UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Progettazione e implementazione hardware di un modulo AES in Verilog su FPGA Tang Nano 9K per la protezione di comunicazioni Modbus

Hardware design and implementation of a Verilog-based AES Module on Tang Nano 9K FPGA for securing Modbus communications

Relatore:
Dr. Galdelli Alessandro

Laureando:
Badiali Federico

Correlatore:
Prof. Biagetti Giorgio

Anno Accademico 2025-2026

Indice

Elenco delle figure

Sommario

L'avvento dell'Industria 4.0 e la crescente digitalizzazione hanno esposto le reti di controllo industriale a minacce informatiche sempre più complesse. Il protocollo Modbus, standard di fatto per la comunicazione in questo settore, è stato concepito per sistemi isolati e risulta strutturalmente privo di meccanismi di sicurezza. L'assenza di crittografia, di autenticazione e di controlli anti-replay espone i dati e i comandi operativi a vulnerabilità critiche, rendendo le infrastrutture suscettibili a intercettazioni e manomissioni.

Per rispondere a queste criticità, questo lavoro di tesi presenta la progettazione e l'implementazione hardware di un acceleratore crittografico basato sullo standard AES-128, specificamente concepito per proteggere le comunicazioni seriali Modbus RTU. L'obiettivo primario è introdurre un robusto livello di sicurezza (layer crittografico) che sia trasparente per l'infrastruttura esistente, garantendo contemporaneamente il rispetto dei rigorosi vincoli temporali (real-time) del protocollo e mantenendo un'elevata efficienza energetica, fondamentale per i dispositivi edge.

L'efficienza dei processi di cifratura inline è strettamente legata all'architettura hardware adottata. Poiché l'elaborazione puramente software a bassa frequenza introduce latenze inaccettabili per i protocolli industriali, il sistema proposto sfrutta le potenzialità delle FPGA (Field-Programmable Gate Arrays). Sviluppato su una scheda Tang Nano 9K in linguaggio Verilog, il progetto integra un processore soft-core RISC-V (PicoRV32) che, tramite l'interfaccia nativa PCPI, delega il pesante carico crittografico al modulo hardware dedicato, superando i colli di bottiglia dell'esecuzione sequenziale.

I test condotti dimostrano che l'acceleratore personalizzato soddisfa pienamente i requisiti operativi. L'hardware è in grado di completare le operazioni di cifratura in tempi estremamente ridotti, mantenendo la latenza ben al di sotto della soglia critica di fallimento della rete (il vincolo $t_{1.5}$ del protocollo Modbus). Questo garantisce un determinismo temporale che le tradizionali implementazioni software non riescono ad assicurare, evitando i ritardi che causerebbero il collasso della comunicazione industriale.

Oltre alle ottime prestazioni temporali, l'implementazione si distingue per un costo energetico estremamente contenuto, dimostrandosi una soluzione altamente sostenibile e competitiva rispetto ai processori commerciali ad alta frequenza. In conclusione, il lavoro conferma la fattibilità tecnica di integrare la sicurezza nei sistemi edge in modo trasparente e pone le basi per futuri sviluppi, orientati verso la crittografia autenticata e la gestione dinamica delle chiavi.

Summary

The advent of Industry 4.0 and increasing digitalization have exposed industrial control networks to complex cyber threats. The Modbus protocol, a standard for communication in this field, was conceived for isolated systems and inherently lacks security mechanisms. The absence of encryption, authentication, and anti-replay controls leaves operational data and commands vulnerable to critical risks, making infrastructures susceptible to interception and tampering.

To address these vulnerabilities, this thesis presents the hardware design and implementation of a cryptographic accelerator based on the AES-128 standard, specifically tailored to protect Modbus RTU serial communications. The primary objective is to introduce a robust security layer that is transparent to the existing infrastructure, while simultaneously ensuring compliance with the strict real-time constraints of the protocol and maintaining the high energy efficiency required by edge computing devices.

The efficiency of inline encryption processes is closely tied to the adopted hardware architecture. Since purely software-based execution on low-power microcontrollers introduces unacceptable latencies for industrial protocols, the proposed system exploits the potential of FPGAs (Field-Programmable Gate Arrays). Developed on a Tang Nano 9K board using the Verilog language, the project integrates a RISC-V soft-core processor (PicoRV32) which, through the native PCPI interface, delegates the heavy cryptographic workload to the dedicated hardware module, overcoming the bottlenecks of sequential execution.

Conducted tests demonstrate that the custom accelerator fully satisfies the operational requirements. The hardware is capable of completing encryption operations in extremely short times, keeping the latency well below the critical network failure threshold (the Modbus $t_{1.5}$ constraint). This ensures a temporal determinism that traditional software implementations fail to provide, avoiding the delays that would otherwise cause the collapse of industrial communication.

In addition to great temporal performance, the implementation distinguishes itself with a very low energy cost, proving to be a highly sustainable and compe-

titive solution compared to high-frequency commercial processors. In conclusion, the work confirms the technical feasibility of seamlessly integrating security into legacy edge networks and lays the foundation for future developments focused on authenticated encryption and dynamic key management.

Capitolo 1

Introduzione

In un'epoca caratterizzata dall'informazione e dall'enorme quantità di dati, questi ultimi hanno ricoperto un ruolo sempre più fondamentale e strategico in ogni settore. Oggi i dati estratti dai processi industriali non sono più semplici grandezze fisiche, ma le fondamenta del vantaggio competitivo di un'azienda: su di essi si basano le analisi di mercato, la pianificazione e il *know-how* tecnologico. Questo enorme valore li ha trasformati in un vero e proprio *asset* e quindi un bersaglio primario per operazioni di concorrenza sleale, spionaggio industriale o semplice furto al pari di un macchinario o di una fornitura di materie prime.

L'avvento dell'*Industria 4.0* e la progressiva digitalizzazione dei processi produttivi hanno radicalmente trasformato gli scenari dell'automazione, esponendo infrastrutture precedentemente isolate a minacce informatiche di qualsiasi tipo [7]. In questo contesto, e per i motivi appena elencati, la protezione dei dati e dei comandi operativi rappresenta una priorità fondamentale per garantire la continuità e la sicurezza dei sistemi industriali.

Per far fronte a questo bisogno è necessario tuttavia affrontare un ostacolo importante: la vulnerabilità dei protocolli industriali *legacy*. Il protocollo *Modbus*, pur essendo ancora oggi lo standard per la comunicazione nell'automazione grazie alla sua semplicità e robustezza, è stato concepito decenni fa per sistemi chiusi e risulta del tutto privo di meccanismi di sicurezza informatica. L'assenza di crittografia e di autenticazione espone i dati e i comandi in transito sulla rete a intercettazioni e manomissioni, lasciando le infrastrutture critiche aperte ad attacchi.

La motivazione principale che mi ha spinto a intraprendere questo lavoro di tesi nasce proprio dalla volontà di colmare questa lacuna senza dover stravolgere le infrastrutture o i protocolli di rete esistenti. Aggiungere sicurezza ai nodi *edge* industriali rappresenta una sfida complessa: l'elaborazione puramente software della crittografia, specialmente se eseguita su microcontrollori a basso consu-

mo, introduce latenze di calcolo inaccettabili che violano i vincoli di *real-time* richiesti dal protocollo. Vi era quindi la necessità accademica e progettuale di esplorare una soluzione che garantisse contemporaneamente sicurezza, prestazioni deterministiche ed efficienza energetica.

Per risolvere queste problematiche questo lavoro presenta la progettazione e implementazione di un acceleratore crittografico basato sullo standard AES-128.

Sviluppata interamente in linguaggio *Verilog* e implementata sulla scheda FPGA *low-cost Tang Nano 9K*, la soluzione proposta integra un processore *soft-core* che sfrutta un modulo hardware dedicato al quale viene delegato l'intero carico computazionale della cifratura. Questo approccio architetturale è stato concepito attorno a due pilastri fondamentali per i moderni sistemi embedded: l'estrema facilità di integrazione e l'elevata efficienza energetica nell'*edge computing*.

L'acceleratore agisce infatti come un *layer* di sicurezza trasparente che si colloca sopra il flusso di comunicazione seriale, senza richiedere modifiche strutturali ai controllori (PLC) o alla rete già installata. Allo stesso tempo, l'esecuzione dell'algoritmo direttamente nello spazio logico dell'FPGA a livello *hardware* permette di mantenere bassi i consumi e assicura prestazioni deterministiche. Il risultato è un dispositivo in grado di cifrare il *payload* in tempi ristretti, mantenendo questa latenza aggiuntiva ben al di sotto della soglia di timeout e rispettando così i vincoli *real-time* della rete industriale.

1.1 Lo standard Modbus

Il protocollo Modbus è uno standard di comunicazione di livello applicazione introdotto originariamente nel 1979 da Modicon (oggi Schneider Electric) per l'impiego nei propri Controllori Logici Programmabili (PLC). Grazie alla sua semplicità, robustezza e alla sua natura aperta, è diventato rapidamente uno standard per la comunicazione nell'automazione industriale, le cui specifiche sono oggi gestite e mantenute dalla *Modbus Organization* [8].

L'architettura del protocollo si basa su un modello di comunicazione di tipo *request/response* governato da una logica *client/server*. All'interno di questo schema, un unico dispositivo *client* ha la facoltà di avviare le transazioni inviando richieste di lettura o scrittura verso uno o più dispositivi *server*, i quali elaborano l'azione richiesta e rispondono fornendo i dati o confermando l'avvenuta esecuzione del comando.

A livello concettuale, il Modbus organizza le informazioni all'interno del server strutturandole in quattro tabelle di memoria distinte, indirizzabili tramite specifici codici di funzione (*Function Codes*):

- **Discrete Inputs:** ingressi digitali a singolo bit, di sola lettura (es. lo stato binario di un sensore).
- **Coils (Discrete Outputs):** uscite digitali a singolo bit, accessibili in lettura e in scrittura (es. l'attivazione di un relè).
- **Input Registers:** registri a 16 bit di sola lettura, tipicamente impiegati per mappare dati analogici o letture complesse provenienti dalla sensoristica di campo.
- **Holding Registers:** registri a 16 bit accessibili sia in lettura che in scrittura, utilizzati per memorizzare i parametri di configurazione, i setpoint e le impostazioni di controllo del PLC.

Il protocollo supporta differenti modalità di trasmissione a seconda del mezzo fisico utilizzato. Nella sua variante seriale classica (su bus RS-485 o RS-232), denominata *Modbus RTU*, i pacchetti vengono codificati in formato binario compatto per massimizzare l'efficienza di trasmissione, e l'integrità del dato è protetta da un controllo di errore a ridondanza ciclica di tipo CRC (*Cyclic Redundancy Check*). Con l'evoluzione delle reti locali di fabbrica, è stato introdotto lo standard *Modbus TCP/IP*, il quale incapsula la PDU (*Protocol Data Unit*) di Modbus all'interno del payload di un pacchetto TCP standard (sulla porta 502), introducendo un preambolo di identificazione specifico chiamato header MBAP (*Modbus Application Protocol*).

1.2 Obiettivi

1.2.1 Aggiunta layer crittografico e vincolo di real-time

Nonostante la sua grande diffusione e la sua evoluzione verso reti basate su Ethernet (come il Modbus TCP/IP), il protocollo è stato concepito in un'epoca in cui le reti industriali erano fisicamente isolate dall'esterno (sistemi *air-gapped*). Come evidenziato dalle linee guida sulla sicurezza dei sistemi di controllo industriale (ICS) pubblicate dal NIST [9], il Modbus è strutturalmente privo di meccanismi di sicurezza informatica. Nello specifico presenta tre criticità intrinseche fondamentali:

- **Assenza di crittografia:** i dati e i comandi di controllo viaggiano sulla rete in chiaro (*cleartext*). Questa mancanza espone l'intero sistema ad attacchi di intercettazione (*eavesdropping*), permettendo a un attaccante di leggere lo stato dei sensori e i parametri di produzione.
- **Mancanza di autenticazione:** il protocollo non prevede alcun meccanismo per verificare l'identità del mittente. Ciò rende l'infrastruttura fortemente vulnerabile ad attacchi di tipo *spoofing* e *man-in-the-middle*, dove un attore malevolo può iniettare comandi non autorizzati verso attuatori critici.
- **Assenza di controllo anti-replay:** i pacchetti non hanno meccanismi di validazione temporale. Possono quindi essere catturati dalla rete e ritrasmessi in un secondo momento per forzare comportamenti anomali nei PLC.

Cercando di risolvere la prima delle vulnerabilità architetturali elencate, si è optato per l'adozione di un *layer* di sicurezza aggiuntivo. L'implementazione di un acceleratore *hardware* AES, oggetto di questo lavoro di tesi, ha come obbiettivo proprio di cifrare il *payload* delle comunicazioni industriali.

Per concretizzare questo livello di protezione, evitando i pesanti ritardi introdotti da un'elaborazione puramente *software*, si è scelto di sviluppare un'architettura di *co-design* HW/SW. Sfruttando le potenzialità della scheda FPGA Tang Nano 9K, il sistema integra un processore *soft-core* RISC-V che orchestra la comunicazione seriale e lo scambio dei dati, delegando il calcolo interamente ad un modulo *hardware* dedicato. Questa scelta progettuale è strettamente dettata dalla necessità di rispettare i rigorosi vincoli temporali del protocollo Modbus RTU: l'obiettivo è garantire un determinismo tale per cui la latenza di cifratura *inline* si mantenga strettamente inferiore al tempo di silenziamento (il vincolo $t_{1.5}$), assicurando così la continuità operativa del bus ed evitando errori di *timeout*.

L'approccio *hardware-based* assicura la riservatezza e l'integrità dei dati scambiati garantendo latenze minime e deterministiche, requisito fondamentale per soddisfare le stringenti specifiche *real-time* imposte dai sistemi di automazione.

1.2.2 Efficienza energetica e integrazione

Accanto alla robustezza temporale, un pilastro cardine degli obiettivi di questo progetto è il drastico incremento dell'efficienza energetica del sistema. L'adozione di un'architettura embedded snella basata sulla scheda *Tang Nano 9K* permette di mappare fisicamente le trasformazioni matematiche dell'algoritmo *Rijndael* [10] direttamente nello spazio logico *hardware*. Rispetto a una tradizionale esecuzione puramente *software* effettuata su microcontrollori general-purpose, la quale richiederebbe intensivi cicli macchina sequenziali e un continuo fetch di istruzioni in memoria elevando i consumi, l'acceleratore custom progettato ed implementato ottimizza i percorsi critici ed esegue i round di calcolo in appena `103` cicli di clock totali. Questa forte riduzione dell'attività di commutazione logica e del tempo complessivo di computazione minimizza il consumo energetico per singolo bit cifrato, rendendo il modulo ideale per l'integrazione in nodi periferici industriali (*edge nodes*) e sistemi distribuiti soggetti a vincoli di alimentazione.

Infine, un ulteriore e fondamentale obiettivo del progetto è garantire un'integrazione che risulti completamente trasparente e di immediata installazione. Costituendo un *layer* di astrazione intermedio, il sistema è concepito per agire come un vero e proprio *overlay* invisibile sovrapposto all'infrastruttura Modbus. Questo approccio ha lo scopo di proteggere il canale di comunicazione in modo inavvertibile, lasciando inalterata la logica dei nodi *legacy* (PLC, sensori e attuatori) già installati sull'impianto senza imporre nessuna modifica strutturale alla rete industriale di partenza.

1.3 Struttura della tesi

La tesi è organizzata per illustrare progressivamente le basi teoriche, le scelte tecnologiche e i dettagli implementativi del progetto, e si struttura nei seguenti capitoli:

- Il **Capitolo 2** analizza l'evoluzione della crittografia sia storica che moderna, per poi concentrarsi sull'architettura interna dello standard AES, descrivendone le fasi operative di cifratura.
- Il **Capitolo 3** ripercorre l'evoluzione storica delle architetture FPGA, delineando i vantaggi tecnici di questa tecnologia (come il parallelismo e il routing) e portando degli esempi commerciali.
- Il **Capitolo 4** descrive i materiali e i metodi impiegati, introducendo la scheda di sviluppo Tang Nano 9K, le basi del linguaggio Verilog, gli strumenti software della toolchain open-source (Yosys, nextpnr) e le caratteristiche dell'architettura RISC-V PicoRV32.
- Il **Capitolo 5** entra nel dettaglio dell'implementazione tecnica, analizzando approfonditamente il codice sviluppato: dalla gestione automatizzata tramite Makefile, alla progettazione dei moduli hardware in Verilog (acceleratore AES e relative macchine a stati), fino al firmware scritto in linguaggio C e Assembly.
- Il **Capitolo 6** è dedicato all'esposizione dei risultati, verificando l'efficacia e l'efficienza energetica del sistema in un contesto di comunicazione real-time ed esaminando i risultati empirici dei test.
- Infine, il **Capitolo 7** trae le conclusioni del lavoro svolto, discutendo le limitazioni riscontrate durante lo sviluppo e proponendo possibili scenari per evoluzioni future del progetto.

Capitolo 2

Cifratura e codifica dell'informazione: Lo standard AES

La sicurezza delle comunicazioni nei moderni sistemi di automazione richiede l'adozione di algoritmi crittografici standardizzati, robusti e performanti. Questo capitolo esplora l'evoluzione storica della crittografia e i fondamenti teorici dell'*Advanced Encryption Standard* (AES), analizzandone nel dettaglio la complessa struttura matematica basata sull'algebra dei campi finiti e le operazioni fondamentali che compongono l'algoritmo.

2.1 Storia ed evoluzione della crittografia

La crittografia, dal greco *kryptós* (nascosto) e *gráhein* (scrittura), rappresenta la scienza di cifrare le informazioni in modo da renderle inaccessibili a chiunque non sia il legittimo destinatario. Sebbene oggi sia una branca fondamentale dell'ingegneria informatica, le sue radici affondano nell'antichità, evolvendo da una pratica puramente linguistica a una disciplina matematica e tecnologica complessa.

2.1.1 La crittografia classica: dal Cifrario di Cesare a Enigma

I primi tentativi documentati di cifratura risalgono all'ambito militare. Il più celebre esempio dell'antichità è il Cifrario di Cesare, un cifrario a sostituzione monoalfabetica utilizzato da Giulio Cesare per le sue corrispondenze segrete [1]. Il funzionamento si basa sullo scorrimento ciclico di un numero fisso di posizioni (k) di ciascuna lettera dell'alfabeto. Matematicamente, denotando con P il carattere in chiaro e con C il carattere cifrato mappati sull'insieme degli interi $\mathbb{Z}_{26}$,

l'operazione si esprime come:

$$C = (P + k) \pmod{26} \tag{2.1}$$

Nonostante la sua semplicità, la crittografia basata sulla sostituzione monoalfabetica si rivelò strutturalmente debole a causa dell'analisi delle frequenze: ogni lingua presenta una distribuzione statistica nota delle lettere, permettendo di violare il codice con estrema facilità.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Figura 2.1: Encoding delle lettere nel cifrario di Cesare [1].

Il passaggio cruciale verso la crittografia moderna avvenne nel XX secolo con l'avvento delle macchine cifranti elettro-meccaniche, il cui apice fu rappresentato dalla macchina tedesca Enigma, ampiamente utilizzata durante la Seconda Guerra Mondiale. Enigma implementava un sistema a sostituzione polialfabetica estremamente avanzato per l'epoca, basato su una serie di rotori meccanici che ruotavano a ogni pressione di un tasto, modificando continuamente il percorso elettrico e, di conseguenza, l'alfabeto di sostituzione corrente.

La macchina includeva un "*riflettore*" che permetteva di utilizzare lo stesso schema sia per cifrare sia per decifrare, introducendo però il vincolo matematico per cui una lettera non poteva mai essere cifrata in se stessa. Questa apparente simmetria divenne una delle vulnerabilità critiche sfruttate dai crittoanalisti polacchi e, successivamente, dal team guidato da Alan Turing a Bletchley Park. Tramite lo sviluppo della "*Bomba*", un calcolatore elettromeccanico progettato per decifrare i messaggi di Enigma, vennero poste le basi concettuali per la nascita dell'informatica moderna e della crittografia formale.

Figura 2.2: Macchina Enigma dal Deutsches Museum [1].

2.1.2 La formalizzazione matematica di Shannon e il Cifrario di Vernam

Nel secondo dopoguerra, la crittografia compì un salto fondamentale, evolvendo da pratica empirica e meccanica a vera e propria disciplina scientifica e matematica. Questo mutamento di paradigma è in gran parte dovuto al lavoro di Claude Shannon, che nel 1949 pubblicò l'articolo seminale *"Communication Theory of Secrecy Systems"* [11]. Shannon applicò i concetti della neonata Teoria dell'Informazione alla crittografia, definendo in modo rigoroso i criteri di valutazione della sicurezza di un algoritmo e introducendo il concetto di segretezza perfetta (*perfect secrecy*).

