Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Criptoprocesador ligero en RISC-V de 32 bits basado en Ascon.

Tesis que presenta

Alberto Josué Ortiz Rosales

para obtener el Grado de

Maestro en Ciencias en Computación

Directores de la Tesis

Dra. Brisbane Ovilla Martínez
Dr. Cuauhtemoc Mancillas López

Ciudad de México **Noviembre 2023**

Resumen

El creciente desarrollo del Internet de las Cosas ha generado interés en muchas áreas, en particular en la seguridad. La característica principal de los dispositivos del internet de las cosas es contar con recursos restringidos, sin embargo, de alguna manera deben garantizar la seguridad de la información que procesan. Es de interés trabajar con la ISA RISC-V, ya que, en los últimos años se ha incrementado su popularidad frente otras arquitecturas por su practicidad, modularidad, simplicidad, personalización, sobre todo por ser abierta. Dependiendo de las instrucciones que se implementen de RISC-V esta arquitectura es funcional como plataforma para los dispositivos del Internet de las Cosas. En el presente año el algoritmo Ascon ganó la competencia de estandarización como algoritmo de cifrado autenticado con datos asociados para criptografía ligera. Esta rama de la criptografía se centraliza en la protección de los dispositivos del Internet de las Cosas. A partir de estas bases, la presente tesis presenta el diseño de un criptoprocesador basado en Ascon y su integración como coprocesador acoplado a un núcleo RISC-V por medio de sus instrucciones disponibles en la ISA. El hardware es capaz de realizar los modos de operación del algoritmo Ascon, Ascon-128, Ascon-Hash y otras primitivas de seguridad propuestas. Estos modos dan soporte a la generación de números pseudoaleatorios, enmascaramiento de llaves y generación de llaves. De esta manera se proporcionan los servicios de confidencialidad, autenticación, integridad, aleatoriedad y protección de llaves con criptografía ligera a una arquitectura RISC-V. El diseño fue descrito con Chisel3 e implementado en la placa de desarrollo Arty A7. El análisis del desempeño del hardware contra el software muestra un aumento en la aceleración desde 50x hasta 160x. Así como una reducción del código en memoria para realizar los algoritmos. Hay un aumento del 22 % en LUTs totales y 29 % en los FFs del SoC con el coprocesador con una ronda por ciclo en comparación al SoC por defecto, la versión con tres rondas aumenta a 28 % la utilización dejando el mismo porcentaje de FFs.

Abstract

The rising development of the Internet of Things has gained the interest of many areas, in particular the security area. The main feature of the internet of thing devices is having constrained resources, however, these devices must ensure the security of the data they process. The RISC-V ISA results an interesting topic, given that in recent years its popularity has grown against other architectures, mainly because of its practicality, modularity, simplicity, and customization, specially due to its openness. According to the implemented instructions, RISC-V is possible to be used as a platform for the Internet of Things devices. Recently, the Ascon algorithm has won the standardization competition for the authenticated encryption with associated data algorithm in lightweight cryptography. This cryptography branch focuses on the protection of these devices. From these premises, the current thesis presents the design of a cryptoprocessor based on the algorithm Ascon and its integration as a coprocessor coupled with a RISC-V core via the available instructions in RISC-V ISA. The designed hardware can perform the modes of operations of the Ascon algorithms, Ascon-128, Ascon-Hash, as well as other proposed security primitives. These operation modes support pseudorandom number generation, key wrapping, and key generation. Therefore, the cryptoprocessor offers the services of confidentiality, authentication, integrity, randomness, and key protection with lightweight cryptography on a RISC-V architecture. The design was described in Chisel3 and implemented in the Arty A7 development board. The analysis of the performance showed an acceleration increase from 50x to 160x. Also, a reduction of the code size of the algorithms in the memory. There is a 22 % increase in the LUT's usage and 29 % increase in the FF's usage in the SoC with the coprocessor performing one round per cycle against the default Soc solution, in the 3-round version the LUT's utilization grows to 28 % and the FF's percentage remained the same.

Agradecimientos

En primer lugar dedico esta tesis a mis familia, a mis padres, abuelos y tíos por todo su enorme apoyo durante toda mi vida. A mi hermana por sus consejos en el diseño de la tesis.

Agradezco enormemente a mis coasesores la Dra. Brisbane Ovilla Martínez y el Dr. Cuauhtemoc Mancillas López por sus enseñanzas, apoyo, consejos y tiempo, así como las experiencias que he vivido junto a ellos durante el mi estancia en el CINVESTAV.

Agradezco al Departamento de Computación del CINVESTAV Zacatenco por darme la oportunidad de de estudiar su programa de posgrado, así como al CONACyT por brindarme la beca durante mis estudios de maestría.

Agradezco la compañía de mis amigos André, Rogelio, Alex, Adriana durante mi maestría. Me brindaron ayuda y consejos en el desarrollo de la tesis. Agradezco la enorme ayuda de Katia.

Estoy agradecido de todos los momentos que he vivido estos dos años y de las nuevas personas que he conocido. También agradezco a mis amigos que me han acompañado desde siempre.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Capítulo 1

Introducción

En los últimos años ha habido una creciente demanda por la conectividad de distintos dispositivos, con el fin de integrarlos al Internet. Esto facilita tareas en distintas áreas, desde el hogar hasta actividades complejas como el control de transito. El incremento en el consumo de de los sistemas embebidos al igual que su extensivo uso demandan el diseño de mecanismos de protección frente a una gama de amenazas en seguridad [2]. Estos sistemas deben proveer un alto nivel de protección contra ataques a pesar de contar con recursos de cómputo restringidos [3]. Los sistemas embebidos se enfrentan a distintos retos en seguridad como: comunicaciones inseguras, ambientes hostiles, protecciones inadecuadas de los datos y administración de privilegios de las aplicaciones [4], especialmente si el sistema está conectado.

Cuando estos dispositivos embebidos están conectados al Intenet, se le conoce como Internet de las Cosas *IdC*. Se define al *IdC* como una interconexion de artefactos de uso cotidiano en un red. El *IdC* está formado de una gran diversidad de dispositivos que están presentes en sectores como: hogar, salud, industria, agricultura, ciudades inteligentes, entre otros.

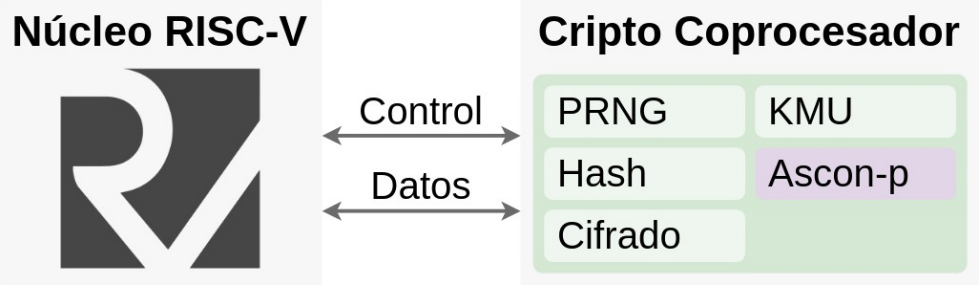
Actualmente, los procesadores de uso general se considera poco apropiado para el área *IdC*, debido a que en el contexto del *IdC* se prefiere hacer un uso eficiente del consumo de energía, reducir costos de producción a gran escala y realizar tareas especificas eficientemente y en tiempo real. Los microcontroladores resultan ser la mejor opción. Cuando existen especificas repetitivas dentro de un sistema es una buena idea implementar las tareas directamente en un hardware dedicado. Este hardware puede acoplarse a un microcontrolador para aumentar su desempeño. En el contexto de seguridad e *IdC*, es conveniente tener un hardware que se encargue de la criptografía.

La arquitectura del conjunto de instrucciones RISC-V desarrollada por la Universidad de California en Berkeley en 2010. Es una excelente opción como procesador base para el diseño de de módulos de hardware que complementen las capacidades del coprocesador. RISC-V tiene la ventaja frente a otras arquitecturas principalmente por ser de código abierto, su simplicidad y modularidad. Esta ISA está estandarizada

para dar soporte a instrucciones de 16 bits hasta 128, extensiones de uso especifico, extensiones personalizadas, instrucciones vectoriales, arquitecturas nultinúcleo, entre otros conjunto de instrucciones que aumenten las capacidades de cómputo. RISC-V tiene sus instrucciones base de 32 bits que son comparables con un microprocesador.

1.1. Propuesta de solución

Con el fin de proteger a los dispositivos del *IdC* se pretende proporcionar una plataforma base que ofrezca seguridad. El diseño del hardware se centra en el algoritmo Ascon que pertenece a la rama de la criptografía ligera, especialmente diseñado para este tipo de dispositivos. Con este algoritmo es posible realizar un coprocesador para acoplarlo a un núcleo RISC-V. Se plantea el uso de un núcleo Rocket de 32 bits como base. Por defecto este núcleo soporta las instrucciones RISC-V denominadas como RV32IMAC. Con este hardware dedicado se busca acelerar y hacer más eficiente los procesos criptográficos. El núcleo se controla e intercambia datos con el criptoprocesador cuando se ejecuta una instrucción criptográfica.



1.2. Objetivos

General

Diseñar e implementar un cripto-coprocesador ligero basado en Ascon incorporado a la ISA de RISC-V.

Particulares

1. Diseñar un módulo de hardware capaz de realizar manejo seguro de llaves simétricas y procedimientos de criptografía ligera basados en Ascon.
2. Implementar e intercomunicar en FPGA el módulo con un núcleo RISC-V, utilizando el espacio disponible de la ISA.
3. Analizar el comportamiento y el desempeño del cripto-coprocesador.

1.3. Organización de la Tesis

En el Capítulo 2 se describe a RISC-V, las extensiones utilizadas en el desarrollo de la tesis. También se incluye el núcleo Rocket y la manera de enlazar un coprocesador.

Posteriormente en el Capítulo 3 se definen las primitivas criptográficas. Se definen a las funciones esponja. Se expone al algoritmo Ascon que esta en proceso de estandarización.

El diseño y los distintos algoritmos que soporta el coprocesador están plasmados en el Capítulo 4. Igualmente en esta capítulo se define e implementa la comunicación con el núcleo Rocket, ya que se implementa el criptoprocesador como un extensiones a la ISA RISC-V. En este se incluyen las propuestas de criptoprocesadores por parte de la industria y academia.

Los resultados de la tesis se encuentran en el Capítulo 5. Incluyen la programación en C para implementar instrucciones personalizadas. Se incluyen los tiempos de ejecución del hardware, la utilización de recursos de la FPGA y pruebas estadísticas para la generación de números pseudoaleatorios.

Finalmente en el Capítulo 6 se encuentra la discusión de los resultados obtenidos junto las conclusiones y trabajo futuro.

Capítulo 2

RISC-V

2.1. ISA RISC-V

La arquitectura del conjunto de instrucciones (ISA, por sus siglas en inglés) RISC-V fue utilizada inicialmente como método de enseñanza como un modelo de diseño por la Universidad de California en Berkeley (UCB) en 2010. Su conjunto de instrucciones está influenciado por la arquitectura MIPS [1], eliminando su complejidad innecesaria y agregando instrucciones propias. Posteriormente ganó popularidad entre la comunidad de desarrolladores, quienes la han puesto en práctica como alternativa a procesadores tradicionales [5]. La visión de este nuevo conjunto de instrucciones es ser comercialmente viable, robusto, flexible y eficiente [6]. La RISC-V está estandarizada para soportar instrucciones de 16-bit hasta 128-bit, extensiones personalizadas, arquitecturas multinúcleo. Está estandarización permite dar soporte a un amplio rango de plataformas, desde sistemas embebidos hasta sistemas de alto rendimiento.

RISC-V tiene cuatro conjuntos de instrucciones base, `RV32I, RV32E, RV64I` y `RV128I`. Para identificar los conjuntos de instrucciones que soporta una plataforma RISC-V se usa un código estándar. Se inicia con la abreviación `RV` para referirse a RISC-V, seguido de la longitud en bits de los registros (32, 64 o 128). El sufijo adjunto indica el conjunto de instrucciones que se está implementando, `I` para las instrucciones enteras y `E` que es una versión reducida de `RV32I` la cual contiene la mitad de los registros (16). Todas las subsecuentes instrucciones se consideran extensiones de conjunto base, la siguiente tabla da una breve descripción de las extensiones [7]. El formato para nombrar el conjunto que se utilice debe seguir el siguiente orden de las extensiones: `RV [32, 64, 128] I, M, A, F, D, G, Q, L, C, ...`, la Tabla 2.1 incluye a las extensiones de mayor importancia. Adicionalmente existe investigación en diversos conjuntos de instrucciones que están en proceso de estandarización como: instrucciones para control, estatus, depuración, operaciones criptográficas vectoriales, entre otras [8]. Todas las implementaciones de la ISA deben de incluir uno de los cua-

[1] Por sus siglas Microprocessor without Interlocked Pipeline Stages, es otro tipo de arquitectura RISC.

tro conjuntos base y pueden o no incluir las extensiones, dependiendo de la aplicación que van a desempeñar.

Tabla 2.1: Conjunto de extensiones a la base de RV.

Conjuntos Base			
ID	Descripción	Estatus	Núm.
RV32I	32 bits	Congelada	49
RV32E	(embebidos) 32 bits, 16 registros	Abierta	49
RV64I	64 bits	Congelada	14
RV128I	128 bits	Abierta	14
Extensiones			
M	Multiplicación y División Entera	Congelada	8
A	Instrucciones Atómicas	Congelada	11
F	Punto Flotante de Precisión Simple	Congelada	25
D	Punto Flotante de Precisión Doble	Congelada	25
G	Abreviación para el conjunto IMAFD	NA	NA
Q	Punto Flotante de Precisión Cuádruple	Congelada	27
L	Punto Flotante Decimal	Abierta	NA
C	Instrucciones Comprimidas	Congelada	36
B	Manipulación de Bits	Abierta	42
J	Traducción Dinámica de Lenguajes	Abierta	NA
T	Memoria transaccional	Abierta	NA
P	Instrucciones SIMD	Abierta	NA
V	Operaciones Vectoriales	Abierta	186
N	Nivel Usuario	Abierta	3
H	Nivel Hipervisor	Congelada	2
S	Nivel Supervisor	Abierta	7

Esta ISA sigue la filosofía RISC (Computadora por Conjunto de Instrucciones Reducidas), la cual se caracteriza por tener un número menor de instrucciones que realizan operaciones simples. Por ejemplo, se requiere un par de instrucciones para cargar dos operandos a los registros, una para realizar una operación en esos registros para finalmente escribirla a memoria con otra instrucción. El Código 2.1 ilustra el anterior ejemplo en lenguaje ensamblador de RISC-V. Para utilizar la aritmética de una forma natural se tienen como máximo tres operandos en las instrucciones. Dos de ellos para los datos de entrada de la operación y uno para guardar el resultado. La arquitectura contiene 31 registros de uso general, uno conectado a cero y un registros que guarda el contador de programa (PC).

Código 2.1: Ejemplo código ensamblador en RISC-V.

```
1    # Carga del primer operando al registro a0
2    lw  a0, op1
3    # Carga del segundo operando al registro a1
4    lw  a1, op2
5    # Suma de ambos registros y guarda el resultado en a2
6    add a2, a0, a1
7    # Almacenamiento del resultado en memoria
8    sw  a2, res_addr[0]
```

2.1.1. Instrucciones base RV32I

La Figura 2.1 ilustra el formato de las instrucciones base de RISC-V. Cada una tiene una longitud de 32 bits, dependiendo del tipo de la instrucción, los bits posteriores al código de operación (opcode) denotan distintas funciones. Los campos `rs1` y `rs2` definen los registro como operandos, `rd` indica el registro que será utilizado para guardar de la operación, el campo `imm` contiene un valor inmediato mientras que `funct3` y `funct7` definen el tipo de operación que se llevará acabo. Los tipos de instrucciones consisten en: `R` para operaciones registro a registro, `I` para cargas inmediatas, `S` para almacenamiento, `B` para saltos condicionales, `U` para cargas con datos de mayor tamaño que las inmediatas y `J` para saltos [2]. En los formatos de instrucciones se puede observar su simpleza, ya que solo se tienen seis formatos con longitud constante que simplifica la decodificación. Tienen campos distintos para el operando fuente y de destino. Se mantienen en la misma posición los operandos especialmente el de destino. Para dejar espacio de decodificación para las extensiones, las instrucciones base usan un octavo del espacio disponible de los códigos de operación [8].

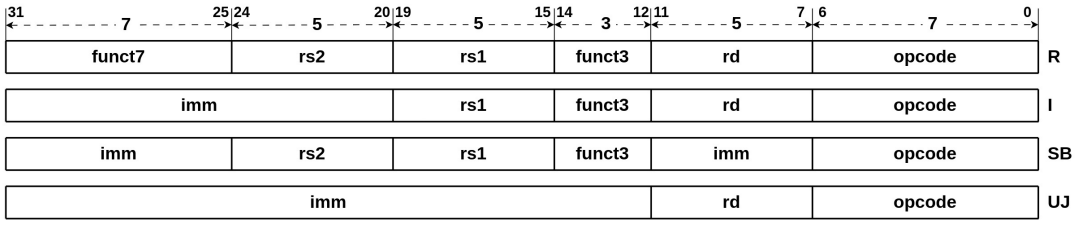


Figura 2.1: Formatos de instrucciones base I de la ISA RISC-V.

Las instrucciones aritméticas `ADD, SUB`, lógicas `AND, OR, XOR` y los corrimientos `SLL, SRL, SRA` cargan dos valores de 32 bits y regresan el resultado a un registro de 32 bits, coincide de una forma natural. Los campos inmediatos tienen el signo extendido en la misma instrucción, lo cual ahorra tiempo de procesamiento. Estas

[2]Para el formato `B` simplemente cambian el orden de los bits del segmento inmediato del formato `S`, lo mismo para el formato `U` con `J`.

instrucciones utilizan los formatos `R` o `I` , en el caso del formato `R` se definen dos registros como operandos, mientras que la variante `I` utiliza un registro y el valor inmediato como operandos[3]. Todas las operaciones hacen uso completo del tamaño de los registros, no existen operaciones en bytes ni medias palabras. También se incluye una instrucción importante `LUI`, que ayuda a subir constates de 20 bits a los registros.

Se provee de la capacidad de leer y escribir datos a memoria de palabra completa `LW, SW`, media palabra `LH, SH` y bytes `LB, SB`. Los datos con signo se almacenan con el signo extendido, por ello existen las versiones de estas instrucciones *unsigned*. Todos los tipos de datos tienen los mismos modos de direccionamiento.

Los saltos condicionales usan el formato `B`. Para realizar los saltos condicionales se comparan dos registros si son iguales `BEQ`, dos desiguales `BNE` mayor o igual `BGE, BGEU` o menor que `BLT, BLTU`. Son utilizadas para el flujo de control (deciden si un procedimiento se ejecutará), ejecuciones condicionales, manejadores de errores y como direcciones de memoria para saltar. A alto nivel corresponden a las construcciones condicionales `if, switch` y a los ciclos `for, while`.

Las instrucciones de transferencia de control consisten en (`JAL, JALR`). El salto incondicional (`JAL`) usa el formato `J`, esta instrucción es utilizada para la llamada a un procedimiento, al regresar se guarda la dirección del `PC + 4` en el registro denominado `ra`. Si se quiere solamente saltar se usa como registro de destino `x0` ya que su valor no puede ser alterado. Para regresar de un procedimiento se usa el salto indirecto que utiliza el formato `I` (`JALR`) dónde la dirección guardada en algún registro más el valor inmediato es utilizado para regresar a la dirección deseada.

2.1.2. Convención de usos de registros

La Tabla 2.2 resume la convención en el nombramiento de los registros en el ABI (*Application Binary Interface*, por sus siglas en inglés), la cual es útil para llamar procedimientos y métodos. Para RISC-V existen distintas ABI si se da soporte a las operaciones con punto flotante `F` y `D`. Son definidas como `ilp32`, `ilp32f`, o `ilp32d`, donde `ilp32` indica que los tipos de datos int, long y pointer son de 32 bits, los argumentos posteriores indican el formato de los datos flotantes. Para compilar correctamente el código en C se ocupa la bandera `-mabi=ilp32` [7].

Instrucciones de Ordenamiento de Memoria

La instrucción `FENCE`, también conocida como una barrera de memoria, es utilizada en arquitecturas multinúcleo para sincronizar los accesos de memoria de los distintas

[3]Estas instrucciones tienen sus versiones inmediatas, para identificarlas se les añade el postfijo `i`.

Tabla 2.2: Convención de registros.

Registros	nemónico	Descripción
x0	zero	Conectado a 0
x1	ra	Dirección de retorno
x2	sp	Apuntador de pila
x3	gp	Apuntador global
x4	tp	Apuntador de hilo
x5	t0	Temporal / link register
x6-7	t1–2	Temporales
x8	s0/fp	Registro guardado / apuntador frame
x9	s1	Registro guardado
x10-11	a0–1	Argumentos de función / valor de retorno
x12-17	a2–7	Argumentos de función
x18-27	s2–11	Registro guardado
x28-31	t3–6	Temporales
PC	PC	Contador de programa

unidades de ejecución. Esto evita un comportamiento inesperado en aplicaciones paralelas, asegurando la integridad de los datos. Esta operación no realiza trasferencias de datos. Sin embargo, ordena las instrucciones de acceso a memoria que realizan los procesos. Además `FENCE` ayuda a la sincronización de los dispositivos de entrada y salida (`I/O`), manejo de interrupciones, dispositivos y coprocesadores. Con la `FENCE` se asegura que las instrucciones de lectura y escritura se ejecutan en el orden establecido.

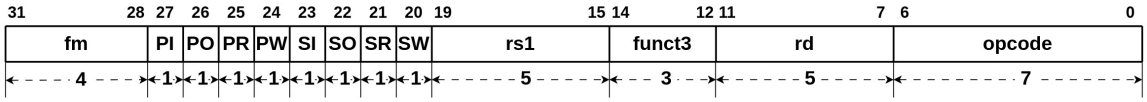


Figura 2.2: Instrucciones tipo FENCE.

El formato de la instrucción `FENCE` se presenta en la Figura 2.2, el campo `fm` indica el modo de `FENCE : 0000` ordena todas las operaciones en memoria en el proceso predecesor antes que todas las operaciones de memoria que del proceso sucesor. `(P)` predecesor, `(S)` sucesor, `(I)` entrada para dispositivos, `(O)` salida para dispositivos, `(R)` lecturas a memoria y `(W)` escritura a memoria.

Con estas instrucciones de base entera `RV32I`, más su banco de 32 registros se provee soporte para la aritmética entera simple, operaciones lógicas, accesos a me-

moria y control del flujo de datos en software. Esta base es especialmente útil para los dispositivos y aplicaciones embebidas ya que sirven como base para desarrollo de software. Mediante el uso de estas instrucciones se pueden emular la funcionalidades de casi todas las extensiones. Estas extensiones facilitan la aritmética de otros tipos de datos, realizan paralelismo a nivel de instrucciones, reducen el tamaño de la codificación, entre otras. Al implementar las extensiones en hardware se incrementa el poder de procesamiento del procesador.

2.1.3. Modos de direccionamiento de memoria

Hay un espacio direccionable de 2^{XLEN} donde $XLEN$ indica la cantidad de bits de las instrucciones base (32, 64). Se define como una palabra a 32 bits, media palabra a 16 bits, a un byte como 8 bits y doble palabra a 64 bits. Se puede acceder a la memoria de forma explícita e implícita (para cargar las instrucciones a ejecutar)[7]. El orden de los datos en la memoria es de tipo *litte endian*, es decir, que el byte menos significativo de las palabras se almacena en la dirección más baja, en consecuencia el byte más significativo se almacena en la dirección más alta.

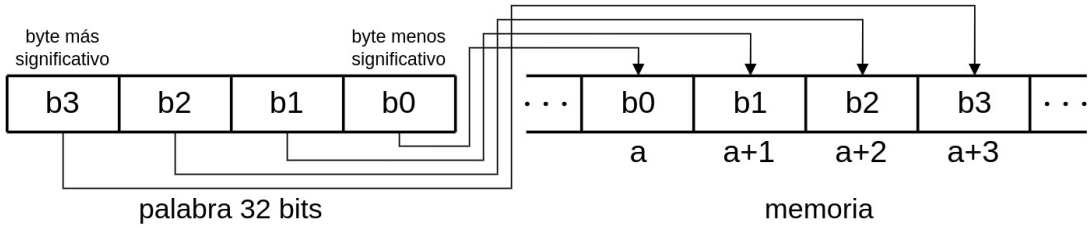


Figura 2.3: Representación *little endian*.

En las aplicaciones generales siempre es conveniente operar datos, constantes o direcciones con un número mayor a 32 bits. [6] Por ello, existen diferentes modos de direccionamiento en RISC-V: direccionamiento directo, direccionamiento por registro, direccionamiento por registro base y direccionamiento relativo al PC. Estos modo están ilustrados en la Figura 2.4.

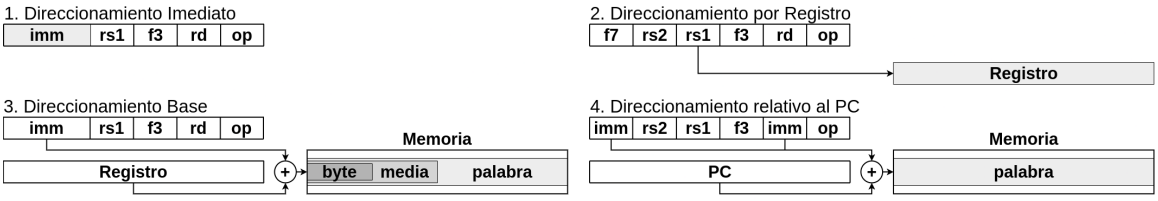


Figura 2.4: Modos de direccionamiento de Memoria.

Dentro de las instrucciones es posible incluir un dato constante de 20 bits para la instrucción `lui` o 12 bits para las operaciones aritméticas inmediatas. El compilador se encarga de descomponer constantes grandes en partes para después juntarlos en un solo registro. Los saltos condicionales tienen 12 bits de valor que permite realizar un salto relativo en el intervalo definido por las direcciones –4096 a 4094 [6]. En el caso de los saltos incondicionales tienen 20 bits para definir la dirección de memoria. Sin embargo, esto limita a los programas para ello se usa direccionamiento base, donde se acumulan el valor inmediato más una dirección base definida en un registro. En la mayoría de las aplicaciones se quiere realizar un salto tomando como base el PC, en este caso la dirección base es aquella contenida dentro del PC y el valor inmediato indica el salto que se dará.

2.1.4. Extensiones

Multiplicación y división entera M

Esta extensión está identificada con la letra `M` y define las instrucciones para multiplicar o dividir dos valores enteros contenidos en los registros. Estas instrucciones tienen un único formato que es el `R`, donde `rs1` es el multiplicando (dividendo), `rs2` es el multiplicador (divisor), y `funct3` define que tipo de multiplicación o división se realiza. En el caso de la multiplicación se incluye las instrucciones `MUL` que regresan la parte baja de una multiplicación de tamaño $XLEN * 2$, mientras que `MULH, MULHS, MULHSU` retornan la parte alta de la multiplicación, sin signo $*$ sin signo, con signo $*$ con signo, `rs1` con signo $*$ `rs2` sin signo, sucesivamente. En el caso de la división se tiene `DIV, DIVU` y para calcular el residuo se tiene `REM, REMU`.

Instrucciones Atómicas A

Estas instrucciones leen, modifican y escriben atómicamente la memoria, con ellas se sincronizan múltiples procesos en los núcleos que se ejecutan en el mismo espacio de memoria. Contiene dos instrucciones `LR.W, SC.W`, carga reservada y escritura condicionada, utiliza el formato `R` con una modificación el `funct7`. Los dos bits menos significativos de `funct7` se utilizan para definir `aq` y `rl` mientras que los cinco bits restantes ahora definen `funct5`. La instrucción `LR.W` carga una palabra a los registros y marca la dirección de memoria como reservada, mientras que `SC.W` guarda un dato a una región de memoria previamente marcada como reservada. Los dos bits `aq` de *acquire* y `rl` de *release* indican cuando un espacio de memoria sigue un orden de ejecución, es similar a los semáforos en los procesos. Para `rl` indica que las operaciones que anteceden a una instrucción `SC.W` no se reordenan y `aq` indica que las instrucciones siguientes no se reordenan. Es una forma más simple de la instrucción `FENCE`.

Las instrucciones `AMO` *Atomic Memory Operation* (operaciones atómicas en memoria) como: `AMOSWAP.W`, `AMOADD.W`, `AMOAND.W`, `AMOOR.W`, `AMOXOR.W`, `AMOMAX[U].W` y `AMOMIN[U].W` realizan la sincronización. Estas instrucciones se definen con el formato

R. En general leen un valor especificado en rs1, le aplican una operación con el dato en rs2, regresan el resultado en rd y escriben el nuevo valor en la dirección contenida en rs1.

Registros de control de estado Zicsr

La ISA define 4096 registros de control de estatus (CSR's) asociados a cada unidad de ejecución, son esenciales para monitorear el desempeño, comportamiento y estado del procesador. Son distintos al archivo de registros dentro del pipeline. Tienen distintas funcionalidades como: manejar las excepciones, establecer configuraciones, comunicación entre el hardware, contadores, proteger a la memoria de accesos, controlar de los modos de privilegio, entre otros. Además, se pueden establecer CSR personalizados para controlar o monitorear algún evento de interés. Con estas instrucción se puede leer, escribir, establecer y limpiar los CSR

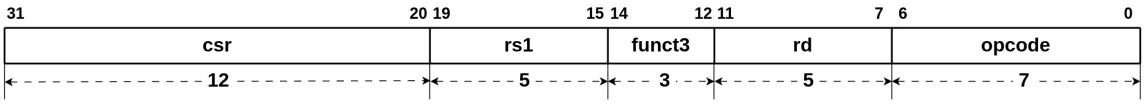


Figura 2.5: Instrucciones para los registros de control y estatus.

La ISA proporciona 32*64 contadores de desempeño y temporizadores de los registros 0xC00 - 0xC1F. Los principales son CYCLE, TIME y INSTRET con los cuales se puede contar los ciclos, reloj de tiempo real e instrucciones retiradas, mientras que los otros si son implementados funcionan como contadores de eventos [7]. Para obtener información y establecer los registros se utilizan la instrucción CSRRS, lectura y establecimiento. La Figura 2.5 muestra el tipo de instrucciones para modificar los registros CRS este coincide con el tipo de formato I, con el cambio que el campo inmediato ahora contiene la dirección del registro.

Los diseñadores definen un conjunto de extensiones para el soporte de software. La combinación IMAFD de instrucciones proporciona las instrucciones suficientes para soportar la gran mayoría de software sin emular operaciones. Se incluyen las instrucciones aritméticas enteras I, multiplicación y división entera M, instrucciones atómicas A, más las instrucciones para operar con número flotantes de simple y doble precisión. Además, se incluyen las instrucciones de control y estatus Zicsr más las barreras de memoria Zifencei. A esta combinación de instrucciones se le denominada de propósito general G, ya que gran parte del software desarrollado se beneficia enormemente en desempeño si la arquitectura cuenta con estas instrucciones. Las demás extensiones son exclusivas de su dominio de aplicación.

2.1.5. Instrucciones comprimidas C

Para mejorar el desempeño, reducir el tamaño del código y tener un consumo eficiente de energía se agregan las instrucciones comprimidas `C`, considerando un ligero aumento en la complejidad del hardware. Esta extensión puede agregarse a cualquier conjunto base (`RV321`, `RV64I` formando el conjunto `RVC`. Con estas instrucciones se puede reducir el tamaño del código ya que alrededor $50\,\% - 60\,\%$ del código puede ser remplazado por estas instrucciones, reduciendo de un $25\,\% - 30\,\%$ el tamaño.

Esta extensión ofrece una versión de 16 bits de las instrucciones más comunes de la base de 32 bits. Para utilizar este tipo de instrucciones las operaciones deben: contener valores inmediatos pequeños al igual que un desplazamiento corto en los saltos, uno de los registros a usar es `x0`, registro `x1` para *link register*, `x2` para el *stack pointer*, el registro destino `rd` es el mismo que el primer registro fuente `rs1` o los registros usados son los ocho más populares. Los registros más populares son `x8 - x15` y `f8 - f15` para punto flotante. Pueden mezclarse con las demás instrucciones, estas están alineadas en cualquier límite de 16 bits.

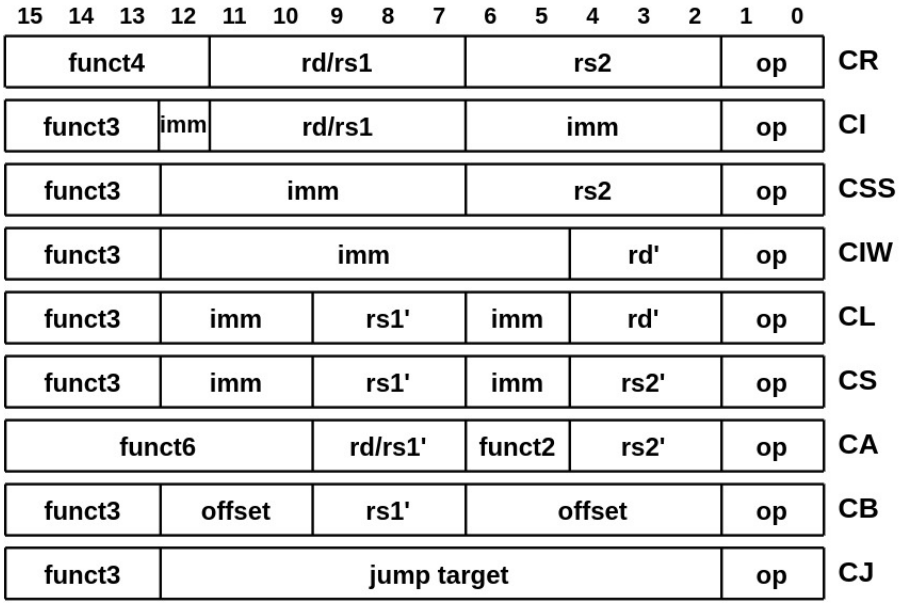


Figura 2.6: Formato de las instrucciones comprimidas.

En el caso de la extensión `C` se tienen distintos formatos que realizan las operaciones análogas a las instrucciones base. La Figura 2.6 presenta todos los formatos del conjunto de instrucciones comprimidas. `CR` es el formato tipo registro comprimido, `CI` para las instrucciones con direccionamiento inmediato, `CSS` almacenamiento relativo al *stack pointer*, `CIW` inmediato amplio, `CL` carga, `CS` almacenamiento, `CA` aritméticas, `CB` saltos condicionales y `CJ` saltos incondicionales. Es trabajo del compilador y del *linker* de remplazar las instrucciones de 32 bits por su representación en la extensión

C. El Código 2.2 muestra un pequeño ejemplo del uso de esta extensión[8].

Código 2.2: Ejemplo de remplazo por instrucción C.

```
1    # Instrucción aritmética inmediata 32 bits.
2    addi a4, x0, 1
3    # Remplazada por una carga inmediata de 16 bits al registro a4.
4    c.li a4, 1
```

2.2. Rocket-Chip

Uno de los proyectos que han surgido en torno a la implementación de la ISA RISC-V es Rocket-Chip. Este proyecto permite crear SoCs[4] con un núcleo Rocket o BOOM, ambos desarrollados y refinados por la UCB. Con él se pueden instanciar los núcleos, crear arquitecturas de 32 bits y 64 bits, elegir las extensiones a las que se desee dar soporte, modificar el sistema de memoria, agregar aceleradores, crear arquitecturas multinúcleo, entre otras. Cuenta con las herramientas necesarias para crear simuladores y emuladores que permiten agilizar el desarrollo de procesadores. Actualmente el proyecto ha pasado a las manos de la empresa SiFive que se encarga de su mantenimiento [9].

