instructions are decoded into an IR form. Basic integer and control-flow instructions are supported, but some complex instruction sets (floating-point, vector/VMX) may still be incomplete (work in progress).
IR Generation & LLVM Integration	**Implemented:** Translated code is converted into LLVM IR and processed by LLVM. The toolchain can produce a compiled x86-64 object and link it into a Windows executable [4]. Further IR optimizations and correctness improvements are ongoing.
Runtime/Emulation Library (Xbox 360 OS calls & environment)	**Partially Implemented:** A runtime library (`XRuntime`) is linked into the output. It includes a **Memory Manager** (for virtual memory allocation) and stubs for many **Xbox kernel (`XboxKrnl`) and XAM** functions. Some critical functions are implemented (e.g. memory allocation, simple string init) [5] [7], while many are **stubbed** with TODOs (logging "not implemented" and returning defaults) [9] [10].

Component / Feature	Implementation Status
Multi-threading Support	**In Progress:** The Xbox 360's multi-threading and synchronization primitives are recognized (e.g. critical sections, as seen in code stubs [10]). Thread context management exists (the code uses a thread-local context pointer), but a refactor is planned to improve this (see TODO list) [13]. True concurrent thread execution and locking is not fully implemented yet.
Graphics/Audio (GPU Emulation)	**Not Yet Addressed:** The project currently focuses on CPU and OS recompile. Direct GPU/graphics emulation is not implemented. In the static recompiler context, handling graphics and audio calls (e.g. translating them to PC graphics APIs or stubbing them out) remains an open problem for the future.
GUI / Tooling (ImGui Interface)	**Prototype Stage:** The inclusion of ImGui [12] suggests a work-in-progress GUI (perhaps for a function browser or debug interface). This is not yet a user-facing feature, but groundwork is being laid for interactive tools to assist development and possibly user configuration.

Overall, LLVM360's current progress demonstrates a working foundation: it can *take an Xbox 360 binary, process its code, and output a native executable* that links against an emulation runtime. The primary execution pathway is established, and now the focus is shifting toward filling in all the missing pieces of functionality.

Key Contributors and Maintainers

The LLVM360 project was originally created and primarily developed by **AleBello7276** (GitHub user *AleBlbl*), who is the author of the repository from which this fork originates [14]. AleBello7276 is the main contributor responsible for the core code, commits, and overall direction of the project to date. On GitHub, the project has attracted interest (nearly 20 stars) but as of now no large external contributions – the original repo shows zero open pull requests and issues [15], implying that development has mostly been a one-person effort so far.

The repository under review here, **Grien25/LLVM360**, is a fork of AleBello7276's project. Thus, **Grien25** (the user requesting this report) is a maintainer of this fork. If Grien25 has made custom changes or additions in their fork, those would position them as a contributor as well; however, no public data indicates significant divergent commits yet. It appears the fork is intended to track and possibly contribute to the upstream project. In summary, *AleBello7276* remains the principal maintainer/author of LLVM360's codebase [14], while *Grien25* maintains their fork and is poised to become an active contributor. The project is open for collaboration – the README explicitly invites others to fork and submit pull requests, and even provides a Discord server link for developers to discuss and coordinate [16]. As development continues, we may see more contributors join, but currently the key person behind LLVM360's progress is the original author.

Architectural Goals and Design

High-Level Design: LLVM360's design centers on leveraging the **LLVM compiler infrastructure** to translate PowerPC code to x86-64. Unlike a typical emulator that would JIT-translate code on the fly, this project

performs an *ahead-of-time (AOT) recompilation*. The architectural goal is to achieve better performance and integration by producing a self-contained executable that runs Xbox 360 code natively. To do this, the project defines its own intermediate representation and translation passes (in the `llvm360/Naive+` module) which convert Xbox 360 machine instructions into LLVM IR. Then it uses LLVM's optimization and code generation capabilities to target a Windows x86_64 binary [4]. The end result is intended to be an `.exe` that can be run on Windows as if it were a normal application, with the Xbox 360 code "baked in" via static recompilation.