Un sistema crittografico possiede segretezza perfetta se l'osservazione del testo cifrato non rivela alcuna informazione utile sul testo in chiaro, a prescindere dalla potenza computazionale o dal tempo a disposizione dell'attaccante. Shannon dimostrò matematicamente che l'unico cifrario in grado di garantire tale proprietà è il Cifrario di Vernam (noto anche come *One-Time Pad*), originariamente concepito da Gilbert Vernam nel 1926 [2]. Il funzionamento si basa sull'applicazione dell'operazione logica XOR ($\oplus$) tra i bit del messaggio in chiaro M e una chiave K completamente casuale, avente lunghezza pari o superiore al messaggio stesso e utilizzata per una sola sessione di comunicazione:

$$C = M \oplus K \tag{2.2}$$

Anche se teoricamente inviolabile, il cifrario di Vernam presenta limiti ingegneristici insormontabili legati alla generazione, distribuzione e memorizzazione sicura di chiavi lunghe quanto i dati stessi, rendendolo impraticabile per i moderni sistemi di automazione e reti di calcolatori.

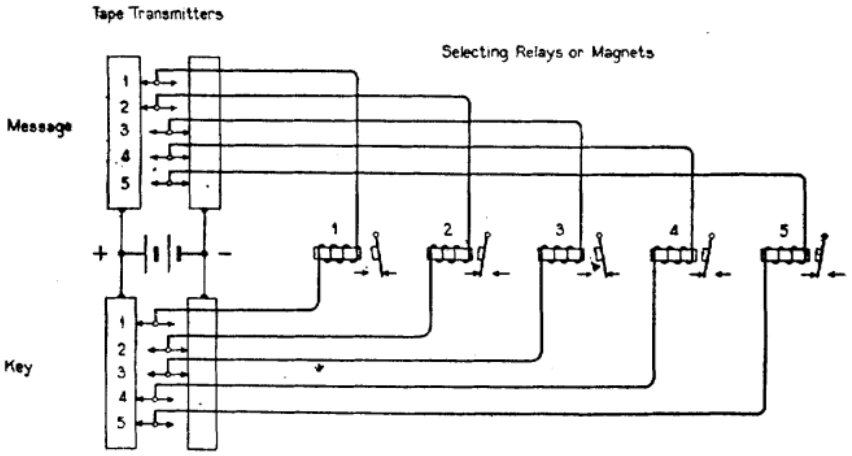

Figura 2.3: Meccanismo di XOR progettato da Vernam. [2].

Nonostante questo, nello stesso trattato, Shannon pose le fondamenta per lo sviluppo dei cifrari a blocchi pratici (come il DES e il successivo AES), teorizzando che l'unione di funzioni crittografiche semplici potesse dare origine a sistemi complessi. Egli individuò i due pilastri della crittografia simmetrica moderna:

1. **Confusione**: l'obiettivo di rendere la relazione statistica tra il testo cifrato e la chiave il più complessa e non lineare possibile, impedendo all'attaccante di intercettare pattern strutturali.

2. **Diffusione**: l'obiettivo di dissipare la struttura statistica del testo in chiaro su tutto il testo cifrato. In questo modo, la modifica anche di un solo bit del messaggio in chiaro deve propagarsi e influenzare idealmente la metà dei bit del testo cifrato (proprietà nota come *effetto valanga*).

2.1.3 Il Cifrario Lucifer e la nascita della crittografia computazionale

Con l'avvento dei primi grandi calcolatori commerciali nei primi anni '70, la necessità di proteggere i dati digitali non fu più una prerogativa esclusiva dei governi e degli apparati militari. Il ponte tecnologico tra le intuizioni teoriche di Shannon e la successiva nascita degli standard pubblici fu rappresentato dal progetto

Lucifer, sviluppato dai laboratori IBM sotto la guida del fisico e crittografo Horst Feistel [12].

Lucifer rappresenta una delle prime e più significative implementazioni concrete di un cifrario a blocchi basato sull'architettura *Substitution-Permutation Network* (SPN) e, successivamente, sulla celebre *Rete di Feistel*. L'algoritmo operava originariamente su blocchi di dati a 48 bit (estesi a 128 bit nelle varianti successive) e sfruttava l'azione combinata e iterativa di moduli di sostituzione non lineare (*S-Box*) e moduli di permutazione lineare (*P-Box*) per concretizzare i concetti di confusione e diffusione [13].

L'architettura di Lucifer fu rivoluzionaria dal punto di vista dell'ingegneria informatica: dimostrò che operazioni matematiche e logiche relativamente semplici, se alternate e ripetute per un numero sufficiente di cicli (detti *round*), erano in grado di generare una sicurezza crittografica elevatissima. Fu proprio questa struttura di base, opportunamente modificata e ottimizzata su richiesta della NSA, a costituire l'ossatura tecnica su cui l'NBS formalizzò il DES, sancendo l'inizio dell'era degli standard crittografici pubblici.

2.1.4 L'era moderna: dal DES alla nascita dell'AES

Con l'aumentare della potenza dei calcolatori , la crittografia abbandonò il concetto di "sicurezza tramite oscurità" dell'algoritmo, allineandosi definitivamente al *Principio di Kerckhoffs* [14]: la sicurezza di un sistema crittografico deve dipendere esclusivamente dal segreto della chiave, mentre l'algoritmo deve essere pubblico e pubblicamente verificabile.

Nel 1977, il governo statunitense formalizzò il primo standard crittografico di riferimento in ambito civile e commerciale: il DES (*Data Encryption Standard*). Il DES era un cifrario simmetrico basato sulla rete di Feistel, che operava su blocchi di 64 bit utilizzando una chiave di soli 56 bit. Con il rapido progresso della potenza di calcolo computazionale, la lunghezza della chiave del DES si dimostrò drammaticamente insufficiente, esponendo l'algoritmo ad attacchi di forza bruta. La soluzione temporanea, nota come *Triple* DES (3DES), incrementava la sicurezza applicando l'algoritmo DES tre volte consecutive con chiavi diverse, ma questa architettura risultava inefficiente e intrinsecamente lenta per le emergenti applicazioni di rete e per i sistemi embedded.

Per ovviare a queste limitazioni e definire uno standard a lungo termine per il nuovo millennio, nel gennaio 1997 il NIST (*National Institute of Standards and Technology*) avviò un concorso pubblico internazionale per la selezione dell' *Advanced Encryption Standard* (AES). I requisiti tassativi imposti dal NIST richiedevano un cifrario a blocchi simmetrico, il supporto nativo a blocchi di dati da 128 bit, chiavi di lunghezza scalabile (128, 192 e 256 bit) e, soprattutto, un'e-

levata efficienza computazionale sia in implementazioni software che hardware (FPGA e ASIC).

Al termine di un processo valutativo durato anni e basato sulla rigorosa analisi di 15 proposte iniziali, nell'ottobre del 2000 il NIST annunciò la vittoria dell'algoritmo *Rijndael*, sviluppato dai crittografi belgi Joan Daemen e Vincent [10]. Rijndael si distinse rispetto agli altri quattro finalisti (MARS, RC6, Serpent e Twofish) per la sua elegante struttura algebrica, le prestazioni eccezionali e il ridotto fabbisogno di memoria. Nel novembre 2001, l'algoritmo venne ufficialmente ratificato come standard federale con la pubblicazione del documento FIPS 197.

2.2 Cifratura AES

L'AES (Advanced Encryption Standard) rappresenta lo stato dell'arte dei cifrari a blocchi simmetrici ed è implementato globalmente per garantire la riservatezza in protocolli critici come TLS, WPA2/WPA3 nell'ambito delle reti wireless e BitLocker per la cifratura dei file system. Operando come algoritmo simmetrico, esso richiede lo scambio sicuro di un'unica chiave segreta condivisa tra i nodi comunicanti.

Il nucleo matematico dell'algoritmo elabora blocchi discreti di dati a 128 bit (16 byte). Nonostante lo standard definisca tre varianti in base alla lunghezza della chiave, la trattazione e il relativo progetto d'architettura hardware si concentrano su AES-128, configurazione che garantisce un livello di sicurezza computazionalmente inattaccabile ottimizzando l'area logica occupata sui dispositivi FPGA industriali. Di seguito verranno elencate le strutture matematiche su cui si basa questo algoritmo, le fasi della cifratura e come le trasformazioni possono essere eseguite tramite hardware.

2.2.1 Fondamenti matematici: I Campi di Galois ($GF(2^8)$)

A differenza di molti algoritmi precedenti, le operazioni di AES non si basano sulla classica aritmetica degli interi, bensì sulle proprietà matematiche di una specifica struttura algebrica nota come Campo di Galois Finito, denotato come $GF(2^8)$.

L'utilizzo di un campo finito è una scelta ingegneristica fondamentale per due ragioni:

1. **Chiusura algebrica**: Qualsiasi operazione interna al campo genera come risultato un elemento che appartiene ancora al campo stesso. Nel caso di $GF(2^8)$, ogni operazione mappa 8 bit di input in esattamente 8 bit di output, evitando fenomeni di espansione dei dati o overflow.

2. **Esistenza dell'inverso**: Ogni elemento non nullo del campo possiede un unico inverso moltiplicativo. Questa proprietà permette l'invertibilità perfetta delle funzioni di cifratura e garantisce eccellenti proprietà di non-linearità crittografica.

Nell'algebra di $GF(2^8)$, un singolo byte composto da 8 bit $(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$ non viene visto come un numero, ma come un *polinomio a coefficienti binari* (ossia appartenenti a $GF(2)$, dove i coefficienti possono essere solo 0 o 1) di grado massimo pari a 7:

$$B(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \quad (2.3)$$

Ad esempio, il byte esadecimale `0x53`, equivalente al binario `01010011`, viene espresso matematicamente dal polinomio $x^6 + x^4 + x + 1$.

L'operazione di Somma in $GF(2^8)$

La somma di due elementi in $GF(2^8)$ si effettua eseguendo la somma algebrica dei coefficienti dei polinomi aventi lo stesso grado. Poiché i coefficienti appartengono a $GF(2)$, l'addizione è governata dalle regole del modulo 2 ($1 + 1 = 0$). Di conseguenza, la somma polinomiale non genera alcun riporto (*carry-free*) e si riduce a un'operazione logica di OR esclusivo (XOR) bit-a-bit. Dati due polinomi $A(x)$ e $B(x)$, la loro somma hardware è esprimibile semplicemente come:

$$C(x) = A(x) \oplus B(x) \tag{2.4}$$

L'operazione di Moltiplicazione e Riduzione Polinomiale

La moltiplicazione tra due byte in $GF(2^8)$ è strutturalmente più complessa. Essa si esegue moltiplicando i due polinomi secondo le normali regole algebriche; tuttavia, il prodotto risultante può raggiungere un grado massimo pari a 14, eccedendo lo spazio fisico del byte. Per ricondurre il risultato all'interno del campo, è necessario eseguire una riduzione modulare rispetto a un polinomio speciale detto *polinomio irriducibile* (l'equivalente dei numeri primi nell'aritmetica standard), che non può essere scomposto in polinomi di grado inferiore.

L'AES adotta tassativamente il seguente polinomio irriducibile di grado 8, definito dallo standard FIPS 197:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.5}$$

In forma esadecimale compatta, escludendo il bit di overflow di nono grado, esso corrisponde al valore `0x1B`.

A livello hardware, la moltiplicazione per un polinomio generico viene scomposta nell'operazione atomica di "moltiplicazione per x" (ovvero per il byte 0x02), denominata **xtime**. Moltiplicare un polinomio $A(x)$ per x equivale a traslare tutti i suoi coefficienti di una posizione a sinistra (uno *shift logico* a sinistra di 1 bit). Se il bit più significativo di partenza (b_7) è pari a 1, lo shift genera un termine di grado x^8, richiedendo l'immediata riduzione hardware tramite un'operazione di XOR con il polinomio riduttore `0x1B`.

La genialità del design di Rijndael risiede proprio in questa perfetta mappatura tra la rigorosa astrazione algebrica e le primitive elettroniche elementari. Per comprendere questa sinergia e dimostrare come la teoria si traduca in efficienza fisica, è utile anticipare l'espressione logica che verrà poi integrata nell'architettura del Capitolo 5. L'adozione di questo approccio puramente combinatorio segue

i principi di ottimizzazione su FPGA introdotti da Chodowiec e Gaj [15], dimostrando come l'operazione *xtime* possa essere risolta in tempo nullo e senza l'uso di pesanti memorie, utilizzando unicamente di shift e multiplexer e permettendo di risparmiare preziose risorse in termini di area logica:

```
wire [7:0] m2 = {b[6:0], 1'b0} ^ (b[7] ?  8'h1b :  8'h00);
```

2.2.2 Lo Stato AES: la matrice 4 × 4

A differenza di altri cifrari che elaborano il flusso di bit linearmente, AES organizza i 16 byte del blocco dati in una struttura bidimensionale interna denominata *State*. L'inserimento dei byte del messaggio in chiaro (*plaintext*) avviene secondo un ordinamento rigoroso per colonne (*column-major order*):

```
Input: b0  b1  b2  b3  b4  b5  b6  b7  b8  b9  b10 b11 b12 b13 b14 b15

Matrice State:
        col0  col1  col2  col3
riga0 [  b0    b4    b8   b12  ]
riga1 [  b1    b5    b9   b13  ]
riga2 [  b2    b6   b10   b14  ]
riga3 [  b3    b7   b11   b15  ]
```

Figura 2.4: Rappresentazione e mappatura della matrice di stato 4 × 4 nell'algoritmo AES.

Ogni successiva trasformazione del round manipolerà gli elementi di questa matrice, aggiornando lo *State* fino all'ottenimento del testo cifrato finale.

2.2.3 Struttura generale dell'algoritmo

L'architettura interna di AES-128 prevede l'esecuzione iterativa di *10 round* di calcolo. Il flusso operativo si apre con un round preliminare (Round 0) composto esclusivamente dalla funzione *AddRoundKey*. Successivamente, lo *State* attraversa 9 round identici standard e un decimo round finale in cui la funzione *MixColumns* viene deliberatamente omessa per garantire la simmetria architetturale in fase di decifratura.

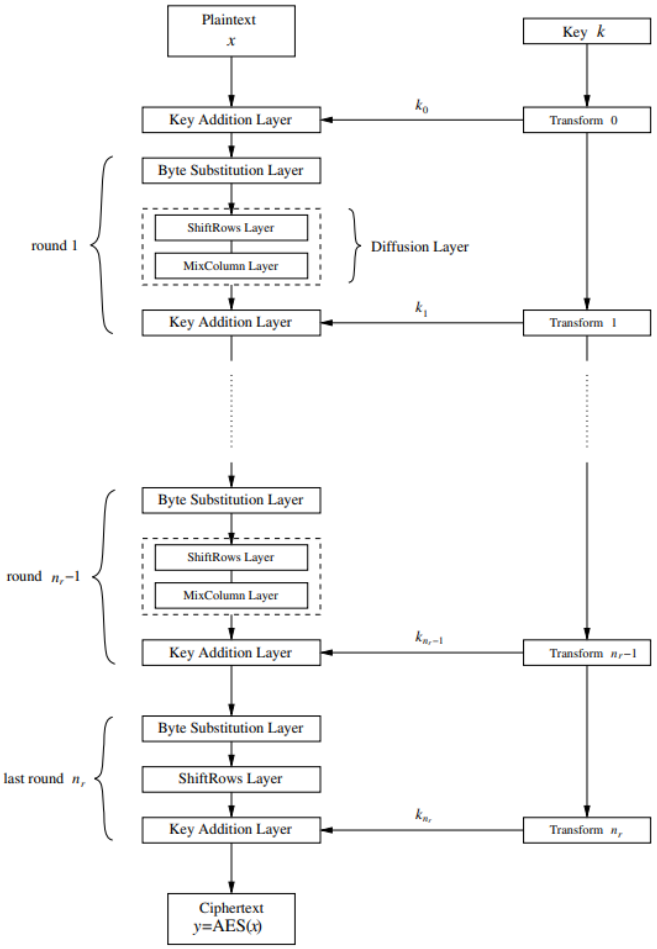

Figura 2.5: Diagramma di flusso sequenziale dei round di cifratura AES-128 [1].

Ogni round standard implementa la sequenza logica di trasformazioni descritta nel paragrafo seguente.

2.2.4 Le quattro operazioni di round

Ciascun round sfrutta le proprietà algebriche di $GF(2^8)$ per garantire i requisiti crittografici fondamentali di confusione e diffusione definiti da Shannon.

1. SubBytes (Sostituzione non-lineare)

La trasformazione `SubBytes` mappa in modo indipendente ogni byte della matrice *State* tramite una tabella di look-up globale non-lineare detta *S-Box*. Ma-

tematicamente, la S-Box viene generata combinando due operazioni distinte in $GF(2^8)$:

1. Il calcolo dell'inverso moltiplicativo A^{-1} del byte (ponendo l'elemento $00 \rightarrow 00$ in questo specifico caso).
2. L'applicazione di una trasformazione affine invertibile sopra il campo $GF(2)$.

Questa combinazione assicura la massima resistenza ad attacchi di crittoanalisi lineare e differenziale.

2. ShiftRows (Rotazione delle righe)

L'operazione ShiftRows agisce sulle righe della matrice di *State*, effettuando una traslazione ciclica verso sinistra di ampiezza variabile in base all'indice della riga stessa:

- Riga 0: nessuna rotazione.
- Riga 1: shift ciclico a sinistra di 1 byte.
- Riga 2: shift ciclico a sinistra di 2 byte.
- Riga 3: shift ciclico a sinistra di 3 byte.

Questa rotazione distrugge la linearità delle colonne, distribuendo l'informazione del singolo byte su più colonne del round successivo.

3. MixColumns (Miscelazione in $GF(2^8)$)

MixColumns opera colonna per colonna, trattando ciascuna di esse come un vettore a quattro elementi. Ogni colonna viene moltiplicata per una matrice quadrata di coefficienti fissi, eseguendo i calcoli interamente secondo le regole algebriche di $GF(2^8)$ descritte precedentemente:

$$\begin{bmatrix} \mathbf{02} & \mathbf{03} & \mathbf{01} & \mathbf{01} \\ \mathbf{01} & \mathbf{02} & \mathbf{03} & \mathbf{01} \\ \mathbf{01} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ \mathbf{03} & \mathbf{01} & \mathbf{01} & \mathbf{02} \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} \mathbf{s'_{0,0}} & \mathbf{s'_{0,1}} & \mathbf{s'_{0,2}} & \mathbf{s'_{0,3}} \\ \mathbf{s'_{1,0}} & \mathbf{s'_{1,1}} & \mathbf{s'_{1,2}} & \mathbf{s'_{1,3}} \\ \mathbf{s'_{2,0}} & \mathbf{s'_{2,1}} & \mathbf{s'_{2,2}} & \mathbf{s'_{2,3}} \\ \mathbf{s'_{3,0}} & \mathbf{s'_{3,1}} & \mathbf{s'_{3,2}} & \mathbf{s'_{3,3}} \end{bmatrix}$$

Figura 2.6: Moltiplicazione matriciale in campo finito dell'operazione. [3].

Sfruttando le proprietà dell'operazione xtime, la moltiplicazione per i coefficienti della matrice (2, 3, 1, 1) viene risolta interamente tramite reti combinatorie di porte XOR in cascata, consentendo di calcolare i nuovi byte della colonna in tempo nullo all'interno del ciclo di clock.

4. AddRoundKey (Miscelazione con la chiave)

AddRoundKey rappresenta il punto di contatto tra la struttura computazionale del cifrario e la chiave segreta. Lo *State* viene combinato bit-a-bit tramite un'operazione di XOR con la corrispondente sottochiave generata dal *Key Schedule* per quel determinato round. Questa trasformazione introduce l'effettiva dipendenza del dato dalla chiave, rendendo impossibile la decifratura senza la conoscenza della stessa.

2.2.5 Key Schedule: L'espansione della chiave

Per alimentare gli 11 passaggi di *AddRoundKey* (dal round 0 al round 10), l'algoritmo espande la chiave master iniziale da 128 bit in un vettore esteso contenente 11 sottochiavi distinte, per un totale di 1408 bit. Questo processo prende il nome di *Key Schedule*.

L'espansione avviene su base ricorsiva a blocchi di 32 bit (*word*). Per ogni nuova chiave di round, la prima *word* viene calcolata applicando all'ultima *word* della sottochiave precedente una funzione di trasformazione non-lineare basata su tre passi sequenziali:

1. **RotWord**: Rotazione ciclica dei byte a sinistra di una posizione.
2. **SubWord**: Sostituzione non-lineare di ciascun byte effettuata tramite la S-Box.
3. **XOR con Rcon**: Somma logica con una costante di round specifica ($Rcon$), definita come potenza di 2 in $GF(2^8)$, introdotta per eliminare eventuali asimmetrie o debolezze statistiche nella chiave.

Le rimanenti tre word della sottochiave corrente vengono calcolate tramite semplici catene di XOR combinatori stesi a partire dalle word precedenti.

Capitolo 3

Architetture FPGA

L'implementazione di un approccio hardware per la cifratura è stata resa possibile grazie all'utilizzo di un dispositivo FPGA. In questo capitolo verranno illustrate le motivazioni che hanno portato allo sviluppo di questa tecnologia, analizzando nel dettaglio le caratteristiche principali e i vantaggi che questa architettura offre.

