Facultad de ciencias

Desarrollo de plataformas OpenRISC-V

(OpenRISC-V platform development)

Trabajo de Fin de Grado
para acceder al
Grado en Ingeniería Informática

Autor: David Gómez Ortiz
Director: Eugenio Villar Bonet
Curso 2021-2022

A Inés por todo su cariño y apoyo.

Resumen

Las tecnologías desarrolladas alrededor de RISC-V están creciendo cada vez más y por tanto es probable que en el futuro se encuentren sistemas basados en esta arquitectura que resulten ser equiparables a los sistemas actuales, tanto en el ámbito de los sistemas de propósito general como en aplicaciones más específicas.

En el presente trabajo se introducen varios conceptos importantes relacionados con la teconología FPGA y su relación con los sistemas basados en RISC-V. Actualmente, estas tecnologías cuentan con los recursos necesarios para construir un sistema de propósito general capaz de ejecutarse en un dispositivo de bajas prestaciones. Se expondrán también varias plataformas que desarrollan *frameworks* de código abierto orientados al diseño de hardware basado en la arquitectura RISC-V.

Se aplicarán parte de los conocimientos obtenidos para realizar la instalación del sistema operativo Linux en un dispositivo virtual emulado con QEMU sobre la arquitectura RISC-V. Más tarde se ejecutará un programa escrito en el lenguaje de programación C compilado para RISC-V sobre este dispositivo virtual para comprobar su correcto funcionamiento. El mismo proceso será realizado en un sistema Linux RISC-V sobre una placa Nexys A7 real programada con un SoC basado en RISC-V, lo que nos permitirá exponiener la viabilidad de esta arquitectura para soportar un sistema de propósito general en un dispositivo de bajas prestaciones.

Palabras clave: RISC-V, FPGA, Linux, Sistemas Embebidos, SoC.

Abstract

The technologies developed around RISC-V are growing more and more and therefore it is likely that in the future systems based on this architecture will be found and proved to be comparable to current existing systems, both in the field of general purpose systems and in more specific applications.

The present work introduces several important concepts related to FPGA technology and its relationship to RISC-V based systems. Currently, these technologies have the necessary resources to build a general purpose system capable of running on a low performance device. Several platforms that develop open source frameworks oriented to the design of hardware based on RISC-V architecture will also be presented.

Part of the knowledge obtained will be applied to install the Linux operating system on a virtual device emulated with QEMU on the RISC-V architecture. Later, a program written in C programming language compiled for RISC-V will be executed on this virtual device to check its correct operation. The same process will be performed on a RISC-V Linux system on a real Nexys A7 board programmed with a RISC-V based SoC, which will allow us to expose the feasibility of this architecture to support a general purpose system on a low performance device.

Keywords: RISC-V, FPGA, Linux, Embedded Systems, SoC.

Índice general

Índice de figuras

1. Introducción

En 2010 se presentó una ISA denominada RISC-V debido a que pertenece a la quinta generación de proyectos de desarrollo de una arquitectura RISC de código abierto. RISC-V destaca principalmente por ser una ISA abierta que pretende convertirse en una ISA universal, dando lugar a un ecosistema libre de licencias para el desarrollo de dispositivos semiconductores. Actualmente, la rama de tecnologías pertenecientes al *open souce* o «código libre» cuenta con gran variedad de programas y distribuciones de sistemas operativos. Al tratarse también de una tecnología libre, RISC-V completa esta rama proporcionando el diseño de las capas más bajas de un sistema informático de forma que posibilita un paradigma donde todos los componentes clave de los sistemas informáticos son libres, desde el diseño del procesador hasta el sistema operativo y sus aplicaciones.

1.1. Motivación

Desde la aparición en 2010 de la ISA de código abierto RISC-V se han sumado los esfuerzos de la comunidad del *open hardware* o «hardware libre» para desarrollar tecnología, tanto software como hardware, que soporte esta arquitectura.

El trabajo presente toma como base el trabajo realizado por Laura Saiz Orza sobre la Implementación de un procesador RISC-V sobre FPGA ZCU102[1]. En dicho trabajo se presenta el proceso de instalación de un SoC basado en la arquitectura de RISC-V capaz de ejecutar un programa escrito en el lenguaje de programación C. Este trabajo tiene como objetivo complementar el desarrollo de un sistema sobre FPGA basado en RISC-V añadiéndole la instalación de un sistema operativo de código abierto como es Linux.

1.2. Objetivos

Este trabajo se propone estudiar las tecnologías relacionadas con RISC-V y las FPGA, los SoC y las plataformas que se encuentran actualmente disponibles, tanto software como hardware, que dan soporte al diseño de «sistemas en chip» basados en RISC-V. Para ello, en primer lugar se introducirán varios conceptos importantes relacionados con RISC-V, FPGA y Linux, para luego aplicar los conocimientos obtenidos realizando una instalación del sistema operativo Linux en un sitema basado en la arquitectura RISC-V, primero mediante la virtualización de una plataforma hardware genérica y más tarde sobre un dispositivo FPGA físico programado con un SoC basado en RISC-V. Para ello se seguirán los siguientes pasos:

- Estudio de la ISA RISC-V como ISA modular y abierta.
- El sistema operativo Linux, los gestores de arranque compatibles con la arquitectura RISC-V y el núcleo de Linux.
- Estudio de la tecnología FPGA, sus características y sus ventajas respecto a los dispositivos embebidos ensamblados.
- Estudio de las plataformas de diseño de hardware libre basadas en RISC-V.
- Instalación de un Linux en una máquina virtual basada en RISC-V.

- Instalación de un Linux en un dispositivo FPGA físico programado con un SoC basado en RISC-V.

2. Instruction Set Architecture (ISA)

Una *Instruction Set Architecture* (ISA) es parte del modelo abstracto de un computador que define cómo la CPU es controlada por el Software[2], define el formato de las instrucciones admitidas por la CPU determinando el orden de los registros, los identificadores de la operación, los tipos de datos aceptados, etc. Las ISAs pueden ser extendidas añadiendo nuevas instrucciones o soportando instrucciones más largas, esto es, de mayor número de bits. Las ISAs actuales se dividen principalmente en dos grandes familias, CISC y RISC.

La ISA CISC (*Complex Instruction Set Computer*) se caracteriza por tener un conjunto de instrucciones muy amplio que permite realizar operaciones con elementos en memoria y en los registros[3]. Una ISA con más instrucciones supone más funcionalidades para las máquinas que la implemente, pero a la vez añade complejidad al sistema y dificulta su mantenibilidad.

Los sistemas RISC (*Reduced Instruction Set Computer*) tienen dos características fundamentales: todas sus instrucciones tienen el mismo tamaño y tienen un número reducido de formatos y sólo las intrucciones de *load* y *store* acceden a memoria de datos[4]. Lás máquinas basadas en RISC suelen precisar de menos hardware que las basadas en RISC y tienen mayor flexibilidad en su diseño. Los sistemas RISC suelen ser sistemas de bajo consumo, por lo que esta ISA es una buena candidata para los sistemas embebidos.

2.1. RISC-V

La arquitectura RISC-V nace en la Universidad de Berkeley en 2010 con el objetivo de crear una ISA que no dependa de una gran multinacional. En 2015 se fundó la RISC-V Foundation con 29 miembros para centralizar el desarrollo de la ISA. Actualmente, es una organización con más de dos mil miembros provenientes de más de 70 países. El objetivo de RISC-V es convertirse en una ISA universal, lo que condiciona enormemente su desarrollo como sistema persistente, escalable y mantenible.

La ISA RISC-V recibe este nombre al ser parte de la quinta generación de proyectos de desarrollo de una arquitectura basada en RISC universal de código abierto. Esto permite el diseño y distribución de chips y software de RISC-V sin necesidad de pagar licencias, además de posibilitar el desarrollo colaborativo por parte de la comunidad del *open hardware* o hardware libre. Sin embargo, RISC-V no es la primera arquitectura de código abierto. Otros proyectos predecesores a esta ISA como RISC DLX o OpenRISC también son arquitecturas de dominio público, pero no llegaron a triunfar en el mercado. Lo que hace destacar a RISC-V y es indudablemente la clave para su éxito hasta la fecha es su diseño modular, que hace a esta arquitectura apropiada para distintos tipos de implementaciones, desde sistemas de propósito general hasta sistemas más específicos.

Esta ISA cuenta con conjuntos de instrucciones básicos (RV32-I, RV64-I, RV128-I) que se componen de instrucciones para operaciones con enteros y direccionamiento de 32, 64 y 128 bits respectivamente, aunque RISCV128 aún se considera en desarrollo experimental. La arquitectura RISC-V también cuenta con el conjunto base RV32-E, que tiene 16 registros, el conjunto base RV32-I, por otra parte, cuenta con un total de 32.

Estos conjuntos base solo implementan las operaciones con enteros, y no implementan multiplicaciones ni divisiones. Para añadir más funcionalidades a la arquitectura los conjuntos

base son expandibles con distintos módulos opcionales. Estos módulos son prescindibles, de forma que es posible reducir la complejidad de un sistema en función de sus requisitos. Los módulos pueden ser identificados mediante una letra mayúscula. El estándar es colocar estas letras detrás del conjunto base. Por ejemplo, un sistema que implemente RISC-V de 32 bits y además añada soporte para operaciones de multiplicación y operaciones de números de coma flotante se indica con RV32-IMF. Entre los principales módulos de RISC-V se encuentran:

- M: Instrucciones adicionales para multiplicar y dividir enteros.
- A: Operaciones atómicas con memoria.
- F: Soporte para coma flotante de 32 bits.
- D: Soporte de coma flotante de 64 bits.
- Q: Soporte de coma flotante de 128 bits.
- C: Permite trabajar con datos comprimidos.

La arquitectura RISC-V también permite crear instrucciones específicas y añadirlas al procesador, con el único requisito de que se mantenga el formato de instrucción utilizado en la ISA base.

2.1.1. Los modos de ejecución en RISC-V

Los modos de ejecución definen distintos perfiles bajo los que el software trabaja con distintos niveles de privilegios. El estandar de RISC-V especifica tres modos de ejecución principales. Estos son:

1. Modo usuario: En este modo de ejecución se tiene acceso a instrucciones privilegiadas y se lanza una excepción cuando un programa en modo usuario intenta realizar una instrucción en modo máquina. No permite el acceso a dispositivos de entrada y salida, a la memoria del núcleo o a otros procesos.
2. Modo supervisor: Este modo de ejecución es implementado pricipalmente para soportar sistemas operativos como Linux ya que es el modo en el que trabaja principalmente el núcleo de Linux. Para implementar este modo de ejecución es necesario un sistema de virtualización de memoria. Se encuentra en un punto medio entre el modo usuario y el modo máquina.
3. Modo máquina: Este modo de ejecución no cuenta con protección de memoria ya se ejecuta directamente sobre el hardware. Se trata del modo de ejecución más privilegiado, en el que los *harts* tienen acceso a memoria, dispositivos de entrada y salida y funcionalidades necesarias para configurar el sistema. Este modo de ejecución se implementa en todas los procesadores RISC-V.

2.2. x86 y ARM

Actualmente el mercado de procesadores está dominado por dos grandes arquitecturas: x86 y ARM. Estas arquitecturas no son abiertas, lo que supone que fabricar procesadores basados en ellas implica pagar una licencia, haciendo sus diseños menos accesibles al público y dejando el mercado de los procesadores en manos de las empresas que paguen dichas licencias.

La arquitectura ARM está basada en RISC por lo que requiere de menos transistores que un procesador con una arquitectura basada en CISC como x86. Un menor número de transistores equivale a un consumo y una generación de calor reducida, lo que hace a ARM un buen candidato para sistemas de bajo consumo, como sistemas embebidos o de relativo bajo coste. Los procesadores de ARM se han convertido en en los dominantes dentro del mercado de la electrónica móvil e integrada, encarnados en microprocesadores y microcontroladores pequeños de bajo consumo y relativamente bajo costo[5].

La familia x86 reagrupa los microprocesadores compatibles con el juego de instrucciones Intel 8086. Por tanto, x86 representa a ese conjunto de instrucciones, siendo también una denominación genérica dada a los correspondientes microprocesadores[6]. La ISA x86 es una arquitectura basada en CISC, por lo que los procesadores de esta arquitectura son más complejos y completos, añadiendo más características y funciones al procesador a cambio de más consumo y más elementos hardware. Esta arquitectura tiene mayor presencia en equipos de propósito general como los ordenadores personales de uso diario, donde los principales fabricantes de procesadores, Intel y AMD, la implementan en sus productos.

3. GNU/Linux

GNU/Linux[7] es un sistema operativo «tipo Unix», compuesto por software libre y de código abierto. Es uno de los sistemas operativos más populares y es utilizado en sistemas de sobremesa, servidores, sistemas embebidos y hasta en supercomputación, debido a su seguridad, eficiencia y fiabilidad[8]. Al ser un software de código abierto no es necesario pagar licencias para instalar, redistribuir o lanzar nuevas distribuciones de Linux.

Linux cuenta con una comunidad muy grande que lo mantiene y desarrolla: La Linux Foundation, fundada en 2007, es una agrupación sin ánimo de lucro que se dedica a incentivar el desarrollo de Linux, llevado a cabo de forma descentralizada a través de la comunidad, asegurándose de que Linux permanece gratuito y tecnológicamente avanzado.

