DEPARTAMENTO DE ELÉCTRICA, ELECTRÓNICA Y TELECOMUNICACIONES
CARRERA DE TELECOMUNICACIONES

"Diseño e implementación de aceleradores en hardware para funciones de seguridad y cifrado, utilizando bibliotecas Vitis y plataformas SoC-FPGA, y evaluando su rendimiento computacional."

TRABAJO DE INTEGRACIÓN CURRICULAR PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN TELECOMUNICACIONES

Autor: Rodriguez Sevilla Paolo Estefano

Director del Proyecto: Ing. Navas Viera Byron Roberto, PhD.
Director de Carrera: Ing. Altamirano Carlos Daniel, PhD.

Sangolquí, Ecuador | 2026

El Problema Real En Telecomunicaciones

Contexto de aplicación: 5G · IoT Masivo · Sistemas Críticos

- El procesamiento en tiempo real satura los sistemas convencionales
- Incremento de la latencia
- Consumo energético
    - La seguridad criptográfica

> *"El problema ya no es solo proteger los datos, sino hacerlo sin detener la red."*

La Solución:
Arquitectura Propuesta: Aceleración con SoC-FPGA

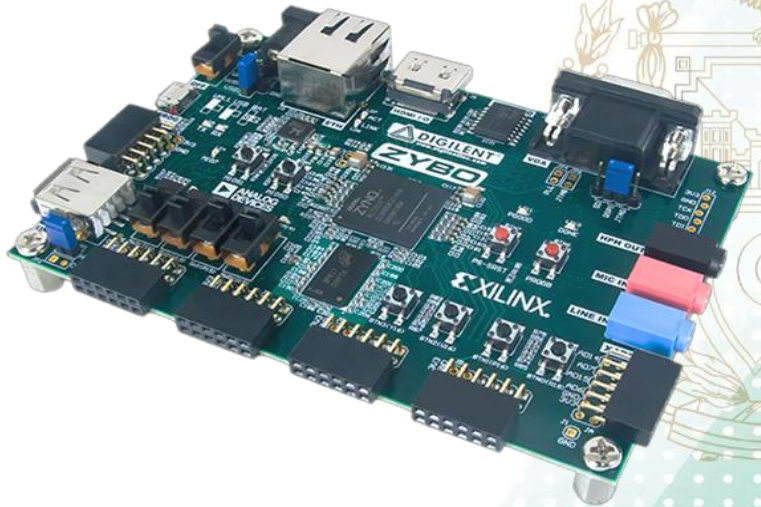
Hardware dedicado vs. Procesamiento General

CPU (GENERALISTA)

- Alta frecuencia (GHz), pero ineficiente por ciclo para tareas específicas.
- Flexible pero secuencial.
- Alto consumo energético bajo carga.

SOC - FPGA (HARDWARE PERSONALIZADO)

- Paralelismo masivo a nivel de hardware.
- Alta eficiencia por ciclo de reloj.
- Reconfigurable según la aplicación.
- Bajo consumo energético.



Objetivo y Actividades Del Proyecto

Objetivo General:

Diseño e implementación de aceleradores en hardware para funciones de seguridad y cifrado, utilizando bibliotecas Vitis y plataformas SoC-FPGA, y evaluando su rendimiento computacional.

Actividades:

Investigación y Modelado

Selección de algoritmos criptográficos y verificación con modelos en Python/C

Integración en Plataforma

Implementación de los módulos en la SoC Zynq bajo el entorno PetaLinux.

Diseño y Experimentación

Desarrollo de los aceleradores mediante Síntesis de Alto Nivel (C/C++).

Validación y Documentación

Pruebas de desempeño vs. CPU y documentación de la metodología e implementación.

Fases de Desarrollo

Modelamiento:

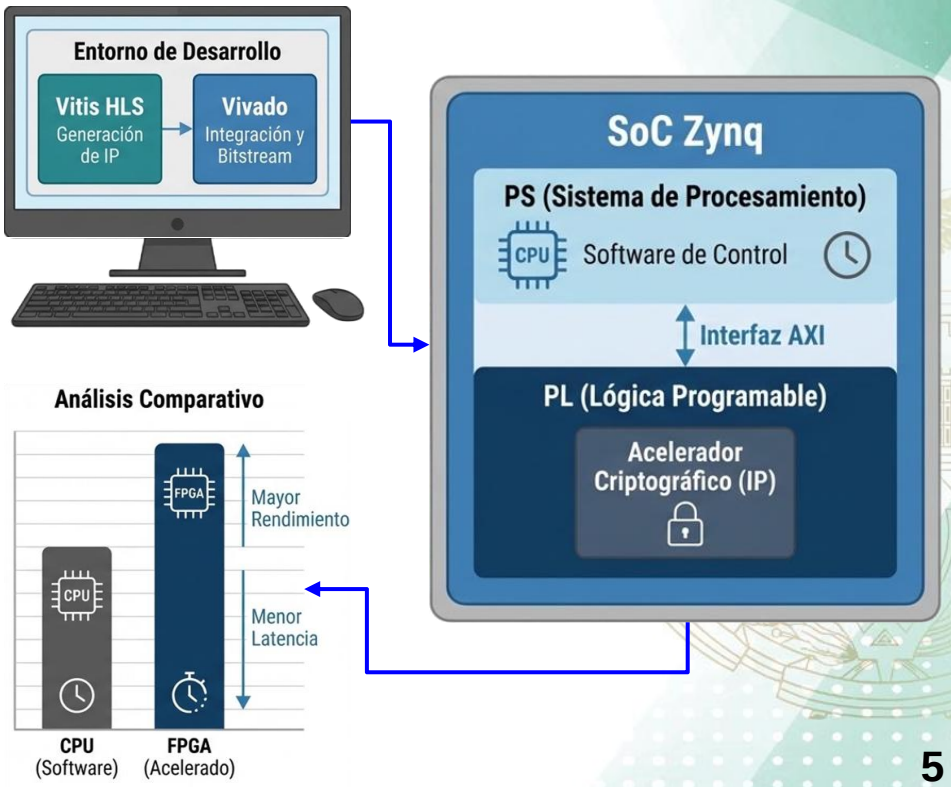
FASE I: MODELO DE REFERENCIA
Software en Python validado con vectores NIST.

FASE II: GENERACIÓN DE IPS
Uso de Vitis Libraries, síntesis de Alto Nivel (HLS) y empaquetado AXI4.

FASE III: INTEGRACIÓN & DESPLIEGUE
Conexión de las IPs, diseño en Vivado y ejecución sobre PetaLinux.

FASE IV: EVALUACIÓN DE RENDIMIENTO
Medición y análisis del desempeño computacional final.

Metodología:



Arquitectura del Sistema: Zynq-7000

Processing System (PS)

- **CPU ARM Cortex-A9.**
- **Linux embebido:** Gestión de archivos y plataforma de desarrollo.
- **Control de aceleradores hardware:** Configuración y transferencia de datos (AXI)

Programmable Logic (PL)

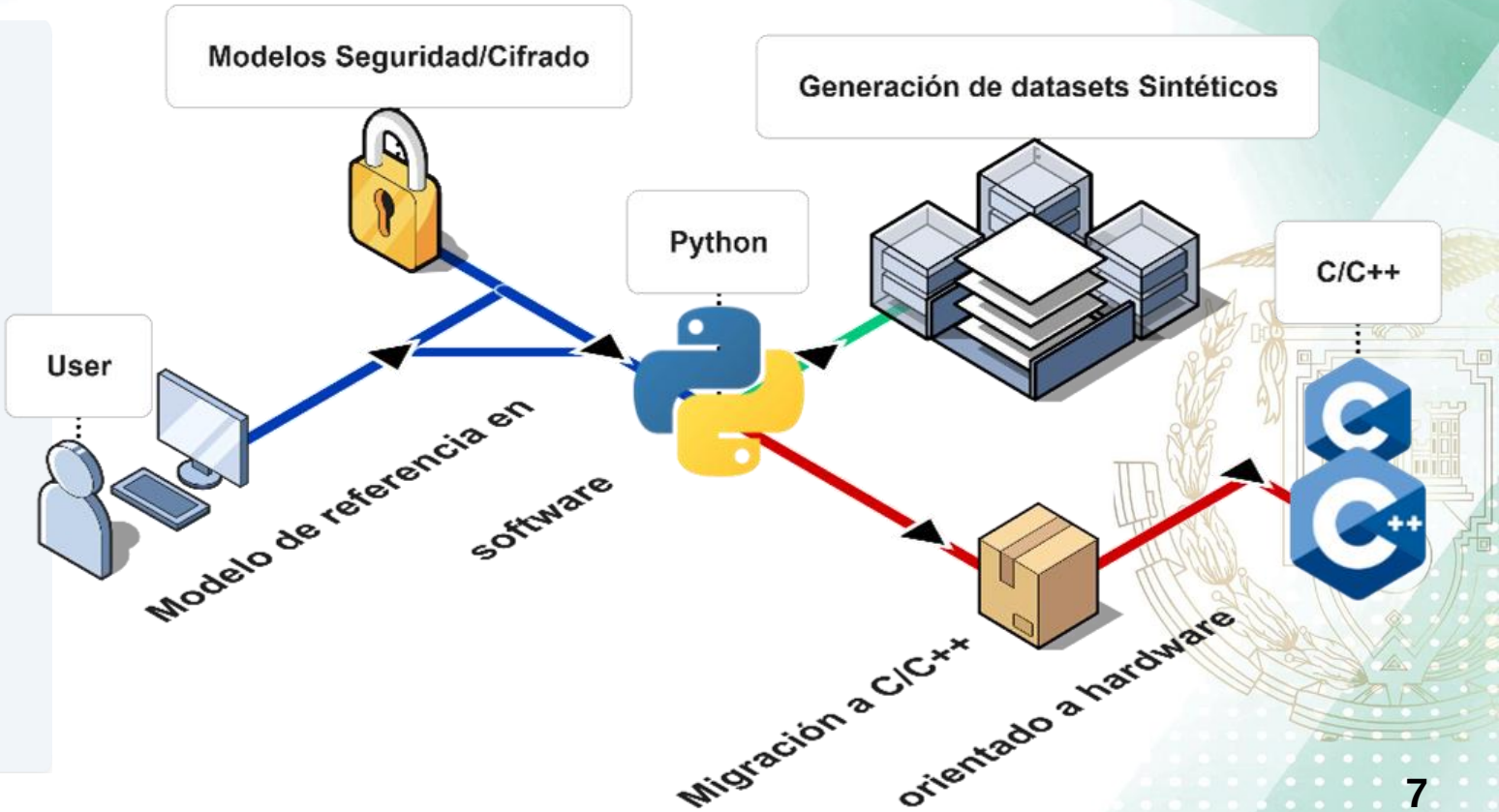
- **Aceleradores IP core:** Núcleos AES-128 y SHA-256.
- **Bloques lógicos configurables**
- **Hardware personalizado:** Hardware creado a partir de VHDL y Verilog.

Modelamiento:
FASE I - Validación Funcional

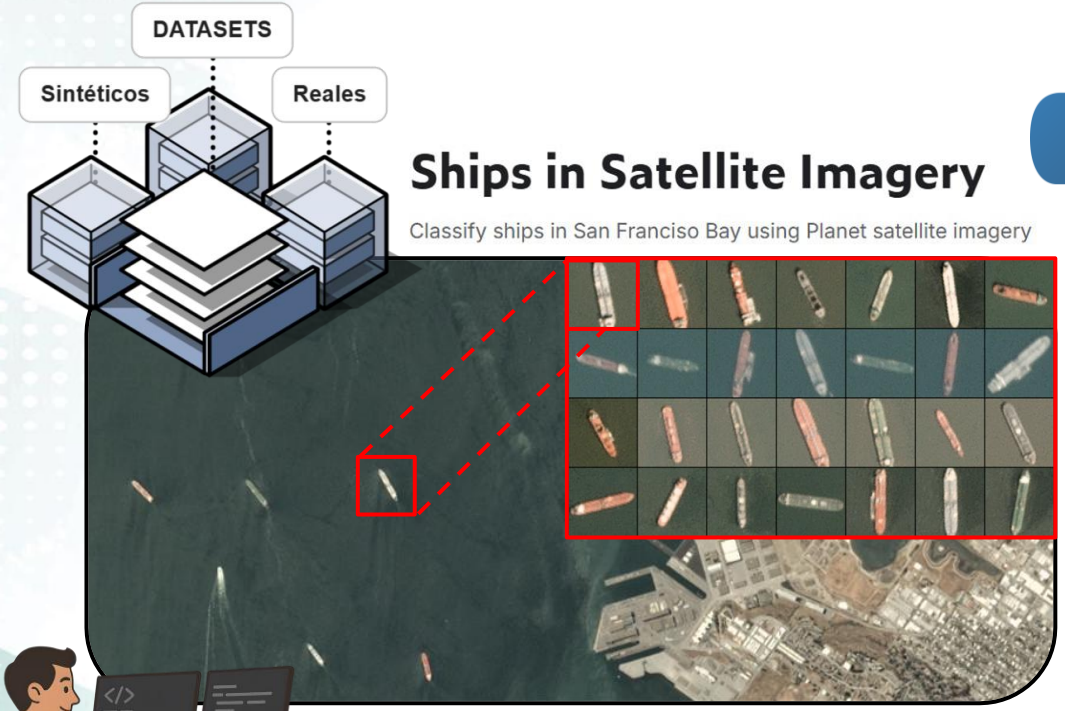
Python Script

Validación

NIST



Modelamiento:
FASE I – DATASETS



dataset_test_512MB.bin	524,288 KB
dataset_test_256MB.bin	262,144 KB
dataset_test_128MB.bin	131,072 KB
dataset_test_64MB.bin	65,536 KB
dataset_test_32MB.bin	32,768 KB
dataset_test_16MB.bin	16,384 KB
dataset_test_8MB.bin	8,192 KB
dataset_test_4MB.bin	4,096 KB
dataset_test_2MB.bin	2,048 KB