2.2.1. Núcleo Rocket

Rocket es un núcleo en orden, escalar con un pipeline de cinco etapas. Este núcleo está descrito en el lenguaje de descripción de hardware (HDL) Chisel. Rocket se considera como un generador más que un núcleo ya que es capaz de implementar distintos conjuntos base y extensiones de acuerdo con el usuario. Por defecto está implementada la versión `RV64GC` más el modo de privilegio `U`. Sin embargo, puede adaptarse para las versiones RV32GC y RV32E, niveles de privilegio supervisor e hipervisor, punto flotante, instrucciones vectoriales, coprocesadores, aceleradores, al igual que, extensiones y CSR personalizadas. Se puede definir las características de la unidad de punto flotante, los ciclos de multiplicación, protecciones a memoria, el sistema de memoria (tamaño de la cache, jerarquía de memoria, políticas de remplazo, políticas de coherencia), agregar historial de los destinos de los saltos(BTH por sus síglas en inglés), buffer de destino de saltos(BTB), memoria virtual (VM), entre otros.

[4]Sistema en un Chip, es un circuito altamente integrado que incorpora todos lo componentes necesarios para un sistema en especifico. Dentro de ellos se pueden encontrar unidades de procesamiento, memorias, dispositivos de entrada y salida, entre otros.

Sin embargo, a la fecha no hay una documentación sólida y detallada de cómo está implementada la arquitectura Rocket, ni de cómo están interactuando las señales internas de núcleo. Por lo tanto, se usa la información disponible de las plataformas de la gamma E300 de SiFive que incluyen un coreplex (procesador) E31. Dentro del el procesador se encuentra el núcleo Rocket que cuenta de un pipeline de cinco etapas: búsqueda de instrucción, decodificación de instrucción y búsqueda de registro, ejecución, acceso a memoria y actualización del estado. La Figura 2.7 muestra las cinco etapas del pipeline. Cuando el pipeline está lleno tiene una latencia de un ciclo, en instrucciones como `LW` tiene que esperar dos ciclos, para multiplicaciones y divisiones puede tardar de dos a 34 ciclos y las lecturas a CSR tardan tres ciclos.

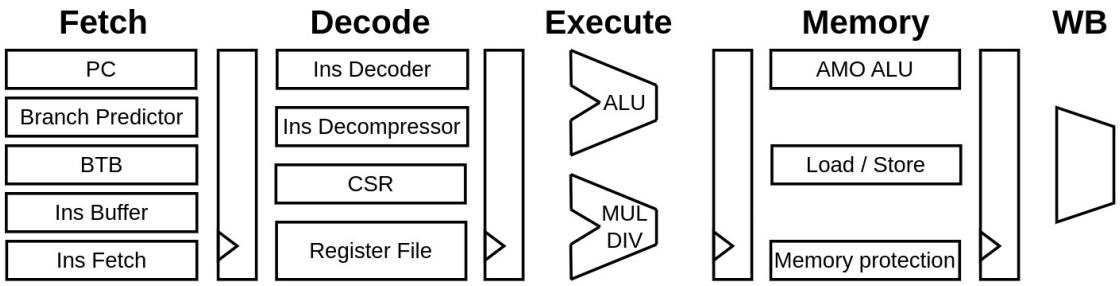


Figura 2.7: Diagrama del pipeline del núcleo Rocket.

Además del pipeline, el procesador contiene distintas unidades para el sistema de memoria de instrucciones y datos, búsqueda de instrucciones e interrupciones. El sistema de instrucciones de memoria puede incluir ROM, RAM dedicada, caché de instrucciones con una latencia por acceso de un ciclo. Se puede configurar el tamaño, el mapeo directo o asociativo de la caché. La unidad buscadora de instrucciones puede contener un predictor de saltos, igualmente configurable su tamaño. El predictor tiene una latencia de un ciclo, mientras que una mala predicción tiene una penalidad de tres ciclos. La caché de datos puede ser modificada al igual que la memoria de instrucciones. [10]

2.2.2. Chisel

Chisel (Constructing Hardware In a Scala Embedded Language), es un lenguaje de descripción de hardware (HDL) que está embebido en Scala. Su característica principal es el paradigma de programación orientado a objeto y la programación funcional, heredado de Scala. En pocos términos es una biblioteca que tienen definidas las clases y objetos que permiten generar módulos de hardware. El compilador de Scala compila el hardware, generando una representación intermedia de RTL para circuitos digitales (FIRRTL), formaliza la representación de circuitos digitales entre Chisel y Verilog. Ese archivo FIRRTL puede ser interpretado por Treadle para simular el circuito y

generar los archivos de simulación o emitir Verilog sintetizable. Usando alguna herramienta como: Quartus o Vivado se puede sintetizar el circuito[11].

2.2.3. Rocket-Chip SoC

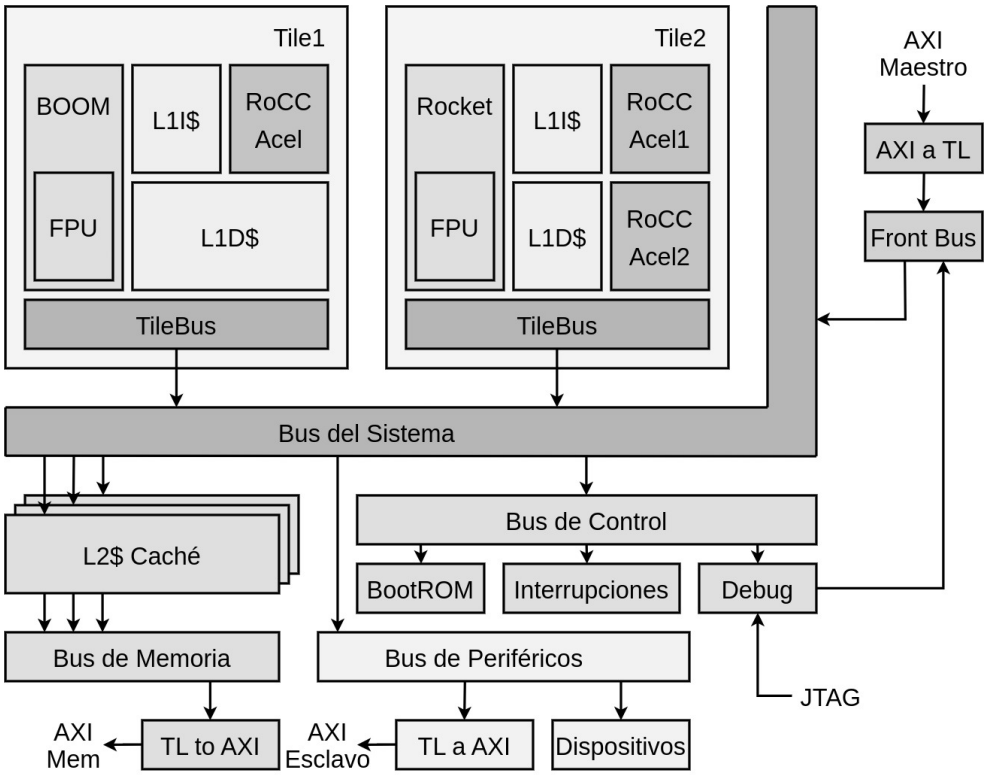


Figura 2.8: Representacipon de un SoC Rocket-Chip.

Rocket-Chip es diferente al núcleo Rocket, ya que incluye distintas construcciones para generar un SoC además del núcleo. Permite crear procesadores multínúcleo, agregar instrucciones personalizadas, definir y ajustar el sistema de memorias, adicionar aceleradores de hardware, establecer los dispositivos I/O, entre otras modificaciones. Permite la simulación del sistema, así como prototiparlo en FPGAs. Cada núcleo es agrupado en un *tile* [5], su interfaz de comunicación con el bus del sistema y opcionalmente un coprocesador. Puede implementar otros unidades de ejecución además del Rocket como Ibex o BOOM. Para lograr una comunicación exitosa se usan las interfaces definidas por Rocket-Chip en componentes para generar un entorno donde se pueden conecta. Cuando tienen están definidos los componentes con forme a la interfaz puede conectarsse cambiando los archivos de configuración [12]. En la Figura 2.8

[5]Rocket-Chip define a un *tile* como un componente que contienen una unidad de ejecución, caches y tablas de paginación

se puede observar una representación gráfica de los distintos módulos y configuraciones posibles.

2.3. Coprocesador para Rocket personalizado (RoCC)

Una de sus metas de la ISA RISC-V es que evita el sobre-diseño para un estilo particular de microarquitectura o una implementación tecnológica, a pesar de ello se puede realizar una implementación eficiente de estas. RISC-V tiene espacio para codificar aceleradores personalizados, extensiones estándar, para dar soporte al desarrollo de software de propósito general. Debido a esto, en el documento del estándar de la ISA RISC-V se tienen reservados códigos de operación (opcode) libres para instrucciones personalizadas, estos corresponden los códigos custom-0 $\mathbf{0001011_2}$, custom-1 $\mathbf{0101011_2}$, custom-2 $\mathbf{1011011_2}$ y custom-3 $\mathbf{1111011_2}$ [6]. En el caso de custom-2 y custom-3 su uso está reservado para la futura estandarización de la ISA base `RV128`.

En [1] se define los distintos niveles de integración de procesamiento reconfigurable, tomando como característica la jerarquía de memoria. Así como los diferentes niveles en la jerarquía de memoria se definen por su mayor integración a la unidad de procesamiento, también define el tipo de coprocesador que se implementa. Cuando está fuertemente acoplado indica que la comunicación entre el coprocesador y la unidad de procesamiento intercambian datos a través de registros, limita su ancho de banda, aun así la unidad de procesamiento ejerce un mayor control sobre el coprocesador. Los demás niveles se caracterizan por un incremento en la cantidad de datos a procesar ya que se comunican directamente con los niveles de cache o memoria principal, la Figura 2.9 ejemplifica esta jerarquía. Cuando es un coprocesador desacoplado este necesita del mecanismo de acceso directo a memoria (DMA por sus siglas en inglés) para transferir los datos entre la memoria principal y el acelerador periférico. Esto libera de carga al núcleo y le indica cuando los datos son validos. En el caso de Rocket-Chip es posible crear coprocesadores fuertemente acoplados y ligeramente acoplados con el RoCC, para los coprocesadores desacoplados se necesita un controlador DMA para implementar el acelerador.

En Rocket-Chip la interfaz RoCC utiliza los códigos de operación custom-0,1,2,3 para controlar los coprocesadores que se incorporan junto al núcleo. El formato que siguen dichas instrucciones corresponde al tipo `R` Figura 2.10, el cual referencia a dos registros fuente y un registro de destino. Siguen el ABI estándar donde se utilizan `a0, a1` y `a2`, es decir, los registros del 10 al 11. En este caso quedan libres los segmentos de `funct7` y `funct3`. Los siete bits de `funct7` son totalmente libres y son utilizados en las instrucciones personalizadas que controlaran al coprocesador. Esto da un total

[6] Los dos bits menos significativos del código de operación, [1:0] al estar en alto, corresponden al conjunto mayor de códigos de operación del conjunto RV32G y RV64G

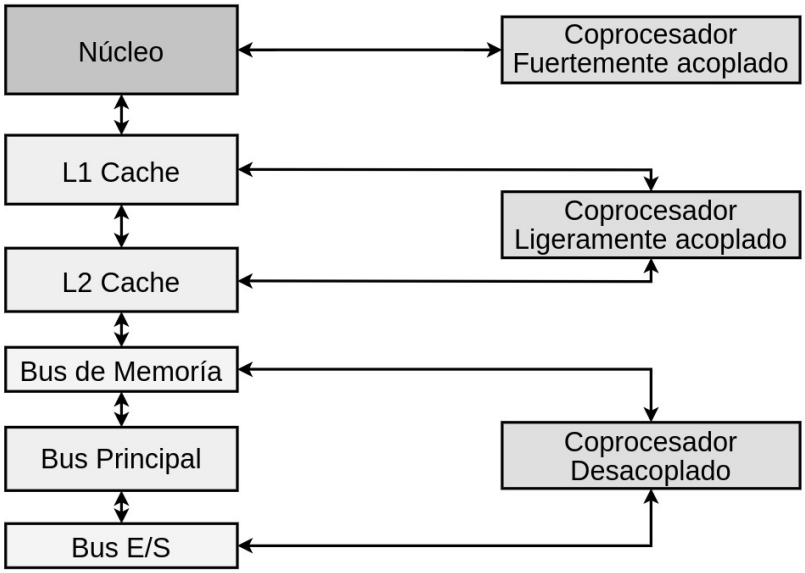


Figura 2.9: Representación de los diferentes niveles de integración de coprocesadores [1].

de $2^7 = 128$ instrucciones distintas por coprocesador. Mientras que `funct3` al estar en alto los bits indica si se usa el registro de destino `xd` o los registros fuente `xs1, xs2`. Por cada núcleo Rocket pueden coexistir hasta cuatro coprocesadores RoCC.

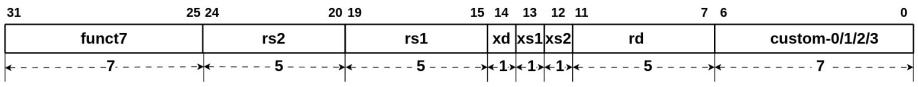


Figura 2.10: Formato de las instrucciones para controlar coprocesadores RoCC.

2.3.1. Señales de control y flujo de datos

Con la interfaz RoCC facilita la comunicación desacoplada entre el procesador y el coprocesador adjunto. Este coprocesador acepta las instrucciones que lleguen desde el núcleo. Esta interfaz permite que el coprocesador tenga acceso a la memoria de datos del núcleo [12]. Esta interfaz está dividida en `cmd`, `resp` y `mem`, la primera controla y manda los datos de registros hacia el coprocesador, `mem` se encarga de la comunicación entre la memoria L1 de datos con el coprocesador y `resp` indica el término del procesamiento al núcleo y devuelve algún valor en el registro de retorno. Además de comunicarse con la memoria L1, también puede comunicarse con la unidad de punto flotante, la caché L2 y el PTW [13]. La Figura 2.11 muestra la dirección de los buses para la comunicación del coprocesador y la lista subsecuente detalla los contenidos dichos buses. Se utiliza `s` para salida y `e` entrada.

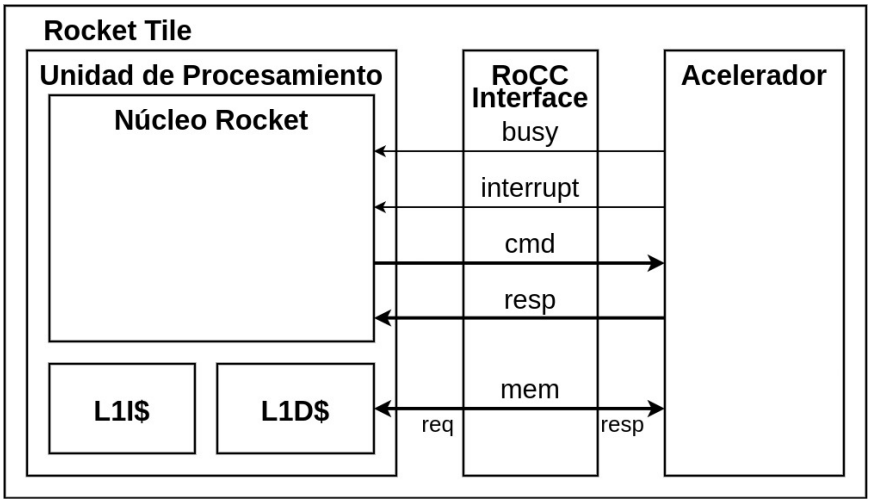


Figura 2.11: Interfaz RoCC.

- `busy (s)`: Indica cuando está procesando.
- `interrupt (s)`: Indica al núcleo que hay una interrupción.
- `cmd`: Controla el acelerador RoCC.
  - `ready (s)`
  - `valid (e)`
  - `inst (e)`: Contiene los 32 bits de la instrucción.
    - `func7 (6:0)`: Establece que tipo de operación se realiza por los coprocesadores.
    - `rs2, rs1, rd (4:0)`: Contiene los identificadores de los registros del núcleo.
    - `xs1, xs2, xd` : Bits que determinan que operandos están activos.
    - `opcode (6:0)`: Identifica que coprocesador se usa.
  - `rs1 (e) (XLEN-1 :0)`: Contiene el valor que está contenido en el registro rs1.
  - `rs2 (e) (XLEN-1 :0)`: Contiene el valor que está contenido en el registro rs2.
- `resp`: Regresa un valor a los registros del núcleo.
  - `valid (s)`: El valor es válido.
  - `rd (s) (4:0)`: Indica el registro donde se regresa el valor.
  - `data (s) (XLEN-1 :0)`: Valor calculado.
- `mem`: Comunicación con la memoria L1 de datos.
  - `req` : Envía una petición a la memoria.
    - `ready (e)`

  - `valid (s)`
  - `addr (s)` : Dirección de memoria.
  - `tag (s)` : Identificador único (Ver Capítulo 5).
  - `cmd (4:0) (s)`: Tipo de petición $\mathbf{00000_2}$ load $\mathbf{00001_2}$ store.
  - `typ (s)` : Tamaño del dato de respuesta $\mathbf{000_2}$ 8 bits, $\mathbf{001_2}$ 16 bits, $\mathbf{010_2}$ 32 bits, $\mathbf{011_2}$ 64 bits.
  - `phys (s)` : Si necesita traducción de memoria virtual.
  - `data (s)` : Dato a almacenar.

- `resp` : Devuelve la petición.
  - `valid (e)`
  - `addr (e)` : Dirección de memoria de la respuesta.
  - `tag (e)` : Identificador único.
  - `cmd (e)` : Tipo de petición.
  - `typ (e)` : Longitud del dato.
  - `data (e)` : Dato de carga.

Capítulo 3

Criptografía

La criptografía se define como el arte de escribir con llave secreta por el diccionario de la lengua española. Su descomposición por sus etimologías es *kryptos*, oculto y *graphia*, escritura. Actualmente, el área de la criptografía abarca más que solo ocultar mensajes con una llave (cifrar). En la actualidad aborda mecanismos para garantizar la integridad, técnicas para el intercambio de llaves secretas, protocolos de autenticación, voto electrónico, criptomonedas, entre otros. Ahora puede definirse como el estudio de técnicas matemáticas para salvaguardar la información digital, sistemas y procesamiento distribuido contra ataques maliciosos [14].

La criptografía tiene dos áreas principales, la criptografía simétrica y la criptografía asimétrica. La característica principal de la simétrica yace en que se utiliza la misma llave para cifrar y descifrar, mientras que la criptografía asimétrica o de llave pública utiliza una llave para cifrar diferente para descifrar. La criptografía simétrica comprende, los cifradores por bloque y por flujo de datos, las funciones hash, las funciones hash con llave, el cifrado autenticado, entre otros. La criptografía simétrica se basa en operaciones simples utilizando una llave para crear distribuciones pseudoaleatorias de permutaciones, mientras que la criptografía asimétrica se basa en problemas matemáticos donde las funciones son fáciles de obtener si se cuenta con el par de llaves pública y privado pero resulta computacionalmente infactible calcular su resultado si no se cuenta con la llave privada.

Criptografía Simétrica

El mensaje que se va a cifrar se denomina texto plano P. El mensaje que se cifra se le conoce como texto cifrado C. Un cifrador E puede realizar la acción de cifrar transformando mediante una llave K a P en C y su acción inversa descifrar utilizando la misma K, restaurando P a partir de C. La función se representa de la siguiente manera $C = E(K, P)$ y $P = D(K, C)$ para descifrar.

Los cifradores por bloque están compuestos por un algoritmo de cifrado $C =$

$E(K, P)$ y otro para descrifrar $P = D(K, C)$, el cifrador por bloques es capaz de producir permutaciones pseudoaleatorias que están determinadas por la llave, es decir, una misma llave junto a los mismos bloques de mensaje produce la misma permutación. Tiene como principal característica sus valores del tamaño de bloque (la cantidad de bits que pueden procesar a la vez) y el tamaño de la llave. Su construcción interna de los cifradores por bloque se reduce a calcular una secuencia de rondas R_i. Que consisten en una transformación simple de especificar, implementar y analizar. Estas rondas deben ser invertibles para poder descrifrar.

Con los cifradores por bloque se pueden definir otros esquemas como el cifrado autenticado (AE). Este tipo de cifrado no solo proporciona confidencialidad de los datos, también su integridad. Se utiliza una etiqueta de autenticación como mecanismo para validar la integridad de los datos, dicha etiqueta solo puede ser obtenida si ambas partes comparten la misma llave secreta. Por lo tanto se define al cifrado como $(C, T) = E(K, P)$ y el descifrado como $(C, T) = D(K, P)$. Algunos esquemas de cifrado autenticado incluyen los datos asociados, que no son confidenciales pero su integridad se garantiza, a este esquema se le denomina cifrado autenticado con datos asociados (AEAD), $(C, T) = E(K, P, A)$ en el cifrado y $(P, \perp) = D(K, C, A, T)$. Donde $\perp$ es la bandera de validación cuando la etiqueta del cifrado y descifrado coinciden.

La generación de llaves criptográficas (KG, por sus siglas en inglés) es la parte fundamental de la criptografía, las llaves deben generarse aleatoriamente para ser impredecibles y secretas. Las llaves simétricas siempre tienen la misma longitud dependiendo del algoritmo que se utilice, la longitud más común es 128 para una seguridad de 128 bits, en ciertos cifradores por bloque como AES, es decir, que existen 2^{128} llaves distintas. Una forma de generarlas es usar algún algoritmo criptográfico de generación de número pseudoaleatorios [15]. Para proteger las llaves y mantenerlas en secreto usa el cifrado de llaves con una llave secundaria.

Es necesario que los sistemas criptográficos tengan aleatoriedad para ser seguros. La generación de aleatoriedad está compuesta por una fuente de entropía, un generador de número aleatorios (RNG) y un algoritmo que produzca bits aleatorios de alta calidad a partir de la fuente de entropía, denominado como generador de números pseudoaleatorios (PRNG). Los PRNGs son inicializados con una semilla que es secreta o generada por un RNG, posteriormente expanden esa semilla en una secuencia de bits. Los RNG generan bits aleatorios lentamente ya que necesitan que la entropía se acumule hasta tener el número de bits requeridos, por ello los PRNG pueden producir secuencias de bits mucho más rápido a partir de la semilla. De esta manera los PRNG cubren la demanda de bits pseudoaleatorios.

El enmascaramiento de llaves (KW, por sus siglas en inglés) es una técnica para proteger una llave. Para ello se necesita de una segunda llave que siempre está disponible, con ella se cifra para protege a la primer llave. La función principal es proteger

a las llaves en almacenamientos no seguros. El estándar es AES en modo KW que es una forma de cifrado autenticado.

Las funciones hash son utilizadas en distintas aplicaciones como firmas digitales, cifrado de llaves públicas, verificación de la integridad, autenticación de mensajes, protección de contraseñas, entre otras. A diferencia de los cifradores éstas producen una salida pequeña de tamaño fijo con una entrada de cualquier tamaño. Las preimágenes de las funciones hash, $Hash(M) = H$, son imposibles de obtener dado un mensaje aleatorio, por eso se le conoce como funciones de solo ida.

Funciones Esponja

Las funciones esponja también denominada construcción esponja es un modo de operación de una permutación pública [1] de longitud fija que se itera para tomar una entrada de longitud variable y producir una salida de longitud variable. Pueden ser utilizadas en distintas aplicaciones de la criptografía simétrica, principalmente en funciones hash y cifradores por flujo de datos. Otras aplicaciones son la generación de bits pseudo aleatorios, la derivación de llaves, el cifrado, los códigos de autenticación de mensajes y el cifrado autenticado.

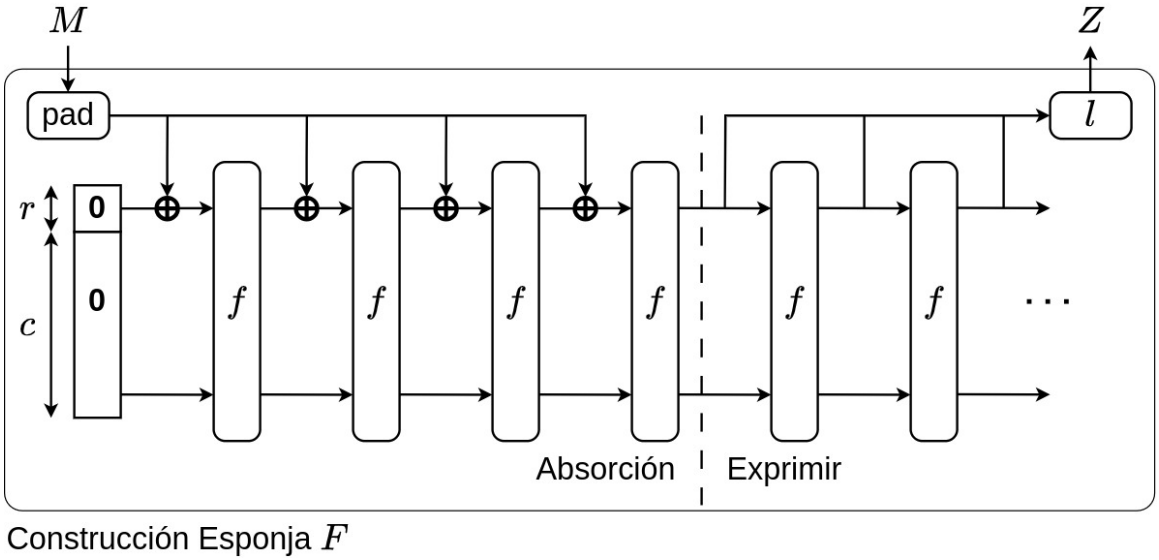


Figura 3.1: Representación de la construcción esponja.

La Figura 3.1 ilustra el modo de operación de una permutación en construcción esponja. Es es una función F con un mensaje de entrada M y salida Z de longitud

[1] Es una permutación criptográfica sin llave diseñada para comportarse como una permutación aleatoria [16].

arbitraria sobre una permutación f de longitud constante de b bits. La permutación opera en un estado compuesto de $b = c + r$ bits donde c es la capacidad y r es la taza. La construcción esponja se divide en dos partes: absorción y exprimir. La parte de absorción comienza con un estado inicial, dependiendo del algoritmo estos bits toman otro valor, los bloques de mensaje M_i en la entrada se les aplica la operación `XOR` con los primeros r bits del estado, mezclándose con la permutación. Cuando todos los bloques de mensaje hayan sido adicionados al estado de la permutación, se pasa a la parte de exprimir, donde se extraen los r bits del estado forman los bloques de salida Z_j. Se puede ajustar la longitud de salida de acuerdo de la aplicación. Los c bits nunca interactúan con los bloques de entrada ni con las salidas. Las entradas se fijan con la función de relleno para tener una longitud fija igual al tamaño de r, en el último bloque de salida se trunca con la longitud deseada de bits. La permutación f produce una distribución pseudoaleatoria de 2^b estados.

Las funciones esponja satisfacen tres propiedades de seguridad en términos de resistencia: a colisiones, a la primera preimagen y a la segunda preimagen. La resistencia a colisiones se define como la dificultad de encontrar dos resultados iguales $F(x) = F(y)$ con distintas entradas $x \neq y$. La resistencia a la primer preimagen hace referencia a encontrar el valor de entrada x, dado $F(y)$, mientras que la resistencia a la segunda preimagen considera que es inviable encontrar una colisión $F(x) = F(x')$ con una entrada especifica $x \neq x'$

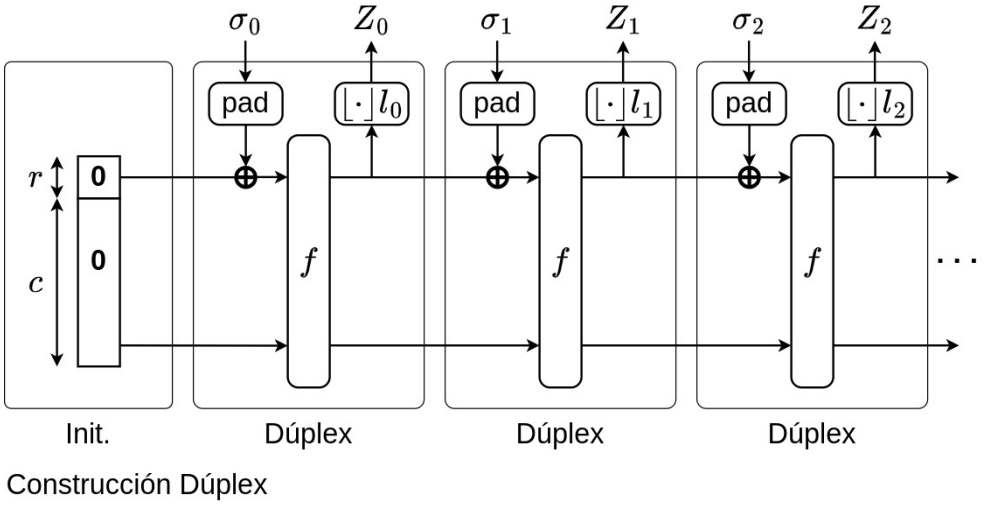


Figura 3.2: Representación de la construcción dúplex.

También existe otro modo de operación, la construcción dúplex mostrada en la Figura 3.2. Con este modo se puede absorber y exprimir a la misma velocidad con solo una llamada a la permutación f. Con ella se pueden implementar generación de secuencias pseudoaleatorias con semillas de entrada, esquemas de cifrado autenticado,

entre otras.

La seguridad de las funciones esponja se basa en la cantidad de bits c que se mantienen secretos en las fases de absorción y exprimir. La probabilidad de que un ataque sea exitoso para un oráculo aleatorio es $\frac{N^2}{2^{c-1}}$, donde N es el número de consultas a f. Todos los ataques a las funciones esponja implican que la permutación puede ser distinguida de una permutación escogida aleatoriamente. Su nivel de seguridad es de $c/2$ bits de resistencia tanto para ataques de colisión como de preimagen. Estos bits de seguridad indican que el atacante necesita $2^{c/2}$ operaciones para romperlo.

3.1. Criptografía Ligera

Con el surgimiento del *IdC*, donde cada dispositivo se comunica a través de distintos ambientes, utilizando distintos protocolos, principalmente porque están especializados en desempeñar una tarea o proveer un servicio en específico. Es fundamental el uso de criptografía para salvaguardar la información que se procesa en estos dispositivos. Sin embargo, los algoritmos criptográficos estándares (AES, SHA-2) resultan demandantes en recursos por las características de los dispositivos restringidos. Por esto, la necesidad de crear y estandarizar algoritmos ligeros que puedan ser implementados eficientemente en dispositivos con bajo consumo energético, limitaciones de tamaño de memoria, poder de procesamiento reducido (procesadores de 8 bits, 16 bits, 32 bits), entre otros. Estos algoritmos se agrupan en la rama llamada criptografía ligera.

La criptografía ligera es capaz de brindar los mismos servicios que los algoritmos estándares, enfocandose en los dispositivos con recursos limitados. Los algoritmos buscan reducir el consumo, aumentar la eficiencia y el desempeño, disminuir la latencia y, en caso de implementaciones en hardware, ocupar un área de chip pequeña.

Dentro de la rama de criptografía ligera se han diseñado y analizado cuatro primitivas criptográficas principales [17], cifradores por bloque, cifradores por flujo de datos, funciones hash y curvas elípticas. En general se busca que tengan tamaño menor de tamaño de llaves y bloques, menor número de rondas al igual que menor complejidad en las rondas, estados internos de menor tamaño y bajar los requerimientos de memoria y recursos. También [17] define tres subcategorías algoritmos ligeros: ultraligeros, bajo costo y ligeros. Estas categorías están definidas por las compuertas utilizadas en el caso de hardware (menor a 1000 hasta 3000) y el software por la cantidad de ROM (4 KB – 32 KB) y RAM (256 B – 8KB).

3.1.1. Historia

Ha habido un trabajo significativo realizado por la comunidad académica relacionada con la criptografía ligera, esto incluye implementaciones eficientes de los algoritmos criptográficos estandarizados, diseño y análisis de nuevas primitivas criptográficas y protocolos [18]. En el 2013 se anunció la competencia CAESAR[2] para diseñar nuevos esquemas de cifrado autenticado con el fin de ofrecer ventajas sobre el algoritmo AES en su modo GCM y que puedan ser aptos para un uso generalizado. Dentro de la competencia existió una categoría que consideró a las aplicaciones ligeras en ambientes con recursos restringidos. Tras una serie de rondas se eligió como primera opción en el caso de aplicaciones ligeras al algoritmo Ascon en 2019. A la par del inicio de la competencia CAESAR en 2013 el NIST[3] comenzó a investigar el desempeño de estándares criptográficos aprobados para los dispositivos restringidos para comprender las necesidades de tener estándares de criptografía ligera.

En 2015 el NIST inició el proceso de estandarización de criptografía ligera para seleccionar uno más esquemas para el Cifrado Autenticado con Datos Asociados con funcionalidad adicional para realizar funciones Hash adecuados para los entornos restringidos. En agosto de 2018 se recibieron 57 propuestas, descartando aquellos que no cumplían con los requisitos, después en abril de 2019 se anunciaron 56 candidatos que pasaron a la primera ronda, donde se evaluó el desempeño y costo, se realizó criptoanálisis, así como propuestas de contramedidas contra ataques por canal lateral [4]. En agosto de 2019, 32 algoritmos pasaron a la segunda ronda, la cual evaluó el desempeño en las implementaciones tanto en hardware como software, así como la resistencia a ataques laterales. La ronda final en 2021 se centró en la seguridad como criterio de selección con el fin de que los algoritmos no liberarán información. Se aumentaron las métricas en el desempeño, implementaciones contra ataques por canal lateral y fallas, además se evaluó con herramientas que usaran propiedad intelectual dentro del algoritmo [19].

Finalmente, la familia de algoritmos Ascon fue seleccionada como estándar para las aplicaciones de criptografía ligera. La familia Ascon incluye AEAD, funciones hash y funciones XOF(extendable output function), satisfaciendo una gran variedad de aplicaciones con un costo menor para extender funcionalidades adicionales, gracias a su diseño de permutación. Ascon presenta distintas variaciones en su tamaño de bloque de entrada. Es un algoritmo maduro por los distintos análisis que se le han realizado. Su desempeño tanto en hardware como el software es bueno. Se requiere

[2]Competition for Authenticated Encryption: Security, Applicability, and Robustness

[3]National Institute of Standards and Technology

[4]Son ataques que obtienen información de la implementación de la primitiva criptográfica a través de su implementación en software o hardware. El atacante observa o mide las características analógicas de la implementación sin alterar su integridad. Esta información puede ser obtenidas mediante variables de ejecución como: tiempos de ejecución, mensajes de error, valores de retorno, saltos condicionales o por variables físicas como: consumo de potencia, radiación electromagnética o ruido acústico [15].

poco costo adicional para las implementaciones protegidas[19].

3.2. Ascon

La familia de algoritmos Ascon es un conjunto de cifradores que ofrecen los servicios de cifrado autenticado con datos asociados (AEAD) y hash. Contiene a los algoritmos Ascon-128, Ascon-128a, Ascon-Hash y Ascon-Xof. Todos los esquemas tienen una seguridad de 128 bits y usan la misma permutación [20]. Estos algoritmos están constituidos por funciones esponja, con la misma permutación se pueden realizar todos los demás algoritmos configurando los datos de entrada y salida.

Ascon fue diseñado por el equipo de criptografía de la Universidad Tecnológica de Graz, Infineon Technologies, Lamarr Security Reseach y la Universidad de Radboud, por Christoph Dobraunig, Maria Eichlseder, Florian Mendel y Martin Schläffer. Fue desarrollado para la competencia CAESAR en 2014 y finalmente en 2023 se convirtió en el algoritmo ganador por el NIST para su eventual estandarización.

3.2.1. Permutación

La permutación de Ascon está conformada por cinco palabras de 64 bits, sus operaciones utilizan las funciones `AND`, `NOT` y `XOR` y rotaciones de las palabras. Es rápida en implementaciones de 32, 16 y 8 bits. El estado S de la permutación está formado por 320 bits divididos en palabras de 64 bits y su valor se actualiza realizando permutaciones p^a o p^b, es decir a o b rondas dependiendo de la etapa del algoritmo. Al igual que las funciones esponja los bits del estado se agrupan en r y c, $c = 320 - r$. El estado S se divide en cinco registros de 64 bits $S = S_r||S_c = x_0||x_1||x_2||x_3||x_4$

Todos los modos de operación de Ascon operan en un estado de 320 bits, el cual se actualiza con el número de rondas dependiendo del modo. Este estado se divide en dos, una parte externa S_r y la parte interna S_c , la cantidad de bits de cada parte está relacionada de la siguiente forma $c = 320 - r$. Cada variante tiene un distinto número de bits en r y parámetros, los cuales se muestran en la Tabla 3.1.