3.1. Componentes de Linux

Como se ha mencionado anteriormente, Linux se compone de varios elementos de software libre de entre los cuales los principales para formar un sistema Linux son:

- El cargador de arranque o *Bootloader*: Es un software encargado del arranque del sistema.
- El núcleo o *Kernel*: Es el núcleo del sistema. Administra los componentes hardware y es el nivel más bajo del sistema operativo.
- El sistema inicial o *Init System*: Se encarga de iniciar el espacio de usuario, el cual es controlado por los *daemons*.
- Los *Daemons*: Son servicios iniciados en el proceso de arranque o tras iniciar sesión en el sistema.
- El servidor gráfico: Es un subsistema que visualiza gráficos en la pantalla.
- El entorno de escritorio: Entorno con el que interactúa el usuario. Cada entorno de escritorio puede contar con varias aplicaciones instaladas de serie. No todos los sistemas Linux cuentna con un entorno de escritorio, con algunos el usuario debe interactuar únicamente a través de una terminal.
- Las aplicaciones: Son el programa o conjunto de programas informáticos que realizan un trabajo específico, diseñado para el beneficio del usuario final.

Linux cuenta con gran variedad de dsitribuciones que implementan distintos elementos de software. Los elementos principales del sistema operativo Linux son generalmente comunes, aunque hay muchas distribuciones de Linux que incluyen diferentes opciones de software. Esto significa que Linux es en gran medida personalizable, de forma que es posible instalar un sistema Linux mínimo e ir añadiendo elementos necesarios según sean necesarios. Los usuarios también pueden elegir los componentes principales, como el servidor gráfico y otros componentes de la interfaz de usuario.

Linux es utilizado en los sistemas embebidos debido a que puede adaptarse su núcleo en función de los requisitos del sistema subyacente, generalmente un sistema con menor tamaño, que requiera de menor potencia de procesamiento y tenga características mínimas.

3.2. El bootloader

El *bootloader* o cargador de arranque es un software que se ejecuta antes que el sistema. En la primera etapa del arranque del sistema carga el kernel en memoria para que el sistema pueda iniciarse. El *bootloader* suele permitir modificaciones en el entorno o en el núcleo antes de arrancarlo.

Existen distintos *bootloaders* como GRUB, cargador de arranque múltiple utilizado con varios sistemas operativos, o LILO, utilizado en sistemas Linux. En este trabajo se verán el *Berkeley Boot Loader* (BBL) y U-boot.

3.2.1. *Berkeley Boot Loader*

El *Berkeley Boot Loader* o BBL nace en la Universidad de Berkeley. Se espera que el BBL haya sido cargado en cadena desde otro gestor de arranque con el punto de entrada ejecutándose en modo máquina. El *Berkeley Boot Loader* transiciona del modo de ejecución máquina al modo supervisor, se le pasa un árbol de dispositivos desde la etapa anterior donde se registran los componentes del sistema.

El *Berkeley Boot Loader* se encarga de arrancar el núcleo de Linux y de gestionar instrucciones que el modo usuario de RISC-V no puede gestionar a nivel hardware. Establece una memoria virtual y un controlador de interrupciones, carga el núcleo embebido en el *bootloader* como binario (*payload*) y lo ejectua en modo supervisor[9].

Actualmente el BBL se considera obsoleto debido a que RISC-V ha establecido como estándar el uso de OpenSBI y U-Boot como gestores de arranque de primera y segunda etapa.

3.2.2. U-Boot y OpenSBI

U-boot es un gestor de arranque para Linux, pensado para utilizarse en dispositivos embebidos, los cuales deben contar con una configuración en los ficheros de U-Boot para ser soportados. Su estructura se basa en imágenes binarias, las cuales carga y ejectua por medio de comandos y variables de entorno. U-boot utiliza distintos tipos de imágenes, donde cada imagen contiene un «número mágico» que identifica el tipo de imagen y la plataforma a la que va destinada, direcciones de carga y más información relevante. U-boot puede ser utilizado en sistemas basados en RISC-V, pero es necesario realizar un portado específico a la plataforma, ya que es muy dependiente del hardware.

OpenSBI es una implementación de código abierto de las especificaciones de RISC-V *Supervisor Binary Interface* (SBI). Su función es proveer servicios en tiempo de ejecución en modo máquina para gestionar instrucciones sobre hardware que RISC-V o Linux no puedan gestionar. También se trata de un programa muy dependiente del hardware y necesita una configuración en función de las características del hardware sobre el que se ejecuta.

3.3. El kernel de Linux

El núcleo o *kernel* es el componente más básico de un sistema operativo. En él se incorporan todo el software fundamental para el funcionamiento y el control del sistema. Su estructura puede serpararse en módulos orientados a sus distintas funcionalidades, como son la gestión de memoria, la gestión de ficheros, el control del acceso a los dispositivos de entrada y salida, las llamadas al sistema y la gestión de procesos, así como la comunicación entre ellos.

El kernel de Linux tiene cumple con el POSIX (Portable Operating Systems Interface for Unix) de IEEE, por lo que la mayoría de aplicaciones para Unix pueden compilarse y ejecutarse sin necesidad de modificar el código fuente[10].

Se pueden diferenciar dos tipos de núcleos en función de su modularidad y del orden de compilación de sus módulos:

- Núcleo monolítico: El núcleo monolítico se puede concebir como un gran «programa» que se ejecuta en un mismo espacio de direccionamiento. La comunicación entre procesos en este núcleo es trivial debido a esto. El inconveniente de este tipo de núcleos es que para añadir nuevas funcionalidades, debe ser recompilado de nuevo en su totalidad. Además, al compartir todos los los procesos el mismo espacio de direcciones, un error en un proceso podría propagarse al sistema entero.
- Micronúcleo ($\mu kernel$): Este núcleo contiene los elementos mínimos para realizar los servicios básicos de un *kernel*, como algunas llamadas al sistema, planificación básica, comunicación entre procesos y la gestión del espacio de direcciones. El resto de servicios se ejecutan como procesos denominados «servidores» en el espacio de usuario.

3.4. Linux en sistemas embebidos

El uso de Linux en sistemas embebidos permite la ejecución de programas complejos que requieran de gestión de memoria, control de acceso a dispositivos de entrada y salida o comunicación entre procesos. El uso de un sistema operativo permite al usuario abstraerse de la complejidad del hardware y centrarse en las tareas de programación de alto nivel.

Linux al ser un sistema operativo altamente configurable puede adaptarse para poder ser ejecutado en dispositivos de baja potencia, construyendo un sistema de bajas prestaciones capaz de realizar tareas complejas e implementando más funcionalidades. El uso de Linux en esta clase de sistemas se aportan múltiples proveedores de software, apoyo y soporte.

4. Field-Programable Gate Arrays (FPGA)

Una FPGA (*Field-Programable Gate Arrays*) es un dispositivo formado por bloques lógicos configurables (CLB) dispuestos en forma de matriz interconectados mediante unas conexiones programables como se muestra en la figura 1. Estos bloques configurables y sus conexiones pueden ser programados mediante un lenguaje de descripción especializado o HDL (*Hardware Description Language*), pudiendo dar lugar a lógicas tan complejas como la de un procesador o tan simples como la de una puerta lógica.

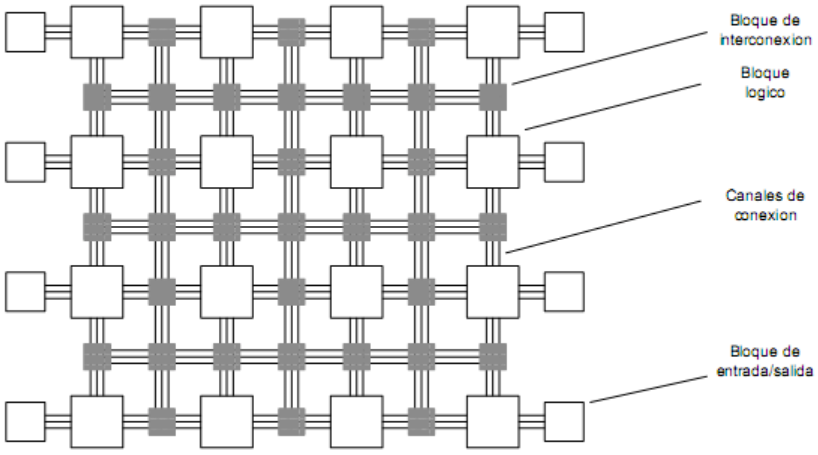

Figura 1: Interconexiones de bloques lógicos programables en una FPGA

Esta tecnología nace en 1984 inventada por Ross Freeman y Bernard Von Der Schmitt, cofundadores de Xilinx.[11] Actualmente Xilinx es uno de los líderes en la fabricación de FPGA. El diseño de las FPGA ha ido evolucionando desde su creación, consiguiendo modelos muy completos capaces de implementar funcionalidades de dispositivos ensamblados de forma eficiente. Los bloques lógicos de las FPGA no son necesariamente homogéneos, pudiendo haber bloques orientados a distintas funcionalidades como el procesado de operaciones o bancos de registros.

Se puede cambiar la funcionalidad del dispositivo FPGA de forma relativamente sencilla, realizando cambios en un fichero incluso de froma dinámica haciendo esta tecnología muy flexible. Puede programarse el comportamiento de distintos tipos de dispositivos que cambien alguna de sus características en función de las necesidades del sistema. Las FPGA reducen la circuitería de muchos dispositivos y son utilizadas en la industria en muchos ámbitos diferentes, como el aeroespacial, el médico[12] o en el de la computación de alto rendimiento.

Otra ventaja de las FPGA es que permiten implementar el comportamiento de dispositivos que se encuentran fuera del mercado, los cuales ya no se fabrican y que de otra forma sería muy complicado replicar. En este trabajo se ha valorado el uso de distintas placas con dispositivos FPGA, las cuales serán analizadas con más detalle en los apartados 8.1 y 8.2.

5. Lenguajes de descripción especializados

Un lenguaje de descripción especializado o lenguaje de descripción hardware (HDL) es una especificación de las conexiones de un cirucuito electrónico o más comunmente de un circuito lógico digital. Los lenguajes de descripción especializados pueden asimilarse a lenguajes de programación de alto nivel, dado que ambos consisten en declaraciones formales de expresiones y estructuras de control, pero tienen una principal diferencia: En los lenguajes de descripción hardware, se definen únicamente descripciones ejecutables en hardware. Esto hace que sea posible la simulación de los circuitos y su comportamiento sin necesidad de ser implementados físicamente. Los HDL más utilizados actualmente son Verilog y VHDL[13].

5.1. Generación de un bitstream

Un *bitstream* es un archivo binario utilizado para programar una FPGA. Este archivo se genera a partir de un código en HDL y es necesario para que la FPGA se comporte como una plataforma de hardware embebido.[14]

La generación de un *bitstream* no es sencilla. Son necesarias herramientas de diseño hardware como Xilinx Vivado que permitan la generación de estos archivos a partir de un código HDL. Para generar un *bitstream* es necesario pasar por varias etapas:

1. Diseño del código HDL. Es necesario partir de un código HDL que especifique el diseño del hardware.
2. Proceso de síntesis. En este proceso se genera una *netlist* que comprueba la conexión de los elementos hardware básicos y define que tipo de conexiones son necesarias.
3. Proceso de implementación y enrutado. En este proceso se asocian los elementos de la *netlist* generada anteriormente en el proceso de síntesis con los componentes físicos de la FPGA y se realiza el enrutado. Esto da lugar al fichero binario que anteriormente ha sido denominado como *bitstream*.

Estas etapas son realizadas de forma secuencial, ya que para comenzar una etapa es necesario el resultado de la etapa anterior, como se muestra en la figura 2. Una vez generado el bitstream, este puede usarse para configurar la FPGA. Normalmente se guarda en una memoria no volátil desde la que se puede volver a programar en cada encendido.

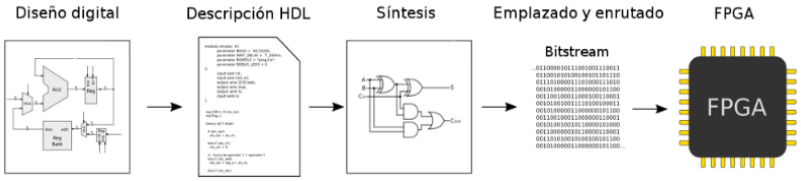

Figura 2: Etapas en la generación de un bitstream para la programación de una FPGA

Las FPGA son una tecnología cerrada y no se conoce el formato de los bitstream, de forma que no es fácil obtener uno si no es a través de alguna herramienta con esa funcionalidad.

Además, estas herramientas suelen estar restringidas por licencias que limitan el uso gratuito de sus funcionalidades.

En 2015 se inició un proyecto en el cual mediante ingeniería inversa se encontró la forma de generar algunos *bitstreams* para FPGA específicas sólo con herramientas *open source*, llamado *Project IceStorm*[15].

5.2. Vivado

Para la generación de un *bitstream* para la programación de una FPGA en este trabajo se utiliza el software de Vivado, creado por el fabricante de FPGA Xilinx en 2012. Este software se enfoca al diseño de dispositivos hardware y cuenta con variedad de herramientas para el diseño de SoC, como el integrador de IPs, con el cual es posible realizar un diseño modular utilizando distintos elementos ya validados y preconstruidos que actúan como «cajas negras», facilitando la tarea del programador, el cual no necesita comprender el funcionamiento interno de estas a un nivel muy detallado, permitiéndole centrarse en la composición del SoC y la interconexión de estas IPs.