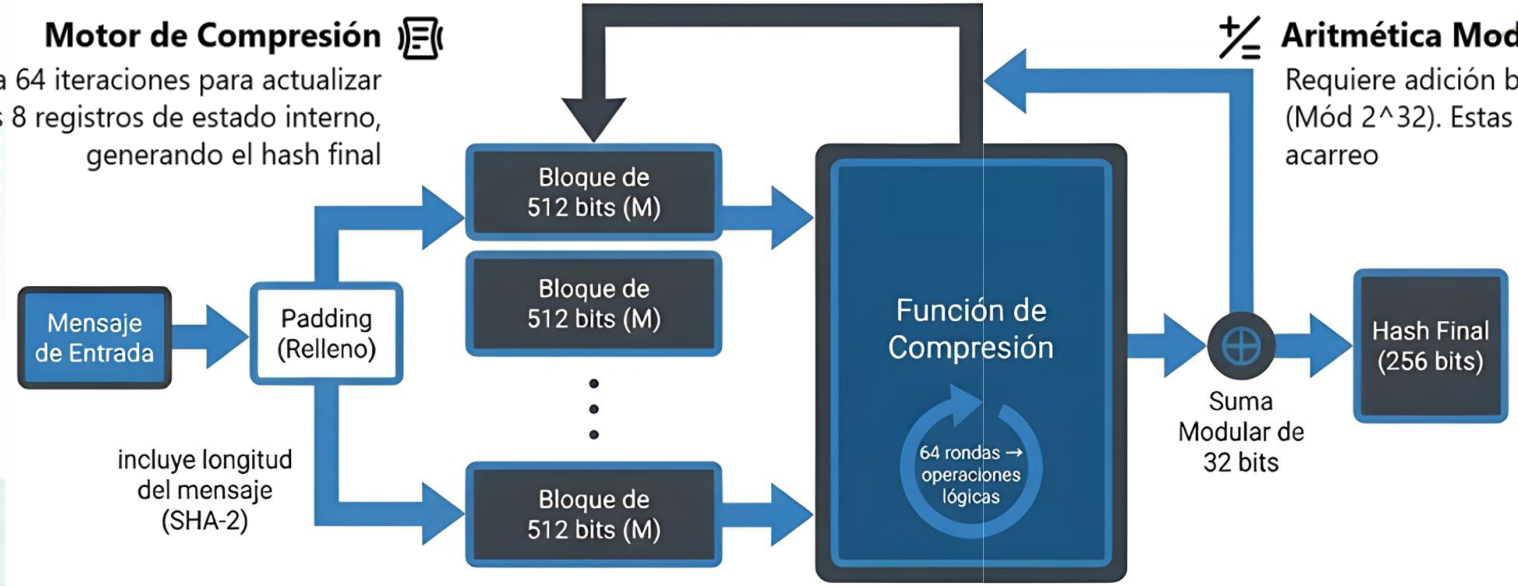
Modelamiento:
FASE I - SHA-256 Función Hash Criptográfica

Preprocesado
Añade bits al mensaje para alcanzar múltiplos de 512 bits

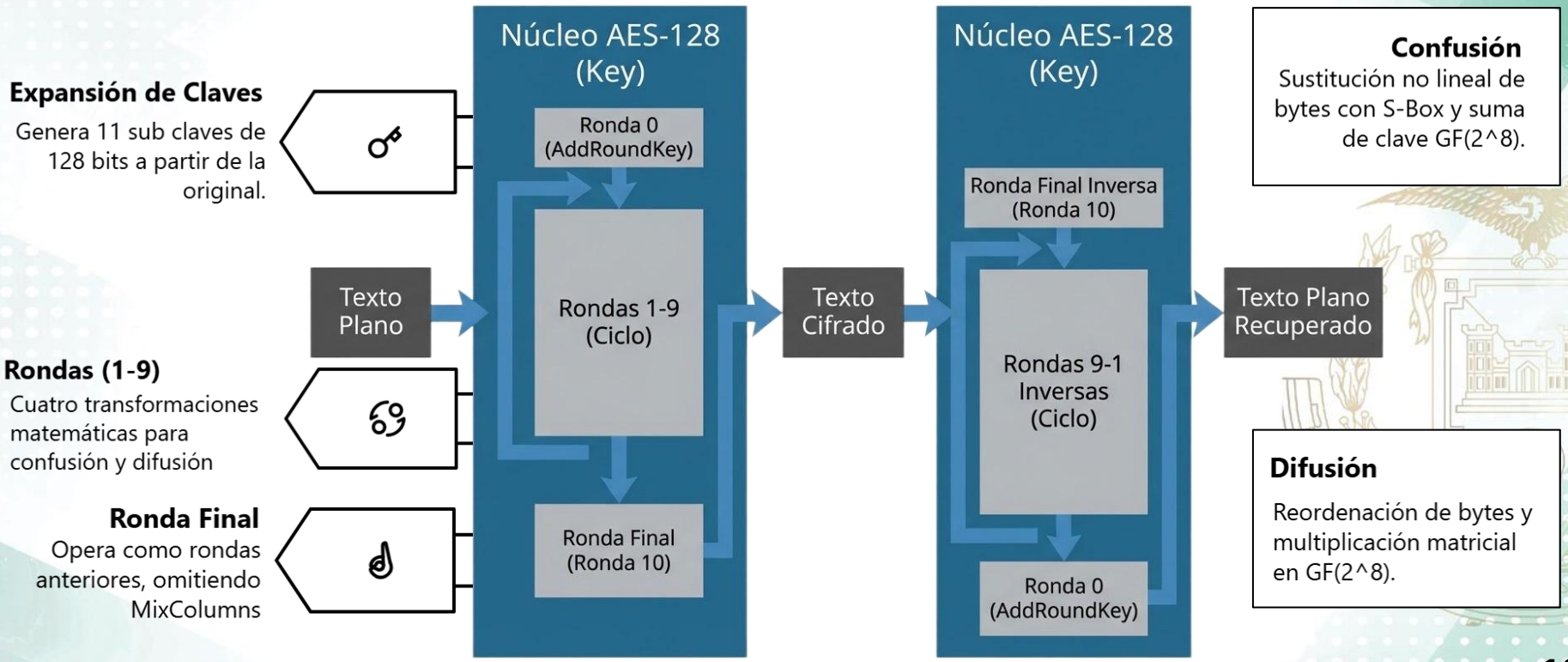
Operaciones Lógicas
Utiliza funciones lógicas y rotaciones de bits

Motor de Compresión
Realiza 64 iteraciones para actualizar los 8 registros de estado interno, generando el hash final

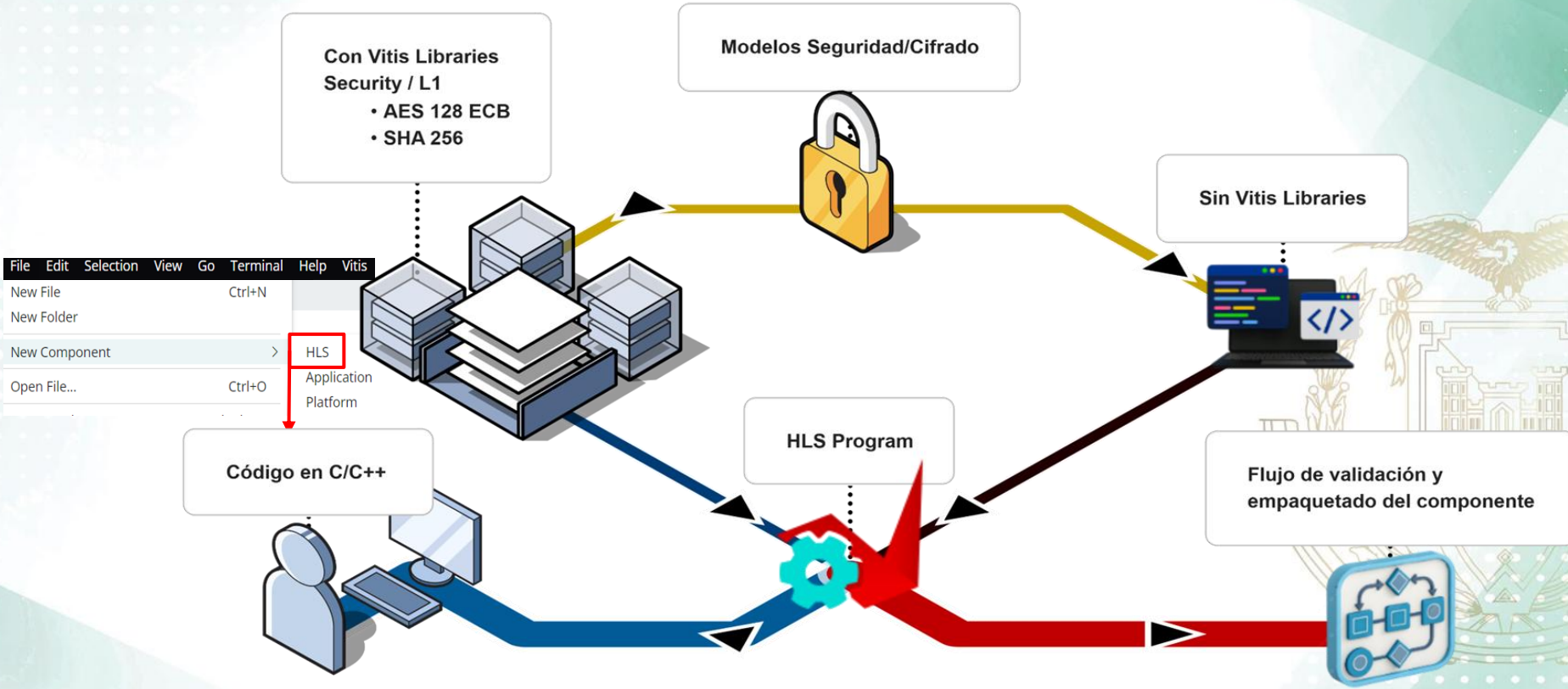
Aritmética Modular
Requiere adición binaria estándar (Mód 2^32). Estas sumas propagan el acarreo



Modelamiento:
FASE I - AES 128 ECB Cifrado simétrico por bloques

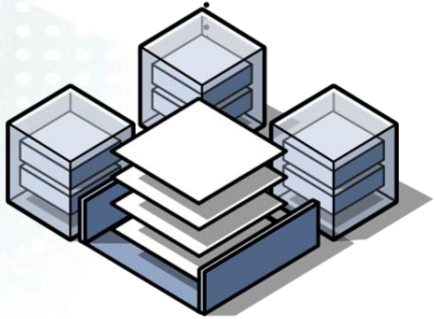


Metodología:
FASE II – Vitis HLS



Metodología:
FASE II – Vitis Library

Vitis Libraries
Security / L1
- **AES 128 ECB**
- **SHA 256**

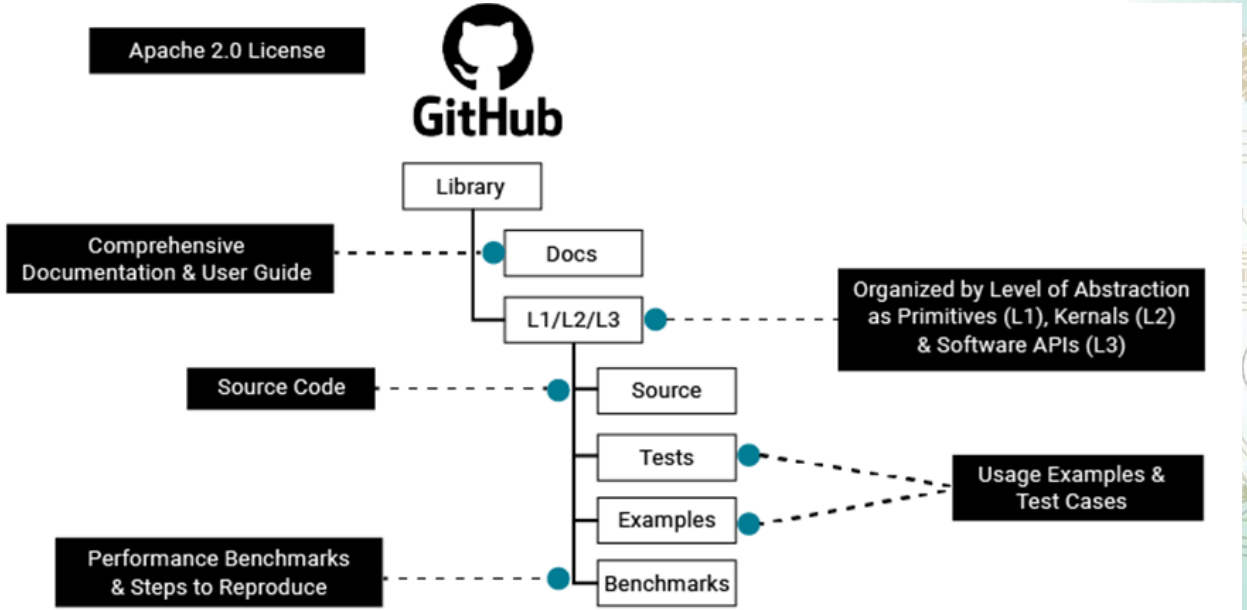


LIB_AES_ENCRYPT > hls_aes128_enc.cpp > ...

```
1  #include <ap_int.h>
2  #include <hls_stream.h>
3  #include <string.h>
4  #include "xf_security/ecb.hpp"
```

LIB_SHA256_HASH > hls_sha256.cpp > ...

```
1  #include <ap_int.h>
2  #include <hls_stream.h>
3  #include <stdint.h>
4  #include "xf_security/sha224_256.hpp"
```