3.1 Evoluzione storica

L'evoluzione degli Field Programmable Gate Arrays (FPGA) rappresenta uno dei capitoli più affascinanti della microelettronica moderna. Fin dalla loro introduzione commerciale nel 1984 a opera di *Xilinx*, la crescita di questi dispositivi è stata guidata non solo dal semplice ridimensionamento tecnologico previsto dalla *Legge di Moore*, ma da veri e propri salti qualitativi a livello architetturale, economico ed energetico.

In una nota retrospettiva, S. M. Trimberger ha formalizzato questa evoluzione dividendola in tre epoche principali, ognuna caratterizzata da sfide tecnologiche ed esigenze di mercato ben precise: l'Età dell'Invenzione, l'Età dell'Espansione e l'Età dell'Accumulazione [4].

3.1.1 Il contesto tecnologico: PAL e ASIC

Prima dell'avvento degli FPGA, negli anni '80, i progettisti di sistemi digitali si dividevano prevalentemente tra l'uso di *Application-Specific Integrated Circuits* ASIC per i volumi elevati e la logica programmabile convenzionale, come i *Programmable Array Logic* PAL, per i volumi ridotti.

I limiti geometrici dei PAL

I PAL consistevano in una struttura logica a due livelli, basata su un array di porte AND programmabili seguito da un array di porte OR fisse. Pur essendo efficienti dal punto di vista produttivo, soffrivano di un insormontabile problema di scalabilità.

Il numero di punti programmabili nell'array AND cresceva con il quadrato del numero di ingressi, mentre il progresso del silicio forniva nuovi transistor in modo lineare.

Questo incremento quadratico rendeva i PAL di grandi dimensioni fisicamente impraticabili a causa dell'eccessivo consumo energetico e del ritardo di propagazione dei segnali.

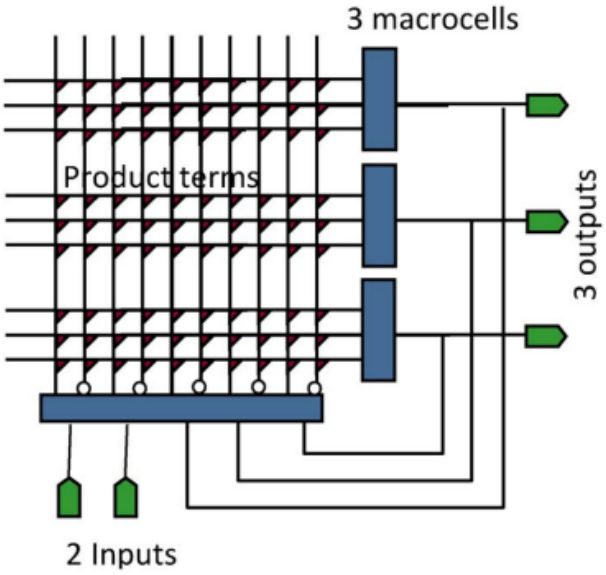

Figura 3.1: Architettura PAL. [4].

Il rischio economico degli ASIC

Gli ASIC garantivano altissime prestazioni, ma l'implementazione fisica della logica richiedeva la creazione di maschere in silicio personalizzate, comportando costi di ingegnerizzazione altissimi e proibitivi.

Inoltre, la progettazione di un ASIC presentava un rischio industriale enorme: un errore di logica costringeva a riprogettare il silicio, causando settimane o mesi di ritardo e conseguenti ripercussioni sulle spese. Gli FPGA risolsero questo problema strutturale fornendo un singolo chip standard che ammortizzava i costi

su migliaia di clienti per i produttori, e a loro volta permettendo di eliminare i costi iniziali a chi le utilizzava garantendo una correzione immediata degli errori tramite la riprogrammazione in laboratorio.

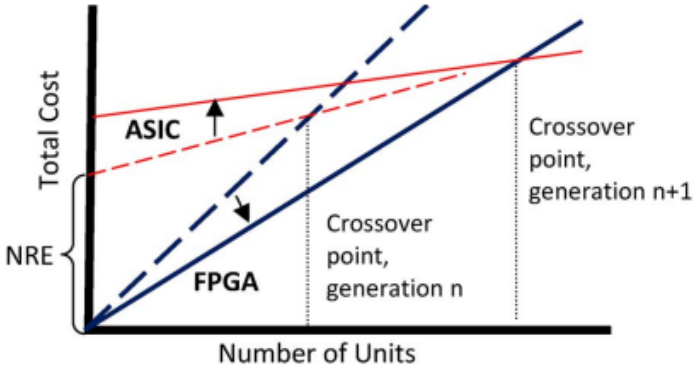

Figura 3.2: Punto di incrocio (crossover point) tra il costo totale di un'implementazione FPGA e ASIC. [4].

3.1.2 Le Tre Età degli FPGA

L'innovazione fondamentale alla base degli FPGA fu l'eliminazione dell'array AND tipico dei PAL, sostituito da memorie di configurazione distribuite per controllare matrici di blocchi logici e reti di routing. Questa libertà architetturale permise agli FPGA di scalare linearmente con la *Legge di Moore*.

1. L'Età dell'Invenzione (1984-1991)

Il primo FPGA, la Xilinx XC2064 [4], implementava appena 64 blocchi logici. Nonostante le capacità limitate, il die di silicio era considerato enorme per gli standard produttivi dell'epoca.

Di conseguenza, il contenimento dell'area di silicio era cruciale per la sopravvivenza stessa delle aziende produttrici. Poiché fili e interruttori aggiungevano dimensioni al chip senza fornire logica vendibile, i primi dispositivi venivano deliberatamente progettati con scarsissime risorse di interconnessione, risultando notoriamente difficili da sbrogliare. Le dimensioni modeste dei circuiti permettevano comunque di eseguire l'instradamento in modo manuale.

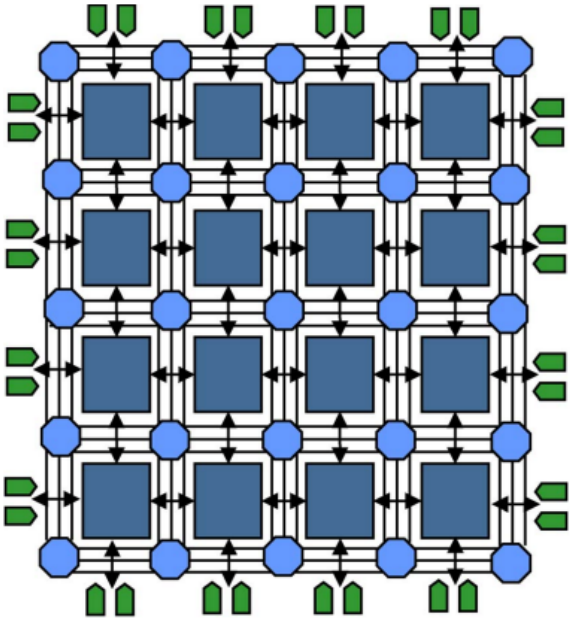

Figura 3.3: Architettura generica ad array con blocchi logici e interconnessioni distribuite. [4].

2. L'Età dell'Espansione (1992-1999)

Durante gli anni '90, l'introduzione della planarizzazione chimico-meccanica (CMP) consentì la sovrapposizione di multipli strati di metallo sul chip. Il costo delle interconnessioni crollò e lo spazio sul silicio divenne meno prezioso rendendo possibili reti di cavi lunghi e complessi, fondamentali per supportare gli emergenti software EDA (*Electronic Design Automation*). Per chip ad architetture complesse che iniziavano ad ospitare decine di migliaia di porte logiche, sintesi, piazzamento e routing automatizzati divennero requisiti obbligatori. In questa fase si affermarono definitivamente le architetture basate su *SRAM e Look-Up Tables (LUT)*, capaci di adattarsi molto più velocemente alle nuove tecnologie a differenza delle più rigide tecnologie precedenti.

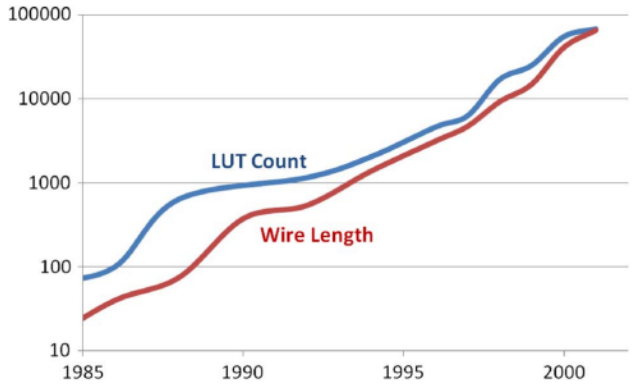

Figura 3.4: Crescita del numeri di LUT e lunghezza dei fili [4].

3. L'Età dell'Accumulazione (2000-2007)

All'inizio degli anni 2000, gli FPGA crebbero a tal punto da superare le dimensioni medie dei progetti standard. Il mercato smise di essere attratto unicamente dall'aumento dei gate logici. Sotto la spinta dell'industria delle telecomunicazioni e di fronte al rallentamento dei benefici termici della *Legge di Moore*, la strategia progettuale mutò radicalmente. Le aziende smisero di aggiungere solo blocchi logici programmabili e iniziarono ad integrare IP *hardcoded* all'interno del silicio. Nacquero così le *Platform FPGA* (come quelle attualmente sul mercato e quella utilizzata), arricchite da enormi blocchi di RAM, moltiplicatori DSP ad altissime prestazioni, *transceiver seriali* e interi *core microprocessoriali* (come il PowerPC). Questi blocchi hardware, estremamente efficienti, trasformarono l'FPGA da un array omogeneo di porte a un complesso motore eterogeneo di calcolo ed elaborazione dati.

3.2 Struttura interna delle FPGA

L'architettura hardware di un *Field Programmable Gate Array* si discosta radicalmente dall'approccio procedurale dei classici microprocessori. Questa si presenta come una griglia bidimensionale simmetrica, studiata per massimizzare la flessibilità di instradamento e il favorire il parallelismo.

In una panoramica fondamentale per la classificazione accademica, i ricercatori S. Brown e J. Rose hanno formalizzato i tre macro-elementi costitutivi che compongono qualsiasi FPGA commerciale moderna: i blocchi logici, le risorse di routing e i blocchi di interfaccia. [5]

3.2.1 Configurable Logic Blocks (CLB)

Il nucleo fondamentale dell'FPGA è rappresentato dai Configurable Logic Blocks (CLB). A differenza delle classiche porte logiche (AND, OR, NOT) stampate sul silicio, le FPGA basate su architettura SRAM implementano la logica attraverso memorie chiamate Look-Up Tables (LUT).

Una LUT a N ingressi non è altro che una piccola memoria statica contenente 2^N bit di configurazione. Invece di calcolare l'operazione algebrica, la LUT si comporta come una tabella di verità. Gli ingressi logici funzionano come indirizzi di memoria, e l'uscita fornisce istantaneamente il bit immagazzinato a quell'indirizzo. Questa struttura permette a un singolo blocco logico di implementare qualsiasi funzione combinatoria complessa in un tempo logico costante ed indipendente dalla complessità. All'interno del CLB, le LUT sono quasi sempre affiancate da circuiti flip-flop, fondamentali per creare macchine a stati sequenziali e per la temporizzazione sincrona dei dati (pipelining).

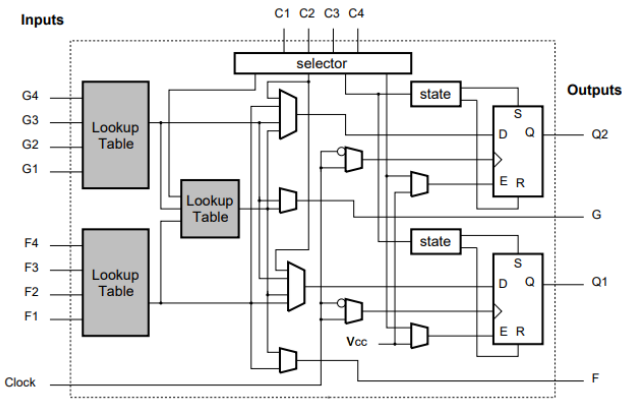

Figura 3.5: Schema logico interno di un singolo Configurable Logic Block, comprendente la Look-Up Table (LUT), i multiplexer e il flip-flop di uscita [5].

3.2.2 Matrice di Interconnessione (Routing)

La capacità di un FPGA di eseguire algoritmi complessi dipende strettamente dalla sua rete di routing, che storicamente occupa la maggior parte dell'area in silicio del chip ed è anche la più complessa da sintetizzare dai tool. L'interconnessione è formata da una griglia gerarchica di canali di tracciamento orizzontali e verticali che attraversano l'intero dispositivo. All'incrocio tra questi canali si trovano le *Switch Box*: matrici di interruttori a transistor programmabili (pass-transistor o multiplexer) che permettono di instradare elettricamente i segnali. Il software di *place and route* calcola la via ottima per connettere l'uscita di una specifica LUT all'ingresso di un'altra, configurando le celle SRAM che governano i nodi del percorso.

3.2.3 Input/Output Blocks (IOB)

Collocati lungo il perimetro esterno della matrice, gli IOB fungono da interfaccia elettrica tra il mondo logico interno e l'hardware del mondo esterno. Questi blocchi sono altamente configurabili e permettono al progettista di gestire dinamicamente gli standard di tensione, l'impedenza, l'attivazione dei resistori di pull-up e la direzione del segnale (input, output o bidirezionale). A differenza dei microcontrollori, dove l'impostazione dei pin è gestita via software tramite la scrittura in registri di memoria (come i registri TRIS o PORT), negli IOB degli FPGA queste caratteristiche sono controllate a livello puramente hardware. Durante l'accensione del dispositivo il file di configurazione (*bitstream*) oltre a configurare le LUT, programma delle celle SRAM statiche dedicate, le quali attivano o disattivano fisicamente i circuiti analogici e i buffer *tri-state* interni al blocco di I/O.

3.3 Vantaggi applicativi

La convergenza tra l'architettura a matrice appena descritta e l'enorme disponibilità di silicio garantita dall'evoluzione tecnologica ha reso gli FPGA la scelta ideale per le applicazioni che richiedono calcolo deterministico ed elevato *processing* in tempo reale e di cui non bisogna fare una produzione su larga scala.

Mentre le architetture software classiche elaborano le istruzioni in modo rigorosamente sequenziale su una CPU, gli FPGA permettono di mappare fisicamente l'algoritmo nello spazio bidimensionale. Ciò si traduce nella creazione di pipeline fisiche e reti di calcolo parallele capaci di processare enormi array di dati in modo simultaneo e con bassissima latenza.

Inoltre, a differenza degli ASIC, l'intrinseca riconfigurabilità degli FPGA permette ai progettisti di aggiornare i dispositivi già installati sul campo tramite semplici modifiche firmware, prolungando il ciclo di vita del prodotto, ottimizzandone le funzionalità e abbattendo i costi.

Oltre ad i vantaggi di computazione e aggiornamento l'utilizzo di FPGA rende sostenibile la progettazione e lo sviluppo di chip per operazioni complesse e specifiche non destinati alla produzione di massa aggirando le problematiche economiche degli ASIC descritte in precedenza.

3.4 Panorama commerciale degli FPGA attuali

L'evoluzione storica e tecnologica descritta precedentemente ha plasmato l'attuale mercato degli FPGA, che si presenta oggi dominato da architetture fortemente eterogenee. Come evidenziato da recenti analisi di settore, la transizione dalle tradizionali logiche programmabili a veri e propri *System-on-Chip* multiprocessore (MPSoC) è ormai uno standard assodato, guidato soprattutto dalle esigenze di telecomunicazioni avanzate, edge computing e inferenza per l'Intelligenza Artificiale [16].

Il mercato moderno (al 2026) si articola principalmente attorno a quattro grandi aziende tecnologiche: AMD, Altera, Lattice Semiconductor e Microchip. Ognuna delle quali ha adottato filosofie architetturali distinte per rispondere a diverse esigenze di calcolo, consumo energetico e tolleranza ai guasti.

3.4.1 AMD (Xilinx) e le architetture ACAP

A seguito della formale acquisizione di Xilinx completata nel 2022, AMD ha consolidato la propria posizione nel settore del calcolo adattivo ad alte prestazioni. Il vertice dell'offerta è rappresentato dalla famiglia Versal, che l'azienda definisce come ACAP (*Adaptive Compute Acceleration Platform*) [17]. A differenza di un FPGA tradizionale, l'architettura Versal integra nella stessa struttura di silicio la matrice programmabile, processori multicore ARM (*Scalar Engines*) e vettori DSP ad altissime prestazioni specificamente progettati per il machine learning (*AI Engines*).

Per il segmento intermedio ad alte prestazioni, la recente linea Kintex UltraScale+ Gen 2 fornisce un'elaborazione deterministica ad altissima larghezza di banda, pensata in maniera specifica per applicazioni medicali, broadcast e di visione industriale complesse [18].

3.4.2 Altera (Intel) e l'integrazione a Chiplet

Nel 2024 Intel ha rilanciato Altera come entità indipendente focalizzata unicamente sulle logiche programmabili [19]. L'offerta aziendale ruota interamente attorno alla piattaforma Agilex (declinata nelle serie 3, 5, 7 e 9), studiata per coprire lo spettro che va dai piccoli dispositivi edge fino all'accelerazione in data center.

La peculiarità tecnologica degli FPGA Agilex risiede nell'uso del packaging 3D e della tecnologia EMIB (*Embedded Multi-die Interconnect Bridge*). Questo approccio "a chiplet" permette di affiancare alla matrice FPGA centrale svariati moduli di I/O (ricetrasmettitori, chip di memoria HBM) creati in silicio in modo indipendente, garantendo grandissima flessibilità [19]. Inoltre, con la serie *Agilex 5*, Altera ha introdotto il primo fabric FPGA orientato all' IA, implementando tensori ottimizzati direttamente nei blocchi DSP per massimizzare le prestazioni per watt nell'*edge computing* [19].

3.4.3 Lattice Semiconductor: l'avanguardia del Low-Power

Mentre AMD e Altera si scontrano sul terreno delle massime prestazioni assolute, Lattice Semiconductor domina il vitale segmento degli FPGA a bassissimo consumo (low-power), fondamentali per l'IoT, la robotica di servizio e la sensoristica a batteria.

Con il rilascio della piattaforma Avant (E, X, G), Lattice ha esteso la sua filosofia progettuale nel segmento "mid-range" [20]. Questi dispositivi offrono fino a 500 mila celle logiche mantenendo al contempo un ingombro fisico (*footprint*) ridottissimo e un consumo di corrente di leakage trascurabile. Un ulteriore elemento chiave di questi integrati è il tempo di reazione: dispositivi come l'Avant-X assicurano tempi di avvio (boot) inferiori ai 60 millisecondi, rendendoli ideali per interfacce di controllo automotive o *smart embedded vision* [20].

3.4.4 Microchip (PolarFire): Sicurezza e Non-Volatilità

Un approccio radicalmente alternativo è quello proposto da Microchip Technology con le famiglie **PolarFire** e PolarFire SoC. Mentre le architetture dei concorrenti si basano quasi esclusivamente su celle di memoria SRAM (che perdono i dati allo spegnimento e necessitano di essere riprogrammate da una ROM esterna a ogni avvio), le matrici PolarFire sfruttano una tecnologia di interconnessione non-volatile su base SONOS/Flash [21].

Questa scelta architetturale presenta due immensi vantaggi: l'assenza totale di correnti statiche di accensione e, soprattutto, l'immunità costituzionale agli upset da radiazione singola (SEU - *Single Event Upset*). Per questo motivo, le

PolarFire rappresentano lo standard nei settori aerospaziale, militare e in quegli ambienti dove l'interferenza cosmica causerebbe alterazioni disastrose alla logica [21]. A completare il quadro innovativo, le versioni *PolarFire SoC* abbandonano il monopolio dei core ARM integrando direttamente nel silicio un cluster di calcolo asimmetrico basato sull'architettura open-source *RISC-V*, garantendo estremo determinismo e sicurezza informatica esaminabile a livello hardware [21].

Capitolo 4

Materiali e metodi

Il capitolo che segue descrive l'hardware utilizzato e tutta la struttura di risorse e software utilizzata per portare a termine questo progetto. Viene presentata la scheda di sviluppo FPGA selezionata, per poi analizzare nel dettaglio il linguaggio di descrizione hardware e l'intera toolchain di sintesi rigorosamente *open-source*. Infine viene trattata l'integrazione con il *soft-core RISC-V* e il suo protocollo di comunicazione con i coprocessori dedicati.

4.1 La scheda di sviluppo Tang Nano 9K

Come precedentemente introdotto, un FPGA (Field-Programmable Gate Array) è un circuito integrato il cui comportamento logico è configurabile tramite software. Questo permette di modellare fisicamente i percorsi elettrici tra le LUT, creando pipeline logiche e circuiti paralleli capaci di eseguire elaborazioni complesse in frazioni di ciclo di clock, che è esattamente il motivo per cui è stata utilizzata questa architettura.