Vivado contiene herramientas para la realización de todas las fases de generación de un bitstream explicadas anteriormente, siendo posible el diseño del código HDL en los lenguajes VHDL y Verilog. Vivado puede realizar a partir de este código las fases de síntesis e implementación. Para esto, Vivado cuenta con la configuración de varios dispositivos soportados para facilitar el diseño en dispositivos específicos.

Por último, Vivado genera un archivo binario como *bitstream* que puede ser utilizado para la programación de una FPGA. Vivado cuenta con la herramienta de «Manejador de Hardware», la cual será utilizada más adelante en este trabajo. Esta herramienta permite conectar de forma sencilla el dispositivo FPGA programable con el PC en el que se está trabajando, estableciendo una comunicación y cargando el *bitstream* en la FPGA.

6. Diseño de plataformas OpenRISC-V

Un sistema en chip o SoC consiste en usar tecnologías que integran todos o gran parte de los módulos que componen un computador o cualquier otro sistema informático o electrónico en un único circuito integrado o chip[16].

Un sistema en chip puede estar implementado por hardware, software o de forma mixta, tanto hardware como software. Es muy frecuente utilizar esta tecnología en sistemas embebidos debido a que se tratan de diseños específicos y más baratos que los procesadores convencionales y a su vez mucho más complejos que un microcontrolador. Además, estos sistemas pueden usarse junto a las FPGA para realizar implementaciones de procesadores o SoC que no se fabrican en el mercado.

Actualmente hay varias plataformas que trabajan activamente en el desarrollo de entornos y *frameworks* orientados al diseño de SoC basados en RISC-V. A continuación se explorarán las plataformas más relevantes.

6.1. PULP

El proyecto PULP (*Parallel Ultra Low Power Platform*) resulta de la colaboración entre el Laboratorio de Sistemas Integrados (IIS) de la Universidad ETH Zürich y el grupo de Sistemas Embebidos de Alta Eficiencia Energética (EEES) de la Universidad de Bolonia en 2013.

La plataforma PULP ha desarrollado implementaciones de código abierto eficientes de 32 y 64 bits basadas en el conjunto de instrucciones de RISC-V, desde periféricos hasta sistemas completos empezando empezando por simples microcontroladores hasta la versión de última generación OPENPULP, que establece un nuevo listón para los procesadores multinúcleo IoT de bajo consumo. Los procesadores desarrollados por la plataforma PULP son los siguientes:

- CV32E40P (RI5CY): Procesador de 32 bits y 4 etapas que implementa las instrucciones RV32-IMC. Opcionalmente puede implementar una FPU de 32 bits que proporcione soporte para las instrucciones de tipo F.
- Ibex (Zero-riscy): Un procesador de 32 bits y 2 etapas optimizado para aplicaciones de control que implementa las instrucciones RV32-IMC.
- CVA6 (Ariane): Procesador de 64 bits que implementa de forma completa las instrucciones de tipo I, M, C y D. Este procesador se encuentra en fase de desarrollo y cuenta con un proyecto de ampliación de privilegios que implementa los modos de ejecución de usuario, máquina y supervisor para soportar completamente un sistema operativo tipo Unix (Linux, BSD, etc.).

Los sistemas más simples de PULP están basados en microcontroladores que pueden ser configurados para manejar cualquiera de sus procesadores de 32 bits, como RI5CY o Zero-riscy. Versiones más avanzadas permiten el uso de aceleradores en el sistema. Estos microcontroladores son los siguientes:

- PULPino: Un SoC con un solo núcleo basado en RISC-V, el primer SoC de la plataforma que se ha hecho popular.

- PULPissimo: Una versión avanzada del microcontrolador, que implementa mejoras en la comunicación entre dispositivos mediante el uso de interconectores logarítmicos y un μDMA.

El anterior trabajo sobre el que se desarrolla este trabajo utiliza el microcontrolador PULPissimo implementado sobre una FPGA para la ejecución de código escrito en el lenguaje C.

6.2. ChipYard

Chipyard es una plataforma para el diseño y la evaluación de hardware de sistemas completos mediante equipos ágiles. Se compone de una colección de herramientas y bibliotecas diseñadas para proporcionar una integración entre herramientas de código abierto y comerciales para el desarrollo de sistemas en chip.

Chipyard es una plataforma de código abierto para el desarrollo ágil de sistemas en chip basados en Chisel, que permite aprovechar el HDL de Chisel, el generador de SoC Rocket Chip y otros proyectos de Berkeley para producir un SoC RISC-V con todo, desde periféricos mapeados por MMIO hasta aceleradores personalizados. Chipyard contiene núcleos de procesador (Rocket, BOOM, CVA6 (Ariane)), aceleradores (Hwacha, Gemmini, NVDLA), sistemas de memoria y periféricos y herramientas adicionales para ayudar a crear un SoC completo. Soporta múltiples flujos concurrentes de desarrollo ágil de hardware, incluyendo la simulación RTL de software, la simulación acelerada de FPGA con el simulador FireSim y la generación de cargas de trabajo de software para sistemas bare-metal y basados en Linux. Chipyard se desarrolla activamente en el Grupo de Investigación de Arquitectura de Berkeley del Departamento de Ingeniería Eléctrica y Ciencias de la Computación de la Universidad de California, Berkeley.

6.2.1. Generador de SoC Rocket Chip

Rocket Chip no es sólo un SoC concreto, sino que es un generador de códigl RTL sintetizable, es decir, es capaz de producir muchas instancias de diseños a partir de una única fuente de alto nivel. El generador de Rocket Chip está escrito en Chisel y construye una plataforma basada en RISC-V. Consiste en una colección de bibliotecas de construcción de chips parametrizadas que podemos utilizar para generar diferentes variantes de SoC. El generador Rocket Chip incluye los siguientes generadores de procesadores:

- Rocket Core: Rocket es un generador de núcleos escalares de 5 etapas que implementa las ISA RV32G y RV64G, donde G viene de *General*, indicando que se implementan las extensiones M, A, F y D. Cuenta con una MMU que admite memoria virtual basada en páginas, una caché de datos no bloqueante y un front-end con predicción de saltos.
- BOOM Core: BOOM es un generador de núcleos superescalares RV64G *out-of-order*. BOOM soporta la especulación de salto completa y utiliza una unidad agresiva de carga/almacenamiento que permite que las instrucciones de carga se ejecuten fuera de orden con respecto a los almacenamientos y otras cargas.

- Z-scale Core: Z-scale es un generador de núcleos de 32 bits dirigido a sistemas embebidos y aplicaciones de microcontroladores. Implementa la ISA RV32IM y está diseñado para interactuar con los buses AHB-Lite. Z-scale utiliza un pipeline de orden único de 3 etapas que soporta los modos de privilegio de máquina y de usuario.

6.3. GRLIB

En el ámbito de los SoC, una IP (*Intellectual Property*) es un bloque de circuitos prediseñado y debidamente testeado para el diseño completo de un dispositico semiconductor[17].

La librería GRLIB es una librería de IPs para el diseño de SoC. Se basa en núcleos IP conectados a un «bus en chip» común. Esta bilbioteca es independiente del proveedor y es compatible con distintas herramientas.

GRLIB ofrece muchos núcleos IP para el diseño de SoC incluyendo procesadores, Controlador SDRAM de 32 bits, PCI de 32 bits con DMA, MAC Ethernet 10/100/1000 Mbit, controlador PROM y SRAM de 8/16/32 bits, controladores DDR/DDR2 de 16/32/64 bits, controladores de dispositivos y host USB 2.0, SPI, I2C, UART con FIFO, controlador de interrupciones y un puerto GPIO de 32 bits.

Entre los procesadores incluídos se encuentra el NOEL-V un procesador que implementa el ISA de RISC-V. NOEL-V es un núcleo de procesador que puede configurarse para ajustarse a la arquitectura RISC-V de 32 bits (RV32) o de 64 bits (RV64). Está diseñado para aplicaciones embebidas, combinando un alto rendimiento con una baja complejidad y bajo consumo de energía. El bloque NOELVSYS combina el procesador NOEL-V con los periféricos esenciales necesarios para crear un sistema en chip basado en NOEL-V.

El núcleo NOEL-V ha sido desarrollado por la empresa CAES, dedicada a proporcionar el ecosistema completo para apoyar el diseño de hardware digital para soluciones *System-on-a-Chip* de misión crítica.

7. Virtualización de Linux en RISC-V

A continuación se explicará el proceso seguido para instalar un sistema operativo Linux en una placa virtual con una arquitectura RISC-V donde se ejecutará un programa sencillo. Este desarrollo consta de los siguientes pasos:

1. Instalación de la cadena de herramientas de RISC-V para la compilación cruzada de ficheros.
2. Compilación cruzada de un núcleo Linux para la arquitectura RISC-V.
3. Construcción y compilación de un bootloader compatible con QEMU para la arquitectura RISC-V.
4. Contrucción de un sistema de ficheros para el sistema operativo con la herramienta BusyBox.
5. Creación de un disco virtual para el almacenamiento del entorno del bootloader y del sistema de ficheros.
6. Arranque del sistema y ejecución de un programa compilado para RISC-V.

7.1. Cadena de herramientas de RISC-V

Un compilador cruzado es un software capaz de generar código ejecutable en otra plataforma distinta a la que se está ejecutando, es decir, un compilador cruzado nos permite generar código ejecutable para un sistema con arquitectura RISC-V desde un sistema con arquitectura x86. Estos compiladores no están integrados por defecto en Linux y, por tanto, es necesaria su instalación.

La RISC-V GNU Compiler Toolchain es una cadena de herramientas que permite compilar de forma cruzada código en los lenguajes de programación C o C++ para RISC-V. Esta cadena soporta dos modos de construcción: con una cadena de herramientas genérica ELF/Newlib o con una cadena de herramientas más sofisticada Linux-ELF/glibc. La RISC-V GNU Compiler Toolchain está disponible el repositorio de GitHub de RISC-V Software Collaboration. El proceso de instalación de la RISC-V GNU Compiler Toolchain se indica en el anexo I de este documento.

7.2. La herramienta de virtualización QEMU

QEMU es un virtualizador y emulador genérico de código abierto que puede ejecutar sistemas operativos y programas hechos para una máquina (por ejemplo, una placa ARM) en una otra diferente (por ejemplo, un PC con arquitectura x86). Al utilizar la traducción dinámica, consigue un buen rendimiento (amp). El traductor dinámico de QEMU realiza una conversión en tiempo de ejecución de las instrucciones de la CPU de destino al conjunto de instrucciones de la máquina donde se ejecuta. El código binario resultante se almacena en una caché para que pueda reutilizarse. La ventaja del traductor dinámico con respecto a un intérprete es que las instrucciones de destino se obtienen y decodifican una sola vez.

QEMU es utilizado para la emulación de una placa virtual basada en RISC-V sobre la que se pretende instalar y ejecutar el sistema operativo de Linux. Para ello se ha de instalar QEMU de la siguiente forma:

```
$ sudo apt install qemu-system-misc
```

Es posible comprobar los dispositivos basados en la arquitectura RISC-V de 64 bits soportados por QEMU al ejecutar la siguiente instrucción, lo que nos permite obtener la lista de dispositivos (figura 3).

```
qemu-system-riscv64 -M ?
```

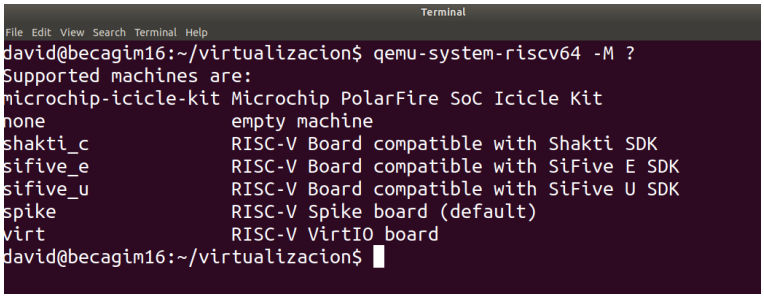

Figura 3: Máquinas que implementan RISCV64 soportadas por QEMU

El dispositivo «virt» no se corresponde con ningún hardware real y está diseñado para su uso en máquinas virtuales, por lo que su simulación es más rápida. Se utiliza virt en este trabajo debido a que no es necesario conocer la complejidad del hardware que emula.