Metodología:
FASE II – Implementación en VITIS HLS

```
// Depth ajustado para absorber latencias sin consumir mucha BRAM
#pragma HLS STREAM variable=plainStrm depth=64
#pragma HLS STREAM variable=cipherStrm depth=64
#pragma HLS STREAM variable=endPlainStrm depth=64
#pragma HLS STREAM variable=endCipherStrm depth=64
#pragma HLS STREAM variable=keyStrm depth=4
```

```
// Interfaces de Control (AXI Lite)
#pragma HLS INTERFACE s_axilite port=key_w0 bundle=control
#pragma HLS INTERFACE s_axilite port=key_w1 bundle=control
#pragma HLS INTERFACE s_axilite port=key_w2 bundle=control
#pragma HLS INTERFACE s_axilite port=key_w3 bundle=control
#pragma HLS INTERFACE s_axilite port=length_bytes bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

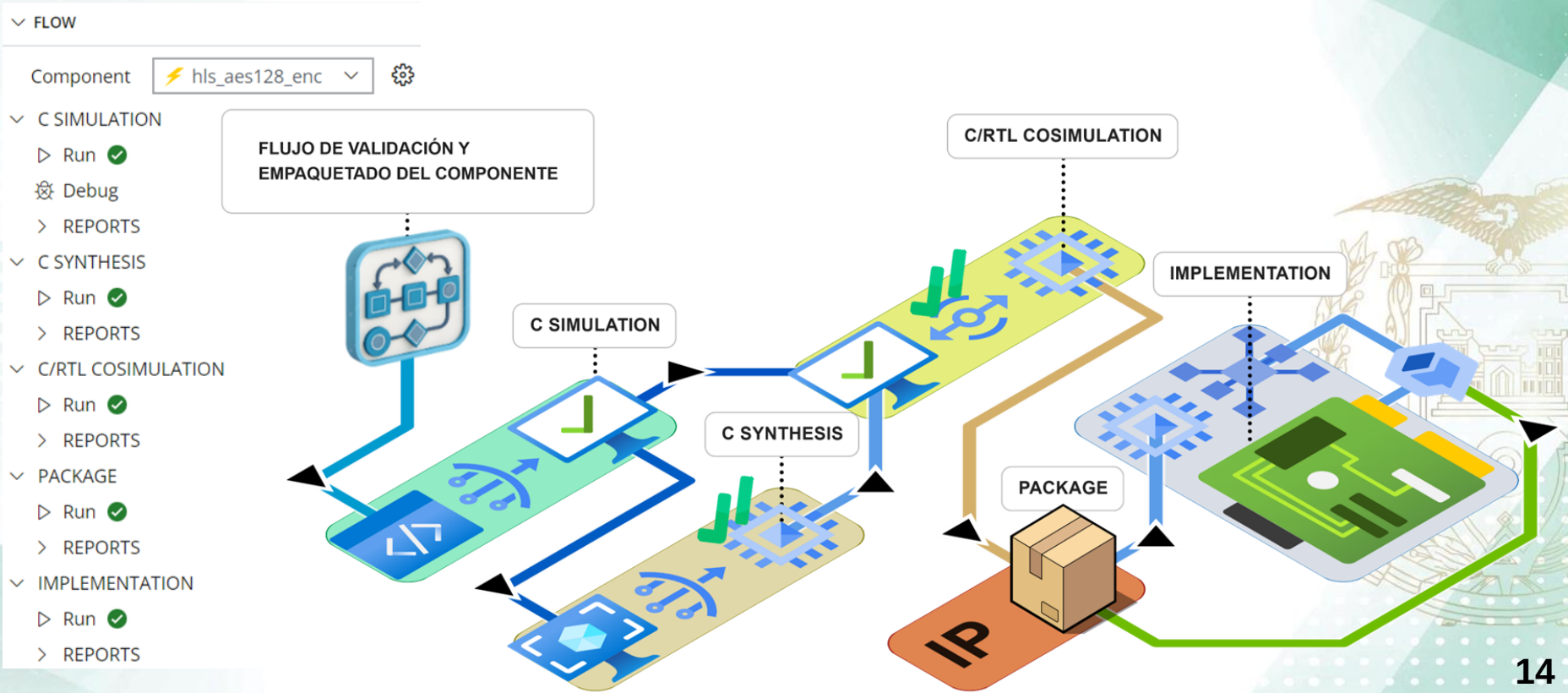
```
#pragma HLS PIPELINE II=1
            // Escribimos en Big Endian (estándar SHA)
            // range(high, low) selecciona 32 bits del total de 256
            out_mem[i] = digest.range(32 * (7 - i) + 31, 32 * (7 - i));
        }
        endf = end_strm.read();
    }
}
```

```
// Usamos LUTRAM (Registros) para ser rápidos y ahorrar BRAM.
#pragma HLS STREAM variable=len_strm      depth=16
#pragma HLS STREAM variable=end_len_strm  depth=16
#pragma HLS STREAM variable=hash_strm     depth=16
#pragma HLS STREAM variable=end_hash_strm depth=16
```

Test Bench

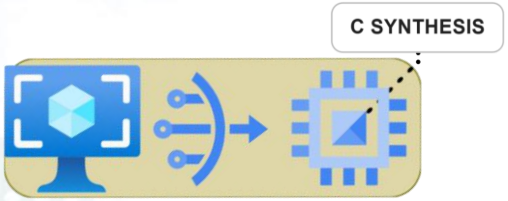
```
// Datos de prueba (Vector NIST)
// Plaintext: 6bc1bee22e409f96e93d7e117393172a
// Key:       2b7e151628aed2a6abf7158809cf4f3c
// Expected:  3ad77bb40d7a3660a89ecaf32466ef97

