Universidad de Alcalá de Henares
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Emulador Basado en QEMU de RISCV-NOEL

Autor: Fermín Verdolini

Tutor: Antonio Da Silva Fariña

Cotutor: Miguel Solinas

2025

UNIVERSIDAD DE ALCALÁ DE HENARES

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Emulador Basado en QEMU de RISCV-NOEL

Autor: Fermín Verdolini

Tutor: Antonio Da Silva Fariña

Cotutor: Miguel Solinas

Tribunal:

Presidente:

Vocal 1º:

Vocal 2º:

Fecha de depósito: 1 de septiembre de 2025

"Si nos detenemos, si aceptamos la persona que somos al caer,
el viaje concluye. Ese fracaso pasa a ser nuestro destino.
Amar el viaje implica no aceptar ese final.
He descubierto, por medio de dolorosas experiencias,
que el paso mas importante que puede dar alguien es siempre el siguiente."
Brandon Sanderson, *Juramentada*

Resumen

Este trabajo presenta el diseño e implementación de una plataforma de emulación para sistemas embebidos basada en QEMU, orientada a replicar una arquitectura compatible con el procesador `NOEL-V` de Gaisler, una implementación RISC-V de propósito general. Se desarrolló una máquina virtual funcional denominada `noel-srg`, que emula periféricos básicos como UART, GPIO y temporizadores, permitiendo ejecutar software embebido real compilado con la toolchain oficial.

Además del desarrollo del modelo, se diseñaron pruebas funcionales automatizadas integradas en un pipeline de integración continua utilizando GitHub Actions. Este enfoque permite validar el entorno emulado sin hardware físico, facilitando la detección de errores y asegurando la estabilidad del sistema a lo largo del tiempo.

El proyecto demuestra la viabilidad del uso de emulación como herramienta educativa y profesional en el ámbito embebido, y destaca el valor del software libre como base para desarrollos flexibles, reproducibles y colaborativos. Se plantea también una serie de mejoras y líneas futuras, como la ampliación del conjunto de periféricos, el soporte para arquitecturas multinúcleo y la eventual integración del código desarrollado al repositorio oficial de QEMU.

Palabras clave: emulación, sistemas embebidos, QEMU, RISC-V, GRLIB, integración continua, software libre.

Abstract

This project presents the design and implementation of an emulation platform for embedded systems based on QEMU, targeting a system architecture compatible with the `NOEL-V` processor from Gaisler, a general-purpose RISC-V implementation. A functional virtual machine, named `noel-srg`, was developed to emulate essential peripherals such as UART, GPIO, and timers, enabling the execution of real embedded software compiled with the official toolchain.

In addition to developing the model, automated functional tests were created and integrated into a continuous integration pipeline using GitHub Actions. This setup allows validating the emulated environment without physical hardware, helping detect errors and ensure system stability over time.

The project demonstrates the feasibility of emulation as both an educational and professional tool in embedded development, and highlights the value of free software as a foundation for flexible, reproducible, and collaborative solutions. Future work includes expanding peripheral support, enabling multicore simulation, and preparing the code for potential contribution to the official QEMU repository.

Keywords: emulation, embedded systems, QEMU, RISC-V, GRLIB, continuous integration, open-source software.

Índice general

Índice de tablas

Índice de listados de código fuente

Capítulo 1

Introducción

El desarrollo de sistemas embebidos modernos presenta crecientes desafíos vinculados con la validación temprana del software, la integración con hardware personalizado y la necesidad de mantener ciclos de desarrollo ágiles y confiables. En este contexto, el uso de herramientas de emulación y virtualización se ha vuelto fundamental para abordar estas problemáticas, permitiendo simular entornos completos de ejecución sin necesidad de contar con el hardware físico desde las primeras etapas del desarrollo.

Este trabajo se enmarca en esa problemática, proponiendo la extensión de una plataforma de emulación basada en QEMU para incorporar soporte funcional a una arquitectura de sistema embebido concreta: la máquina `NOEL-V` desarrollada por Gaisler para aplicaciones aeroespaciales, se basa en el estándar abierto RISC-V y se integra con la biblioteca GRLIB, ampliamente utilizada en sistemas críticos.

A través del diseño, implementación y validación de nuevos periféricos dentro del modelo de máquina virtual, este proyecto busca ofrecer un entorno reproducible y automatizado para la ejecución de pruebas funcionales sobre software embebido. La solución se integra, además, dentro de un flujo de integración continua, lo que permite verificar de forma automática la estabilidad de la plataforma ante cada cambio en el código.

En las siguientes secciones se detallan la motivación del trabajo, los objetivos planteados y el enfoque de desarrollo adoptado para su implementación.

1.1 Motivación

El desarrollo de sistemas embebidos conlleva múltiples desafíos técnicos, especialmente en las etapas iniciales donde el hardware aún no está disponible o es costoso de replicar. Esta limitación impacta tanto en el entorno profesional como en el académico. La posibilidad de emular plataformas completas a nivel de sistema permite mitigar estos obstáculos, facilitando el desarrollo temprano, la depuración intensiva y la validación funcional sin requerir acceso físico al hardware [1], [2].

En el contexto educativo, aprender sobre sistemas embebidos suele implicar el uso de placas específicas, dispositivos periféricos y entornos de programación restrictivos. Esto genera barreras de entrada importantes para estudiantes y centros con recursos limitados. En este sentido, la emulación ofrece una vía accesible para explorar arquitecturas complejas, experimentar con software de bajo nivel y adquirir experiencia práctica sin los costos ni riesgos asociados al hardware real [1].

En el ámbito industrial, y especialmente en empresas pequeñas o con procesos informales, el testing de software embebido suele estar subestimado o directamente ausente. La necesidad de disponer de

múltiples bancos de prueba físicos, junto con la dificultad para reproducir entornos consistentes, complica la implementación de flujos de validación automatizados. Esta situación lleva a una falta de control sobre la calidad del software, mayor riesgo de regresiones y una dependencia excesiva de pruebas manuales [3].

El uso de plataformas de emulación como QEMU, combinadas con estrategias de integración continua y diseño cuidadoso de los tests, permite abordar estos problemas de manera eficaz. No solo se facilita el desarrollo incremental y reproducible del software embebido, sino que se sientan las bases para prácticas más profesionales y escalables en el diseño de sistemas críticos [4].

1.2 Objetivos

El presente trabajo de fin de grado tiene como objetivo general desarrollar un entorno virtualizado funcional y confiable que permita validar software embebido sin requerir hardware físico. Para ello, se extiende el emulador QEMU con una nueva máquina virtual basada en la arquitectura `NOEL-V`, de Gaisler, replicando periféricos comunes como UART, GPIO y temporizadores incluidos en la biblioteca `GRLIB`.

Este entorno busca ser suficientemente flexible como para permitir la incorporación de nuevas plataformas o variantes de hardware con un esfuerzo razonable. De esta forma, cualquier desarrollador podría adaptar fácilmente el modelo virtual a uno nuevo basado en GRLIB o similar, acelerando las etapas de desarrollo y prueba.

Los objetivos específicos del proyecto son:

- Analizar las características de QEMU como plataforma de emulación para sistemas embebidos, en particular para arquitecturas RISC-V.
- Estudiar el entorno de ejecución de la plataforma `NOEL-V` y su integración con el ecosistema GRLIB.
- Diseñar y desarrollar una máquina virtual personalizada en QEMU que represente una plataforma basada en `NOEL-V`.
- Emular periféricos básicos (UART, GPIO, timer) de forma funcional, garantizando la comunicación con software real.
- Diseñar y ejecutar pruebas automáticas sobre el entorno emulado, asegurando la validez del modelo virtual.
- Integrar un sistema de integración continua (CI) que automatice la ejecución de los tests ante cambios en el código.
- Documentar el proceso seguido para extender QEMU, facilitando su replicación o adaptación a nuevas plataformas similares.

Con este enfoque, se espera contribuir a la reducción de la dependencia del hardware físico durante las etapas tempranas del desarrollo embebido, al tiempo que se habilita un marco reproducible, automatizable y de bajo costo para pruebas funcionales.

1.3 Desarrollo general del sistema

Este trabajo se centró en la extensión del emulador `QEMU` [2] para incorporar una nueva plataforma basada en la arquitectura `RISC-V` [5], modelando una plataforma virtual compatible con el ecosistema de la empresa `Gaisler`. La plataforma diseñada, denominada `noel-srg`, reproduce el comportamiento basado en el procesador `NOEL-V` [6] y periféricos típicos incluidos en la biblioteca `GRLIB` [7], ampliamente utilizada en sistemas embebidos críticos.

Uno de los pilares fundamentales de este desarrollo es el aprovechamiento de herramientas de código abierto. La elección de `QEMU` no solo se debe a su madurez técnica y su amplio soporte de arquitecturas [3], sino también a su naturaleza *open-source*, que permite acceder, comprender y modificar el código fuente del emulador. Esta característica resulta clave tanto en contextos educativos —donde promueve la experimentación y el aprendizaje profundo— como en entornos profesionales, donde la adaptabilidad es esencial para afrontar requisitos particulares sin depender de soluciones privativas [1].

El proceso de desarrollo se llevó a cabo respetando las convenciones del proyecto oficial de `QEMU`, manteniendo la compatibilidad con sus herramientas y estructuras internas. Se definió un modelo de máquina que describe la topología de dispositivos, la asignación de memoria y el comportamiento funcional de periféricos como la UART, el temporizador y los GPIOs. Estos componentes fueron implementados y verificados mediante binarios de prueba escritos en lenguaje C, compilados con la toolchain específica de Gaisler `riscv-gaisler-elf-gcc`.

Las pruebas iniciales se realizaron en un entorno local basado en Linux, y posteriormente se implementó un sistema de integración continua utilizando `GitHub Actions` [4]. Este *pipeline* permite ejecutar de manera automática los tests funcionales cada vez que se introduce un cambio en el repositorio, asegurando la estabilidad de la plataforma virtual y fomentando un flujo de trabajo colaborativo y profesional.

El modelo resultante fue concebido con una arquitectura modular y extensible, lo que permite su adaptación a nuevas variantes de plataformas o la incorporación de periféricos adicionales con relativa facilidad. Esto lo convierte en una base sólida tanto para proyectos de investigación como para iniciativas de formación en diseño de sistemas embebidos.

Estructura del documento

El presente trabajo se organiza en los siguientes capítulos:

- **Capítulo 2 – Estudio teórico y estado del arte:** Se presentan los conceptos fundamentales sobre emulación y virtualización en sistemas embebidos. También se revisan casos reales del uso de QEMU en contextos académicos e industriales, y se analiza su relevancia en entornos RISC-V y pipelines de integración continua.
- **Capítulo 3 – Emulador Basado en QEMU de RISCV-NOEL:** Se describe el proceso de implementación de una nueva máquina virtual en QEMU, basada en la arquitectura NOEL de Gaisler. Se detallan las decisiones de diseño, el soporte de periféricos y las herramientas utilizadas para extender QEMU.
- **Capítulo 4 – Pruebas de validación funcional e integración continua:** Se documentan los tests funcionales desarrollados para validar la plataforma virtual (UART, GPIO, temporizador, interrupciones), junto con la automatización de pruebas mediante un entorno de integración continua basado en GitHub Actions.

- **Capítulo 5 – Conclusiones y líneas futuros:** Se sintetizan los logros del proyecto, se reflexiona sobre los aportes obtenidos y se plantean posibles líneas de trabajo a futuro.

Capítulo 2

Estudio teórico y estado del arte

Este capítulo presenta los conceptos fundamentales sobre emulación y virtualización en el contexto de los sistemas embebidos, así como un análisis del uso actual de herramientas como QEMU en investigación, industria y plataformas abiertas como RISC-V. Se introduce además la relevancia de Gaisler y su arquitectura NOEL-V, dado su vínculo directo con el sistema virtualizado desarrollado en este trabajo [6], [7].

2.1 Introducción

El diseño y validación de sistemas embebidos ha evolucionado significativamente gracias a las técnicas de emulación y virtualización. Estas herramientas permiten desarrollar y probar software en ausencia del hardware final, reduciendo costos, acelerando el ciclo de desarrollo y facilitando tareas como la depuración, la automatización del testing o la validación funcional temprana [1], [3].

QEMU, una de las plataformas de emulación más utilizadas, ofrece un entorno flexible y extensible para modelar sistemas heterogéneos, incluyendo SoCs personalizados y arquitecturas no convencionales como RISC-V [2], [8], [9]. Este capítulo profundiza en su rol actual en proyectos reales, tanto en academia como en la industria [10], [11].