Se utiliza QEMU para probar a ejecutar un programa compilado para RISC-V con la RISC-V GNU Compilation Toolchain. Para ello se usa un programa simple escrito en el lenguaje de programación C:

```c
#include <stdio.h>
int main() {
    // Say hello to the world!
    printf("Hello, World!\n");
    return 0;
}
```

El programa es compilado por la RISC-V GNU Compilation Toolchain y ejecutado con qemu de la siguiente manera. Tras la ejecución se obtiene la salida esperada, como se muestra en la figura 4.

```
$ riscv64-unknown-linux-gnu-gcc -static -o hello hello.c
$ qemu-riscv64 ./hello
```

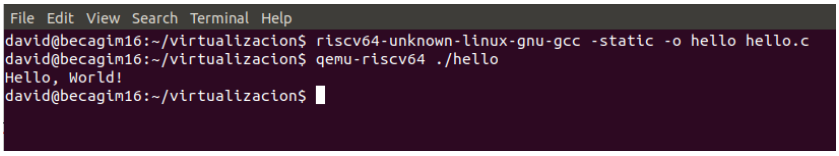

Figura 4: Salida del programa compilado «hello»

7.3. Construcción de un cargador de arranque con U-boot

El cargador de arranque utilizado es U-Boot. Se obtiene el código fuente de este programa a través de la plataforma GitHub. Para construir U-Boot es necesaria una configuración predefinida dependiente del dispositivo donde se va a ejecutar. Se pueden mostrar los dispositivos para los cuales U-Boot cuenta con esta configuración bajo la carpeta de «configs». Se muestran entonces las configuraciones predefinidas para la arquitectura RISC-V (figura 5).

```
$ git clone https://source.denx.de/u-boot/u-boot.git
$ cd u-boot
$ ls -la configs  grep riscv64
```

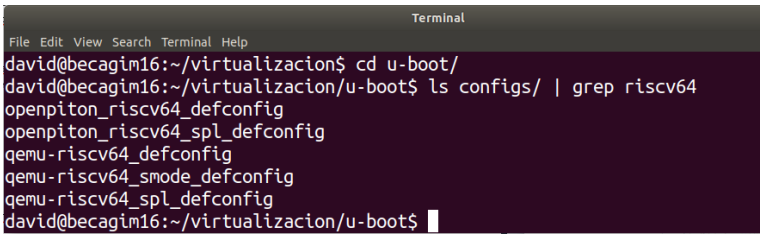

Figura 5: Configuración predefinida para dispositivos que implementan RISCV64 soportados por U-Boot

U-Boot debe ejecutarse en modo supervisor, por lo que se escoge la configuración «qemu-riscv64_smode_defconfig». Se construye U-Boot con esta configuración y se compila con el comando *make*.

```
$ make qemu-riscv64_smode_defconfig
$ make
```

7.4. Construcción del *firmware* con OpenSBI

Para iniciar U-boot es necesario tener un firmware como el que proporciona OpenSBI. El código de OpenSBI se obtiene desde la plataforma GitHub.

```
$ git clone http://github.com/riscv/opensbi.git
```

Para construir el firmware es necesario indicar cuál es el programa que debe iniciar. En este caso, se indica que debe iniciar el cargador de arranque U-Boot, pasando como parámetro el fichero «u-boot.bin» generado en el anterior apartado. Esto genera el archivo build/platform/generic/firmware/fw_payload.elf listo para ser arrancado con QEMU.

```
$ git clone http://github.com/riscv/opensbi.git
$ cd opensbi
$ make PLATFORM=generic FW_PAYLOAD_PATH=../u-boot/u-boot.bin
```

Se crea un script para el arranque de QEMU donde se especifica los parámetros de la máquina a emular. Esta máquina cuenta con 2 GB de memoria RAM y 4 núcleos, se emula sobre el dispositivo «virt» y no cuenta con una interfaz gráfica. El script, nombrado «run.sh» tiene el siguiente contenido:

```
#!/bin/sh
qemu-system-riscv64 -m 2G -nographic \
-machine virt \
-smp 4 \
-bios opensbi/build/platform/generic/firmware/fw_payload.elf \
```

Es necesario otorgar permisos de ejecución al script. Una vez concedidos se ejecuta y se inicia una máquina virtual que arranca OpenSBI, el cual después inicia U-Boot, como se observa en la figura 6:

```
$ chmod +x run.sh
$ ./run.sh
```

Figura 6: Inicio de los programas OpenSBI y U-Boot en una máquina virtual con QEMU

7.5. Compilación del núcleo de Linux

Con la cadena de herramientas de RISC-V es posible compilar el núcleo de Linux para la arquitectura RISC-V desde un sistema nativo x86. Para ello es necesario el código fuente, que puede ser obtenido desde Linux.org, por lo que se descarga la última versión de Linux disponible:

```
$ wget https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.18.2.tar.gz
$ tar -xvf linux-5.18.2.tar.gz
$ cd linux-5.18.2.tar.gz
```

Una vez se ha extraído el archivo, se accede a la carpeta del kernel y se establece la configuración por defecto, especificando la arquitectura de RISC-V. Se definen las variables de entorno «ARCH» y «CROSS_COMPILE» a la arquitectura objetivo y al compilador cruzado respectivamente.

La instrucción *make defconfig* establece la configuración por defecto, la cual tiene en cuenta el valor de la variable de entorno «ARCH», que ha sido previamente establecido. Para compilar el núcleo, se utiliza la instrucción *make*.

```
$ export ARCH=riscv
$ export CROSS_COMPILE=riscv64-unknown-linux-gnu-
$ make defconfig
$ make -j4
```

Tras esto los archivos generados son:

- vmlinux: Es el núcleo «crudo», el cual no está pensado para el arranque pero puede ser utilizado para depurar.
- arch/riscv/boot/Image: El núcleo listo para ser arrancado.
- arch/riscv/boot/Image.gz: El núcleo comprimido listo para ser arrancado.

Se ha iniciado una máquina virtual donde OpenSBI inicia U-Boot. El objetivo es que U-Boot arranque el núcleo de Linux, para ello es necesario darle a U-Boot el entorno necesario para el arranque. Se crea un disco virtual donde se aloja el entorno de U-Boot y el sistema de ficheros de Linux.

7.6. Creación de un disco virtual

Con el comando *dd if=/dev/zero of=disk.img bs=1M count=128* se crea un disco virtual con 128 bloques de 1 MB (128 MB). Se utiliza el comando *cfdisk* para particionar el disco creado en dos particiones 2 distintas, como se muestra en la figura 7.

1. boot: Partición de arranque de 64MB, con formato FAT32, que contiene el gestor de arranque.
2. rootfs: Partición tipo ext4 de 63MB que contiene el sistema de ficheros donde se iniciará el núcleo de Linux.

```
$ dd if=/dev/zero of=disk.img bs=1M count=128
$ cfdisk disk.img
```

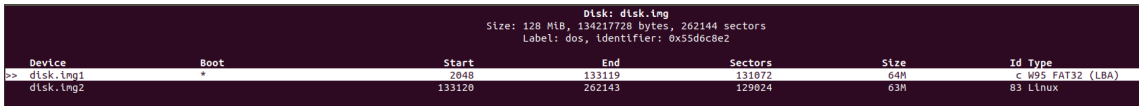

Figura 7: Particionado del disco virtual con cfdisk

Con el comando «losetup» es posible asociar dispositivos *loop* con archivos como este nuevo disco virtual. Se utiliza el comando *losetup -f –show –partscan disk.img*, que muestra el dispositivo *loop* asociado al disco virtual. Una vez asociado al disco, se procede a crear los sistemas de ficheros de las particiones como se muestra en la figura 8.

```
File  Edit  View  Search  Terminal  Help
david@becagim16:~/virtualizacion$ sudo losetup -f --show --partscan disk.img
/dev/loop23
david@becagim16:~/virtualizacion$ ls -la /dev/loop23*
brw-rw---- 1 root disk   7, 23 jul  1 18:52 /dev/loop23
brw-rw---- 1 root disk 259,  6 jul  1 18:52 /dev/loop23p1
brw-rw---- 1 root disk 259,  7 jul  1 18:52 /dev/loop23p2
david@becagim16:~/virtualizacion$ sudo mkfs.vfat -F 32 -n boot /dev/loop23p1
mkfs.fat 4.1 (2017-01-24)
mkfs.fat: warning - lowercase labels might not work properly with DOS or Windows
david@becagim16:~/virtualizacion$ sudo mkfs.ext4 -L rootfs /dev/loop23p2
mke2fs 1.44.1 (24-Mar-2018)
Discarding device blocks: done
Creating filesystem with 64512 1k blocks and 16128 inodes
Filesystem UUID: a1c01001-54e6-4b5f-8837-a708eb947c0b
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done

david@becagim16:~/virtualizacion$
```

Figura 8: Se crean los sistemas de ficheros en las particiones del disco virtual

Una vez creados los sistemas de ficheros en el disco virtual, se montan las particiones del disco sobre directorios del equipo para acceder a su contenido y cargar los ficheros necesarios. Se crean los directorios /mnt/boot y /mnt/rootfs y se montan las particiones como se muestra en la figura 9. Con los sistemas de ficheros ya creados se copia en el directorio /mnt/boot la imagen de Linux compilada en el apartado 7.5.

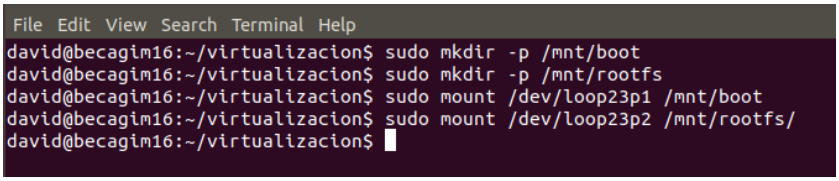

Figura 9: Montaje de las particiones del disco virtual

```
$ sudo cp linux-5.18.2/arch/riscv/boot/Image /mnt/boot
```

Es necesario recompilar U-Boot para que soporte un entorno no volátil, por lo que se ejecuta el comando *menuconfig* para acceder a la interfaz gráfica de la configuración de U-Boot, donde se establecen los siguientes parámetros:

- CONFIG_ENV_IS_IN_FAT=y
- CONFIG_ENV_FAT_INTERFACE="virtio"
- CONFIG_ENV_FAT_DEVICE_AND_PART="0:1"

Tras aplicar los cambios se recompila U-Boot con el comando *make* para volver a generar el archivo u-boot.bin y se recompila OpenSBI de nuevo con los comandos:

```
$ cd opensbi
$ make PLATFORM=generic FW_PAYLOAD_PATH=../u-boot/u-boot.bin
```

Ahora que OpenSBI está compilado con el binario de U-Boot, se ejecuta de nuevo la máquina virtual con QEMU, introduciendo nuevos parámetros para utilizar el disco virtual creado anteriormente. El script «run.sh» modificado es el siguiente:

```
#!/bin/sh
qemu-system-riscv64 -m 2G -nographic \
-machine virt \
-smp 8 \
-bios opensbi/build/platform/generic/firmware/fw_payload.elf \
-drive file=disk.img,format=raw,id=hd0 \
-device virtio-blk-device,drive=hd0 \
```

7.7. Creación del sistema de ficheros con BusyBox

Linux necesita un sistema de ficheros, el cual se puede crear con la herramienta BusyBox. BusyBox es un programa de código abierto que proporciona un entorno bastante completo

para cualquier sistema pequeño o embebido. BusyBox ha sido escrito teniendo en cuenta la optimización del tamaño y los recursos limitados. También es extremadamente modular, por lo que se pueden incluir o excluir fácilmente comandos (o características) en tiempo de compilación. Esto facilita la personalización de sistemas embebidos.

El software de BusyBox puede descargarse desde su página oficial y se configura con el comando *make allnoconfig* para excluir todo. Luego con *make menuconfig* se despliega una interfaz gráfica a través de la que es posible seleccionar qué excluir o incluir en el entorno que será creado.

```
$ wget https://busybox.net/downloads/busybox-1.33.2.tar.bz2
$ tar -xvf busybox-1.33.2.tar.bz2
$ cd busybox-1.33.2
$ make allnoconfig
$ make menuconfig
```

Dentro de la interfaz gráfica se incluyen las utilidades básicas para crear un entorno mínimo: La consola ash, comandos init y halt, comandos como cat, echo, mkdir, ls, chmod, editores como vi.

En el apartado de «SETTINGS» seleccionamos la opción *Build Static Binary* y establecemos la opción de *Cross compiler prefix* a «riscv64-unknown-linux-gnu-». Cuando termine la configuración se ejecutan los siguientes comandos para compilar BusyBox y crear una carpeta con bin, sbin y usr.

```
$ make
$ make install
```

Se puede observar el sistema de ficheros creado ejecutando los siguientes comandos. El sistema de ficheros será similar al mostrado en la figura 10.

```
$ sudo apt install tree
$ tree _install
```

Se monta la partición del disco virtual destinada al sistema de ficheros en un directorio y se introduce el sistema de ficheros de la siguiente forma. Además, es necesario crear los directorios de sys, proc y dev que BusyBox no ha creado.

```
$ sudo mount /dev/loop23p2 /mnt/rootfs
$ sudo rsync -aH _install/ /mnt/rootfs
$ mkdir /mnt/rootfs/dev
$ mkdir /mnt/rootfs/proc
$ mkdir /mnt/rootfs/sys
$ mkdir /mnt/rootfs/etc
$ mkdir /mnt/rootfs/etc/init.d
```

También creamos el archivo rcS, que es el primer script en ser ejecutado al inicio del sistema y permite ejecutar programas adicionales en el momento del arranque. Este script se usa habitualmente para montar sistemas de archivos adicionales. En este caso, será usado para montar los directorios virtuales /proc y /sys. Se indica que se debe ejecutar el archivo rcS en el archivo /etc/inittab. El contenido del archivo inittab y rcS es el siguiente:

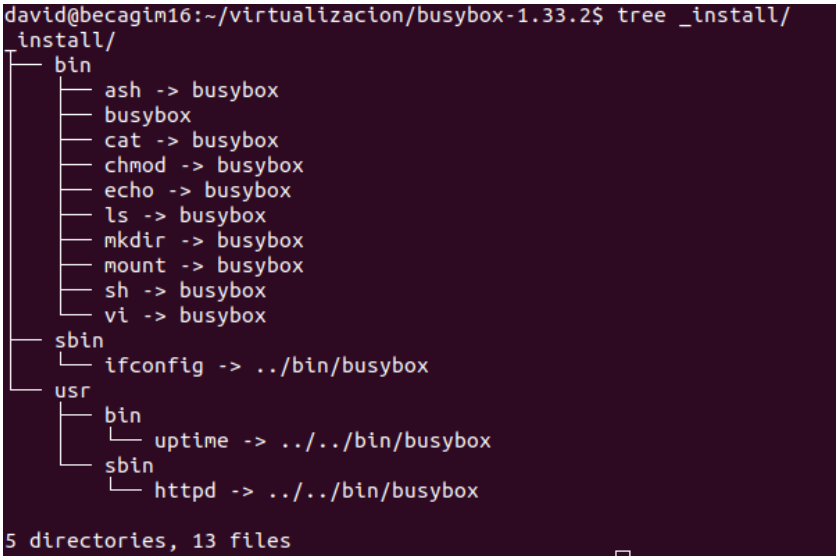

Figura 10: Sistema de ficheros creado con BusyBox

```
# inittab
# Primer script en ser ejecutado
::sysinit::/etc/init.d/rcS
# Establece en la consola un askfirst
::askfirst::/bin/sh
```

```
# rcS
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
```

Se ubica el archivo inittab en el directorio /etc y el archivo rcS en el directorio /etc/init.d. Dentro de la máquina virtual será necesario establecer permisos de ejecución al archivo rcS.

```
$ cp inittab /mnt/rootfs/etc
$ cp rcS /mnt/rootfs/etc/init.d
$ sudo umount /dev/loop23p2
```

Para indicar el arranque del núcleo por parte de U-Boot es necesario crear variables de entorno que definan dónde se carga el núcleo y cómo se debe iniciar. Se ejecuta entonces la máquina virtual con el script «run.sh». Dentro de U-Boot es necesario establecer las siguientes variables en el entorno:

```
$ setenv bootargs 'root=/dev/vda2 rootwait console=ttyS0 earlycon=sbi rw'
$ setenv bootcmd 'fatload virtio 0:1 84000000 Image; booti 0x84000000 -
    ${fdtcontroladdr}'
```

Con «bootargs» se está indicando que el dispositivo que Linux debe montar como sistema de ficheros es /dev/vda2. «rootwait» indica que se debe esperar a que el root esté listo antes de intentar montarlo y «console=ttyS0» indica el dispositivo por donde Linux imprimirá mensajes.

La variable de entorno «bootcmd» indica los comandos que U-Boot ejecuta tras un retraso indicado por la variable «bootdelay». Estos comandos indican el dispositivo y partición desde donde se debe cargar la imagen de Linux, sobre qué dirección debe ser cargada y luego le indica que arranque el sistema. Al ejecutar de nuevo la máquina virtual se iniciará Linux, como se muestra en la figura 11.

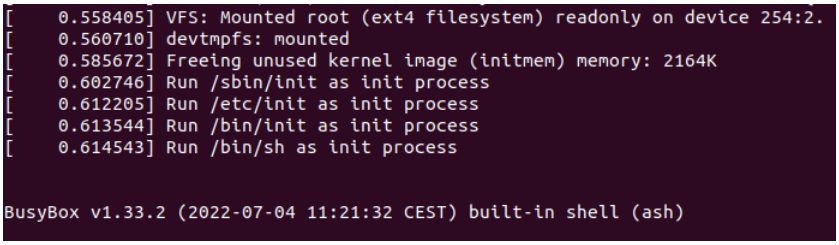

Figura 11: Arranque de Linux en la máquina virtual QEMU

Por último, se comprueba el funcionamiento de un código escrito en el lenguaje de programación C compilado para RISC-V dentro de la máquina virtual. Se monta de nuevo el disco virtual para introducir en el sistema el fichero compilado y se arranca de nuevo la máquina. Para ejecutar el programa se le debe dar permisos de ejecución. El programa se ejecuta de forma correcta obteniendo la salida esperada como en la figura 12.

```
$ sudo mount /dev/loop23p2 /mnt/rootfs
$ sudo cp hello /mnt/rootfs/
$ sudo umount /dev/loop23p2
```

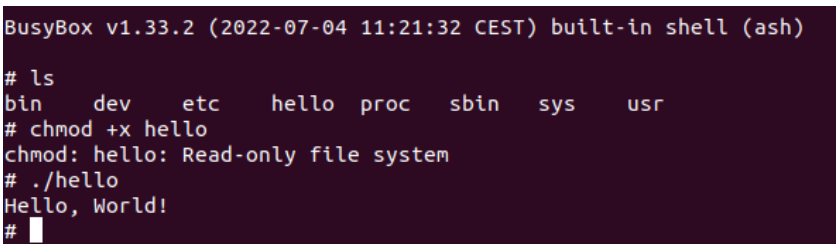

Figura 12: Ejecución del programa hello en la máquina virtual

8. Instalación en FPGA

Para la instalación de Linux en una placa con FPGA se debe programar en el dispositivo un SoC basado en RISC-V. Para la instalación de Linux, es necesario que el SoC soporte el modo supervisor explicado en el apartado 2.1.1, para lo que se necesita una unidad de virtualización de memoria como una MMU y soporte para las intrucciones CSR.

Para la programación de un SoC y la instalación de Linux se han considerado/ tenido en cuenta la placa Xilinx Zynq Ultrascale+ MPSoC ZCU102 y la placa Nexys A7. A continuación se hablará

8.1. Xilinx Zynq Ultrascale+ MPSoC ZCU102

La placa Xilinx Zynq UltraScale+ MPSoC ZCU102, mostrada en la figura 13, es un dispositivo fabricado por Xilinx para proyectos de diseños para aplicaciones de automoción, industriales, de vídeo y de comunicaciones. Esta placa contiene 4 GB de memoria RAM DDR4, una memoria Quad SPI flash de 64 MB, es programable por JTAG a través del puerto microUSB y tiene varios periféricos como un puerto HDMI, un conector ethernet RJ45 y una ranura para tarjetas SD entre otros[19].

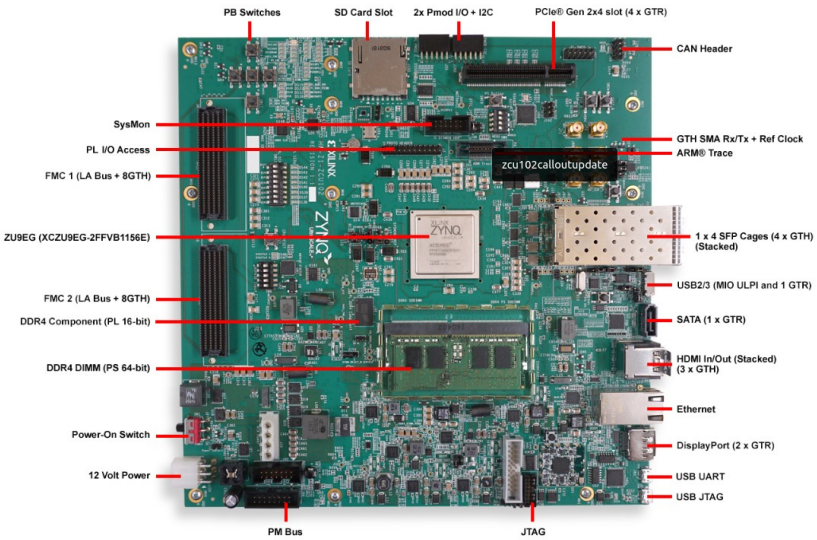

Figura 13: Placa Zynq Ultrascale+ MPSoC ZCU102

La característica principal de esta placa es el dispositivo FPGA Zynq UltraScale+ MPSoC, que cuenta con un procesador de cuatro núcleos Arm Cortex-A53, un procesador de tiempo real Cortex-R5F de dos núcleos y una unidad de procesamiento gráfico Mali-400 MP2 basada en el tejido lógico programable FinFET+ de 16 nm de Xilinx. Estas características aportan a la placa ZCU102 una gran capacidad de procesamiento.

Esta placa no cuenta con mucho soporte en la web más allá del proporcionado por la propia plataforma de Xilinx, seguramente debido a su elevado coste que hace del dispositivo poco accesible y por tanto es una tarea difícil encontrar proyectos relacionados con RISC-V que den soporte a esta máquina.

La plataforma PULP introducida en el apartado 6.2 proporciona el microcontrolador PULPissimo compatible con este dispositivo. Dicho microcontrolador fue instalado en la placa como se muestra en el trabajo de fin de master de Laura Saiz Orza mencionado en el apartado de motivación, pero el SoC que implementa no es capaz de ejecutar un sistema Linux debido a que carece de MMU, necesaria para la virtualización de memoria basada en páginas.

PULP ofrece un procesador compatible con un dispositivo MMU en su proyecto Ariane, pero este proyecto se encuentra aún en desarrollo y es necesario crear un SoC que lo implemente y pueda ser compatible con la ZCU102. Además, son necesarias otras modificaciones en el gestor de arranque U-Boot para soportar este dispositivo, por lo que en este trabajo se han explorado otras alternativas a la placa ZCU102 para realizar la instalación física.

8.2. Nexys A7

La placa Nexys A7 es una placa orientada al desarrollo de circuitos digitales completa y lista para usar, basada en el dispositivo Artix-7 *Field Programmable Gate Array* (FPGA) de Xilinx, que cuenta con 128 MB de memoria DDR2, una memoria Quad SPI no volátil y varios periféricos como una ranura para tarjetas SD, un conector Ethernet RJ45, un puerto VGA, un puerto USB y un puerto microUSB entre otros[20].

Este dispositivo implementa varios modos de arranque, que son indicados cambiando de posición los «puentes» o *jumpers* del dispositivo (elementos azules de la figura 14). Estos modos de arranque determinan si el dispositivo arranca leyendo el contenido de la memoria Quad SPI para programar la FPGA, espera a recibir una programación por el puerto UART a través del microUSB o si debe leer información de la SD para arrancar.

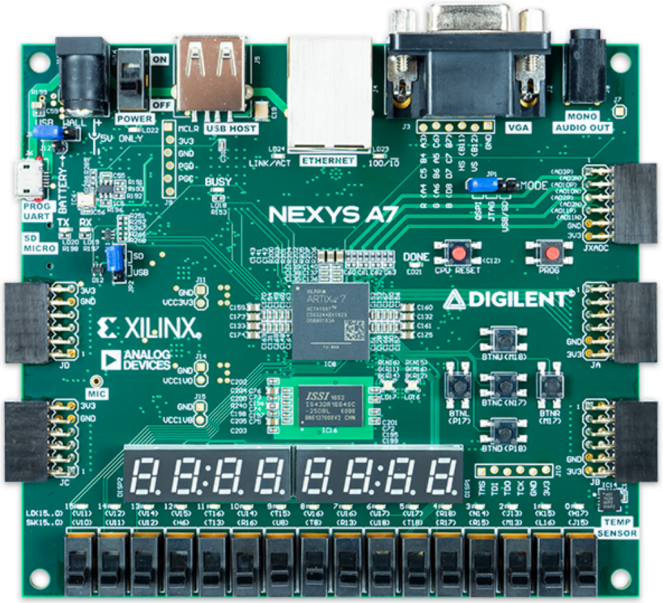

Figura 14: Placa Nexys A7

Esta placa cuenta con la memoria necesaria para ejecutar un sistema Linux simple y se trata de una placa más accesible que la ZCU102, tiene más soporte online y hay varios SoC

basados en la arquitectura RISC-V que son compatibles con este dispositivo, por lo que ha sido elegido como alternativa a la Zynq Ultrascale+ MPSoC ZCU102. A continuación se exploran distintos SoC basados en RISC-V capaces de ejecutar Linux.

8.3. RVSoC

El proyecto RVSoC es un proyecto de investigación y desarrollo del sistema informático RISC-V dirigido a FPGAs en Verilog HDL en el Arch Lab, Tokyo Tech. RVSoC es un sistema portable capaz de ejecutar Linux basado en RISC-V para dispositivos FPGA.

El SoC está compuesto por un procesador que implementa las instrucciones RV32-IMAC y un microcontrolador, conectados ambos a través de una MMU a una memoria principal, un disco y una consola. RVSoC cuenta con una MMU, lo que es fundamental para el soporte de un sistema Linux, ya que esta unidad proporciona protección y vitualización de memoria. También es esencial para un sistema que pretenda ejecutar Linux el soporte para las instrucciones CSR y las instrucciones atómicas. Para esto último el procesador de RVSoC se apoya en un microcontrolador gracias al cual puede ejecutar las instrucciones atómicas en un solo ciclo.

Aunque RVSoC es funcional, no es un sistema construido con los entornos de diseño de SoC basado en RISC-V vistos anteriormente en este trabajo.

8.4. LowRISC

LowRISC es una empresa sin ánimo de lucro con sede en Cambridge, Reino Unido. Utiliza la ingeniería colaborativa para desarrollar y mantener diseños y herramientas de silicio de código abierto.

LowRISC está involucrado en proyectos como el RISC-V LLMV y también tiene su propio proyecto de desarrollo de SoC, dedicado a desarrollar un SoC para distintos núcleos RISC-V como Rocket o BOOM, desarrollados por la Universidad de Berkeley o Ibex, desarrollado por la plataforma PULP.