unsigned char test_bytes[16] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
                                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a};
```

Metodología:
FASE II – Flujo de Validación



Metodología:
FASE II – C Simulation & C Synthesis

C SIMULATION → La simulación en Vitis HLS verifica que el algoritmo descrito en C o C++ funciona correctamente antes de convertirlo en hardware.



AES-128

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	7.882 ns	2.70 ns

SHA-256

TARGET	ESTIMATED	UNCERTAINTY
20.00 ns	14.600 ns	5.40 ns

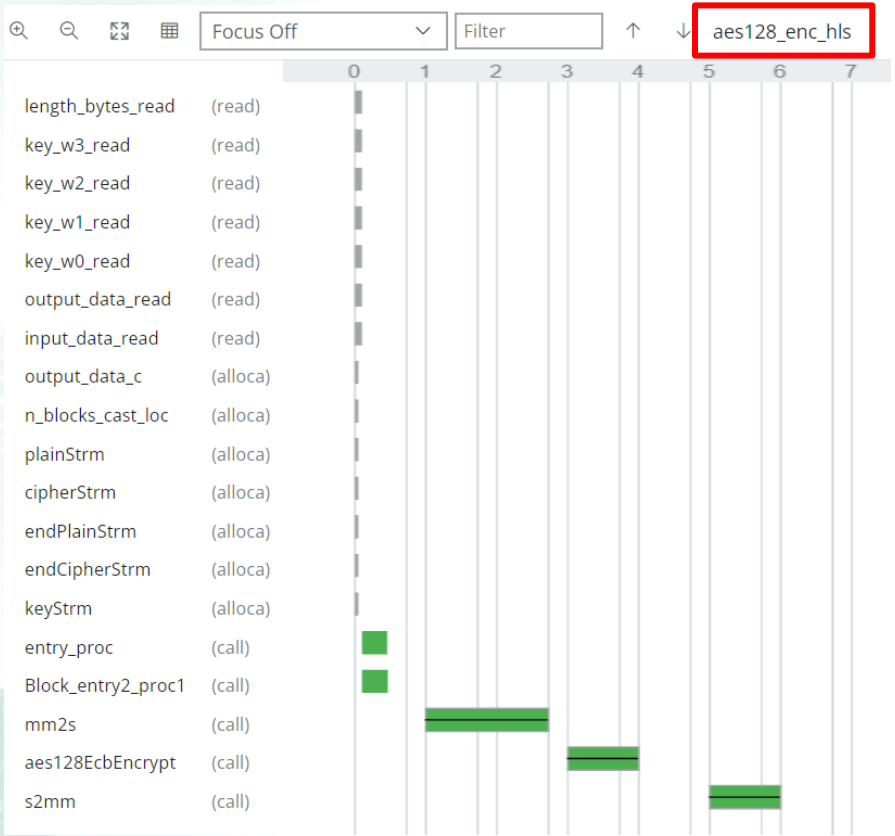
Performance & Resource Estimates

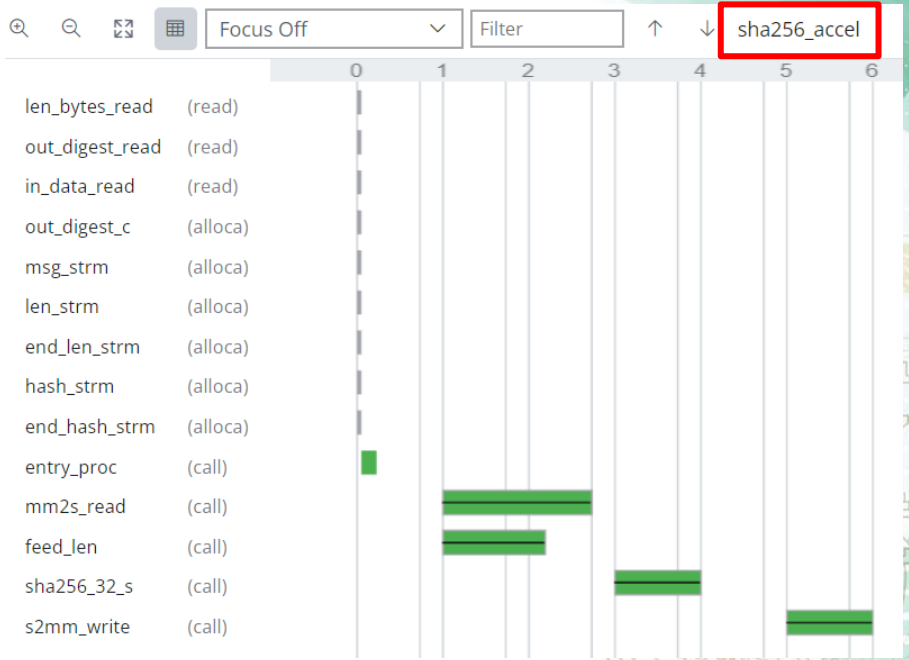
Modules Loops Hide empty columns

MODULES & LOOPS	ISSUE TYPE	SLACK	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
aes128_enc_hls (5)		-0.58	-	-	-	-	-	dataflow	52	0	7096	12523	0
entry_proc		-	0	0.0	-	0	-	no	0	0	3	29	0

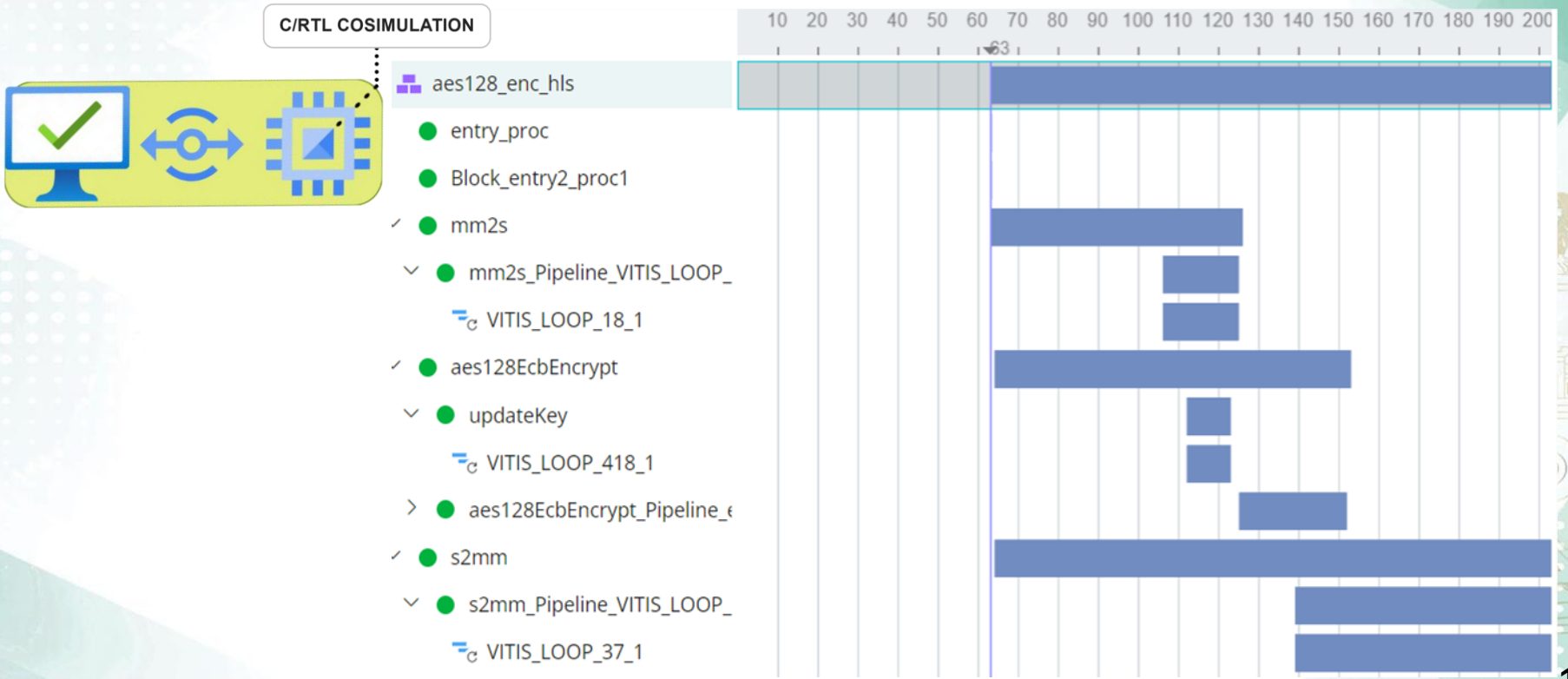
MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
sha256_accel (5)	-	-	-	-	-	dataflow	16	0	8761	14590	0
entry_proc	0	0.0	-	0	-	no	0	0	3	29	0

Metodología:
FASE II – C Simulation & C Synthesis





Metodología:
FASE II – C/RTL Co-Simulation

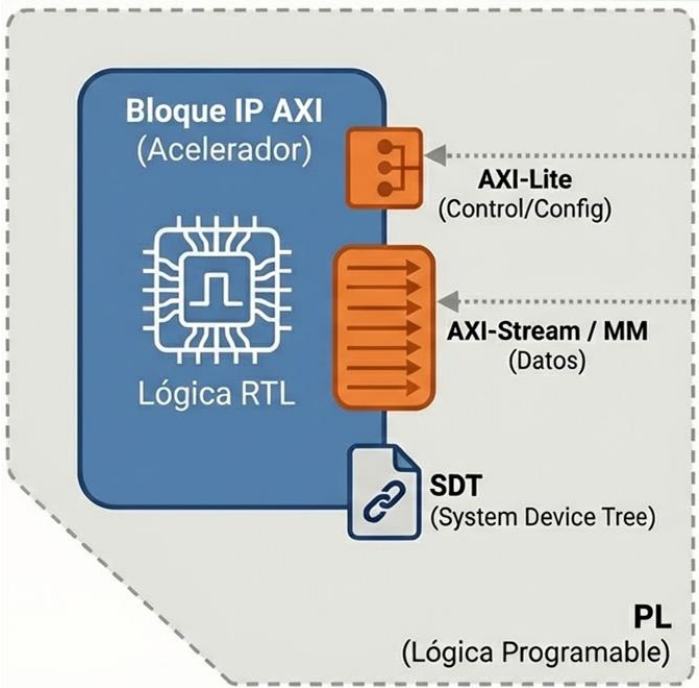


Metodología:
FASE II – Package

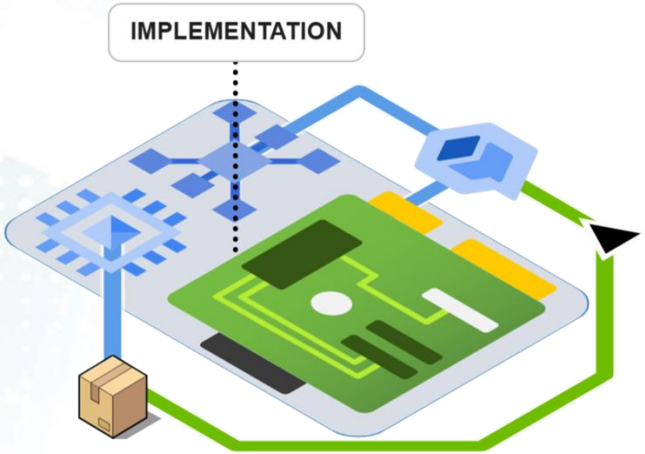
Un IP Core es un bloque hardware reutilizable que implementa una función específica dentro de la FPGA.

- Implementado en lógica digital (RTL)
- Integrable mediante interfaces estándar AXI
- Puede funcionar como acelerador hardware

No es software: es un circuito digital real dentro del chip



Metodología:
FASE II - Implementation



AES-128 Encrypt

Resource Usage

NAME	VERILOG
SLICE	2631
LUT	9493
FF	5379
DSP	0
BRAM	22

SHA-256

Resource Usage

NAME	VERILOG
SLICE	2589
LUT	5562
FF	8528
DSP	0
BRAM	9

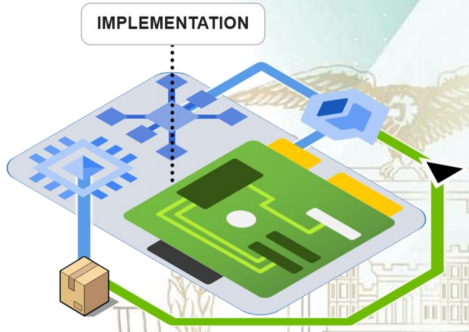
Costo del Rendimiento:
Uso de Recursos en AES-128

Recurso / AES-128 Encrypt	Vitis Library IP	Custom IP
LUT (Logic)	9493	7018
FF (Flip-Flops)	5379	6275
BRAM (Memory)	22	74

AES-128 Encrypt

Resource Usage

NAME	VERILOG
SLICE	2631
LUT	9493
FF	5379
DSP	0
BRAM	22



Análisis: El aumento significativo en BRAM en el diseño Custom fue una *decisión arquitectónica deliberada* para implementar buffers internos más grandes, lo que reduce drásticamente el overhead de comunicación con el bus AXI.

Manejo e Integración del Hardware

Vivado Design Suite

Integración del IP generado con el Zynq Processing System.

Generación del Bitstream final (.bit) y archivo de plataforma hardware (.xsa)

PetaLinux

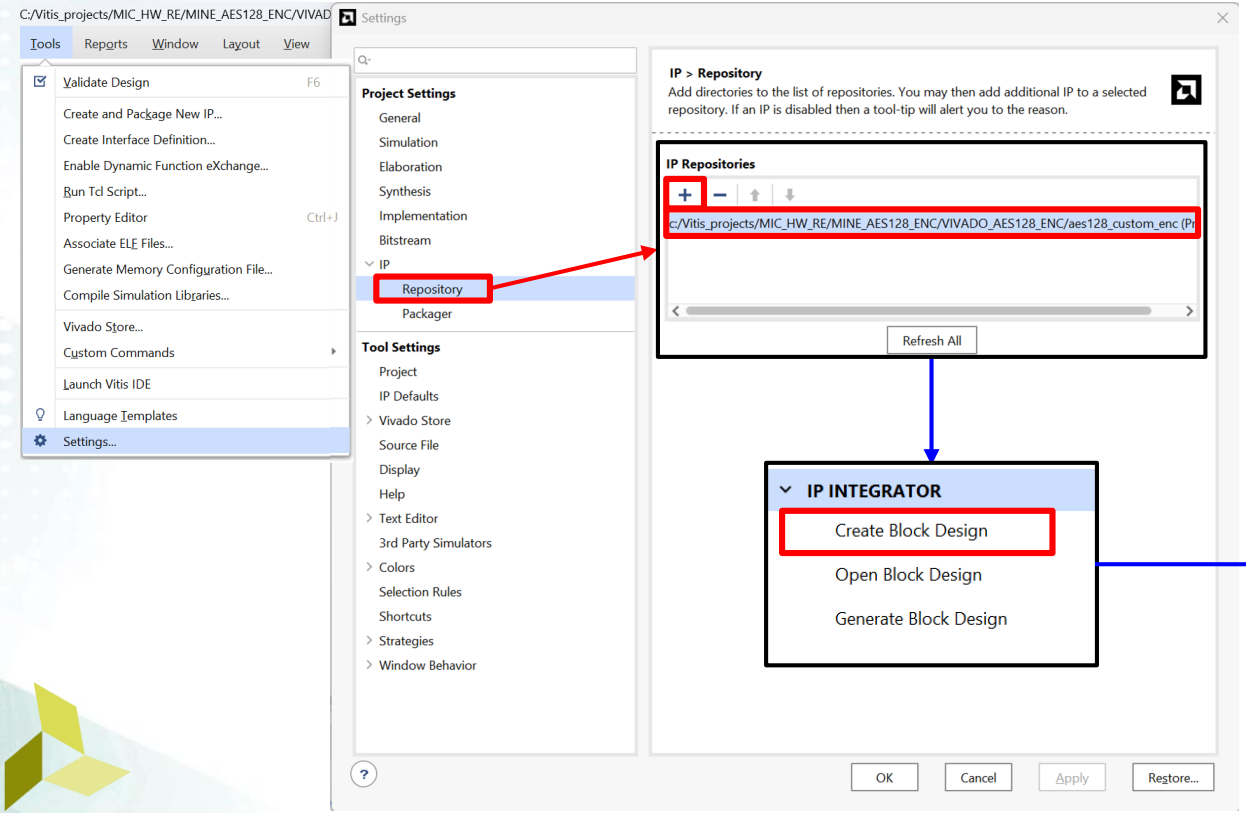
Construcción de la plataforma de desarrollo.

Configuración del Device Tree para reconocer los aceleradores en el PL.

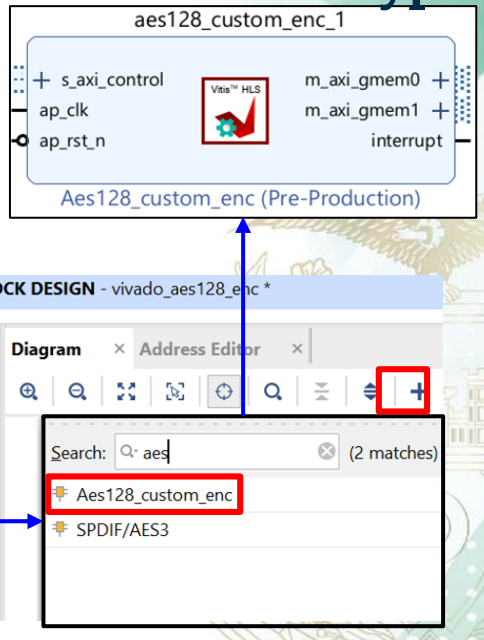
Drivers UIO

Implementación de drivers udmabuf y UIO para acceso directo a memoria desde espacio de usuario.

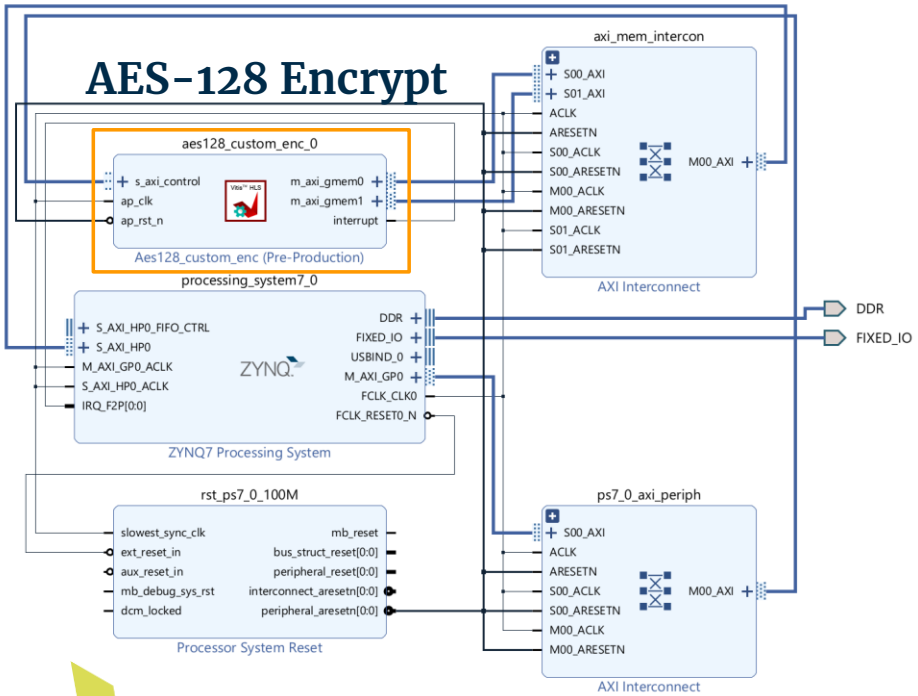
Metodología:
FASE III – Implementacion de la IP en Vivado

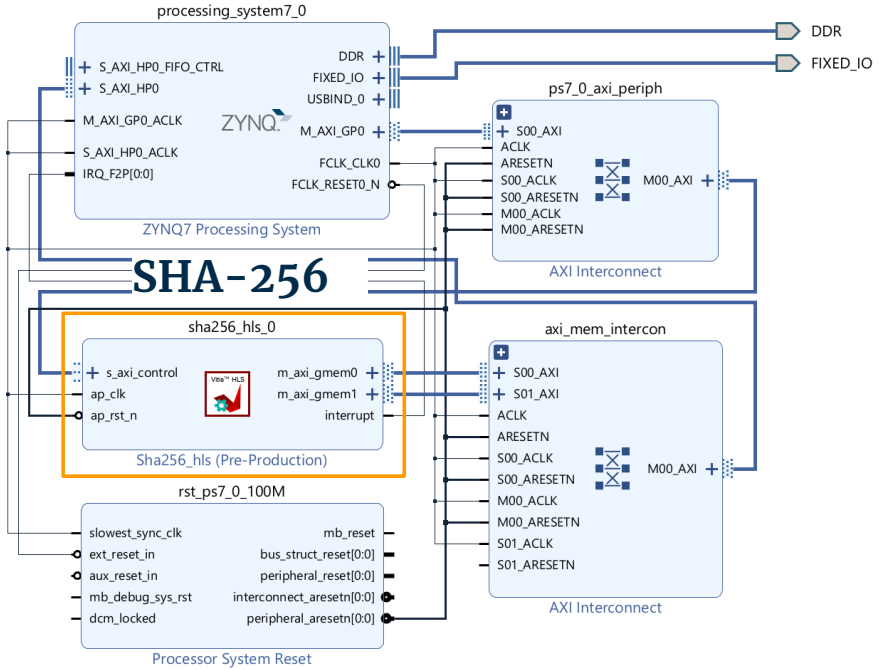


AES-128 Encrypt

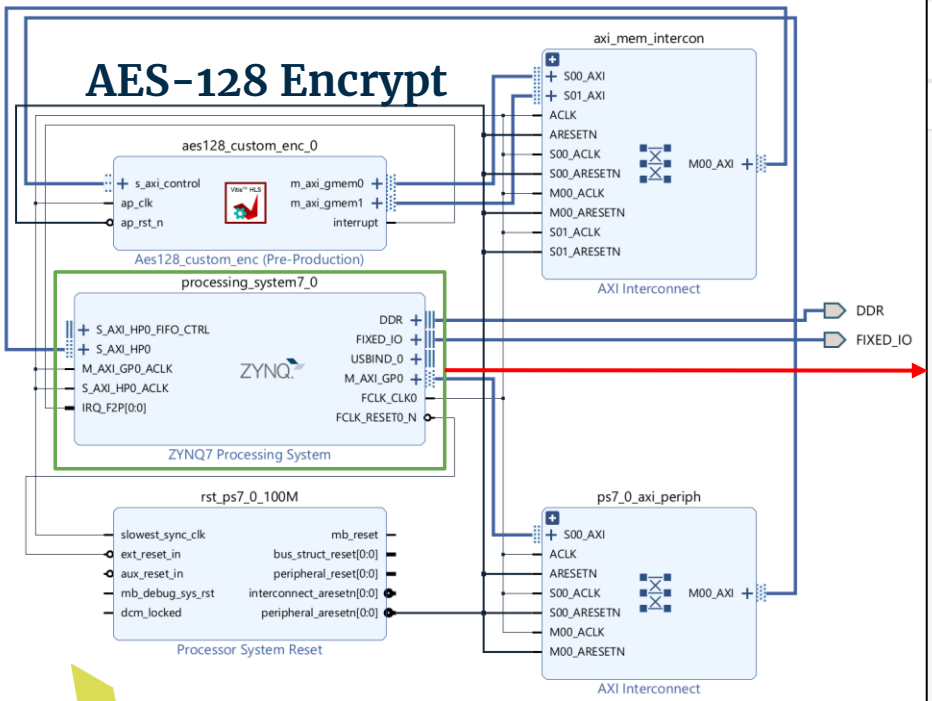