2.2 Emulación y virtualización en sistemas embebidos

Conceptos fundamentales

- **Virtualización:** Técnica que permite ejecutar múltiples sistemas operativos sobre una misma plataforma física mediante hipervisores, compartiendo recursos del hardware. Es común en entornos cloud y servidores [12].
- **Emulación:** Técnica que replica el comportamiento completo de un sistema hardware en otro diferente, tanto a nivel de CPU como de periféricos. Resulta clave para arquitecturas personalizadas, ISA alternativas o hardware aún no fabricado [13], [14].

En el ámbito embebido, la emulación resulta especialmente útil para validar periféricos, probar firmwares y desarrollar software sin necesidad de acceso físico al hardware [1].

2.3 QEMU como emulador abierto y extensible

2.3.1 Historia y evolución

QEMU (Quick Emulator) es una plataforma de emulación y virtualización desarrollada inicialmente por Fabrice Bellard en el año 2003 [8]. Desde sus comienzos, QEMU se diseñó como un emulador rápido, capaz de ejecutar código binario destinado a una arquitectura de hardware distinta a la del host. A lo largo de los años, ha evolucionado hasta convertirse en una herramienta de referencia para el desarrollo, prueba y validación de sistemas operativos, firmwares y sistemas embebidos [2].

Una de sus características más destacadas es su capacidad para emular una amplia variedad de arquitecturas, incluyendo x86, ARM, RISC-V, PowerPC, SPARC, MIPS, entre otras [8]. Esta versatilidad lo convierte en una solución ideal tanto para entornos académicos como industriales que requieren validar software sobre arquitecturas heterogéneas. QEMU también ofrece integración nativa con herramientas de depuración como GDB, lo que permite analizar en detalle el comportamiento del sistema emulado, incluyendo registros, memoria y puntos de interrupción [2].

Adicionalmente, su arquitectura modular ha permitido que sea adoptado como base para otras herramientas y tecnologías de virtualización, como KVM (Kernel-based Virtual Machine), virt-manager, y plataformas de automatización de pruebas [8], [15]. Esta extensibilidad ha sido clave para su adopción masiva y su continua evolución dentro del ecosistema del software libre.

2.3.2 Casos de uso comunes

QEMU ha encontrado aplicaciones en una amplia gama de escenarios. En el desarrollo de sistemas operativos y firmwares, permite a los ingenieros ejecutar y depurar código sin necesidad de disponer de hardware físico, facilitando la validación temprana de componentes críticos [1], [16]. Es especialmente útil en el caso de nuevas arquitecturas como RISC-V, donde el hardware puede no estar aún disponible o puede ser costoso [5], [9].

En el contexto de la integración continua (CI/CD), QEMU se emplea para ejecutar automáticamente pruebas de regresión, validación funcional y pruebas de integración [3], [4]. Gracias a su capacidad de ser ejecutado en entornos desatendidos y su soporte para salida sin interfaz gráfica (`-nographic`), es una herramienta ideal para pipelines de testing en entornos embebidos.

También es ampliamente utilizado en entornos educativos, donde permite a estudiantes interactuar con sistemas reales emulados, experimentar con arquitecturas de bajo nivel, y entender mejor el funcionamiento de sistemas operativos y microprocesadores [1]. Además, permite simular periféricos específicos, lo cual es de gran utilidad para el estudio de controladores y subsistemas de entrada/salida [2].

En definitiva, QEMU ha trascendido su papel original como emulador para convertirse en una plataforma versátil y esencial en el desarrollo moderno de sistemas informáticos.

2.4 Aplicaciones reales de QEMU en investigación e industria

La emulación de hardware mediante herramientas como QEMU ha dejado de ser un recurso limitado a laboratorios académicos o pruebas aisladas, para convertirse en un componente esencial dentro del ciclo de vida del software embebido. En la actualidad, QEMU se emplea activamente tanto en entornos de investigación como en procesos industriales, cumpliendo un rol central en el desarrollo temprano, la validación funcional y la automatización de pruebas [1], [3].

El auge de arquitecturas abiertas como RISC-V, sumado a la creciente complejidad de los SoCs modernos, ha impulsado la adopción de plataformas virtualizadas como medio para reducir la dependencia del hardware físico y acelerar los ciclos de desarrollo [5], [9]. La emulación permite validar periféricos, probar sistemas operativos, depurar errores y realizar regresiones en múltiples configuraciones de hardware desde entornos reproducibles y automatizables.

A continuación, se presentan ejemplos representativos que ilustran las capacidades actuales de QEMU:

- **Sistemas DAQ embebidos (Zabołotny, 2021–2022):** Se empleó QEMU para validar de forma simultánea el firmware FPGA, el controlador de kernel en Linux y la aplicación de usuario, facilitando el desarrollo iterativo de sistemas de adquisición de datos sin requerir tarjetas físicas PCIe [16].
- **Plataformas MIPS64 embebidas (Mehmood et al., 2016):** QEMU fue extendido para emular Octeon, una arquitectura MIPS64 utilizada en sistemas embebidos, demostrando que la emulación puede alcanzar desempeños comparables al hardware real durante las etapas iniciales del desarrollo [17].
- **Rehosting de firmware embebido (Jiang et al., 2021):** Se desarrolló una técnica para reemplazar periféricos reales por equivalentes virtuales compatibles, permitiendo ejecutar imágenes Linux embebidas originales dentro de QEMU. Esta técnica, conocida como *trasplante de periféricos*, alcanzó una tasa de éxito del 87 % en más de 800 firmwares [18].
- **Emulación de SoCs IoT (Osman, 2020):** El SoC nRF51 fue emulado exitosamente en QEMU, ejecutando sistemas operativos como Zephyr. Aunque con ciertas limitaciones temporales, la velocidad y funcionalidad fueron suficientes para validar aplicaciones embebidas IoT [8][19].
- **Sistemas críticos y virtualización (Cinque et al., 2021):** En entornos aeroespaciales y automotrices, QEMU se ha empleado como base para pruebas certificables, habilitando aislamiento funcional y validación de componentes críticos sin acceso al hardware final [3].

Estos casos reflejan una tendencia creciente: el uso de QEMU no se limita a simular CPU, sino que se ha convertido en una herramienta clave para modelar sistemas embebidos completos, incluyendo buses, periféricos personalizados, controladores de interrupciones y más [15]. Su integración con pipelines CI, su compatibilidad con arquitecturas modernas y su naturaleza extensible lo convierten en un recurso estratégico para el diseño confiable de sistemas embebidos contemporáneos.

2.5 Emulación con QEMU en RISC-V y entornos de integración continua

El auge de la arquitectura `RISC-V`, impulsado por su naturaleza abierta y modular [5], [9], ha favorecido el desarrollo de herramientas y flujos de trabajo que dependen fuertemente de la emulación. En este contexto, `QEMU` [2] se ha consolidado como una plataforma fundamental para ejecutar, depurar y validar software en arquitecturas `RISC-V` incluso en ausencia de hardware físico. Su integración con pipelines de integración continua (CI) permite una validación sistemática y automatizada de firmwares, sistemas operativos y componentes críticos, habilitando el desarrollo confiable de productos embebidos [4].

CI para verificación funcional en RISC-V: Proyecto Caliptra

Antmicro, en colaboración con Google y otras compañías del consorcio CHIPS Alliance, implementó un pipeline de CI para validar de forma continua el núcleo VeeR de `RISC-V` dentro del proyecto Caliptra, utilizado en soluciones de raíz de confianza por empresas como AMD, Microsoft y NVIDIA. Esta infraestructura ejecuta pruebas funcionales y de cobertura RTL en cada commit sobre múltiples configuraciones del core (EH1, EH2, EL2), garantizando estabilidad, seguridad y mantenibilidad del hardware [20], [21].

Testing de Linux RISC-V en CI con hardware real y virtual

En Codethink, se desarrolló un pipeline de CI sobre GitLab que combina la emulación en `QEMU` con despliegue automático sobre hardware real (placas SiFive Unmatched). Se utilizan herramientas como LAVA y OpenQA para validar el arranque, login y funcionamiento básico de una imagen Linux construida para `RISC-V`, asegurando que las modificaciones en el kernel o el entorno de usuario no introduzcan regresiones [22], [23].

Pruebas automatizadas de RTOS y firmware

Proyectos como Apache NuttX emplean `QEMU` dentro de pipelines CI para validar binarios de firmware embebido. En cada ejecución, el sistema arranca una imagen y analiza su salida para detectar errores de acceso a memoria, fallos de segmentación o comportamiento incorrecto, todo ello sin necesidad de hardware físico [24].

Adopción comunitaria en entornos profesionales

Foros y espacios técnicos de desarrollo embebido (como Reddit r/embedded) reflejan una adopción generalizada de `QEMU` como entorno de pruebas unitarias y de integración. Muchas compañías y desarrolladores independientes emplean `GitHub Actions` o `Jenkins` junto con `QEMU` para ejecutar baterías de tests en CI, simulando arquitecturas `RISC-V`, `ARM` o `MIPS`. En ocasiones, estos entornos se complementan con hardware real conectado mediante runners personalizados, permitiendo una validación híbrida [4].

En conjunto, estos casos confirman que el uso de `QEMU` en CI ha trascendido el ámbito experimental. Hoy es una herramienta central en procesos DevOps aplicados a sistemas embebidos, ofreciendo una base sólida para la validación funcional de arquitecturas abiertas como `RISC-V`, con beneficios concretos en tiempo de desarrollo, reproducibilidad y robustez del software.

2.6 Procesadores Gaisler y la arquitectura NOEL-V

Gaisler, una división de CAES, es reconocida por sus procesadores orientados a sistemas espaciales y críticos. Sus productos incluyen [6], [7]:

- **LEON:** Familia de procesadores basados en SPARC V8, ampliamente utilizados en aplicaciones espaciales y sistemas críticos.
- **NOEL-V:** Primer procesador `RISC-V` desarrollado por Gaisler [5], [6].
- **GRLIB:** Biblioteca IP en VHDL que incluye UARTs, timers, GPIOs y otros periféricos [7].

- **NCC Toolchain:** Toolchain GCC/GDB adaptada para sistemas Gaisler.

Este trabajo replica el entorno de una plataforma NOEL-V compatible con GRLIB, utilizando QEMU como emulador de sistema completo. Dada la dificultad de acceso al hardware real en contextos críticos, como el aeroespacial, el uso de entornos virtuales como QEMU resulta clave para validar el software en condiciones reproducibles [1], [3].

2.7 Conclusión del capítulo

La emulación mediante QEMU representa hoy una herramienta consolidada tanto en la investigación como en la industria. En el contexto de este TFG, su uso permite abordar el diseño y prueba de un sistema embebido complejo basado en RISC-V y periféricos GRLIB, replicando entornos industriales como los de Gaisler sin requerir hardware físico. El marco teórico y estado del arte presentado sienta así las bases para el desarrollo práctico abordado en los capítulos siguientes.

Capítulo 3

Emulador Basado en QEMU de RISCV-NOEL

3.1 Introducción

Este capítulo expone de forma detallada el proceso de desarrollo llevado a cabo para lograr la emulación funcional del softcore *RISC-V NOEL* en el emulador de código abierto *QEMU* [2], incluyendo la integración de los periféricos esenciales y la implementación de un entorno de pruebas automatizadas mediante técnicas de integración continua [4].

El propósito principal de este capítulo es documentar la creación de una nueva plataforma virtual incorporada al código fuente de QEMU, describiendo los componentes necesarios para su funcionamiento, así como el proceso de diseño e implementación de cada uno de ellos. Se incluye además una explicación del funcionamiento interno del emulador QEMU [2], con énfasis tanto en la estructura general del sistema como en la emulación de periféricos.

A lo largo del proyecto, el proceso de desarrollo adoptó un enfoque iterativo y evolutivo. Se comenzó con una implementación mínima funcional, sobre la cual se fueron incorporando progresivamente los distintos módulos y periféricos requeridos. Este método permitió una validación continua del sistema y una detección temprana de errores, lo que resultó en un diseño más robusto y flexible.

Las herramientas principales empleadas en el desarrollo fueron las siguientes:

- **QEMU**: Emulador de hardware de código abierto que permite ejecutar sistemas operativos y aplicaciones en diferentes arquitecturas [2]. En este proyecto se utilizó para emular el procesador RISC-V NOEL y los periféricos asociados.
- **RISC-V NOEL**: Softcore basado en la arquitectura RISC-V [5], desarrollado por Cobham Gaisler [6], orientado a sistemas embebidos y entornos críticos. Su emulación permite la validación de software sin necesidad de hardware físico, aprovechando además la biblioteca GRLIB para periféricos comunes [7].
- **GDB (GNU Debugger)**: Herramienta de depuración utilizada para examinar el estado interno del sistema emulado, establecer puntos de interrupción y analizar el flujo de ejecución del código [25].
- **Git**: Sistema de control de versiones distribuido, empleado para el seguimiento de los cambios en el código fuente durante el desarrollo del proyecto [26].