La permutación está compuesta de tres capas: adición de constante de ronda p_C, la capa de sustitución p_S y la capa de difusión lineal p_L. Por lo tanto, la permutación está definida como $p = p_L \circ p_S \circ p_C$. En la capa p_C se le realiza una operación XOR al segundo registro del estado $x_2 \rightarrow x_2 \oplus c_r$, la Tabla 3.2 indica los valores de las constantes dependiendo del número de rondas de la permutación y la ronda en la que se encuentre. En la capa de sustitución p_S el estado se actualiza aplicando 64 cajas S de cinco bits, una forma de realizarlo es mediante el uso de una tabla de consulta que se muestra en la Tabla 3.3 o por operaciones booleanas como se observa

Tabla 3.1: Parámetros de los algoritmos de la famila Ascon.

	Tamaño en Bits						Rondas	
Algoritmo	Llave K	Nonce N	Etiqueta (Tag)	Taza r	Capacidad c	Salida Hash	p^a	p^b
Ascon-128	128	128	128	64	256	-	12	6
Ascon-128a	128	128	128	128	192	-	12	8
Ascon-Hash	-	-	-	64	256	256	12	12
Ascon-Xof	-	-	-	64	256	arbitraria	12	12
Ascon-Hasha	-	-	-	64	256	256	12	8
Ascon-Xofa	-	-	-	64	256	arbitraria	12	8

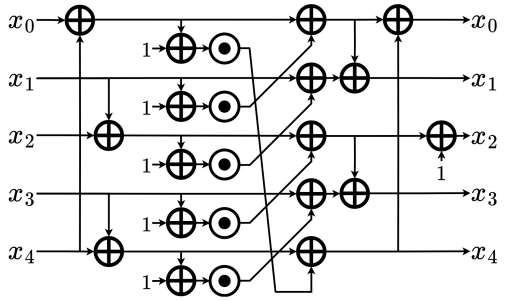
en la Figura 3.3(a). Finalmente se realiza la difusión lineal p_L, para cada registro x se aplican dos rotaciones y se reducen con una operación `XOR`, lo cual se ilustra en la Figura 3.3(b). En conjunto la actualización de estado S por las tres capas se representa gráficamente como la Figura 3.4.

Tabla 3.2: Constantes de ronda c_r usadas en cada ronda de p^a y p^b.

p^{12}	p^8	p^6	c_r	p^{12}	p^8	p^6	c_r
0			00000000000000F0	6	2	0	0000000000000096
1			00000000000000E1	7	3	1	0000000000000087
2			00000000000000D2	8	4	2	0000000000000078
3			00000000000000C3	9	5	3	0000000000000069
4	0		00000000000000B4	10	6	4	000000000000005A
5	1		00000000000000A5	11	7	5	000000000000004B

Tabla 3.3: Caja S de 5 bits.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
$S(x)$	4	B	1F	14	1A	15	9	2	1B	5	8	12	1D	3	6	1C	1E	13	7	E	0	D	11	18	10	C	1	19	16	A	F	17



(a) Caja S de 5 bits $S(x)$.

$$x_0 \leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$$
$$x_1 \leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$$
$$x_2 \leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$$
$$x_3 \leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$$
$$x_4 \leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$$

(b) Capa lineal de 64 bits $\Sigma(x_i)$

Figura 3.3: Capa de sustitución y capa linear de Ascon.

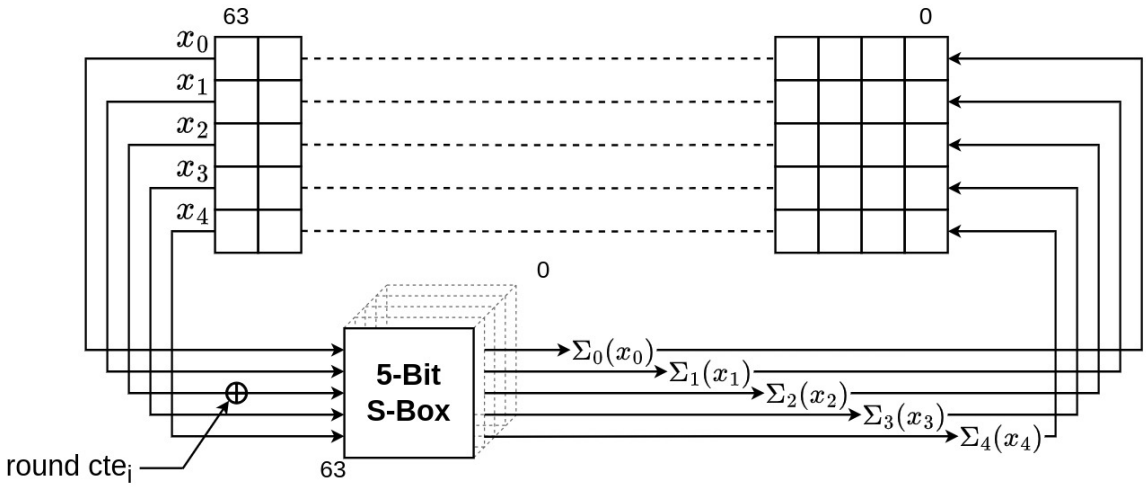


Figura 3.4: Ronda ASCON.

3.2.2. Modo Cifrado Autenticado

Los algoritmos protegen la confidencialidad, realizan la autenticación y verificación de la integridad de los datos que esten cifrados. En la autenticación se genera una etiqueta T que es un tipo de firma [15]. Con la autenticación se protege la integridad del mensaje, indicando cuándo los datos se han corrompido. Al descifrar el mensaje y generar la etiqueta T^*, esta debe coincidir con la etiqueta del cifrado T. Algunos de ellos incluyen el procesamiento de datos asociados, a estos algoritmos se les conoce AEAD. Estos datos asociados son cualquier dato utilizado para autenticar sin cifrarlos. Algunos ejemplos de datos asociados incluyen a las cabeceras de los paquetes en la red, metadatos, estampas de tiempo, identificadores, entre otros.

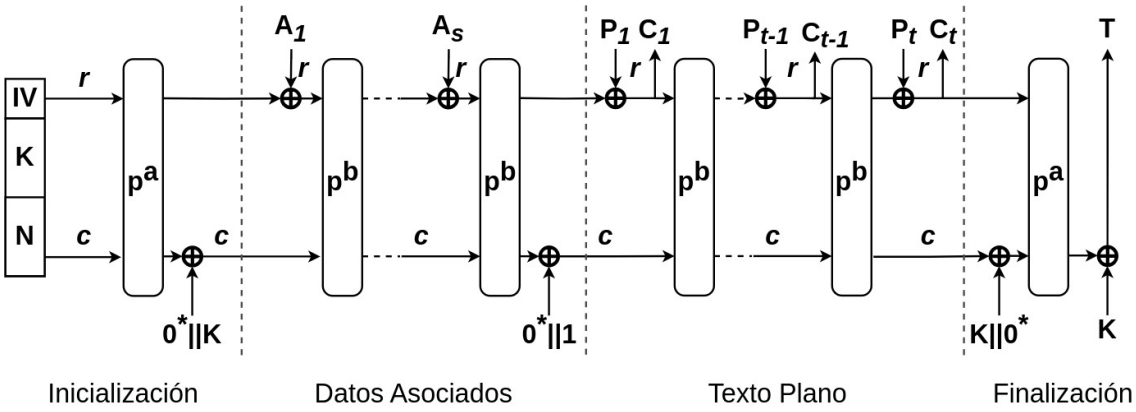


Figura 3.5: Diagrama de ASCON en modo cifrado.

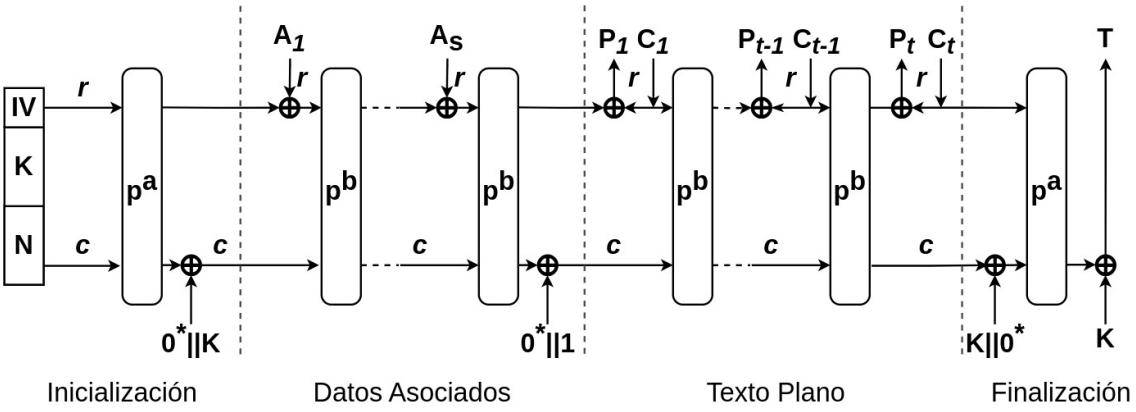


Figura 3.6: Diagrama de Ascon en modo descifrado.

$$IV_{k,r,a,b} \leftarrow k||r||a||b||0^{160-k} \begin{cases} 80400c0600000000 & \text{Para Ascon-128} \\ 80800c0800000000 & \text{Para Ascon-128a} \end{cases}$$

Los miembros de AEAD de Ascon están parametrizados por la longitud de la llave $k \leq 160$ bits, la longitud de la taza r y el número de rondas a y b, la Tabla 3.1 indica los valores sugeridos. Como salida se tiene un cifrado C de la misma longitud que el texto plano P y regresa la etiqueta de autenticación T. En el caso del descifrado se regresa P y una bandera de error $\perp$ si las etiquetas no coinciden. Los pasos detallados del cifrado $E_{k,r,a,b}(K, N, A, P) = (C, T)$ se encuentra en el Algoritmo 1, mientras que el descifrado $D_{k,r,a,b}(K, N, A, C, T) = (P, \perp)$ están el Algoritmo 2. También la Figuras 3.5 y 3.6 esquematizan ambos algoritmos. Esta es una construcción esponja para los datos asociados y dúplex para el cifrado. Está dividida en 4 etapas:

1. Inicialización: Inicializa el estado con la llave K el número público N y un vector de inicialización IV. Con a rondas a la permutación. Al terminar la inicialización se vuelve a agregar la llave en los últimos bits del estado.

2. Absorción de los datos asociados: Actualiza el estado absorbiendo los bloques de datos asociados $A = A_1||...||A_s$ de tamaño r. El número de rondas en esta etapa corresponde a b. Si el último bloque es de tamaño menor a r se rellena con $1||0^{r-|A_s|-1}$. Cuando todos los bloques de los datos asociados sean procesados se le adiciona $(0^{319}||1$ para separar el dominio.

3. Procesamiento del texto plano: Absorbe los bloques del texto plano P y produce el texto cifrado C. También son separados en bloques de tamaño r $P = P_1||...||P_t$ y se utiliza el mismo relleno como en los datos asociados. Cada que un bloque sea procesado se dan p^b rondas. El cifrado se produce al realizar `XOR` de los bloques con S_r. En el caso del descifrado se realiza una `XOR` de S_r con los bloques C_i para restaurar los bloques del texto plano P_i y se sustituye S_r por C_i.

4. Finalización: se adiciona la llave en en los primeros 128 bits de S_c, se dan a últimas rondas y se produce una etiqueta de autenticación T al hacer `XOR` de la llave con los primeros bits de S_c. En el caso del descifrado se agrega un paso extra donde se comprueba $T = T^*$.

Algoritmo 1 Cifrado Autenticado $E_{k,r,a,b}(K, N, A, P)$

Entrada: llave $K \in \{0,1\}^k : k \leq 160$
nonce $N \in \{0,1\}^{128}$
datos asociados $A \in \{0,1\}^*$
texto plano $P \in \{0,1\}^*$
Salida: texto cifrado $C \in \{0,1\}^{|P|}$
etiqueta $T \in \{0,1\}^{128}$

Inicialización
$S \leftarrow IV_{k,r,a,b}||K||N$
$S \leftarrow p^a(S) \oplus (0^{320-k}||K)$
Procesamiento de Datos A.
if $|A| > 0$ **then**
$A_1...A_s \leftarrow$ bloques r_bit $A||1||0^*$
for $i \leftarrow 1$ `to` s **do**
$S \leftarrow p^b((S_r \oplus A_i)||S_c)$
$S \leftarrow S \oplus (0^{319}||1)$
Procesamiento de Texto P.
$P_1...P_t \leftarrow$ bloques r_bit $P||1||0^*$
for $i \leftarrow 1$ `to` t **do**
$S_r \leftarrow S_r \oplus P_i$
$C_i \leftarrow S_r$
$S \leftarrow p^b(S)$
$S_r \leftarrow S_r \oplus P_t$
$C_t \leftarrow \lfloor S_r \rfloor_{|P| \bmod r}$
Finalización
$S \leftarrow p^a(S \oplus (0^r||K||0^{320-r-k}))$
$T \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$
return $C_1||...||C_{t-1}||C_t, T$

Algoritmo 2 Decifrado Autenticado $D_{k,r,a,b}(K, N, A, P, T)$

Entrada: llave $K \in \{0,1\}^k : k \leq 160$
nonce $N \in \{0,1\}^{128}$
datos asociados $A \in \{0,1\}^*$
texto cifrado $C \in \{0,1\}^*$
etiqueta $T \in \{0,1\}^{128}$
Salida: texto plano $P \in \{0,1\}^{|C|}$ o $\perp$

Inicialización
$S \leftarrow IV_{k,r,a,b}||K||N$
$S \leftarrow p^a(S) \oplus (0^{320-k}||K)$
Procesamiento de Datos A.
if $|A| > 0$ **then**
$A_1...A_s \leftarrow$ bloques r_bit $A||1||0^*$
for $i \leftarrow 1$ `to` s **do**
$S \leftarrow p^b((S_r \oplus A_i)||S_c)$
$S \leftarrow S \oplus (0^{319}||1)$
Procesamiento de Texto P.
$C_1...C_t \leftarrow$ bloques r_bit C
$0 \leq |C_t| < r$
for $i \leftarrow 1$ `to` t **do**
$P_i \leftarrow S_r \oplus C_i$
$S \leftarrow C_i||S_c$
$S \leftarrow p^b(S)$
$P_t \leftarrow \lfloor S_r \rfloor_{|C_t|} \oplus C_t$
$S_r \leftarrow S_r \oplus (P_t||1||0^*)$
Finalización
$S \leftarrow p^a(S \oplus (0^r||K||0^{320-r-k}))$
$T^* \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$
if $T = T^*$ **then**
return $P_1||...||P_{t-1}||P_t$
else
return $\perp$

3.2.3. Hash

El modo de operación Hash es una construcción esponja tradicional. Existen dos modos principales: Hash, donde la longitud de la salida es fija con $l = 256$ normalmente; y Xof, cuya la salida es variable. Ambos modos tienen la misma construcción esponja mostrada en la Figura 3.7 y su procedimiento es similar al presentado en el

Algoritmo 3) solo se van a diferenciar en sus vectores de inicialización y la longitud de salida.

Algoritmo 3 Hash y función de salida extendida $X_{h,r,a}(M, l)$

Entrada: Mensaje $M \in \{0,1\}^*$ tamaño de salida $l \leq h$ o tamaño arbitrario si $h = 0$
Salida: Hash $H \in 0, 1^l$

Inicialización
$S \leftarrow p^a(IV_{h,r,a}||0^c)$
Absorción del Mensaje
$M_1...M_s \leftarrow$ bloques r_bit $M||1||0^*$
for $i \leftarrow 1$ `to` s **do**
$S \leftarrow p^a((S_r \oplus M_i)||S_c)$
Obtención del Hash
for $i \leftarrow 1$ `to` $t = \lceil \frac{l}{r} \rceil$ **do**
$H_i \leftarrow S_r$
$S \leftarrow p^a(S)$
return $\lfloor H_1||...||H_t \rfloor_l$

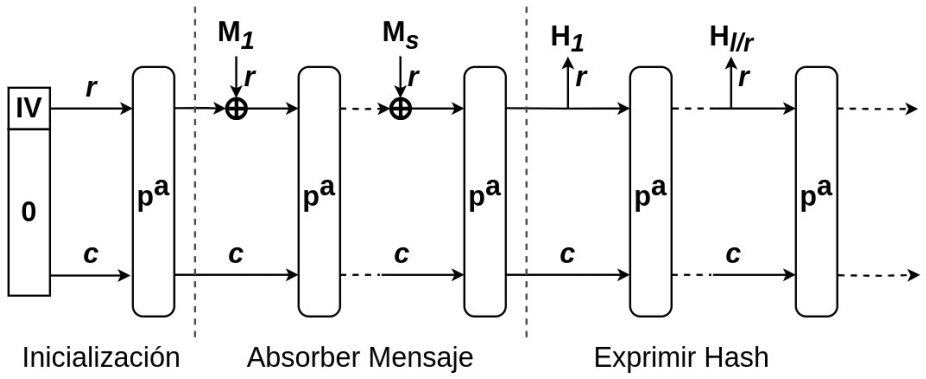


Figura 3.7: Diagrama de Ascon modo Hash.

$$IV_{h,r,a} \leftarrow 0^8||r||a||0^8||h \begin{cases} 00400c0000000000 & \text{Para Ascon-Xof} \\ 00400c0000000100 & \text{Para Ascon-Hash} \end{cases}$$

El algoritmo de Hash está dividido en tres etapas:

1. Inicialización: Los registros de S tienen un valor inicial de $IV||0^{256}$. Posteriormente se realizan a rondas de la permutación. Al ser un valor constante, se puede omitir este paso iniciando desde el valor que toman los registros al dar a rondas.

2. Absorción del mensaje: Se descompone el mensaje en bloques de r bits generando $M = M_1||...||M_s$. Si el último bloque es incompleto, M_s se rellena con

$1||0^{r-|M_s|-1}$. A diferencia de los algoritmos anteriores en la parte de absorción y exprimir el Hash el número de rondas es a.

3. Hash exprimido: Simplemente corresponde a S_r cada que se extrae un nuevo bloque de $H = H_1||\ldots||H_t$ se dan a rondas a la permutación. Si l no es divisible entre r el bloque final H_t se trunca.

3.3. Generador de números pseudoaleatorios

En [21] proponen el uso de funciones hash como generadores de números pseudoaleatorios, más específicamente en aquellas basados en construcciones esponja y dúplex. También se menciona que es viable absorber distintas semillas en distintos puntos, teniendo múltiples fases de absorción y extracción de números pseudoaleatorios. A esta construcción se le denomina resemillable. Particularmente, una construcción resemillable es conveniente, ya que se puede adicionar semillas frescas a los bits pseudoaleatorios ya generados en lugar de descartar los bits pseudoaleatorios al agregar una nueva semilla y reinicializar el estado. Este tipo de generador de números pseudoaleatorios se probó con la permutación del algoritmo keccak [21]. Debido a esto también es posible adaptarlo a la permutación Ascon. En esta tesis se propone utilizarlo como una construcción esponja resemillable.

Para usar la permutación de Ascon como un PRNG se optó por usar como referencia al modo Ascon-Xof con unas ligeras modificaciones utilizando la construcción esponja, lo que se puede observar en la Figura 3.8. Se inicializa S con ceros dando a rondas a la permutación. En todas las etapas del algoritmo se utilizan $a = 12$ rondas. Se considera que las semillas Φ_i siempre coinciden con el tamaño de S_r para ser absorbidas, por lo tanto, no se necesita rellenar las semillas. Se considera que se pueden hacer llamadas al generador en cualquier momento, por lo tanto, el estado con los bits pseudoaleatorios está disponible. El Algoritmo 4 muestra el comportamiento cuando se realiza una llamada, ya sea para generar un nuevo número pseudoaleatorio o para absorber una nueva semilla. Al igual que Ascon-Xof tiene tres fases: Inicialización, Absorción y Exprimir Hash, con la particularidad que puede tener múltiples etapas de Absorción y Exprimir Hash.

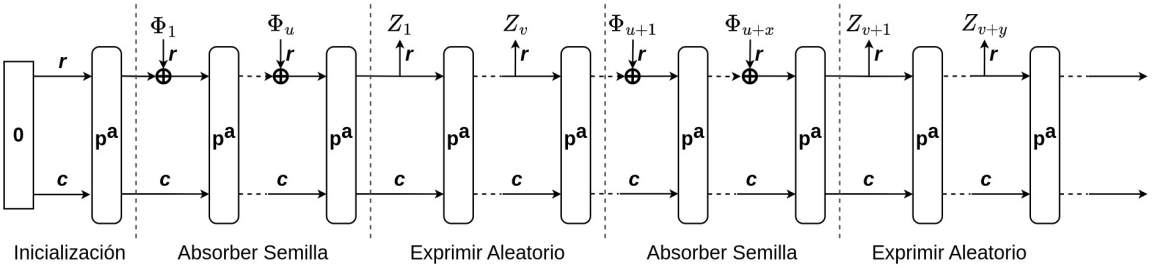


Figura 3.8: Diagrama de la propuesta modo resemillado para la permutación de Ascon

Algoritmo 4 Generación de Números Pseudoaleatorios $R_a(\Phi, rand)$

Entrada: semilla $\Phi \in \{0,1\}^r$, semilla generada por una fuente de entropía
selector `rand` $\in \{0,1\}$ selecciona si se requiere un número pseudoaleatorio o absorber una nueva semilla
Salida: pseudoaleatorio $Z \in 0,1^r$ si $rand = 1$

```
PSEUDOALEATORIO(Φ,rand)
  if primer llamada then
    Inicialización( )
    Semilla(Φ)
    if rand then
      Z ← Aleatorio( )
  if rand then
    Z ← Aleatorio( )
    return Z
  else
    Semilla(Φ)
INICIALIZACIÓN
  S ← p^a(0^320)
SEMILLA(Φ)
  S ← p^a((S_r ⊕ Φ)||S_c)
ALEATORIO
  Z ← S_r
  S ← p^a(S)
  return Z
```

Posterior a la ejecución de las tres primeras fases se inicia a resemillar y extraer nuevas secuencias pseudoaleatorios. En las fases de resemillado se absorben semillas frescas $\Phi_{u+1}||...||\Phi_{u+x}$ con las cuales se perturba el estado interno. En consecuencia, se genera una secuencia de bits distinta. Cuando se hayan absorbido las semillas los nuevos números pseudoaleatorios generados $Z_{v+1}||...||Z_{v+y}$ no siguen la secuencia de los anteriores, aunque se conozca el estado de los anteriores números pseudoaleatorios si la siguiente ronda de semillas se mantiene secreta, la predicción de la secuencia de número aleatorios se mantiene protegida. Este ciclo entre absorción de semillas y extracción de número pseudoaleatorios se repite indefinidamente.

3.4. Cifrador por flujo de datos Trivium

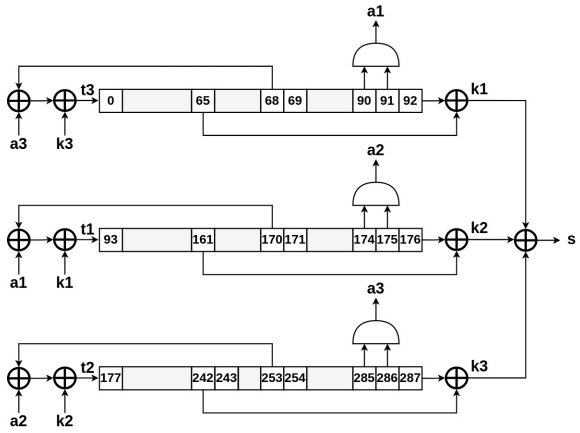


Figura 3.9: Diagrama del cifrador por flujo Trivium.

Trivium es un cifrador por flujo de datos, a diferencia de los cifradores por bloque el texto plano no se mezcla con la llave. Este tipo de cifradores generan secuencias de bits pseudoaleatorias a partir de inicializar su estado S con una llave K. Para obtener el texto cifrado al texto plano se le realiza `XOR` con el flujo de bits pseudoaleatorios que se vaya teniendo. Estos cifradores son considerados como generadores de bits aleatorios deterministas y no un PRNG ya que utilizan una llave y con la misma llave se regeneran los bits usados para cifrar [15].

Trivium se diseñó para la competencia eSTREAM [5] de 2004 a 2008, la cual tuvo como objetivo promover el diseño eficiente y compacto de cifradores por flujo de datos. Trivium quedó en los ganadores en la categoria de cifradores orientados a hardware. Este cifrador utiliza una caja de sustitución de uno a un bit, un estado S de 288 bits, donde el bit más significativo y el menos significativo son contiguos, una llave K de tamaño de 80 bits y un vector de inicialización IV de 80 bits. Contiene tres subgeneradores los cuales toman 15 bits específicos de distintos lugares del estado. Al reducir estos tres subgeneradores con `XOR` se obtiene el bit pseudoaleatorio [22]. El Algoritmo 5 muestra las operaciones y bits utilizados para generar un bit, la Figura 3.9 esquematiza el cifrador. Cuando un bit es generado el estado se actualiza realizando un desplazamiento e introduciendo los bits de los generadores. Este cifrador es capaz de generar 2^{64} bits de flujo. Trivium puede aumentar su desempeño instanciado la misma cantidad de operaciones como bits de salida, siempre y cuando 1152 sea divisible entre el tamaño de bits de salida. Por ejemplo, Trivium tiene sus versiones de 1, 8, 16, 32, hasta 64 bits.

[5] ECRYPT Stream Cipher Project

Algoritmo 5 Cifrador por Flujo de Datos TRIVIUM
$Tri(K, IV, n)$

Entrada: llave $K \in \{0,1\}^{80}$
vector de inicialización $IV \in \{0,1\}^{80}$
número de bits $n \in \mathbb{Z}_0^+$
Salida: secuencia de bits $Z \in \{0,1\}^n$

INICIALIZACIÓN
$(s_1, s_2, ..., s_{93}) \leftarrow K||0^{13}$
$(s_{94}, s_{95}, ..., s_{177}) \leftarrow IV||0^4$
$(s_{178}, s_{179}, ..., s_{288}) \leftarrow 0^{108}||1^3$
for $i \leftarrow 1$ `to` $4 * 288$ **do**
$t_1 \leftarrow s_{66} \oplus s_{91} \cdot s_{92} \oplus s_{93} \oplus s_{171}$
$t_2 \leftarrow s_{162} \oplus s_{175} \cdot s_{176} \oplus s_{177} \oplus s_{264}$
$t_3 \leftarrow s_{243} \oplus s_{286} \cdot s_{287} \oplus s_{288} \oplus s_{69})$
$(s_1, s_2, ..., s_{93}) \leftarrow (t_3, s1, ..., s_{92})$
$(s_{94}, s_{95}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176}$
$(s_{178}, s_{179}, ..., s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287})$
GENERACIÓN DEL FLUJO DE DATOS
$Z \leftarrow \varnothing$
for $i \leftarrow 1$ `to` n **do**
$t_1 \leftarrow s_{66} \oplus s_{93}$
$t_2 \leftarrow s_{162} \oplus s_{177}$
$t_3 \leftarrow s_{243} \oplus s_{288}$
$z \leftarrow t_1 \oplus t_2 \oplus t_3$
$t_1 \leftarrow t_1 \oplus s_{91} \cdot s_{92} \oplus s_{171}$
$t_2 \leftarrow t_2 \oplus s_{175} \cdot s_{176} \oplus s_{264}$
$t_3 \leftarrow t_3 \oplus s_{286} \cdot s_{287} \oplus s_{69})$
$(s_1, s_2, ..., s_{93}) \leftarrow (t_3, s1, ..., s_{92})$
$(s_{94}, s_{95}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176}$
$(s_{178}, s_{179}, ..., s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287})$
$Z \leftarrow Z||z$
return Z

Capítulo 4

Diseño del Criptoprocesador

Los FPGAs (Field Programmable Gate Array) juegan un papel importante en el diseño digital. Estos están compuestos por arreglos de lógica programable que puede ser rápidamente reconfigurable para proporcionar otra funcionalidad. El componente principal de un FPGA de Xilinx[1] se le conoce como bloques lógicos configurables (CLB, por sus siglas en inglés), este contiene una colección de tablas de consulta (LUT) que implementan las funciones booleanas y registros que son utilizados en la parte secuencial. Los actuales FPGA, también contienen procesadores digitales de señales (DSP) para operaciones costosas como sumadores y multiplicadores, bloques de RAM, manejadores de la frecuencia de reloj, memoria de configuraciones y bloques de entrada y salida. Estos componentes están conectados por la matriz de interconexión, esta matriz genera una red compleja de un sistema de ruteo definida por la memoria de configuración para que la lógica programable realice las operaciones adecuadamente. Muchas veces es conveniente utilizar los SoC que contenga un FPGA lógica programable (PL) con un sistema de procesamiento (PS) que se encargue de la transferencia de datos y el control de la lógica programable, para aumentar la eficiencia y desempeñar distintas aplicaciones.

Por medio de los FPGA es posible implementar un diseño de hardware, analizar su comportamiento, probar su correctitud, todo a un bajo costo comparado con la síntesis en silicio. Con ellos se reduce el tiempo de desarrollo ya que agilizan de manera iterativa el desarrollo. Los aceleradores hardware tienen como objetivo reducir los ciclos de reloj que toma ejecutar un procedimiento. Algunas de las ventajas de transformar parte de las operaciones en software a un hardware dedicado incluyen, el aumento del procesamiento de un mayor número de datos, paralelizar operaciones y realizar operaciones costosas a nivel software de una forma más eficiente.

[1]Es una empresa fundada por los creadores de los FPGAs Ross Freeman y Bernard Von Der Schmitt en Silicon Valley. Ofrecen distintas gamas de productos Virtex(alto rendimiento), Kintex(gama media) y Artix(bajo costo). Además, disponen del software necesario para programar los dispositivos.

4.1. Propuesta

El diseño propuesto toma en consideración la seguridad de los dispositivos restringidos reutilizando la mayor parte del hardware para ofrecer una gran parte de los servicios que proporciona la criptografía simétrica. Es por esto que las funciones esponja son ventajosas ya que la sincronización de las entradas y salidas utilizando la misma permutación posibilita la creación de construcciones que pueden ofrecer distintos servicios. En el contexto de los dispositivos restringidos la permutación de Ascon y sus modos de operación resultan adecuados, al ser Ascon una función esponja, también puede ser utilizada para otros algoritmos, con el fin de crear un criptoprocesador ligero. Con el siguiente diseño se implementa los algoritmos de cifrado autenticado con datos asociados, la función Hash, la generación de números pseudoaleatorios, generación de llaves y enmascaramiento de llaves. Todos hacen uso de la permutación de Ascon junto a sus rondas para proveer los servicios de confidencialidad, autenticación, integridad y aleatoriedad. Este criptoprocesador es nombrado como **LWCP-V** (**L**ight**W**eight **C**rypto**P**rocessor-**V**).

La Figura 4.1 esquematiza la interconexión de los módulos para implementar los algoritmos definidos anteriormente. Se tiene como módulo principal a la permutación de Ascon (Ascon-p), simplemente para realizar adecuadamente los distintos algoritmos se tiene que controlar las entradas y salidas a la permutación. Dichas entradas y salidas están definida por el tipo de algoritmo, la etapa del algoritmo y la cantidad de datos que se van a procesar. Contiene dos Controles, el principal (AsconCtrl) este controla a los algoritmos AEAD, Hash y el enmascaramiento de las llaves, mientras que (RandCtrl) controla la generación de números pseudoaleatorios y la creación de nuevas llaves simétricas que son compatibles para el algoritmo AEAD. Se utiliza por el momento al algoritmo Trivium simulando ser la fuente de entropía para probar el concepto y funcionalidad de la construcción resemillable.

Como parámetro general se utiliza un valor fijo del tamaño de r de 64 bits, entonces todos los algoritmos usan su versión de 64 bits de tamaño de bloque. Las entradas son bloques de 64 bits P_i de texto plano, A_i de datos asociados, C_i texto cifrado y M_i para los bloques a realizar hash. De la misma manera las salidas son de 64 bits para los bloques C_i texto cifrado, P_i texto descifrado, Z_i número pseudoaleatorios y H_i para los bloques de hash. Los datos que tienen como 128 bits son la llave K el número público N y la etiqueta de autenticación T en la salida. Para controlar las etapas de los algoritmos y la comunicación con la lectura y escritura de los datos se cuenta con las siguientes señales y datos.

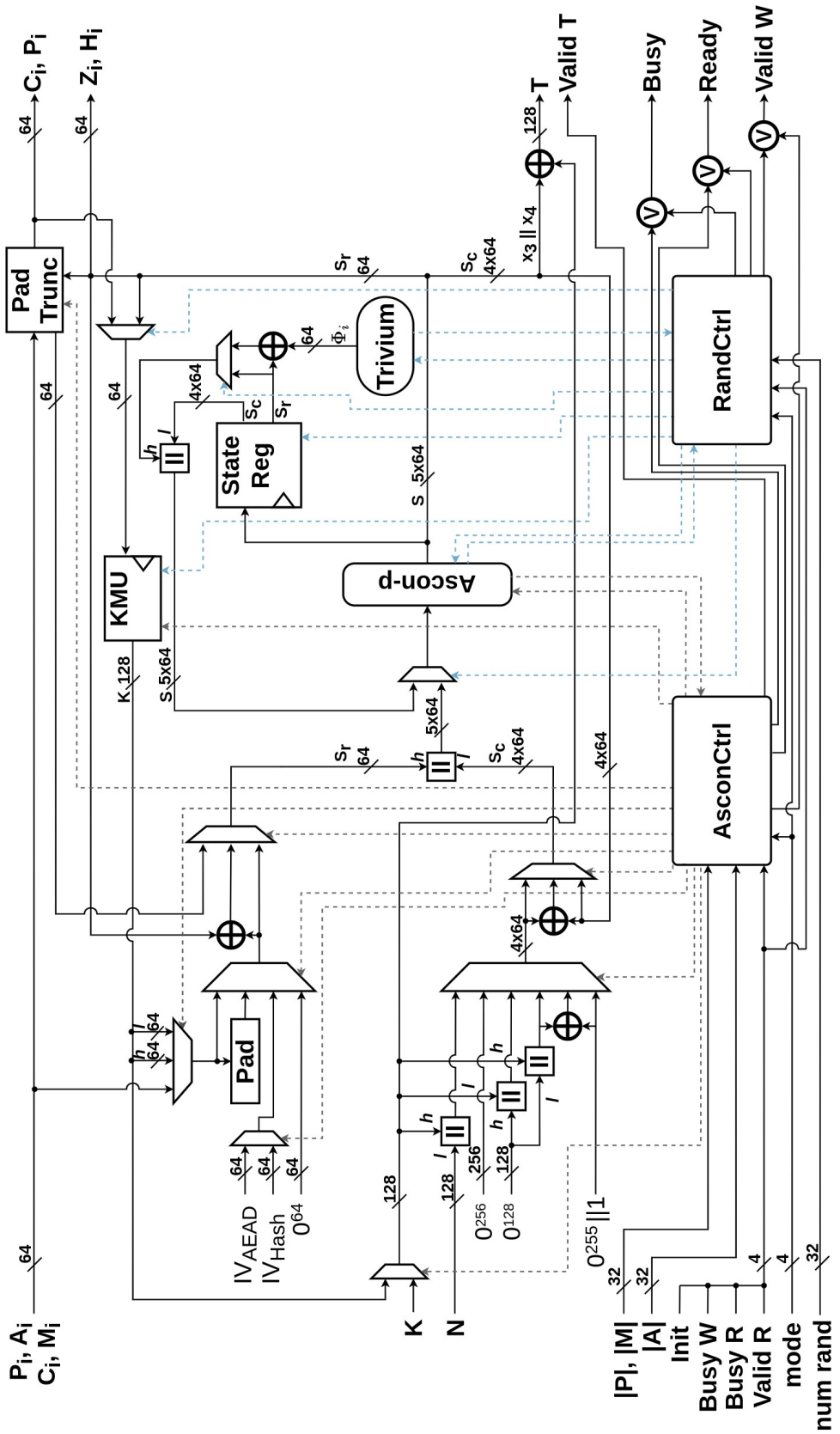


Figura 4.1: Diagrama general del Criptoprocesador ligero LWCP-V.