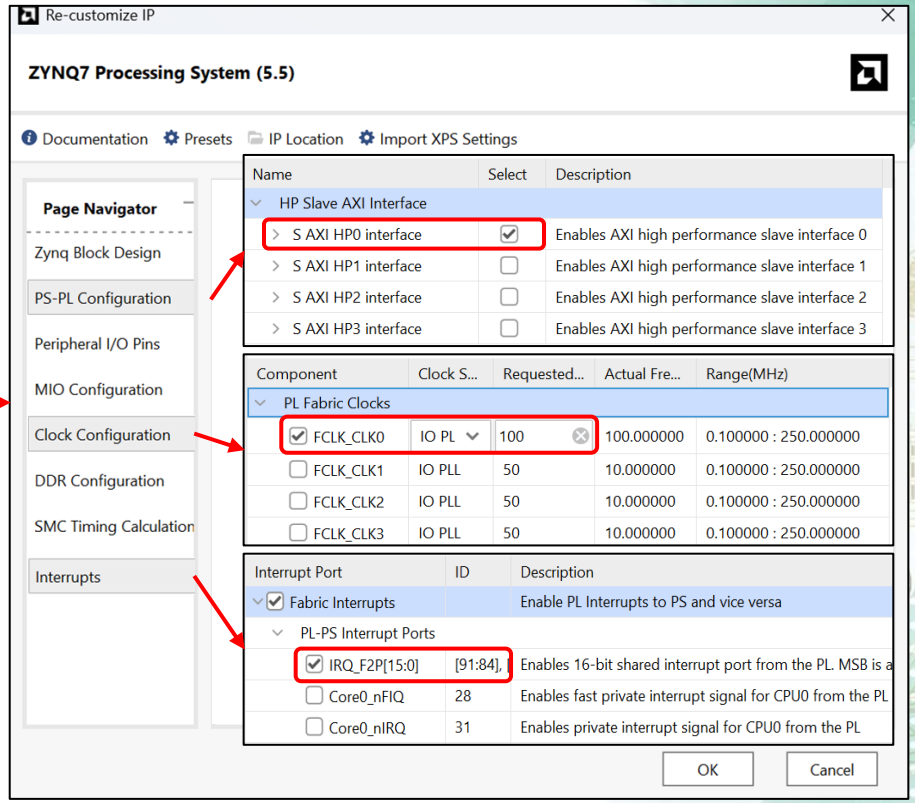
Metodología:
FASE III – IPs Core Implementadas en su Hardware



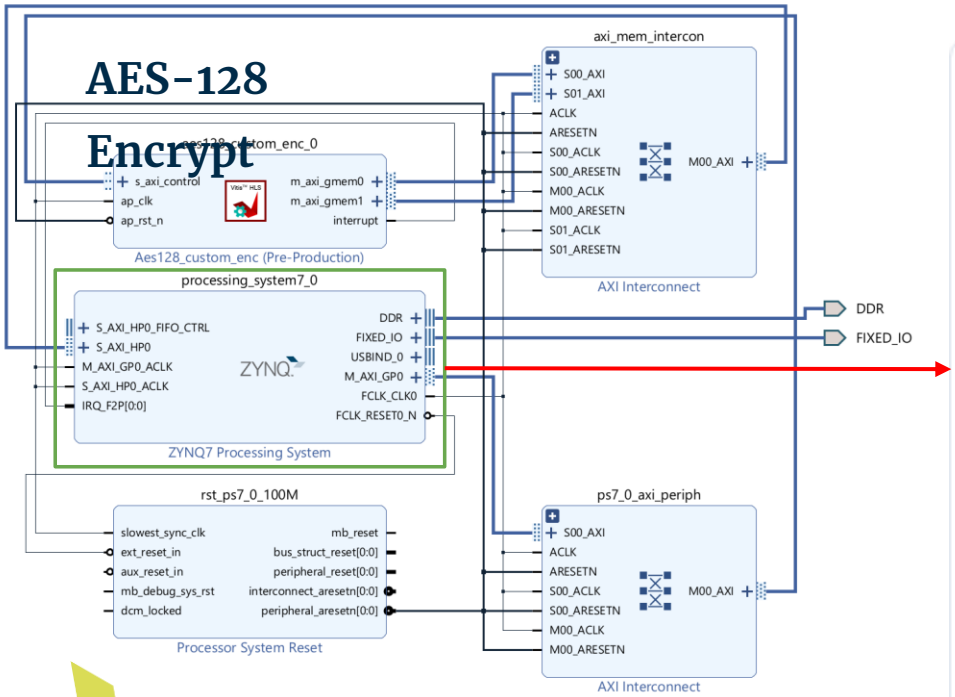


Metodología:
FASE III – Configuración del Zynq PS





Metodología:
FASE III - Comunicación AXI



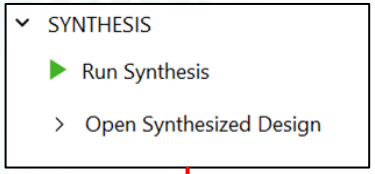
AXI-Lite:

Canal de Bajo ancho de banda para el **control, configuración de registros y estado de los aceleradores**.

AXI HP (High Performance):

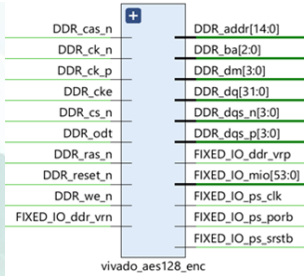
Canal de Alto rendimiento para la **transferencia masiva de datos** (DMA) directamente hacia/desde la memoria DDR.

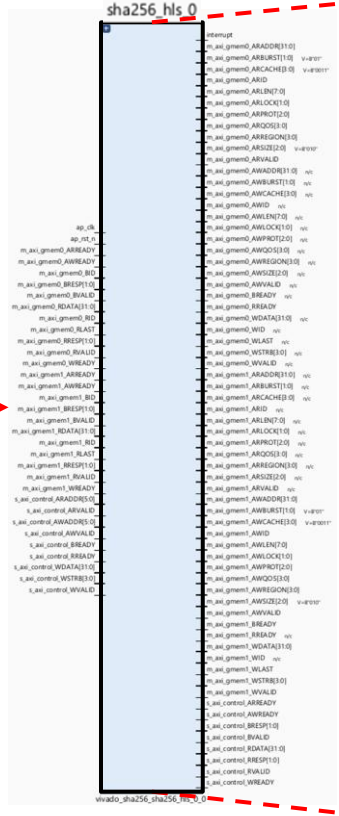
Metodología:
FASE III - SYNTHESIS DEL IP SHA-256

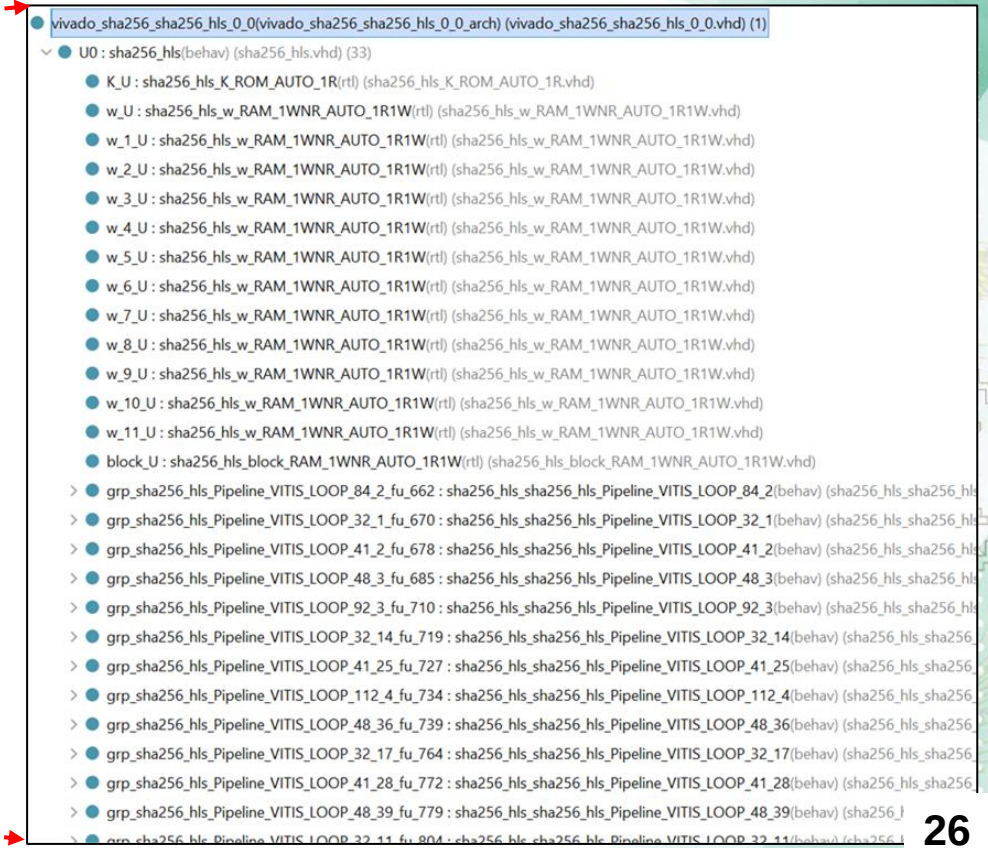


SHA-256

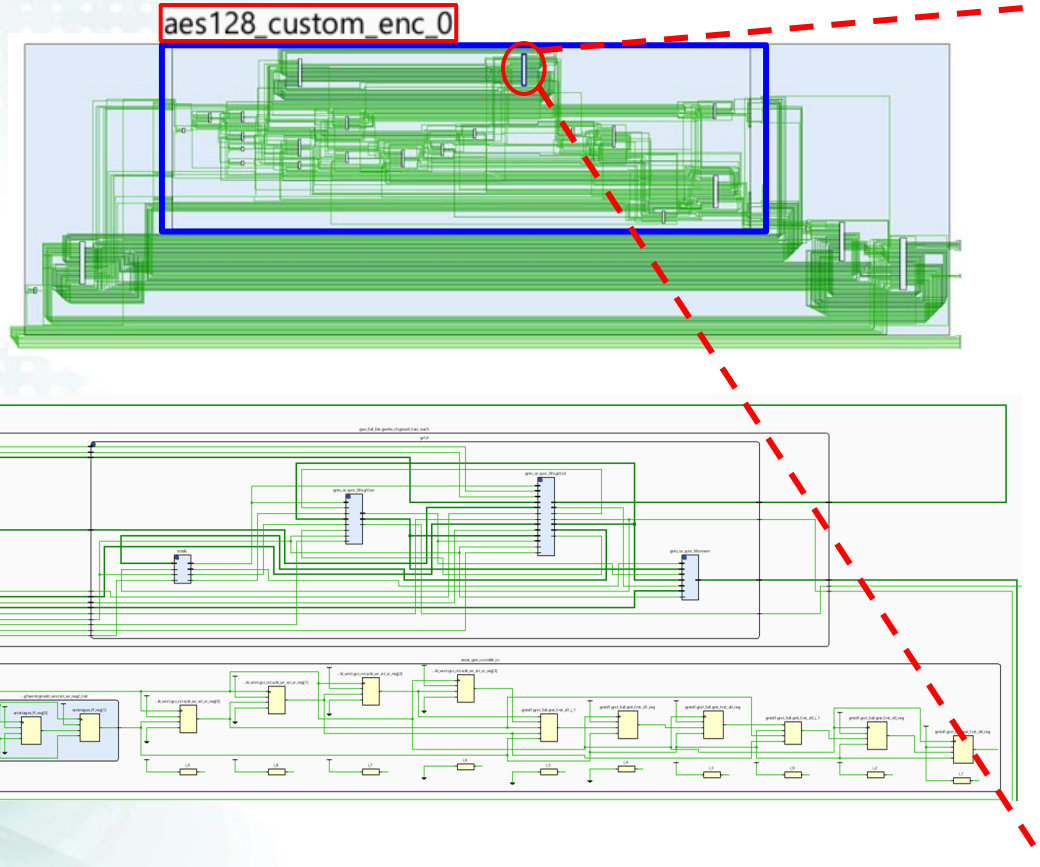
AES-128 Encrypt







Metodología:
FASE III - Vivado RTL del AES-128 Encrypt



Metodología:
FASE III - Vivado VHDL - AES-128 Encrypt

aes128_custom_enc.vhd | vivado_aes128_enc_aes128_custom_enc_0_0.vhd | aes128_custom_enc_aes_process.vhd

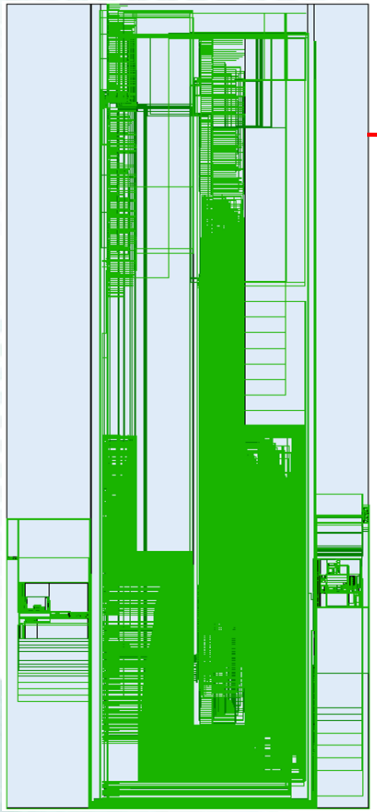
Sources | Source File Properties

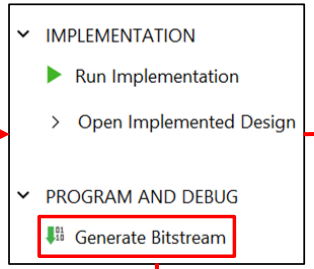
_projects/MIC_HW_RE/MINE_AES128_ENC/VIVADO_AES128_ENC/vivado_aes128_enc/vivado_aes128_enc.gen/sources_1/bd/vivado_aes128_enc/ipshared/326c/hdl/vhdl/aes128_custom_enc_aes_process.vhd

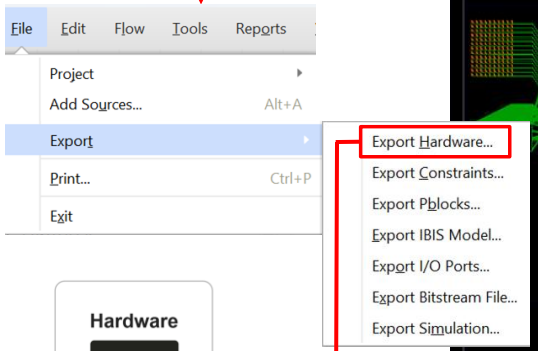
Read-only

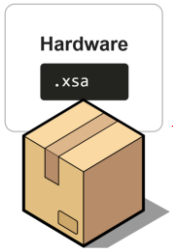
```
7340        else
7341            sbox_ce40 <= ap_const_logic_0;
7342        end if;
7343    end process;
7344
7345
7346    sbox_ce41_assign_proc : process(grp_aes_process_Pipeline_VITIS_LOOP_126_2_fu_1903_sbox_ce41, ap_CS_fsm_state15)
7347    begin
7348        if ((ap_const_logic_1 = ap_CS_fsm_state15)) then
7349            sbox_ce41 <= grp_aes_process_Pipeline_VITIS_LOOP_126_2_fu_1903_sbox_ce41;
7350        else
7351            sbox_ce41 <= ap_const_logic_0;
7352        end if;
7353    end process;
7354
7355
7356    sbox_ce42_assign_proc : process(grp_aes_process_Pipeline_VITIS_LOOP_126_2_fu_1903_sbox_ce42, ap_CS_fsm_state15)
7357    begin
7358        if ((ap_const_logic_1 = ap_CS_fsm_state15)) then
7359            sbox_ce42 <= grp_aes_process_Pipeline_VITIS_LOOP_126_2_fu_1903_sbox_ce42;
7360        else
7361            sbox_ce42 <= ap_const_logic_0;
7362        end if;
7363    end process;
7364
7365
7366    sbox_ce43_assign_proc : process(grp_aes_process_Pipeline_VITIS_LOOP_126_2_fu_1903_sbox_ce43, ap_CS_fsm_state15)
7367    begin
7368        if ((ap_const_logic_1 = ap_CS_fsm_state15)) then
7369            sbox_ce43 <= grp_aes_process_Pipeline_VITIS_LOOP_126_2_fu_1903_sbox_ce43;
```

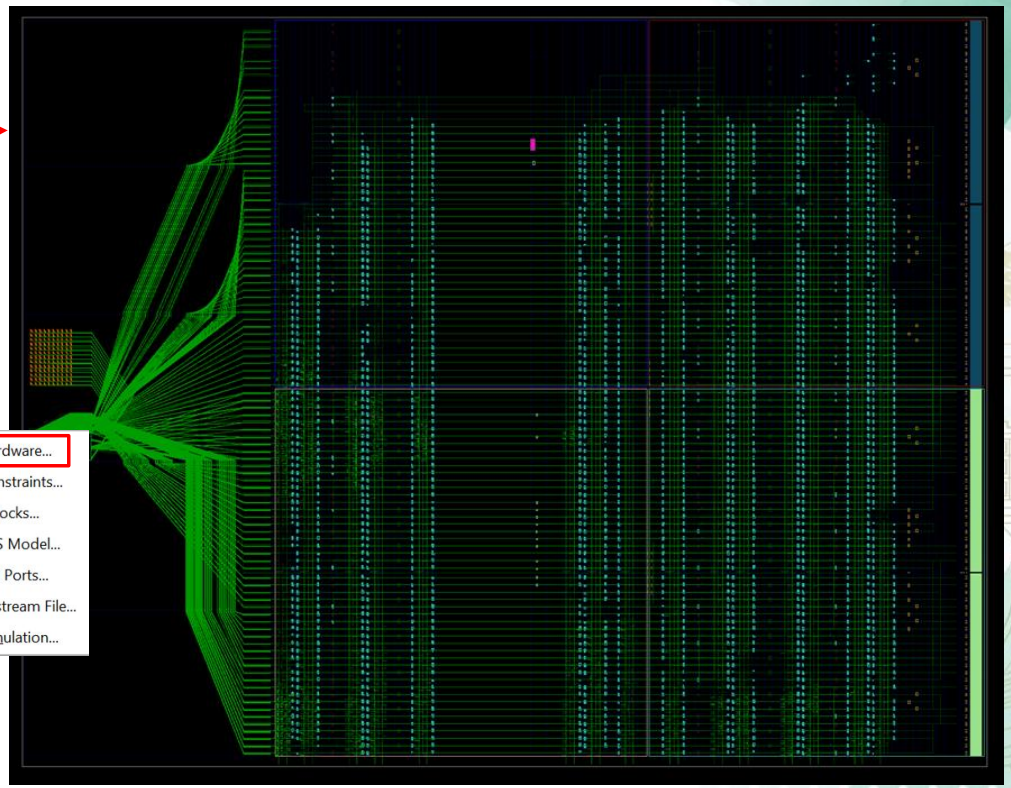
Metodología:
FASE III- IMPLEMENTATION y Bitstream



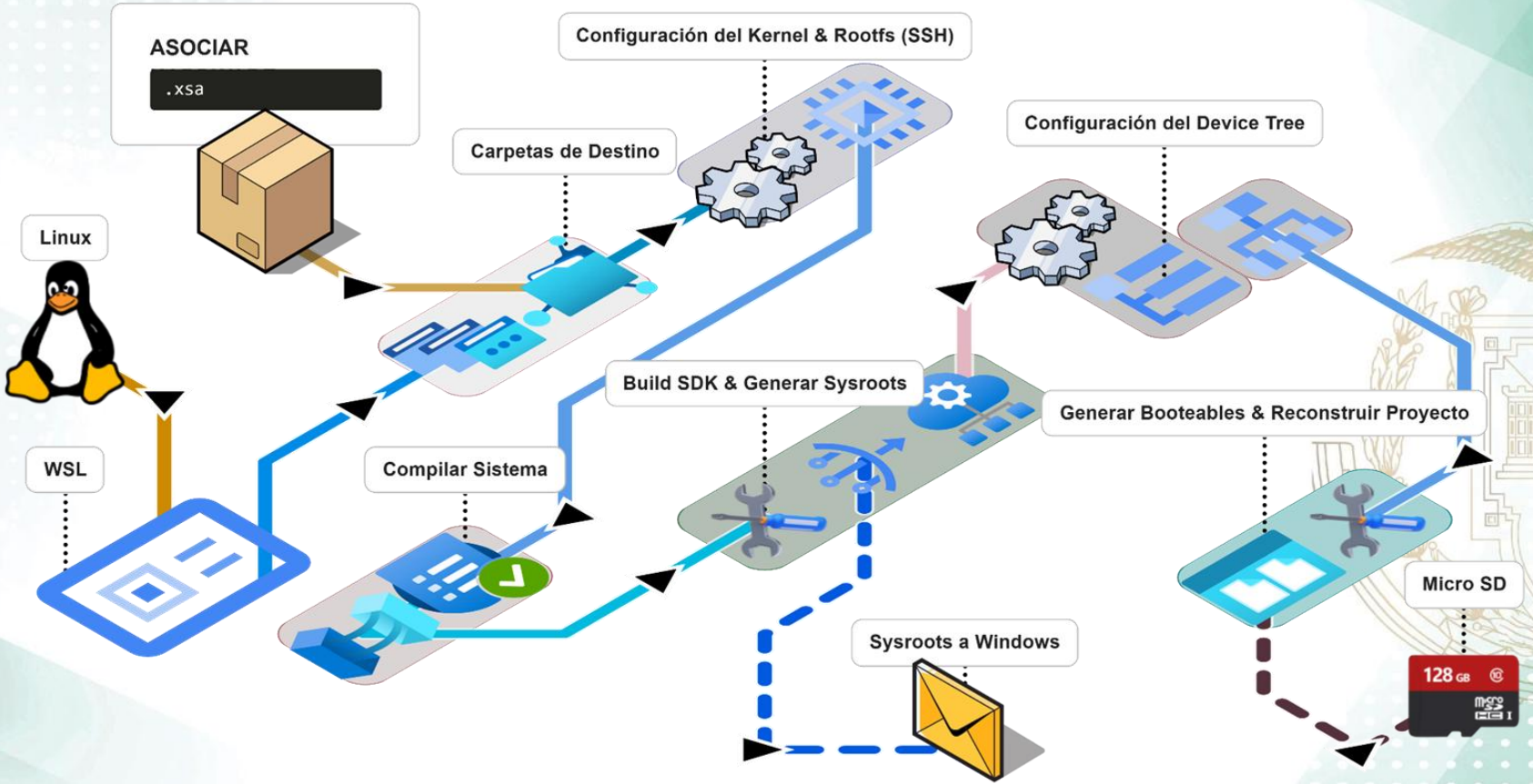




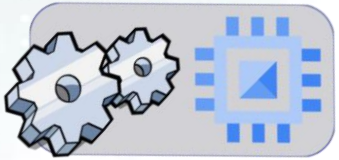




Metodología:
FASE IV – Linux Embebido – WSL



Metodología:
FASE IV – Configuración del kernel – Rootfs (SSH)

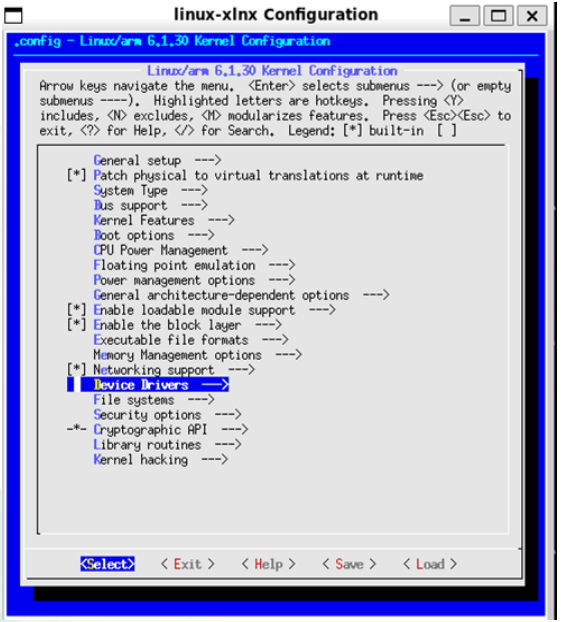


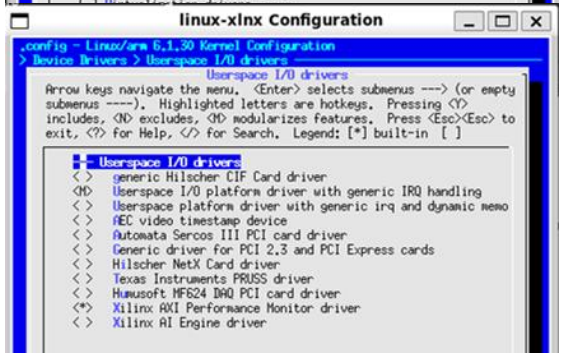
CONFIGURAR KERNEL Y ROOTFS (SSH, LOGIN ROOT, SD)