- **GitHub Actions**: Plataforma de integración continua y entrega continua (CI/CD), utilizada para automatizar la compilación, ejecución y validación de pruebas sobre el sistema emulado [4].

Este capítulo se encuentra estructurado de la siguiente manera:

- **Sección 3.1** – Introducción: Contextualiza los objetivos del desarrollo.
- **Sección 3.2** – Funcionamiento interno de QEMU: Expone la arquitectura modular del emulador y su flujo de inicialización.
- **Sección 3.3** – Adaptación del core y diseño de la plataforma virtual: Detalla la implementación de la nueva máquina virtual y la integración de los periféricos.
- **Sección 3.4** – Entorno de ejecución y depuración: Describe el uso de GDB para la depuración del sistema emulado y la configuración del entorno de pruebas.

3.2 Funcionamiento interno de QEMU

El emulador QEMU (*Quick EMUlator*) es una plataforma de virtualización y emulación de hardware de código abierto que permite replicar el comportamiento de distintas arquitecturas de procesador, dispositivos y sistemas completos en máquinas de propósito general [2]. Su diseño modular y extensible, basado en un traductor dinámico eficiente, lo convierte en una herramienta fundamental para el desarrollo y prueba de software en entornos donde el acceso al hardware físico es limitado o costoso [27].

Esta sección tiene por objetivo describir en profundidad la arquitectura interna de QEMU y su funcionamiento, con énfasis en los componentes relevantes para la implementación de nuevas máquinas y periféricos, como en el caso de la emulación del softcore *RISCV-NOEL*.

Para ello, se analizarán los siguientes aspectos:

- La estructura modular del código fuente de QEMU.
- El proceso de inicialización y arranque de una máquina virtual.
- La representación y conexión de dispositivos y periféricos.
- El estilo de codificación y las convenciones internas del proyecto.

3.2.1 Estructura modular del código fuente de QEMU

QEMU se caracteriza por poseer una arquitectura modular y jerárquica que facilita la incorporación de nuevas plataformas, procesadores y periféricos. Esta modularidad se refleja en la organización de su código fuente, el cual se encuentra dividido en múltiples directorios que encapsulan funcionalidades específicas del emulador. Comprender esta estructura resulta fundamental para extender QEMU de forma correcta y coherente con sus convenciones internas [28].

A continuación, se describen los directorios más relevantes para la implementación de una nueva máquina basada en una arquitectura determinada, como en el caso de *RISCV-NOEL*:

- **`hw/`**: Contiene la implementación de hardware virtualizado. Este directorio se subdivide por tipo de dispositivo o arquitectura. Por ejemplo, `hw/riscv/` contiene las máquinas específicas de RISC-V, mientras que `hw/char/`, `hw/timer/`, y `hw/gpio/` incluyen las implementaciones de periféricos de carácter, temporizadores y controladores de entrada/salida, respectivamente.

- **`target/`**: Incluye la implementación del backend de CPU y las instrucciones específicas de cada arquitectura. Para RISC-V, se encuentra en `target/riscv/`. Aquí se define el modelo de ejecución de la CPU, su decodificación de instrucciones, registros, excepciones y el manejo de interrupciones.
- **`include/`**: Contiene los encabezados públicos y privados utilizados a lo largo del proyecto. Las definiciones de estructuras, macros, interfaces entre módulos y funciones compartidas están aquí. En particular, `include/hw/` contiene las declaraciones de periféricos y máquinas virtuales.
- **`docs/`**: Contiene documentación técnica del proyecto, incluyendo guías para contribuir al código, estándares de codificación, y especificaciones de arquitectura para dispositivos virtuales.
- **`softmmu/`** y **`exec/`**: Encargados de la capa de ejecución, manejo de memoria virtual, traducción dinámica y control del bucle principal de emulación (*main loop*) [27].
- **`qapi/`** y **`monitor/`**: Módulos utilizados para la comunicación con el entorno externo (por ejemplo, mediante QMP - QEMU Machine Protocol). Aunque no son necesarios para una emulación básica de una nueva plataforma, sí son relevantes para desarrollos avanzados.

La modularidad del diseño no solo organiza el código, sino que establece interfaces bien definidas entre componentes, facilitando la reutilización de dispositivos virtuales ya implementados. Por ejemplo, un controlador UART puede ser utilizado en múltiples arquitecturas con mínimas modificaciones, simplemente mediante su inclusión y configuración adecuada en el archivo de definición de la máquina virtual correspondiente.

Esta estructura también permite una separación clara entre el modelo de CPU (que define cómo se ejecuta el código máquina) y el modelo de plataforma (que define cómo se conectan periféricos y memoria). Esto es esencial para el desarrollo escalable y para garantizar la consistencia del proyecto a medida que se agregan nuevas funcionalidades [28].

3.2.2 Proceso de inicialización y arranque de una máquina virtual

La inicialización de una máquina virtual en QEMU sigue un flujo bien definido, en el cual se construyen e interconectan los componentes esenciales de la plataforma emulada. Este proceso es altamente configurable, lo cual permite definir plataformas virtuales personalizadas mediante código en el propio núcleo del emulador [2], [28].

Cuando QEMU se ejecuta con una arquitectura y máquina específicas, como por ejemplo:

```
qemu-system-riscv32 -M noel-srg -kernel test.elf
```

se activa una secuencia de inicialización cuyo flujo principal se detalla a continuación [27], [28]:

1. **Selección y registro de la máquina virtual:** Al compilar QEMU, las distintas máquinas disponibles para una arquitectura se registran mediante la macro `machine_init()`. Cada una de ellas implementa una estructura del tipo `MachineClass`, que incluye funciones de inicialización, parámetros por defecto y dispositivos asociados.
2. **Ejecución de la función de inicialización de la máquina:** En el caso de RISC-V, esta función suele estar definida en archivos como `hw/riscv/virt.c` o una variante propia, como `noel.c`. Dentro de esta función, se crean los dispositivos virtuales, se asignan direcciones en el mapa de memoria, y se configuran buses e interconexiones.

3. **Creación de la CPU:** Se instancia la CPU mediante la función `cpu_create()`, utilizando el tipo de procesador especificado. QEMU utilizará la definición correspondiente del directorio `target/riscv/` (por ejemplo, `riscv_cpu_realize()`), donde se configura el modelo de CPU, sus extensiones ISA, y otros parámetros relevantes (tamaño de RAM, privilegios, etc.).

4. **Asignación de memoria y carga del binario:** Se reserva un bloque de memoria RAM emulada con la función `memory_region_init_ram()`, y se carga el archivo ejecutable indicado con el parámetro `-kernel`. El cargador de binarios puede interpretar formatos como ELF o binarios planos, y ubica el contenido en la dirección correspondiente del espacio de direcciones.

5. **Inicialización de periféricos:** Se instancian e inicializan los periféricos declarados en la máquina (UART, timer, GPIO, etc.), utilizando funciones como `qdev_create()`, `sysbus_connect()` y otras del sistema de dispositivos de QEMU. Cada periférico puede mapearse a direcciones de memoria específicas mediante `memory_region_add_subregion()`.

6. **Inicio de la emulación:** Una vez creada la máquina, instanciada la CPU, y configurada la memoria y los dispositivos, se lanza el bucle de emulación. Este ejecuta instrucciones del binario cargado, emulando el comportamiento del sistema embebido en su conjunto.

Este diseño basado en objetos y registros dinámicos permite que nuevas máquinas sean definidas con relativa facilidad, reutilizando periféricos ya implementados y adaptando su comportamiento según las necesidades del sistema [28]. Asimismo, cada etapa del arranque puede ser personalizada para reflejar el hardware real que se desea emular.

3.2.3 Representación y conexión de dispositivos y periféricos

QEMU utiliza un modelo de dispositivos altamente modular y orientado a objetos, diseñado para facilitar la emulación precisa de componentes hardware. Cada dispositivo (periférico) se modela como una instancia de una clase derivada de `DeviceState`, y puede conectarse a buses virtuales, recibir interrupciones y mapear regiones de memoria de forma controlada y reutilizable [28], [29].

Modelo de dispositivo

La base de todo periférico en QEMU es la estructura `DeviceState`, definida en `hw/qdev-core.h`. Esta representa la instancia de un dispositivo particular, incluyendo punteros a sus regiones de memoria, funciones de inicialización y conexiones a buses. Los periféricos se definen mediante macros como `DEFINE_TYPE_INFO`, que permiten registrar el tipo y vincular funciones específicas de inicialización y reseteo.

Para los dispositivos que se conectan a un bus de sistema (como la mayoría de los periféricos en una SoC embebida), se utiliza la clase `SysBusDevice`, que extiende `DeviceState` y facilita la asignación de direcciones de memoria y líneas de interrupción [2], [28].

Inicialización y mapeo de memoria

Durante la inicialización de la máquina (función `machine_init()`), los periféricos se instancian con `qdev_new()` o su equivalente moderno `object_new()`. Luego, se conectan al bus del sistema con `sysbus_realize()` y se les asigna una dirección en el espacio de direcciones mediante:

```
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, direccion_base);
```

También se pueden declarar múltiples regiones de memoria utilizando `memory_region_init_io()` o `memory_region_init_ram()`, y luego añadirlas como subregiones [28]:

```
memory_region_add_subregion(&address_space, base_addr, &region);
```

De esta forma, cualquier acceso del binario emulado a esa dirección se redirige a las funciones `read/write` asociadas al periférico.

Conexión de interrupciones

Los dispositivos pueden generar interrupciones al procesador a través de líneas de IRQ. En QEMU, estas conexiones se realizan con funciones como:

```
sysbus_connect_irq(SYS_BUS_DEVICE(dev), 0, cpu_irq_line);
```

Donde `cpu_irq_line` es una línea de interrupción expuesta por el modelo de CPU. La relación entre IRQs, controladores de interrupciones (como CLINT o PLIC en RISC-V) y dispositivos se configura explícitamente en el código de la máquina [28].

Reutilización y extensión

Una de las fortalezas del diseño de QEMU es que los periféricos pueden reutilizarse entre múltiples arquitecturas. Por ejemplo, el controlador UART `ns16550` se utiliza tanto en x86 como en RISC-V. Para ello, basta con instanciar el dispositivo y mapearlo correctamente en la nueva plataforma, sin modificar su implementación interna.

Esta reutilización es clave para acelerar el desarrollo de nuevas plataformas virtuales y reducir errores. Además, si se requiere un nuevo periférico, este puede implementarse como una nueva clase derivada de `SysBusDevice`, con sus propias funciones de `read/write/reset` y comportamiento específico [2], [29].

En resumen, la conexión de dispositivos en QEMU se basa en el principio de encapsulación modular, donde cada componente puede integrarse mediante interfaces estandarizadas. Esto permite simular con precisión sistemas completos y facilita tanto el desarrollo como la validación de software embebido sobre plataformas virtuales [2], [28].

3.2.4 Estilo de codificación y convenciones internas de QEMU

El proyecto QEMU sigue un conjunto de convenciones estrictas en cuanto al estilo de codificación, estructura de archivos, nomenclatura y organización del código. Estas reglas no sólo facilitan la lectura y el mantenimiento del código fuente, sino que también permiten la colaboración entre cientos de desarrolladores alrededor del mundo. Las directrices se encuentran documentadas en la guía oficial de estilo del proyecto [30], la cual se inspira en gran medida en las reglas del kernel de Linux [31]. En este apartado se describen las principales normas que deben seguirse al desarrollar nuevos módulos o modificar componentes existentes en QEMU.

Formato del código

El código de QEMU está mayoritariamente escrito en lenguaje C, y sigue un estilo basado en la guía de codificación del kernel de Linux [31]. Entre las reglas más importantes se encuentran:

- Identación de 4 espacios (no se utilizan tabulaciones).
- Líneas de código con una longitud máxima de 80 caracteres.
- Uso de llaves incluso en bloques de una sola línea.
- Separación clara entre declaraciones, lógica de control y comentarios.
- Comentarios en estilo C (`/* ... */`) preferentemente sobre el bloque que describen.

Para verificar que el código cumple con las reglas de estilo, QEMU proporciona un script llamado `scripts/checkpatch.pl`, adaptado del kernel de Linux, además de la posibilidad de utilizar herramientas como `clang-format` con configuraciones específicas. El uso de estas herramientas es obligatorio para los parches propuestos al repositorio oficial [30].

```
./scripts/checkpatch.pl --no-tree --strict archivo.c
```

Estructura y modularidad

Cada nuevo periférico, máquina o componente debe residir en un archivo dedicado dentro del subdirectorio correspondiente a su tipo:

- `hw/`*`tipo`*`/`: Implementación del dispositivo o máquina.
- `include/hw/`*`tipo`*`/`: Archivos de encabezado exportados.
- `configs/devices/`*`arch`*`/`: Activación de dispositivos por arquitectura.