Structure: The repository is organized into a few key modules reflecting this architecture. The `Naive+` module contains the **recompiler core** – it includes the instruction decoder, IR generator, and likely the control flow logic that drives the translation. The naming "Naive" suggests that initially a straightforward 1:1 translation approach is used (each PowerPC instruction translated to a sequence of IR instructions) before any sophisticated optimizations. Another module, `Emulator`, contains the **runtime environment** that the recompiled code will interface with. This includes subcomponents like `Runtime` (global runtime context, memory manager) and `Xen` (which houses implementations for Xbox 360 kernel (Xboxkrnl) and XAM APIs). The design follows a **High-Level Emulation (HLE)** approach for system calls: instead of emulating hardware devices, many Xbox OS calls are implemented in C++ on the host, at a high level, or stubbed out if not critical [9] [10]. For example, memory allocation in the guest is mapped to memory allocation in the host process (with appropriate alignment) [5] [6], and certain OS routines like `HalReturnToFirmware` (console reboot/shutdown) are handled by calling `exit(0)` on the host [17].

Technologies Used: The project is written in **C++** and uses **CMake** for its build system (with a Visual Studio solution generation on Windows) [18]. It relies on the **LLVM library** (specifically LLVM tools and libs corresponding to "LLVM 19" as noted in documentation) for assembling and optimizing IR, as well as generating machine code. The build instructions mention linking against pre-built LLVM Windows libraries [18]. Additionally, **Microsoft Visual C++ (MSVC)** toolchain is used at the final link stage to produce the executable, linking against Microsoft's standard libraries and a custom runtime library [19]. The inclusion of **Dear ImGui** (a popular immediate-mode GUI library) as a submodule [12] indicates that the developers plan to incorporate a graphical interface – likely for debugging, visualization of the recompiled code, or as a front-end for the tool. This could manifest as a GUI that allows browsing the decompiled functions or monitoring execution when running the recompiled program.

Design Goals: One high-level goal of LLVM360 is **accuracy and completeness** in executing Xbox 360 binaries. By statically analyzing and recompiling the entire program, the project aims to handle all code paths in the binary, which means the recompiler must support the vast majority of the Xenon (Xbox 360's CPU) instruction set and system calls. Another goal is **performance** – the expectation is that a statically recompiled game could run near-natively fast, since the heavy translation work is done ahead of time and LLVM's optimizations can be applied. The choice of static recompilation also suggests a goal of *portability*: the output does not require an emulator to run, just the normal OS environment and some bundled runtime libs. Of course, this design comes with challenges: any dynamic behavior (like runtime-generated code, or expecting certain hardware timings) must be accounted for in the static process, and completeness of system libraries (graphics, audio, input) is needed. The architecture of LLVM360 leans towards a hybrid of static recompilation for CPU logic and HLE for system APIs – an approach that, if successful, could allow Xbox 360 games to run as standalone applications on PC.

In summary, the architecture marries **LLVM's compilation strengths** with a bespoke **Xbox 360 compatibility layer**. It is structured to translate the bulk of game code into optimized native code, while

intercepting or emulating OS-level interactions via C++ implementations. The presence of a UI library (ImGui) also hints at tooling to support the development process (for instance, debugging or profiling the translated code). This design is forward-looking, aiming for a finished product where an end-user might input a Xbox 360 game binary and receive a Windows executable in return.

Steps to Completion and Remaining Tasks

While LLVM360 has laid down a strong foundation, there are several key tasks and missing components that must be completed for the project to reach its full goals:

- **Complete Instruction Set Coverage:** The Xbox 360's CPU (IBM Xenon, a PowerPC variant) includes advanced features like vector/SIMD instructions (VMX), floating-point operations, and special-purpose registers. Not all of these are fully handled yet. The recompiler needs to implement decoding and translation for *all* remaining instruction formats. As of now, certain instruction categories might still be unimplemented or only partially supported (e.g. some floating-point or graphics-related instructions). Finishing this will involve writing translation logic and testing each instruction's behavior.

- **Finish OS Function Implementations:** Many Xbox 360 **kernel (`Xboxkrnl`) and system (`XAM`) calls are stubbed out** in the current code (they log "not implemented" warnings) [9] [10]. To run real games, these need proper implementation. For example, functions for thread management (`RtlEnterCriticalSection_X` etc.), synchronization primitives, file I/O, and other OS services must be fleshed out. The project will likely implement these by mapping them to Windows API calls or high-level logic. A concrete item on the to-do list is improving thread support – the code currently uses a thread-local context hack and plans to remove that in favor of cleaner thread handling [13]. Completing critical sections, thread creation, and other concurrency features is crucial for any multi-threaded game to function correctly.