Il progetto è stato sviluppato, simulato e caricato fisicamente sulla *Tang Nano 9K*, una scheda di sviluppo *low-cost* progettata da Sipeed e basata sul chip FPGA Gowin GW1NR-LV9 [6] . Questa board, estremamente compatta, rappresenta un compromesso ideale tra densità logica, flessibilità di interfacciamento e totale supporto per toolchain open-source. Nella board sono presenti inoltre connettori usb-c e hdmi, pulsanti, led, un regolatore di tensione e un piccolo microcontrollore per gestire protocollo JTAG e UART

La board dispone delle seguenti risorse principali:

- **8.640 LUT4** (Look-Up Table a 4 ingressi), moduli elementari impiegati per la mappatura della logica combinatoria.
- **6.480 Flip-Flop**, registri dedicati alla memorizzazione degli stati sequenziali.

- **468 Kbit di Block RAM (BRAM)**, suddivisi in 26 macro-blocchi da 18 Kbit indipendenti ad alta velocità.
- **20 blocchi DSP** (Digital Signal Processing), moltiplicatori hardware dedicati alle operazioni matematiche avanzate(non utilizzati nel progetto).
- **2 PLL** (Phase-Locked Loop), per la sintesi e la moltiplicazione dei segnali di clock(non utilizzato nel progetto).
- Un **oscillatore on-board da 27 MHz**, utilizzato direttamente per derivare il segnale di clock di sistema globale del progetto.
- **608 Kbit di User Flash interna**, memoria non volatile integrata nel die (non utilizzata nel progetto a causa delle specifiche modifiche hardware adottate).
- **32 Mbit di memoria Flash Puya esterna** (SPI NOR esterna al die), adoperata per l'immagazzinamento non volatile dei dati e accessibile tramite protocollo SPI.
- **64 Mbit di PSRAM** integrata nel package GW1NR, per l'espansione della memoria volatile (non utilizzata nel progetto).

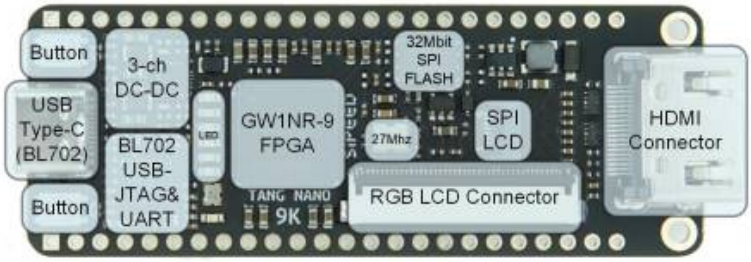

Figura 4.1: Componenti tang nano 9k [6].

L'utilizzo della BRAM

Nel corso della progettazione la BRAM si è rivelata una risorsa indispensabile in quanto è stata utilizzata sia come memoria centrale per la cpu sia per l'istanziazione hardware della tabella di sostituzione non lineare (*S-Box*) richiesta dall'algoritmo AES. Questa scelta ha permesso la massimizzanzione delle prestazioni e risparmiato moltissime LUT che sarebbero servite per calcolarle la funzione non lineare portando inoltre ad un pericoloso percorso critico.

Bootloading e modifica hardware

In fase di accensione, dopo un periodo di Power-On e configurazione hardware (pari a circa 4.9 ms), un bootloader minimale provvede al caricamento sequenziale del firmware dalla memoria *Flash* esterna Puya verso la BRAM interna all'FPGA, innescando infine l'avvio e l'esecuzione del processore RISC-V. Per fare leggere il *bitstream hardware* dalla flash esterna è stato necessario eseguire una modifica hardware alla board per modificare il comportamento di default che prevede il caricamento dalla flash interna. E' stato quindi necessario portare il pin MODE1(87) alto. Per fare ciò è sufficiente rimuovere la resistenza R17 da 4.7K ohm collegata a ground, non è necessario sostituirla dato che sul pin è presente un pull-up interno.

4.2 Verilog: Hardware Description Language

La codifica dell'architettura digitale e della logica di controllo è stata interamente realizzata in Verilog, un *Hardware Description Language* (HDL) che rappresenta uno standard nell'industria microelettronica non solo degli FPGA. A differenza di un linguaggio di programmazione software tradizionale (come il C o il Python), il codice Verilog non viene eseguito linearmente. Esso viene invece si limita a descrivere fisicamente l'hardware e viene *sintetizzato* da un compilatore specializzato che traduce le espressioni testuali in una *netlist* di porte logiche reali (LUT, multiplexer e Flip-Flop) operanti in stretto e continuo parallelismo.

L'unità fondamentale del linguaggio è il modulo, che può essere visto come un blocco elettronico indipendente provvisto di pin di ingresso e uscita. I moduli supportano l'istanziazione gerarchica, permettendo la costruzione di sistemi complessi partendo da funzioni elementari. Per fare un esempio, nel progetto presentato il modulo di vertice (*Top*) racchiude e interconnette i sub-moduli funzionali principali: l'acceleratore hardware AES, il processore, il blocco di interfacciamento UART per la trasmissione seriale e i controller di decodifica degli indirizzi di memoria.

Un aspetto fondamentale durante la fase di scrittura del codice è stato l'utilizzo degli *attributi di sintesi*. Questi decoratori testuali consentono di fornire direttive esplicite al sintetizzatore che altrimenti caricherebbe tutto nelle LUT, forzando le scelte architetturali per ottimizzare timing e area occupata. Ad esempio, per vincolare la mappatura dei banchi di memoria della S-Box e della memoria principale direttamente sui blocchi fisici BSRAM del chip Gowin, evitando il consumo inutile di migliaia di LUT, è stato applicato questo attributo:

```
(* ram_style = "block" *) reg [31:0] memory [0:8191];
```

4.3 Toolchain Open-Source

Il flusso operativo completo, dalla descrizione HDL fino alla configurazione fisica del dispositivo, è stato automatizzato mediante un sistema di *build* basato su file `Makefile`. Al contrario dello standard classico di sviluppo che obbliga l'uso di IDE proprietari (come Gowin EDA), il progetto si basa integralmente su una toolchain *Free and Open-Source Software* (FOSS). Questa scelta garantisce massima trasparenza del codice, possibile condivisione e miglioramento da parte di chiunque e indipendenza architetturale dal vendor di FPGA.

4.3.1 Yosys

La traduzione logica del codice è affidata a Yosys (Yosys Open SYnthesis Suite) [22], un framework avanzato per la sintesi RTL. Partendo dai sorgenti Verilog, Yosys elabora la struttura del codice, ottimizza le reti booleane combinatorie attraverso algoritmi algebrici ed esegue il cosiddetto *technology mapping*, generando come output una netlist strutturale formattata in JSON, in cui i concetti astratti del linguaggio risultano tradotti in primitive logiche native del chip Gowin GW1NR-9.

4.3.2 nextpnr

La netlist logica sintetizzata viene successivamente passata in ingresso a nextpnr [23], uno strumento di *Place and Route* (P&R) portatile e indipendente dal fornitore, focalizzato sull'ottimizzazione dei percorsi di temporizzazione. Il software si occupa prima di mappare fisicamente le primitive logiche all'interno della griglia bidimensionale dell'FPGA (*Place*), per poi calcolare l'attivazione ottimale delle matrici di instradamento per collegare elettricamente i nodi (*Route*), conformemente alle assegnazioni spaziali e ai vincoli fisici imposti nel file `.cst`. Il risultato è infine pacchettizzato nel file binario *bitstream* (`.fs`) tramite l'utility `gowin_pack`.

4.3.3 openFPGALoader

Per la configurazione hardware del dispositivo si è fatto ricorso a openFPGALoader [24], un'utility universale compatibile con le interfacce JTAG e USB-UART. Il tool è stato configurato per scrivere sia in modalità volatile direttamente sulla SRAM statica del chip (ideale per test rapidi), sia in modalità persistente sul chip di Flash SPI esterna, caricando il bitstream hardware a partire dall'indirizzo `0x000000` e il file binario del firmware software dall'offset `0x100000`.

4.3.4 Compilatore GCC e libreria Picolibc (riscv64-unknown-elf)

Sul fronte del codice software, l'eseguibile destinato a operare sul soft-core è stato implementato in linguaggio C e compilato mediante la GCC (*GNU Compiler Collection*) [25], configurata per il cross-compiling su architetture RISC-V a 32 bit (`-march=rv32i`). Al fine di limitare drasticamente la richiesta di memoria dal firmware (entro il limite dei 32 KB di BRAM), la libreria standard C è stata sostituita totalmente con *picolibc* [26], un set di librerie essenziali sviluppato appositamente per microcontrollori e target *bare-metal*.

4.3.5 Icarus Verilog e GTKWave

La fondamentale fase di testing e validazione delle funzioni logiche hardware, la compatiblità del software ed il rispetto del timing è stato effettuato tramite Icarus Verilog (*iverilog*) [27], un potente simulatore RTL event-driven capace di fare verifiche tramite *testbench*. Le transizioni di stato e le forme d'onda logiche in formato VCD prodotte dalla simulazione sono state analizzate temporalmente attraverso GTKWave [28], strumento di ispezione visiva cruciale per effettuare il debug dell'handshake del bus di memoria e misurare analiticamente la latenza interna espressa in cicli di clock dell'acceleratore AES.

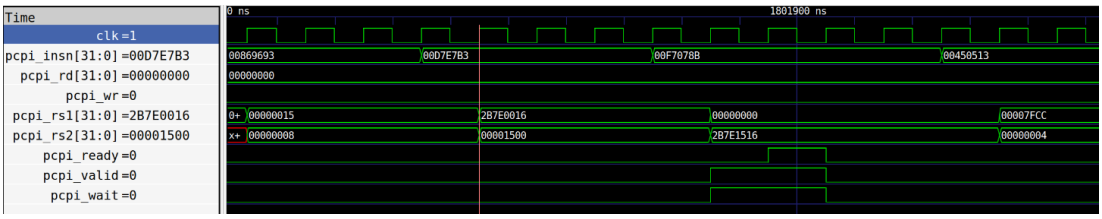

Figura 4.2: Forme d'onda in GTKWave.

4.4 Architettura RISC-V e PicoRV32

Il cuore gestionale del sistema incaricato di eseguire l'applicazione e di interfacciarsi con il modulo crittografico custom è basato sull'architettura RISC-V. RISC-V è un'Instruction Set Architecture (ISA) aperta, modulare e priva di royalty, nata nel 2010 all'interno dell'Università della California a Berkeley. Considerata la limitata disponibilità di spazio logico sul dispositivo Tang Nano 9K, è stata adottata la configurazione `RV32I`: una CPU operante strettamente su numeri interi a 32 bit basata su sole 37 istruzioni primarie, priva di estensioni pesanti in

silicio come i moltiplicatori hardware nativi (estensione M) o logica per numeri a virgola mobile.

Il processore instanziato nel progetto è il PicoRV32 [29], un soft-core RISC-V di classe embedded rigorosamente open-source (sotto licenza ISC), sviluppato dallo stesso autore di Yosys, Clifford Wolf. Il design di PicoRV32 sacrifica intenzionalmente le massime prestazioni per ciclo di clock allo scopo di ridurre radicalmente le dimensioni in termini di celle LUT richieste, rendendolo ottimale per FPGA a ridotta densità e sistemi SoC compatti.

```
picorv32 #(
    .ENABLE_PCPI    (1),
    .ENABLE_MUL     (0),
    .ENABLE_DIV     (0),
    .BARREL_SHIFTER (1),
    .COMPRESSED_ISA (0)
) cpu (
    .clk      (clk),        .resetn   (resetn),
    .mem_valid(mem_valid), .mem_instr(mem_instr),
    .mem_ready(mem_ready), .mem_addr (mem_addr),
    .mem_rdata(mem_rdata), .mem_wdata(mem_wdata),
    .mem_wstrb(mem_wstrb),
    .pcpi_valid(pcpi_valid), .pcpi_insn (pcpi_insn),
    .pcpi_rs1  (pcpi_rs1),   .pcpi_rs2  (pcpi_rs2),
    .pcpi_wr   (pcpi_wr),    .pcpi_rd   (pcpi_rd),
    .pcpi_wait (pcpi_wait),  .pcpi_ready(pcpi_ready)
);
```

Snippet 4.1: Istanziazione in Verilog del soft-core PicoRV32.

4.4.1 Pico Co-Processor Interface (PCPI)

L'aspetto fondamentale che ha motivato la scelta di questo specifico processore rispetto ad altre alternative risiede nella *Pico Co-Processor Interface (PCPI)*. Questo bus nativo permette l'espansione e la personalizzazione estrema dell'Instruction Set della CPU inserendo logica di accelerazione esterna, senza imporre alcuna modifica invasiva al delicato codice sorgente interno della pipeline del processore.

Il protocollo opera secondo un semplice paradigma a *handshake*. Ogni volta che l'ALU di decodifica della CPU incontri una macro-istruzione software non standard e irriconoscibile (ad esempio, una specifica istruzione custom `.insn` inserita nel codice C con opcode `0x0B`), questa evita di lanciare un'eccezione di errore software. Al contrario, la CPU entra in uno stato di stallo hardware, congelando l'avanzamento dei cicli macchina, ed instrada il pacchetto di informazioni verso

l'esterno attivando il segnale logico `pcpi_valid`. L'intero payload dell'istruzione e i valori attuali dei registri generici impiegati (`pcpi_rs1` e `pcpi_rs2`) diventano elettricamente disponibili sui pin di uscita dell'interfaccia PCPI.

Da questo istante, la gestione passa integralmente all'acceleratore. Ultimato il proprio lavoro il coprocessore alza in uscita la linea `pcpi_ready`, indicando alla CPU di poter uscire dallo stallo, e deposita il risultato crittografico calcolato sul bus dati `pcpi_rd`, azionando il pin `pcpi_wr` per forzarne la scrittura definitiva in un registro di destinazione all'interno del core RISC-V.

Questa metodologia di interfacciamento, a differenza di un'implementazione basata su memory-mapped I/O, garantisce una maggiore flessibilità e personalizzazione a livello software. Inoltre non richiede l'implementazione di un ulteriore controller di istradamento degli indirizzi e garantisce il rispetto del timing evitando possibili trap ed eccezioni hardware.

Capitolo 5

Implementazione

In questo capitolo vengono descritti tutti i componenti progettati ed implementati per la realizzazione di un sistema di crittografia hardware gestito dalla cpu soft-core PicoRV. Verrà quindi dettagliato il codice del Makefile, i moduli hardware scritti in verilog e il codice C e Assembly che compone il firmware bare metal. L'implementazione completa è disponibile nel repository github [30] .

5.1 Makefile

Il makefile è stato sviluppato in modo da gestire tutto in modo centralizzato e garantire la massima flessibilità permettendo di operare separatamente su tutte le fasi del flusso di lavoro dalla compilazione alla sintesi e al caricamento.

```
$ make
Target disponibili:
  make build    compila il software, produce ELF, BIN e HEX
  make sim      lancia la simulazione Icarus (richiede build
   )
  make fpga     sintetizza per Tang Nano 9K
  make firmware carica program.bin in flash esterna (
   richiede build)
  make program  programma la FPGA in SRAM (richiede fpga)
  make programf programma la FPGA in flash esterna (richiede
    fpga)
  make wave     apre GTKWave sul .vcd
  make clean    rimuove tutti i file generati
  make info     verifica gli strumenti installati
```

Snippet 5.1: Lista dei target.

5.1.1 Variabili principali

Nel file vengono definite inizialmente una serie di variabili con le specifiche di compilazione, le librerie da usare, la struttura della directory e la tipologia di board FPGA.

```
# Toolchain software RISC-V
CROSS   := riscv64-unknown-elf
CC      := $(CROSS)-gcc
OBJCOPY := $(CROSS)-objcopy
OBJDUMP := $(CROSS)-objdump

CFLAGS  := -march=rv32i -mabi=ilp32 -O1 -g -Wall -nostdlib -
    nostartfiles
LDFLAGS := -T sw/link.ld -Wl,--gc-sections --specs=picolibc.
    specs
LIBS    := -lc -lm -lgcc

SW_SRCS := sw/start.S sw/syscalls.c sw/main.c
ELF     := sw/program.elf
BIN     := sw/program.bin
HEX     := sw/program.hex

# Toolchain FPGA
TOP    := fpga/top
DEVICE := GW1NR-LV9QN88PC6/I5
FAMILY := GW1N-9C

HW_SRCS := hw/top.v hw/uart_tx.v hw/picorv32.v hw/my_pcpi.v
    hw/bootloader.v
```

Snippet 5.2: Variabili Makefile.

```
.
├── Makefile          # Build system unificato (sw + fpga + sim)
├── hw/
│   ├── top.v         # Top-level: reset POR, bootloader, bus, CPU, AES
│   ├── picorv32.v    # Soft-core RISC-V (upstream, non modificato)
│   ├── my_pcpi.v     # Acceleratore AES via PCPI (+ aes_core, sbox, keysched)
│   ├── bootloader.v  # Bootloader SPI: carica firmware dalla flash
│   ├── uart_tx.v     # Trasmettitore UART 8N1
│   └── tb.v          # Testbench Icarus Verilog
├── sw/
│   ├── main.c        # Test AES: 3 vettori incluso NIST FIPS-197
│   ├── syscalls.c    # Stub picolibc: printf → UART, heap, _exit
│   ├── start.S       # Entry point ASM: stack, BSS, jump a main
│   └── link.ld       # Linker script: BRAM 32KB, stack, heap
└── fpga/
    └── tang9k.cst    # Constraint file: pin assignment Tang Nano 9K
```

Figura 5.1: Struttura directory.

5.1.2 Implementazione dei target

Build

Si occupa della compilazione del codice sorgente prima in elf e poi in due formati diversi: .bin per essere caricato in memoria e .hex per essere utilizzato in simulazione che ha una word per riga in esadecimale, il formato che `readmemh` in Verilog si aspetta nel testbench. Il file .lst (disassembly annotato con il sorgente C) è utile per debug: si può verificare che il compilatore abbia effettivamente emesso l'istruzione custom .insn r 0x0B.

```
1 $(ELF): $(SW_SRCS) sw/link.ld
2     $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $(SW_SRCS) $(LIBS)
3     $(OBJDUMP) -d -S $@ > $(LST)    # genera anche il
    disassembly .lst
4
5 $(HEX): $(ELF)
6     $(OBJCOPY) -O binary $< $(BIN)
7     od -An -tx4 -v $(BIN) | awk '{for(i=1;i<=NF;i++) print
    $$i}' > $@
8
9 build: $(HEX)
```

Snippet 5.3: make build.

Sim e Wave

La simulazione usa tb.v che carica il firmware direttamente con readmemh, non c'è bootloader, la CPU parte subito. L'output printf appare direttamente sul terminale tramite la UART fake del testbench. GTKWave apre il file hw/sim.vcd generato dalla simulazione, utile per leggere i segnali PCPI ciclo per ciclo.

```
SIM_SRCS := hw/picorv32.v hw/my_pcpi.v hw/tb.v

sim: $(HEX) $(SIM_SRCS)
    iverilog -g2012 -o $(SIM_BIN) $(SIM_SRCS)
    cd hw && vvp sim

wave:
    gtkwave $(VCD) &
```

Snippet 5.4: make sim e make wave.

Fpga

Questa parte si occupa della sintesi del bitstream. Questa avviene in 3 passi concatenati ed attraverso la toolchain descritta in precedenza. Il flag –mspi_as_gpio in gowin_pack libera i pin SPI del chip che sarebbero altrimenti riservati alla configurazione, è necessario perché il progetto usa quei pin per la flash esterna.

```
# Passo 1: Yosys - RTL Verilog -> netlist JSON
$(TOP).json: $(HW_SRCS)
    yosys -p synth_gowin -top top -json $(TOP).json $(
   HW_SRCS)

# Passo 2: nextpnr - Place & Route con constraint file
$(TOP).pack: $(TOP).json
    /usr/local/bin/nextpnr-himbaechel \
        --device $(DEVICE) \
        --vopt family=$(FAMILY) \
        --vopt cst=fpga/tang9k.cst \
        --json $(TOP).json \
        --write $(TOP).pack

# Passo 3: gowin_pack - genera il bitstream .fs
$(TOP).fs: $(TOP).pack
    ~/.local/bin/gowin_pack -d $(FAMILY) --mspi_as_gpio -o $
   (TOP).fs $(TOP).pack

```

```
fpga: $(TOP).fs
```

Snippet 5.5: make fpga.

Program, programf e firmware

Questi tre target struttano il programma openFPGALoader per scrivere all'interno delle memorie della board. Il primo carica il bitstream direttamente nelle celle di S-RAM utile per il testing e la prototipazione in quanto molto più veloce ma volatile. Gli altri due scrivono sulla flash esterna Puya rispettivamente il bitstream che è la prima cosa che viene letta all'indirizzo `0x0` ed il firmware all'indirizzo `0x100000` che verrà cercato e caricato dal bootloader.

```
# Volatile: scrive il bitstream in SRAM, si perde al riavvio
program: fpga
    openFPGALoader -b tangnano9k $(TOP).fs

# Persistente: scrive il bitstream in flash esterna (primi 1 MB)
programf: fpga
    openFPGALoader -b tangnano9k --external-flash $(TOP).fs

firmware: build
    openFPGALoader -b tangnano9k --external-flash -o 0x100000 $(BIN)
```

Snippet 5.6: make program, make programf e make firmware.