Esta modularidad asegura la separación entre arquitectura, periféricos, CPU y lógica general del sistema, lo cual es fundamental en una base de código de gran tamaño [30].

Convenciones de nomenclatura

Los nombres de funciones y estructuras siguen una jerarquía lógica basada en su tipo:

- `noel_init()`, `noel_cpu_realize()`: Funciones de inicialización de una máquina o CPU personalizada.
- `noel_uart_write()`, `noel_timer_read()`: Accesos a periféricos específicos.
- `NOEL_STATE`, `NOEL_UART_STATE`: Estructuras de estado de cada componente.

Las macros y constantes suelen escribirse en mayúsculas con guiones bajos, y las funciones públicas deben ser prefijadas con el nombre del componente para evitar colisiones [30].

Buenas prácticas para contribuir

El equipo de QEMU mantiene una lista de correo (`qemu-devel@nongnu.org`) como canal principal de colaboración. Todo parche debe ser enviado en formato `git send-email`, acompañado de una descripción detallada del cambio, cumplimiento del formato de código, licencia explícita (`Signed-off-by:`) y comentarios adecuados en el código. Estas prácticas se documentan en la guía oficial de envío de parches [32].

Antes de enviar contribuciones, se recomienda ejecutar la batería completa de pruebas con `make check` y validaciones específicas para la arquitectura modificada [32].

Adherirse a estas convenciones no solo facilita la revisión del código por parte de la comunidad, sino que también incrementa la probabilidad de aceptación en el repositorio principal. En este proyecto se siguieron dichas pautas para garantizar la compatibilidad con futuras versiones de QEMU y permitir la reutilización del trabajo desarrollado.

3.3 Adaptación del core y diseño de la plataforma virtual RISCV-NOEL

En este apartado se detalla el proceso de adaptación de una máquina base del emulador QEMU para modelar una máquina virtual compatible con el softcore RISCV-NOEL. Esta implementación se basa en la reutilización y modificación de la arquitectura RISC-V ya soportada en QEMU [2], [33], integrando periféricos compatibles con la biblioteca GRLIB [7].

La máquina virtual resultante, denominada `noel-srg`, fue desarrollada a partir del diseño de referencia `virt` de QEMU [34], extendida mediante periféricos derivados de la infraestructura GRLIB (Gaisler Research Library), ampliamente utilizada en entornos espaciales y sistemas embebidos críticos [6].

La implementación considera aspectos clave como:

- La definición del sistema en chip (SoC) y su instancia en la máquina virtual.
- El mapeo de memoria correspondiente a cada dispositivo.
- La integración de periféricos esenciales: UART, GPIO y temporizadores.
- El manejo de interrupciones a través del PLIC y el controlador ACLINT.

El código fuente de esta máquina está estructurado en los archivos `noel_srg.c` y `noel_srg.h`.

3.3.1 Estructura general del SoC y la máquina `noel-srg`

La arquitectura de la máquina virtual `noel-srg` se basa en dos estructuras principales definidas en `noel_srg.h`:

- `NOELSRGSoCState`: Define el estado interno del SoC, incluyendo CPU, PLIC, GPIO y regiones de memoria.
- `NOELSRGState`: Representa el estado completo de la máquina virtual, encapsulando el SoC y parámetros adicionales.

Estas estructuras son instanciadas y registradas en QEMU como tipos personalizados mediante macros como `OBJECT_CHECK` y `type_init()` [28].

La función principal de inicialización del sistema es `noel_srg_machine_init()`, que se encarga de:

1. Verificar y asignar el tamaño de memoria RAM disponible.
2. Inicializar el objeto SoC (`s->soc`) y realizar su configuración.
3. Mapear la memoria DTIM (Data Tightly Integrated Memory) en la dirección base 0x00000000.
4. Cargar el binario ejecutable si se ha proporcionado vía parámetro `-kernel`.

Listado 3.1: Función de inicializacion del sistema noel_srg_machine_init()

```
static void noel_srg_machine_init(MachineState *machine)
{
    MachineClass *mc = MACHINE_GET_CLASS(machine);
    const MemMapEntry *memmap = noel_srg_memmap;

    NOELSRGState *s = RISCV_NOEL_SRG_MACHINE(machine);
    MemoryRegion *sys_mem = get_system_memory();

    if (machine->ram_size != mc->default_ram_size) {
        char *sz = size_to_str(mc->default_ram_size);
        error_report("Invalid RAM size, should be %s", sz);
        g_free(sz);
        exit(EXIT_FAILURE);
    }

    /* Initialize SoC */
    object_initialize_child(OBJECT(machine), "soc", &s->soc, TYPE_RISCV_NOEL_SRG_SOC);
    qdev_realize(DEVICE(&s->soc), NULL, &error_fatal);

    /* Data Tightly Integrated Memory */
    memory_region_add_subregion(sys_mem,
    memmap[NOEL_SRG_DEV_DTIM].base, machine->ram);

    if (machine->kernel_filename)
    {
        riscv_load_kernel(machine, &s->soc.cpus,
        memmap[NOEL_SRG_DEV_DTIM].base,
        false, NULL);
    }
}
```

Por su parte, el método `noel_srg_soc_realize()` realiza la creación concreta de los dispositivos internos del SoC. En esta función se inicializan:

- **CPU**: Mediante un array de hilos RISC-V con propiedades configurables [5].

- **PLIC**: Controlador de interrupciones compatible con SiFive [34].
- **ACLINT**: Controlador de interrupciones a nivel de núcleo (CLINT + MTIMER).
- **GRLIB GPIO**: Un controlador GPIO compatible con GRLIB, conectado dinámicamente a IRQs [7].
- **GRLIB UART**: Dispositivo `grlib-apbuart` mapeado a `0xFC001000`.
- **GRLIB GPTIMER**: Temporizador con dos canales, frecuencia definida por `CPU_CLK`.

El diseño sigue una arquitectura jerárquica donde la máquina `noel-srg` instancia un SoC (`riscv.noel.srg.soc`), que a su vez encapsula y realiza todos los dispositivos internos conectados por bus. Esta separación mejora la mantenibilidad del código y permite extender el sistema fácilmente con nuevos periféricos compatibles con la infraestructura GRLIB [6].

3.3.2 Mapeo de memoria del sistema

El mapa de memoria representa la distribución de las direcciones físicas asignadas a los distintos componentes del sistema, incluyendo memoria principal, dispositivos periféricos y regiones reservadas. Esta asignación debe ser coherente con la arquitectura esperada por el software que se ejecutará en el sistema emulado, ya que determina cómo accede el procesador a los periféricos [33].

En la implementación de la máquina `noel-srg`, el mapa de memoria se define en el arreglo `noel_srg_memmap[]` dentro del archivo `noel_srg.c`. Este define las direcciones base asociadas a cada componente del sistema. A continuación se resume la asignación utilizada:

Componente	Dirección base	Descripción
RAM (DTIM)	0x00000000	Memoria principal accesible por CPU
PLIC	0x0C000000	Controlador de interrupciones externo (Platform-Level Interrupt Controller)
ACLINT	0x02000000	Controlador de interrupciones a nivel de hart (CLINT + Timer)
UART (GRLIB APBUART)	0xFC001000	Periférico serie compatible con GRLIB
GPIO	0xFC000000	Controlador GPIO GRLIB simple
GPTIMER	0x80000300	Temporizador compatible GRLIB

Tabla 3.1: Mapa de memoria del sistema NOEL-SRG

Cada uno de estos dispositivos se instancia dinámicamente durante la función de inicialización del SoC (`noel_srg_soc_realize()`) y se mapea en el espacio de direcciones del bus principal del sistema utilizando las funciones de QEMU:

```
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, noel_srg_memmap[NOEL_SRG_DEV_XXX]);
```

donde `NOEL_SRG_DEV_XXX` es un identificador simbólico del componente (e.g., `NOEL_SRG_DEV_UART`).

Además del mapeo de direcciones, cada periférico se registra con el sistema de dispositivos de QEMU para que sus funciones de `read()` y `write()` sean invocadas automáticamente ante accesos del software emulado. Esta asociación es crítica para garantizar que el comportamiento del sistema refleje

el hardware real esperado por los binarios RISC-V generados, por ejemplo, mediante compiladores como `ncc gaisler` [6].

El cumplimiento de este mapa de memoria garantiza la compatibilidad del sistema emulado con el software de prueba, incluyendo drivers y controladores diseñados para el entorno NOEL-V y GRLIB, y permite replicar con precisión el entorno de ejecución típico de estos sistemas embebidos.

3.3.3 Integración de periféricos GRLIB (UART, GPIO, Timer)

La infraestructura GRLIB (Gaisler Research Library) proporciona una colección de periféricos ampliamente utilizados en sistemas embebidos críticos, especialmente en aplicaciones aeroespaciales [7]. En el contexto de este proyecto, se integraron en la máquina virtual `noel-srg` los siguientes módulos GRLIB: `apbuart` (UART), `gptimer` (temporizador) y `gpio` (entrada/salida digital), emulados como dispositivos QEMU compatibles [35].

Estos periféricos fueron instanciados durante la función `noel_srg_soc_realize()` mediante el uso del sistema de dispositivos de QEMU basado en `SysBusDevice` y su sistema de objetos (`QOM`) [28], [29].

UART (GRLIB APBUART)

El periférico UART se modeló utilizando el dispositivo `grlib-apbuart`, ya presente en QEMU bajo `hw/char/grlib_apbuart.c` [35]. Este dispositivo se conecta al bus del sistema mediante:

Listado 3.2: Instanciación del periférico UART GRLIB en QEMU

```
dev = qdev_new("grlib-apbuart");
sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, noel_srg_memmap[NOEL_SRG_DEV_UART]);
```

Como extensión a la funcionalidad original del `grlib-apbuart`, se implementó un *modo loopback*, destinado a facilitar la validación funcional sin necesidad de dispositivos externos. En este modo, cada byte escrito en el registro de datos es automáticamente reenviado al buffer de recepción como si hubiese sido recibido desde una fuente externa. La implementación de esta funcionalidad se realizó en la función `grlib_apbuart_write()`, donde al detectar que el modo loopback está habilitado mediante el registro de control, se inserta el dato escrito en la FIFO interna.

La integración del modo loopback en el periférico UART es esencial para validar su funcionamiento en entornos de prueba automatizados, ya que permite verificar tanto la transmisión como la recepción de datos sin depender de hardware físico [7]. Además, este dispositivo posibilita la salida estándar (`stdout`) desde el sistema emulado, facilitando la comunicación básica del software de prueba mediante impresión de mensajes y validación de tests.

GPIO

El periférico GPIO se instanció mediante el tipo `grlib-gpio`, definido en el archivo `hw/gpio/grlib_gpio.c` [28]. Este dispositivo proporciona registros de entrada, salida y dirección, lo que permite simular líneas digitales básicas de acuerdo con la especificación de GRLIB [7]. La configuración se realizó de la siguiente forma:

Listado 3.3: Instanciación del periférico GPIO GRLIB en QEMU

```
dev = qdev_new("grlib-gpio");
sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, noel_srg_memmap[NOEL_SRG_DEV_GPIO]);
qdev_connect_gpio_out(dev, 0, irq_out);
```

Este periférico puede utilizarse para emular interrupciones externas generadas por cambios en las entradas, lo que resulta útil para probar controladores de interrupciones y rutinas de servicio en software embebido [7].

GPTIMER

El temporizador se modeló utilizando el módulo `grlib-gptimer`, ubicado en `hw/timer/grlib_gptimer.c` [28]. Este dispositivo emula un temporizador de propósito general con múltiples canales, configurables en frecuencia y modo de operación. En el caso del sistema `noel-srg`, se inicializó con una frecuencia basada en la constante `CPU_CLK` y con dos canales habilitados:

Listado 3.4: Instanciación del periférico GPTIMER GRLIB en QEMU

```
dev = qdev_new("grlib-gptimer");
qdev_prop_set_uint32(dev, "nr-timers", 2);
qdev_prop_set_uint64(dev, "frequency", CPU_CLK);
sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, noel_srg_memmap[NOEL_SRG_DEV_GPTIMER]);
```

La implementación del temporizador en QEMU recrea el comportamiento especificado en la documentación oficial de GRLIB [7], donde se establece que los eventos del temporizador son generados de manera sincronizada y periódica, permitiendo una generación predecible de interrupciones. Este tipo de temporizador es comúnmente utilizado para planificación de tareas, medición de tiempos y rutinas de temporización precisa en sistemas embebidos.