- Entradas
  - |P| y |M| (32 bits): Cantidad de bytes que contiene el texto plano y el mensaje. Al estar divididos en bytes hace más eficiente el truncamiento y el relleno de los bloques de texto plano y mensaje.
  - |A| (32 bits): Tamaño en bytes de los datos asociados.
  - Init : Inicializa el algoritmo cuando se establece en '1'.
  - Busy W, Busy R: Indican que la escritura o lectura de un bloque está en proceso cuando es '1'.
  - Valid R: Indica que el bloque de entrada A_I, P_i, C_i o M_i son validos para procesar.
  - mode (4 bits): Indica que algoritmo se va a realizar.
  - num rand (32 bits): Indica la cantidad de número aleatorios que se van a producir.
- Salidas
  - Valid T: Indica que la etiqueta de autenticación ha sido realizada.
  - Busy : El algoritmo se encuentra procesando.
  - Ready: Indica que está listo para realizar el proceso.
  - Valid W, señala que el bloque de salida es válido para ser escrito en memoria.

La unidad de procesamiento se utilizó fue el softcore (implementado en lógica programable) Rocket con el conjunto de instrucciones `RV32IMAC`. Existe el repositorio Freedom[2] que es un entorno que contiene los archivos RTL (Register Transfer Level) para las plataformas E300 y U500 de SiFive[3]. Estas plataformas usan como unidad de procesamiento los procesadores tipo Rocket, E300 para los procesadores de 32 bits y U500 para procesadores de 64 bits. El repositorio Freedom tiene como submódulo a Rocket-Chip[4] para general núcleos especializados para distintas aplicaciones y contienen descripciones de hardware propias de SiFive, sin embargo, lo más importantes es que contienen los archivos que dan soporte para que los diseños de hardware puedan ser implementado en las FPGAs de desarrollo ArtyA7 y VC707.

Al igual que se proporciona un entorno para el desarrollo de hardware por parte de SiFive, también dan el soporte para el desarrollo de software para dichas plataformas con fredoom-e-sdk [5]. Este repositorio cuenta con las herramientas necesarias para

[2]https://github.com/sifive/freedom

[3]SiFive es una empresa de semiconductores sin fábrica y proveedor de chips de silicio e IP de procesador RISC-V comerciales basados en la arquitectura del conjunto de instrucciones RISC-V. https://www.sifive.com/

[4]https://github.com/chipsalliance/rocket-chip

[5]https://github.com/sifive/freedom-e-sdk

desarrollar software para los procesadores RISC-V diseñados por SiFive, emuladores, implementaciones de FPGA y placas de desarrollo. Todos los desarrollos de software se realizaron a nivel *bare-metal*, es decir, cuando el software se ejecuta directamente sobre el hardware físico sin ninguna capa de software adicional. En pocos términos se tienen control total del hardware.

Todos los diseños para implementar el criptoprocesador fueron realizado en Chisel3 para ser consistentes con el repositorio Rocket-Chip. Con este lenguaje se pueden implementar los mismos diseños descritos en lenguajes tradicionales como Verilog o VHDL siempre y cuando sean síncronos. Chisel3 ofrece la misma granularidad, es decir el control de los valores de las señales bit a bit. También agiliza la simulación y escritura de las pruebas ya que el comportamiento del circuito puede ser comprobado con funciones a nivel software en Scala. Al ser un lenguaje orientado a objetos, todos los módulos están descritos como un objeto.

4.2. Implementación de la permutación de Ascon

Ya que la permutación de Ascon usa trasformaciones sencillas se puede implementar una ronda en un solo ciclo a altas frecuencias. La Figura 4.2 muestra la integración de las tres capas de transformación de la permutación en una sola ronda. Comienza con los valores de los registros de 64 bits x_0, x_1, x_2, x_3, x_4 que conforman al estado S o una nueva entrada de datos, en el caso de la inicialización de los algoritmos. Posteriormente pasan por la capa de la adición de la constante de ronda, las distintas rondas son implementadas con un multiplexor doce a uno. Posteriormente los datos pasar por la capa de substitución, esta se realiza a nivel columnas, por lo tanto, se pueden sustituir de golpe todos los valores de S con el uso de compuertas `AND, NOT` y `XOR`. Finalmente se realiza la difusión lineal aplicada por filas (64 bits), esta capa consiste en la operación `XOR` de dos rotaciones a la derecha con el valor original. Cuando se haya propagado el estado a través de las tres capas, los resultados actualizan el estado completando una ronda.

Una vez construida la permutación se tienen que definir los mecanismos que controlen las entradas, salidas y el almacenamiento del estado. La Figura 4.3 ilustra la construcción interna del módulo Ascon-p del Diagrama general. Cuenta con los cinco registros de 64 bits (State Reg), una unidad de control de la permutación (CtrlPerm), un multiplexor que controla la entrada a la permutación y la parte central de todo el diseño, la permutación. Con estos módulos se controlan las rondas que se deban dar, los datos de entrada y salida, así como el reinicio junto a las señales de control. Ascon-p es el núcleo de los algoritmos que se van a implementar, para lograrlo, los módulos de alrededor solo se dedican a controlar las entradas, salidas, inicio y rondas que realice Ascon-p.

El control interno del módulo Ascon-p se lleva a cabo por una máquina de esta-

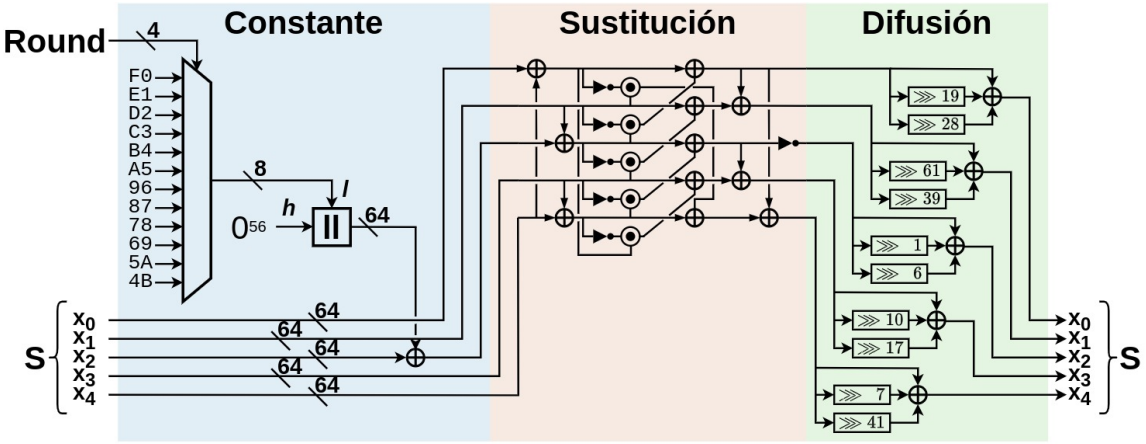


Figura 4.2: Implementación en hardware de la permutación de Ascon.

dos sencilla, esquematizada en la Figura 4.4) por tener tres estados (PermCtrl). Su principal función de esta máquina es controlar el flujo de datos hacia la permutación del estado de entrada o el estado guardado en los registros, al igual que el número de rondas. El estado `rst` simplemente se encarga de inicializar en cero lo registros internos y cambiar al estado `wait`. En el estado `wait` se espera a que la señal `start` esté en alto para iniciar con el conteo de rondas y el flujo de datos atreves de la permutación. En el mismo ciclo de reloj se indica el tipo de permutación con `type`, indica el parámetro a o b que son las rondas. Cuando `typ` es $\mathbf{00_2}$ establece doce rondas, $\mathbf{01_2}$ para ocho y $\mathbf{10_2}$ para seis. Se tiene un contador interno que cuenta las rondas que se han realizado, como se puede observar en la Tabla del capítulo anterior. En la primera ronda siempre se deja pasar el estado de entrada ya que cuenta con los datos de inicialización, los datos absorbidos en el S_r, cambio de dominio o la parte final del cifrado donde se absorbe la llave, posteriormente se recirculan los datos de los registros a la permutación hasta alcanzar el número de rondas deseado. Mientras se mantiene completando las rondas el control manda la señal de `busy` en alto.

Dependiendo de la ronda se adiciona una constante diferente. Todas las rondas deben terminar en el valor **0x4B**. Por lo tanto, cuando se dan ocho rondas el contador se inicializa en cuatro, con seis en seis y cuando son las doce rondas se inicializa en cero. En cada ronda se manda como salida el valor de contador, esto indica al multiplexor de la permutación que constante de ronda se va a adicionar. Cuando se hayan completado las rondas se manda la señal de `valid` para indicar que la salida contiene datos válidos, `ready` en alto para indicar que puede iniciar una nueva serie de rondas y la señal de `busy` se baja. También se cambia al estado `wait` a esperar una nueva señal de `start`.

Como se mencionó anteriormente que la sencillez de las operaciones de la permutación de Ascon hace que la propagación de señal sea consistente con la frecuencia a la cual se esté trabajando, incluso se pueden realizar configuraciones de rondas des-

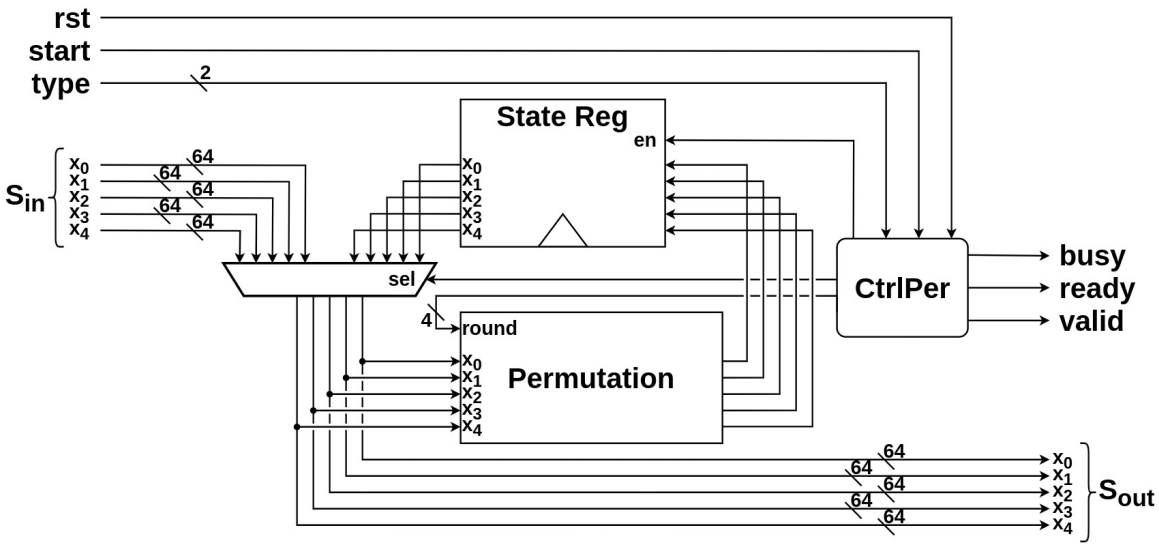


Figura 4.3: Implementación en hardware de la permutación de Ascon junto a su unidad de control.

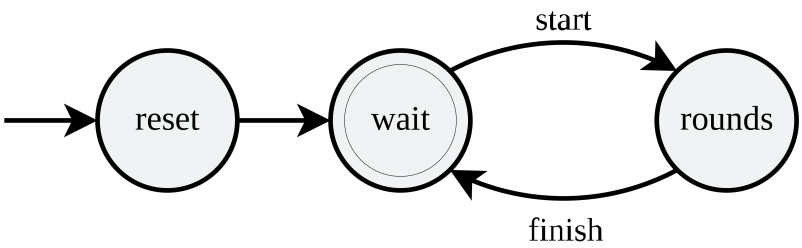


Figura 4.4: Máquina de estados para el control de la permutación de Ascon.

dobladas que produzcan valores validos en un ciclo. En particular se trabajaron con otras dos configuraciones, dos rondas desdobladas mostradas en la Figura 4.5 y tres rondas desdobladas que se muestran en la Figura 4.6, adicionales a una ronda por ciclo. El desdoblamiento de rondas se refiere a una técnica para aumentar el desempeño realizando un número de mayor de rondas en un ciclo. Las permutaciones se conectan en cascada, la salida de una es la entrada para otra. Por ejemplo, al propagarse los datos a través de dos instancias, se habrán realizado dos rondas en un ciclo, solo hay que utilizar la constante de ronda adecuada en cada instancia.

El control de las permutaciones con rondas desdobladas se lleva a cabo por la misma unidad de control descrita anteriormente, lo único que cambia es el aumento del contador de permutación. En lugar de incrementar su valor en uno en cada ciclo, ahora se incrementa dos unidades y tres unidades. Ahora se reinicia el contador cuando se alcanza el valor de diez en dos rondas desdobladas y nueve en el caso de tres rondas, en lugar de once. Se eligieron estas dos versiones adicionales ya que concuerdan con el número de rondas de los algoritmos que se implementaron. Estos tienen en común seis y doce rondas, por ello estos dos valores de rondas son divisibles entre

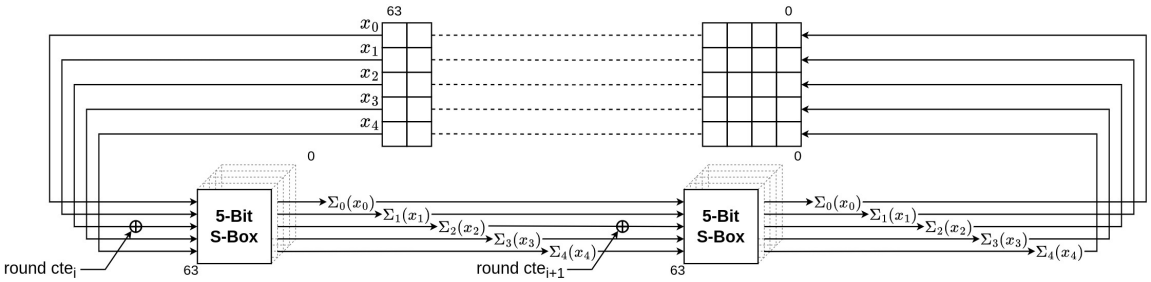


Figura 4.5: Permutación de Ascon con dos rondas desdobladas.

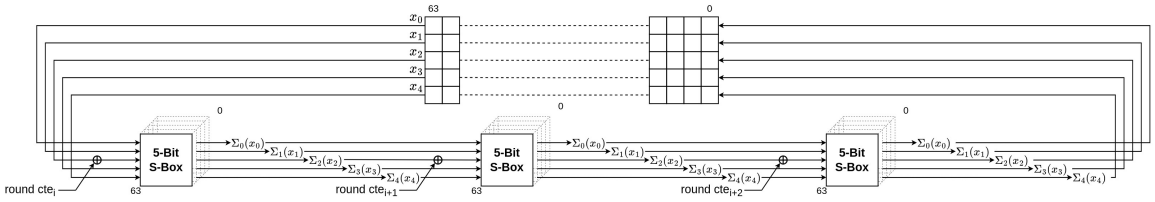


Figura 4.6: Permutación de Ascon con tres rondas desdobladas.

las dos versiones de desdoblamientos.

4.3. Implementación de los modos de operación de Ascon

Se implementaron los modos de operación Ascon-128 (cifrado y descifrado) y Ascon-Hash descritos en la presentación de propuesta para la competencia del NIST. En el capítulo anterior en la Tabla 3.1 se menciona los parámetros que estos algoritmos utilizan, de los parámetros más importantes de estos dos algoritmos son el mismo tamaño de S_r y utilizan un número de rondas de seis p^6 y doce p^{12}. Como puede observarse en el diagrama general del criptoprocesador estos algoritmos tienen módulos en común de hardware. A continuación, se detalla la implementación de ambos algoritmos, así como su hardware que utilizan para llevar a cabo el procesamiento de datos.

4.3.1. Cifrado y descifrado autenticado con datos asociados

El cifrado y descifrado son los algoritmos que requieren un mayor número de elementos de control de flujo de datos para poder operar. Como se puede contrastar en sus algoritmos del capítulo anterior, estos siguen los mismos pasos en la mayor parte. Todos los pasos son los mismos desde la inicialización, absorción de los datos asociados hasta la finalización, solo difieren en la absorción de los bloques de texto plano con los de texto cifrado y en la parte final donde se comparan las dos etiquetas

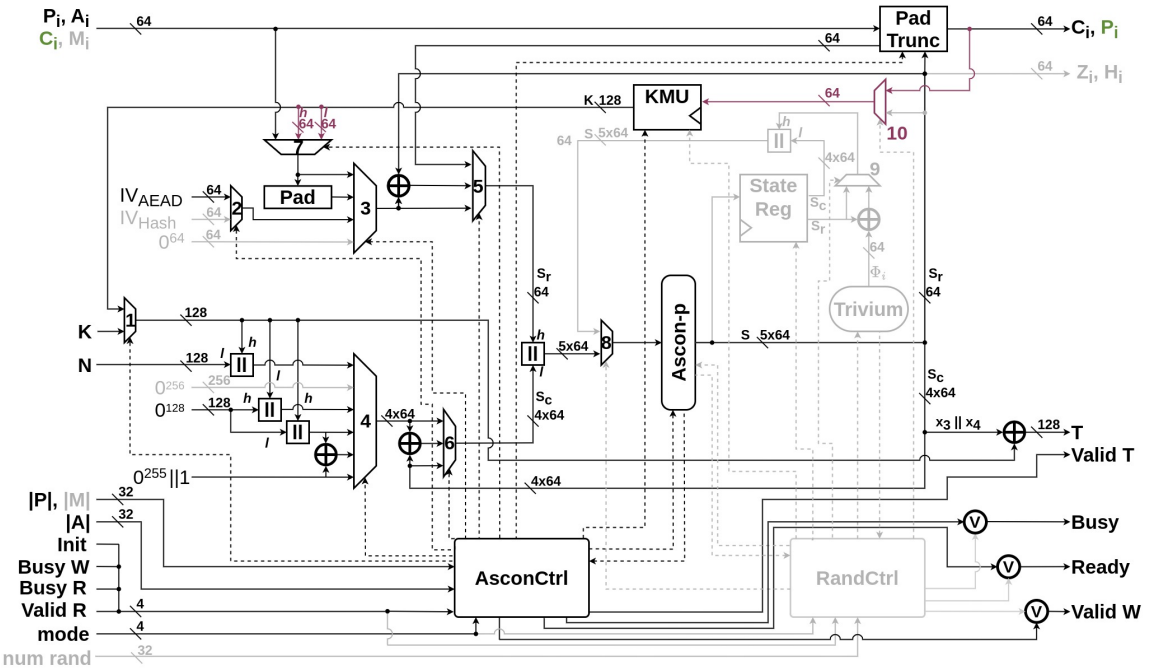


Figura 4.7: Diagrama con el flujo de datos para el modo cifrado y descifrado autenticado con datos asociados.

de autenticación. La Figura 4.7 resalta las direcciones necesarias y los módulos que realizan estos dos algoritmos.

Flujo de datos

Inicialización: Comienza con los valores iniciales de S_r y S_c cuando se haya detectado la señal `init` en alto y `mode` coincida con el modo cifrado o descifrado. Para S_r se utiliza el vector de inicialización IV_{AEAD}, por lo tanto, los multiplexores Mux2, Mux3, Mux5 y Mux7 deben dejar pasar ese valor. En cuanto a S_c este debe contener los valores de la K y N. En este caso K puede venir de los registros de la Unidad Administradora de llaves (KMU) o cargada desde afuera la selección de estas llaves está determinada por el Mux1. Posteriormente K se concatena con N para formar el valor inicial de S_c pasan por Mux4 y Mux6 sin ninguna modificación adicional. Finalmente S_r se concatena con S_c para iniciar las rondas de iniciación en Ascon-p.

Absorción de datos asociados: Cuando se absorba el primer bloque de la llave a S_c se le debe de adicionar la llave en Mux6, por ello el Mux4 contiene la concatenación de 0^{256} con K. Los bloques posteriores no requieren modificación alguna de S_c, hasta el bloque final. En S_r cada que se adiciona un nuevo bloque A_i este pasa por Mux3 sin modificaciones, antes de Mux5 se hace `XOR` con S_r para dar una nueva serie de rondas. Si el último bloque es incompleto A_s a este se rellena con $1||0^*$ con el modulo

Pad. Tan pronto haya terminado toda la fase de absorción, a S_c se le debe adicionar un 1 en el último bit, nuevamente Mux4 contiene dicho valor.

Cifrado: En esta fase los bloques P_i entran por Mux3 cuando son completos. Antes de Mux5 se absorben con S_r, para S_c no hay cambios hasta el final. El bloque cifrado C_i es S_r una vez que se haya absorbido el bloque P_i. Ambos flujos S_r y S_c se unen para realizar las rondas. Cuando el último bloque P_t sea incompleto al igual que A_s incompleto este tiene que rellenarse, sin embargo, el cifrado C_t tienen que truncarse a la longitud $|P_t|$ ambas operaciones realizadas por el módulo Pad Trunc. Al final del cifrado de todos los bloques se adiciona la llave, similar a la parte de inicialización solo que se encuentra de la forma $|K||0^{256}|$ seleccionado por Mux4.

Decifrado: Es igual que la fase de *Cifrado*, en lugar de absorber un bloque C_i este sustituye a S_r y el texto cifrado se restablece haciendo `XOR` de S_c con C_i. El bloque final C_t si es incompleto se hace el padding y sustituye a S_r mientras que P_t obtiene realizando `XOR` de S_r con P_t truncando el resultado al tamaño $|P_t|$. Se puede observar que la entrada par el C_i descifrado está en verde, así como la salida P_i

Finalización: Al finalizar las 12 rondas se produce la etiqueta de autenticación con los últimos 128 bits de S_c realizando una `XOR` con la llave y la señal `valid T` se mantienen en alto.

En esta implementación se considera a los casos cuando los datos asociados son vacíos, cuando el texto plano está vació o ambos, debido a esto el Mux4 puede mandar los datos adecuados para cada uno de estos casos.

En el diagrama de la Figura 4.7 resalta en color rojo el flujo de datos para el enmascaramiento de las llaves. De esta forma se protegen cuando abandonan el criptoprocesador y se almacenan en memoria no volátil. En la parte de cifrado se direcciona el flujo con el Mux7 dejando pasar la parte alta o la baja de la llave, de esta forma el cifrado contienen la llave enmascarada y puede ser almacenada. En el caso contrario cuando se descifra, en lugar de salir del criptoprocesador, los bloques entrar a la unidad administradora de llaves donde son almacenadas, por lo tanto, se tienen que bloquear la señal para que se escriba a memoria. En ambos casos se usa una llave maestra, de momento se implementa como una ROM.

Módulos de relleno

Pad y Pad Trunc, ambos rellenan los bloques incompletos, en la implementación es más sencillo hacer ceros la parte faltante para completar el bloque con una máscara de unos del tamaño del bloque y finalmente realizar una `OR` con un uno recorrido a la posición deseada. Entonces estos están compuestos de dos multiplexores que contienen los valores de las máscaras y el corrimiento de uno. Para simplificar las operaciones, el menor tipo de dato es un byte, entonces hay siete rellenos diferentes, siete tipos

de bloques incompletos según el número de bytes que tengan. Para truncar los datos simplemente se hace realizando una máscara de los bytes que contiene el bloque. En todos los algoritmos, los tamaños de datos asociados, mensaje, texto plano y texto cifrado se indican en bytes.

Unidad de control

Esta unidad básicamente se encarga de controlar las entradas a la permutación, recircular los datos y producir las salidas. Esta unidad es la que se comunica con el dispositivo que esté utilizando el criptoprocesador por ello tiene como señales de salida `valid T`, `busy`, `ready`, `valid w`, para indicarle al dispositivo su estado de procesamiento. A su vez el dispositivo controla esta unidad con las señales `init`, `busy w`, `busy r`, `valid r`, `mode`. Entre el dispositivo y el criptoprocesador se intercambian los parámetros, datos de entrada, así como la salida del procesamiento de los datos. La Figura 4.8 clarifica la construcción máquina de estados que funge como control principal para el criptoprocesador. Esta máquina de estados controla directamente a la máquina que controla la permutación. La máquina consiste de los siguientes estados:

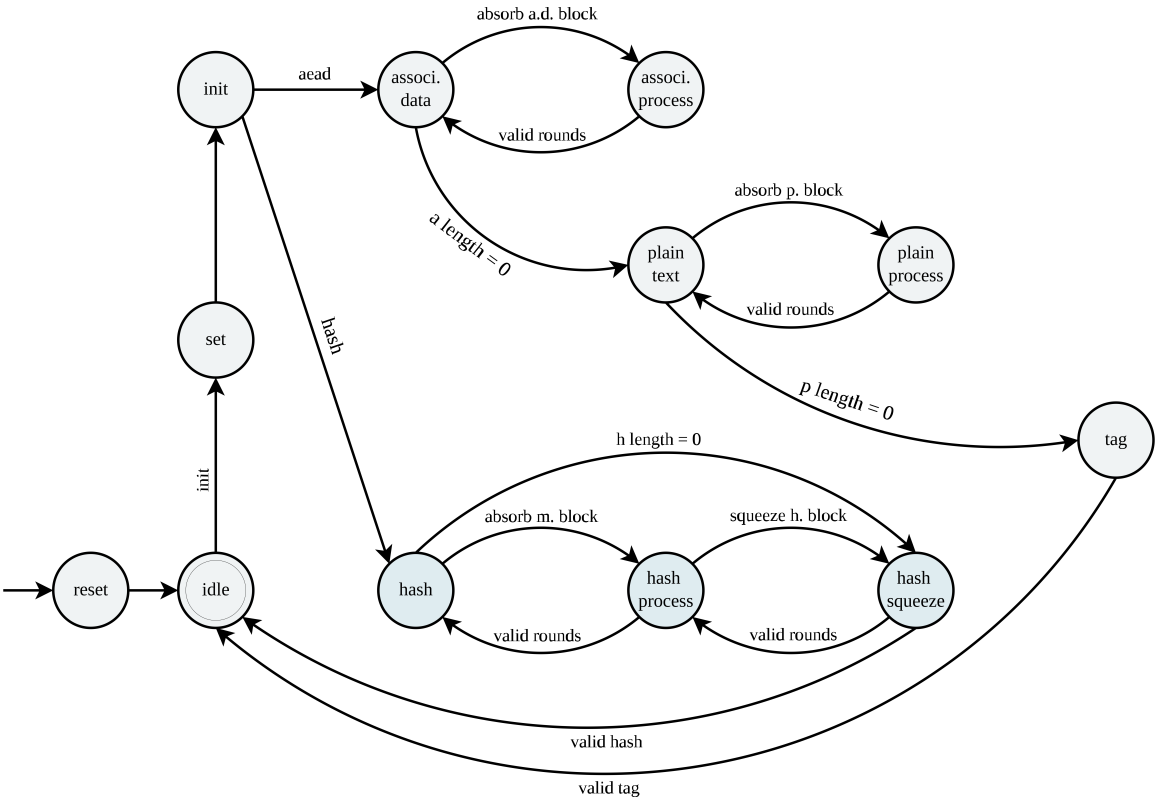


Figura 4.8: Máquina de estados para la ejecución de Ascon-128 y Ascon-Hash.

Reset: En este estado se inicializan en cero todos los contadores y registros.

Idle: Cuando se encuentra en el estado `idle` se espera a la señal de `init` a que esté en alto para pasar al siguiente estado `set`. Se asume que las entradas y parámetros ya fueron establecidos previamente, los tamaños del texto plano $|P|$ y los datos asociados $|A|$, y el modo en el que se va a operar $\mathbf{0000_2}$ para cifrado, $\mathbf{0001_2}$ descifrado, $\mathbf{0010_2}$ hash y $\mathbf{0100_2}$ aleatorios. Este es el estado de aceptación ya que cuando se regresa a este estado indica que ha terminado el proceso de los algoritmos, nuevamente se reinicializan los registros y contadores internos.

Set: En este estado se les establece los valores internos de los módulos a un valor inicial, incluyen los valores que van a pasar en los multiplexores dependiendo de $|A|$ y $|P|$ y el número de rondas `type` y el estado `wait` en el control de la permutación. Al siguiente ciclo de reloj la señal de `start` se establece en verdadero para indicar a la permutación que de la primera serie de rondas para la parte inicial del algoritmo.

Init: Al primer ciclo de reloj se baja la señal `start`. Posteriormente se espera a que la permutación tenga su señal `busy` en bajo para poder pasar al estado de absorción de los datos asociados ya que se seleccionó el modo cifrado o descifrado.

Associated data: La primera comparación que se hace en este estado es confirmar si existen datos asociados, de lo contrario pasa al estado de procesamiento del texto plano. Este estado se cicla con el procesamiento de datos asociados (`Associated process`) hasta agotar los bloques que conforman a $|A|$, cuando no quedan más bloques se pasa al procesamiento del texto plano. Primero se debe esperar a que el bloque de entrado sea válido es decir `valid r` esté en uno. Del total de bloques en cada ciclo entre estos dos estados se restan ocho unidades ya que se cuentan los bytes. En este estado se detecta cuando un bloque no está completo y se indica a los multiplexores para que dejen pasar el dato con el relleno. Cuando se pasa al estado `Associated process` se pone en alto la señal `start` para indicar a la permutación que los datos de su entrada son válidos para la siguiente serie de rondas.

Associated process: En el primer ciclo se baja la señal `start` y se espera a que la permutación haya terminado para regresar al estado `Associated data`.

Plain text: Es un proceso similar al estado `Associated data` solo que este se espera a que se tenga un dato valido de entrada y la que la escritura de un bloque este libre para recibir otro bloque. En este caso se van decrementando los bytes de texto plano o el cifrado. Cada que un bloque de cifrado o texto plano se produzca se lanza la señal `valid w` para escribir el bloque. Igualmente, este estado esta ciclado con `plain process` para esperar a que las rondas se hayan completado. Si no existen bytes de texto plano o se han procesado todos los bloques se pasa al estado `Tag` y se inicializa la última serie de rondas.

Plain process: Espera a que la permutación concluya con las rondas determinadas.

Tag: En este estado se espera a que se hayan terminado las doce rondas y se produce la etiqueta. También se indica con una señal que la etiqueta es validad. Finalmente se regresa a estado `idle` a esperar otra llamada al criptoprocesador.

4.3.2. Función Hash

Usando los mismos módulos que direccionan el flujo de datos del modo de cifrado y descifrado se pueden utilizar para implementar la función hash con Ascon. El siguiente diagrama mostrado en de la Figura 4.9 destaca los módulos y rutas donde los datos pasan para dar soporte a la función hash. Este flujo inicia inicializando a S_r con el vector de inicialización IV_{Hash} y S_c lleno de ceros, controlado por Mux3 y Mux4. Después de dar las rondas de dar las doce rondas de inicialización se comienza absorbiendo los bloques de mensaje M_i, en adelante S_c solo se recircula. Cuando sea el último bloque del mensaje se verifica si es completo de otra forma se le agrega el relleno con el módulo Pad, similar al algoritmo anterior. Finalmente, el algoritmo termina cuando se hayan exprimido los cuatro bloques H_i para formar el hash de 256 bits. Al igual que el algoritmo anterior se utilizan las mismas señales de control que determinan el algoritmo a usar, cuando se inicia, la cantidad de bytes que contienen M y las señales de control de acceso a memoria.

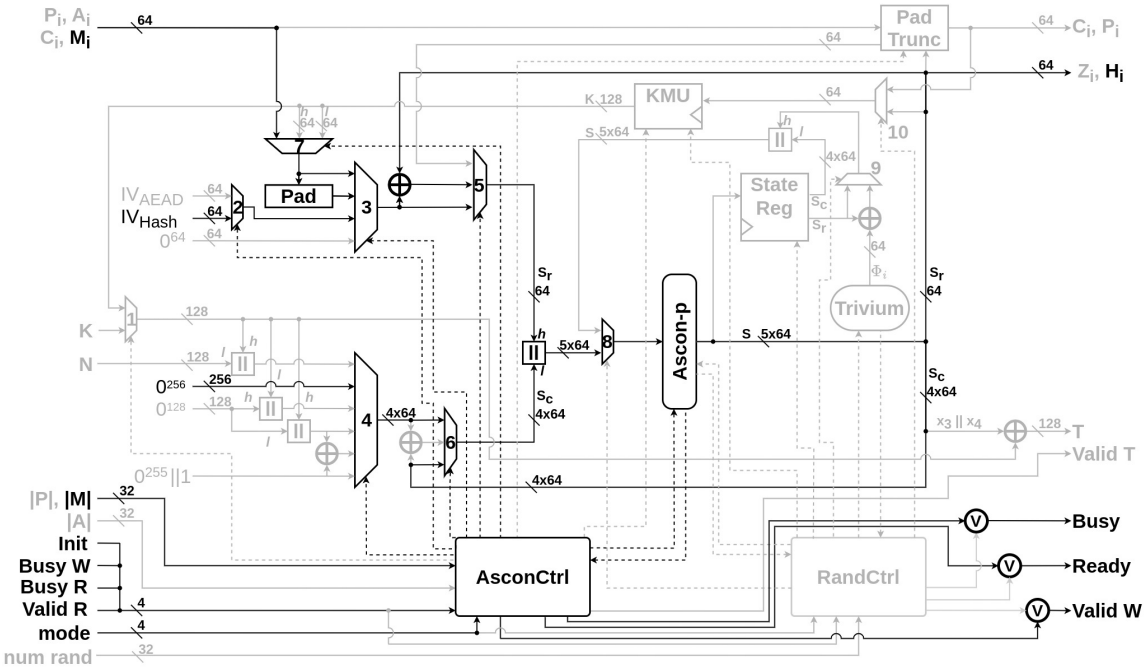


Figura 4.9: Diagrama con flujo de datos para el modo hash.

Control para el modo hash

Se utiliza el mismo control que el modo de operación de cifrado Figura, solo que los estados de color azul son los que son únicos para el hash. Como se puede observar en el estado `init` se bifurca en dos dependiendo del modo de operación. A continuación, se explican los estados en azul cuando la señal de `hash` está en alto y no `aead`, inicialmente se sigue la misma ruta de estado solo van cambiando los datos con los que se inicializa el estado.

Hash: Este es un estado similar a `associated data` solo que se realiza con los bloques M_i y el parámetro de las rondas se incrementa a doce. Cada que se haya absorbido un bloque, es decir que el dato cargado sea válido, el siguiente estado es `hash process` donde se inician las rondas. En cada ciclo entre estos dos estados se restan 8 bytes hasta agotar la cantidad de bloques, entonces se pasa al estado para exprimir el hash.

Hash process: Su función de este estado es esperar a que la permutación tenga un valor valido, entonces vuelve al estado `hash`.

Hash squeeze: Finalmente en este estado se producen los bloques que conforman al hash. Se espera hasta que se haya escrito correctamente el bloque antes de producir uno nuevo. Este estado usa nuevamente a `hash process` para esperar a que las rondas terminen. Una vez que se complete el hash se lanza la señal `valid hash` para indicar que ha terminado el proceso.

4.4. Generador Pseudo-aleatorio

En la Figura 4.10 se remarca el flujo de datos para producción números pseudo aleatorios. Este algoritmo inicializa con el estado en ceros. Para este modo el particular se guarda el estado de la permutación independientemente del registro que se tienen en Ascon-p, con el fin de disponer de datos aleatorios en cualquier fase de una aplicación. De esta forma se aprovechan los bits pseudoaleatorios generados. Para absorber una nueva semilla Φ_i o producir aleatorios Z_j se controla mediante Mux9. Se puede ver a este algoritmo similar al hash con múltiples fases de absorción y exprimido de tamaño arbitrario. La fuente, por el momento está siendo simulada por el algoritmo Trivium que es un cifrador por flujo de datos. En esta implementación se utilizan bloques completos, tanto en la entrada como la salida. Lo cual simplifica el procesamiento al no requerir de relleno para bloques incompletos.