```
cd ~/petalinux_acelerador_projects/hw_aes128_enc/aes128_enc_linux
petalinux-config -c kernel
```

Configurar rootfs y resolver conflicto openssh

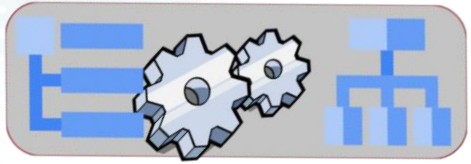
```
petalinux-config -c rootfs
```

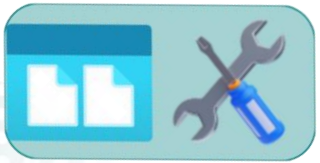





```
[*] openssh
[ ] openssh-misc
[ ] openssh-dbg
[*] openssh-sshd
[*] openssh-keygen
[ ] openssh-ssh
[ ] openssh-dev
[*] openssh-sftp
[*] openssh-sftp-server
[ ] openssh-scp
```

Metodología:
FASE IV – Configuración del Buffer, el SDT y Generar Booteables





Crear el módulo (Buffer)

```
cd ~/petalinux_acelerador_projects/hw_aes128_enc/aes128_enc_linux
petalinux-create -t modules --name u-dma-buf --enable
```

Sobre escribir el Buffer

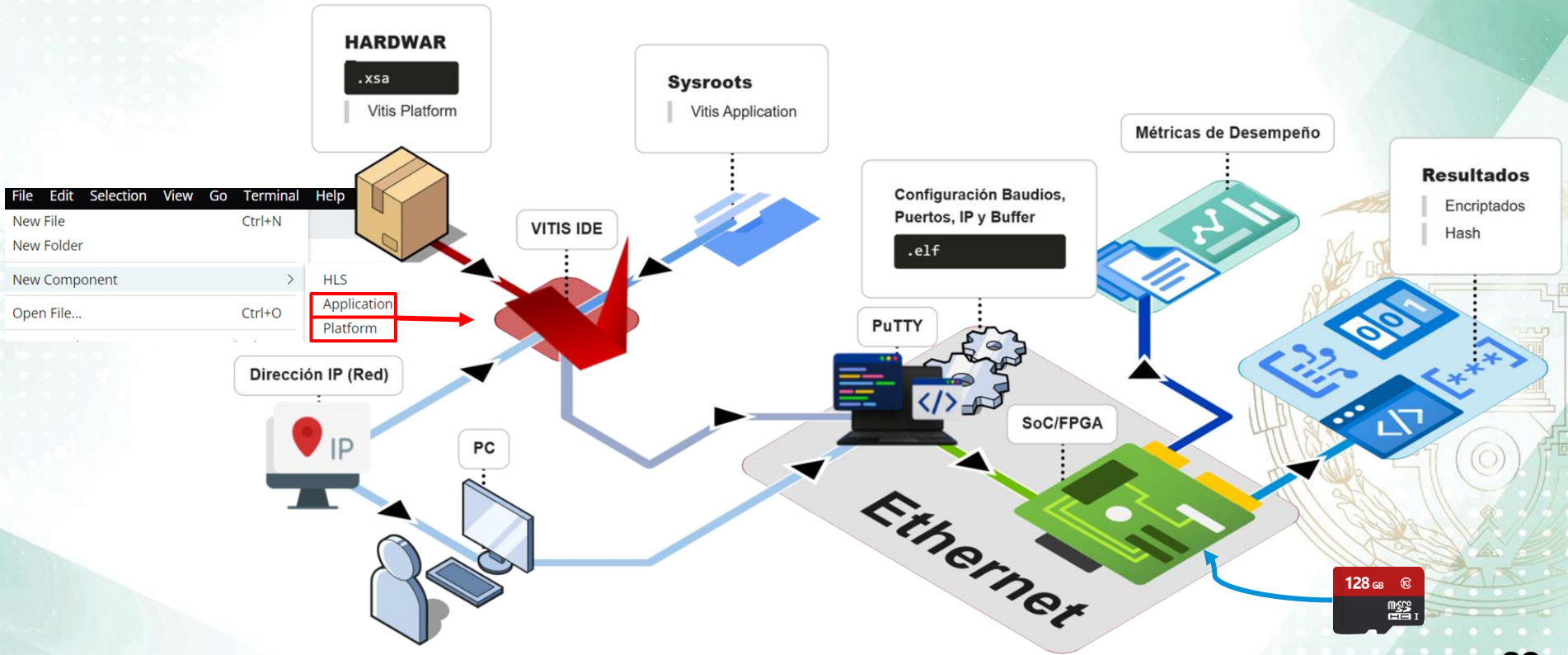
Vamos a usar el comando `cat` para sobre escribir el archivo `u-dma-buf.bb` con la configuración correcta que descarga el driver desde GitHub.