Resumen de integración

La reutilización de los periféricos GRLIB disponibles en QEMU permitió una integración eficiente y realista de dispositivos comúnmente empleados en el ecosistema NOEL-V [7], [28]. El uso del sistema de dispositivos de QEMU junto con el sistema de objetos QOM (QEMU Object Model) facilitó la configuración modular y la conexión limpia de cada componente al mapa de memoria y al subsistema de interrupciones [28], [29].

Esta integración fue clave para lograr un entorno emulado funcionalmente representativo del hardware objetivo, permitiendo ejecutar binarios compilados con soporte para GRLIB y validar su comportamiento sin necesidad de hardware físico.

3.3.4 Manejo de interrupciones (PLIC y ACLINT)

El sistema RISCV-NOEL implementado en QEMU replica el modelo de interrupciones definido por la *RISC-V Privileged Architecture* [5]. Este modelo contempla dos niveles de control principales:

- **PLIC (Platform-Level Interrupt Controller)**: Administra las interrupciones externas provenientes de periféricos.
- **ACLINT (Advanced Core Local Interruptor)**: Gestiona las interrupciones locales por *hart*, como temporizadores o interrupciones de software.

Ambos controladores fueron integrados dentro del SoC virtual definido en la máquina `noel-srg`, empleando dispositivos ya disponibles en QEMU [36].

PLIC: Controlador de interrupciones externas

El PLIC se instanció mediante el dispositivo `riscv.plic`, ubicado en `hw/intc/sifive_plic.c`. Este componente permite recibir interrupciones desde múltiples fuentes y distribuirlas entre los diferentes hilos de ejecución (*harts*) [36]. Su configuración incluye:

- Número de fuentes IRQ: Cada periférico con capacidad de interrupción (UART, GPIO, Timer) constituye una fuente.
- Número de contextos: Cada *hart* dispone de un contexto de interrupción asociado.

La instancia se realiza con:

Listado 3.5: Instanciación del periférico PLIC en QEMU

```
object_initialize_child(obj, "plic", &s->plic, TYPE_RISCV_PLIC);
qdev_prop_set_uint32(DEVICE(&s->plic), "ndev", total_irqs);
sysbus_realize(SYS_BUS_DEVICE(&s->plic), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(&s->plic), 0, base_plic_addr);
```

Cada dispositivo con capacidad de generar interrupciones se conecta al PLIC mediante llamadas a `qdev_connect_gpio_out()`.

ACLINT: interrupciones locales y temporizador

El componente ACLINT, que combina CLINT y MTIMER, es responsable de las interrupciones internas del sistema, incluyendo:

- **Timer interrupts (MTIP)**: Generadas periódicamente por temporizadores.
- **Software interrupts (MSIP)**: Utilizadas para comunicación entre hilos de ejecución.

En la máquina `noel-srg`, el dispositivo `riscv.aclint` fue instanciado de la siguiente forma [37]:

Listado 3.6: Instanciación del periférico ACLINT en QEMU

```
object_initialize_child(obj, "aclint", &s->aclint, TYPE_RISCV_ACLINT);
qdev_prop_set_uint32(DEVICE(&s->aclint), "hartid-base", 0);
sysbus_realize(SYS_BUS_DEVICE(&s->aclint), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(&s->aclint), 0, base_aclint_addr);
```

Este controlador se integra directamente con los hilos de CPU simulados, permitiendo emular interrupciones temporizadas o de software como en una plataforma real. Los temporizadores como el GPTIMER pueden conectarse a través de IRQs externas, redirigidas vía el PLIC, mientras que los contadores del ACLINT generan eventos internos [5].

Conexión de interrupciones entre componentes

La conexión de dispositivos periféricos al sistema de interrupciones se realiza mediante líneas virtuales IRQ proporcionadas por el modelo de QEMU. Por ejemplo, la conexión de un temporizador a una línea IRQ gestionada por el PLIC puede realizarse así:

```
qdev_connect_gpio_out(dev, 0, qdev_get_gpio_in(DEVICE(&s->plic), irq_id));
```

Donde `irq_id` identifica la línea de interrupción correspondiente. Estas conexiones son fundamentales para que el sistema operativo o el firmware emulado reciban las señales adecuadas en respuesta a eventos de hardware.

Durante el desarrollo fue necesario ajustar el comportamiento de generación de interrupciones de algunos periféricos. Inicialmente, la UART y el temporizador generaban interrupciones mediante pulsos breves con la función `qemu_irq_pulse()`. Sin embargo, dado que el PLIC implementa un modelo de interrupciones *level-triggered*, se modificó este comportamiento para que las interrupciones permanecieran activas hasta ser gestionadas por el software [36].

En la UART, por ejemplo, se reemplazó la invocación a `qemu_irq_pulse()` por `qemu_irq_raise()` cuando se detectaban datos disponibles en el buffer de recepción. Una vez que el buffer quedaba vacío, se llamaba a `qemu_irq_lower()` para desactivar la línea de interrupción:

Listado 3.7: Manejo de interrupciones en el periférico UART

```
if (uart->control & UART_RECEIVE_INTERRUPT) {
    qemu_irq_raise(uart->irq);
}

// ...

if (!uart_data_to_read(uart)) {
    uart->status &= ~UART_DATA_READY;
    if (uart->control & UART_RECEIVE_INTERRUPT) {
        qemu_irq_lower(uart->irq);
    }
}
```

Un mecanismo similar fue implementado en el temporizador (`grlib-gptimer`), donde la interrupción se mantenía activa mientras el bit de *int pending* permanecía establecido.

Este enfoque garantiza una emulación coherente con el comportamiento esperado por el subsistema de interrupciones de la arquitectura RISC-V, donde las interrupciones externas deben permanecer activas hasta ser atendidas y confirmadas por el software [5].

En conjunto, el uso de PLIC y ACLINT permite emular fielmente el comportamiento esperado por plataformas RISC-V modernas. Gracias a esto, se pueden probar sistemas multitarea, rutinas de inte-

rrupción, temporizadores y controladores de bajo nivel en un entorno completamente virtualizado, sin necesidad de disponer de hardware físico [36].

3.3.5 Integración de la máquina en el sistema de construcción de QEMU

Para que QEMU compile y reconozca la nueva máquina `noel-srg`, fue necesario integrarla correctamente en su sistema de construcción. QEMU utiliza el sistema de build `meson`, complementado por un sistema de configuración modular basado en `Kconfig`, siguiendo una organización similar a la empleada en proyectos de gran escala como el kernel de Linux [28], [31].

Modificaciones en `meson.build`

En el archivo `hw/riscv/meson.build`, que define los archivos fuente que componen la arquitectura RISC-V, se añadió la referencia al archivo `noel_srg.c`, controlada mediante una opción de configuración [28]:

```
riscv_ss.add(when: 'CONFIG_NOEL_SRG', if_true: files('noel_srg.c'))
```

Al final del archivo, el conjunto de fuentes RISC-V es incluido dentro de los targets compilados:

```
hw_arch += {'riscv': riscv_ss}
```

De forma análoga, para soportar el periférico `grlib-gpio`, se añadió la opción correspondiente en `hw/gpio/meson.build`:

```
system_ss.add(when: 'CONFIG_GR_GPIO', if_true: files('gr_gpio.c'))
```

Definición en `Kconfig`

La máquina `noel-srg` y sus periféricos asociados fueron registrados en el archivo `hw/riscv/Kconfig` mediante la siguiente definición [28]:

Listado 3.8: Configuración de la máquina NOEL-SRG en Kconfig

```
config NOEL_SRG
    bool
    select RISCV_ACLINT
    select GR_GPIO
    select GRLIB
    select SIFIVE_PLIC
    select SIFIVE_E_PRCI
    select UNIMP
```

Además, el periférico GPIO se habilitó como una opción independiente en `hw/gpio/Kconfig`:

```
config GR_GPIO
    bool
```

Estas configuraciones aseguran que, cuando se activa la opción `NOEL_SRG`, también se habilitan automáticamente los módulos necesarios para el funcionamiento completo de la máquina virtual: controladores de interrupciones (PLIC y ACLINT), periféricos GRLIB, el GPIO, y módulos auxiliares como `UNIMP`.

Activación en las configuraciones por defecto

Para que la máquina `noel-srg` esté habilitada por defecto en una build típica de QEMU, se añadió su configuración en los siguientes archivos [28]:

- En `default-configs/devices/riscv32-softmmu.mak`:

  ```
  CONFIG_NOEL_SRG=y
  ```

- Y adicionalmente, en `config/devices/riscv32-softmmu/default.mak`, que agrupa las configuraciones base para la arquitectura:

  ```
  CONFIG_NOEL_SRG = y
  ```

Estas modificaciones permiten que, al construir QEMU para `riscv32-softmmu`, la máquina `noel-srg` y sus periféricos queden integrados y disponibles sin necesidad de configuración adicional.

Con estos cambios, el sistema de construcción de QEMU reconoce y compila correctamente todos los componentes necesarios para la emulación de la máquina `noel-srg`, garantizando su coherencia dentro del ecosistema modular del proyecto.

3.3.6 Estructura del binario y entorno de ejecución inicial

Para ejecutar correctamente los binarios en la máquina virtual `noel-srg`, fue necesario definir la estructura del ejecutable y la secuencia de inicialización mínima requerida por el sistema. Esta sección describe la composición del binario ELF utilizado en las pruebas y los elementos clave del proceso de arranque.

Estructura del ELF generado

Los binarios fueron generados en formato ELF de 32 bits (`elf32-littleriscv`), utilizando el ensamblador y enlazador de la toolchain `riscv-gaisler-elf-gcc` [6]. El script de enlazado (`linker.ld`) define las secciones básicas que conforman el binario:

- **.text**: contiene el código ejecutable del firmware. Se ubica en la dirección `0x00000000`, correspondiente al arranque por defecto de la máquina virtual.
- **.rodata**: almacena datos de solo lectura, como cadenas de texto constantes.
- **.data** y **.sdata**: contienen datos inicializados en tiempo de ejecución.
- **stack**: el script define la ubicación del tope de pila mediante la etiqueta `stack_top`, alineada a 4 bytes.

La directiva `ENTRY(_start)` establece el punto de entrada del binario en la etiqueta `_start`, definida en el archivo de ensamblador.

Listado 3.9: Script de enlazado linker.ld

```
OUTPUT_ARCH( "riscv" )
OUTPUT_FORMAT("elf32-littleriscv")
ENTRY( _start )
SECTIONS
{
  /* text: test code section */
  . = 0x00000000;
  .text : { *(.text) }
  /* gnu_build_id: readonly build identifier */
  .gnu_build_id : { *(.note.gnu.build-id) }
  /* rodata: readonly data segment */
  .rodata : { *(.rodata) }

  /* data: Initialized data segment */
  /* . = 0x80000000; */
  .data : { *(.data) }
  .sdata : { *(.sdata) }
  .debug : { *(.debug) }
   . = ALIGN(4);
  . += 0x1000;
  stack_top = .;

  /* End of uninitalized data segement */
  _end = .;
}
```

Startup assembly

El arranque del sistema se implementó mediante un archivo de ensamblador (`start.s`), que realiza la configuración inicial del entorno de ejecución. Sus funciones principales son:

- Leer el identificador del hart (`mhartid`) utilizando la instrucción `csrr`, conforme a la especificación RISC-V Privileged Architecture [5].
- Detener la ejecución de hilos secundarios mediante un bucle infinito si el hart no es el 0. Esta lógica permite que únicamente el primer núcleo ejecute el firmware de prueba, simplificando el entorno.
- Inicializar el puntero de pila (`sp`) con la dirección definida como `stack_top` en el linker script.
- Llamar a la función `kmain()`, definida en C, que representa el punto de inicio del firmware de prueba.

El flujo general de arranque es el siguiente:

1. `_start` es invocado automáticamente por el hardware virtual (QEMU) al cargar el binario.
2. El ensamblador verifica el hartid y configura el entorno mínimo.
3. Se transfiere el control a la función principal del firmware (`kmain()`).

Listado 3.10: Arranque del sistema de ejecución start.s

```
.align 2
.section .text
.globl _start

_start:
        csrr  t0, mhartid            # read hardware thread id
                                     #  (‘hart‘ stands for ‘hardware thread‘)

        bnez  t0, halt               # run only on the first hardware thread
                                     # (hartid == 0), halt all the other threads

        la    sp, stack_top          # setup stack pointer

        call    kmain

halt:   j     halt                   # enter the infinite loop
```

Importancia del arranque personalizado

Este esquema de arranque minimalista es característico de los sistemas embebidos, donde no existe un sistema operativo que inicialice el entorno. Al definir manualmente la pila y el flujo de ejecución, se asegura un entorno controlado y predecible, facilitando la depuración y el análisis de errores durante el desarrollo inicial.