Además de poder producir números pseudoaleatorio, también es útil para producir nuevas llaves simétricas. De manera similar que el descifrado de una llave enmascarada, la producción de llaves toma a S_r para ser almacenada en la unidad administradora

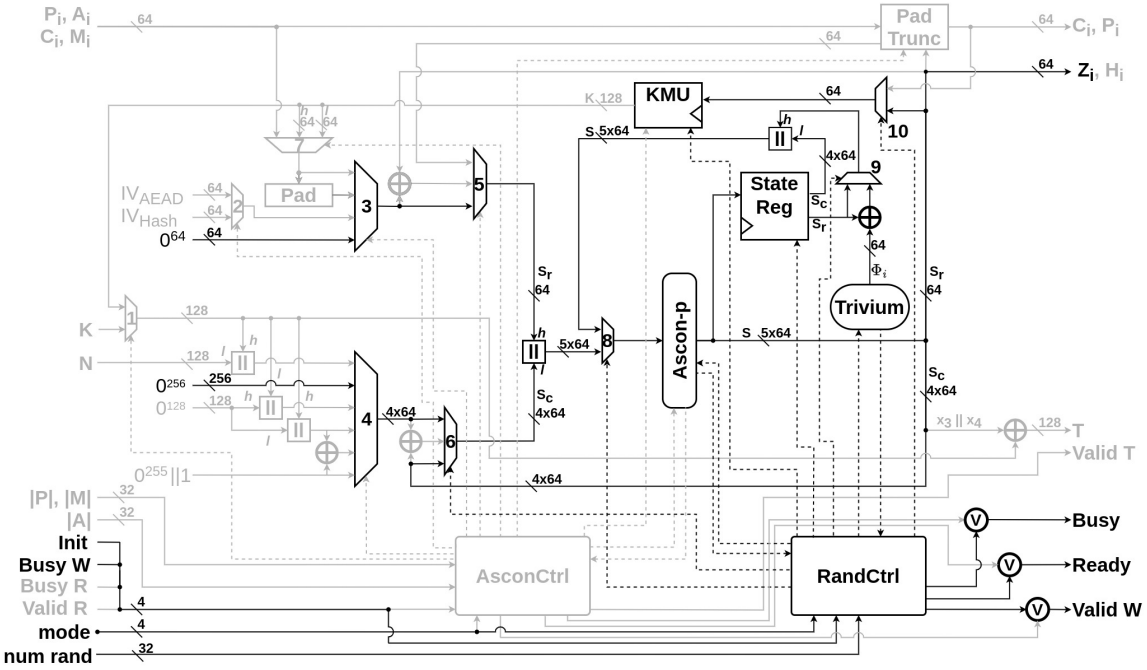


Figura 4.10: Diagrama con flujo de datos para el modo resemillable.

de llaves. Como las llaves utilizadas tienen un tamaño de 128 bits, solo se necesitan dos etapas de exprimido para completar los bits necesarios. La nueva llave puede ser utilizada para realizar nuevos procesos de cifrado y descifrado.

La máquina de estados que controla el modo resemillable es similar a la de cifrado y hash, se muestra en la Figura 4.11. Empieza con doce rondas con un estado inicializado completamente en ceros, cuando el estado `idle` recibe un pulso de inicio. Después se verifica la señal del proceso que se quiere realizar, ya sea absorber una semilla o producir n números pseudoaleatorios. Internamente se tienen una señal que indica si nunca se ha absorbido ninguna semilla. Si no hay ninguna semilla absorbida, antes de realizar la producción de aleatorios, se tienen que absorber una semilla, por lo tanto, se redirecciona el control por los estados que absorben una semilla.

Siempre que una semilla sea absorbida se tienen que dar 12 rondas a la permutación para poder difundir correctamente los 64 bits en todo el estado, cuando haya terminado se regresa al estado `idle`. El nuevo estado no se pierde con una nueva llamada u otro algoritmo, se queda guardado en los registros *State Reg*. El algoritmo Trivium puede producir 64 bits en cada ciclo, siempre que se hayan realizado las 18 rondas de inicialización (para la versión de bit son cuatro ciclos completos, es decir 4 * 288), por lo tanto, con un generador de número de aleatorios se tiene que esperar a que la entropía se haya acumulado y los bits producidos sean válidos.

En la producción de números pseudoaleatorios en lugar del número de bytes de

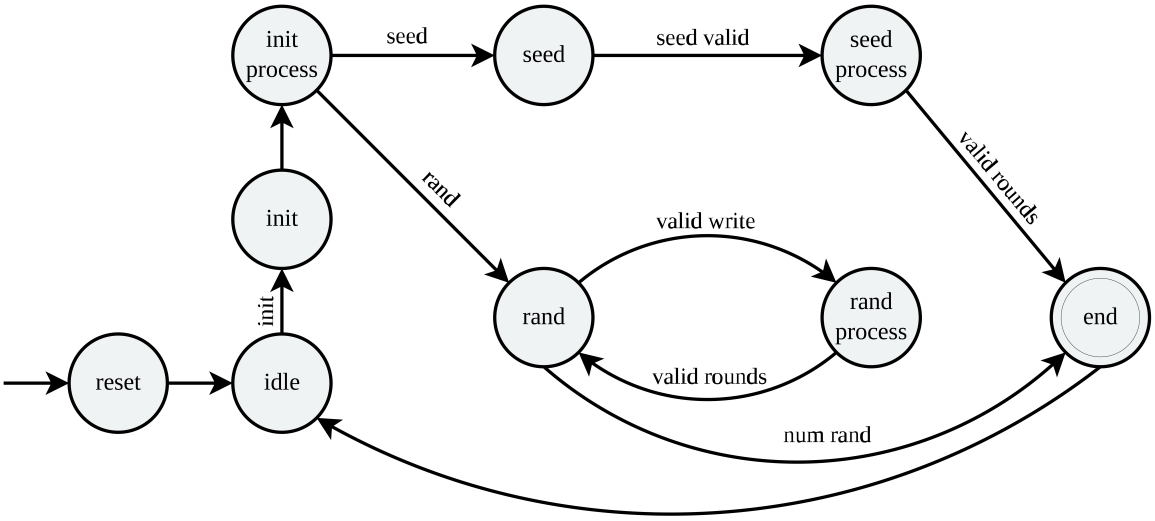


Figura 4.11: Máquina de estados para el control de la absorción de semillas y generación de aleatorios.

los bloques como los algoritmos anteriores, se tiene la cantidad de número que se van a producir. En el estado `rand` se inicializan las rondas, mientras que el `rand process` se espera a que terminen las rondas. Canda que se haya obtenido un nuevo número aleatorio de 64 bit este se tiene que almacenar. Se generan nuevos números cuando se cicla entre estos dos estados, una vez que se obtenga la cantidad deseada, entonces termina el algoritmo y se regresa al estado `idle`. Al igual que en la absorción de semillas, el estado se guarda para un uso posterior.

4.5. Administrador de llaves

Anteriormente se han mencionado los usos de la unidad administradora de llaves (KMU). Este módulo se encarga de almacenar, distribuir y eliminar las llaves simétricas que corresponden a la misma longitud de las llaves utilizadas en el cifrado. Está compuesta de n registros de 128 bits, usada identificadores para acceder a las llaves. De esta forma se direcciona a las llaves por referencia, evitando ser almacenadas en la memoria cache. Se tiene dentro una llave maestra que es usada para cifrar las llaves cuando necesiten ser almacenadas fuera del criptoprocesador.

Su lógica es sencilla, con las tres señales `delete, write, read` se indica si se desea eliminar una llave, si se escribe una llave o se quiere cargar una llave, con K_{id} se hace referencia a que llave se utiliza. En el almacenamiento de llaves se debe de indicar con `high` si se escribe en la parte alta o en la parte baja cuando es cero. Esta unidad está presente en el cifrado, proporcionando las llaves y en la generación de números pseudoaleatorios al crear nuevas llaves.

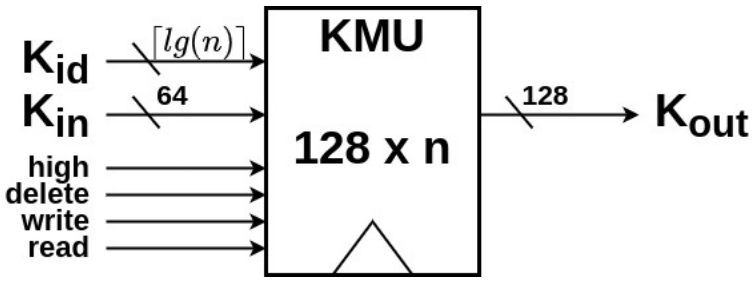


Figura 4.12: Diagrama de la unidad administradora de llaves KMU.

4.6. Conexión con el núcleo Rocket

Ya que se tiene el diseño, lo que falta definir es la manera de suministrar los datos de entrada, guardar las salidas y controlar al criptoprocesador. Se utilizó la interfaz especial RoCC de Rocket-Chip para agregar el criptoprocesador. Ese procesador está en una configuración de coprocesador fuertemente acoplado ya que intercambia datos directamente con los registros del núcleo y ligeramente acoplado porque se comunica con la memoria cache de nivel 1 para incrementar el flujo de datos. Se necesita de hardware adicional que sea capaz de manejar las instrucciones que lleguen al coprocesador, para guardar datos que se requieran cargar o almacenar y para coordinar los accesos a memoria.

4.6.1. Manejador de la interfaz RoCC

La Figura 4.3 muestra los módulos necesarios para conectar el criptoprocesador con la interfaz RoCC, llamada interfaz administradora. Consiste de tres componentes principales, una unidad de decodificación, registros y unidad de control a accesos a memoria. Los registros almacenan temporalmente los datos de entrada como los bloques, el tamaño de los mensajes, la etiqueta de autenticación, el número público y la llave, también se van a encargar de guardar los bloques resultantes y la etiqueta de autenticación. En resumen, se encargan del intercambio entre el procesador, la memoria y el criptoprocesador. Lo que resta es coordinar cuando los registros tienen datos válidos para que puedan ser procesados o puedan ser escritos a memoria.

Instrucciones personalizadas para el control de criptoprocesador

La interfaz RoCC proporciona la facilidad de controlar a los coprocesadores con instrucciones propias de la ISA RISC-V que aún estén disponibles, que no se hayan definido para un conjunto de instrucciones. Se usa el formato de instrucciones del

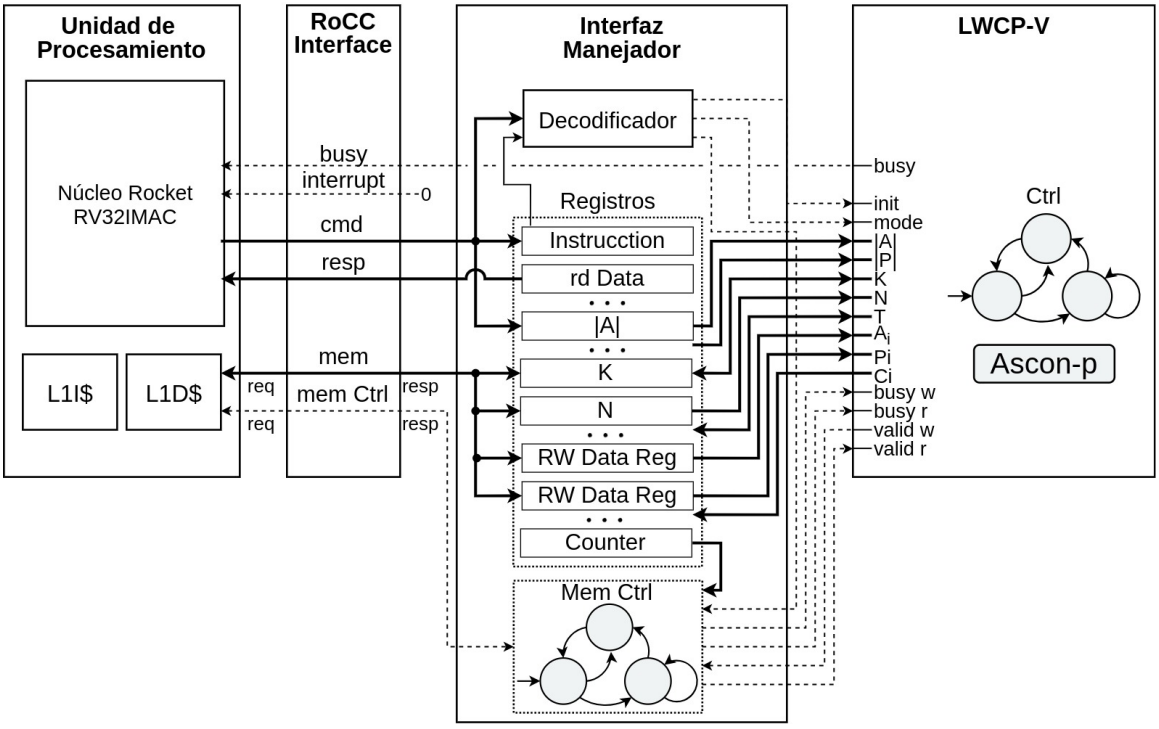


Figura 4.13: Comunicación del procesador con el criptoprocesador.

Capítulo 2, ilustrado en la Figura 2.10. Como se mencionó anteriormente se pueden pasar al coprocesador los datos contenidos en los dos registros `rs1` y `rs2`, a su vez el coprocesador puede regresar un dato a registro `rd` del procesador. En todas las instrucciones siguientes se usa el `opcode custom-0` que tienen la siguiente cadena binaria como identificador $\mathbf{0001011_2}$, por lo tanto, deja libre los siete bits de `funct7` para instrucciones propias, `xd, xs1` y `xs2` solo van a indicar que registros se utilizan. Las instrucciones para cada modo están definidas en las siguientes tablas, Tabla 4.1 para el cifrado y descifrado, Tabla 4.2 para la función Hash, Tabla 4.3 para la generación de números pseudo aleatorios y la Tabla 4.4 para administrar las llaves. Para los distintos modos se utiliza los tres bits más significativos para clasificar a las instrucciones. En el descifrado se emplean las mismas instrucciones que el cifrado (instrucciones 1 a 5) solo se cambia la sexta instrucción por la octava y se agrega la instrucción para cargar la etiqueta con la que se compara la nueva etiqueta. Si se desea utilizar las llaves de la KMU entonces se utiliza la instrucción 9 en lugar de la 4.

En general dentro de los identificadores del registro se definen las direcciones de memoria a donde se van a escribir los datos o de donde se van a cargar los datos [ ], el tamaño de las direcciones de memoria en bytes | |, identificadores de las llaves en la unidad administradora de llaves y de retorno se obtiene el valor que indica que el algoritmo terminó correctamente. Cuando se utiliza una instrucción que requiere cargar un dato el coprocesador devuelve el control hasta el dato contenido en memoria

Tabla 4.1: Definición de las instrucciones para el cifrado `funct7(6:4)` = 001_2.

Operación	`funct7(3:0)`	`rd`	`rs1`	`rs2`
Definir texto plano	$\mathbf{0001_2}$	-	[P]	|P|
Definir datos a.	$\mathbf{0010_2}$	-	[A]	|A|
Definir cifrado	$\mathbf{0011_2}$	-	[C]	-
Cargar nonce	$\mathbf{0100_2}$	-	[N]	-
Cargar llave	$\mathbf{0101_2}$	-	[K]	-
Iniciar cifrado	$\mathbf{0110_2}$	final	-	-
Cargar etiqueta T	$\mathbf{0111_2}$	-	[T]	-
Iniciar descifrado	$\mathbf{1000_2}$	final	-	-
Definir llave ID	$\mathbf{1001_2}$	-	ID	-

Tabla 4.2: Definición de las instrucciones para realizar hash `funct7(6:4)` = 011_2.

Operación	`funct7(3:0)`	`rd`	`rs1`	`rs2`
Definir mensaje	$\mathbf{0001_2}$	-	[M]	|M|
Iniciar hash	$\mathbf{0010_2}$	final	-	-

sea cargado. Siempre se tiene que establecer y cargar los datos antes de inicializar los algoritmos. Cada uno de los modos está definido por los 3 bits más significativos de `funct7`, $\mathbf{001_2}$ para cifrado autenticado, $\mathbf{011_2}$ en la producción de un hash y $\mathbf{100_2}$ para los números pseudoaleatorios. A nivel software al utilizar estas nuevas instrucciones, no tienen que compilarse, en otras palabras, en lenguaje maquina se tienen que definir estas instrucciones con los 32 bits de la instrucción. Sin embargo, se necesitan tres instrucciones adicionales que van a definir los registros a utilizar y cargar los datos a los registros fuente. Para el registro de retorno se utiliza en registro `x10 = a0`, para el primer registro fuente se utiliza `x11 = a1` y el último registro fuente es `x12 = a2`.

Control de accesos a memoria

Como unidad de coordinación entre en coprocesador y el procesador se tienen una máquina de estados. Esta máquina se encarga de gestionar la escritura y lectura de memoria a los registros internos. Esta máquina esta activa cuando se requiera un acceso a memoria, en el caso de la comunicación entre los registros del procesador y los registros internos se mantiene en un estado `wait`. La Figura 4.14 muestra el diagrama de dicha máquina de estados. A continuación, se describen los estados de la máquina de estados.

Idle: En este estado se espera a que se reciba un comando valido para el criptoprocesador, si el comando o instrucción requiere que se carguen datos se pasa al estado `wait`. Si es un comando no válido termina en el estado `end`. En este estado la señal

Tabla 4.3: Definición de las instrucciones para generar números pseudoaleatorios `funct7(6:4)` = 100_2.

Operación	`funct7(3:0)`	`rd`	`rs1`	`rs2`
Obtener números	$\mathbf{0001_2}$	-	[R]	cantidad
Absorber semilla	$\mathbf{0010_2}$	final	-	-

Tabla 4.4: Definición de las instrucciones para administrar llaves `funct7(6:4)` = 101_2.

Operación	`funct7(3:0)`	`rd`	`rs1`	`rs2`
Generar llave	$\mathbf{0001_2}$	-	ID	-
Eliminar llave	$\mathbf{0010_2}$	-	ID	-
Cifrar llave	$\mathbf{0011_2}$	-	ID	-
Descifar llave	$\mathbf{0100_2}$	-	ID	-

`busy` es cero para que el procesador pueda seguir trabajando.

Wait: En `wait` se esperan las señales del criptoprocesador cuando necesite un nuevo bloque, quiera guardar un bloque, o se realice la comparación de las etiquetas de autenticación cuando termina el descifrado. Hay que recordar que el tamaño de bloque que trabaja en criptoprocesador es de 64 bits y la arquitectura es de 32 bits. Debido a esto los bloques que se carguen al coprocesador tienen que construirse a partir de palabras de 32 bits, en contra parte la escritura se tienen que escribir de palabra en palabra. Cuando se requiere cargar un bloque se pasa al estado `load req` y si se va a guardar el control pasa al estado `write req`.

Load request: Al pasar a este estado se manda a la memoria la solicitud, se indica que es una carga, se indica la dirección de memoria, el tamaño del dato que se requiere (en este caso 32 bits o una palabra) y un tag como identificador de la solicitud. Se pasa al estado `load` cuando la memoria ha reconocido a la solicitud. Este ciclo que repite hasta que se haya cargado la cantidad de palabras para formar un bloque, o en el caso de llaves y nonces, la cantidad de palabras para formar los 128 bits. En este estado y en `load` se le indica al criptoprocesador que la carga aún no ha sido completada.

Load: En este estado se espera a que el dato que venga de la memoria sea válido y se guarda. Regresa el control a `load req.` hasta que se complete el tamaño del dato que se desea y aumenta la dirección de memoria más 4 unidades, si se ha alcanzado el tamaño entonces se espera a otro acceso a memoria en `wait`.

Write request: Es un caso similar al estado de carga solo que en este se indica que se va a realizar una escritura a memoria y se pasa el dato de 32 bits que se quiere guardar. Cuando se ha solicitado una escritura se pasa al estado `write`. En este estado y en `write` se le indica al criptoprocesador que aún no se ha escrito el dato, hasta

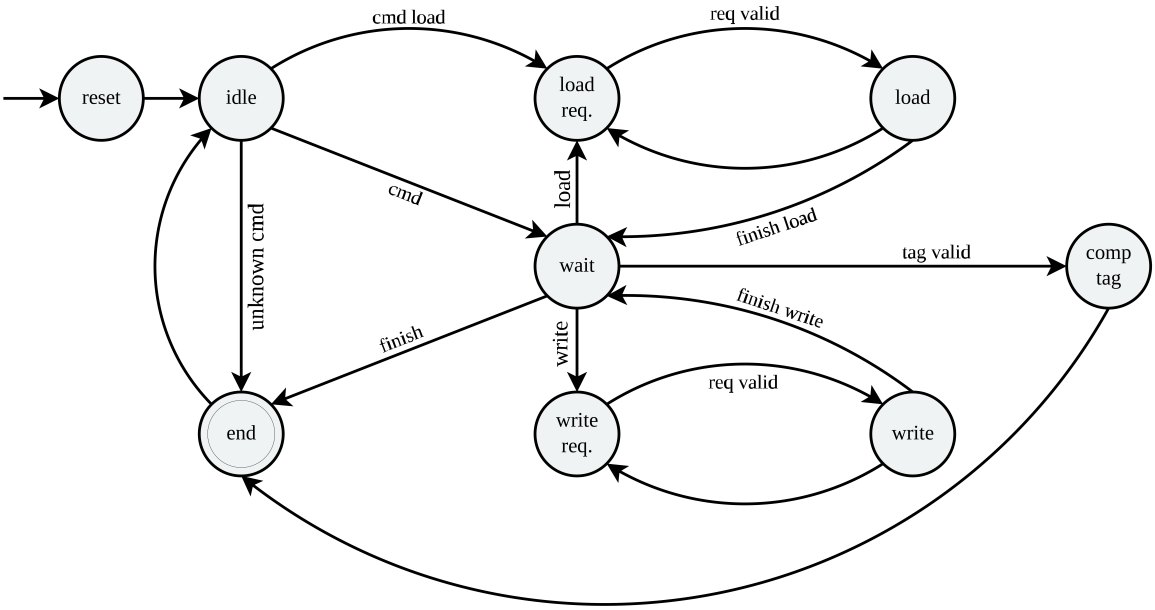


Figura 4.14: Máquina de estados para el control de escritura y lectura a la caché nivel L1.

que se pase a `wait`.

Write: Al igual que el estado `load` se espera a la respuesta de la memoria, de que la palabra se ha escrito correctamente. Si aún quedan palabras que guardar se aumenta la dirección de memoria de la escritura en cuatro unidades y se regresa al estado `write req`. Si ya se han escrito todas las palabras se regresa al estado `wait`. Cuando algún algoritmo haya terminado se va al estado de termino `end`.

Compare tag: Este estado indica el final del descifrado en donde se compara la etiqueta del cifrado con la etiquete recién obtenida si estas coinciden se manda la señal de validez $\perp$. En este caso la señal se retorna al procesador como el dato del registro de destino que es **0xFFFFFFFF**. Al finalizar se va al estado `end`.

End: Indica el termino y vuelve al estado `idle`.

4.6.2. Caché nivel 1 de Rocket-Chip

El tamaño máximo de las palabras que se pueden almacenar en la memoria cache L1 es consistente con el tamaño de los registros de la arquitectura. En el caso de 32 bits se incluyen los tamaños de media palabra (16 bits) y un byte. A pesar de que la documentación menciona que la señal `tag` para realizar una petición tienen un tamaño de 20 bits y que estos bits pueden ser arbitrarios, esta necesita un número menor de bits que están definidos por las características del SoC y solo ciertos bits puede ser

modificada.

La longitud en bits de la señal `tag` está determinada por el $\lceil log_2 \rceil$ de los puertos disponibles y los bits de la etiqueta de petición a memoria. Los puertos disponibles son la suma de núcleos, el uso de direcciones virtuales, las memorias fuertemente integradas de datos (DTIM, por sus siglas en inglés), más los coprocesadores y los bits de petición a memoria por defecto son seis bits. En el caso de la configuración E310 se utiliza: `1 core + 6 bits = 7 bits` de longitud de la señal `tag`. Cuando se agrega el coprocesador esta cuenta aumenta a ocho. Solo los seis bits del `tag` pueden ser modificados, para los otros bits se concatenan ceros hasta obtener el tamaño de bits deseados. Si se modifican los bits más significativos que no sean los seis o son iguales en solicitudes concurrentes, el procesador se bloquea y detiene. El mismo comportamiento ocurre cuando se leen direcciones fuera del tamaño de la RAM. Cuando se detiene el procesador, la única forma de salir de ese estado es reiniciarlo y cambiar el software o hardware. Para solucionar el problema, en hardware se implementó un contador de seis bits que aumenta la cuenta por cada petición y se le concatenan ceros en los bits más significativos. Con los bits más significativos el procesador puede distribuir los datos al módulo que realizo esa petición.

4.6.3. SoC personalizado

Finalmente al tener la descripción del criptoprocesador, controlado con el hardware para la comunicación del el criptoprocesador con el procesador este se coloca en la plantilla mínima para instanciar un coprocesador, esta se muestra en el Código 4.1 [9]. Esta clase que hereda de `LazyRoCC` contiene la interfaz RoCC. Consiste en conectar esta interfaz con todos los módulos que se han creado para implementar el criptoprocesador. En esta clase se instancia el criptoprocesador junto al hardware que lo controlara. Desde esta clase también se tiene acceso a la memoria mediante de la interfaz.

Código 4.1: Instancia mínima para un coprocesador RoCC.

```
class CustomAccelerator(opcodes: OpcodeSet)
    (implicit p: Parameters) extends LazyRoCC(opcodes) {
  override lazy val module = new CustomAcceleratorModule(this)
}

class CustomAcceleratorModule(outer: CustomAccelerator)
    extends LazyRoCCModuleImp(outer) {
  val cmd = Queue(io.cmd)
  // The parts of the command are as follows
  // inst - the parts of the instruction itself
  //   opcode
  //   rd - destination register number
  //   rs1 - first source register number
  //   rs2 - second source register number
  //   funct
  //   xd - is the destination register being used?
```

```
  //   xs1 - is the first source register being used?
  //   xs2 - is the second source register being used?
  // rs1 - the value of source register 1
  // rs2 - the value of source register 2
  ...
}
```

Ya que el criptoprocesador contiene la interfaz se puede agregar al SoC RISC-V. Para ello se utiliza la siguiente clase mostrada en el Código 4.2, donde se definen los coprocesadores, en este caso solo con el `opcode-custom0` basta. Con esta clase de configuración ya puede se agregada a otras arquitecturas dentro de Rocket-Chip.

Código 4.2: Configuración del SoC

```
class WithCustomAccelerator extends Config((site, here, up) => {
  case BuildRoCC => Seq((p: Parameters) => LazyModule(
    new CustomAccelerator(OpcodeSet.custom0 | OpcodeSet.custom1)(p)))
})

class CustomAcceleratorConfig extends Config(
  new WithCustomAccelerator ++
  new RocketConfig)
```

El Código 4.3 representa al archivo de configuración para toda la arquitectura del SoC, que contiene al procesador y a el criptoprocesador ligero, en rosa se resalta `WithLWCustom` que tiene la definición del hardware del criptoprocesador y `TinyConfig` que define al un núcleo RV32IMAC, con una memoria caché de datos de 16KB, junto a una memoria de instrucciones de 4KB. La Figura 4.15 representa gráficamente los módulos que se interconectan con el la configuración, como puede observarse dentro del procesador (Rocket Tile) se encuentra el núcleo de 32 bits, las caches y el criptoprocesador. Al estar en el mismo nivel se puede pueden intercambiar datos más rápido.

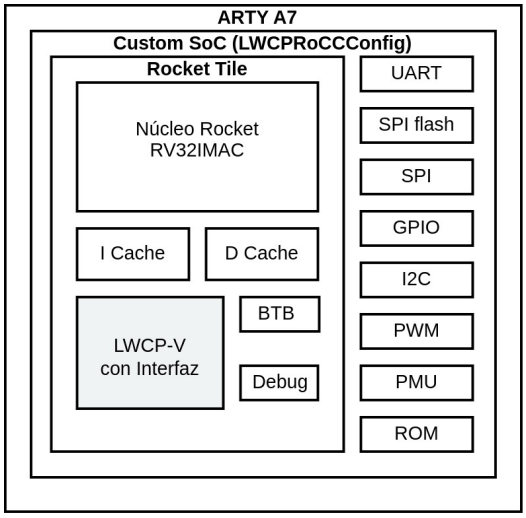


Figura 4.15: SoC Rocket-Chip con el criptoprocesador ligero integrado.

Código 4.3: Configuración del SoC

```
class LWRoCCConfig extends Config(
    new WithNBreakpoints(2) ++
    new WithNExtTopInterrupts(0) ++
    new WithJtagDTM ++
    new WithL1ICacheWays(2) ++
    new WithL1ICacheSets(128) ++
    new WithDefaultBtb ++
    new WithLWCustom ++
    new TinyConfig)
```

Estos archivos de configuración modifican la arquitectura del núcleo y el procesador que se crea. A la configuración por defecto solo se le agrega el criptoprocesador. Los archivos de configuración pueden verse como una colección de parámetros, estos a su vez pueden contener otras configuraciones. La interconexión es automática entre módulos, se hace al programado agnóstico de el proceso para conectarlos. Ya que se tiene el SoC este puede ser mapeado con la FPGA, en este caso es la placa de desarrollo Arty A7. Finalmente se puede sintetizar el archivo binario para implementar el hardware y posteriormente probar su comportamiento con el software.

4.6.4. Consideraciones para la programación a nivel software

Estos registros utilizados en las instrucciones personalizadas son los que se consideran los estándares para paso de parámetros por parte de la ABI. El siguiente Código 4.4 muestra en el lenguaje C para RISC-V como generar las instrucciones personalizadas que consisten en formar la instrucción parte por parte de acuerdo con el formato R. El código se extrajo del repositorio hwacha-template [6] del archivo test/rocc.h.

Código 4.4: Tranformación de instrucciones a ASM

```
#define STR1(x) #x
#define STR(x) STR1(x)
#define EXTRACT(a, size, offset) (((~(~0 << size) << offset) & a) >> offset)

#define CUSTOMX_OPCODE(x) CUSTOM_ ## x
#define CUSTOM_0 0b0001011
#define CUSTOM_1 0b0101011
#define CUSTOM_2 0b1011011
#define CUSTOM_3 0b1111011

#define CUSTOMX(X, xd, xs1, xs2, rd, rs1, rs2, funct) \
  CUSTOMX_OPCODE(X)                     |             \
  (rd                 << (7))           |             \
  (xs2                << (7+5))         |             \
  (xs1                << (7+5+1))       |             \
  (xd                 << (7+5+2))       |             \
  (rs1                << (7+5+3))       |             \
  (rs2                << (7+5+3+5))     |             \
  (EXTRACT(funct, 7, 0) << (7+5+3+5+5))

#define ROCC_INSTRUCTION_DSS(X, rd, rs1, rs2, funct) \
  ROCC_INSTRUCTION_R_R_R(X, rd, rs1, rs2, funct, 10, 11, 12)

 #define ROCC_INSTRUCTION_R_R_R(X, rd, rs1, rs2, funct, rd_n, rs1_n, rs2_n) { \
    register uint32_t rd_  asm ("x" # rd_n);                                  \
    register uint32_t rs1_ asm ("x" # rs1_n) = (uint32_t) rs1;                \
    register uint32_t rs2_ asm ("x" # rs2_n) = (uint32_t) rs2;                \
    asm volatile (                                                            \
        ".word " STR(CUSTOMX(X, 1, 1, 1, rd_n, rs1_n, rs2_n, funct)) "\n\t"   \
        : "=r" (rd_)                                                          \
        : [_rs1] "r" (rs1_), [_rs2] "r" (rs2_));                              \
    rd = rd_;                                                                 \
  }
```

[6] git@github.com:ucb-bar/hwacha-template.git

Para controlar el criptoprocesador se deben implementar las funciones de la siguiente manera. Antes de realizar alguna operación que se ejecute en los coprocesadores Rocket, se debe modificar el registro de control de estatus para que sea reconocido por el procesador. Este registro `mstatus`, que es aquel que controla los hart's que se estén operando. Se establecen los bits 15 y 16 del registro. Estos bits corresponden al campo `XS` que codifica el estado de extensiones de modos de usuario adicionales y estado asociado. En otros términos, habilita al coprocesador.

Cada instrucción para el coprocesador se encapsula entre dos instrucciones `fence`, para que fuerce al procesador a que todas las operaciones de memoria hayan terminado antes de ejecutar una instrucción. Esta es una instrucción de ensamblador en línea en C. Debe de estar entre cada conjunto de datos que se compartan entre el procesador y el coprocesador.

Se lee el CSR que contiene el contador de ciclos de reloj antes y después de la ejecución del algoritmo para obtener el desempeño. El registro que contiene esta cuenta es `mcycle`. Este contador internamente está compuesto de la parte alta `mcycleh` y la parte baja `mcycle`, para la aplicación que se le dará basta con usar la parte baja.

Todos los arreglos son de tipo `static` para que su tamaño esté definido en el tiempo de compilación. También la región de memoria del arreglo es asignada dentro del rango de direcciones definidas para la memoria de datos, tras cada ejecución el apuntador de cada arreglo se mantiene constante. Este rango inicia desde la dirección `0x80000000` hasta la dirección `0x80004000` abarcando un tamaño de 16 KB.

La implementación en software se realizó basándose en el código de referencia ascon-c[7] adaptado para 32 bits para las plataformas de RISC-V y la de ARM. Los algoritmos de Ascon se benefician de arquitecturas que cuenten con registros y unidades aritméticas de 64 bits ya que las operaciones están diseñadas para esta longitud de bits, debido a esto se requiere dividir a los registros de 64 bits en dos de 32 que contengan la parte alta y otro la parte baja. La compilación de los archivos en C se realizó con las banderas de optimización `-OS` para reducir el tamaño y `-O2` para optimizar el tiempo de reloj. La función a evaluar de los algoritmos es ejecutada 30 veces continuas para saturar la cache de instrucciones con aquellas que se utilicen frecuentemente.

4.7. Propuestas del estado del arte:

Dentro de la industria y de la comunidad académica existen propuestas de criptoprocesadores para los núcleos RISC-V, mostrados en la Tabla 4.5. A diferencia de la

[7] git@github.com:ascon/ascon-c.git

propuesta LWCP-V todos los demás criptoprocesadores utilizan los algoritmos estándares para las funciones de cifrado, funciones hash, firma digital, generación de llaves (KG), enmascaramiento de llaves (KW) y generadores de número aleatorios. Muchos de ellos contienen los algoritmos que les ayudan a desempeñar una tarea específica para el sistema que fueron diseñados.

Tabla 4.5: Comparación de propuestas de criptoprocesadores para RISC-V

	PUFcc [23]	TEE HW [24]	SiFive [25] Crypto-Engine	FAC-V [26]	DITES [27] CryptoCore	LWCP-V
Cifrado	AES/SM4	AES	AES	AES	AES	Ascon128
Hash	SHA-2/SM3	SHA-3	SHA-2/SHA-3	-	SHA-1	Ascon-Hash
Algoritmos Públicos	ECDSA/ECDH/RSA	EC 25519	RSA/ECDSA	-	RSA	-
TRNG	PUF	-	✓	-	-	-
KG	KDF	-	-	-	-	Ascon-Prng
KW	AES	-	-	-	-	Ascon128
PRNG	-	LFSR	-	-	-	Ascon-Prng

Además de implementación en hardware de algoritmos de criptografía ligera, existen algunos que forman parte de criptoprocesadores, tal como se refleja en la Tabla 4.6. A estos se le denomina criptoprocesadores ligeros, muchos de estos solo ofrecen el cifrado y generación de números aleatorios. Solo uno de ellos cuenta con el algoritmo Ascon para el cifrado y hash. En cuanto a LWCP-V se buscó solo usar una permutación para ofrecer varios servicios de criptografía, reutilizándolo con distintos modos de operación. A diferencia de los criptoprocesadores presentados en las tablas anteriores, LWCP-V no da soporte a los algoritmos de llave pública.

Tabla 4.6: Comparación de propuestas de criptoprocesadores ligeros.