LowRISC utliza el un núcleo generado por Rocket, introducido en el apartado 6.2.1, el cual cuenta con una MMU que soporta memoria virtual basada en páginas, una caché de datos no bloqueante y un front-end con predicción de salto. El SoC de LowRISC, permite el uso de periféricos de código abierto, como Ethernet o SD. LowRISC tiene un proyecto abierto en el que se propone dar soporte al procesador Ariane, pero este proyecto lleva sin ser actualizado desde 2019.

8.5. Vivado-RISC-V

El proyecto Vivado RISC-V es un proyecto de Eugene Tarassov, ingeniero de AMD/Xilinx, el cual proporciona un prototipo para FPGA de Linux sobre una arquitectura RISC-V completamente funcional. Incluye scripts y fuentes para generar el HDL del SoC RISC-V, el proyecto Vivado de Xilinx, el flujo de bits de la FPGA y la tarjeta SD de arranque. La tarjeta SD contiene RISC-V *Open Source Supervisor Binary Interface* (OpenSBI), U-Boot, el Linux kernel y Debian root FS.

El proyecto Vivado-RISC-V ofrece un método sencillo de utilizar el generador Rocket para instanciar diseños de SoC personalizando el número de núcleos, la arquitectura de 32 o 64 bits y en algunos casos el tamaño de la memoria cache L1. Debido a que este proyecto es compatible con la placa Nexys A7 y ofrece los parches necesarios para el soporte de U-Boot y OpenSBI es con el cual se genera el SoC utilizado en la Nexys A7 para la instalación de Linux.

8.5.1. Instalación del SoC Vivado-RISC-V en la Nexys A7

A continuación se explica el proceso de instalación del *System on Chip* proporcionado por Eugene Tarassov compatible con la placa Nexys A7. Este proceso puede dividirse en 3 fases distinguibles:

- Generación del bitstream con la herramienta Vivado y programación de la FPGA.
- Generación de una tarjeta SD booteable para arrancar el sistema en la FPGA. Se utilizarán el bootloader presentado en el apartado 3.2. U-boot.
- Arranque del dispositivo y ejecución de un programa escrito en el lenguaje de programación C compilado para la arquitectura RISC-V.

El proceso de instalación de la herramienta Vitis se explica en el anexo II de este documento. Además, es necesaria la instalación de controladores específicos para los dispositivos de Diligent, como la placa con la que se trabaja. El proceso de instalación de controladores de Diligent se explica en el anexo III de este documento.

Para generar el bitstream de Vivado-RISC-V es necesario descargar el repositorio GitHub. Una vez descargado el repositorio se ejecutan las instrucciones *make apt-install*, que instala todas las dependencias necesarias para el proyecto, y *sudo update-submodules* que actualiza y descarga todos los componentes del proyecto pertenecientes a otros repositorios en una versión estable, como el repositorio de OpenSBI, RocketChip y U-Boot.

```
$ git clone https://github.com/eugene-tarassov/vivado-risc-v
$ make apt-install
$ make update-submodules
```

Una vez actualizado y descargado todo se ejecutan los siguientes comandos para configurar el entorno apropiado que permite el uso de Vivado para generar el bitstream. Vivado-RISC-V ofrece dsitintas opciones en función del tipo de SoC que se quiera generar. En el parámetro «CONFIG» puede especificarse el tipo de procesador a implementar (Rocket o BOOM) y el número de núcleos. La placa Nexys A7 cuenta con 128 MB de memoria RAM, suficiente para ejecutar un Linux sencillo sobre un núcleo a 50 MHz.

```
$ source /tools/Xilinx/Vivado/2022.1/settings64.sh
$ make CONFIG=rocket64b1 BOARD=nexys-a7-100t bitstream
```

Este último comando ejecuta Vivado y realiza el proceso de generación de un *bitstream* visto en el apartado 5.1. Cuando el proceso termina se genera el archivo workspace/ rocket64b1/nexys-a7-100t-riscv.mcs. Para programar la FPGA, se inicia Vivado, se accede al *Hardware Manager* y se conecta la placa Nexys A7 al PC mediante un cable USB-microUSB.

En el manejador de hardware de Vivado se seleccionan las opciones *Open target* → *Auto Connect* para que el software de Vivado detecte con la placa Nexys A7. Una vez conectada, se programa el dispositivo de memoria no volátil donde se instala la configuración de la FPGA. Para esto se enciende la placa mediante el interruptor SWI6, se escoge el dispositivo FPGA xc7a100t_0 en el entorno gráfico de Vivado y con el botón derecho del ratón se selecciona *Add Configuration Memory Device*. Esto despliega un menú como el mostrado en la figura 15, donde se debe introducir el dispositivo de memoria a instalar, el cual en este caso es s25fl128sxxxxxx0.

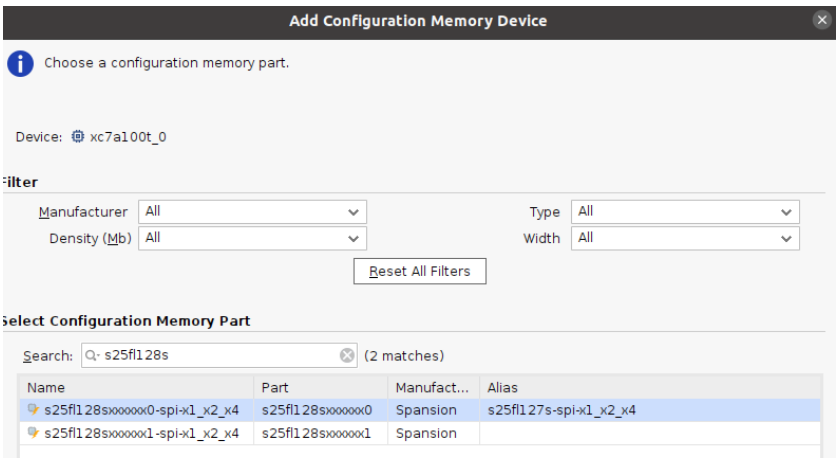

Figura 15: Selección del dispositivo de memoria no volátil para el dispositivo FPGA

Tras especificar el dispositivo de memoria Vivado mostrará un cuadro en el que pregunta si se desea programar el dispositivo ahora. Se selecciona la opción *OK*. Es necesario indicar el bitstream del SoC que se quiere programar como *Configuration file*. Este archivo es el creado en los pasos anteriores, encontrado en .../vivado-risc-v/workspace/rocket64b1/nexys-a7-100t-riscv.mcs. Por último se selecciona *OK* otra vez y comienza la programación del dispositivo.

Ya se ha conseguido instalar un SoC basado en RISC-V en la FPGA de la Nexys A7, tan solo falta instalar sobre ella el sistema Linux, para lo que se construye U-Boot y OpenSBI, además de un sistema de ficheros sobre el que se instala el núcleo. Todo este proceso es realizado mediante la macro «mk-sd-card». Vivado-RISC-V incluye una versión estable de linux y la configura con los parámetros definidos en el fichero .../patches/linux.config (figura 16), luego descarga un sistema de ficheros y un initrd para el arranque (figura 17). Linux es compilado para RISC-V (figura 18).

```
if [ -s patches/linux.patch ] ; then cd linux-stable && ( git apply -R --check ../patches/linux.patch 2>/dev/null || git apply ../patches/linux.patch ) ; fi
cp -p patches/fpga-axi-eth.c  linux-stable/drivers/net/ethernet
cp -p patches/fpga-axi-sdc.c  linux-stable/drivers/mmc/host
cp -p patches/fpga-axi-uart.c linux-stable/drivers/tty/serial
cp -p patches/linux.config linux-stable/.config
```

Figura 16: Parcheado del núcleo de Linux

```
mkdir -p debian-riscv64
curl --netrc --location --header 'Accept: application/octet-stream' \
  https://api.github.com/repos/eugene-tarassov/vivado-risc-v/releases/assets/59988402 \
  -o debian-riscv64/initrd.tmp
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 3711k  100 3711k    0     0  3482k      0  0:00:01  0:00:01 --:--:-- 3482k
mv debian-riscv64/initrd.tmp debian-riscv64/initrd
mkdir -p debian-riscv64
curl --netrc --location --header 'Accept: application/octet-stream' \
  https://api.github.com/repos/eugene-tarassov/vivado-risc-v/releases/assets/59988403 \
  -o debian-riscv64/rootfs.tar.gz.tmp
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  151M  100  151M    0     0  10.6M      0  0:00:14  0:00:14 --:--:-- 11.1M
mv debian-riscv64/rootfs.tar.gz.tmp debian-riscv64/rootfs.tar.gz
```

Figura 17: Descarga del sistema de ficheros debian e initrd

```
mkdir -p debian-riscv64
curl --netrc --location --header 'Accept: application/octet-stream' \
  https://api.github.com/repos/eugene-tarassov/vivado-risc-v/releases/assets/59988402 \
  -o debian-riscv64/initrd.tmp
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 3711k  100 3711k    0     0  3482k      0  0:00:01  0:00:01 --:--:-- 3482k
mv debian-riscv64/initrd.tmp debian-riscv64/initrd
mkdir -p debian-riscv64
curl --netrc --location --header 'Accept: application/octet-stream' \
  https://api.github.com/repos/eugene-tarassov/vivado-risc-v/releases/assets/59988403 \
  -o debian-riscv64/rootfs.tar.gz.tmp
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  151M  100  151M    0     0  10.6M      0  0:00:14  0:00:14 --:--:-- 11.1M
mv debian-riscv64/rootfs.tar.gz.tmp debian-riscv64/rootfs.tar.gz
```

Figura 18: Compilación del núcleo de Linux

Como se ya se ha visto en el apartado 3.2.2, U-Boot requiere de la especificación del dispositivo sobre el que se va a ejecutar y Vivado-RISC-V proporciona esta información, que es parcheada bajo el directorio de U-Boot para poder realizar la construcción del binario, construyendo un nuevo dispositivo denominado «vivado-riscv». Tras conseguir el binario de U-Boot se construye el binario de OpenSBI, cuya configuración también debe ser parcheada para adaptarse al dispositivo sobre el que se ejecuta.

U-Boot y OpenSBI son dependientes del hardware sobre los que se ejecutan y por tanto es complicado realizar una instalación de Linux en un dispositivo no soportado.

Antes de ejecutar la macro «mk-sd-card» es necesario introducir una tarjeta SD en nuestro PC, ya que además de todo el proceso anterior, esta macro detecta la tarjeta SD y crea una nueva tabla de particiones para alojar el cargador de arranque y el sistema de ficheros. Primero es creada la imagen en un disco virtual que no esté siendo utilizado (figura 19) y más tarde se escribe sobre el dispositivo físico. Antes de escribir información sobre la tarjeta, se preguntará si el volumen indicado es donde se pretenden realizar estos procesos para asegurarse de no borrar datos del usuario, como se observa en la figura 20.

```
SD image device: /dev/loop10
Checking that no-one is using this disk right now ... OK

Disk /dev/loop10: 1,48 GiB, 1572864000 bytes, 3072000 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

>>> Created a new DOS disklabel with disk identifier 0x7a0d18e4.
/dev/loop10p1: Created a new partition 1 of type 'W95 FAT16 (LBA)' and of size 64 MiB.
/dev/loop10p2: Created a new partition 2 of type 'Linux' and of size 1,4 GiB.
/dev/loop10p3: Done.

New situation:
Disklabel type: dos
Disk identifier: 0x7a0d18e4

Device        Boot  Start     End Sectors  Size Id Type
/dev/loop10p1 *      2048  133119  131072   64M  e W95 FAT16 (LBA)
/dev/loop10p2      133120 3071999 2938880  1,4G 83 Linux

The partition table has been altered.
mkfs.fat 4.1 (2017-01-24)
mke2fs 1.45.5 (07-Jan-2020)
Creating filesystem with 367360 4k blocks and 91968 inodes
Filesystem UUID: 68d82fa1-1bb5-435f-a5e3-862176586eec
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912

Allocating group tables: done
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

~/Desktop/vivado-risc-v/debian-riscv64/rootfs ~/Desktop/vivado-risc-v
~/Desktop/vivado-risc-v
~/Desktop/vivado-risc-v/debian-riscv64/boot ~/Desktop/vivado-risc-v
~/Desktop/vivado-risc-v

Boot partition:
Filesystem     1K-blocks  Used Available Use% Mounted on
/dev/loop10p1      65390 24262     41128  38% /home/master/Desktop/vivado-risc-v/debian-riscv64/boot
total 1292
-rwxr-xr-x 1 root root 1319256 jul  3 19:26 boot.elf
drwxr-xr-x 2 root root    2048 jul  3 19:26 extlinux

Root partition:
Filesystem     1K-blocks   Used Available Use% Mounted on
/dev/loop10p2    1413536 694808    628872  53% /home/master/Desktop/vivado-risc-v/debian-riscv64/rootfs
total 262236
drwxr-xr-x  2 root root      4096 mar 19 09:36 bin
drwxr-xr-x  2 root root      4096 mar 19 09:52 boot
drwxr-xr-x  4 root root      4096 ene 19 23:35 dev
drwxr-xr-x 62 root root      4096 mar 19 09:52 etc
drwxr-xr-x  3 root root      4096 mar 13 07:32 home
drwxr-xr-x 11 root root      4096 mar 19 09:12 lib
drwx------  2 root root     16384 mar 13 07:40 lost+found
drwxr-xr-x  2 root root      4096 mar 13 07:29 media
drwxr-xr-x  2 root root      4096 mar 13 07:29 mnt
drwxr-xr-x  2 root root      4096 mar 13 07:29 opt
drwxr-xr-x  2 root root      4096 ene 19 23:35 proc
drwx------  3 root root      4096 mar 13 07:34 root
drwxr-xr-x  4 root root      4096 mar 13 07:34 run
drwxr-xr-x  2 root root      4096 mar 19 09:36 sbin
drwxr-xr-x  2 root root      4096 mar 13 07:29 srv
-rw-------  1 root root 268435456 mar 19 09:52 swapfile
drwxr-xr-x  2 root root      4096 ene 19 23:35 sys
drwxrwxrwt  2 root root      4096 mar 19 09:51 tmp
drwxr-xr-x 11 root root      4096 mar 13 07:29 usr
drwxr-xr-x 11 root root      4096 mar 13 07:29 var

fsck.fat 4.1 (2017-01-24)
/dev/loop10p1: 6 files, 12131/32695 clusters
rootfs: 15972/91968 files (0.1% non-contiguous), 187678/367360 blocks
```