La correcta configuración del *entry point* y la disposición de las secciones en memoria fueron claves para que QEMU pudiera cargar y ejecutar el binario directamente mediante el dispositivo `loader`, sin necesidad de un bootloader intermedio [33].

3.4 Entorno de ejecución y depuración

Una vez implementados los distintos componentes de la máquina virtual `noel-srg`, fue necesario preparar un entorno capaz de ejecutar binarios compilados para dicha plataforma y facilitar su depuración. Esta sección describe cómo se diseñó dicho entorno, incluyendo la configuración del emulador, los binarios de prueba desarrollados y las herramientas de depuración empleadas durante el desarrollo.

Compilación de binarios de prueba

Los binarios utilizados para validar el correcto funcionamiento del sistema fueron desarrollados en lenguaje C y ensamblador, utilizando el compilador `riscv-gaisler-elf-gcc`, parte de la toolchain *NCC* proporcionada por Gaisler [6]. Esta herramienta genera ejecutables en formato ELF compatibles con la arquitectura RISC-V implementada en NOEL-V, respetando sus convenciones específicas de inicialización y enlazado.

Cada binario fue diseñado para activar un subsistema particular del SoC emulado. Se desarrollaron pruebas específicas para los siguientes componentes:

- Comunicación serie por UART, mediante el envío y recepción de datos.

- Escritura y lectura en registros GPIO, verificando su correcto direccionamiento.
- Generación y manejo de interrupciones a través del temporizador GPTIMER.
- Soporte básico para ejecución multiproceso (multihart), evaluando el arranque simultáneo de varios núcleos.

Ejecución de binarios en QEMU

Los binarios fueron ejecutados utilizando QEMU con el backend RISC-V, cargando directamente los ejecutables ELF mediante el dispositivo `loader`, que permite ubicar el binario en memoria y comenzar su ejecución automáticamente [33]. El comando utilizado fue:

```
qemu-system-riscv32 -M noel-srg -nographic -no-reboot \
    -device loader,file=kernel.elf
```

Se empleó el modo `-nographic` para redirigir la salida UART al terminal, facilitando la observación de la salida del sistema embebido sin necesidad de interfaces gráficas. El parámetro `-no-reboot` evita que QEMU reinicie automáticamente al finalizar la ejecución del binario, permitiendo conservar la salida para su análisis.

Depuración con GDB

Durante el desarrollo, se utilizó `riscv-gaisler-elf-gdb` para depurar el comportamiento del sistema [25]. QEMU se ejecutó en modo *GDB server*, que permite conectar un depurador externo:

```
qemu-system-riscv32 -M noel-srg -S -gdb tcp::1234 \
    -device loader,file=test_gpio.elf
```

Donde el parámetro `-S` detiene la CPU al inicio, y `-gdb tcp::1234` habilita el puerto de depuración.

Desde otro terminal, se inició el depurador con el binario correspondiente:

```
riscv-gaisler-elf-gdb test_gpio.elf
(gdb) target remote :1234
(gdb) break main
(gdb) continue
```

Esto permitió inspeccionar el estado de los registros, examinar el contenido de memoria y verificar el flujo del programa paso a paso. Fue especialmente útil para depurar la inicialización del entorno y el manejo correcto de interrupciones [28].

Observación del comportamiento del sistema

Durante la ejecución, se monitorearon distintos indicadores para validar el comportamiento esperado:

- **Salida por UART**: utilizada como indicador primario del funcionamiento del sistema.
- **Estado de registros GPIO**: confirmando que las lecturas y escrituras correspondían a las operaciones realizadas.

- **Interrupciones atendidas**: observadas a través de mensajes en consola y verificaciones en los registros del PLIC.
- **Trazas internas de QEMU**: generadas con la opción `-d in_asm,cpu,exec` para obtener un log detallado de las instrucciones ejecutadas y los eventos de hardware simulados [28].

Estos mecanismos de monitoreo permitieron diagnosticar comportamientos anómalos y ajustar tanto la máquina virtual como el firmware de prueba. La información recolectada sirvió como base para la validación formal que se analiza en el capítulo siguiente.

Capítulo 4

Pruebas de validación funcional e integración continua

4.1 Introducción

En este capítulo se presentan las pruebas funcionales desarrolladas para validar el correcto funcionamiento del sistema emulado `noel-srg`. Las pruebas se enfocan en verificar el comportamiento esperado de los principales periféricos del SoC (UART, GPIO y temporizador), así como en comprobar la correcta gestión de interrupciones. Además, se describe el entorno de integración continua implementado para automatizar la ejecución de estas pruebas.

A través de estos escenarios de prueba, se buscó validar no solo la correcta implementación del hardware virtualizado, sino también el flujo completo de interacción entre el software embebido, los periféricos y el sistema de interrupciones, en línea con el enfoque de validación funcional que ha sido aplicado en otros proyectos basados en QEMU [1], [3], [15], [16]. Cada prueba se diseñó con el propósito de replicar condiciones típicas de uso en aplicaciones embebidas, asegurando así la aplicabilidad del sistema en contextos reales.

Por último, se presentan los resultados obtenidos durante las distintas fases de validación, incluyendo el análisis del comportamiento observado en la consola de QEMU y el éxito o fallo de cada escenario de prueba.

4.2 Entornos de experimentación y ejecución

El proceso de validación funcional y depuración inicial de la máquina `noel-srg` se llevó a cabo en dos entornos diferenciados: una estación de trabajo local para el desarrollo y pruebas preliminares, y un entorno automatizado basado en *GitHub Actions* para la ejecución continua de los tests en cada modificación del código.

Entorno local de desarrollo

La mayor parte del desarrollo y las pruebas iniciales se realizaron sobre una máquina con sistema operativo Ubuntu 22.04.5 LTS (`jammy`), ejecutando el kernel `6.8.0-60-generic`. Las características del hardware son las siguientes:

- **Procesador**: 13th Gen Intel Core i5-1340P (12 núcleos físicos, 16 hilos, frecuencia máxima de 4.6 GHz).
- **Memoria RAM**: 32 GiB.
- **Almacenamiento**: SSD NVMe de 435 GiB, con aproximadamente 293 GiB libres durante las pruebas.
- **Arquitectura del sistema**: `x86_64`, con soporte completo para virtualización por hardware (VT-x).
- **Número de núcleos detectados**: 16 (`nproc`).

La compilación del firmware de prueba se realizó utilizando la herramienta `riscv-gaisler-elf-gcc`, versión `ncc-v1.0.4`, y la emulación fue llevada a cabo con QEMU `8.2.50`, versión base sobre la cual se desarrollaron y validaron las modificaciones [2].

Este entorno permitió una rápida iteración durante el desarrollo inicial de la máquina virtual y sus periféricos, así como la ejecución manual de pruebas utilizando QEMU y GDB [25].

Automatización mediante Integración Continua

Tras la validación preliminar en entorno local, se configuró un pipeline de integración continua sobre *GitHub Actions* [4], garantizando que cada nueva modificación del repositorio ejecutara automáticamente las pruebas funcionales.

En este entorno automatizado, se descargan e instalan las dependencias necesarias (incluyendo la toolchain NCC), se compila QEMU para `riscv32-softmmu`, y posteriormente se construyen y ejecutan los binarios de prueba (`uartTest`, `gpioTest`, `timerTest`, `uartIrqTest`), validando su salida.

Este enfoque se alinea con las mejores prácticas de validación continua aplicadas en proyectos abiertos de RISC-V, como los desarrollos de CHIPS Alliance y Codethink [20], [22]-[24].

Esta automatización permite detectar de forma temprana posibles regresiones o errores introducidos por nuevas modificaciones del código, garantizando así la estabilidad funcional de la máquina `noel-srg` en todo momento.

El empleo conjunto del entorno local y de la integración continua permitió cubrir tanto las necesidades de desarrollo iterativo como la validación sistemática en cada cambio del código fuente.

4.3 Prueba de UART: Loopback test

Una de las primeras pruebas desarrolladas para validar la funcionalidad básica del sistema fue un test sobre el periférico UART, empleando su modo *loopback*. El objetivo principal de este test fue verificar que el flujo de datos desde la CPU hasta el periférico y su retorno al software funcionara correctamente, sin necesidad de hardware externo.

Objetivo de la prueba

El test buscó validar los siguientes aspectos del periférico `grlib-apbuart` [7], [35]:

- La capacidad de transmisión (TX) y recepción (RX) básica de la UART.
- El correcto funcionamiento del modo *loopback*, donde los datos transmitidos son redirigidos internamente al receptor.
- La lectura correcta del registro de datos y del estado del periférico por parte del software.

Implementación del test

El binario del test, denominado `uartTest`, fue implementado en el archivo `kernel.c`, utilizando las funciones definidas en el módulo `uart.h`. La lógica principal del test consiste en:

1. Habilitar los modos de transmisión y recepción de la UART mediante los registros de control.
2. Activar el modo *loopback*.
3. Transmitir un carácter (en este caso, la letra `'X'`) utilizando la función `outchar()`.
4. Leer desde el registro de datos mediante `getcar()` hasta recibir un dato o agotar un contador de espera.
5. Comparar el carácter recibido con el transmitido:
   - Si coinciden, imprimir por UART el mensaje `UART loopback test PASSED`.
   - Si no coinciden o hay timeout, imprimir un mensaje de error indicando el dato recibido.

Acceso directo a registros

La interfaz de acceso a la UART se realizó mediante una estructura que mapea los registros del periférico a una dirección fija (`0xFC001000`), siguiendo la convención GRLIB [7]. Esto permite realizar lecturas y escrituras de bajo nivel directamente sobre los registros del hardware virtualizado.

Importancia del test

Esta prueba es esencial porque permite validar el canal de comunicación serial básico del sistema emulado. La UART es una herramienta fundamental en sistemas embebidos, ya que constituye la vía principal de salida en entornos donde no existen interfaces gráficas. Además, al ejecutarse completamente en modo *loopback*, no requiere dispositivos externos ni interacción manual, lo que permite su automatización dentro del pipeline de integración continua.

Discusión de resultados

El resultado de la prueba confirmó el correcto funcionamiento del subsistema UART en el entorno emulado. La ejecución fue consistente a través de múltiples pruebas locales y en la automatización de la integración continua, sin que se detectaran variaciones en el comportamiento o fallos intermitentes.

Dado el carácter determinista de la prueba y la estabilidad del entorno emulado, no se consideró necesario realizar un análisis estadístico adicional. La validez de la prueba se sostiene en la reproducibilidad del resultado esperado y en la coherencia con la especificación funcional del periférico [7], [35].

Esta prueba constituye una validación esencial del camino básico de entrada/salida serial del sistema, sobre el cual se fundamentan tanto la interacción con el usuario como el diagnóstico en etapas posteriores del desarrollo.

4.4 Prueba de GPIO: validación de lectura y escritura digital

El segundo componente validado fue el periférico de propósito general GPIO (*General Purpose Input/Output*), incluido dentro de la librería GRLIB [7]. Esta prueba permite verificar el correcto direccionamiento de sus registros, así como el funcionamiento básico de las operaciones de escritura, lectura y configuración de dirección en el entorno emulado provisto por QEMU [2].

Objetivo de la prueba

La prueba de GPIO busca confirmar los siguientes aspectos:

- Configuración correcta de la dirección del pin (entrada/salida).
- Escritura correcta en el registro de salida (`OUTPUT_REG`).
- Lectura confiable desde el registro de entrada (`INPUT_REG`).
- Coherencia del valor leído con el valor escrito.

Implementación del test

El binario `gpioTest` implementa una secuencia simple que configura el pin número 22 como salida, escribe un valor alto y bajo, y verifica mediante lectura si el cambio se refleja correctamente:

1. Configura el pin 22 (`GPIO_PIN_MASK`) como salida mediante `gpio_dir_write()`.
2. Escribe un valor alto en el pin y verifica mediante `gpio_read()` que el bit correspondiente esté activo.
3. Escribe un valor bajo y vuelve a verificar la lectura.
4. Si ambas comprobaciones son correctas, se imprime el mensaje `GPIO test PASSED`. En caso contrario, se reporta el fallo.

Se introdujo una breve función de *delay* mediante bucles vacíos para permitir la propagación del estado del registro y facilitar la depuración.

Acceso a los registros del GPIO

El periférico fue mapeado en la dirección base `0xFC083000`, accediendo a sus registros mediante un puntero a memoria. La interfaz fue encapsulada en un módulo (`gpio.h` y `gpio.c`), que proporciona funciones de lectura, escritura y configuración de dirección. Este mecanismo reproduce el esquema de mapeo definido en GRLIB [7], asegurando coherencia entre la especificación y la emulación.