	PRESENT CC [28]	Humingbird CC [29]	SIMECK CC [30]	PRESENT CC [31]	LWC SoC [32]	RECO-HCON [33]	LWCP-V
Núcleo	-	-	-	LEON3	PICORV32	-	Rocket
Cifrado	PRESENT	Hummingbird	Simeck32/64	PRESENT	PRESENT, NEW	Ascon-128 Ascon-128a	Ascon-128
Autenticado	-	-	-	-	HB, HB+, HBMP, HBMP+	Ascon-128 Ascon-128a	Ascon-128
Hash	-	-	-	-		Ascon-Hash Ascon-Hasha	Ascon-Hash
Algoritmos públicos	-	-	-	-	Intercambio llaves basado en HB	-	-
KG	-	-	LFSR	-	-	-	Ascon-Prng
KW	-	-	-	-	-	-	Ascon-128
PRNG	PRESENT	LFSR	LFSR	-	LFSR	-	Ascon-Prng

Nota: CC (Crypto Core).

Capítulo 5

Resultados

La implementación en hardware del criptoprocesador junto al núcleo de Rocket se realizó en la placa de desarrollo Digilent Arty A7-100T y se comparó el procesamiento del hardware con la implementación en software. Las implementaciones en software se realizaron con dos procesadores de arquitectura RISC. En la placa de desarrollo de SiFive HiFive1 Rev B que contiene en silicio la misma configuración del procesador utilizado en la FPGA, RV32IMAC core E310, sin contar con el coprocesador. Además, se utilizó un ARM Cortex M7 en una placa de desarrollo NUCLEO-F746ZG. De acuerdo con SiFive la serie de procesadores E3 es comparable a los núcleos ARM Cortex M7, R4 y R5. La Tabla 5.1 resume las características de las plataformas.

Tabla 5.1: Principales características de los dispositivos utilizados.

	Plataforma		
Características	**E310 Arty A7**	**HiFive1 Rev B**	**NUCLEO-F746ZG**
Núcleo	RV32IMAC E310	RV32IMAC FE310-G002	Cortex M7 + FPU
Frecuencia	32.5 MHz	320 MHz	216 MHz
Voltaje de operación	0.95 V - 3.3 V	1.8 V - 3.3V	1.7 V - 3.6 V
Memoria Flash	16 MB	4 MB	1 MB
Memoria	16KB Ins. 16KB RAM	16KB Ins. 16KB SRAM	320 KB SRAM
UART	1	2	8
I2C	1	1	4
SPI	1	3	6
Conectividad	Ethernet 10/100 Mbs	WiFi fuera del chip	Ethernet 10/100 Mbs
Depurador	JTAG	J-Link	ST-LINK

5.1. Tiempos de ejecución

Para una comparación justa entre los dispositivos se consideró como parámetro los ciclos de reloj que toma la ejecución de tamaño de datos que formen todos los bloques completos. Con los ciclos de reloj se puede obtener los ciclos por byte, que corresponden a los ciclos que se necesitan para procesar un byte. Como indicador de comparación entre dos implementaciones, se utiliza la aceleración. Esta consiste en

comprar una solución optimizada con la solución original, en este caso se compara la solución en software contra la solución en hardware. Los tiempos de ejecución son los ciclos por reloj de cada tipo de ejecución, de esta forma es independiente de la frecuencia del dispositivo.

$$\begin{aligned} CPB &= \frac{ciclos}{num. \quad bytes} \\ Acc &= \frac{t_{software}}{t_{hardware}} \end{aligned} \tag{5.1}$$

Las comparaciones utilizan tamaños de mensaje de 0, 8, 96, 496, 1000, 5000, 10000 bytes. Todos estos valores son divisibles entre ocho bytes que corresponde al tamaño de S_r para formar solo bloques completos. Con este incremento se puede visualizar el efecto de las fases de inicialización y finalización del desempeño de cada algoritmo.

5.1.1. Ejecuciones en hardware

Las siguientes tablas muestran los ciclos y ciclos por bytes de cada una de las versiones de LWCP-V. Cada versión corresponde con el número de rondas desdobladas en la permutación, LWCP-V_v1 para una ronda por ciclo, LWCP-V_v2 para dos rondas por ciclo y LWCP-V_v3 para tres rondas en un ciclo. Las Tablas 5.8 y 5.9 tienen los resultados del cifrado, las Tablas 5.10 y 5.11 contienen los datos de desempeño del descifrado. Los resultados de la función hash se encuentran en la Tabla 5.12 y el desempeño de generación de pseudoaleatorios está en la Tabla 5.13. El software fue compilado con la bandera de optimización `-O2`, en este caso no hay diferencia con la bandera de optimización `-Os`, tanto en tamaño como en ciclos de ejecución. Se debe principalmente a que las instrucciones se construyen en una sola instrucción y no se compilan.

Tabla 5.2: Rendimiento del cifrado en hardware con texto plano.

	LWCP-V_1		**LWCP-V_2**		**LWCP-V_3**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	218	NA	206	NA	202	NA
8	234	29.250	224	28.000	220	27.500
96	410	4.271	374	3.896	370	3.854
496	1210	2.440	1074	2.165	1070	2.157
1000	2218	2.218	1956	1.956	1952	1.952
5000	10218	2.044	8956	1.791	8952	1.790
10000	20218	2.022	17706	1.771	17702	1.770

Tabla 5.3: Rendimiento del cifrado en hardware con datos asociados.

	LWCP-V_1		**LWCP-V_2**		**LWCP-V_3**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	213	NA	201	NA	197	NA
8	239	29.875	224	28.000	218	27.250
96	415	4.323	367	3.823	350	3.646
496	1215	2.450	1017	2.050	950	1.915
1000	2226	2.226	1836	1.836	1706	1.706
5000	10223	2.045	8336	1.667	7706	1.541
10000	20226	2.023	16461	1.646	15206	1.521

Tabla 5.4: Rendimiento del descifrado en hardware con texto plano.

	LWCP-V_1		**LWCP-V_2**		**LWCP-V_3**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	232	NA	220	NA	216	NA
8	248	31.000	234	29.250	230	28.750
96	424	4.417	388	4.042	384	4.000
496	1224	2.468	1088	2.194	1084	2.185
1000	2232	2.232	1970	1.970	1966	1.966
5000	10232	2.046	8970	1.794	8966	1.793
10000	20232	2.023	17720	1.772	17716	1.772

Tabla 5.5: Rendimiento del descifrado en hardware con datos asociados.

	LWCP-V_1		**LWCP-V_2**		**LWCP-V_3**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	232	NA	220	NA	216	NA
8	258	32.250	240	30.000	234	29.250
96	434	4.521	383	3.990	366	3.813
496	1234	2.488	1033	2.083	966	1.948
1000	2242	2.242	1852	1.852	1722	1.722
5000	10242	2.048	8352	1.670	7722	1.544
10000	20242	2.024	16477	1.648	15222	1.522

Tabla 5.6: Rendimiento del hash en hardware.

	LWCP-V_1		**LWCP-V_2**		**LWCP-V_3**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	163	NA	133	NA	124	NA
8	202	25.250	160	20.000	151	18.875
96	444	4.625	336	3.500	327	3.406
496	1544	3.113	1136	2.290	1127	2.272
1000	2930	2.930	2144	2.144	2135	2.135
5000	13930	2.786	10144	2.029	10135	2.027
10000	27680	2.768	20144	2.014	20135	2.014

Tabla 5.7: Rendimiento de la generación de números pseudoaleatorios en hardware.

	LWCP-V_1		**LWCP-V_2**		**LWCP-V_3**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	68	NA	65	NA	61	NA
8	131	16.375	119	14.875	118	14.750
96	307	3.198	229	2.385	228	2.375
496	1107	2.232	729	1.470	728	1.468
1000	2115	2.115	1359	1.359	1358	1.358
5000	10115	2.023	6359	1.272	6358	1.272
10000	20115	2.012	12609	1.261	12608	1.261

5.1.2. Ejecuciones en software

Las siguientes tablas muestran los ciclos y ciclos por bytes de cada plataforma. El software para cada plataforma se compiló con la bandera *-O2*. Se presentan los resultados con el softcore E310 en la FPGA ArtyA7-100T, la placa de desarrollo de SiFive RevB y una placa NUCLEO con un microprocesador ARM Cortex M7. Las Tablas 5.8 y 5.9 muestran los resultados del cifrado autenticado, las Tablas 5.10 y 5.11 contienen los ciclos y ciclos por byte del descifrado. Los resultados de la función hash están reflejados en la Tabla 5.12 y el desempeño de generación de pseudoaleatorios puede observarse en la Tabla 5.13.

Tabla 5.8: Rendimiento del cifrado en software con texto plano.

	E310 softcore		**HiFive revb**		**NUCLEO M7**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	4276	NA	4276	NA	5702	NA
8	5138	642.250	5138	642.250	6522	815.236
96	14270	148.646	14270	148.646	14849	154.676
496	55720	112.339	55720	112.339	52588	106.025
1000	107947	107.947	107947	107.947	100141	100.141
5000	522447	104.489	522447	104.489	477546	95.509
10000	1040565	104.057	1040565	104.057	949262	94.926

Tabla 5.9: Rendimiento del cifrado en software con datos asociados.

	E310 softcore		**HiFive revb**		**NUCLEO M7**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	4276	NA	4276	NA	5707	NA
8	15327	1915.875	15327	1915.875	7901	987.574
96	29026	302.354	29026	302.354	16085	167.548
496	67250	135.585	67250	135.585	53221	107.300
1000	118092	118.092	118092	118.092	100010	100.010
5000	523434	104.687	523434	104.687	471373	94.275
10000	1028092	102.809	1028092	102.809	935577	93.558

Tabla 5.10: Rendimiento del descifrado en software con texto plano.

	E310 softcore		**HiFive revb**		**NUCLEO M7**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	3661	NA	3661	NA	5770	NA
8	4512	564.000	4512	564.000	6623	827.912
96	13604	141.708	13604	141.708	14902	155.230
496	54904	110.694	54904	110.694	52443	105.731
1000	106942	106.942	106942	106.942	99740	99.740
5000	519942	103.988	519942	103.988	475136	95.027
10000	1036192	103.619	1036192	103.619	944382	94.438

Tabla 5.11: Rendimiento del descifrado en software con datos asociados.

	E310 softcore		**HiFive revb**		**NUCLEO M7**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	3661	NA	3661	NA	5771	NA
8	5649	706.125	5649	706.125	7983	997.907
96	14532	151.375	14532	151.375	16088	167.585
496	54882	110.649	54882	110.649	52881	106.615
1000	105723	105.723	105723	105.723	99240	99.240
5000	509223	101.845	509223	101.845	467146	93.429
10000	1013598	101.360	1013598	101.360	927027	92.703

Tabla 5.12: Rendimiento del hash en software.

	E310 softcore		**HiFive revb**		**NUCLEO M7**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	14848	NA	14848	NA	9156	NA
8	21165	2645.625	21165	2645.625	13346	1668.292
96	36266	377.771	36266	377.771	29559	307.905
496	116098	234.069	116098	234.069	103138	207.940
1000	220581	220.581	220581	220.581	195857	195.857
5000	1011548	202.310	1011548	202.310	931659	186.332
10000	2005924	200.592	2005924	200.592	1851373	185.137

Tabla 5.13: Rendimiento de la generación de número pseudoaleatorios en software.

	E310 softcore		**HiFive revb**		**NUCLEO M7**	
bytes	ciclos	CPB	ciclos	CPB	ciclos	CPB
0	4391	NA	4391	NA	5250	NA
8	6039	754.875	6039	754.875	6806	850.796
96	23854	248.479	23854	248.479	23176	241.415
496	104804	211.298	104804	211.298	97545	196.664
1000	206801	206.801	206801	206.801	191251	191.251
5000	1016301	203.260	1016301	203.260	934957	186.991
10000	2028176	202.818	2028176	202.818	1864587	186.459

5.1.3. Comparación de la implementación en software de los algoritmos contra el criptoprocesador

Finalmente se compara la aceleración ganada al usar la implementación en hardware contra la solución en software. La aceleración se calculó con los ciclos que se tardaron las ejecuciones de cada una de las soluciones. Las tres versiones del criptoprocesador

LWCP-V se compararon con las ejecuciones en software del softcore E310, con la siguiente ecuación $Acc = \frac{ciclos_{software}}{ciclos_{hardware}}$. El comportamiento de la aceleración frente al tamaño de datos de cada algoritmo se observa en las siguientes figuras. El cifrado se encuentra en las Figuras 5.1 y 5.2, el comportamiento del descifrado está ilustrado en las Figuras 5.3 y 5.4, la función hash se muestra en la Figura 5.5 y la generación de números pseudoaleatorios se encuentra en la Figura 5.6.

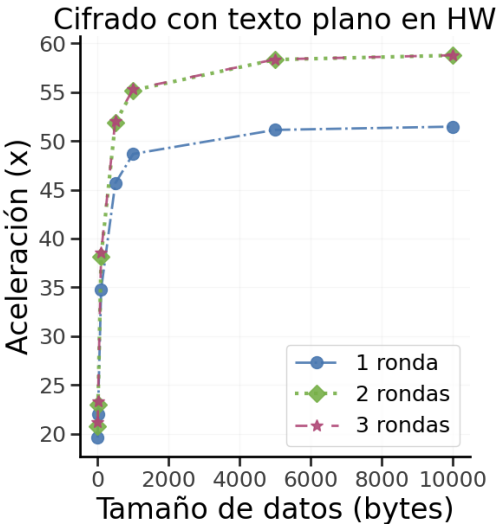


Figura 5.1: Gráfica de comparación entre las aceleraciones del cifrado con texto plano.

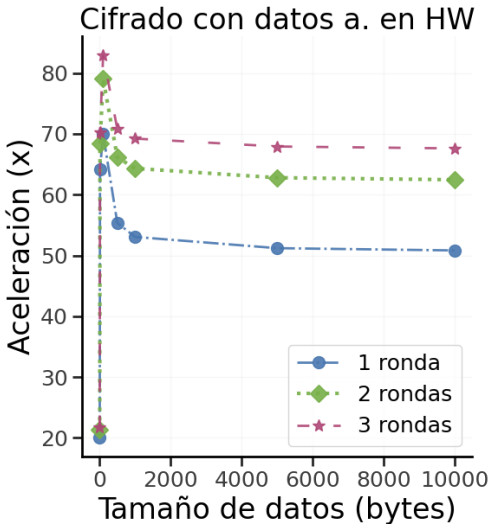


Figura 5.2: Gráfica de comparación entre las aceleraciones del cifrado con datos asociados.

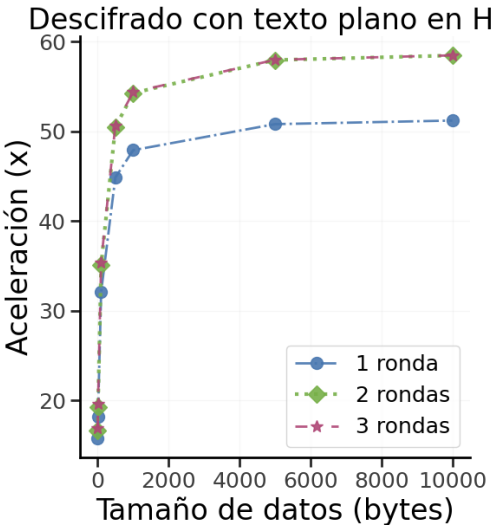


Figura 5.3: Gráfica de comparación entre las aceleraciones del descifrado con texto plano.

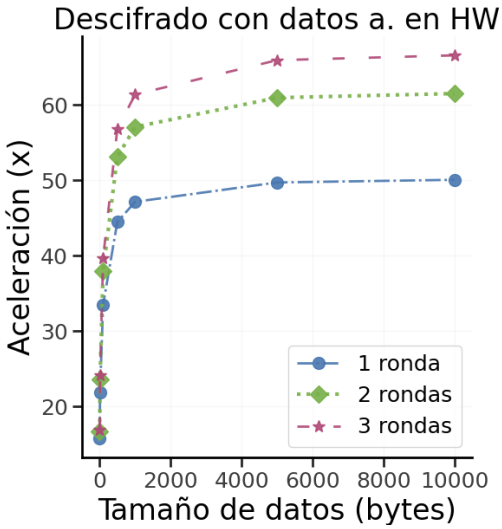


Figura 5.4: Gráfica de comparación entre las aceleraciones del descifrado con datos asociados.

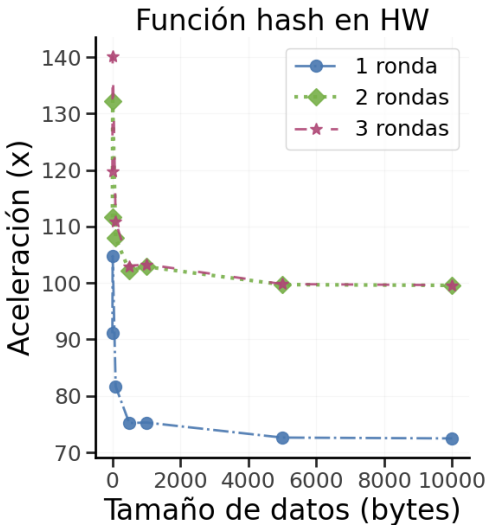


Figura 5.5: Gráfica de comparación entre las aceleraciones de la función hash.

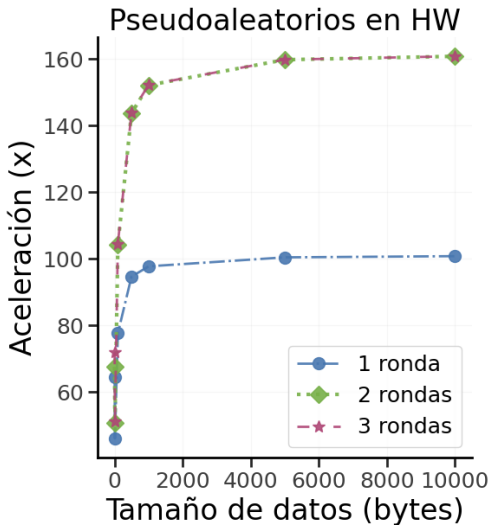


Figura 5.6: Gráfica de comparación entre las aceleraciones de la generación de número pseudoaleatorios.

Tabla 5.14: Comparación de las aceleraciones entre las tres versiones del hardware contra el software con optimización `Os` y `O2` con tamaño de datos de 10000 bytes.

		Cifrado		Descifrado			
		Texto plano	D. Asociados	Texto plano	D. Asociados	Hash	PRNG
ciclos	-Os	1215787	1210889	1214883	1210855	2393482	2388565
	-O2	1040565	1028092	1036192	1013598	2005924	2028176
	v1	20218	20226	20232	20242	27680	20115
	v2	17706	16461	17720	16477	20144	12609
	v3	17702	15206	17716	15222	20135	12608
aceleración (x)	-Os/v1	60.134	59.868	60.048	59.819	86.470	118.745
	-O2/v1	51.467	50.830	51.216	50.074	72.468	100.829
	-Os/v2	68.665	73.561	68.560	73.488	118.819	189.433
	-O2/v2	58.769	62.456	58.476	61.516	99.579	160.851
	-Os/v3	68.681	79.632	68.575	79.546	118.872	189.448
	-O2/v3	58.782	67.611	58.489	66.588	99.624	160.864

La Tabla 5.14 muestra la comparación de la aceleración obtenida con las tres versiones del hardware con el software con las banderas de optimización `-O2` y `-Os`. En esta ocasión se utilizó el máximo de bytes para comprar los datos.

Finalmente, la Tabla 5.15 muestra el tamaño en bytes de las funciones en memoria y la cantidad de instrucciones que en ensamblador que forman cada función. También se muestra la relación de bytes por cada instrucción. `Round` corresponde a una ronda de la permutación, `P6` realiza seis rondas de la permutación y `P12` realiza doce rondas de la permutación, en el caso de la optimización `-O2` solo se utiliza en la

Tabla 5.15: Comparación del tamaño de memoria y cantidad de instrucciones necesarias entre hardware y software.

		Round	P6	P12	Cifrado	Descifrado	Hash	Semilla	PRNG
bytes	-Os	526	60	108	700	678	298	50	60
	-O2	622	-	66	10098	7532	10164	112	6162
	HW	-	-	-	94	108	38	18	20
ins	-Os	154	24	142	259	248	117	28	26
	-O2	186	-	42	2817	2104	2752	103	1653
	HW	-	-	-	29	33	13	5	7
$\frac{bytes}{ins}$	-Os	3.416	2.500	0.761	2.703	2.734	2.547	1.786	2.308
	-O2	3.344	-	1.571	3.585	3.580	3.693	1.087	3.728
	HW	-	-	-	3.241	3.273	2.923	3.600	2.857

en las funciones hash y rand.

5.2. Utilización de recursos

Tabla 5.16: Características, desempeño y utilización de implementaciones de los algoritmos Ascon en Artix-7 .

Implementación	Rondas	Cifrado Datos A. CPB	Cifrado Texto P. CPB	Hash CPB	LUT's	FF's	Slices	[MHz]
ASCON_Graz-v1	1	1.000	1.000	1.750	1465	666	396	191
ASCON_Graz-v3	2	0.625	0.625	1.000	2142	665	582	201
ASCON_Graz-v5	3	0.500	0.500	0.750	2797	666	785	150
ASCON_VT-v1	1	1.250	1.250	NA	2410	539	518	233
ASCON_VT-v2	1	1.250	1.125	1.875	1790	544	515	219
ASCON_GMU2-v1h	1	0.875	0.875	1.625	1375	862	436	276
ASCON_GMU2-v2h	2	0.500	0.500	0.875	2126	861	632	234
ASCON_GMU2-v3h	3	0.375	0.375	0.625	2493	860	689	142
LWCP-V_1	**1**	**0.875**	**0.875**	**1.625**	**1326**	**528**	**374**	**190**
LWCP-V_2	**2**	**0.500**	**0.500**	**0.875**	**1784**	**526**	**502**	**119**
LWCP-V_3	**3**	**0.375**	**0.375**	**0.625**	**2080**	**526**	**558**	**82**

En la Tabla 5.16 se muestra la comparación de implementaciones de Ascon128 y Ascon-Hash en un FPGA Artix7 con 1, 2 y 3 rondas por ciclo, los datos provienen de [34]. También se muestran los ciclos por bytes del cifrado (A, datos asociados y P, texto plano), ciclos por bytes del hash. La tabla incluye los datos de utilización y la frecuencia máxima a la cual se puede operar el hardware diseñado. En la parte inferior se muestra la comparación de LWCP-V con sus tres versiones, en este caso para la comparación solo se implementaron los algoritmos de cifrado y hash. Cabe recalcar que se tienen interfaces distintas para operar el criptoprocesador, LWCP-V

tiene la interfaz RoCC descrita por Rocket-Chip y las otras implementaciones tiene la interfaz LWC API descrita en [35].

La comparación en utilización del softcore E310 por defecto contra el softcore con el criptoprocesador con sus tres versiones se muestra en la Tabla 5.17. Se muestra la jerarquía de los componentes con sangrías. Como componente principal se encuentra el SoC, todos los demás están dentro de un Tile de Rocket-Chip. Dentro del componente RoCC se encuentra el criptoprocesador. El porcentaje de aumento es realizado con la utilización del SoC por defecto contra las tres versiones.

Tabla 5.18: Comparación de la utilización de la propuesta contra los criptoprocesadores del estado del arte.

	TEE HW	FAC-V		DITES CC	RECO-HCON	**LWCP-V_v1**	**LWCP-V_v2**	**LWCP-V_v1**
FPGA	Virtex-7	Artix-7		Kintex-7	Artix-7	Artix-7		
	AES-128/256 SHA3-512	AES-128	AES-256	AES-128/256 SHA1	Ascon-128/a Ascon-Hash/a	Ascon-128 Ascon-Hash	Ascon-PRNG	
LUT total	11855	4086	4923	4055	1548	2862	3308	3755
FF's	5685	2169	2818	2891	1045	2364	2365	2370

La Tabla 5.14 Muestra la comparación de utilización de los criptoprocesadores del estado del arte contra las tres versiones de la propuesta. La propuesta con una menor utilización de los algoritmos estándares DITES CC es un 7.98 % mayor en LUT's y un 21.98 % en FF's que la versión tres de LWCP-V. Los datos de la Tabla 5.14 reflejan los algoritmos de cifrado y hash, sin incluir la parte de llave pública. En comparación la versión uno del criptoprocesador es 84.88 % más grande en LUT's y 126.22 % en FF's más grande que el criptoprocesador RECO-HCON. Sin embargo, la propuesta soporta modos extras y contiene un registro de estados adicional para guardar los bits del PRNG.

5.3. Pruebas estadísticas para el PRNG

Con el fin de comprobar la calidad de los bits aleatorios generados por la permutación Ascon en su modo de generador de número aleatorios se realizaron las pruebas NIST-STS [36] y Dieharder [37]. El conjunto de pruebas del NIST-STS[1] consiste en 15 algoritmos estadísticos que tienen como un valor de aceptación al *valor-p*. El proceso de evaluación es el siguiente: Se inicia con la generación de bits aleatorios provenientes de un generador, se agrupan en secuencias de longitud n. Cada secuencia se evalúa y arroja un o más *p-valores* que se encuentran en el rango $[0, 1]$. Los *p-valores* confirman o rechazan la hipótesis nula, que la secuencia proviene de una distribución uniforme. Para pasar las pruebas se usa como valor critico que se encuentra en las colas de la distribución, en el 99 %. Si el valor cae en el 0.01 % se rechaza la hipótesis nula. Se determina el nivel de significancía $\alpha = 0.001$, es decir, que una de 1000 secuencias se

[1]Descarga en: https://csrc.nist.gov/CSRC/media/Projects/Random-Bit-Generation/documents/sts-2_1_2.zip

Tabla 5.17: Utilización de los recursos del FPGA.

Softcore E310 por defecto								
Modulo	LUT totales	LUT lógicas	LUTRAM	SRL	FF	RAMB36	RAMB18	DSP
SoC	15040	14394	586	60	9873	8	2	2
Rocket	2558	2514	44	0	1351	0	0	2
Dcache	610	610	0	0	274	4	0	0
Icache	1262	1261	0	0	389	4	2	0
Softcore E310 con LWCP-V_v1								
Modulo	LUT totales	LUT lógicas	LUTRAM	SRL	FF	RAMB36	RAMB18	DSP
SoC	18379	17647	672	60	12772	12	2	2
Rocket	2601	2557	44	0	1405	0	0	0
Dcache	632	632	0	0	280	4	0	0
Icache	1577	1577	0	0	651	8	2	0
RoCC	2862	2862	0	0	2364	0	0	0
LWCP-V	2535	2535	0	0	1341	0	0	0
AsconCtrl	1661	1661	0	0	126	0	0	0
Ascon-p	237	237	0	0	328	0	0	0
CtrlPer	65	65	0	0	320	0	0	0
Permutation	172	172	0	0	8	0	0	0
RandCtrl	180	180	0	0	74	0	0	0
StateReg	96	96	0	0	384	0	0	0
Trivium	368	368	0	0	364	0	0	0
Aumento %	22.20	22.60	14.68	0	29.36	50	0	0
Softcore E310 con LWCP-V_v2								
Modulo	LUT totales	LUT lógicas	LUTRAM	SRL	FF	RAMB36	RAMB18	DSP
SoC	18825	18093	672	60	12773	12	2	2
Rocket	2600	2556	44	0	1405	0	0	2
Dcache	632	632	0	0	280	4	0	0
Icache	1579	1579	0	0	651	8	2	0
RoCC	3308	3308	0	0	2365	0	0	0
LWCP-V	2981	2981	0	0	1342	0	0	0
AsconCtrl	1538	1538	0	0	126	0	0	0
Ascon-p	822	822	0	0	329	0	0	0
CtrlPer	65	65	0	0	320	0	0	0
Permutation	757	757	0	0	9	0	0	0
RandCtrl	182	182	0	0	74	0	0	0
StateReg	96	96	0	0	384	0	0	0
Trivium	368	368	0	0	364	0	0	0
Aumento %	25.17	25.70	14.68	0	29.37	50	0	0
Softcore E310 con LWCP-V_v3								
Modulo	LUT totales	LUT lógicas	LUTRAM	SRL	FF	RAMB36	RAMB18	DSP
SoC	19271	18539	672	60	12784	12	2	2
Rocket	2623	2579	44	0	1411	0	0	0
Dcache	615	615	0	0	280	4	0	0
Icache	1260	1260	0	0	651	8	2	0
RoCC	3755	3755	0	0	2370	0	0	0
LWCP-V	2685	2685	0	0	1341	0	0	0
AsconCtrl	1714	1714	0	0	126	0	0	0
Ascon-p	405	405	0	0	328	0	0	0
CtrlPer	65	65	0	0	320	0	0	0
Permutation	340	340	0	0	8	0	0	0
RandCtrl	164	164	0	0	74	0	0	0
StateReg	96	96	0	0	384	0	0	0
Trivium	368	368	0	0	364	0	0	0
Aumento %	28.13	28.80	14.68	0	29.48	50	0	0

rechaza. Las 15 pruebas son las siguientes [36]:

- *Frecuencia de monobit*: Con esta prueba se mide la proporción de ceros y unos. Se espera que la secuencia tenga aproximadamente la misma proporción.
- *Frecuencia dentro de un bloque*: Similar a la prueba anterior en esta prueba se particiona la secuencia en bloques y se mide la proporción de ceros y unos en cada bloque.
- *Rachas*: Esta prueba se centra en las rachas de ceros y unos en una secuencia. Donde una racha es una secuencia ininterrumpida de ceros o unos. Se determina si estas secuencias de puros unos de distintos tamaños son esperadas de una secuencia aleatoria. El procedimiento Prueba de Rachas contrasta si es aleatorio el orden de aparición de dos valores de una variable. Una muestra con un número excesivamente grande o excesivamente pequeño de rachas sugiere que la muestra no es aleatoria. En otras palabras, mide la oscilación entre secuencias de unos y ceros.
- *Racha más larga de unos en un bloque*: Su objetivo consiste en analizar si la racha más larga de unos es esperada en una secuencia aleatoria.
- *Rango de matriz binaria aleatoria*: El propósito de la prueba es evaluar la dependencia linear de subsecuencias de tamaño fijo.
- *Transformada de Fourier discreta (spectral)*: Con esta prueba se detectan los patrones repetitivos cercanos.
- *Coincidencia de plantillas no superpuestas (aperiódicas)*: Se enfoca en las coincidencias de subsecuencias establecidas. Rechaza a las secuencias que tienen un alto valor de coincidencias aperiódicas de un cierto patrón.
- *Coincidencia de plantillas superpuestas (periódicas)*: Rechaza a las secuencias que muestran un patrón diferente de rachas de unos esperadas.
- *Prueba estadística universal de Maurer*: Prueba la cantidad de bits que se encuentren entre dos patrones. Esta subsecuencia de bits es evaluada si puede ser comprimida sin pérdida de información.
- *Complejidad lineal*: Determina si esta secuencia es lo suficientemente compleja. Si es aleatoria entonces puede ser caracterizada con un registro retroalimentado largo.
- *Serial*: Mide la frecuencia de superposición de cada patrón de m bits en toda la secuencia. Determina las ocurrencias de patrones de $2m$ bits.
- *Entropía aproximada*: Prueba la frecuencia de cada patrón de m bits. Se compara la frecuencia de bloques superpuestos de tamaños consecutivos (m, $m+1$).
- *Suma acumulativa*: Se modela una caminata aleatoria con los bits de la secuencia siendo (cero = -1, uno = $+1$). La suma total tiene que estar cercana a cero.
- *Excursiones aleatorias*: Cuenta el número de ciclos en la suma acumulativa. Se usan sumas parciales de la suma acumulativa y se determina las veces que se pasa a un estado (por ejemplo, el estado -4).
- *Excursiones aleatorias variante*: Determina las veces que un estado ocurre en

la caminata aleatoria de la suma acumulativa.

Tabla 5.19: Resultados de las pruebas de NIST-STS

Nombre de la prueba	valor p	Proporción	Resultado
Frecuencia Monobit	0.039587	989/1000	aprobada
Frecuencia dentro de un bloque	0.189625	988/1000	aprobada
Rachas	0.278461	990/1000	aprobada
Racha más larga de unos	0.508172	986/1000	aprobada
Rango de matriz binaria	0.128132	995/1000	aprobada
FFT	0.055010	988/1000	aprobada
Plantillas no superpuestas	0.065230	983/1000	aprobada
Plantillas superpuestas	0.049984	985/1000	aprobada
Prueba de Maurer	0.851383	991/1000	aprobada
Complejidad lineal	0.446556	991/1000	aprobada
Serial	0.382115	990/1000	aprobada
Entropía aproximada	0.465415	988/1000	aprobada
Suma acumulativa	0.834308	987/1000	aprobada
Excursiones aleatorias	0.837157	605/613	aprobada
Excursiones aleatorias variante	0.034234	603/613	aprobada

Nota: La prueba se realizó con los parámetros de tamaño en bits de bloque por defecto: Frecuencia dentro de un bloque = 128, Plantillas no superpuestas = 9, Plantillas superpuestas = 9, Entropía aproximada = 10, Serial = 16 y Complejidad lineal = 500. El reporte que arroja la prueba indica que la tasa mínima para aprobar es 980 para 1000 secuencias. En el caso de las excursiones es 599 para 613 secuencias.

Las pruebas Dieharder[2] tiene un propósito igual que las NIST-STS que es verificar si la secuencia es verdaderamente aleatoria e independiente. Las pruebas de este conjunto pueden evaluar el desbalanceo de números, correlaciones de corta y larga distancia, distribución en hiperplanos, patrones complejos, entre otros. El conjunto consiste originalmente de 16 pruebas que se muestran a continuación:

- *Separación de cumpleaños*: Se escogen puntos aleatorios en un intervalo grande. Los espacios entre estos dos tienen que tener una distribución de Poisson. La prueba está basada en la paradoja del cumpleaños.
- *Cinco permutaciones superpuestas*: En esta prueba se analizan cinco números aleatorios consecutivos en una secuencia grande, alrededor de millones de bits. Se analiza que las 120 permutaciones son igualmente probables.
- *Rango binario de matrices 32 x 32*: Se forma una matriz de 32 números de 32 bits. Se espera que el rango de la matriz sea mayor a 29.
- *Rango binario de matrices 6 x 8*: Se escogen seis números de 32 bits, se especifica un byte y se obtiene el rango. Se espera que sea mayor a cuatro.

[2]Descarga e instalación: `$ sudo apt-get install dieharder`

Tabla 5.20: Resultados de las pruebas Diehard.

Nombre de la prueba	valor p	Resultado
Separación de cumpleaños	0.0333198	aprobada
Cinco permutaciones superpuestas	0.0818873	aprobada
Rango binario de matrices 32 x 32	0.4620313	aprobada
Rango binario de matrices 6 x 8	0.3542130	aprobada
Flujo de bits	0.8229718	aprobada
OPSO	0.0318645	aprobada
OQSO	0.0000069	débil
ADN	0.8991851	aprobada
Conteo de unos en flujo	0.7340823	aprobada
Conteo de unos en bytes	0.7932477	aprobada
Estacionamiento	0.3064419	aprobada
Distancia mínima	0.9959555	débil
Esfera 3D	0.5588227	aprobada
Compresión	0.0252532	aprobada
Sumas	0.2223030	aprobada
Rachas	0.7125542	aprobada
Dados	0.1711485	aprobada

Nota: La prueba se realizó con la ejecución `dieharder -g 201 -f rand.txt -a`. Como parámetro para determinar si la prueba se aprobó se utiliza el p-valor. Si está fuera del intervalo de confianza $(0.05, 0.95)$, entonces el resultado no es fiable.

- *Secuencia de bits*: Se trata a la secuencia de bits como palabra y se cuentan las palabras que se superponen en toda la secuencia.
- *Superposición de pares con ocupación dispersa OPSO*: Considera dos letras de un alfabeto de 1024 letras. Se toman dos letras de 10 bits de números de 32 bits en la secuencia. Cuenta las letras faltantes, tienen que estar normalmente distribuidas.
- *Superposición cuádruple con ocupación dispersa OQSO*: Es similar a la prueba anterior solo que se toman cuatro letras de un alfabeto de 32.
- *ADN*: En este caso se toman a un alfabeto de cuatro letras (A, C, G, y T). Se forman palabras de 10 letras.
- *Conteo de unos en secuencia*: Cuenta la cantidad de unos convirtiendo a palabras de cinco letras.
- *Conteo de unos en bytes*: Similar al anterior con la diferencia que en esta se extraen bytes de los números de 32 bits.
- *Estacionamiento*: Se establece un cuadrado de 100 x 100, se colocan círculos de diámetro 1. Si se sobreponen, se intenta colocar en otro lado aleatorio. Se cuenta el número de superposiciones.
- *Distancia mínima 2D*: Se eligen 8000 puntos aleatorios en un cubo de 1000 x 1000 x 1000. Se mide la distancia entre dos pares. La distribución debe aproxi-

marse a una exponencial.