Il layout della flash SPI è quindi:

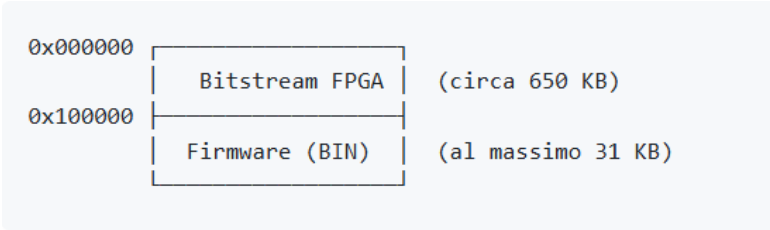

Figura 5.2: Organizzazione flash SPI.

5.2 Moduli hardware

Di seguito verranno descritti tutti i file .v Verilog e i moduli presenti all'interno. La sezione si divide in due parti, la prima tratta di tutti i componenti che orchestrano il *layout* e gestiscono il flusso di dati, l'interazione con il mondo esterno e

il normale funzionamento della CPU. Nella seconda invece viene dettagliato con precisione il co-processore per la crittografia AES che è appunto la parte centrale di tutto il progetto.

5.2.1 Moduli di gestione

L'architettura di questa parte del progetto è stata organizzata in modo tale che ogni file contenga la definizione di un singolo modulo. Quindi in questa parte della trattazione i termini 'file' e 'modulo' verranno utilizzati in modo intercambiabile.

top.v

Questo file rappresenta il modulo principale (top-level) dell'intero sistema hardware. Il suo compito primario è orchestrare la sequenza di avvio, che si divide in tre fasi: un reset esteso (Power-On-Reset) per garantire la stabilità elettrica, l'esecuzione del bootloader per caricare il firmware nella BRAM, e infine l'avvio della CPU.

All'interno di questo modulo viene infatti istanziato il core centrale del sistema, il processore PicoRV32 il quale non verrà incluso nella trattazione in quanto modulo open-source scaricato e descritto in precedenza. Diventa invece fondamentale evidenziare come nel `top.v` avvenga l'abilitazione e l'intero cablaggio dell'interfaccia proprietaria `PCPI` (*Pico Co-Processor Interface*) esposta dal core. Questo bus permette alla CPU di delegare istruzioni *custom* direttamente a un modulo hardware esterno, fungendo da ponte per il collegamento dell'acceleratore crittografico AES, anch'esso istanziato all'interno del `top.v`.

Inoltre, il `top.v` definisce il *memory mapping* (mappa degli indirizzi) e agisce da bus controller generale. Indirizza le richieste di memoria della CPU verso la BRAM (indirizzi da `0x00000000`), verso la periferica UART per la trasmissione seriale (`0x10000000`) oppure verso un registro fittizio di uscita (`0x20000000`) utilizzato per segnalare il termine del programma e accendere un LED.

Di seguito è riportato il frammento del controllore del bus che si occupa di gestire la mappatura degli indirizzi principali e intercettare le scritture verso la UART, lo schema dei collegamenti in Figura 5.3 ed una tabella riassuntiva in Figura 5.4:

```
// Mappa indirizzi
localparam UART_ADDR   = 32'h1000_0000;
localparam EXIT_ADDR   = 32'h2000_0000;
// ...
// Intercettazione delle richieste della CPU
if (mem_addr == EXIT_ADDR) begin
    led       <= 0;     // Programma completato
    mem_ready <= 1;
```

```
9  end
10 else if (mem_addr == UART_ADDR && |mem_wstrb) begin
11     uart_byte_latch <= mem_wdata[7:0]; // Cattura il byte da
       trasmettere
12     state           <= S_UART_WAIT;
13 end
14 // ...
```

Snippet 5.7: Bus controller e memory mapping nel modulo Top.

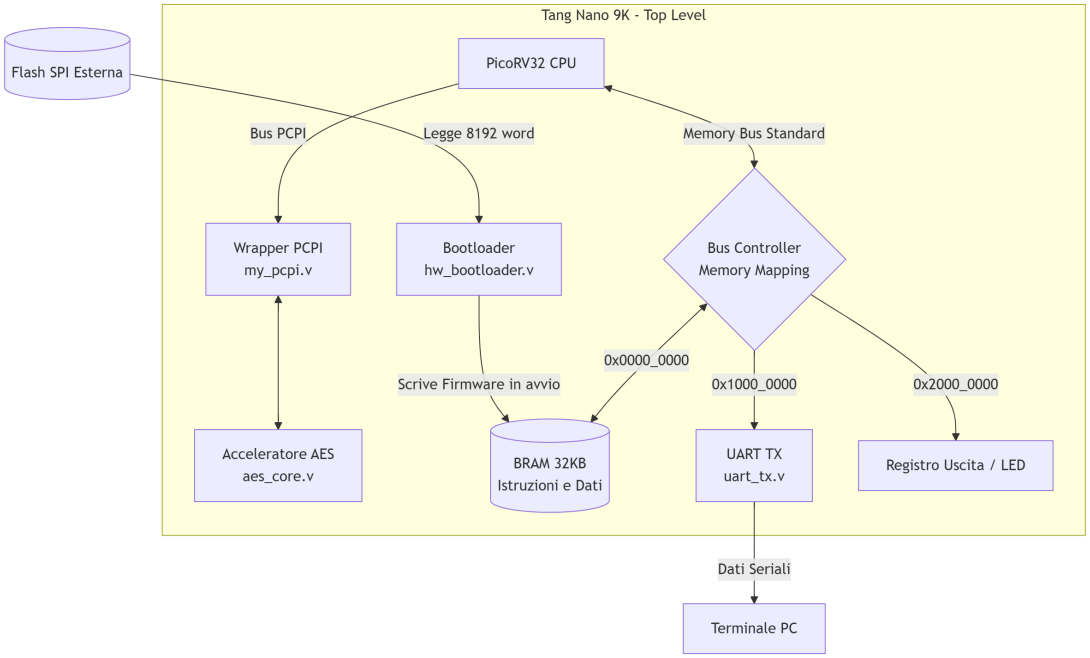

Figura 5.3: Schema dei collegamenti del modulo Top.

Indirizzo	Periferica	Comportamento
`< 0x8000`	BRAM	Lettura/scrittura con 1 ciclo di latenza
`0x1000_0000`	UART TX	Scrittura byte → attende UART libera → trasmette
`0x2000_0000`	EXIT	Accende il LED; in simulazione termina la sim
Qualsiasi altro	—	Risponde `0xDEAD_BEEF`

Figura 5.4: Tabella riassuntiva indirizzi.

bootloader.v

Il modulo `hw_bootloader` è fondamentale per l'inizializzazione del sistema. Immediatamente dopo l'accensione, questo componente si interfaccia tramite protocollo SPI con la memoria flash esterna della scheda Tang Nano 9K. Il suo scopo è leggere il file binario del firmware (eseguendo una lettura burst di 8192 word a partire da uno specifico offset) e scriverlo direttamente nella BRAM della CPU.

Una particolarità importante gestita da questo modulo è la conversione dell'endianness: la memoria flash trasmette i dati in formato *big-endian*, mentre l'architettura RISC-V richiede un formato *little-endian*.

```
localparam FLASH_OFFSET  = 24'h100000;
localparam WORDS_TO_READ = 8192;
// ...
S_LATCH_DATA: begin
    // Inversione byte per compatibilita little-endian RISC-V
    spi_clk   <= 0;
    ram_wdata <= {shift_reg[7:0], shift_reg[15:8],
                  shift_reg[23:16], shift_reg[31:24]};
    state <= S_DO_WRITE;
end
S_DO_WRITE: begin
    ram_we     <= 1;
    word_count <= word_count + 1;
    if (word_count == WORDS_TO_READ - 1) state <= S_DONE;
    else                                 state <= S_INC_ADDR;
end
```

Snippet 5.8: Macchina a stati del bootloader: conversione endianness e scrittura in BRAM.

uart_tx.v

Il file `uart_tx.v` implementa un semplice trasmettitore seriale asincrono (UART) configurato per il formato standard 8N1 (8 bit di dati, nessuna parità, 1 bit di stop). Il modulo riceve i dati paralleli dal `top.v` e utilizzando uno shift register e una macchina a stati finiti, li trasmette in seriale rispettando il baud rate configurato (determinato dal parametro `CLKS_PER_BIT`).

```
DATA: begin
```

```
2      o_tx <= shift_reg[0]; // Trasmette il bit meno
       significativo
3      if (clk_cnt == CLKS_PER_BIT - 1) begin
4          clk_cnt   <= 0;
5          shift_reg <= shift_reg >> 1; // Shift per il bit
       successivo
6          if (bit_cnt == 7)
7              state <= STOP;
8          else
9              bit_cnt <= bit_cnt + 1;
10     end else
11         clk_cnt <= clk_cnt + 1;
12 end
```

Snippet 5.9: Macchina a stati della UART: trasmissione del payload.

tb.v (Testbench)

Anche se non sia un modulo sintetizzabile su FPGA, il file `tb.v` è essenziale per la gestione e il testing del sistema. Istanziando la CPU, il co-processore AES e simulando la RAM (caricata dinamicamente dal file `program.hex`), permette di verificare la corretta esecuzione del codice prima della sintesi. Il testbench integra inoltre una "UART fittizia" che intercetta le scritture all'indirizzo `0x10000000` per stamparle direttamente sulla console del simulatore, consentendo l'utilizzo e la visualizzazione delle funzioni `printf` del linguaggio C.

```
1  // ...
2  // RAM 32KB caricata dinamicamente dal firmware compilato
3  reg [31:0] memory [0:8191];
4  initial begin
5      $readmemh("../sw/program.hex", memory);
6  end
7  // ...
8
9  // Istanza CPU (PicoRV32) con PCPI abilitato
10 picorv32 #(
11     .ENABLE_PCPI    (1),
12     // ...
13 ) cpu (
14     .clk      (clk),
15     // ...
16 );
17
18 // Istanza co-processore crittografico AES
```

```
my_pcpi pcpi_inst (
    // ...
);

// ...
localparam UART_ADDR = 32'h1000_0000;
// ...

// Gestione del bus di memoria emulato
always @(posedge clk) begin
    // ...
    if (mem_valid && !mem_ready) begin
        // ...
        // UART fake: intercetta le scritture a 0x10000000
        else if (mem_addr == UART_ADDR) begin
            if (|mem_wstrb)
                $write("%c", mem_wdata[7:0]); // Stampa a
   console per la printf()
            mem_ready <= 1;
        end
        // ...
    end
end
```

Snippet 5.10: Parti fondamentali del Testbench con emulazione RAM e Fake UART.

5.2.2 Acceleratore AES

Questa sezione descrive l'architettura interna e le scelte progettuali relative all'acceleratore hardware per l'algoritmo di cifratura AES-128. L'intero coprocessore crittografico è implementato nel file sorgente `my_pcpi.v`, strutturato secondo una gerarchia di moduli annidati: il wrapper di interfaccia bus (`my_pcpi`), il nucleo di calcolo iterativo (`aes_core`), l'unità di espansione della chiave (`aes_key_expansion`), lo stadio combinatorio di diffusione (`aes_mixcolumn`) e le memorie sincrone per la sostituzione dei byte (`aes_sbox_bram`).

my_pcpi (Wrapper e protocollo di handshake PCPI)

Il modulo `my_pcpi` agisce da stazione di decodifica e smistamento dati tra il bus proprietario `PCPI` della CPU PicoRV32 e il core crittografico. Quando il processore incontra nel flusso d'esecuzione l'istruzione custom (identificata dall'opcode `0x0B`, ossia `7'b0001011`), attiva la linea `pcpi_valid`.

La complessità di questo modulo risiede nella corretta gestione del protocollo di controllo. Per evitare corse critiche o disallineamenti del bus, il wrapper implementa una macchina a stati finiti (FSM) a 4 stati (`0` a `3`) che esegue un preciso schema di *handshake*:

1. **Stato 0 (IDLE):** Il modulo si trova in attesa passiva sul bus. Quando la CPU invia un'istruzione valida (`start` alto), la FSM esamina i due bit meno significativi del registro sorgente (`pcpi_rs1[1:0]`) per determinare il comando richiesto.

   Se il comando è di tipo I/O istantaneo (valori `0`, `1` o `3`), non è necessario attivare il core crittografico. In questi casi, la FSM esegue l'operazione in un singolo ciclo di clock:

   - **Comandi 0 e 1 (WRITE_KEY e WRITE_TEXT):** La word a 32 bit presente sul bus dati (`pcpi_rs2`) viene scritta direttamente in uno dei 4 registri hardware che compongono la chiave o il testo in chiaro. La scelta del registro esatto (demultiplexing) è determinata dall'indice `idx` (`pcpi_rs1[3:2]`). Effettuata la scrittura, la FSM alza `pcpi_ready` e salta allo Stato 3.
   - **Comando 3 (READ_TEXT):** La logica esegue l'operazione inversa. Seleziona la word corretta di ciphertext tramite l'indice `idx`, la espone sul bus di ritorno (`pcpi_rd`), alza il segnale di validità del dato in lettura (`pcpi_wr`) insieme a `pcpi_ready`, e passa allo Stato 3.

   Se invece viene decodificato il comando **2 (START)**, l'operazione richiede un tempo di calcolo prolungato. La FSM non alza il segnale di ready, ma passa allo Stato 1 per innescare la computazione iterativa dell'acceleratore.

2. **Stato 1 (START_AES):** Genera un impulso sincrono isolato (`aes_start <= 1`) della durata di un solo ciclo di clock per svegliare il core AES, muovendosi immediatamente allo Stato 2.

3. **Stato 2 (WAIT_DONE):** L'acceleratore hardware è in esecuzione autonoma. Il wrapper azzera l'impulso di start e rimane in attesa del segnale `aes_done`. Al suo arrivo, cattura l'output a 128 bit (`aes_out`) e lo carica nei registri di testo locali, alza il flag di completamento bus `pcpi_ready <= 1` e passa allo Stato 3.

4. **Stato 3 (WAIT_RESET):** Questo stato agisce come un blocco di sicurezza per chiudere correttamente la comunicazione ed evitare il problema del "doppio avvio". Quando l'acceleratore termina un'operazione, avvisa la CPU alzando la linea `pcpi_ready`. Tuttavia, la CPU ha bisogno di un intero ciclo di clock per leggere il risultato e abbassare fisicamente il segnale di avvio (`pcpi_valid`). Se la macchina a stati tornasse immediatamente

in ascolto (Stato 0), leggerebbe il segnale ancora alto e crederebbe di aver appena ricevuto un nuovo comando, eseguendolo due volte di fila per errore. Lo Stato 3, quindi, congela il modulo finché la CPU non abbassa il segnale, garantendo una sincronizzazione perfetta.

Un punto chiave per il coordinamento tra CPU e co-processore è la linea `pcpi_wait`: essa è implementata per via puramente combinatoria (`always @(*) pcpi_wait = start;`). In questo modo, non appena la CPU presenta un'istruzione valida, la linea di stallo sale nello stesso ciclo di clock, congelando il *Program Counter* del processore prima che possa avanzare a vuoto.

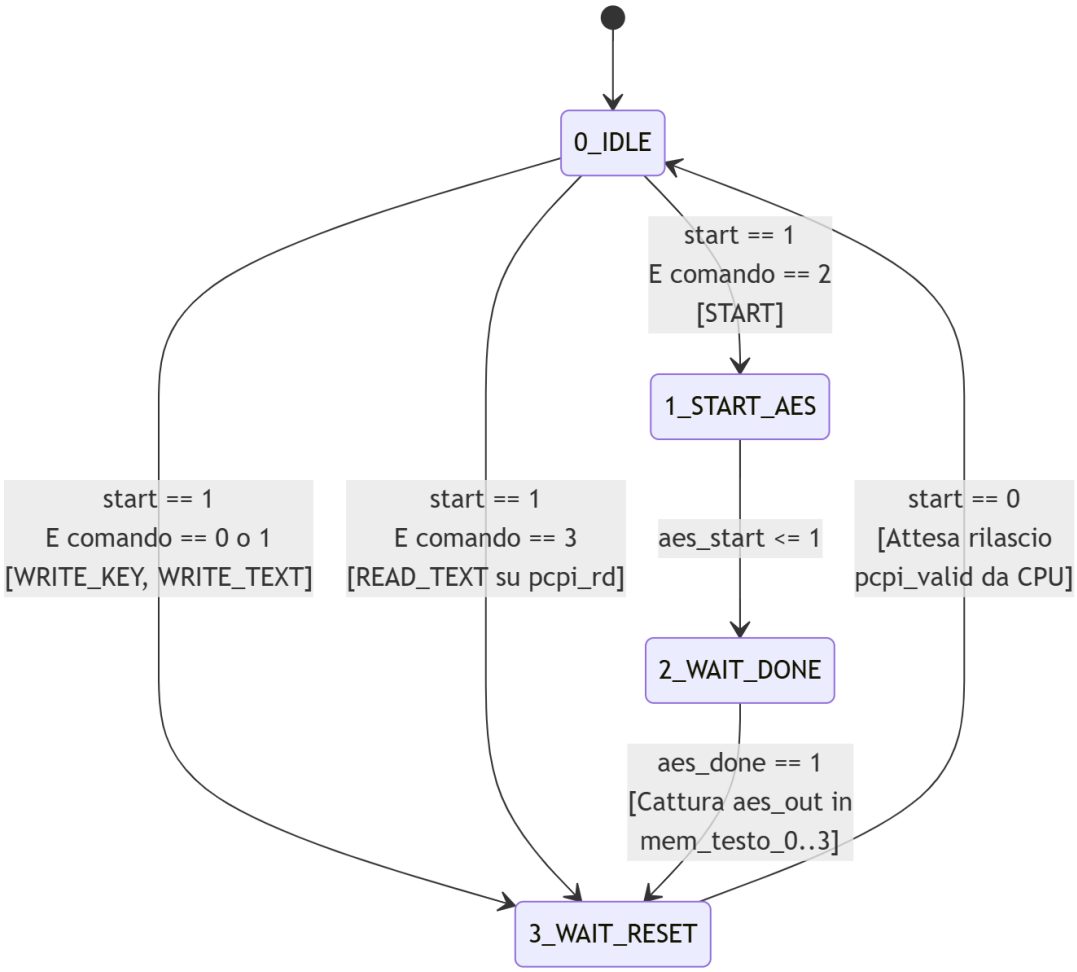

Figura 5.5: Macchina a stati del modulo wrapper.

Tabella 5.1: Codifica dei comandi accettati dal wrapper PCPI.

Comando (bit 1:0)	Operazione	Comportamento
0	`WRITE_KEY`	Scrive una word nel registro chiave all'indice `idx`
1	`WRITE_TEXT`	Scrive una word nel registro plaintext all'indice `idx`
2	`START`	Avvia l'algoritmo iterativo di cifratura sul core AES
3	`READ_TEXT`	Sulla linea `pcpi_rd` espone la word di ciphertext richiesta con idx

```
always @(posedge clk or negedge resetn) begin
    if (!resetn) begin
        stato <= 0;
        pcpi_ready <= 0; pcpi_wr <= 0; pcpi_rd <= 0;
    aes_start <= 0;
        // ... azzeramento registri interni ...
    end else begin
        pcpi_ready <= 0; pcpi_wr <= 0;

        case (stato)
            0: begin
                aes_start <= 0;
                if (start) begin
                    if (comando == 2'd0) begin // WRITE_KEY
                        // ... logica di scrittura demux su
    idx ...
                        pcpi_ready <= 1; stato <= 3;
                    end else if (comando == 2'd1) begin //
    WRITE_TEXT
                        // ... logica di scrittura demux su
    idx ...
                        pcpi_ready <= 1; stato <= 3;
                    end else if (comando == 2'd2) begin //
    START_AES
                        stato <= 1; // Richiede computazione
     iterativa
                    end else if (comando == 2'd3) begin //
    READ_TEXT
                        pcpi_wr    <= 1; pcpi_ready <= 1;
                        // ... esposizione della word su
    pcpi_rd tramite case(idx) ...
                        stato <= 3;
                    end
```

```
26                  end
27              end
28              1: begin
29                  aes_start <= 1; // Impulso di avvio per il
    core
30                  stato     <= 2;
31              end
32              2: begin
33                  aes_start <= 0; // Rimozione immediata dell'
    impulso
34                  if (aes_done) begin
35                      // Cattura in little-endian dell'output
    a 128 bit
36                      mem_testo_0 <= aes_out[127:96];
37                      mem_testo_1 <= aes_out[95:64];
38                      mem_testo_2 <= aes_out[63:32];
39                      mem_testo_3 <= aes_out[31:0];
40                      pcpi_ready  <= 1; stato <= 3;
41                  end
42              end
43              3: begin
44                  if (!start) stato <= 0; // Ritorna in IDLE
    solo quando la CPU rilascia il bus
45              end
46          endcase
47      end
48  end
```

Snippet 5.11: Logica di transizione della FSM di handshake del modulo `my_pcpi`.

aes_core (Architettura di computazione ed analisi dei cicli)

Il modulo `aes_core` implementa l'architettura iterativa dell'algoritmo di cifratura. Invece di srotolare i round nello spazio (soluzione che avrebbe saturato le risorse logiche della piccola FPGA), il modulo riutilizza le stesse risorse hardware nel tempo per 10 volte consecutive. L'orchestrazione è affidata a una FSM interna organizzata su 7 stati logici.