Importancia del test

La correcta operación del periférico GPIO es esencial para la emulación de interacciones básicas con hardware externo, tales como botones, switches o LEDs virtuales. Además, su prueba confirma que el mapeo de memoria y el acceso desde la CPU emulada se realizan sin errores. Al igual que el test de UART, este ejercicio puede automatizarse fácilmente en el entorno de integración continua, permitiendo detectar regresiones en el comportamiento del periférico sin intervención humana [16].

Resultados de la prueba

La ejecución del binario `gpioTest` mostró el siguiente resultado en la consola de QEMU:

```
GPIO test PASSED
```

Durante el proceso de validación no se detectaron discrepancias entre los valores escritos y leídos en el periférico, confirmando el correcto funcionamiento del acceso a registros y la coherencia de la configuración de dirección.

Discusión de resultados

El test fue ejecutado repetidamente en entornos locales y dentro del pipeline de integración continua, mostrando un comportamiento estable y predecible. El acceso a los registros del GPIO a través de la emulación no presentó errores ni inconsistencias, lo que valida tanto la implementación del periférico en QEMU [2] como la interacción del software embebido con el mismo.

Dado que la prueba reproduce un flujo básico de lectura y escritura digital, su éxito representa una verificación fundamental del sistema de mapeo de memoria y del acceso programático a dispositivos. Esta validación es especialmente relevante para aplicaciones que requieran manipulación de hardware externo simulado.

4.5 Prueba de temporizador: interrupciones periódicas

La última prueba funcional desarrollada para validar la máquina virtual `noel-srg` fue la prueba del temporizador (*GPTIMER*). Este periférico, definido en la librería GRLIB [7], se evaluó como generador de interrupciones periódicas, verificando tanto la correcta emisión de dichas señales como su atención por parte del núcleo de la CPU emulado en QEMU [2].

Objetivo de la prueba

Los objetivos principales de esta validación fueron:

- Configurar el temporizador para generar interrupciones periódicas.
- Registrar un manejador de interrupciones (*IRQ handler*) que contabilice los eventos.
- Validar que las interrupciones son generadas por el GPTIMER, enrutadas a través del PLIC y atendidas correctamente por el firmware [36], [37].

Implementación del test

El binario `timerTest` implementa los siguientes pasos:

1. Invoca la rutina `startup()`, que configura el entorno de interrupciones:
   - Inicializa el PLIC y el manejador global de excepciones.
   - Deshabilita y vuelve a habilitar globalmente las interrupciones.
2. Habilita la UART como salida estándar.
3. Registra la función `gptimer_handler()` como manejador de interrupciones externas mediante `install_irq_handler()`.
4. Configura el temporizador:
   - Establece un *scaler* bajo, para lograr una alta frecuencia de interrupciones.
   - Define un contador que provoca una interrupción tras 2 millones de ticks.
   - Activa el modo de reinicio automático e interrupciones.
5. Habilita las interrupciones externas mediante las instrucciones CSR correspondientes.
6. Entra en un bucle donde espera recibir tres interrupciones, ejecutando la instrucción `wfi` (*wait for interrupt*) para optimizar el consumo del CPU virtual.
7. Una vez recibidas las tres interrupciones, imprime un mensaje confirmando el éxito de la prueba.

El manejador `gptimer_handler()` incrementa una variable global cada vez que es invocado y limpia el estado de la interrupción en el temporizador mediante `timer1_clear_pending()`.

Manejo de interrupciones

La infraestructura de interrupciones utilizada incluyó:

- El temporizador GPTIMER genera una señal IRQ [7].
- Esta señal es entregada al PLIC, que la enruta al hart activo [37].
- El manejador global `bare_isr_handle()` definido en `startup.c` decodifica la fuente de la interrupción y llama a `irq_dispatch()`.
- Finalmente, `irq_dispatch()` invoca al manejador previamente registrado (`gptimer_handler()`).

Importancia del test

La correcta generación y gestión de interrupciones periódicas es esencial para el funcionamiento de sistemas operativos o aplicaciones embebidas que requieren temporización. Validar el GPTIMER demuestra que el entorno emulado en QEMU puede gestionar correctamente flujos asincrónicos de eventos, lo que reproduce fielmente el comportamiento del hardware real [16].

Este tipo de pruebas son fundamentales para el desarrollo de planificadores (*schedulers*), rutinas periódicas y subsistemas dependientes del tiempo.

Resultados de la prueba

Durante la ejecución del binario `timerTest`, la salida por UART mostró la secuencia de mensajes `Tick!`, confirmando que las interrupciones fueron generadas y atendidas correctamente. Tras cumplirse el número esperado de eventos, el sistema imprimió:

```
Timer test PASSED
```

Este resultado valida que el flujo completo —desde la configuración del temporizador, pasando por la generación de la interrupción, el enrutamiento a través del PLIC y la ejecución del manejador— opera correctamente en el entorno emulado.

Discusión de resultados

Las pruebas confirmaron que el GPTIMER puede generar interrupciones periódicas confiables y que el firmware puede atenderlas mediante su infraestructura básica de control de interrupciones. El comportamiento observado fue consistente en ejecuciones repetidas y dentro del pipeline de integración continua, confirmando la estabilidad del modelo emulado.

La verificación de este flujo es crítica en arquitecturas embebidas, ya que temporizadores como el GPTIMER suelen utilizarse para el control del tiempo, la planificación de tareas y la ejecución periódica de rutinas en tiempo real. Esta validación constituye una base sólida sobre la cual construir funcionalidades más complejas en sistemas multitarea o con requisitos temporales estrictos.

4.6 Automatización del testing mediante Integración Continua

Con el propósito de garantizar la calidad del código y la estabilidad de la máquina virtual `noel-srg`, se implementó un pipeline de integración continua (CI) basado en *GitHub Actions* [4]. Este mecanismo de automatización permite ejecutar las pruebas funcionales descritas en las secciones anteriores de forma sistemática ante cada modificación del repositorio, facilitando la detección temprana de fallos en el ciclo de desarrollo. Estrategias similares han demostrado ser eficaces en proyectos de verificación embebida y de ecosistemas RISC-V [20]-[22].

Estructura del pipeline

El pipeline, definido en el archivo `.github/workflows/ci.yml`, se activa automáticamente en cada evento de `push` o `pull request`, y está compuesto por las siguientes etapas:

1. **Preparación del entorno:**
   - Descarga del repositorio.
   - Inicialización del submódulo `firmware-tests`, que contiene los binarios de validación.
   - Instalación de dependencias requeridas (`glib`, `pixman`, compiladores y utilidades de compilación).
2. **Instalación de la toolchain Gaisler NCC:**
   - Se emplea `riscv-gaisler-elf-gcc` para compilar los binarios de prueba y `riscv-gaisler-elf-gdb` para tareas de depuración.

- La toolchain se descarga y almacena en caché, optimizando la ejecución del pipeline.

3. **Compilación de QEMU:**
   - Se construye únicamente el target `riscv32-softmmu`, habilitando las opciones de depuración que facilitan el diagnóstico de errores.
4. **Compilación de los binarios de prueba:**
   - Se generan los programas diseñados para ejercitar los periféricos implementados (`uartTest`, `gpioTest`, `timerTest`, `uartIrqTest`).
5. **Ejecución automatizada de pruebas:**
   - Cada binario se ejecuta en QEMU sobre la plataforma virtual `noel-srg`.
   - Se utiliza el script `runQEMU`, que automatiza la invocación de QEMU con los parámetros necesarios (`-M noel-srg`, `-nographic`, entre otros).
   - La salida de ejecución se redirige a logs, posteriormente analizados con `grep` para verificar la aparición de los mensajes de éxito.
6. **Validación de resultados:**
   - Cada prueba busca la cadena `PASSED` en su log correspondiente. En caso de no hallarse, la acción falla y el pipeline se detiene.

Importancia de la automatización

La adopción de un entorno de integración continua en proyectos de emulación embebida presenta ventajas significativas:

- Permite identificar errores en fases tempranas, reduciendo el impacto de defectos en etapas avanzadas del desarrollo.
- Asegura que nuevas modificaciones en QEMU o en los binarios de prueba no introduzcan regresiones funcionales.
- Favorece la colaboración, al proporcionar verificación automática en cada contribución.
- Mejora la trazabilidad de fallos mediante un historial claro de ejecuciones exitosas y fallidas.

En línea con prácticas aplicadas en proyectos como los de CHIPS Alliance [20], [21] y la validación de kernels RISC-V [23], [24], este pipeline constituye un paso clave hacia procesos de desarrollo más profesionalizados en entornos embebidos. La verificación sistemática en plataformas virtuales permite acelerar el ciclo iterativo de desarrollo y disminuir la dependencia de hardware físico.

4.7 Conclusiones del capítulo

Las pruebas desarrolladas permitieron comprobar el correcto funcionamiento de la máquina virtual `noel-srg` y de sus periféricos emulados en QEMU. Cada test ejecutado validó un aspecto crítico del sistema: la comunicación serie mediante UART, la manipulación digital con GPIO y la gestión de interrupciones periódicas a través del temporizador.

La automatización de estas pruebas en un pipeline de integración continua aportó un marco de verificación confiable, capaz de detectar errores en forma temprana y de garantizar la estabilidad del proyecto en el tiempo.

En conjunto, los resultados confirman que la emulación implementada reproduce de manera fiel el comportamiento esperado por el software embebido, ofreciendo un entorno robusto y autónomo de validación que elimina la necesidad de hardware físico en las fases iniciales de desarrollo y pruebas.

Capítulo 5

Conclusiones y líneas futuras

5.1 Conclusiones generales

En este trabajo se ha demostrado que es posible extender de manera efectiva el emulador QEMU para modelar plataformas embebidas específicas basadas en RISC-V, replicando tanto la arquitectura del procesador como el comportamiento funcional de periféricos clave. A partir del desarrollo de la máquina virtual `noel-srg`, se logró emular un entorno compatible con GRLIB y NOEL-V, permitiendo la ejecución y validación de software embebido real.

Uno de los logros más significativos fue integrar esta plataforma emulada dentro de un flujo automatizado de pruebas mediante integración continua. Este enfoque, habitual en el desarrollo de software general, no siempre se aplica en el ámbito embebido debido a la fuerte dependencia del hardware físico. Sin embargo, el uso de QEMU como backend de pruebas permitió ejecutar tests funcionales reproducibles, detectar regresiones de forma temprana y mantener la estabilidad del entorno en cada iteración del código.

Además, el trabajo puso en valor el potencial del software libre y abierto en el desarrollo embebido. La posibilidad de estudiar, modificar y extender QEMU resultó fundamental para alcanzar los objetivos del proyecto, al tiempo que se alinea con buenas prácticas de transparencia, reutilización y colaboración dentro de la comunidad técnica.

En síntesis, se ha desarrollado una herramienta funcional, extensible y automatizable que reduce la necesidad de hardware durante las etapas tempranas del desarrollo embebido. Este modelo no solo contribuye a mejorar la eficiencia en entornos profesionales, sino que también abre nuevas posibilidades para la enseñanza y la experimentación en contextos académicos.

5.2 Valoración crítica y aportes del trabajo

Desde una perspectiva crítica, el desarrollo de este Trabajo de Fin de Grado permitió explorar con profundidad las posibilidades reales de la emulación como estrategia viable para el desarrollo de sistemas embebidos, particularmente en contextos donde el acceso a hardware físico es limitado o costoso. El trabajo no solo validó técnicamente esta hipótesis, sino que también evidenció las dificultades prácticas asociadas al modelado preciso de periféricos y a la integración de sistemas virtuales con herramientas de testing automatizado.

Uno de los principales aportes del trabajo es la creación de una máquina virtual funcional y extensible basada en QEMU que replica una arquitectura inspirada en el procesador NOEL-V de Gaisler. Esta implementación permite ejecutar software real, escrito en C y compilado con la toolchain oficial de la compañía, validando el funcionamiento de periféricos como UART, GPIO y temporizadores. Al mantener compatibilidad con las estructuras internas del proyecto QEMU, se garantiza que el trabajo pueda ser mantenido, comprendido y extendido por otros desarrolladores.

Otro aporte significativo radica en la incorporación de pruebas funcionales automatizadas dentro de un entorno de integración continua (CI) mediante GitHub Actions. Esto demuestra que es posible llevar las buenas prácticas del desarrollo de software —como el testing temprano, la verificación reproducible y el análisis de regresiones— al mundo del software embebido, habitualmente más reacio a estas metodologías por su dependencia del hardware físico.