- **Graphics and Audio Emulation:** So far, LLVM360 has not tackled the GPU and audio processing aspects of the Xbox 360. These are complex subsystems (Xbox 360 uses a custom ATI GPU and XAudio). A static recompiler cannot easily "translate" GPU shader code or API calls into PC equivalents without significant work. Therefore, a major future task is to decide how to handle graphics: possibly by intercepting graphics API calls (D3D9 calls used by games) and mapping them to a PC graphics API (or using an existing emulator's GPU module). Similarly, audio APIs would need mapping to something like XAudio2 on Windows. These components might be integrated as additional libraries or by interfacing with projects like Xenia for those subsystems. As of now, any game's graphics calls would be no-ops – implementing at least basic functionality or a pass-through to PC APIs is needed for games to be playable.

- **Optimization and Stability:** The current "naive" translation approach prioritizes correctness and simplicity over optimization. As the project progresses, there will be tasks to improve the **performance** of the recompiled code – e.g. implementing more aggressive LLVM optimization passes, handling constant propagation, inlining, etc., specifically tailored to patterns of translated PowerPC code. Additionally, ensuring the **stability** of the output executables is a task: debugging the recompiled games to make sure they don't crash due to unhandled corner cases. This will likely involve extensive testing with different titles and writing fixes or adding support for any game-

specific behaviors (for instance, games that use certain undocumented system calls or rely on specific memory layouts).

- **Tooling and User Interface:** Another set of tasks revolves around the developer and end-user experience. The inclusion of ImGui suggests plans for **debugging tools or a user interface**. The project might develop a GUI application that wraps the recompiler, providing features like selecting a game, configuring settings, and visualizing the recompilation process. There may also be internal debugging UIs (for the developers) to inspect translated code or track execution. These tools need to be built out and polished. In the context of completion, a user-friendly front-end and thorough documentation would be important so that others can easily use LLVM360 on their own games.

- **Project Maintenance and Community Involvement:** Since there are no formal GitHub issues or PRs open [15], the remaining task tracking is done informally (e.g., in a `TODO.md` file and on Discord). One task is to possibly start organizing these tasks more publicly as the project gains contributors. Encouraging community testing and contributions will help cover more games and use-cases. For example, if someone tries a particular game and it fails, that could lead to discovering an unimplemented instruction or OS call, which can then be added. Establishing a testing suite or a list of target games to validate would be a beneficial step toward completion.

The **TODO list** in the repository highlights some immediate technical tasks. For instance, it notes the need to *"remove the need of the ctx TLS variable"* used for thread context, implying a refactor for how thread-local data is handled [13]. This is one specific example of the kind of cleanup needed as the project matures. Beyond that, a likely roadmap to a "complete" state would involve: getting one or two real games fully working as a proof of concept (possibly using simpler titles as test cases), then generalizing that support to more titles. Each new game might reveal additional functions or edge cases to implement.

In summary, **LLVM360 is still in an early-to-mid development phase**, where the core framework is operational but a lot of functionality remains unimplemented. No critical *show-stopper* issues have been reported on GitHub (since formal issue tracking isn't used yet), but the work ahead is clearly defined by the gaps in implementation. Filling in those gaps – from CPU instruction support to OS service emulation and beyond – constitutes the bulk of the remaining effort. As these tasks are addressed one by one, the project will move closer to its goal: allowing an Xbox 360 program to be run on a PC seamlessly after static recompilation. The developers are actively working through this to-do list, and with each commit (the project was updated as recently as mid-2025), LLVM360 is moving closer to a complete Xbox 360 static recompiler solution. [9] [13]

---

[1] [11] [14] [15] GitHub - AleBello7276/LLVM360: LLVM PowerPC Static Recompiler for the Xbox 360 OS and Games
https://github.com/AleBello7276/LLVM360

[2] [3] [16] [18] README.md
https://github.com/AleBello7276/LLVM360/blob/3a1fcf96924c6a9a756cdd56ce3a920efb4ef2b0/README.md

[4] [19] compile.bat
https://github.com/AleBello7276/LLVM360/blob/3a1fcf96924c6a9a756cdd56ce3a920efb4ef2b0/compile.bat

5 6 7 8 9 10 17 XboxKrnl_Impl.h

https://github.com/AleBello7276/LLVM360/blob/3a1fcf96924c6a9a756cdd56ce3a920efb4ef2b0/llvm360/Emulator/src/Runtime/Xen/XboxKrnl_Impl.h

12 .gitmodules

https://github.com/AleBello7276/LLVM360/blob/3a1fcf96924c6a9a756cdd56ce3a920efb4ef2b0/.gitmodules

13 TODO.md

https://github.com/AleBello7276/LLVM360/blob/3a1fcf96924c6a9a756cdd56ce3a920efb4ef2b0/TODO.md