Il problema più complesso riscontrato in questo modulo riguarda il blocco *SubBytes*. La trasformazione non lineare richiede la sostituzione di tutti i 16 byte dello stato. Poiché l'inferenza delle S-Box sulle memorie dedicate Gowin (`aes_sbox_bram`) impone una lettura sincrona vincolata al clock (1 ciclo di latenza), un'architettura interamente parallela avrebbe richiesto 16 istanze di BRAM distinte, superando le risorse a disposizione sulla scheda.

Il design adotta quindi un approccio misto basato su un datapath a 32 bit, in perfetta linea con le architetture ripiegate (*folded*) ottimizzate per FPGA introdotte in letteratura da Chodowiec e Gaj [15]. Vengono istanziate 4 *S-Box in parallelo*, dividendo la matrice dello stato AES in 4 colonne. La fase di *SubBytes* viene così serializzata in una sequenza interna di 4 step ripetitivi gestiti dal contatore `sub_cnt`: ad ogni ciclo un multiplexer combinatorio seleziona un gruppo di 4 byte e lo invia alle memorie; nel ciclo successivo, per via della latenza di sincronizzazione, i dati convertiti vengono memorizzati nel registro d'appoggio `subbytes_reg`

Un ulteriore aspetto critico affrontato durante la progettazione è stato quello del *critical path* (percorso critico). Poiché le trasformazioni crittografiche generano reti logiche combinatorie molto profonde, eseguirle interamente in un singolo ciclo di clock avrebbe comportato tempi di propagazione elettrica troppo lunghi, causando un crollo drastico della frequenza massima operativa. Di conseguenza, le operazioni sono state spezzate su più stati sequenziali, allungando la pipeline a vantaggio della stabilità del sistema.

L'esatta scomposizione temporale e il comportamento degli stati della FSM del core determinano una latenza fissa pari a 103 cicli di clock per blocco cifrato, divisa in questo modo:

- **IDLE e WAIT_KEY (2 cicli):** Al ricevimento del segnale `start`, la FSM predispone il modulo di espansione della chiave (`key_init = 1`) e attende che la prima porzione del *Key Schedule* sia allineata.
- **INIT (1 ciclo):** Esegue l'operazione iniziale di *AddRoundKey* (XOR combinatorio tra il plaintext di ingresso e la chiave master di round 0). Il contatore dei round viene inizializzato a 1 (`round_counter <= 1`).
- **Loop dei Round 1–9 (9 round × 10 cicli = 90 cicli):** Ciascun round intermedio richiede esattamente 10 cicli di clock per completare le quattro trasformazioni canoniche:
  - `CALC_SUB` **(6 cicli):** Il contatore `sub_cnt` evolve da 0 a 4. Nei primi 4 cicli invia le colonne alle S-Box. A causa della latenza della Block RAM, i byte sostituiti della quarta colonna sono disponibili e salvati in `subbytes_reg` solo al sesto ciclo. Al completamento, viene attivata l'unità di calcolo della chiave successiva (`key_enable <= 1`).
  - `WAIT_EXPANSION` **(2 cicli):** Stato di stallo hardware necessario per attendere che il modulo `aes_key_expansion` termini la computazione della nuova chiave di round (la quale richiede anch'essa l'accesso sincrono alle S-Box dedicate del Key Schedule).
  - `CALC_P1` **(1 ciclo):** Effettua il latching dei dati nello stadio intermedio `state_mid` applicando la trasformazione `ShiftRows`. Trattandosi

di una pura rilocazione di indici di bit, questa operazione è cablata direttamente in hardware (*rewiring*) tramite un costrutto `assign`, consumando zero componenti logiche e risolvendosi istantaneamente.

- `CALC_P2` **(1 ciclo):** Alimenta le 4 colonne di `state_mid` all'interno di 4 moduli paralleli combinatori `aes_mixcolumn`. L'output risultante viene combinato in XOR con la chiave di round corrente (*AddRoundKey*) e riversato nuovamente nel registro principale `state` per l'iterazione successiva.

- **Round Finale 10 (10 cicli):** L'ultimo round segue la stessa temporizzazione dei precedenti per le fasi di SubBytes ed espansione chiave (8 cicli). Nello stadio `CALC_P2`, tuttavia, lo standard AES prevede l'omissione della trasformazione di MixColumns. La FSM esegue quindi lo XOR di *AddRoundKey* direttamente sul flusso proveniente dallo `ShiftRows` (latchato in `CALC_P1`), deposita il ciphertext finale in `data_out`, alza la linea `done <= 1` per notificare il wrapper e ritorna in `IDLE`.

Di seguito vengono riportati il diagramma di flusso delle operazioni svolte dal core in Figura 5.6 e la tabella degli stati nella Figura 5.7.

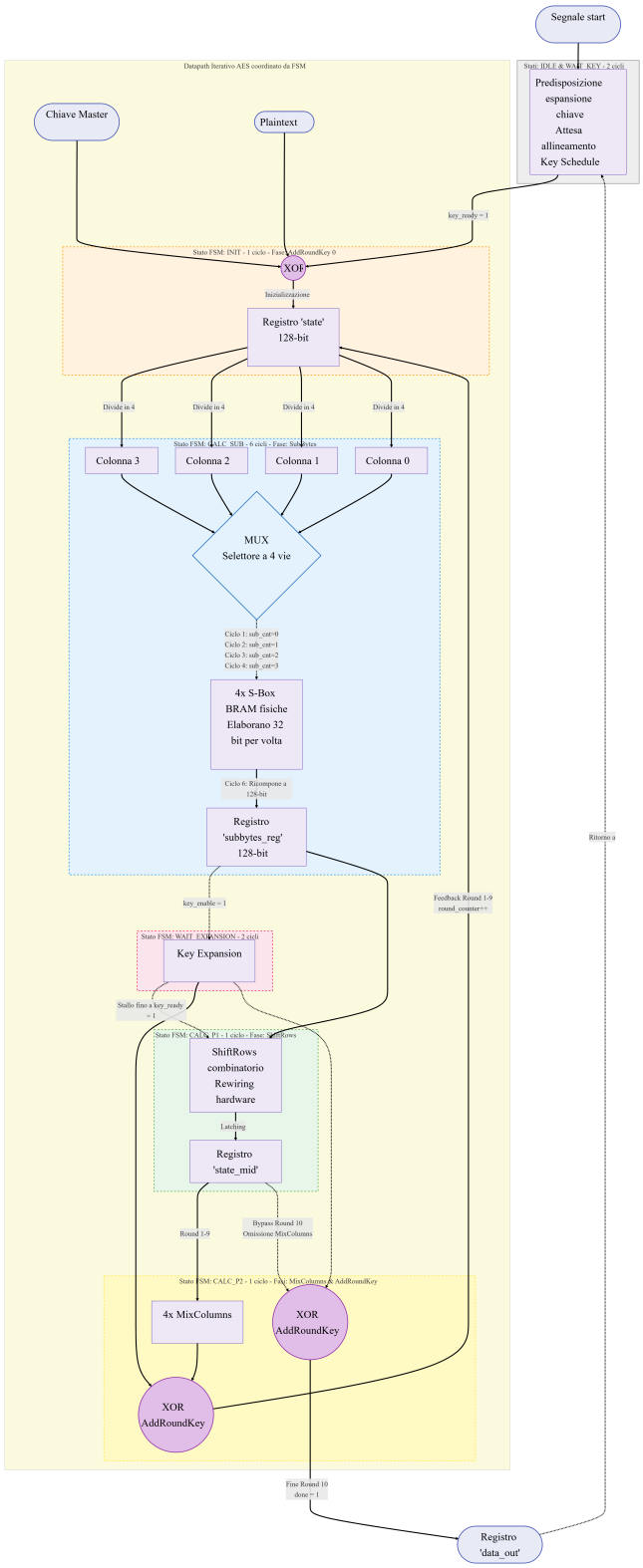

Figura 5.6: Diagramma di flusso modulo aes_core.

Fase	Cicli
`WAIT_KEY` (init key schedule)	2
`INIT` (AddRoundKey iniziale)	1
Per ciascuno dei 10 round:	
— `CALC_SUB` (SubBytes, 4 step + latenza BRAM)	6
— `WAIT_EXPANSION` (key schedule)	2
— `CALC_P1` (ShiftRows latch)	1
— `CALC_P2` (MixColumns + AddRoundKey)	1
Totale	**103**

Figura 5.7: Tabella degli stati del modulo aes_core e latenza di esecuzione.

```
// ... all'interno del case (stato_attuale) della FSM dell'aes_core ...
CALC_SUB: begin
    // Cattura differita dei dati in uscita dalle BRAM (latenza di 1 ciclo)
    if (sub_cnt > 0 && sub_cnt <= 4) begin
        case (sub_cnt - 1)
            3'd0: subbytes_reg[31:0]   <= {sb_out_w[3], sb_out_w[2], sb_out_w[1], sb_out_w[0]};
            3'd1: subbytes_reg[63:32]  <= {sb_out_w[3], sb_out_w[2], sb_out_w[1], sb_out_w[0]};
            3'd2: subbytes_reg[95:64]  <= {sb_out_w[3], sb_out_w[2], sb_out_w[1], sb_out_w[0]};
            3'd3: subbytes_reg[127:96] <= {sb_out_w[3], sb_out_w[2], sb_out_w[1], sb_out_w[0]};
        endcase
    end

    // Controllo avanzamento: quando la quarta colonna e' campionata
    if (sub_cnt == 4) begin
        if (round_counter < 11) begin
```

```
16              key_enable    <= 1; // Sveglia il Key Schedule
    per il calcolo parallelo
17              stato_attuale <= WAIT_EXPANSION;
18          end
19      end else begin
20          sub_cnt <= sub_cnt + 1;
21      end
22  end
23
24  WAIT_EXPANSION: begin
25      if (key_ready) stato_attuale <= CALC_P1; // Attende la
    sincronizzazione della chiave
26  end
27
28  CALC_P1: begin
29      state_mid     <= shiftrows_out; // Applica lo ShiftRows
    cablato a costo zero
30      stato_attuale <= CALC_P2;
31  end
32
33  CALC_P2: begin
34      if (round_counter < 10) begin
35          // Round standard: MixColumns attivo + AddRoundKey
    con chiave appena espansa
36          state         <= mixcols_out ^ round_key;
37          round_counter <= round_counter + 1;
38          sub_cnt       <= 0;
39          stato_attuale <= CALC_SUB;
40      end else begin
41          // Round 10: Salta MixColumns, XOR diretto sullo
    ShiftRows (state_mid)
42          data_out      <= state_mid ^ round_key;
43          done          <= 1; // Notifica di fine operazione
    al wrapper PCPI
44          stato_attuale <= IDLE;
45      end
46  end
```

Snippet 5.12: Sezione della FSM del modulo aes_core.

aes_key_expansion (Key Schedule)

Il modulo `aes_key_expansion` è incaricato di generare dinamicamente le chiavi di round (*Round Keys*) partendo dalla chiave master a 128 bit. Al fine di minimizzare l'utilizzo di memoria e logica, il *Key Schedule* non pre-calcola tutte le

chiavi all'avvio, ma avanza calcolando una singola chiave per volta, parallelamente all'esecuzione dei round nel core principale.

L'algoritmo di espansione richiede tre trasformazioni sulla word più significativa della chiave precedente: *RotWord*, *SubWord* e l'aggiunta di una costante di round (*Rcon*). Il design hardware di questo modulo adotta due eleganti ottimizzazioni:

- L'operazione di *RotWord* (rotazione ciclica dei byte) non consuma risorse attive, ma è ottenuta implicitamente sfalsando il cablaggio dei byte in ingresso allo stadio successivo.
- L'operazione di *SubWord* riutilizza l'architettura basata sulle 4 BRAM (`aes_sbox_bram`). Poiché la lettura in Block RAM impone una latenza di un ciclo di clock, la FSM di questo modulo richiede due cicli totali per asserire il segnale `key_ready`.

```
// RotWord ottenuto implicitamente incrociando l'ordine dei byte (es: w3[23:16] in sb0)
aes_sbox_bram sb0 (.clk(clk), .in_byte(w3[23:16]), .out_byte(sub_w3[31:24]));
aes_sbox_bram sb1 (.clk(clk), .in_byte(w3[15:8]),  .out_byte(sub_w3[23:16]));
aes_sbox_bram sb2 (.clk(clk), .in_byte(w3[7:0]),   .out_byte(sub_w3[15:8]));
aes_sbox_bram sb3 (.clk(clk), .in_byte(w3[31:24]), .out_byte(sub_w3[7:0]));

// ... generazione del rcon_val basata sul round_counter ...

// Calcolo combinatorio della nuova chiave (disponibile il ciclo successivo alle BRAM)
wire [31:0] temp    = sub_w3 ^ rcon_val;
wire [31:0] next_w0 = w0 ^ temp;
wire [31:0] next_w1 = w1 ^ next_w0;
wire [31:0] next_w2 = w2 ^ next_w1;
wire [31:0] next_w3 = w3 ^ next_w2;
```

Snippet 5.13: Espansione della chiave: RotWord implicito e logica combinatoria XOR.

aes_mixcolumn

Il modulo `aes_mixcolumn` implementa la trasformazione di diffusione spaziale dell'algoritmo, operando su una singola colonna da 32 bit dello stato. Matematicamente, questa operazione equivale al prodotto di matrici all'interno del campo

finito di Galois $GF(2^8)$, utilizzando il polinomio irriducibile $x^8+x^4+x^3+x+1$ (rappresentato esadecimalmente come `0x11B`).

A livello di sintesi hardware, in accordo con le direttive di ottimizzazione per implementazioni hardware compatte descritte in [15], questo modulo è puramente combinatorio: non contiene alcun elemento sequenziale (flip-flop) né necessita di clock. La moltiplicazione per 2 nel campo di Galois, nota come *xtime*, come illustrato nei paragrafi precedenti, viene risolta tramite un semplice shift logico a sinistra, seguito da uno XOR condizionale con `0x1B` nel caso in cui il bit più significativo (MSB) originario fosse a 1. All'interno dell'`aes_core`, questo modulo viene istanziato 4 volte in parallelo per processare l'intera matrice di stato senza introdurre alcuna latenza sequenziale.

```
1  // Estrazione dei byte della colonna
2  wire [7:0] b0 = col_in[7:0];
3  wire [7:0] b1 = col_in[15:8];
4  wire [7:0] b2 = col_in[23:16];
5  wire [7:0] b3 = col_in[31:24];
6
7  // Operazione xtime (moltiplicazione per 2 nel campo di
       Galois)
8  wire [7:0] m2_b0 = {b0[6:0], 1'b0} ^ (b0[7] ? 8'h1b : 8'h00)
       ;
9  wire [7:0] m2_b1 = {b1[6:0], 1'b0} ^ (b1[7] ? 8'h1b : 8'h00)
       ;
10 wire [7:0] m2_b2 = {b2[6:0], 1'b0} ^ (b2[7] ? 8'h1b : 8'h00)
       ;
11 wire [7:0] m2_b3 = {b3[6:0], 1'b0} ^ (b3[7] ? 8'h1b : 8'h00)
       ;
12
13 // Sintesi delle 4 equazioni della matrice MixColumns
14 assign col_out[31:24] = m2_b3 ^ (m2_b2 ^ b2) ^ b1 ^ b0;
15 assign col_out[23:16] = b3 ^ m2_b2 ^ (m2_b1 ^ b1) ^ b0;
16 assign col_out[15:8]  = b3 ^ b2 ^ m2_b1 ^ (m2_b0 ^ b0);
17 assign col_out[7:0]   = (m2_b3 ^ b3) ^ b2 ^ b1 ^ m2_b0;
```

Snippet 5.14: Implementazione puramente combinatoria della moltiplicazione in GF(2^8).

aes_sbox_bram

Questo sottomodulo costituisce l'elemento non lineare dell'architettura AES, ovvero la *Substitution Box* (S-Box). Piuttosto che implementare la trasformazione algebrica inversa in hardware (che risulterebbe estremamente complessa e impegnativa in termini logici) o sintetizzare una Look-Up Table (LUT) consumando

eccessive celle logiche distribuite dell'FPGA, si è optato per una mappatura su memoria dedicata.

La ROM da 256 entry per 8 bit viene allocata esplicitamente nei blocchi di memoria nativi (BSRAM) del chip Gowin. Per istruire il sintetizzatore in tal senso, viene impiegata la direttiva di ottimizzazione `(* ram_style = "block" *)`. Poiché l'architettura fisica delle BSRAM richiede che gli accessi in lettura siano registrati, il modulo introduce un ciclo di *latenza*: il dato richiesto in `in_byte` sarà disponibile su `out_byte` solo al fronte di salita del clock successivo, costringendo le FSM dell'`aes_core` e del *Key Schedule* a implementare stati di attesa dedicati.

```
(* ram_style = "block" *) // Forza l'uso delle memorie integrate
reg [7:0] rom [0:255];

initial begin
    // Mappatura statica dei 256 valori della S-Box AES
    rom[8'h00]=8'h63; rom[8'h01]=8'h7c; rom[8'h02]=8'h77; rom[8'h03]=8'h7b;
    // ...
    rom[8'hfc]=8'hb0; rom[8'hfd]=8'h54; rom[8'hfe]=8'hbb; rom[8'hff]=8'h16;
end

// La lettura sincrona causa 1 ciclo di latenza e garantisce la sintesi su BSRAM
always @(posedge clk) begin
    out_byte <= rom[in_byte];
end
```

Snippet 5.15: Inizializzazione della S-Box e inferenza della Block RAM (BSRAM).

5.3 Firmware

In questa sezione viene descritto il livello software del progetto, ovvero il firmware **bare-metal** scritto in linguaggio C e Assembly che viene eseguito sul processore PicoRV32. Il software ha il compito di inizializzare il sistema, configurare l'ambiente di runtime, fornire le interfacce per la comunicazione seriale e, soprattutto, implementare il driver per comunicare con il co-processore crittografico hardware tramite il bus PCPI, verificandone il corretto funzionamento tramite una suite di test.

5.3.1 Inizializzazione e librerie di base

Eseguire codice C in un ambiente *bare-metal* (ovvero senza un sistema operativo sottostante) richiede una fase di preparazione manuale dell'ambiente di esecuzione. Questa fase comprende la corretta mappatura della memoria, l'inizializzazione dei registri fondamentali (come lo stack pointer) e l'implementazione delle chiamate di sistema (system call) necessarie alla libreria standard.

Linker Script (link.ld)

Il linker script definisce la mappa della memoria fisica disponibile e indica al compilatore dove posizionare le varie sezioni del programma. Nel nostro sistema, tutta la memoria a disposizione è costituita dai 32 KB di BRAM istanziati su FPGA. Il vincolo più critico definito in questo file è che la sezione dell'*entry point* (`.text.start`) debba trovarsi esattamente all'indirizzo `0x00000000`. Questo perché il processore PicoRV32, dopo un reset, inizia a fare fetch delle istruzioni a partire da quell'indirizzo. Inoltre, il linker script definisce la posizione dello stack alla fine della BRAM (crescente verso il basso) e i limiti per l'heap allocabile dinamicamente.

```
ENTRY(_start)

MEMORY {
    RAM (rwx) : ORIGIN = 0x00000000, LENGTH = 32K
}

SECTIONS {
    .text : {
        *(.text.start)   /* entry point per primo a 0x0 */
        *(.text*)
        *(.rodata*)
    } > RAM
    /* ... sezioni .data e .bss ... */
    _end = .;
```

```
    __stack    = 0x00007FFC;   /* top of stack */
    __heap_end = 0x00006000;   /* limite heap   */
}
```

Snippet 5.16: Mappatura della memoria nel linker script.

Entry point Assembly (start.S)

Prima che la funzione `main()` in C possa essere eseguita, è necessario uno stub in linguaggio Assembly che configuri l'ambiente. Il file `start.S` assolve a tre compiti fondamentali:

- Inizializza lo *Stack Pointer* (`sp`) caricando l'indirizzo `__stack` definito nel linker script.
- Azzera la sezione `.bss`, che contiene le variabili globali non inizializzate, garantendo che partano da un valore nullo come richiesto dallo standard C.
- Salta alla funzione `main()` e, al suo ritorno, gestisce l'uscita scrivendo il codice di terminazione all'indirizzo `0x20000000`, il quale hardware-side spegnerà un LED per segnalare la fine dell'esecuzione (o terminerà la simulazione nel testbench).

```
.section .text.start
.global _start

_start:
    la   sp, __stack          # imposta stack pointer al top
    della BRAM

    # azzera .bss
    la   a0, __bss_start
    la   a1, __bss_end
bss_loop:
    bge  a0, a1, bss_done
    sw   zero, 0(a0)
    addi a0, a0, 4
    j    bss_loop
bss_done:
    call main
    # ... gestione terminazione ...
```

Snippet 5.17: Inizializzazione dello stack e azzeramento BSS in start.S.