En términos metodológicos, el proyecto promueve una visión moderna del desarrollo de sistemas embebidos: orientada a la reproducibilidad, centrada en herramientas libres y abierta a la colaboración. El uso de QEMU como base demuestra que el software libre no solo es útil para el aprendizaje, sino también capaz de soportar desarrollos complejos y profesionales.

Finalmente, el proyecto ofrece una base técnica reutilizable para futuras implementaciones. Gracias a su diseño modular, la plataforma puede ser adaptada a otras arquitecturas basadas en GRLIB, facilitando el desarrollo de nuevas pruebas, periféricos o incluso el estudio comparativo entre configuraciones alternativas.

5.3 Limitaciones y aspectos mejorables

A pesar de los resultados obtenidos y de la funcionalidad alcanzada por la plataforma emulada, es importante destacar ciertas limitaciones del trabajo que podrían ser abordadas en desarrollos futuros:

- **Cobertura parcial de periféricos:** Si bien se implementaron y validaron periféricos esenciales como UART, GPIO y temporizador, la máquina virtual no emula todavía todos los dispositivos disponibles en la biblioteca GRLIB. Esto limita la capacidad de ejecutar software embebido más complejo que requiera, por ejemplo, controladores de interrupciones más avanzados, buses específicos o periféricos de comunicación como SPI o I2C.

- **Ausencia de verificación temporal precisa:** Al tratarse de una emulación funcional, el modelo no incluye un análisis exhaustivo del comportamiento temporal del sistema. Esto implica que el entorno es adecuado para validar lógica de software, pero no es aplicable a análisis de tiempo real o de rendimiento determinista, algo crítico en ciertos dominios embebidos.

- **Modelo estático de hardware:** La máquina `noel-srg` fue diseñada con una configuración fija de dispositivos. Sería deseable, a futuro, implementar una parametrización que permita generar dinámicamente instancias de hardware personalizadas desde archivos de configuración, acercándose a una arquitectura más genérica y reutilizable.

- **Interfaz limitada de observación y depuración:** Aunque se utilizaron logs y herramientas como `gdb` para validar el comportamiento del sistema, no se desarrollaron herramientas específicas de visualización o trazabilidad que faciliten el análisis en tiempo de ejecución.

- **Complejidad del entorno de integración continua:** Si bien se logró automatizar la validación funcional mediante GitHub Actions, este pipeline requiere recursos computacionales relativamente

elevados para compilar QEMU desde cero, lo cual puede dificultar su uso en entornos con restricciones de tiempo o hardware.

Estas limitaciones no invalidan los logros alcanzados, pero sí indican áreas donde sería valioso profundizar y robustecer la solución, especialmente si se busca escalarla a entornos industriales o integrarla en proyectos de mayor envergadura.

5.4 Líneas de trabajo futuras

El presente trabajo sienta las bases para el desarrollo de entornos de validación virtuales en sistemas embebidos, pero abre a su vez diversas posibilidades de mejora y expansión. A continuación, se detallan algunas líneas de trabajo que podrían abordarse en el futuro:

- **Ampliación del conjunto de periféricos:** Aunque se ha logrado la emulación funcional de UART, GPIO y temporizadores, el ecosistema GRLIB incluye una variedad de periféricos adicionales (SPI, I2C, PWM, etc.) cuya integración permitiría cubrir un espectro más amplio de aplicaciones embebidas.
- **Soporte de interrupciones complejas y DMA:** Incorporar sistemas de interrupciones más realistas y mecanismos de acceso directo a memoria (DMA) permitiría modelar comportamientos más próximos al hardware real.
- **Extensión a arquitecturas multinúcleo:** Actualmente el entorno está orientado a una CPU simple. Adaptarlo para simular procesadores multinúcleo (como los disponibles en versiones avanzadas de NOEL-V o VexRiscv) abriría la puerta a estudios sobre concurrencia, sincronización y escalabilidad.
- **Automatización avanzada del testing:** Si bien el sistema actual permite validar funcionalmente el entorno mediante integración continua, sería deseable incorporar pruebas de cobertura, tests de estrés y análisis de regresión para robustecer el ciclo de pruebas.
- **Pruebas comparativas con hardware real:** Ejecutar los mismos binarios en un entorno real fisico basado en NOEL-V y comparar los resultados con la emulación ayudaría a evaluar el grado de fidelidad del entorno virtual y detectar posibles inconsistencias.
- **Contribución oficial al repositorio de QEMU:** Mejorar la documentación, estilo y mantenibilidad del código implementado con el objetivo de presentar un *pull request* al repositorio oficial de QEMU. Esto favorecería su adopción por parte de la comunidad y garantizaría su mantenimiento a largo plazo.
- **Documentación ampliada y guía para nuevos desarrolladores:** Una mejora en la documentación técnica del entorno, incluyendo ejemplos, diagramas y guías paso a paso, facilitaría su adopción por parte de nuevos usuarios o instituciones educativas.

Estas posibles líneas de desarrollo buscan aprovechar la modularidad del sistema creado y su integración con herramientas abiertas, extendiendo su utilidad en ámbitos académicos, industriales y colaborativos.

El código fuente desarrollado, junto con los scripts de automatización y binarios de prueba, se encuentra disponible en los repositorios listados en el Apéndice A, lo que facilita su reutilización y extensión por parte de terceros.

Bibliografía

[1] W. M. Zabołotny, "Using QEMU-based co-simulation for embedded systems verification", en *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, IEEE, 2021, págs. 157-162. DOI: 10.1109/DDECS52668.2021.9417080.

[2] F. Bellard y the QEMU Project, *QEMU: The Fast Processor Emulator*, Accedido: 2025-08-18, 2023. dirección: https://www.qemu.org/.

[3] M. Cinque, D. Cotroneo et al., "Assessing the use of QEMU for dependability evaluation of critical systems", en *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2021, págs. 229-236. DOI: 10.1109/ISSREW53611.2021.00060.

[4] GitHub, Inc., *GitHub Actions Documentation*, Accedido: 2025-08-18, 2025. dirección: https://docs.github.com/en/actions.

[5] A. Waterman y K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA", 2011. dirección: https://riscv.org/specifications/.

[6] C. Gaisler, *NOEL-V Processor User Manual*, Accedido: 2025-08-18, 2022. dirección: https://www.gaisler.com/products/noel-v/.

[7] C. Gaisler, *GRLIB IP Core User's Manual*, Accedido: 2025-08-18, 2022. dirección: https://www.gaisler.com/products/grlib/.

[8] Wikipedia contributors, *QEMU*, Wikipedia, Accedido: 2025-08-18, 2025. dirección: https://en.wikipedia.org/wiki/QEMU.

[9] Wikipedia contributors, *RISC-V*, Wikipedia, Accedido: 2025-08-18, 2025. dirección: https://en.wikipedia.org/wiki/RISC-V.

[10] X. Guo y R. Mullins, "Fast TLB Simulation for RISC-V Systems", *arXiv preprint arXiv:1905.06825*, 2019.

[11] P. Vizcaíno, F. Mantovani, J. Labarta y R. Ferrer, "RAVE: RISC-V Analyzer of Vector Executions, a QEMU tracing plugin", *arXiv preprint arXiv:2409.13639*, 2024.

[12] Wikipedia contributors, *Virtualization*, Wikipedia, Accedido: 2025-08-18, 2025. dirección: https://en.wikipedia.org/wiki/Virtualization.

[13] E. S. D. Module, *Simulation vs Emulation*, Gitbook, Accedido: 2025-08-18, 2025. dirección: https://zshujon.gitbook.io/embedded-system-design/module-06.

[14] E. by PatSnap, *Emulation vs virtualization: Theoretical boundaries and implementations*, Accedido: 2025-08-18, 2025. dirección: https://eureka.patsnap.com/article/emulation-vs-virtualization-theoretical-boundaries-and-implementations.

[15] G. Delbergue, M. Burton, F. Konrad, B. Le Gal y C. Jego, "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0", en *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Disponible en HAL, 2016.

[16] W. M. Zabołotny, "QEMU-based hardware/software co-development for DAQ systems", *Journal of Instrumentation*, 2022, Preprint available as arXiv:2109.14735. DOI: 10.1088/1748-0221/17/04/C04004.

[17] M. A. Mehmood, Q. U. Ain, A. Akram, A. Qadeer y A. Waheed, "Emulating an Octeon MIPS64 based embedded system on x86 in QEMU", en *2016 19th International Multi-Topic Conference (INMIC)*, IEEE, 2016, págs. 1-7.

[18] M. Jiang, L. Ma, Y. Zhou et al., "ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels", *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021. DOI: 10.1145/3460120.3484753.

[19] G. O. Osman, "Emulating the Internet of Things with QEMU", Master's thesis; evaluation of QEMU platform for nRF51, including execution of Zephyr and performance comparison with hardware, Tesis de mtría., Chalmers University of Technology, 2020.

[20] *Open source and CI-driven RTL testing and verification for Caliptra's VeeR*, Chips Alliance blog post, 2023.

[21] K. Gugala y M. Cockrell, "Enabling Collaborative Chip Design in the RISC-V VeeR Core and Caliptra RoT Project with CHIPS Alliance tools", en *RISC-V Summit Europe*, Presentation with details on CI infrastructure for VeeR, 2023.

[22] C. Team, *Automated Kernel Testing on RISC-V Hardware*, Codethink blog post, 2023.

[23] C. Team, *Testing in a Box: Streamlining Embedded Systems Testing*, Codethink blog post, 2023.

[24] RISE Project, *Working with Igalia to improve RISC-V LLVM Continuous Integration*, Rise Project blog, 2024.

[25] F. S. Foundation, *GDB: The GNU Project Debugger*, Accedido: 2025-08-18, 2023. dirección: https://www.gnu.org/software/gdb/.

[26] T. G. D. Community, *Git Documentation*, Accedido: 2025-08-18, 2023. dirección: https://git-scm.com/doc.

[27] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator", en *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, USENIX Association, 2005, págs. 41-46. dirección: https://www.usenix.org/legacy/events/usenix05/tech/freenix/bellard.html.

[28] *QEMU Developer Documentation*, Disponible en https://www.qemu.org/docs/master/devel/, QEMU Project, 2024.

[29] P. Maydell y A. Liguori, "QOM: an object model for QEMU", en *KVM Forum*, 2012.

[30] Q. Project, *QEMU Coding Style*, Disponible en https://qemu-project.gitlab.io/qemu/devel/style.html, 2024.

[31] L. K. Community, *Linux Kernel Coding Style*, Disponible en https://www.kernel.org/doc/html/latest/process/coding-style.html, 2023.

[32] Q. Project, *Submitting a Patch*, Disponible en https://www.qemu.org/docs/master/devel/submitting-a-patch.html, 2024.

[33] Q. Project, *System Emulation — QEMU Documentation*, Disponible en https://www.qemu.org/docs/master/system/index.html, 2025.

[34] Q. Project, *'virt' Generic Virtual Platform (virt) — QEMU RISC-V System Emulator*, Disponible en https://www.qemu.org/docs/master/system/riscv/virt.html, 2025.

[35] QEMU Project Contributors, *QEMU: GRLIB APBUART Device Implementation*, https://gitlab.boonchuy.com/macemu/qemu/-/blob/56a60dd6d619877e9957ba06b92d2f276e3c229d/hw/grlib_apbuart.c, 2023.

[36] Q. Project, *RISC-V 'virt' Machine Devices — QEMU Documentation*, 2025.

[37] QEMU Project Contributors, *QEMU ACLINT Device Implementation (riscv_aclint.c)*, https://gitlab.com/qemu-project/qemu/-/blob/master/hw/intc/riscv_aclint.c, 2025.

Apéndice A

Repositorios del proyecto

Con el objetivo de garantizar la transparencia y la reproducibilidad de los resultados, se ponen a disposición pública los repositorios que contienen el código fuente desarrollado y los programas de prueba utilizados en este Trabajo de Fin de Grado.

Listado de repositorios

- **Fork de QEMU con la máquina `noel-srg`:** https://github.com/FerminVerdolini/TFG-QemuNoelSrg Incluye las modificaciones realizadas al emulador QEMU para soportar la plataforma virtual `noel-srg`. Dentro de este mismo repositorio se encuentra la configuración de GitHub Actions, que permite la ejecución automática de las pruebas funcionales.
- **Repositorio de firmware y pruebas funcionales:** https://github.com/FerminVerdolini/TFG-FirmaweTests Contiene los binarios y código fuente de las, junto con los scripts necesarios para su compilación y ejecución en el entorno emulado.

Todos los repositorios se encuentran bajo licencia libre, con el fin de favorecer la reutilización, la colaboración y la extensión de este trabajo por parte de la comunidad.

Universidad de Alcalá de Henares
Escuela Politécnica Superior