Figura 19: Creación de la imagen en un disco virtual

```
Copy disk image to:
Disk /dev/sdb: 14,44 GiB, 15489564672 bytes, 30253056 sectors
Disk model: Storage Device
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x7a0d18e4

Device     Boot  Start     End Sectors  Size Id Type
/dev/sdb1  *      2048  133119  131072   64M  e W95 FAT16 (LBA)
/dev/sdb2       133120 3071999 2938880  1,4G 83 Linux
Are you sure? [y/N] y
1569652736 bytes (1,6 GB, 1,5 GiB) copied, 79 s, 19,9 MB/s
24000+0 records in
24000+0 records out
1572864000 bytes (1,6 GB, 1,5 GiB) copied, 215,02 s, 7,3 MB/s
Checking that no-one is using this disk right now ... OK

Disk /dev/sdb: 14,44 GiB, 15489564672 bytes, 30253056 sectors
Disk model: Storage Device
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x14fe0eb3

Old situation:

Device     Boot  Start     End Sectors  Size Id Type
/dev/sdb1  *      2048  133119  131072   64M  e W95 FAT16 (LBA)
/dev/sdb2       133120 3071999 2938880  1,4G 83 Linux

/dev/sdb2:
New situation:
Disklabel type: dos
Disk identifier: 0x14fe0eb3

Device     Boot  Start      End  Sectors  Size Id Type
/dev/sdb1  *      2048   133119   131072   64M  e W95 FAT16 (LBA)
/dev/sdb2       133120 30253055 30119936 14,4G 83 Linux

The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
fsck from util-linux 2.34
e2fsck 1.45.5 (07-Jan-2020)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
rootfs: 15972/91968 files (0.1% non-contiguous), 187681/367360 blocks
resize2fs 1.45.5 (07-Jan-2020)
Resizing the filesystem on /dev/sdb2 to 3764992 (4k) blocks.
The filesystem on /dev/sdb2 is now 3764992 (4k) blocks long.
```

Figura 20: Copia de la imagen en el dispositivo físico

Antes de meter la tarjeta en la placa se introduce en el sistema de ficheros el archivo «hello» compilado en el apartado 7 para poder ejecutarlo en la Nexys A7. Para ello se monta la partición correspondiente en un directorio del sistema y se introduce el archivo.

```
$ sudo mkdir /mnt/rootfs
$ sudo mount /dev/sdb2 /mnt/rootfs/
$ sudo cp hello /mnt/rootfs/home/debian/
$ sudo umount /mnt/rootfs
```

Se introduce la tarjeta SD a la placa Nexys A7 y se inicia de nuevo. Para poder interactuar con la placa es necesario establecer una conexión a través del puerto USB. Para esto se usa «miniterm» de la siguiente forma:

```
$ sudo apt-get install python-serial
$ sudo miniterm /dev/ttyUSB1 115200
```

Al establecer conexión con la placa se muestra el inicio de OpenSBI (figura 21), que inicia en modo máquina y da paso al gestor de arranque U-Boot, que carga la imagen del núcleo de Linux y lo arranca como se muestra en la figura 22.

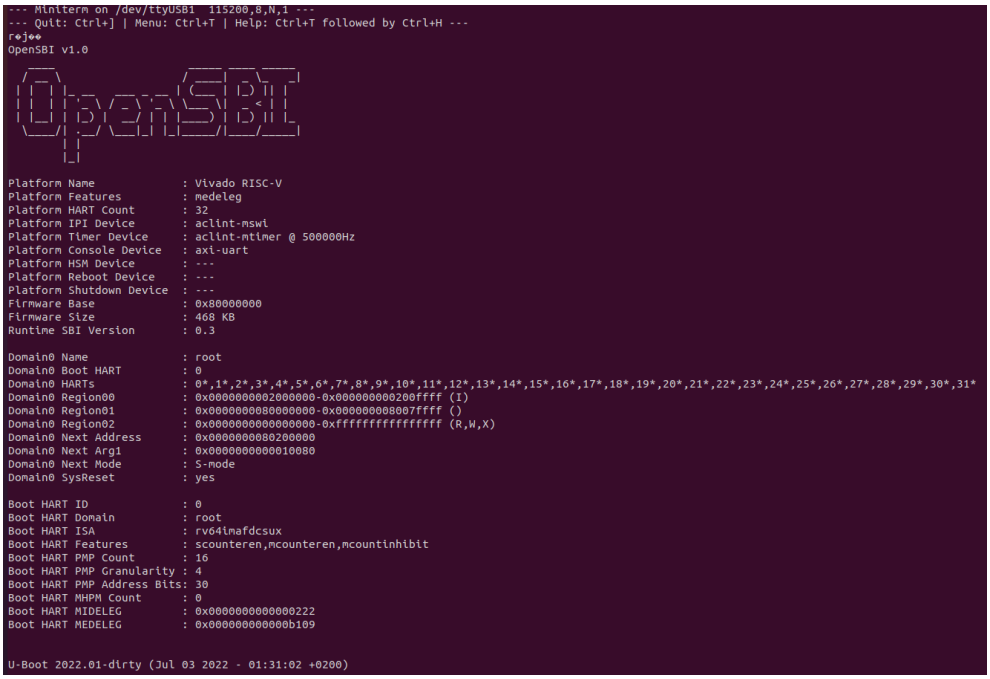

Figura 21: Arranque de OpenSBI en la Nexys A7

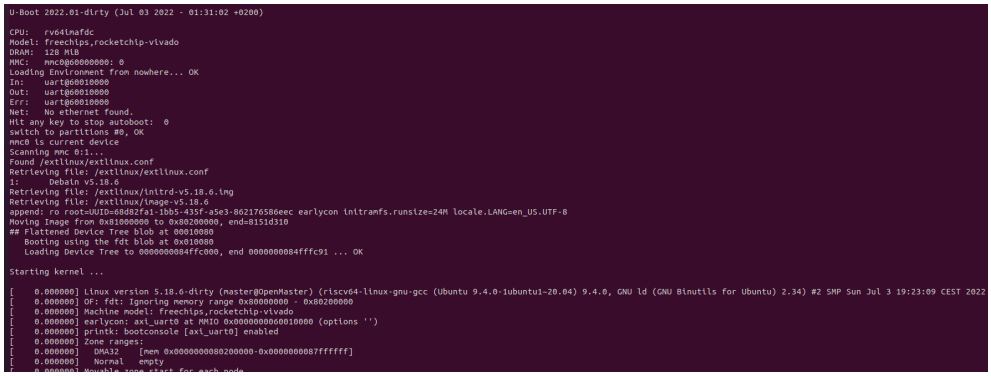

Figura 22: Inicio de U-Boot y principio del arranque del núcleo de Linux

El kernel se inicia tras unos minutos y ya es posible acceder al sistema. El sistema Linux tiene dos usuarios, «root» y «debian», a los cuales se puede acceder con las credenciales *login*: root *password*: root o *login*: debian *password*: debian respectivamente como se observa en la figura 23.

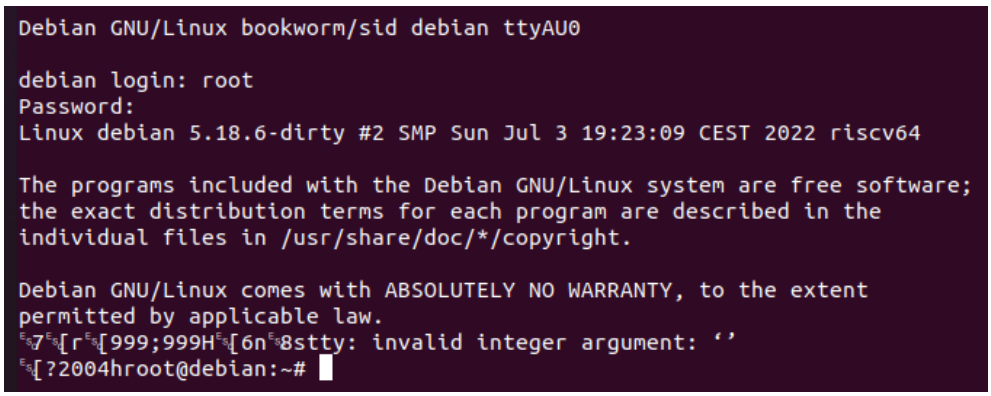

Figura 23: Inicio de sesion como root en Linux

Para la comprobación del programa compilado escrito en el lenguaje de programación C se deben otorgar al programa permisos de ejecución y luego se ejecuta. El programa se termina correctamente e imprime la salida esperada como se muestra en la figura 24.

```
$ cd /home/debian
$ chmod +x hello_world.o
$ ./hello_world.o
```

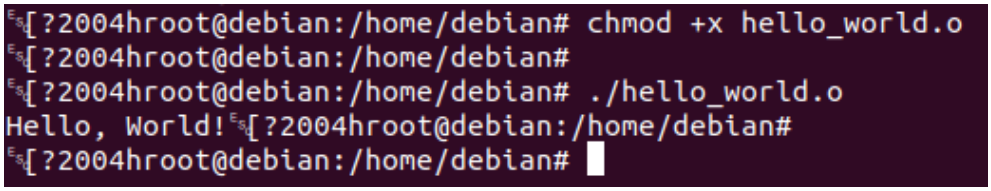

Figura 24: Ejecución de Hello World en Linux

9. Conclusión

Al tratarse de una arquitectura hardware libre, la arquitectura RISC-V tiene el potencial de convertirse en una ISA universal mantenida y desarrollada una comunidad que la conserve actualizada y fiable, siendo el símil de Linux en el ámbito del hardware. A pesar de hallarse en su fase inicial, RISC-V ya cuenta con varias plataformas dedicadas al desarrollo de tecnologías basadas en esta arquitectura tales como las vistas en este trabajo y con soporte para programas como los gestores de arranque trabajados o emuladores como QEMU.

En este trabajo se ha aplicado una solución para la instalación de un sistema operativo Linux, lo que aporta a los sistemas basados en la arquitectura RISC-V múltiples proveedores de software de código abierto y una mayor flexibilidad debido a la alta capacidad de personalización del sistema altamente modulable.

A lo largo de este trabajo se han encontrado diversas dificultades. Las soluciones SoC basadas en RISC-V no están disponibles para muchas FPGA, lo que dificulta la compatibilidad con distintas placas. Así mismo, empezar trabajando con la ZCU102 fue un error debido a que los diseños disponibles requiren de modificaciones para ser compatibles con esta placa. Las limitaciones temporales han sido sin duda otro de los factores determinantes en la realización de este proyecto, lo que nos ha impedido explorar con más detalle el uso de herramientas de código abierto para la personalización de los *System on Chip*.

En el futuro sería interesante explorar los frameworks presentados en este trabajo para el diseño de *System on Chip*, consiguiendo modelos de sistemas propios y comparándolos a los del actual estado del arte.

Bibliografía

[1] L. Saiz Orza, «Implementación de un procesador RISC-V sobre FPGA ZCU102», Trabajo fin de Master. Departamento TEISA, UC, Santander, Cantabria, 2021.

[2] ARM.com (06-04-2022). *Instruction Set Architecture (ISA)*. [Online]. Disponible: `https://www.arm.com/glossary/isa`

[3] Wikipedia. (07-04-2022). *Complex Instruction Set Computing*. [Online]. Disponible: `https://es.wikipedia.org/wiki/Complex_instruction_set_computing`

[4] Wikipedia. (07-04-2022). *Reduced Instruction Set Computing*. [Online]. Disponible: `https://es.wikipedia.org/wiki/Reduced_instruction_set_computing`

[5] Wikipedia. (11-04-2022). *Arquitectura ARM*. [Online]. Disponible: `https://es.wikipedia.org/wiki/Arquitectura_ARM`

[6] Wikipedia. (11-04-2022). *x86*. [Online]. Disponible: `https://es.wikipedia.org/wiki/X86`

[7] Wikipedia. (03-05-2022). [Online]. *GNU/Linux* Disponible: `https://es.wikipedia.org/wiki/GNU/Linux`

[8] Linux.com (03-05-2022). *What is Linux*. [Online]. Disponible: `https://www.linux.com/what-is-linux/`

[9] Palmer Dabbelt. (22-05-2022). *Booting a RISC-V Linux Kernel*. [Online]. Disponible: `https://www.sifive.com/blog/all-aboard-part-6-booting-a-risc-v-linux-kernel`

[10] Kernel.org. (25-05-2022). *About Linux Kernel*. [Online]. Disponible: `https://www.kernel.org/linux.html`

[11] Wikipedia. (15-04-2022). *Field-prgramable gate array*. [Online]. Disponible: `https://es.wikipedia.org/wiki/Field-programmable_gate_array`

[12] D. Yusseff, «Circuitos embebidos aplicados a equipos médicos», Trabajo final de grado, Universidad Ricardo Palma, Lima, Perú, 2011.