System Calls per picolibc (syscalls.c)

Per poter utilizzare funzioni di utilità standard come la `printf`, la libreria C (in questo caso *picolibc*, ottimizzata per sistemi embedded) richiede la definizione di alcune chiamate di sistema hardware-dipendenti. Il collegamento più importante è quello della funzione `uart_putc`, che dirotta l'output standard verso l'indirizzo di memoria mappato alla periferica UART hardware (`0x10000000`). In questo modo, ogni chiamata a `printf` nel codice C si traduce nella scrittura di byte sul bus, che l'hardware UART provvederà poi a serializzare e trasmettere verso il PC.

```
#define UART_TX_ADDR ((volatile unsigned int *)0x10000000)

static int uart_putc(char c, FILE *f) {
    (void)f;
    *UART_TX_ADDR = (unsigned int)c;   // Scrittura su bus per uart_tx.v
    return (unsigned char)c;
}

static FILE __stdio = FDEV_SETUP_STREAM(uart_putc, NULL, NULL, _FDEV_SETUP_WRITE);
FILE *const stdout = &__stdio;
FILE *const stderr = &__stdio;
```

Snippet 5.18: Collegamento della printf alla periferica UART.

In questo file sono definite anche la funzione `_sbrk` per la gestione dinamica della memoria (heap) allocabile con `malloc`, e la funzione `_exit` per il blocco del sistema a fine esecuzione.

5.3.2 Applicativo principale e Driver AES (main.c)

Il file `main.c` rappresenta il cuore applicativo del firmware. Il suo scopo è astrarre la comunicazione con l'acceleratore hardware AES, fornendo un'interfaccia ad alto livello utilizzabile in C, e testarne l'efficacia tramite la cifratura di blocchi noti.

Driver hardware, pack/unpack e funzione wrapper

L'interazione con l'acceleratore PCPI non avviene tramite registri mappati in memoria (Memory-Mapped I/O), ma tramite un'istruzione custom del set RISC-V. Dato che il compilatore GCC standard non conosce questa istruzione, è necessario ricorrere all'uso di codice assembly *inline* (direttiva `.insn r`) all'interno del C. La funzione `hw_aes_inst` agisce come primitiva di base per iniettare l'istruzione avente **opcode** `0x0B`. Riceve come parametri i valori da posizionare nei registri

sorgente `rs1` (che codifica il comando e l'indice) e `rs2` (che porta il payload dei dati), restituendo il valore fornito dal registro destinazione `rd`.

Dato che l'architettura AES richiede blocchi da 128 bit (16 byte), ma i registri del processore sono a 32 bit, sono state implementate le funzioni `pack_word` e `unpack_word` per accorpare e scomporre correttamente array di 4 byte in singole word (e viceversa), rispettando la corretta *endianness*. La scelta di eseguire lo swap via *software* è stata fatta per mantenere l'acceleratore crittografico completamente standard e indifferente rispetto all'architettura del processore. Poiché l'algoritmo AES opera nativamente in *big-endian*, delegare questo adattamento al livello software garantisce un'elevata *portabilità*: in caso di futura migrazione verso una CPU con endianness differente, sarà sufficiente abilitare o disabilitare la logica di swap via codice, senza dover apportare alcuna modifica al design dell'hardware dedicato.

```
static inline uint32_t hw_aes_inst(uint32_t rs1_val,
    uint32_t rs2_val) {
    uint32_t rd_val;
    __asm__ volatile (
        ".insn r 0x0B, 0, 0, %0, %1, %2"
        : "=r" (rd_val)
        : "r" (rs1_val), "r" (rs2_val)
    );
    return rd_val;
}

static inline uint32_t pack_word(const uint8_t *b) {
    return ((uint32_t)b[0] << 24) | ((uint32_t)b[1] << 16) |
           ((uint32_t)b[2] <<  8) |  (uint32_t)b[3];
}

static inline void unpack_word(uint32_t w, uint8_t *b) {
    b[0] = (w >> 24) & 0xFF;
    b[1] = (w >> 16) & 0xFF;
    b[2] = (w >>  8) & 0xFF;
    b[3] =  w        & 0xFF;
}
```

Snippet 5.19: Istruzione PCPI inline assembly e funzioni di data packing/unpacking.

Costruita sopra queste primitive, troviamo la funzione wrapper ad alto livello `cifra_blocco_aes`. Questa nasconde completamente i dettagli dell'handshake hardware, ricevendo due *array* da 16 byte (chiave e plaintext) e popolando un *array* di output. La funzione itera 4 volte per caricare la chiave e 4 volte per il testo (usando rispettivamente `CMD_WRITE_KEY` e `CMD_WRITE_TEXT`). Successiva-

mente invia il comando di avvio (`CMD_START`): in questa fase il segnale hardware `pcpi_wait` congela automaticamente la CPU, evitato a chi la utilizza di dover implementare cicli di polling o interrupt. Al risveglio della CPU, il calcolo è completato e la funzione itera altre 4 volte con `CMD_READ_TEXT` per estrarre il ciphertext.

```
void cifra_blocco_aes(const uint8_t key[16], const uint8_t
    in[16], uint8_t out[16]) {
     // Carica la chiave (4 word da 32 bit)
     for (int i = 0; i < 4; i++)
         hw_aes_inst((i << 2) | CMD_WRITE_KEY, pack_word(&key
    [i * 4]));

     // Carica il plaintext in ingresso
     for (int i = 0; i < 4; i++)
         hw_aes_inst((i << 2) | CMD_WRITE_TEXT, pack_word(&in
    [i * 4]));

     // Avvia il co-processore - La CPU va in stallo hardware
     fino al completamento
     hw_aes_inst(CMD_START, 0);

     // Estrae il ciphertext calcolato
     for (int i = 0; i < 4; i++)
         unpack_word(hw_aes_inst((i << 2) | CMD_READ_TEXT, 0)
    , &out[i * 4]);
}
```

Snippet 5.20: Funzione wrapper per la cifratura completa di un blocco AES

Suite di test e utilities

Per validare matematicamente l'acceleratore hardware sviluppato, la funzione `main` definisce una suite di test strutturata. Il codice impiega funzioni di utilità ausiliarie, come `print_hex` per formattare la stampa esadecimale su seriale e `compare` per verificare che gli array generati combacino con le aspettative a livello di singolo byte ed il risultato è visibile in Figura 5.8.

La suite esegue tre prove mirate per stressare la FSM dell'acceleratore e l'interfaccia bus:

1. **Vettore Ufficiale NIST:** Viene testato il *Test Vector* contenuto nell'Appendice B della specifica ufficiale FIPS-197. Questo test garantisce che l'algoritmo sia stato implementato fedelmente secondo lo standard crittografico.

2. **Cifratura nulla:** Vengono inviati una chiave e un testo formati esclusivamente da zeri (0x00). Serve ad accertare che non ci siano errori sistematici di inizializzazione nei registri del core AES.

3. **Cifratura sequenziale:** Lo stesso blocco viene cifrato per due volte consecutive. Questo test è fondamentale per assicurare che lo stato interno hardware (il registro di stato, i contatori di round, l'estensione chiave) ritorni perfettamente in stato di IDLE e venga azzerato correttamente tra un'operazione e l'altra, senza trascinare dati "sporcizia" dalle computazioni passate.

```
int main(void) {
    printf("=== Test Acceleratore AES Hardware (PCPI) ===\r\n\n");

    /* Test 1: Vettore ufficiale NIST FIPS-197 Appendice B */
    const uint8_t key1[16] = {
        0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6,
        0xAB, 0xF7, 0x15, 0x88, 0x09, 0CF, 0x4F, 0x3C
    };
    const uint8_t plain1[16] = {
        0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30, 0x8D,
        0x31, 0x31, 0x98, 0xA2, 0xE0, 0x37, 0x07, 0x34
    };
    const uint8_t expected1[16] = {
        0x39, 0x25, 0x84, 0x1D, 0x02, 0xDC, 0x09, 0xFB,
        0xDC, 0x11, 0x85, 0x97, 0x19, 0x6A, 0x0B, 0x32
    };
    uint8_t result1[16];

    printf("-- Test 1: vettore NIST FIPS-197 Appendice B --\r\n");
    cifra_blocco_aes(key1, plain1, result1);
    print_hex("Output  ", result1, 16);
    printf("Risultato: %s\r\n\n", compare(result1, expected1, 16) ? "OK v" : "ERRORE x");

    // ... Esecuzione dei successivi Test 2 e Test 3 ...
    return 0;
}
```

Snippet 5.21: Estratto del main.c con l'esecuzione del primo Test NIST.

```
port is        : /dev/ttyUSB1
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : no
hangup is      : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        :
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none
initstring     : none
exit_after is  : not set
exit is        : no

Type [C-a] [C-h] to see available commands
Terminal ready
=== Test Acceleratore AES Hardware (PCPI) ===

-- Test 1: vettore NIST FIPS-197 Appendice B --
Chiave  : 2B7E151628AED2A6ABF7158809CF4F3C
Input   : 3243F6A8885A308D313198A2E0370734
Output  : 3925841D02DC09FBDC118597196A0B32
Atteso  : 3925841D02DC09FBDC118597196A0B32
Risultato: OK v

-- Test 2: chiave e testo tutti zero --
Chiave  : 00000000000000000000000000000000
Input   : 00000000000000000000000000000000
Output  : 66E94BD4EF8A2C3B884CFA59CA342B2E
Atteso  : 66E94BD4EF8A2C3B884CFA59CA342B2E
Risultato: OK v

-- Test 3: due cifrature consecutive (stesso blocco) --
Prima   : 3925841D02DC09FBDC118597196A0B32
Seconda : 3925841D02DC09FBDC118597196A0B32
Risultato: OK v (identici)

=== Fine test ===
```

Figura 5.8: Output del terminale per suite di test.

Capitolo 6

Risultati - MODBUS in real-time

In questo capitolo vengono descritti i risultati tramite test empirici eseguiti per verificare il rispetto dei vincoli di correttezza, prestazioni, real-time ed efficienza energetica imposti dal progetto. Vengono inoltre presentati i confronti effettuati con altre architetture hardware permettendo successive considerazioni analitiche sui risultati ottenuti.

6.1 Setup sperimentale e misure effettuate

Per valutare le prestazioni crittografiche e i consumi energetici, le misurazioni sono state condotte direttamente sull'hardware tramite test empirici. In questa sezione vengono dettagliati i vincoli temporali presi in considerazione, viene descritto il *setup* dei test e le metodologie utilizzate per l'acquisizione dei dati.

6.1.1 Il meccanismo di framing del Modbus RTU e il vincolo $t_{1.5}$

A differenza dei protocolli di rete moderni che utilizzano header specifici o caratteri di start/stop (es. *Ethernet* o delimitatori ASCII) per definire i confini di un pacchetto, il protocollo seriale Modbus RTU gestisce il *framing* dei dati basandosi esclusivamente sulle tempistiche di inattività del bus, ovvero sul silenzio della linea seriale [31].

Lo standard ufficiale definisce due vincoli temporali critici che ogni nodo della rete deve rigorosamente rispettare:

- **Pausa $t_{1.5}$:** Il tempo massimo consentito tra la ricezione di due *byte* consecutivi appartenenti allo stesso frame. Se si verifica un silenzio superiore a questo limite, il frame viene considerato frammentato o corrotto e deve essere scartato.

- **Pausa $t_{3.5}$:** Il tempo di silenzio che decreta la fine di un messaggio. Se la linea rimane inattiva per questo lasso di tempo, il nodo ricevitore considera il *frame* chiuso e lo passa al livello applicativo per l'elaborazione (controllo CRC e parsing).

La nomenclatura di questi vincoli deriva direttamente dalla fisica della trasmissione seriale: i pedici 1.5 e 3.5 esprimono letteralmente il tempo richiesto per trasmettere sul mezzo fisico $1{,}5$ e $3{,}5$ caratteri (comunemente identificati come byte comprensivi di bit di overhead). Nella configurazione standard a 10 bit per carattere (composta da 1 bit di start, 8 bit di dati, nessuna parità e 1 bit di stop), il limite inter-carattere $t_{1.5}$ equivale quindi esattamente al tempo di trasmissione di 15 bit sul cavo elettrico. Questa soglia è stata storicamente definita come il compromesso ottimale per discriminare in modo affidabile una reale frammentazione del pacchetto da una semplice e temporanea latenza di trasmissione tra i byte.

Poiché l'introduzione di un layer crittografico *inline* obbliga il microcontrollore o l'FPGA a cifrare i blocchi di dati sequenzialmente, se il tempo di calcolo tra un blocco e l'altro supera la soglia critica del $t_{1.5}$, si genera un silenzio sulla linea. Il nodo ricevitore interpreterà questo silenzio come una frammentazione fatale del pacchetto, scartando il frame e causando il collasso della comunicazione sulla rete industriale.

Definizione della soglia critica a 115200 bps

Per valutare la conformità dell'architettura proposta, è necessario definire il valore temporale assoluto del limite $t_{1.5}$. E' stato quindi considerato uno scenario applicativo di riferimento operante a una velocità di trasmissione (*Baud Rate*) di 115200 bps, tipica delle moderne implementazioni Modbus RTU ad alte prestazioni su bus RS485. Secondo le specifiche ufficiali (*Modbus Serial Line Protocol and Implementation Guide V1.02* [31]), per baud rate fino a 19200 bps i tempi di latenza si calcolano analiticamente in base al tempo di trasmissione del singolo carattere. Invece, per velocità superiori, come in questo caso, l'elevata frequenza di bit rende i margini di campionamento troppo stringenti per i ricevitori. Pertanto, lo standard impone l'utilizzo di valori temporali fissi per garantire la robustezza della comunicazione:

$$t_{1.5} = 750\,\mu\text{s}$$

$$t_{3.5} = 1{,}750\ \text{ms}\ (1750\,\mu\text{s})$$

Considerando che l'architettura *System-on-Chip* sviluppata sulla Tang Nano 9K opera con un clock di sistema $f_{\text{clk}} = 27$ MHz, il periodo di clock del processore è pari a $T_{\text{clk}} \approx 37{,}037$ ns. È quindi possibile convertire i cicli di clock misurati nel precedente benchmark in latenze temporali assolute, per confrontarle con il vincolo fisso inter-carattere di $750\,\mu$s.

6.1.2 Piattaforme hardware e librerie crittografiche

I test sono stati eseguiti su tre diverse piattaforme hardware in Figura 6.1:

1. **Tang Nano 9K (Gowin GW1NR-9 a 27 MHz):** Scheda di riferimento del progetto, utilizzata per valutare sia l'esecuzione software sul *soft-core* PicoRV32, sia l'accelerazione del coprocessore hardware.
2. **STMicroelectronics Nucleo-H755ZI-Q (Cortex-M7 a 480 MHz):** Possiede un microcontrollore industriale high-end basato su architettura dual-core composta da due processori ARM: Cortex-M7 e Cortex-M4 anche se quest'ultimo è stato disattivato in questi test.
3. **Espressif ESP32 DevBoard (Xtensa a 240 MHz):** Scheda orientata all' IoT ad alta che integra un SoC completo. Questo dispone inoltre di un acceleratore AES integrato che viene sfuttato tramite librerie del suo framework.

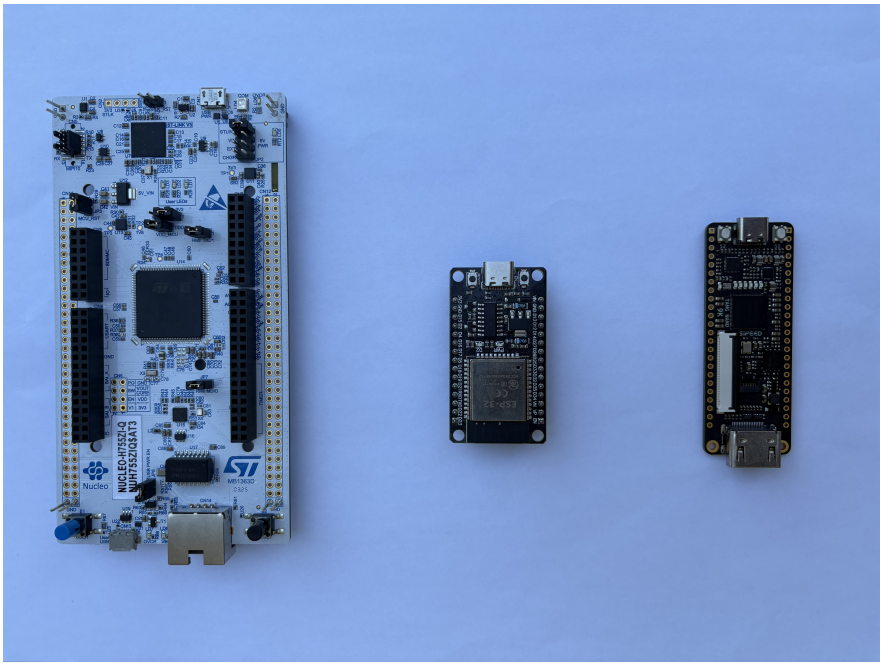

Figura 6.1: Board utilizzate per i test: Nucleo-H755ZI-Q, Esp32 DevBoard, Tang Nano 9K.

Per garantire un confronto coerente sull'efficienza di calcolo tra CPU profondamente diverse, sul processore PicoRV32 e sull'STM32H7 è stata impiegata la stessa libreria crittografica software: `tiny-AES-c`. Si tratta di un'implementazione *open-source* in C standard, ottimizzata per sistemi *embedded* e priva di dipendenze esterne. Sull'ESP32, diversamente, si è utilizzata l'implementazione AES fornita nativamente dal framework di sistema tramite la libreria `mbedtls`.

6.1.3 Tecniche di misurazione

Sui microcontrollori PicoRV32 e STM32H7, per evitare imprecisioni o latenze introdotte da timer periferici, il tempo di esecuzione è stato misurato contando direttamente i cicli di clock della CPU.

Nel SoC basato su PicoRV32, la misurazione è avvenuta leggendo i registri di controllo interni (CSR) del core RISC-V. Tramite una funzione in *inline assembly*, è stato interrogato il registro `cycle`, che si incrementa a ogni fronte di salita del clock di sistema:

```
// Funzione per leggere il contatore di cicli interno del
   RISC-V
static inline uint32_t read_cycles(void) {
    uint32_t cycles;
    __asm__ volatile ("rdcycle %0" : "=r" (cycles));
    return cycles;
}
```

Snippet 6.1: Estrazione dei cicli di clock su architettura RISC-V (PicoRV32).

Sull'architettura ARM Cortex-M7 (STM32H7) è stata adottata la stessa tecnica. Sbloccando e attivando il registro hardware `DWT_CYCCNT` (*Data Watchpoint and Trace Cycle Count Register*), interno al core, è stato possibile ottenere il conteggio esatto dei cicli:

```
// Sblocco e avvio del DWT per la misurazione dei cicli
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
DWT->LAR = 0xC5ACCE55;
DWT->CYCCNT = 0;
DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;

uint32_t start_cycles = DWT->CYCCNT;
// ... Esecuzione cifratura AES ...
uint32_t stop_cycles = DWT->CYCCNT;

uint32_t total_cycles = stop_cycles - start_cycles;
```

Snippet 6.2: Configurazione e lettura del DWT Cycle Counter su STM32H7.

Sull'ESP32, il framework operativo astrae l'accesso di basso livello ai contatori di ciclo. Di conseguenza, le misurazioni sono state effettuate rilevando il tempo assoluto tramite la funzione nativa `micros()`, che restituisce i microsecondi trascorsi dall'avvio della cifratura, fornendo una risoluzione idonea per il test:

```
// ----- INIZIO MISURA -----
uint32_t t_start = micros();
```

```
3
4 // Cifratura dei due blocchi da 16 byte
5 mbedtls_aes_crypt_ecb(&aes, MBEDTLS_AES_ENCRYPT, input,
    output);
6 mbedtls_aes_crypt_ecb(&aes, MBEDTLS_AES_ENCRYPT, input + 16,
     output + 16);
7
8 uint32_t t_end = micros();
9 // ----- FINE MISURA -----
```

Snippet 6.3: Misurazione del tempo assoluto in esecuzione su ESP32.

6.1.4 Raccolta dati e plotting

I dati ottenuti da queste routine di test sono stati successivamente trasmessi via interfaccia UART, lette tramite terminale seriale utilizzando il software *picocom*, e registrati per le elaborazioni analitiche sui tempi di latenza e sui consumi energetici presentate nei paragrafi successivi. Per la realizzazione dei grafici e approssimazioni matematiche sono state utilizzate le librerie *Python* standard quali *Matplotlib* e *Numpy*.

6.2 Efficienza del Payload: mismatch architetturale e impatto del padding

6.2.1 Vincolo crittografico e la logica di padding

Come anticipato nei capitoli precedenti, la scelta di adottare lo standard AES-128 è dettata dalla necessità di garantire al sistema un livello di sicurezza elevato e consolidato. Tuttavia, l'algoritmo AES elabora i dati esclusivamente in matrici fisse da 128 bit (16 byte), non prevedendo varianti ufficiali per blocchi più piccoli. Questo introduce un disallineamento geometrico con il protocollo Modbus RTU, i cui pacchetti (*Protocol Data Unit*) hanno dimensioni variabili, ottimizzate per occupare la minore banda possibile.