```
cat <<EOF > project-spec/meta-user/recipes-modules/u-dma-buf/u-dma-buf.bb
SUMMARY = "Recipe for building an external u-dma-buf Linux kernel module"
SECTION = "PETALINUX/modules"
LICENSE = "BSD-2-Clause"
LIC_FILES_CHKSUM = "file://LICENSE;md5=bebf0492502927bef0741aa04d1f35f5"

inherit module

SRC_URI = "git://github.com/ikwzm/udmabuf.git;protocol=https;branch=master"
SRCREV = "\${AUTOREV}"

S = "\${WORKDIR}/git"

RPROVIDES:\${PN} += "u-dma-buf"
EOF
```

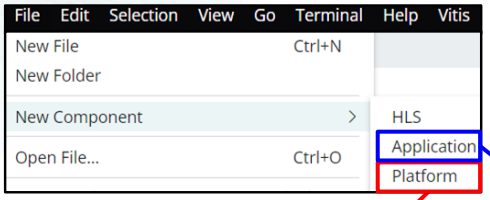
Reconstruimos el divece tree

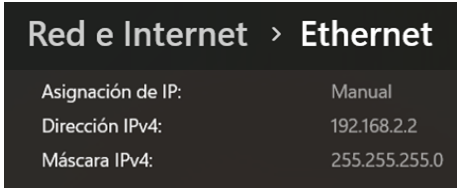
```
petalinux-build -c device-tree -x clean
petalinux-build -c device-tree
petalinux-build
petalinux-package --boot --force \
--fsbl images/linux/zynq_fsbl.elf \
--fpga images/linux/system.bit \
--u-boot
```

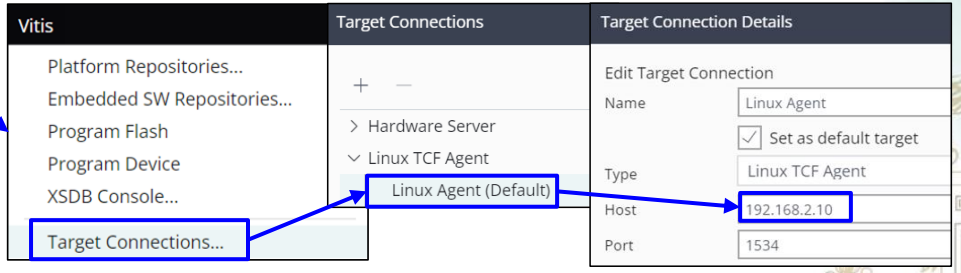
Metodología:
FASE IV - Implementación en Vitis IDE & PuTTY



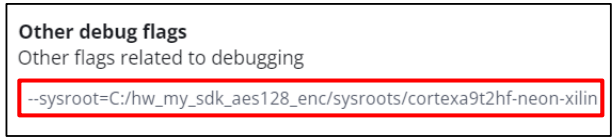
Metodología:
FASE IV – Vitis IDE

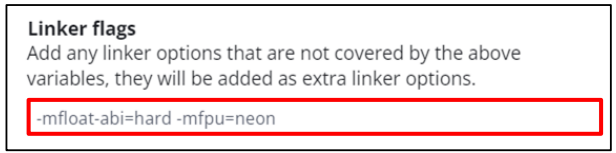






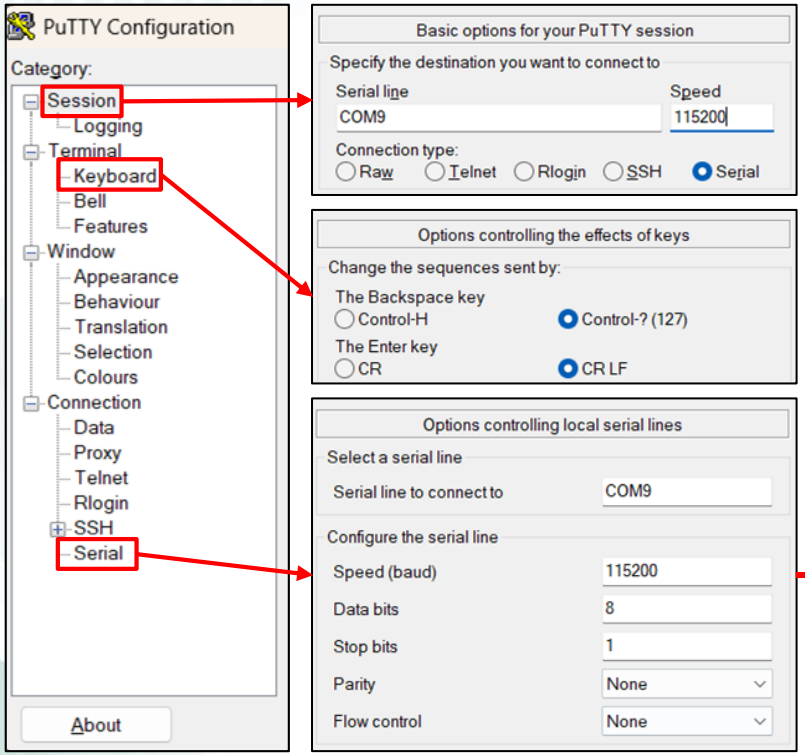
VITIS IDE






```
// Offsets de registros AXI-Lite del IP (según Vitis HLS).
#define REG_CTRL        0x00
#define REG_GIE         0x04
#define REG_IER         0x08
#define REG_ISR         0x0C

#define REG_PTR_INPUT   0x10
#define REG_PTR_OUTPUT  0x18
#define REG_KEY_W0      0x20
#define REG_KEY_W1      0x28
#define REG_KEY_W2      0x30
#define REG_KEY_W3      0x38
#define REG_LEN         0x40
```

Metodología:
FASE IV – Configuración en el PuTTY



Configuración en el PuTTY

```
sudo ifconfig eth0 192.168.2.10 netmask 255.255.255.0 up
```

UDMABUF + UIO

```
sudo su
cd /lib/modules/$(uname -r)/

# Cargar udmabuf de 8MB (cambia si quieres 1MB: 1048576)
rmmod u_dma_buf 2>/dev/null
insmod ./extra/u-dma-buf.ko udmabuf0=8388608
chmod 666 /dev/udmabuf0      # permisos
```

```
# Cargar UIO y bindear la IP
modprobe uio_pdrv_genirq 2>/dev/null || insmod ./kernel/drivers/uio/uio_pdrv_genirq.ko

# Confirma el device (cambia 40000000.aes128_enc_hls si tu base cambia)
ls -l /sys/bus/platform/devices | grep -i 40000000
ls -l /sys/bus/platform/devices | grep -i sha

# Fuerza override y bind
echo "" > /sys/bus/platform/devices/40000000.aes128_enc_hls/driver_override
echo uio_pdrv_genirq > /sys/bus/platform/devices/40000000.aes128_enc_hls/driver_override
echo 40000000.aes128_enc_hls > /sys/bus/platform/drivers/uio_pdrv_genirq/bind

chmod 666 /dev/uio0        # permisos
```

Ejecución y Envío de Resultados

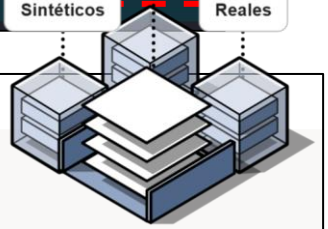
```c
int main() {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) return 1;

    // Mapeo de la IP
    volatile uint32_t *ip = mmap(NULL, IP_RANGE, PROT_READ|PROT_WRITE, MAP_SHARED, fd,

    // Obtener datos de udmabuf
    FILE *f_addr = fopen("/sys/class/u-dma-buf/udmabuf0/phys_addr", "r");
    uint64_t phys_addr;
    fscanf(f_addr, "%llx", &phys_addr);
    fclose(f_addr);

    int fd_udma = open("/dev/udmabuf0", O_RDWR | O_SYNC);
    uint8_t *buffer = mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_SHARED, fd_udma, 0);

    // Preparar mensaje
```

OUTPUT × DEBUG CONSOLE × PROBLEMS 1 ×

```
Uso: /home/petalinux/app_sha256.elf
```

DATASETS

Sintéticos Reales



Cargar los Datasets desde el WSL y Ejecutar

```
ssh-keygen -R 192.168.2.10

scp /mnt/c/Vitis_projects/Datasets/Pruebas/dataset_test_4MB.bin petalinux@192.168.2.10:/home/
petalinux/datasets/
```

Correr programa

```
../app_sha256.elf dataset_test_4KB.bin
```

Traer los Resultados de la FPGA a tu WSL

```
scp petalinux@192.168.2.10:/home/petalinux/datasets
/custom_metricas_sha256_fpga.csv
~/metricas_rendimiento/
```

Pasarlo de WSL a Windows

```
cd ~/metricas_rendimiento
cp custom_metricas_sha256_fpga.csv /mnt/c
/Vitis_projects/Metricas_Rendimiento/FPGA_Mine/
cd
```

Métricas Clave de Rendimiento

Tiempo End-to-End (E2E)

Visión real del usuario desde Linux. Incluye el tiempo de procesamiento + el overhead de movimiento de datos (DMA).

Throughput Efectivo

Velocidad de procesamiento real en

$$T = \frac{\text{Data Size}}{\text{Time}}$$

Eficiencia de Recursos

Relación costo-beneficio. Consumo de LUTs, BRAM y Flip-Flops vs. la aceleración obtenida.

Reglas de Comparación:

- **Misma Plataforma:** Misma placa Zynq-7000.
- **Mismos Datasets**
- **Escenarios Reales:** Incluyendo latencia de transferencia de datos.

Resultados AES 128

COM9 - PuTTY

```
my_aes128_enc_linux:/home/petalinux/datasets# ls -l
-rwxr-xr-x    1 petalinu petalinu  46192724 Mar  9 12:40 Vitis_Unified_IDE_Reference_Manu
-rw-r--r--    1 root     root      46192736 Mar  9 12:42 custom_Vitis_Unified_IDE_Referen
-rw-r--r--    1 root     root           277 Mar  9 12:42 custom_metricas_aes128_enc_fpga.c
my_aes128_enc_linux:/home/petalinux/datasets#
my_aes128_enc_linux:/home/petalinux/datasets#
ence_Manual_ENC.pdf:/home/petalinux/datasets# cat custom_Vitis_Unified_IDE_Refere
```

custom_dataset_test_256MB_ENC.bin (0 ~ 5MB)

Resultados SHA 256

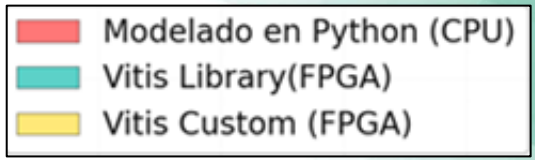
```
==========================================================
        EJECUTANDO SHA-256 EN FPGA (Zynq PL)
==========================================================
Archivo           : dataset_test_4MB.bin
Tamano procesado  : 4.00 MB (4194304 bytes)
----------------------------------------------------------

==========================================================
Hash (SHA-256)      d9b5532296bd468e570ad278519ded700f7bc9d34496e89fb92217265901240b5
==========================================================
METRICAS DE RENDIMIENTO (SHA-256 - FPGA)
==========================================================
[1] Tiempo total End-to-End                 : 0.315899457 s
[2] Throughput End-to-End                   : 12.66 MB/s
----------------------------------------------------------
[3] Tiempo de CPU del proceso (ARM)         : 0.315777000 s
[4] Uso de CPU del proceso (ARM)            : 99.96 %
----------------------------------------------------------
[5] Tiempo de ejecucion del acelerador (PL) : 0.266740251 s
[6] Throughput del acelerador (PL)          : 15.00 MB/s
----------------------------------------------------------
[7] Tiempo de transferencia de datos        : 0.048399792 s
[8] Overhead de control y sincronizacion    : 0.000015768 s
==========================================================

>> Metricas agregadas a: metricas_sha256_fpga.csv
```

```c
int main() {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) return 1;

    // Mapeo de la IP
    volatile uint32_t *ip = mmap(NULL, IP_RANGE, PROT_READ|PROT_WRITE, MAP_SHARED, fd,

    // Obtener datos de udmabuf
    FILE *f_addr = fopen("/sys/class/u-dma-buf/udmabuf0/phys_addr", "r");
    uint64_t phys_addr;
    fscanf(f_addr, "%llx", &phys_addr);
    fclose(f_addr);

    int fd_udma = open("/dev/udmabuf0", O_RDWR | O_SYNC);
    uint8_t *buffer = mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_SHARED, fd_udma, 0);