[13] J. Chacon, A. Villamizar y D. Ardila, «Verilog y VHDL diferencias ventajas y desventajas», Universidad Industrial de Santander, Colombia, 2019.

[14] Xilinx. (08-05-2022). *FPGA bitstream*. [Online]. Disponible: `https://www.xilinx.com/htmldocs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html`

[15] YosysHQ. (28-06-2022). *Project IceStorm*. [Online]. Disponible: `https://github.com/YosysHQ/icestorm`.

[16] Wikipedia. (18-04-2022). *System on a chip*. [Online]. Disponible: `https://es.wikipedia.org/wiki/System_on_a_chip`

[17] K.S. Yeo, K.T. Ng, Z.H. Kong, y T.B.Y. Dang, *Intellectual Property for Integrated Circuits*. Edición Ilustrada. J. Ross Publishing, 2010, pp: 13-14.

[18] J. Miura, H. Miyazaki y K. Kise, *A portable and Linux capable RISC-V computer system in Verilog HDL*. ArXiv, preprint arXiv:2002.03576. 2020.

[19] Xilinx Zynq Ultrascale+ MPSoC ZCU102, Documentación del dispositivo, UG1182 (v1.6), 12 de Junio de 2019.

[20] Diligent, *Nexys A7™ FPGA Board Reference Manual*, Revisión del 10 de julio de 2019.

Anexo I: Instalación de RISC-V GNU Compiler Toolchain

Esta instalación se ha realizado sobre una distribución Ubuntu 20.04.4 LTS de Linux. A continuación se detallan los pasos a seguir:

Primero se descargará el código fuente de la cadena de herramientas del repositorio del repositorio de GitHub de la RISC-V Software Collaboration. Para ello se utilizará la herramienta Git:

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
```

Ahora es necesario descargar todas las dependencias para no obtener errores al ejecutar comandos. Estas dependencias se encuentran especificadas en el repositorio GitHub para distintos tipos de sistemas operativos. Se instalarán aquellas necesarias para Ubuntu:

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev
    libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
    patchutils bc zlib1g-dev libexpat-dev
```

Una vez instaladas todas las dependencias, se procederá a la instalación de la cadena de herramientas. Para ello se creará la variable de entorno «INST_PATH» y se le dará el valor del directorio donde se instalará la cadena de herramientas. Por ejemplo, en /opt/riscv. Tras esto se establecerá la configuración mediante el script «config» que proporciona el respositorio.

```
$ export INST_PATH=/opt/riscv
$ cd riscv-gnu-toolchain
$ ./configure --prefix=$INST_PATH
```

Por defecto este script establece el formato RV64-GC y la librería glibc. La G incluida en el nombre del formato viene de *General*, donde se recogen los modulos M, A, F y D. Para construir una cadena de herramientas de 32 bits han de incluirse los parámetros que determinan la arquitectura y el ABI (*Application Binary Interface*). Por ejemplo:

```
$ ./configure --prefix=$INST_PATH --with-arch=rv32gc --with-abi=ilp32d
```

Esta configuración dará lugar a una cadena de herramientas dirigida a una arquitectura de 32 bits con precisión de coma flotante «dura». En este caso se continuará con los parámetros por defecto.

Por último se procederá a la compilación e instalación de la cadena de herramientas con el comando «make linux». En caso de preferir el uso de la librería libc en vez de glibc, en este paso se utilizará la librería «make musl». El proceso de instalación dura unos minutos.

```
$ make linux
```

Tras esto, la cadena de herramientas se encuentra instalada en el directorio $INST_PATH. Es necesario añadir el directorio $INST_PATH/bin a la variable de entorno PATH.

```
$ export PATH=$INST_PATH/bin:$PATH
```

Anexo II: Instalación de la herramienta Vitis

Para instalar la herramienta Vitis es necesario descargar el instalador desde el centro de descargas de la página oficial de Xilinx, como se muestra en la figura 25. Para descargar el archivo es necesario utilizar una cuenta registrada de Xilinx. El registro es gratuito y requiere de un correo electrónico perteneciente a una entidad empresarial o educativa.

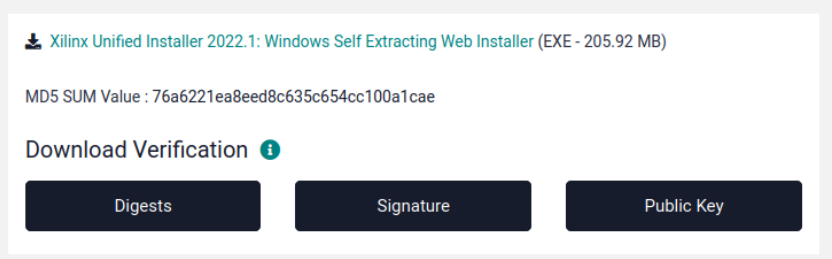

Figura 25: Instalador unificado para Linux

A continuación se accede a la carpeta donde se ha descargado el instalador y se le otorgan permisos de ejecución al archivo descargado. Para iniciar el instalador es preciso ejecutar el archivo. Se ha de tener en cuenta que pueden ser necesarios privilegios de superusuario para instalar el software en ciertos directorios. Tras esto se abrirá el menú de bienvenida (figura 26).

```
$ chmod +x Xilinx_Unified_2022.1_0420_0327_Lin64.bin
$ sudo ./Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

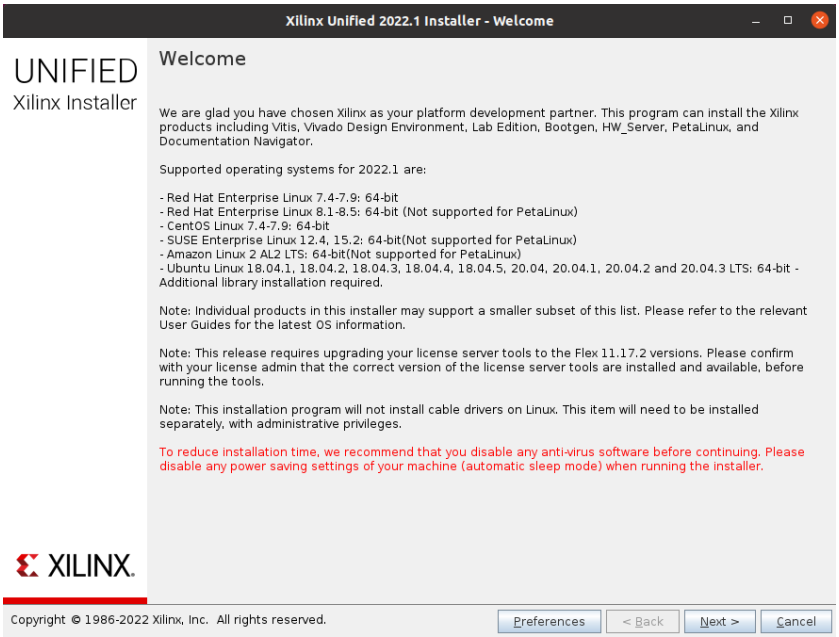

Figura 26: Menú de bienvenida del instalador de Xilinx

Para proceder se ha de seleccionar *Next*. En el siguiente menú se ha de iniciar sesión con una cuenta de Xilinx y se pulsará *Next*. Seguidamente se seleccionará el software Vitis y se

presionará de nuevo *Next* (figura 27). Vitis instala consigo la herramienta Vivado, por lo que no es necesario instalarla por separado.

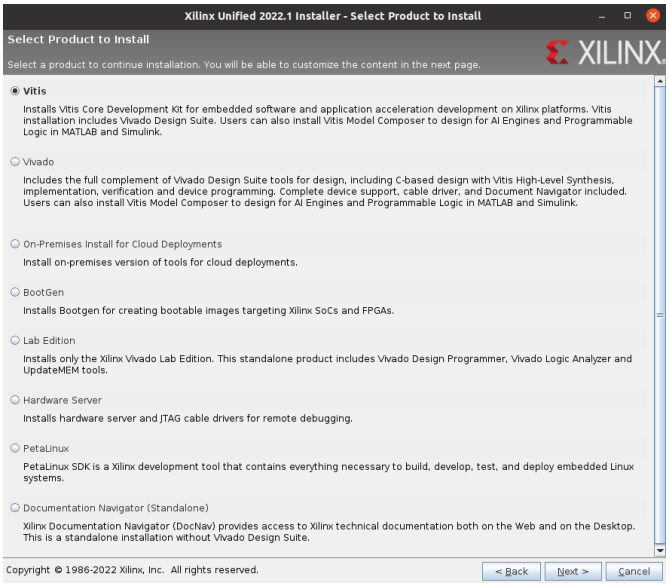

Figura 27: Selección de la herramienta a instalar

El siguiente menú permite seleccionar qué extensiones se quieren instalar para dar soporte a tecnologías o dispositivos. Para el desarrollo del trabajo tan solo es necesaria la instalación del soporte para las FPGA de las *7 series*, por lo que es la única extensión seleccionada (figura 28).

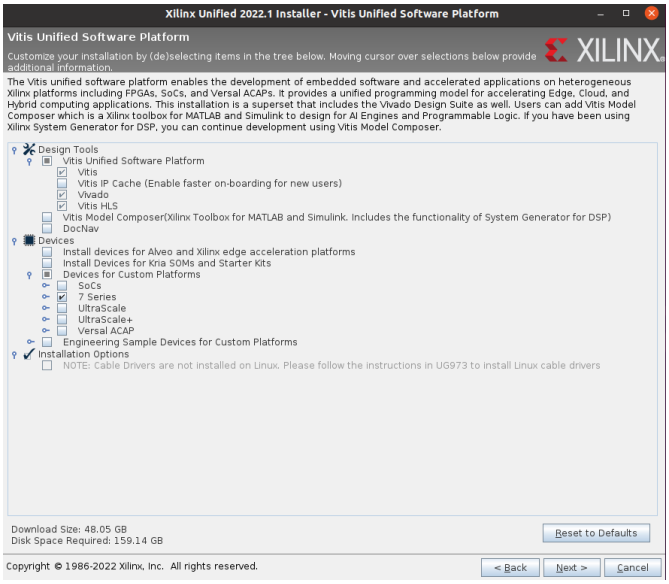

Figura 28: Selección de soporte para dispositivos y tecnologías

Tras seleccionar *Next* será necesario establecer el directorio donde se desea realizar la instalación. Como se observa en la figura 29, se ha decidido instalar dicho directorio bajo /tools/Xilinx. Tras esto se iniciará la descarga y la instalación de la herramienta. Es necesario indicar que en la figura anteriormente mencionada existe un error debido a que la herramienta ya está instalada y no hay suficiente espacio de almacenamiento en el disco.

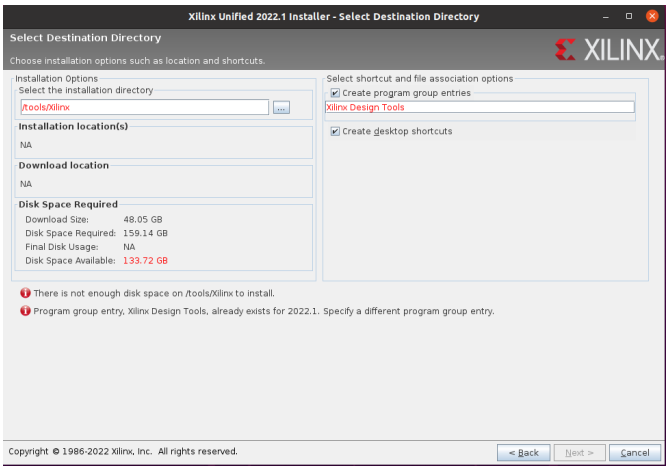

Figura 29: Selección del directorio de instalacion de la herramienta

Anexo III: Instalación de los controladores para dispositivos Diligent

Por defecto la instalación de la herramienta Vivado no cuenta con los controladores necesarios para los dispositivos Diligent ni con los archivos para las placas Diligent. Por lo tanto, es necesaria una instalación manual de los mismos.

En primer lugar se descargan los archivos de las placas Diligen desde el repositorio GitHub de Diligent. Una vez hecho esto, se deben incluir en el directorio /data/board/board_files bajo el directorio de instalación de Vivado.

```
$ git clone https://github.com/Digilent/vivado-boards
$ cd vivado-boards
$ sudo mkdir /tools/Xilinx/Vivado/2022.1/data/boards/board_files/
$ sudo cp new/* /tools/Xilinx/Vivado/data/board/board_files
```

Tras haber incluido todos los archivos para las placas Diligent es necesario instalar los controladores ejecutando el archivo de instalación de la siguiente manera:

```
$ cd
    /tools/Xilinx/Vivado/data/xicom/cable_drivers/lin64/install_script/install_drivers/
$ sudo ./install_drivers
```

También se recomienda permitir al usuario acceder a los dispositivos USB para que no haya problemas para detectar los dispositivos conectados. A continuación se presenta el procedimiento a seguir, en el que $USER debe corresponder al usuario objetivo:

```
$ sudo adduser $USER dialout
```