- *Esfera 3D*: Para esta pruebas se escogen 4000 puntos. en cada punto se centra una esfera hasta alcanza el punto más cercano. Se espera una distribución exponencial.
- *Compresión*: Se transforman los enteros a un valor en $[0, 1)$ U Y se cuenta el número de veces para reducir el número a 1 usando $k = \lceil k * U \rceil$ iniciando con $k = 2^{31}$.
- *Sumas*: Se generan secuencias de flotantes entre $(0, 1)$ y se suman 100 flotantes consecutivos. Deben de estar normalmente distribuidos.
- *Rachas*: Se generan flotantes entre 0 y 1, se cuentan las rachas ascendentes y descendentes.
- *Dados*: Se juegan 200000 juegos de dados se cuenta la tasa de éxitos contra los datos lanzados y se analiza la distribución.

Los dos conjuntos de pruebas utilizaron los mismos datos. Para generar la secuencia de números pseudoaleatorios se inició absorbiendo solo una semilla de 64 bits que corresponde a la secuencia $\mathbf{1234567887654321_{16}}$, posteriormente todas las secuencias extrayendo 64 bits en cada serie de rondas, no hubo resemillado. La prueba NIST-STS necesita como mínimo 1000 cadenas de 1000000 de bits, en su totalidad equivale a 125 MB de bits pseudoaleatorios. El formato del archivo que contiene los bits es un archivo binario, por lo tanto, en C se imprimen como `%c`, sin saltos de línea ni espacios. Estos mismos datos se introdujeron a las pruebas Dieharder. Los resultados de las pruebas NIST-STS se muestran en la Tabla 5.19 y los resultados de las pruebas Dieharder están en la Tabla 5.20.

5.4. Administración de llaves

Las Figuras 5.7 y 5.8 muestran el uso de la generación de números pseudoaleatorios para una nueva aplicación que es la generación de llaves simétricas. En estas imágenes se representa a los componentes con sus señales y datos más importantes para esta aplicación. Se inicia con el estado guardado de anteriores llamados al PRNG. Para romper la secuencia de generación de números aleatorios se absorbe una nueva semilla `S0 seed0`. Posterior a la absorción de la semilla, el resultado de la doceava ronda de la fila cero del estado `R12`, en color naranja, se almacena en los registros de la parte alta de las llaves. Tras la siguiente serie de doce rondas `R12`, en verde, es almacenado como la parte baja de las llaves (ver la figura 5.8). En este ejemplo se utilizó a las llaves con el `id = 2`. Nuevamente para romper la secuencia se absorbe una nueva semilla `S0 seed1` y la generación de llaves termina. En todo el proceso de generación se mantiene al control principal `AsconCtrl` inactivo.

Con un correcto direccionamiento del flujo de datos, es posible dar soporte al enmascaramiento de las llaves. Las Figuras 5.9 y 5.10 muestran el proceso para realizar

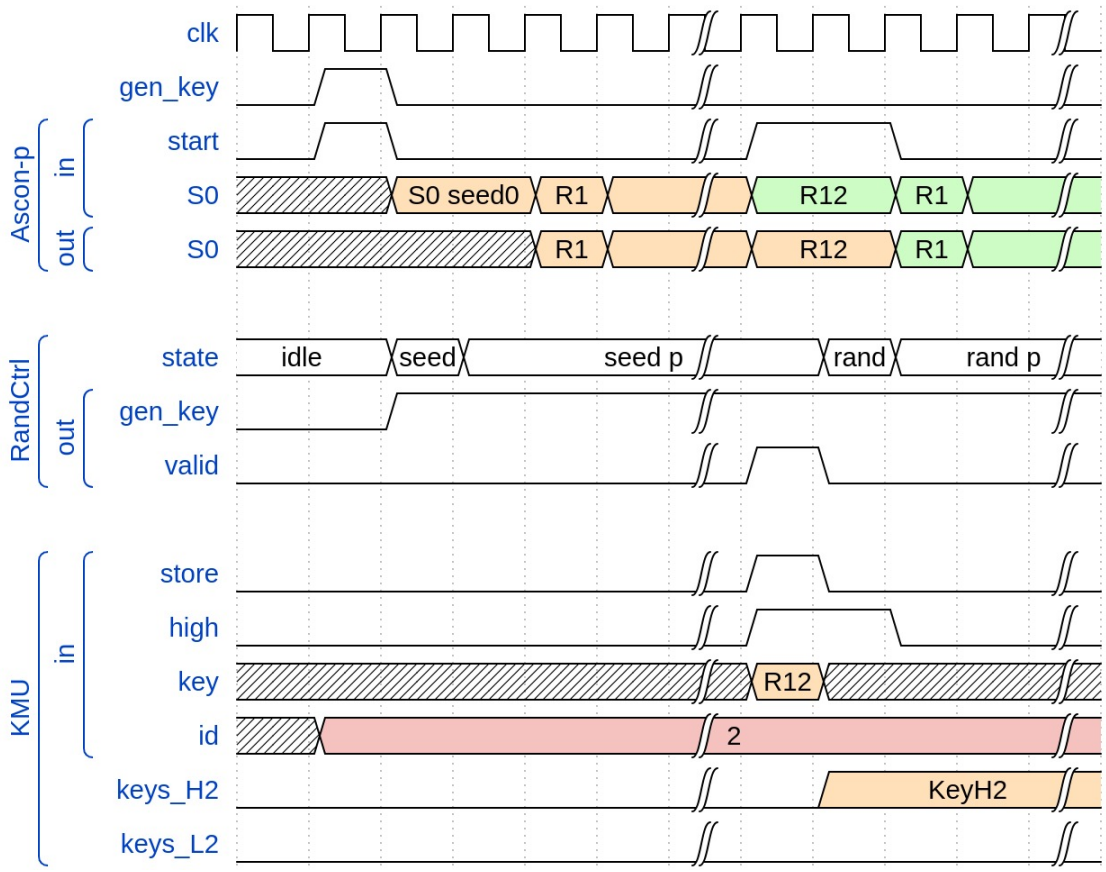


Figura 5.7: Cronograma de la generación de llaves simétricas.

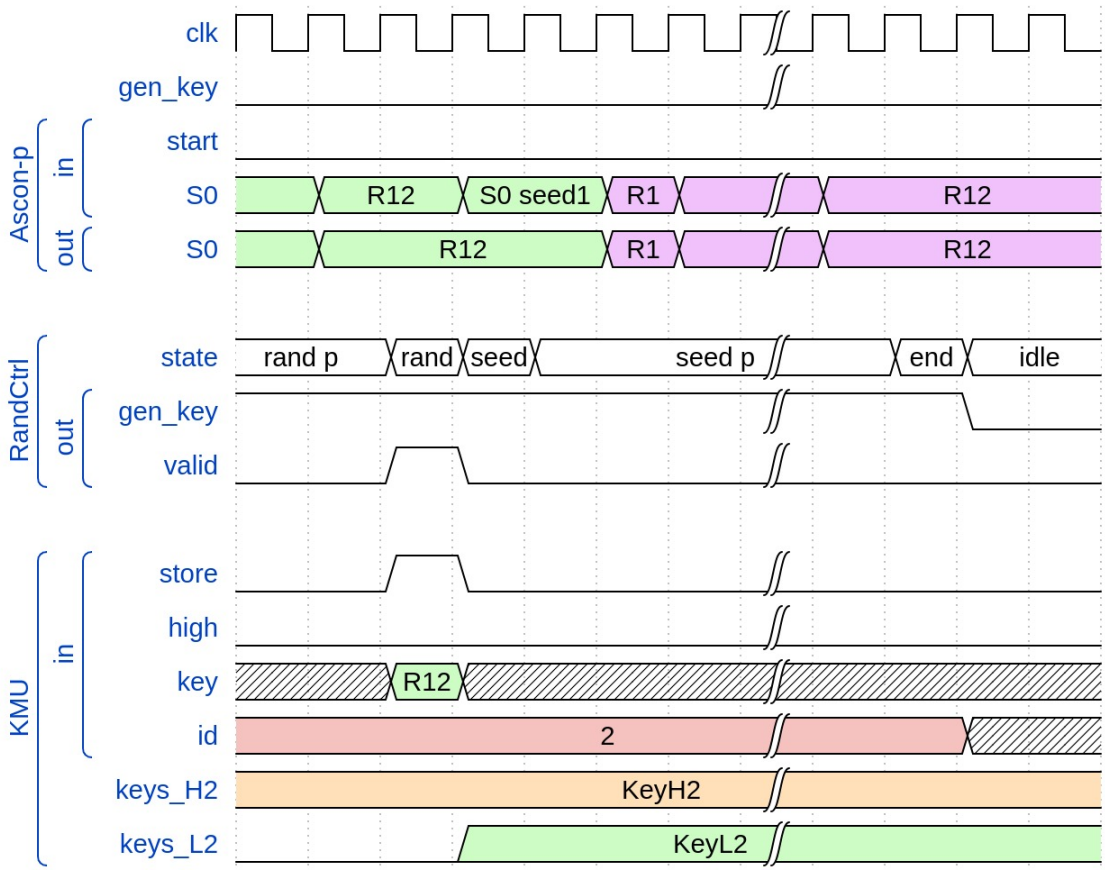


Figura 5.8: Cronograma de la generación de llaves simétricas continuación.

el enmascaramiento de llaves. Para ello es necesario contar con una llave maestra `MK` dentro del criptoprocesador. La llave maestra es cargada en las filas uno y dos del estado. En este caso se considera que previamente se han generado llaves y se encuentran en los registros de `KMU`. Con el `id` se indica que llave se quiere cifrar, entre el flujo de componentes esta llave se divide en su parte alta y baja. La parte alta es la primera en ser absorbida `R12 KH2` y cifrada `R12 KH2w`. Cuando ya está cifrada entonces, `c_valid` indica que puede ser escrita a memoria la primera parte de la llave. La parte baja de la llave `KL2` se procesa y escribe de forma similar, identificada con el rojo. La finalización es realizada con la llave maestra para producir la etiqueta de autenticación.

De manera similar se puede subir las llaves cifradas al criptoprocesador para reintegrarse a `KMU`. Para esta opción se utiliza el modo descifrado con la misma llave maestra. El proceso es representado en las Figuras 5.11 y 5.12. Es similar a la generación de llaves en el almacenamiento de estas. Cuando el descifrado es válido entonces entran de nuevo a la `KMU`. Hay que tomar en cuenta que la señal `c_valid` se encuentra abajo en todo momento para evitar escribir a la memoria la llave sin mascara. De esta forma se restablecen las llaves guardadas en archivos externos cuando el dispositivo se encuentra sin alimentación.

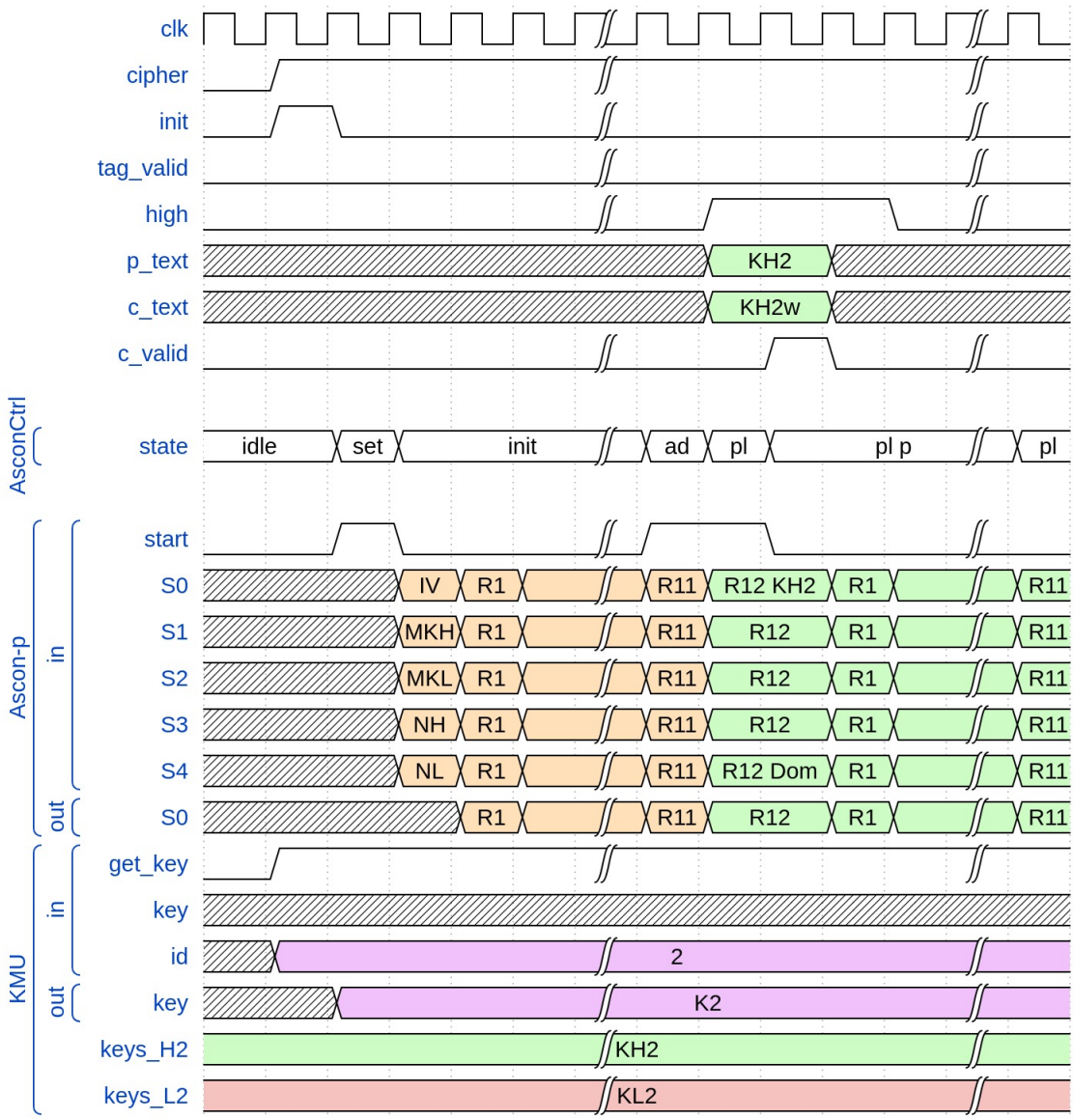


Figura 5.9: Cronograma del enmascaramiento de llaves.

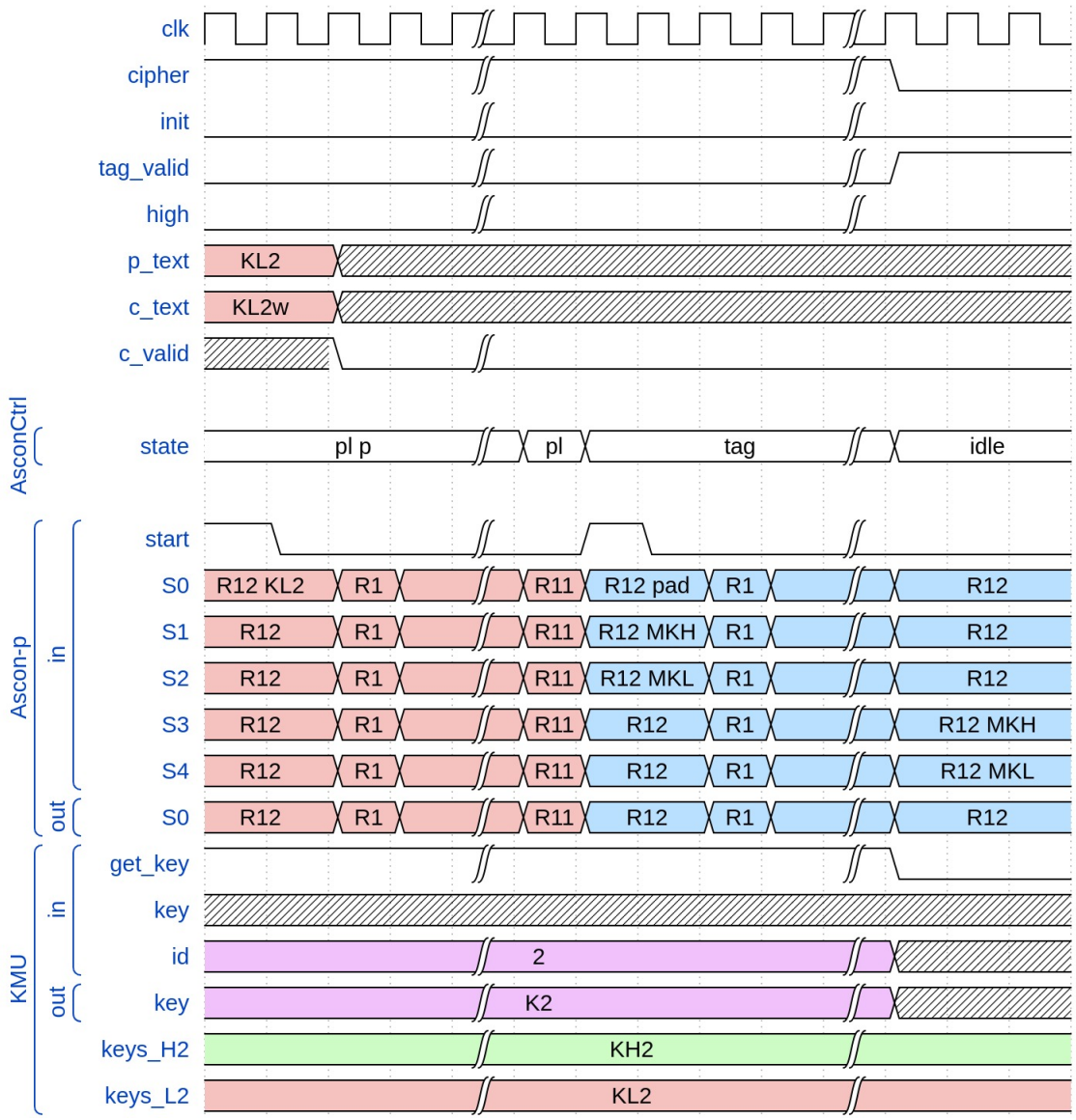


Figura 5.10: Cronograma del enmascaramiento de llaves continuación.

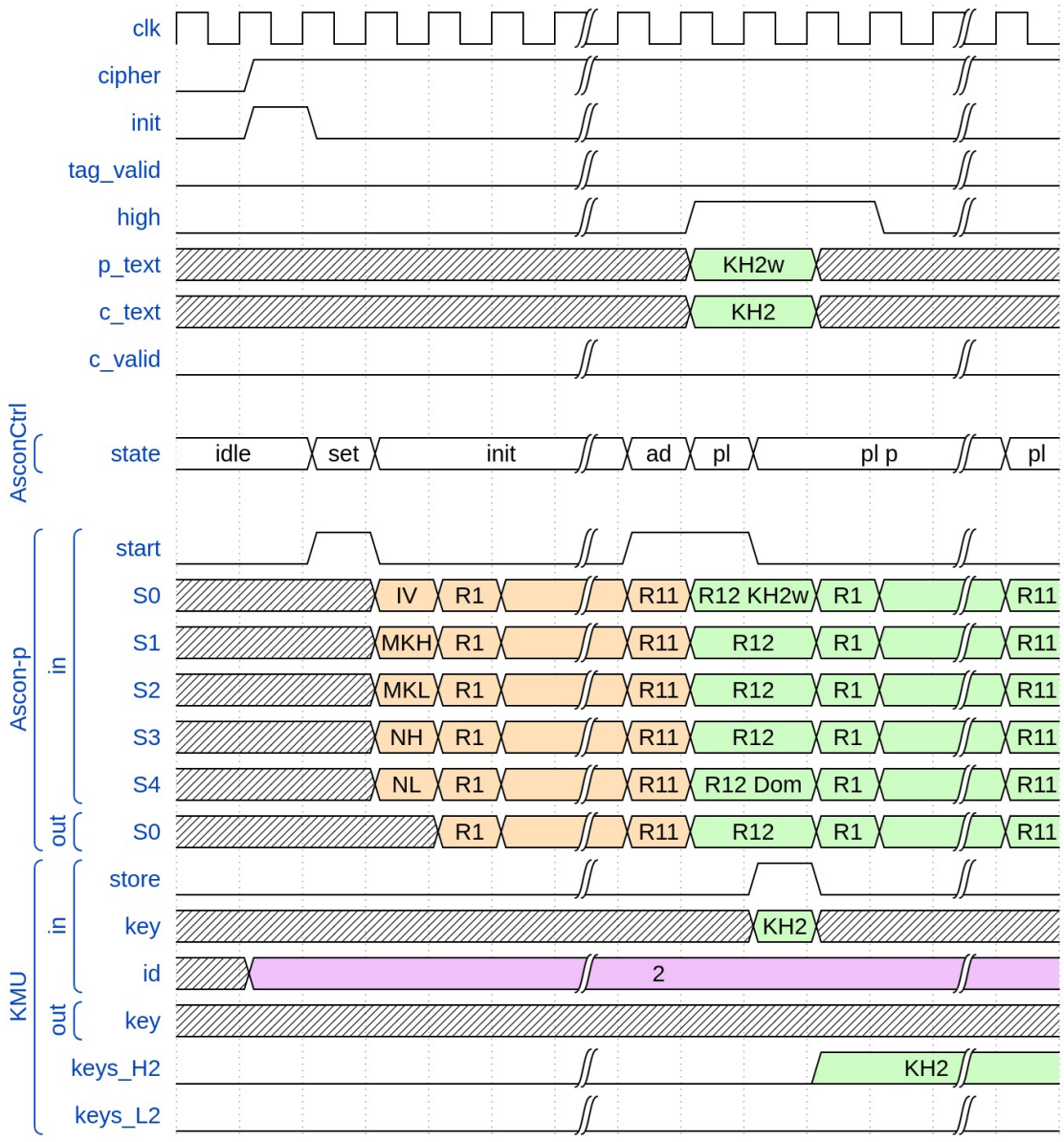


Figura 5.11: Cronograma del desenmascaramiento de llaves.

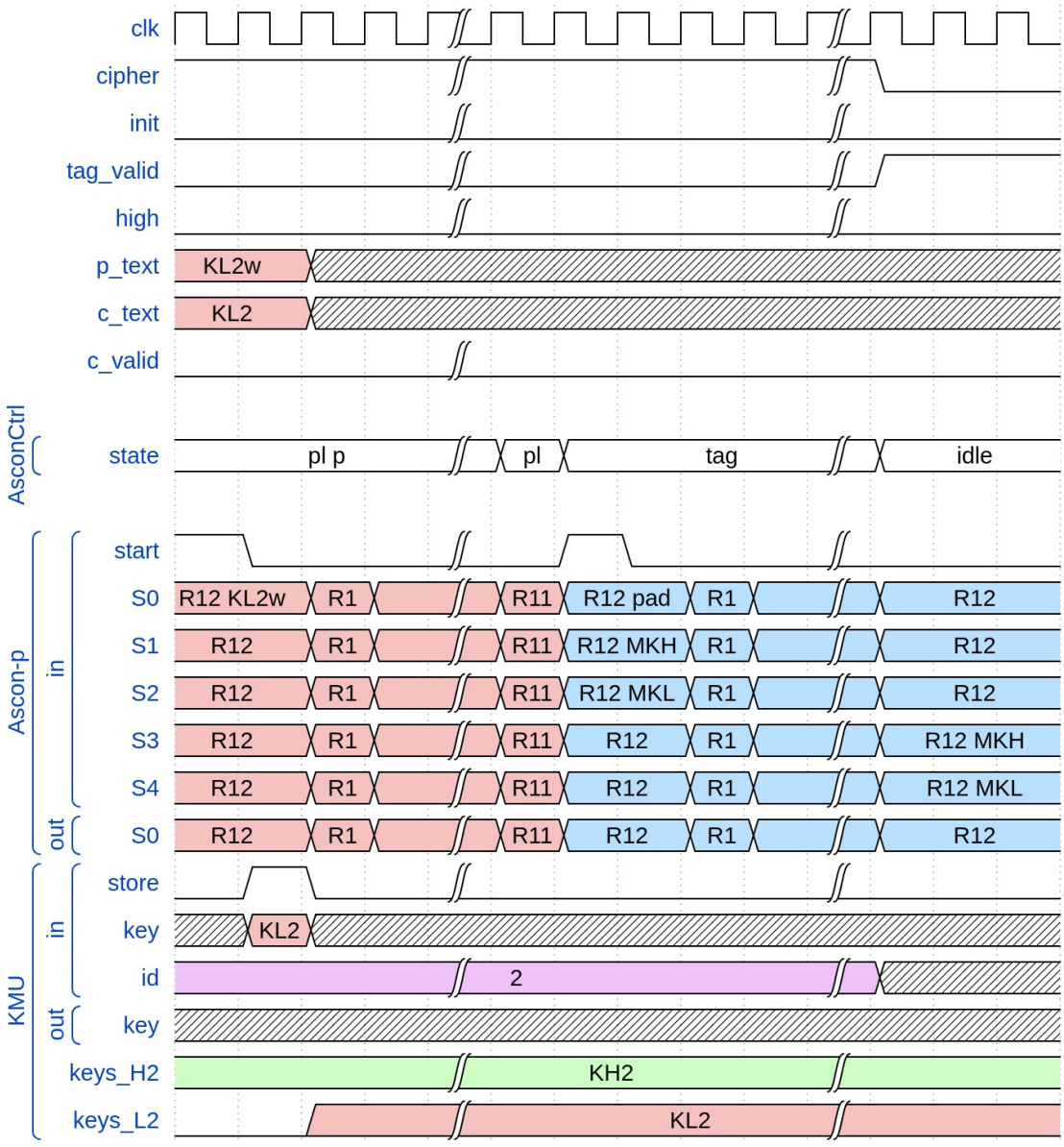


Figura 5.12: Cronograma del desenmascaramiento de llaves continuación.

Capítulo 6

Conclusiones

Se decidió llevar a cabo el diseño del hardware en torno al algoritmo Ascon por estar en proceso de estandarización y ser una función esponja. De esta forma se podían realizar distintos algoritmos de criptografía con una sola permutación. Para lograrlo se efectuó un manejo en el flujo de datos hacia la permutación mediante registros, multiplexores, condiciones y controles. Con el criptoprocesador diseñado es posible realizar seis distintos algoritmos : Cifrado autenticado, código de autenticación de mensajes basado en hash (HMAC), cifrado autenticado con datos asociados, generación de número pseudoaleatorios, generación de llaves simétricas y enmascaramiento de llaves. En el caso del cifrado autenticado es el AEAD sin los datos asociados y el HMAC es el mismo AEAD sin cifrar datos. La generación de llaves reutiliza al PRNG y el enmascaramiento de llaves usa a Ascon-128 sin absorber los datos asociados.

Tanto el PRNG, la generación de llaves simétricas y el enmascaramientos de llaves son propuestas. El NIST [38] define las especificaciones de tres tipos de generadores de bits aleatorios, aquellos basados en funciones hash, basados en funciones HMAC y cifradores por bloque en modo contador. La propuesta queda dentro de las funciones hash, dentro de esta clasificación se encuentran las funciones SHA-1 y SHA-2. Los resultados de las pruebas estadísticas muestran que Ascon en su modo resemillable tiene potencial para ser un PRNG. Sin embargo, no muestra que sea seguro utilizarlo, para ello debe llevarse acabo un proceso de validación para aprobar su uso como un PRNG.

Para el enmascaramiento de las llaves también existen las recomendaciones del NIST, donde se define AES en modo KW (de enmascaramiento de llaves) y AES en modo KWP (enmascaramiento de llaves con relleno) [39]. En pocos términos es una red de Feistel de cinco rondas con AES como núcleo, puede cifrar una gran cantidad de bloques y realizar autenticación. Para la generación de llaves el NIST menciona que la generación directa de llaves simétricas debe realizarse con los generadores de bits aleatorios aprobados [40]. Debido a esto se debe realizar un análisis detallado del uso de Ascon para dar soporte a estos dos algoritmos.

La intención de diseñar una unidad administradora de llaves fue mejorar la seguridad de éstas al realizar un direccionamiento indirecto de ellas. Cuando bajan a memoria están cifradas y pueden descifrarse dentro del criptoprocesador sin revelar información. La seguridad de las llaves es producto de la secrecía de la llave maestra. En una aplicación el usuario no podría acceder a ella. Adicionalmente se dejó habilitada la opción de cargar las llaves desde memoria sin la necesidad de enmascararlas.

El instanciar el criptoprocesador como un coprocesador de Rocket-Chip facilitó la programación y el análisis de desempeño, no obstante, la falta de documentación sólida dificultó la comunicación con la memoria caché de nivel uno. Especialmente por el uso de las etiquetas para referenciar un dato que no tiene relación con la dirección de memoria ni la asociatividad.

El diseño del control de acceso a memoria junto a la decodificación de las instrucciones y los registros ayudaron a la gestión del flujo de datos entre el procesador y el criptoprocesador. Al establecer el control de escritura y lectura separado del criptoprocesador se obtuvo un diseño modular, facilitando la depuración del criptoprocesador mediante. El criptoprocesador es agnóstico de las solicitudes de escritura y lectura así como el formato de los bloques a palabras, solo necesita los bloques válidos y que los registros tengan los datos necesarios e invariantes durante el procesamiento. Los tres bits más significados simplificaron el control de los modos de operación ya que hacen referencia a un conjunto de un modo en especifico.

El uso de la misma ISA para controlar al criptoprocesador abstrae su uso a nivel software. El programador solo debe indicar los parámetros de entrada como tamaños de mensajes y apuntadores. Hay instrucciones utilizadas en otros modos, como el caso del descifrado que utiliza todas las instrucciones del cifrado sin la inicialización de esta, el cifrado de llaves utiliza la instrucción para indicar el retorno de la llave cifrada y su descifrado utiliza la instrucción para subir la etiqueta de autenticación de esta forma se verifica si la llave a sido modificada. Si las etiquetas no son iguales entonces se utiliza la eliminación de dicha llave. Al utilizar directamente las instrucciones formándolas parte por parte facilitó la programación al no requerir modificar el compilador para dar soporte a las nuevas instrucciones. Aun se tiene espacio disponible de codificación para nuevas instrucciones que controlen nuevos modos de operación o otros módulos dentro del criptoprocesador, ya que se utilizaron 17 instrucciones de las 2^7 posibles.

Los resultados de las ejecuciones con las tres propuestas de la permutación muestran que son capaces de procesar un byte cada dos ciclos cuando se tienen un tamaño de datos grandes, para el cifrado, descifrado y generación de número pseudoaleatorios. Se nota una mejora al usar las rondas desdobladas. Sin embargo, comparando los resultados de las implementaciones con dos o tres rondas desdobladas, no se observa una diferencia significativa. Esto se debe a la interfaz que encargada de cargar

y escribir datos a la memoria L1 de datos, ya que tarda más tiempo en leer y escribir que en realizar las rondas completas cada que se procesa un bloque.

La comparación de la aceleración con el software con optimización `O2` muestra una mejora arriba de 50x para los algoritmos de cifrado y descifrado. Cuando se necesitan un mayor número de rondas como es el caso de la función hash y la generación de números pseudoaleatorias, la mejora es más significativa, llegando a los 180x como es el caso de la comparación de ejecución con tres y dos rondas desdobladas contra la optimización en tamaño. Cuando el costo de procesar los datos supera al costo de la inicialización y finalización en los algoritmos, el valor de la aceleración llega a un valor constante. En la inicialización del cifrado y descifrado se deben cargar al criptoprocesador la llave y el número público seguidos de 12 rondas, mientras que la finalización toma 12 rondas y se debe regresar la etiqueta de autenticación. No obstante, el procesamiento de un bloque solo necesita seis rondas más la carga y la escritura de dicho bloque. Conforme aumentan los datos a procesar el costo de inicialización y finalización es insignificante.

Sin embargo, para función hash el comportamiento de la aceleración es inverso. A menor número de bytes se tienen una mejor aceleración. La causa se debe a la inicialización, como no se requiere de cargar ningún dato antes de inicializar, esta puede realizarse fácilmente en el hardware, al igual que producir los 256 bits del hash, ya que no existen tiempos de espera para cargar datos al algoritmo.

En el caso de cifrado con datos asociados hay un aumento en la aceleración cuando el tamaño de los datos a procesar es pequeño, más no cero. Esto se debe a que el hardware ejecuta en menor tiempo las rondas. En el algoritmo siempre se hace una serie de rondas extras al finalizar la fase de absorción para cambiar de dominio. Conforme aumenta el tamaño de los datos esta diferencia incide mínimamente en el desempeño. Se pensaría que el descifrado con datos asociados tendría la misma tendencia, sin embargo, en este caso se debe cargar la etiqueta para compararla al final.

Al comparar la arquitectura ARM Cortex M7 con el RV32IMAC. Se puede observar que es ligeramente mejor en ARM. Es un tema del compilador y del diseño de la arquitectura. Dentro de las instrucciones de las rondas en ARM se usa el corrimiento opcional en las operaciones. Esta es una característica de las instrucciones de ARM que todas las instrucciones aritméticas pueden manipular uno de sus operandos aplicándoles un corrimiento antes de realizar la operación. Es un pequeño campo de cinco bits que permite realizar de 0 a 31 bits de corrimiento. Los corrimientos que se indican en el segundo operando son: los corrimientos lógicos a la derecha e izquierda, corrimiento aritmético a la derecha y rotación a la derecha. Esto le da una gran ventaja en la capa de difusión lineal de la permutación de Ascon donde se necesitan rotaciones de 64 bits, las cuales se compilan a corrimientos lógicos a la izquierda en ARM. Se ve reflejado en una reducción del número de instrucciones necesarias.

Se puede confirmar la optimización de tamaño del código. Existe una diferencia significativa en bytes y menor cantidad de instrucciones en la optimización `Os` que la optimización `O2`. De igual manera se puede apreciar el efecto del uso de la extensión `C` de comprimidas. El valor esperado de realizar la operación de número de bytes entre número de instrucciones es cuatro, es decir, que se necesitan cuatro bytes por instrucción, sin embargo el resultado es un número menor que cuatro, lo cual indica que existen las instrucciones comprimidas dentro del código compilado, esto es independiente de la bandera de optimización utilizada. En el caso de hardware, no influye la bandera de optimización al compilar ya que las instrucciones son creadas con ensamblador en linea y solo están encapsuladas entre dos instrucciones `FENCE`. Los bytes e instrucciones son menores que la compilación con `Os` del software. Igualmente puede observarse que la relación bytes entre instrucciones es menor a 4, debido a que las instrucciones `FENCE` son las que se compilan como instrucciones comprimidas.

El aumento del número de instrucciones y tamaño de bytes en la optimización O2 se debe a que las rondas son desdobladas. En el caso de la función hash es la única que hace uso de la función P12 al compilarla con O2, todos los demás algoritmos usan rondas desdobladas dentro de sus funciones.

Al comparar la utilización de LWCP-V, sin la interfaz RoCC ni la generación de pseudoaleatorios contra las implementaciones de Ascon junto a la función hash, muestra un menor número de LUTs, FFs y Slices. Cabe mencionar que estas implementaciones de Ascon tiene otra interfaz con lo cual las comparaciones no son justas. Puede observarse que las implementaciones de LWCP-V tiene una frecuencia máxima menor. Los resultados muestra en aumento de utilización de componente en la FPGA cada que se aumenta una ronda desdoblada. Como se quiere que estas rondas desdobladas de implementen en un ciclo, esto aumenta la frecuencia máxima ya que se tienen más compuertas en las rondas. Cuando se comparó con los criptoprocesadores que instancian los algoritmos estándares las tres versiones de LWCP-V fue menor.