    // Preparar mensaje
```

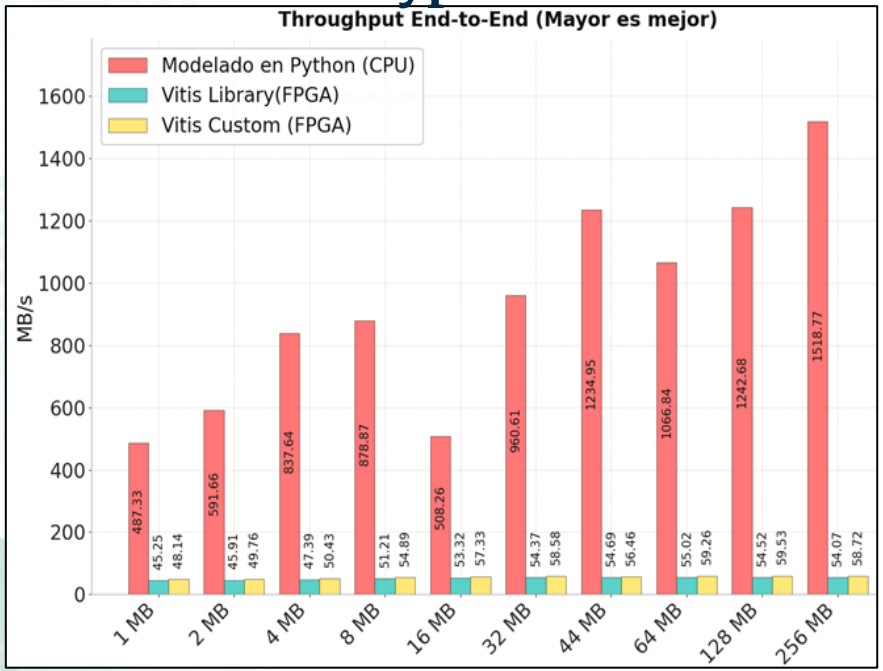
OUTPUT × DEBUG CONSOLE × PROBLEMS 1 ×

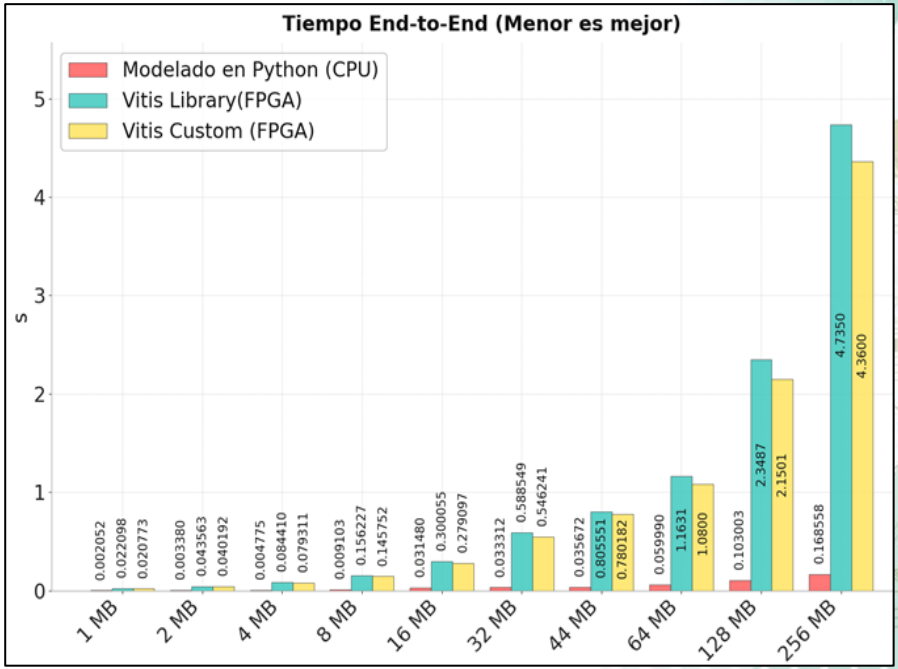
```
Uso: /home/petalinux/app_sha256.elf
```

Resultados AES-128:
Comparativa de Rendimiento

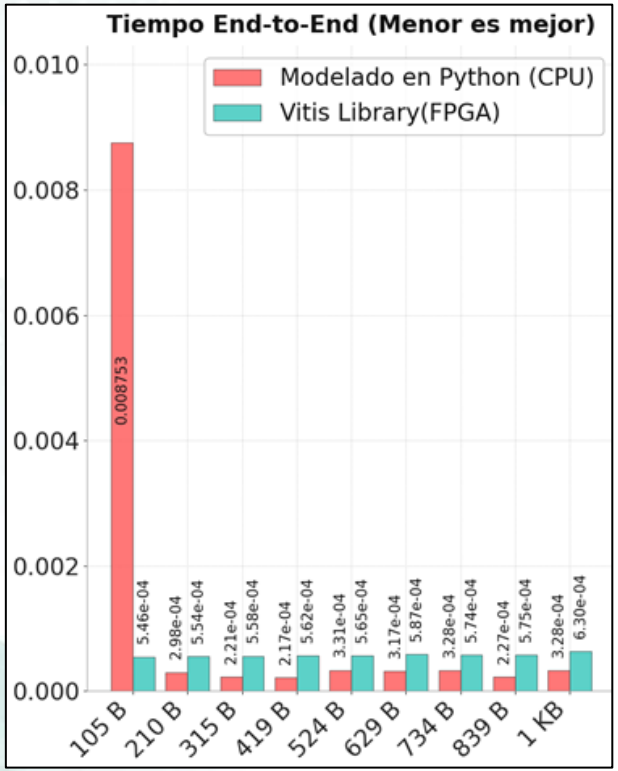


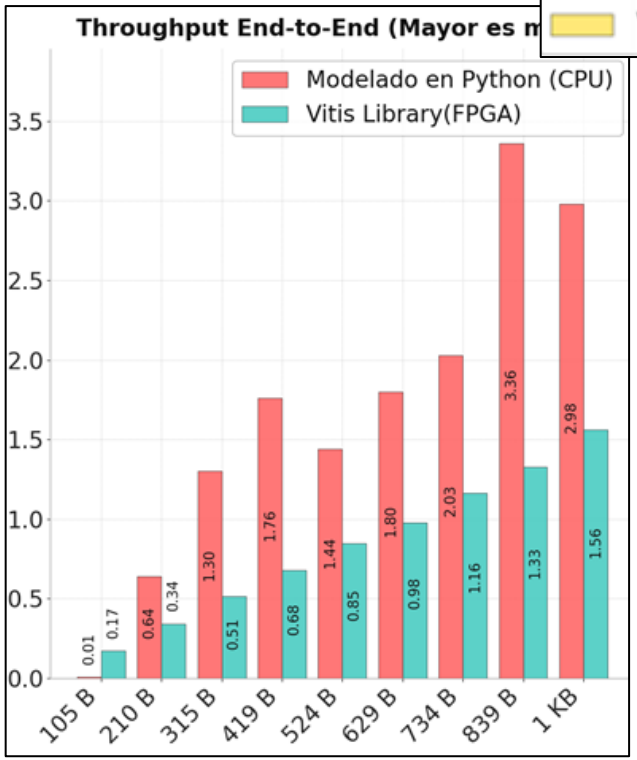
AES-128 Encrypt

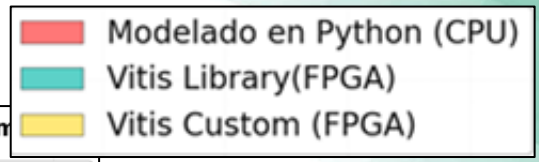




Resultados SHA-256:
Comparativa de Rendimiento

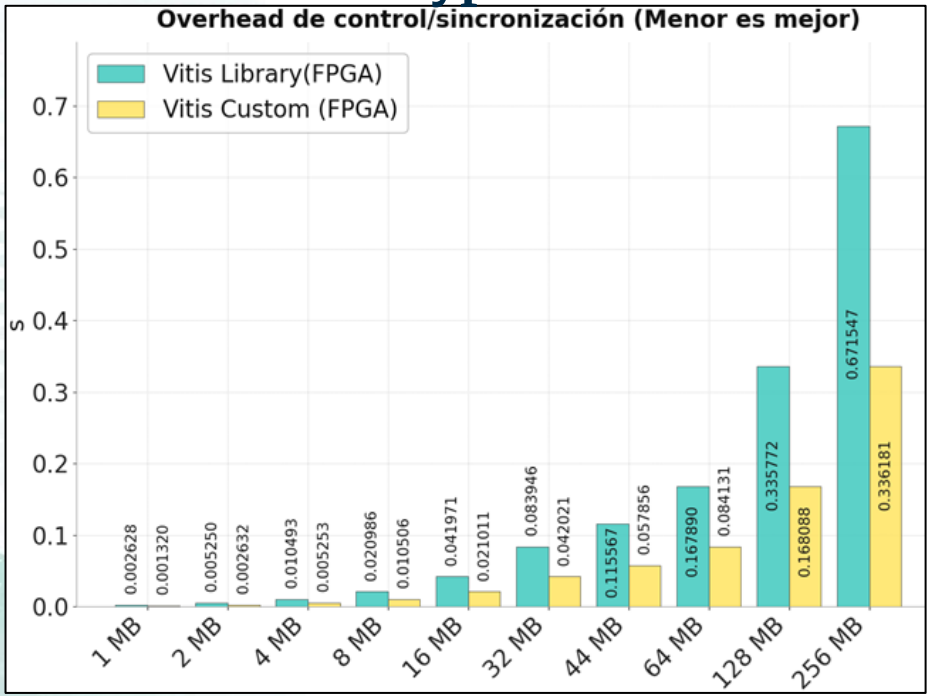


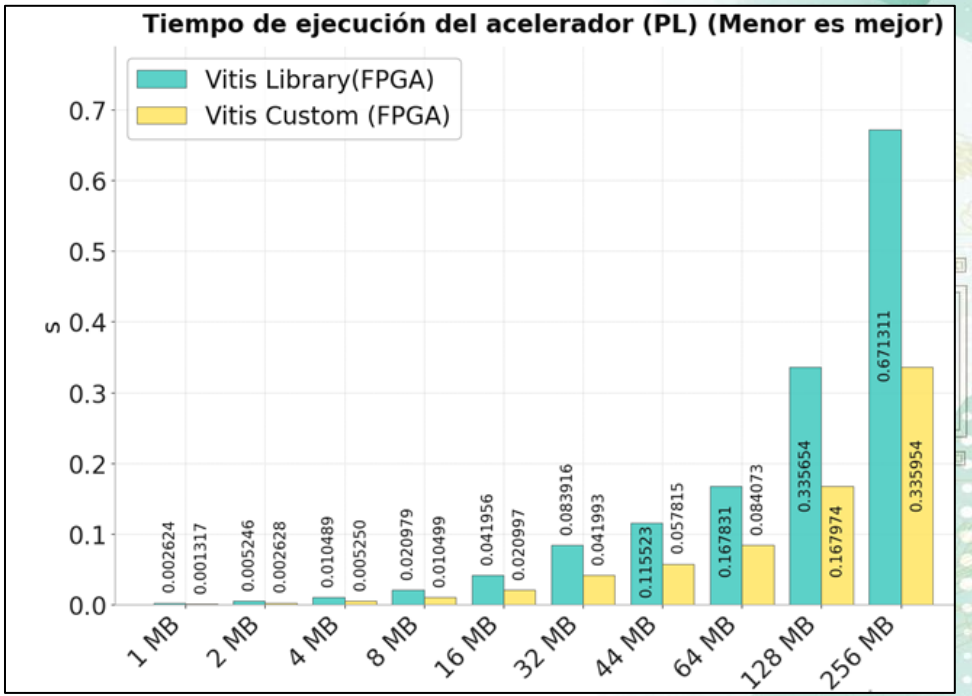




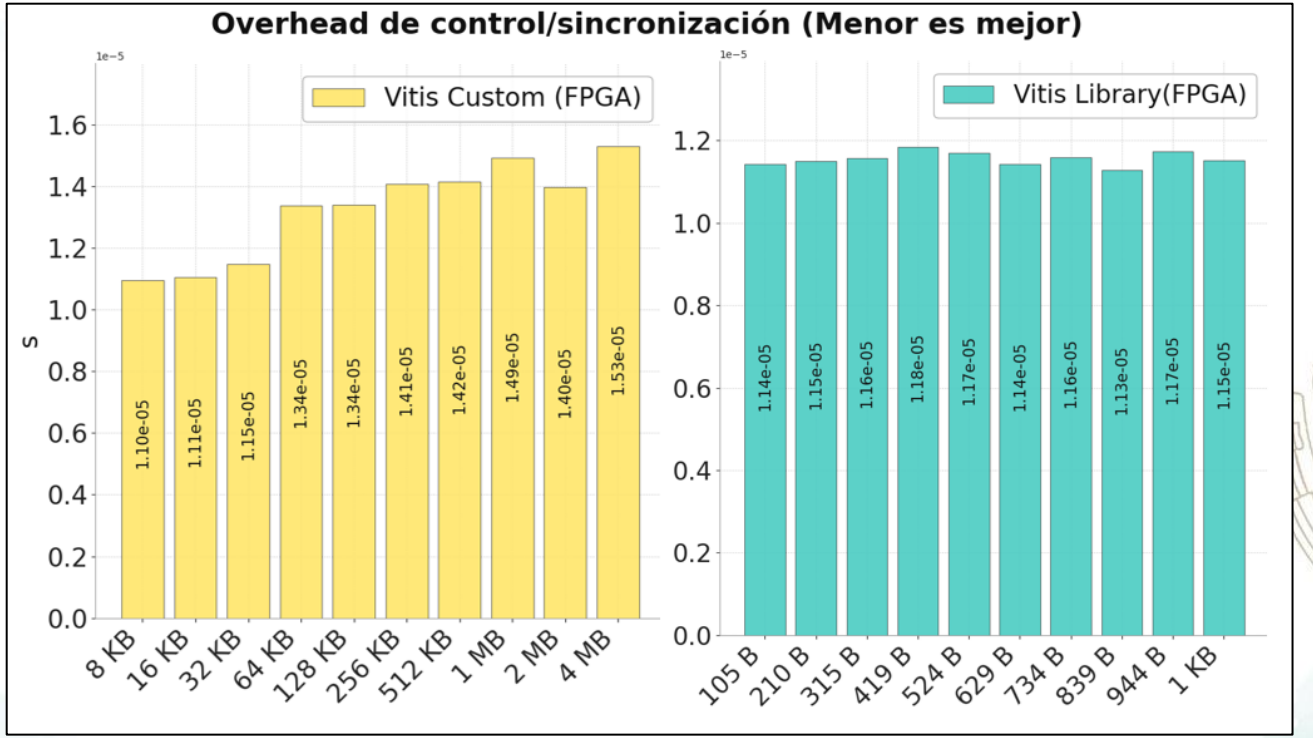
El Cuello de Botella Real: Overhead

AES-128 Encrypt





Overhead de control y tiempo de transferencia



Conclusiones

HLS > VHDL/RTL

High-Level Synthesis permite una aceleración real reduciendo drásticamente el tiempo de desarrollo frente a RTL puro, sin sacrificar rendimiento crítico.

Estabilidad Zynq-7000

La plataforma Zynq-7000 demuestra ser robusta para offloading criptográfico, validando su uso en seguridad embebida industrial.

Personalizado > Librería

En entornos de recursos limitados (IoT/Edge), las soluciones a medida superan a las IPs "caja negra" en eficiencia de área y latencia.

Recomendaciones y Trabajo Futuro

Fase 1: Optimización

Implementación de **DMA Scatter-Gather** para permitir streaming continuo de datos y eliminar tiempos muertos de transferencia.

Fase 2: Seguridad Real

Evolución a modos de operación **AES-GCM o CTR**. Implementación de contramedidas contra ataques de canal lateral.

Fase 3: Migración

Salto a arquitectura **UltraScale+** para romper los límites de frecuencia de reloj y ancho de banda del bus AXI actuales.

ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

¡MUCHAS GRACIAS!