Desde el punto de vista de la utilización en la comparación del SoC E310 por defecto contra los softcores con el criptoprocesador, se obtienen los resultado esperados en un aumento del 22 % En el uso de las LUTs totales y 29 % en los FFs. al usar la versión con solo una ronda por ciclo. Al agregar más rondas por ciclos, el único valor que aumenta es el porcentaje de las LUTs lógicas que se utilizan, debido a que solo se representan compuertas lógicas en las rondas. El aumento es de un 25 % para dos rondas y 28 % para la versión de tres rondas. Como la mejora en el desempeño no es significativa entre la versión de dos contra la de tres rondas, es recomendable utilizar la primera ya que ocupa menos recursos en hardware. Estos resultados no consideran a la KMU ya que el tamaño de esta depende de la cantidad de llaves que se deseen almacenar. Puede observarse la utilización de recursos del coprocesador es superior a todo el pipeline Rocket sin contar a las memorias de nivel uno, sin embargo inter-

namente el coprocesador cuenta con registros, controladores y las operaciones tienen una mayor longitud de bits.

6.1. Trabajo futuro

- Cambiar el algoritmo Trivium por un TRNG y evaluar la generación de números aleatorios con modo resemillable.
- Dar soporte a los algoritmos de Ascon con un tamaño de bloque de 128 bits.
- Agregar un algoritmo de llave pública.
- Realizar versiones protegidas de la permutación de Ascon.
- Adaptar el criptoprocesador para otras arquitecturas o como un módulo DMA.

Bibliografía

[1] L. Shannon, "Reconfigurable Computing Architectures," in *Reconfigurable Computing*, pp. 29–46, Elsevier, 2008.

[2] V. Desnitsky and I. Kotenko, "Expert Knowledge Based Design and Verification of Secure Systems with Embedded Devices," in *International Conference on Availability, Reliability, and Security*, pp. 194–210, Springer, 2014.

[3] W. Wang, B. Jungk, J. Wälde, S. Deng, N. Gupta, J. Szefer, and R. Niederhagen, "XMSS and Embedded Systems," in *International Conference on Selected Areas in Cryptography*, pp. 523–550, Springer, 2019.

[4] W. H. Hassan *et al.*, "Current Research on Internet of Things (IoT) Security: A Survey," *Computer networks*, vol. 148, pp. 283–294, 2019.

[5] R.-V. International, "History of RISC-V." https://riscv.org/about/history/, 2020. [accedido el 03 de Octubre de 2022].

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan kaufmann, 2nd ed., 2021.

[7] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213," 2019.

[8] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[9] B. A. Research, "Chipyard's Documentation," 2023.

[10] SiFive, "SiFive E3 Coreplex Series Manual, version 1.2," 2017. [Online Manual].

[11] M. Schoeberl, "Digital Design in CHISEL," 2020.

[12] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The Rocket Chip Generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[13] A. Rao, "The RoCC doc v2: An Introduction to the Rocket Custom Coprocessor Interface," tech. rep., Tech. Rep, 2020.

[14] J. Katz and Y. Lindell, *Introduction to Modern Cryptography: Principles and Protocols*. Chapman and hall/CRC, third ed., 2021.

[15] J.-P. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2017.

[16] D. Chakraborty, A. Dutta, and S. Kundu, "Designing Tweakable Enciphering Schemes Using Public Permutations," *Cryptology ePrint Archive*, 2021.

[17] S. S. Dhanda, B. Singh, and P. Jindal, "Lightweight Cryptography: A Solution to Secure IoT," *Wireless Personal Communications*, vol. 112, pp. 1947–1980, 2020.

[18] L. Bassham, Ç. Çalık, K. McKay, and M. S. Turan, "Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process," *US National Institute of Standards and Technology*, 2018.

[19] M. Sonmez Turan, K. McKay, D. Chang, L. E. Bassham, J. Kang, N. D. Waller, J. M. Kelsey, and D. Hong, "Status Report on the Final Round of the NIST Lightweight Cryptography Standardization Process," NIST Interagency or Internal Report (IR) NIST IR 8454, National Institute of Standards and Technology, Gaithersburg, MD, 2023.

[20] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1. 2: Lightweight Authenticated Encryption and Hashing," *Journal of Cryptology*, vol. 34, pp. 1–42, 2021.

[21] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge-Based Pseudo-Random Number Generators," in *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*, pp. 33–47, Springer, 2010.

[22] C. De Canniere and B. Preneel, "Trivium," in *New Stream Cipher Designs: The eSTREAM Finalists*, pp. 244–266, Springer, 2008.

[23] S. Chung, S. H. Wu, and E. Yang, "PUFcc: An Essential Crypto Coprocessor for RISC-V." White Paper, 2021.

[24] T.-T. Hoang, C. Duran, R. Serrano, M. Sarmiento, K.-D. Nguyen, A. Tsukamoto, K. Suzaki, and C.-K. Pham, "Trusted Execution Environment Hardware by Isolated Heterogeneous Architecture for Key Scheduling," *IEEE Access*, vol. 10, pp. 46014–46027, 2022.

[25] J. Prior, "SiFive Shield: An Open, Scalable Platform Architecture for Security," 2019.

[26] T. Gomes, P. Sousa, M. Silva, M. Ekpanyapong, and S. Pinto, "FAC-V: An FPGA-Based AES Coprocessor for RISC-V," *Journal of Low Power Electronics and Applications*, vol. 12, no. 4, p. 50, 2022.

[27] Y. Chen, H. Chen, S. Chen, C. Han, W. Ye, Y. Liu, and H. Zhou, "DITES: A Lightweight and Flexible Dual-Core Isolated Trusted Execution SoC Based on RISC-V," *Sensors*, vol. 22, no. 16, p. 5981, 2022.

[28] B. Varughese and K. Riyas, "A Novel Method for Built-In-Self-Test Architecture using Crypto Core in IoT," in *2020 Third International Conference on Advances in Electronics, Computers and Communications (ICAECC)*, pp. 1–6, IEEE, 2020.

[29] M. A. Haque and M. L. Ali, "Design of a Hummingbird Crypto Core Implementing BIST Technique," in *2016 9th International Conference on Electrical and Computer Engineering (ICECE)*, pp. 82–85, IEEE, 2016.

[30] S. Limnaios, N. Sklavos, and O. Koufopavlou, "Lightweight Efficient SIMECK32/64 Crypto-Core Designs and Implementations, for IoT Security," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 275–280, IEEE, 2019.

[31] R. Bansal, "PRESENT Crypto-Core as Closely-Coupled Coprocessor for Efficient Embedded SOCs," in *2020 24th International Symposium on VLSI Design and Test (VDAT)*, pp. 1–6, IEEE, 2020.

[32] D. A. N. Gookyi and K. Ryoo, "A Lightweight System-On-Chip Based Cryptographic Core for Low-Cost Devices," *Sensors*, vol. 22, no. 8, p. 3004, 2022.

[33] X. Wei, M. El-Hadedy, S. Mosanu, Z. Zhu, W.-M. Hwu, and X. Guo, "RECOHCON: A High-Throughput Reconfigurable Compact ASCON Processor for Trusted IoT," in *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, pp. 1–6, IEEE, 2022.

[34] K. Mohajerani, R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results," *Cryptology ePrint Archive*, 2020.

[35] J.-P. Kaps, W. Diehl, M. Tempelmeier, E. Homsirikamol, and K. Gaj, "Hardware API for Lightweight Cryptography," *Cryptographic Engineering Research Group, GMU*, pp. 1–26, 2019.

[36] L. E. Bassham III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, *et al.*, *Sp 800-22 rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. 2010.

[37] Brown, Robert G and Eddelbuettel, Dirk and Bauer, David, "Dieharder," *Duke University Physics Department Durham, NC*, pp. 27708–0305, 2018.

[38] E. B. Barker and J. M. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," *National Institute of Standards & Technology*, 2015.

[39] M. J. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping," *National Institute of Standards & Technology*, 2012.

[40] E. Barker, E. Barker, A. Roginsky, and R. Davis, "Recommendation for Cryptographic Key Generation," *National Institute of Standards & Technology*, 2012.

Apéndice A

Chisel

El código del criptoprocesador y sus simulaciones se encuentra en el repositorio RV_SPU[1], El fork que contiene a la interfaz para generar el bitstream para cargarlo a la FPGA ArtyA7 freedom fork[2] y los códigos en C se encuentran en freedom-e-sdk fork[3].

A.1. Inicialización de un ambiente Chisel

Chisel es un lenguaje de descripción de hardware *HDL* embebido en Scala. En pocos términos es un paquete de Scala, por lo tanto dentro del código Chisel se pueden usar listas, map, reduce como una aplicación en Scala. Este lenguaje describe a los circuitos desde el paradigma orientado a objetos. Permite generar circuitos complejos y parametrizables para generar Verilog sintetizable.

Para inicial con el desarrollo de Hardware en Chisel es recomendable seguir la siguiente plantilla Plantilla[4]. Es necesario instalar el JDK junto a JRE de preferencia 8, 11 o 17 aquellos que sean LTS (Algunos proyectos funcionan con la versión 8 y otros con la versión 17). También se debe instalar tanto Scala (aveces se usa Scala 2 y aveces Scala 3) como SBT. Si se desea realizar test y visualizar los archivos que contienen las señales de onda, el siguiente archivo sbt es recomendable.

Código A.1: Instalación JDK Linux Ubuntu.

```
sudo apt-get install default-jdk
sudo apt install python-is-python3
```

[1] git@github.com:AlbertoJOR/RV_SPU.git
[2] git@github.com:AlbertoJOR/freedom.git
[3] git@github.com:AlbertoJOR/freedom-e-sdk.git
[4] https://github.com/freechipsproject/chisel-template/tree/main

Código A.2: Instalación de Scala3 Linux Debian

```
curl -fL https://github.com/coursier/coursier/releases/latest/
  download/cs-x86_64-pc-linux.gz | gzip -d > cs && chmod +x cs &&
  ./cs setup
```

Código A.3: Instalación de sbt Linux Ubuntu X86-64.

```
echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" |
  sudo tee /etc/apt/sources.list.d/sbt.list
echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee
  /etc/apt/sources.list.d/sbt_old.list
curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0
  x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
sudo apt-get update
sudo apt-get install sbt
```

Código A.4: Archivo build.sbt utilizado para desarollar y probar distintos circuitos.

```
1  ThisBuild / scalaVersion     := "2.13.8"
2  ThisBuild / version          := "0.1.0"
3  ThisBuild / organization     := "%ORGANIZATION%"
4
5  val chiselVersion = "3.5.6"
6
7  lazy val root = (project in file("."))
8    .settings(
9      name := "ASCON",
10     libraryDependencies ++= Seq(
11       "edu.berkeley.cs" %% "chisel3" % chiselVersion,
12       "edu.berkeley.cs" %% "chisel-iotesters" % "2.5.5",
13       "edu.berkeley.cs" %% "chiseltest" % "0.5.4" % "test",
14       "edu.berkeley.cs" %% "treadle" % "1.5.4",
15       "org.scalatest" %% "scalatest" % "3.2.15" % "test",
16       "org.scalacheck" %% "scalacheck" % "1.17.0"
17     ),
18     scalacOptions ++= Seq(
19       "-language:reflectiveCalls",
20       "-deprecation",
21       "-feature",
22       "-Xcheckinit",
23       "-P:chiselplugin:genBundleElements",
24     ),
25     addCompilerPlugin("edu.berkeley.cs" % "chisel3-plugin" %
     chiselVersion cross CrossVersion.full),
26   )
```

Es importante recalcar que no hay estabilidad en los distintos proyectos que usen Chisel, algunos usan Chisel 2 o Chisel 3. Al igual es altamente dependiente de la versión de JDK, la versión de Scala, la versión de SBT y de cada una de los paquetes que se utilicen. El Código A.5 muestra un ejemplo sencillo de una compuerta `AND` parametrizable.

Código A.5: Circuito ejemplo de una compuerta and de tamaño arbitrario descrito en Chisel3.

```
import chisel3._
class AndModule(len : Int) extends Module{
  val io = IO( new Bundle {
    val inA = Input(UInt(len.W))
    val inB = Input(UInt(len.W))
    val S = Output(UInt(len.W))
  })
    io.S := io.inB & io.inA
}
// Generacion de Verilog en el directorio AndModuleVerilog.
object AndModuleGen extends App {
  val myverilog = (new ChiselStage).emitVerilog(
    new AndModule(8),
    Array("--target-dir", "AndModuleVerilog/")
  )
}
```

La generación de una simulación del hardware o prueba se realiza utilizando las pruebas de escala mas las pruebas de Chisel. Por ello dentro del build.sbt se tiene que incluir las bibliotecas chisel-iotesters, chiseltest. Los archivos de simulación deben estar dentro del directorio test/scala. Chisel3 tiene la capacidad de realizar tests junto a una función de Scala que simule la misma operación y generar los diagramas de tiempo, el Código A.6 muestra la simulación de una compuerta `AND` junto a la creación del archivo con el diagrama de tiempo.

Código A.6: Prueba simple del módulo and con generación de señales lógicas.

```
import chisel3._
import chiseltest._
import org.scalatest.freespec.AnyFreeSpec
class AndModuleTest extends AnyFreeSpec with ChiselScalatestTester {
  "And module test" in {
  test(new AndModule(8)).withAnnotations(Seq(WriteVcdAnnotation)) {
   dut =>
      dut.io.inA.poke(5.U)
      dut.io.inB.poke("h9".U)
      dut.io.S.expect("h1".U)
      dut.clock.step(1)
      println("Last output value :" + dut.io.S.peek().litValue.
   toString(16))
    }
    println("And module working correctly")
  }
}
```

Código A.7: Código Verilog generado a partir de Chisel del circuito AndModule.

```
module AndModule(
  input         clock,
```

```
3   input         reset,
4   input   [7:0] io_inA,
5   input   [7:0] io_inB,
6   output  [7:0] io_S
7 );
8   assign io_S = io_inB & io_inA; // @[AndModule.scala 11:20]
9 endmodule
```

Cuando se compilen los archivo en la terminal se inicia el compilador sbt para ello se introduce sbt en el directorio raíz. El Código A.7 muestra la transformación a verilog de la compuerta `AND` con un tamaño de 8 bits de entrada.

Código A.8: Inicialización de la consola sbt.

```
$ sbt
```

Una vez dentro de la consola de sbt se le dan los siguientes comandos, el primero compila y realiza las pruebas que se encuentren en el directorio de test, si uno de las pruebas no termina satisfactoriamente. Entonces la compilación termina. Cuando se genere una prueba, los archivos con los diagramas de tiempo se encuentran en un nuevo directorio denotado por la cadena de caracteres antes del **in** en el Código A.6. En este caso el directorio que se genera lleva el nombre de **And_module_test** dentro del directorio llamado **test_run_dir**, la Apéndice A.1 muestra los directorios principales.

Código A.9: Compilación de los archivos fuente y prueba del archivo AndModule.

```
> compile   # Compilacion de todos lo archivos.
> testOnly AndModuleTest    # Prueba de un solo archivo.
```

Código A.10: Salida de la prueba del modulo AndModule

```
Testing started at 11:08 ...

Last output value :1
And module working correctly

Process finished with exit code 0
```

Si se desea visualizar los archivos con los diagramas generados, se debe utiliza el programa gtkwave. La Figura A.2 muestra el diagrama de tiempo de la prueba a la compuerta `AND`. Su instalación y ejecución es la siguiente.

Código A.11: Instalación y ejecución de gtkwave Linux Debian

```
$ sudo apt-get update
$ sudo apt-get install gtkwave
$ gtkwave archivoOnda.vcd formatoSenales.gtkw
```

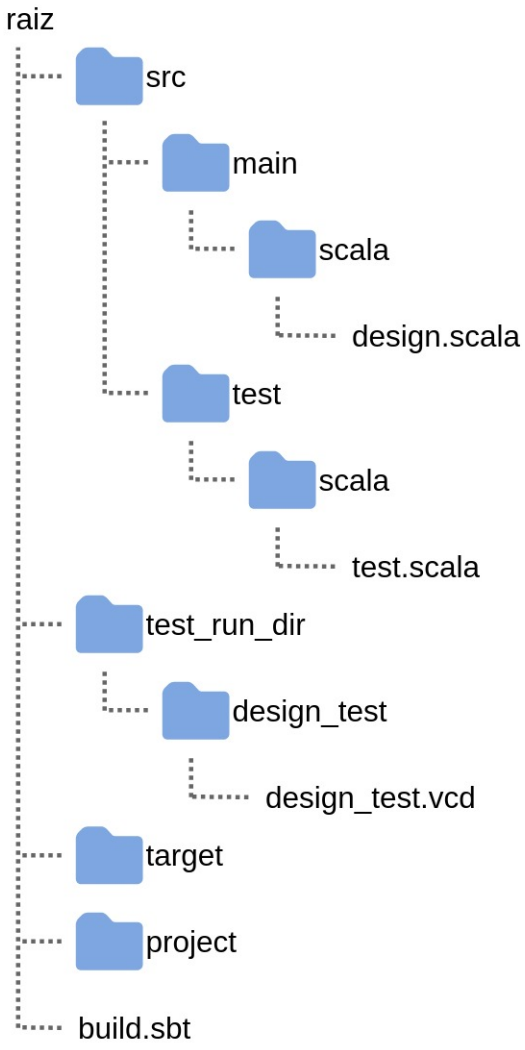


Figura A.1: Estructura de un general de un proyecto Chisel3 en Scala.

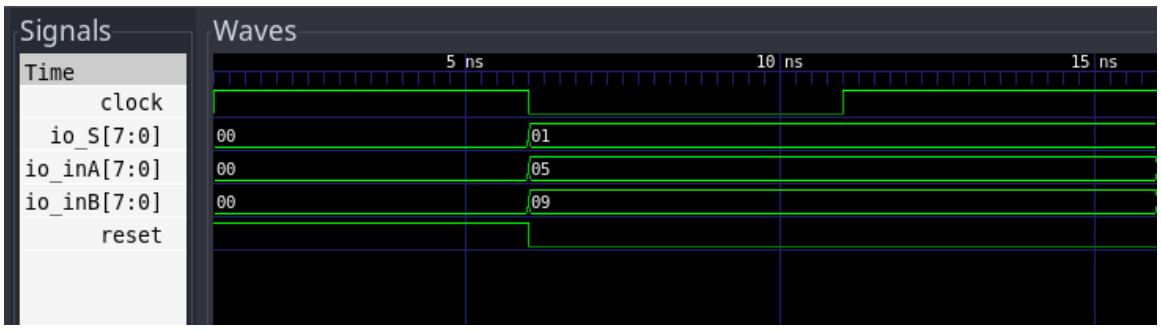


Figura A.2: Visualización de las señales generadas en las prueba de módulo And-Module en gtkwave.

Apéndice B

RocketChip Scala

En los últimos de años el mantenimiento y desarrollo de Rocket-Chip queda a cargo de la empresa SiFive. El proyecto sigue siendo código abierto con licencias permisivas para el desarrollo y comercialización. Sin embargo, ya no se le da mantenimiento al proyecto Freedom ni Freedon-e-sdk. Freedom incluye el hardware por lo tanto el directorio de rocket-chip a quedado desactualizado, es posible que se tenga que actualizar la interfaz para comunicarse con el procesador Rocket. Dentro de el repositorio Freeedom se encuentra el repositorio de RocketChip que contiene el código fuente en Scala, emuladores y simuladores, mas los archivos make y scripts para cada uno de los componentes que se deseen usar en el repositorio. Este incluye los núcleo Rocket y Boom.

En cuanto a Rocketchip este incluye todo sus bus de sistema para generar un SoC. Este núcleo puede personalizarse al instanciar los conjunto de la ISA que se deseen. Se encuentran las versiones de 32 y 64 bits, adicionalmente se puede modificar para crear nuevas instrucciones, agregar aclaradores o componentes de hardware adicionales. Permite la generación de SoC con arquitecturas multinúcleo heterogéneas.

B.0.1. Generador Rocket-Chip

En el se puede emular el núcleo y simular su funcionamiento con códigos en C. Para ello se tiene que clonar el repositorio.

Código B.1: Clonación y actualización del repositorio Rocket Chip Linux Debian

```
$ git clone https://github.com/ucb-bar/rocket-chip.git
$ cd rocket-chip
$ git submodule update --init
```

Posterior a la descarga y actualización del repositorio rocket-chip, se debe apuntar al conjunto de herramientas (*toolchain*) de RISC-V. Esta *toolchain* se caracteriza por no ser estable en las versiones, cada proyecto tiende a modificarla para sopor-

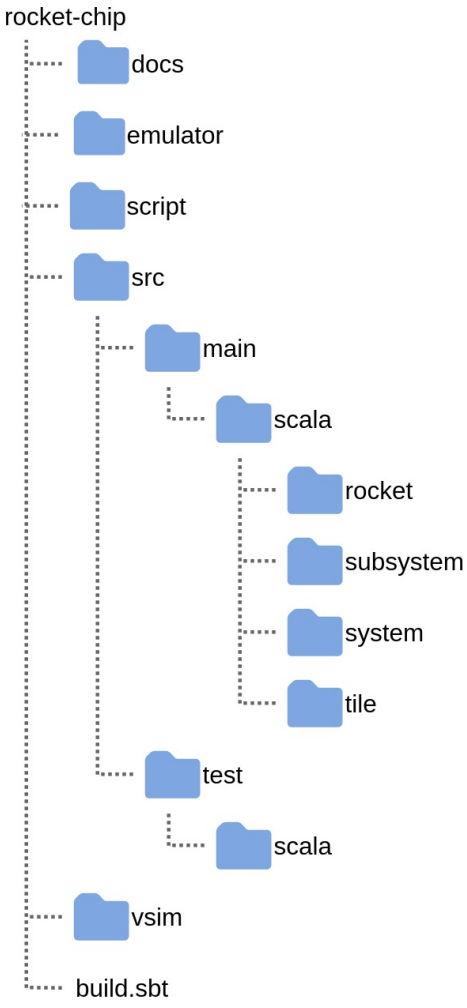


Figura B.1: Estructura del repositorio rocket-chip con los directorios mas importantes

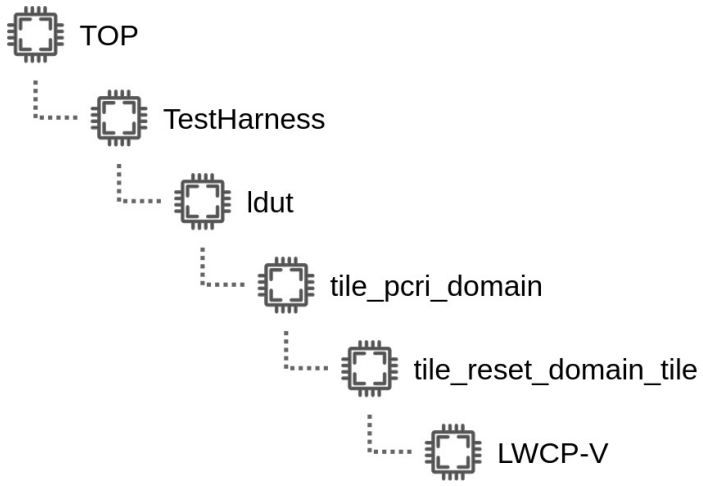


Figura B.2: Ruta de los componentes del coprocesador RoCC

tar nuevas capacidades, por ello siempre se debe usar la versión que se utiliza en los repositorios. En el caso de rocket-chip indica la versión de la *toolchain* ya que se encuentra en el repositorio rocket-tools. Antes de poder instalar tanto rocket-chip como rocket-tools se tiene que construir la *toolchain*, se debe usar la que está contenida en rocket-tool en el directorio **riscv-gnu-toolchain**. Es un proceso tardado, por lo tanto hay que ser pacientes y que la construcción no tenga errores en la compilación.

Código B.2: Instalación de rocket-tools

```
# Dependencias para Ubuntu, instalarlas previamente.
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
    libmpfr-dev libgmp-dev libusb-1.0-0-dev gawk build-essential
   bison flex texinfo gperf libtool patchutils bc zlib1g-dev device-
   tree-compiler pkg-config libexpat-dev libfl-dev
$ git submodule update --init --recursive
$ export RISCV=/direccion/a/toolchain #apunta al directorio dentro
   de rocket-tools llamado riscv-gnu-toolchain
$ ./build.sh
```

Código B.3: Instalación de la toolchain RISC-V

```
# Dependencias necesarias para Ubuntu.
$ sudo apt-get install autoconf automake autotools-dev curl python3
   python3-pip libmpc-dev libmpfr-dev libgmp-dev gawk build-
   essential bison flex texinfo gperf libtool patchutils bc zlib1g-
   dev libexpat-dev ninja-build git cmake libglib2.0-dev
# Construcciión para las plataformas de 32 bits.
$ ./configure --prefix=/opt/riscv --with-arch=rv32gc --with-abi=
   ilp32d
$ make linux
```

La figura B.1 ejemplifica gráficamente los directorios más importantes para pruebas y simulación de los procesadores diseñados. En el directorio **emulator** se encuen-

tran los archivos para generar un emulador del núcleo usando verilator. Con el se pueden simular con un proxy kernel en terminal, ver los datos en los registros ciclo a ciclo y obtener el archivo de tiempos para depurar los circuitos internos del núcleo. Para poder localizar el coprocesador la figura B.2 muestra la ruta donde se localiza en los diagramas de tiempo, los componentes en verilog y los reportes de utilización. El coprocesador se encuentra en los componentes tile junto a su procesador.

Código B.4: Instalación de rocket-chip

```
git clone https://github.com/freechipsproject/rocket-tools
cd rocket-tools
git submodule update --init --recursive
export RISCV=/direccion/a/la/instalacion/de/riscv/toolchain # en el
   directorio rocket-tools.
export MAKEFLAGS="$MAKEFLAGS -jN" # N nucleos del sistema
./build.sh
./build-rv32ima.sh (para los procesadores RV32).
```

Antes de realizar la compilación en scala de rocket-chip o repositorios que cuenten con rocket-chip siempre hay que modificar el archivo de python que genera la ROM para el arranque, esta se encuentra en la siguiente dirección rocket-chip/scripts/vlsi_rom_gen. Cambiar el segundo **raise** por **return** línea 97.

Una vez que se haya compilado y configurado rocket-chip se pueden crear distintos cores. Para este ejemplo se genera un núcleo con el conjunto de instrucciones rv32imac, que puede ser depurado usando el GDB. Todas las configuraciones se encuentran en la dirección rocket-chip/src/main/scala/system/Config.scala.En este archivo se encuentra el diseño principal del núcleo.

Código B.5: Adición de una nueva configuración del núcleo rocket en el directorio rocket-chip/src/main/scala/system/Configs.scala.

```
1 class MyTinyConfigJTAG extends Config(
2     new WithJtagDTMSystem ++
3     new TinyConfig
4 )
```

Antes de emular el procesador se debe de generar el emulador en el directorio **emulator**. Los siguientes comandos permiten generarlo.

Código B.6: Generació del emulador con Verilator.

```
1 $ cd emulator
2 $ make
```

Cuando haya sido construido el emulador con éxito, ya es apto para poder emular los núcleos que se vayan compilando. Para generar un emulador exclusivo para el núcleo que se está probando se realiza de la siguiente manera.

Código B.7: Generació del emulador para el núcleo que se definió.

```
$ cd emulator
$ make
```

Código B.8: Generació del emulador para el núcleo que se definió.

```
$ make CONFIG=freechips.rocketchip.system.MyTinyConfigJTAG
$ make CONFIG=freechips.rocketchip.system.MyTinyConfigJTAG debug
```

```
export RISCV=~/GitHub/rocket-tools/riscv-gnu-toolchain
export PATH=$PATH:${RISCV}/bin
    rocket-tools/riscv-tests/benchmarks
    Makefile
    XLEN ?= 32
    bmarks = \
          mult \
    mult/mult_main.c
#include <stdio.h>
int main(){
  int a = 8;
  int b = 12;
  int res = a * b;
  printf("El resutado de a * b es:\n");
  printf("res = %d * %d\n");
  printf("res = %d \n");
  return 0;}
mult.riscv
mult.riscv.dump
```

```
    sudo apt install libncurses5-dev  libncursesw5-dev  libncurses5
    libtinfo5 libtinfo-dev
```

B.0.2. SoC Arty A7-100T

Sifive proporciona dos repositorios para el desarrollo de hardware en rocket-chip freedom [1] y otro para la programación de cores basados en rocket-chip mas sus targetas de desarrollo freedom-e-sdk[2]. El desarrollo en hardware si se quiere implementar en un fpga este está fuertemente soportado para las tarjetas de desarrollo artyA7. Para cagar los bitstreams a la tarjeta es necesario contar con vivado, sin embargo puede usarse la versión WebPack. Para cargar programas a la tarjeta es bueno seguir el tutorial artyA7 guide `https://sifive.cdn.prismic.io/sifive%2Fed96de35-065f-474c-a432-9f6a364af9c8_sifive-e310-arty-gettingstarted-v1.0.6.pdf`

Este proyecto necesita de Vivado y que exista una variable de ambiente `vivado`. La versión instalada de Vivado corresponde a Vivado ML 2022.2 la cual se obtiene

[1] git@github.com:sifive/freedom.git
[2] git@github.com:sifive/freedom-e-sdk.git

de la su página[3] antes de iniciar la instalación, se necesitan las siguientes aplicaciones en Ubuntu Debian:

```
1     sudo apt install libncurses5-dev  libncursesw5-dev  libncurses5
      libtinfo5 libtinfo-dev
```

Posteriormente con lo siguientes comandos se puede generar los componentes en verilog y sintetizar el bitstream para la placa de desarrollo ArtyA7-100T.

Código B.9: comandos para generar el bitstream (freedom)

```
1 make BOARD=arty_a7_100 -f Makefile.e300artydevkit clean
2 make BOARD=arty_a7_100 -f Makefile.e300artydevkit verilog
3 make BOARD=arty_a7_100 -f Makefile.e300artydevkit mcs
```

Para generar el bitstream de una configuración distinta se modifican las clases config. En este caso se realizan los aceleradores RoCC de ejemplo para probar el funcionamiento de estos. El siguiente archivo es el que tiene que ser modificado. Si se desaa una configuración del SoC diferente a la de defecto en los comandos anteriores make se tienen que agregar la etiqueta `CONFIG=NuevaConfig`. Este nombre debe de coincidir con el nombre de la configuración final, la que describe a todo el SoC. Lo mejor es copiar la clase de configuración y modificarla para implementar una nueva arquitectura.

Código B.10: src/main/scala/everywhere/e300artydevkit/Config.scala

```
1  class DefaultFreedomEConfig extends Config(
2    new WithNBreakpoints(2) ++
3      new WithNExtTopInterrupts(0) ++
4      new WithJtagDTM ++
5      new WithL1ICacheWays(2) ++
6      new WithL1ICacheSets(128) ++
7      new WithDefaultBtb ++
8      new WithRoccExample ++
9      new TinyConfig
10 )
```

[3]https://www.xilinx.com/support/download.html

Apéndice C

Programación de las plataformas RISC-V

La pasos para programar los SoC contruidos y la placa de desarrollo HiFive se realizo siguiendo el tutorial Digikey[1]. A continuación se resumen los pasos para usar los repositorios freedom y freedom-e-sdk para desarrollar software

En primer lugar se tienen que descargar las versiones de la cadena de herramientas GNU risc-v y la versión de OpenOCD. Estas de obtienen de freedom-tools[2]. La programación se realizo con las versiones `8.3.0-2019.08.0` para las herramientas y `0.10.0-2019.08.2` para openOCD.

```
wget -c https://static.dev.sifive.com/dev-tools/riscv64-unknown-elf-
    gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14.tar.gz
wget -c https://static.dev.sifive.com/dev-tools/riscv-openocd
    -0.10.0-2019.08.2-x86_64-linux-ubuntu14.tar.gz
tar xf riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14
    .tar.gz
tar xf riscv-openocd-0.10.0-2019.08.2-x86_64-linux-ubuntu14.tar.gz
```

Se crean las reglas para udev Olimex creado el archivo: /etc/udev/rules.d/99-openocd.rules , con el siguiente contenido.

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba",ATTRS{idProduct}=="002a",
    MODE="664", GROUP="plugdev"
SUBSYSTEM=="usb", ATTR{idVendor}=="15ba",ATTR{idProduct}=="002a",
    MODE="664", GROUP="plugdev"
```

Posteriormente se instala udev en vivado:

[1] https://forum.digikey.com/t/digilent-arty-a7-with-xilinx-artix-7-implementing-sifive-fe310-risc-v/13311

[2] https://github.com/sifive/freedom-tools/releases

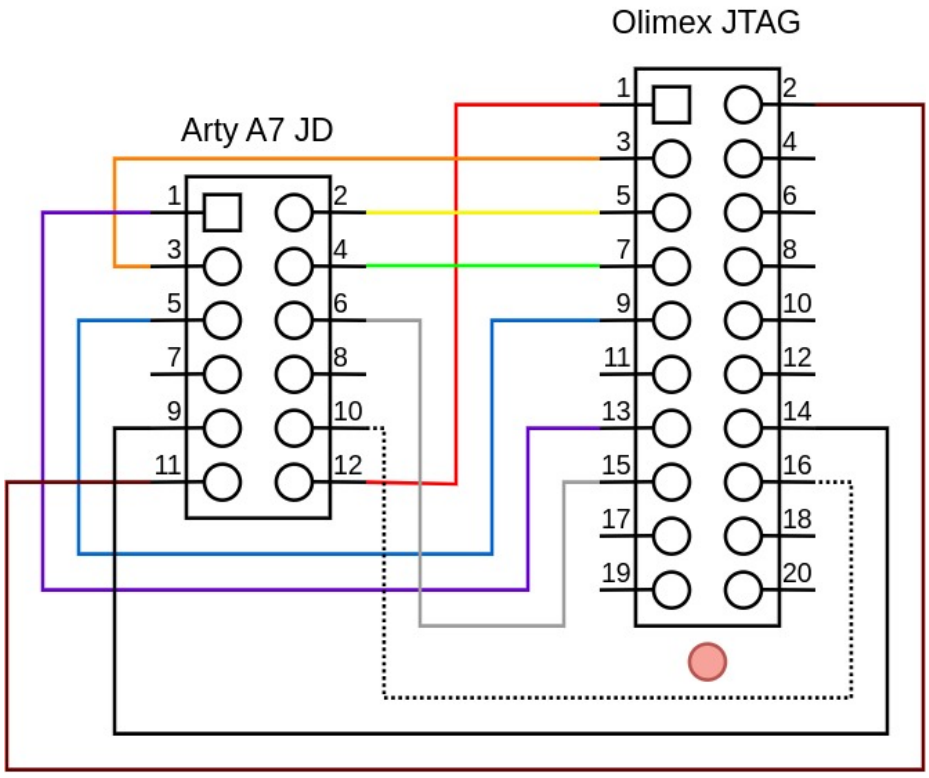


Figura C.1: Conexión Arty A7 con el depurador Olimex

```
cd  /opt/Digilent/Xilinx/Vivado/2017.1/data/xicom/cable_drivers/
   lin64/install_script/install_drivers/
sudo ./install_drivers
```

J-Link Segger Se pueden cargar los programas a las distintas plataformas:

```
1 make BSP=metal PROGRAM=hello TARGET=freedom-e310-arty clean
2 make BSP=metal PROGRAM=hello TARGET=freedom-e310-arty software
3 make BSP=metal PROGRAM=hello TARGET=freedom-e310-arty upload
```

Para obtener las instrucciones que conforman al programa se usa:

```
1    riscv64-unknown-elf-objdump -d archivo.elf > desensablado.txt
```

Para programar los softcores en la tarjeta Arty A7 se realiza la conexión mostrada en la Figura C.1 del Depurador Olimex con el Pmod JD de la Arty A7.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará la **Sr. Alberto Josué Ortiz Rosales**, declaramos que hemos revisado la tesis titulada:

Criptoprocesador ligero en RISC-V de 32 bits basado en ASCON

Y consideramos que cumple con los requisitos para obtener el Grado de Maestría en Ciencias en Computación.

Atentamente,

Dr. Amilcar Meneses Viveros
Investigador del Departamento de Computación

Dr. Arturo Díaz Pérez
Investigador de la Unidad Guadalajara