Per permettere la cifratura, è quindi obbligatorio applicare una tecnica di riempimento (*Zero Padding in-place*), accodando byte nulli (0×00) al frame Modbus fino a raggiungere il multiplo di 16 byte più vicino. È fondamentale sottolineare che questa operazione rappresenta un vincolo strutturale dell'algoritmo: sia un'elaborazione software che una coprocessazione hardware devono obbligatoriamente cifrare anche i byte di riempimento, introducendo un *overhead* sui dati.

6.2.2 Definizione degli scenari di benchmark

Per valutare l'impatto delle prestazioni di questo adattamento, il test è stato strutturato su tre scenari applicativi con livelli di inefficienza differenti:

- **Scenario A (Caso ottimale - 13 byte):** Un comando di scrittura multipla (*Function Code 16*) per 2 registri. Il frame viene completato a 16 byte (1 blocco AES) con un overhead minimo di soli 3 byte.
- **Scenario B (Caso corto - 8 byte):** Un tipico comando di lettura (*Function Code 03*). Il frame viene esteso a 16 byte (1 blocco AES), sprecando il 50% del blocco in byte fittizi.
- **Scenario C (Salto di blocco - 17 byte):** Un comando di scrittura (*Function Code 16*) per 4 registri. Superando la soglia dei 16 byte, il frame deve essere esteso a 32 byte (2 blocchi AES), richiedendo la cifratura di ben 15 byte nulli.

I tre scenari sono stati elaborati misurando i cicli di clock visibili in 6.2 sia tramite la libreria software sulla CPU PicoRV32, sia tramite l'acceleratore hardware.

```
omap is          :
emap is          : crcrlf,delbs,
logfile is       : none
initstring       : none
exit_after is    : not set
exit is          : no

Type [C-a] [C-h] to see available commands
Terminal ready

=== BENCHMARK PADDING MODBUS: HW vs SW ===
Scenario             | Len Padded | Blocchi AES | Cicli HW |
-------------------------------------------------------------
Scenario A (13 byte) | 16 byte    | 1 blocchi   |     1261 |   33313
Scenario B (8 byte)  | 16 byte    | 1 blocchi   |     1261 |   33313
Scenario C (17 byte) | 32 byte    | 2 blocchi   |     2506 |   66610
=============================================================
```

Figura 6.2: Tabella risultati empirici.

6.2.3 Analisi prestazionale: costo computazionale per byte utile

Per quantificare in modo rigoroso l'impatto dell'overhead introdotto dal padding, i dati estratti dal banco di prova non sono stati valutati unicamente in termini di tempo assoluto, ma normalizzati attraverso il costo computazionale per singolo

byte utile. Questo valore si ottiene dividendo i cicli totali di elaborazione per i soli byte informativi del pacchetto modbus originale. Questa metrica evidenzia l' *overhead* variabile imposto dal vincolo geometrico dell'algoritmo AES.

A partire dai tre scenari testati empiricamente sulla scheda (che hanno fornito i tempi esatti di elaborazione per il singolo e il doppio blocco), è stato possibile ricavare l'andamento dell'efficienza per tutte le possibili lunghezze del *payload* Modbus (da 1 a 32 byte). Poiché il tempo di esecuzione dell'algoritmo crittografico è strettamente deterministico e dipende unicamente dal numero di blocchi da 16 byte generati dal padding, l'elaborazione matematica restituisce una mappatura esatta del comportamento del sistema.

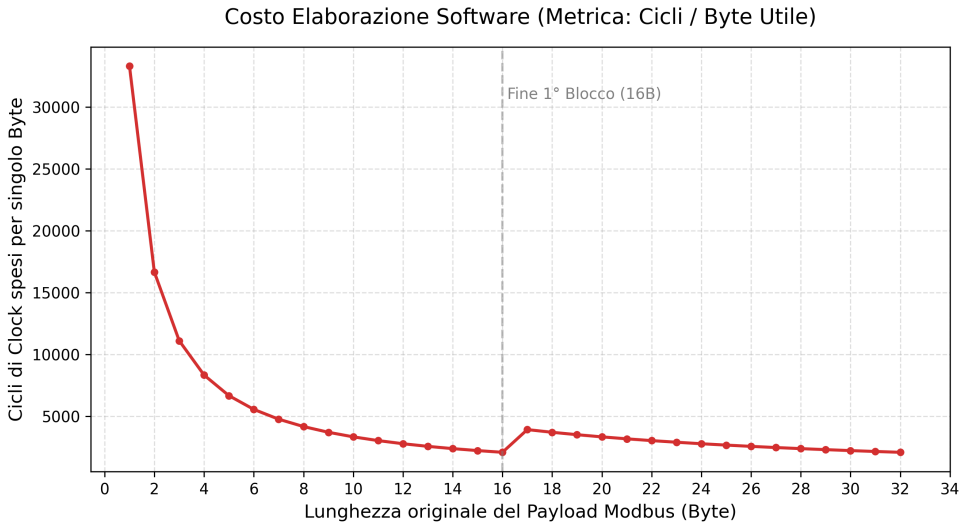

Figura 6.3: Costo computazionale Software in cicli per byte utile.

Analizzando i risultati dell'implementazione puramente software (Figura 6.3), emerge palesemente l'inefficienza causata dal mismatch dei blocchi. L'andamento decrescente conferma che l'efficienza relativa migliora man mano che il blocco viene riempito di dati utili. Tuttavia, nello Scenario B (8 byte), dove l'overhead raggiunge il 50%, il processore è costretto a spendere 4164 cicli di clock per ogni singolo byte di vera informazione. Questo accade perché la CPU esegue in modo strettamente sequenziale complesse trasformazioni in $GF(2^8)$ su zeri inutili.

Il limite critico si osserva al superamento della soglia dei 16 byte (linea tratteggiata): nello Scenario C (17 byte), l'aggiunta di un singolo byte utile costringe il sistema a calcolare un intero secondo blocco AES vuoto. Il picco di inefficienza raddoppia il tempo totale di elaborazione a $66\,610$ cicli (pari a circa $2,46$ ms). Questo ritardo assoluto viola palesemente il limite temporale inter-carattere $t_{1.5}$ del Modbus (fissato a 750 µs), spezzando il flusso della comunicazione seriale e

rendendo l'approccio software inutilizzabile su bus di campo a 115200 bps. I vincoli e le problematiche legate ai limiti temporali del protocollo verranno esaminati con maggiore attenzione nella prossima sottosezione.

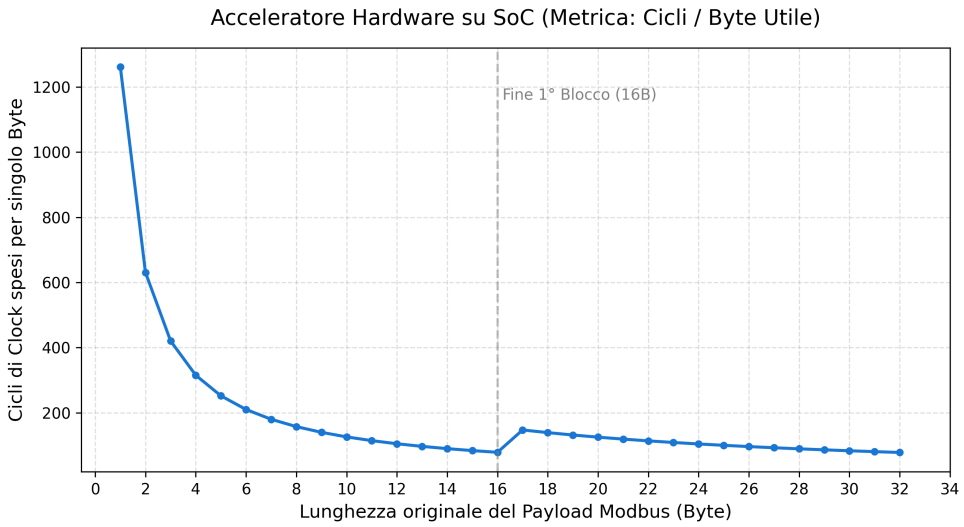

Figura 6.4: Costo computazionale Hardware in cicli per byte utile.

Osservando i dati dell'acceleratore hardware (Figura 6.4), la curva assume lo stesso identico andamento fisiologico, dato che il coprocessore è comunque vincolato a cifrare l'intero blocco paddato. I valori assoluti sono però traslati su un ordine di grandezza completamente diverso.

Per giustificare i dati è comunque necessario fare una precisazione, dato che il core AES hardware esegue di per sé i 10 round dell'algoritmo in soli 103 cicli. Il valore totale registrato empiricamente di 1261 cicli per un singolo blocco comprende l'esecuzione crittografica unita al necessario *overhead* del firmware. Questo collo di bottiglia è la diretta conseguenza della scelta progettuale di lasciare l'adattamento dei dati al software per mantenere standard il design dell'acceleratore. Questo sacrificio prestazionale è reso sostenibile proprio dall'estrema velocità con cui avviene la cifratura vera e propria in hardware, che offre un ampio margine per assorbire le latenze introdotte dal codice. Di conseguenza, il processore PicoRV32, il cui obiettivo primario è il risparmio di area logica e non la massimizzazione delle performance, si trova a spendere oltre mille cicli di clock per gestire tutto ciò che riguarda la manipolazione e passaggio dei dati. Per esempio deve processare tutte le istruzioni Assembly di *packing* a 32 bit che impiegano esattamente 64 cicli di clock per ogni chiamata di *pack* e 62 per *unpack* non avendo il *barrel -shift* abilitato oltre che i cicli iterativi in C e l'handshake sul bus PCPI.

Anche includendo la totalità di questo overhead di interfacciamento del SoC, l'architettura dedicata riduce il costo per byte utile a soli 157 cicli nel caso pessimo dello Scenario B. Il coprocessore su FPGA si dimostra in ultima analisi oltre 26 volte più efficiente della controparte puramente software. Questa estrema velocità operativa permette al sistema di assorbire integralmente l'impatto temporale dello spreco causato dal *padding*, annullandone gli effetti collaterali e garantendo il pieno rispetto del sincronismo *real-time* della rete industriale.

6.3 Analisi dei vincoli Real-Time: conformità al limite Modbus $t_{1.5}$

Come evidenziato nella sezione precedente, il protocollo Modbus RTU impone rigorosi vincoli di silenzio sul bus: $t_{1.5}$ e $t_{3.5}$. Nella progettazione del *layer* crittografico *inline*, la vera soglia critica invalicabile è rappresentata dal limite inter-carattere $t_{1.5}$ (750 μs a 115200 bps).

Mentre il rispetto del $t_{3.5}$ (1750 μs) definisce la fine del messaggio e incide prevalentemente sul *throughput* di rete, la violazione del $t_{1.5}$ ha effetti distruttivi. Se la latenza di calcolo dell'algoritmo AES-128 genera un'inattività sul bus superiore a questa soglia, il ricevitore interpreta il prolungato silenzio come un'anomala frammentazione del pacchetto. Il *frame* viene quindi scartato ancor prima della decifratura o della validazione CRC, causando il collasso immediato della comunicazione industriale e quindi questo definisce il nostro vincolo di *real-time*.

6.3.1 Valutazione della latenza crittografica: Software vs Hardware

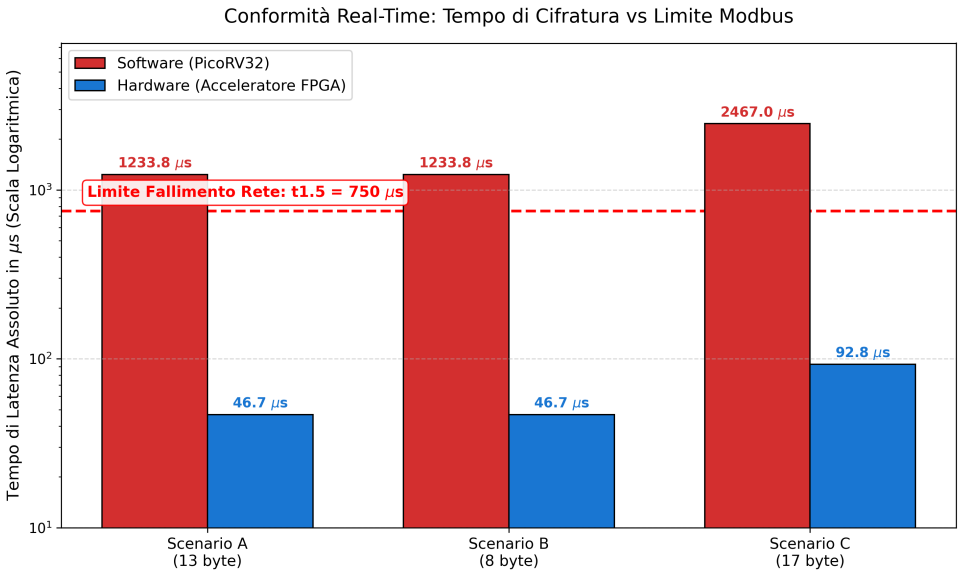

Figura 6.5: Latenza assoluta di cifratura a confronto con il vincolo inter-carattere del Modbus RTU ($t_{1.5}$ fisso a $750\,\mu$s).

La conversione dei cicli di elaborazione nei corrispondenti valori temporali (Figura 6.5) evidenzia una criticità insormontabile per l'approccio puramente software: l'incapacità strutturale di rispettare le stringenti tempistiche imposte dal bus seriale.

Negli Scenari A e B (singolo blocco AES), il processore PicoRV32 conclude l'elaborazione in circa $1233{,}8\ \mu$s. Valutando questo dato rispetto al vincolo inter-carattere $t_{1.5}$, emerge palesemente il fallimento dell'approccio software: questo tempo di esecuzione sfonda già ampiamente la soglia critica di $750\ \mu$s. L'operazione crittografica da sola consuma un tempo quasi doppio rispetto al budget temporale massimo consentito per il silenzio sul bus, senza nemmeno includere l'esecuzione delle restanti routine applicative (parsing del protocollo, interrogazione dei sensori fisici e calcolo del CRC). Di conseguenza, la comunicazione seriale viene inevitabilmente spezzata e il nodo di rete entra in una condizione di blocco.

Se la situazione è critica per il blocco singolo, il collasso del protocollo diventa catastrofico non appena il *payload* supera questa soglia. Nello Scenario peggiore testato (Scenario C - 17 byte paddati a 32 byte), la libreria software richiede $66\,610$ cicli di clock, corrispondenti a un tempo di esecuzione di circa $2467\ \mu$s. Questo valore è oltre il triplo del limite consentito. Di conseguenza, un nodo

basato esclusivamente su software risulterebbe del tutto inutilizzabile a 115200 bps, generando timeout irreversibili e costanti troncamenti del frame per qualsiasi tipologia di richiesta.

Al contrario, la cooperazione tra CPU e acceleratore dedicato garantisce tempistiche operative estremamente deterministiche e conservative. Nello scenario più gravoso (Scenario C), l'architettura dedicata conclude le operazioni in soli 2506 cicli, pari a una latenza assoluta di appena 92,8 μs. Il sistema hardware opera mantenendo un notevole margine di sicurezza anche nel caso peggiore di circa l'87,6% rispetto alla "linea rossa" della frammentazione inter-carattere. Questo dimostra che l'FPGA è in grado di inserire uno strato crittografico AES-128 rendendolo totalmente trasparente e preservando l'integrità *real-time* del bus di campo, indipendentemente dalla lunghezza del pacchetto richiesto dal Master.

6.4 Valutazione energetica per l'edge computing: metrica nJ/bit

6.4.1 Efficienza energetica nei nodi industriali

Nel contesto dell'*Industrial Internet of Things* (IIoT), l'efficienza computazionale deve essere valutata considerando l'autonomia energetica e i vincoli di dissipazione termica dei dispositivi. La metrica di riferimento per confrontare le prestazioni crittografiche su architetture eterogenee è il *nanoJoule per bit* (nJ/bit), che esprime il costo energetico netto speso dal sistema per cifrare ogni singolo bit di informazione. Il valore si ottiene integrando la potenza attiva del chip nel tempo di esecuzione e dividendo il risultato per la dimensione del *payload* (fissato a 256 bit, corrispondenti allo Scenario C).

Come analizzato nella Sezione 6.3, l'implementazione puramente software su un *soft-core* a basso consumo non riesce a soddisfare i vincoli temporali del protocollo. L'approccio industriale più comune per superare questo limite consiste nell'impiegare microcontrollori commerciali ad altissima frequenza. Sebbene questa scelta architetturale garantisca il rispetto del limite *real-time* $t_{1.5}$, essa introduce un notevole svantaggio in termini di consumo attivo (*Active Power*).

6.4.2 Confronto architetturale e risultati

Per giustificare la scelta dell'architettura proposta rispetto agli standard di mercato, i dati del SoC basato su FPGA sono stati confrontati con i consumi attestati e le prestazioni misurate empiricamente di due piattaforme commerciali ampiamente diffuse nell'edge computing industriale, utilizzando gli strumenti descritti nella Sezione 6.1 che hanno prodotto i risultati nella tabella 6.1

Tabella 6.1: Confronto dei tempi di esecuzione e dei cicli di clock delle architetture analizzate (payload 256 bit).

Piattaforma	**Frequenza** (MHz)	**Tempo Esecuzione** (μs)	**Cicli di Clock**
PicoRV32 su Tang Nano 9K (SW)	27	2467,0	66610
ESP32 (HW Accel.)	240	46,0	Non disponibile
STM32H7 (SW)	480	135,8	65.192
Tang Nano 9K (HW Dedicato)	27	92,8	2506

- **Soft-Core PicoRV32 su FPGA Tang Nano 9K (Elaborazione Software a 27 MHz):** Il sistema viola pesantemente il vincolo $t_{1.5}$, registrando un tempo di esecuzione misurato di $2467\ \mu$s. Inoltre, il sistema risulta energeticamente inefficiente a causa dei prolungati tempi di attività della CPU. Il chip GW1NR-9 opera con tre banchi di alimentazione separati. Tramite i tool di sviluppo Gowin EDA è stato possibile ricavare i consumi simulati specifici per questa architettura, come riportato in Figura 6.6.

Voltage Source	Voltage	Dynamic Current(mA)	Quiescent Current(mA)	Power(mW)
VCC	1.200	7.441	3.499	13.129
VCCX	3.300	0.076	5.000	16.751
VCCIO18	1.800	0.075	1.927	3.604

Figura 6.6: Consumi simulati tramite Gowin EDA per i banchi di alimentazione.

Sommando i contributi di potenza delle tre linee di alimentazione (VCC, VCCX e VCCIO18) visibili nella simulazione, si ottiene il consumo totale stimato per l'architettura:

$$P_{totale} = 13{,}129 + 16{,}751 + 3{,}604 = \mathbf{33{,}484\ mW} \tag{6.1}$$

Utilizzando questo valore di potenza assorbita, è possibile determinare il seguente bilancio energetico complessivo per l'operazione:

$$E \approx \frac{33\ \text{mW} \cdot 2467\ \mu\text{s}}{256\ \text{bit}} \approx \mathbf{318{,}0\ nJ}/\text{bit} \tag{6.2}$$

- **Espressif ESP32 (Misurazione esecuzione software con accelerazione hardware a 240 MHz):** Il test empirico condotto sulla libreria crittografica integrata ha registrato un tempo di esecuzione di $46\ \mu$s. La rapidità di calcolo soddisfa ampiamente le specifiche temporali del Modbus (grazie al suo hardware dedicato), ma l'assorbimento energetico risente della potenza attiva richiesta dal SoC. Considerando una tensione nominale

di 3.3 V [32] e un consumo di corrente in modalità di calcolo attivo che il datasheet attesta tra 30 mA e 68 mA [32] (a seconda del carico), la potenza assorbita quindi varia all'interno di un range.
La minima risulta:

$$P = 3.3 \text{ V} \cdot 30 \text{ mA} = \mathbf{99 \text{ mW}} \tag{6.3}$$

mentre la massima pari a:

$$P = 3.3 \text{ V} \cdot 68 \text{ mA} = \mathbf{224 \text{ mW}} \tag{6.4}$$

Questi valori pesano sul bilancio energetico complessivo secondo le formule:

$$E \approx \frac{99 \text{ mW} \cdot 46\,\mu\text{s}}{256 \text{ bit}} \approx \mathbf{17{,}8 \text{ nJ}}/\text{bit} \tag{6.5}$$

$$E \approx \frac{224 \text{ mW} \cdot 46\,\mu\text{s}}{256 \text{ bit}} \approx \mathbf{40{,}3 \text{ nJ}}/\text{bit} \tag{6.6}$$

- **STMicroelectronics STM32H7 (Misurazione esecuzione software a 480 MHz):** Il test empirico ha rivelato che, eseguendo l'algoritmo AES (con la stessa libreria tiny-AES-c usata per i test su tang nano 9k) via software, il processore impiega circa $65\,192$ cicli di clock, traducendosi in $135{,}8\,\mu$s. Nonostante il limite temporale $t_{1.5}$ sia rispettato grazie all'altissima frequenza utilizzata, per lo stesso motivo l'assorbimento energetico di questo microcontrollore rende l'operazione di *brute-force* molto dispendiosa. Fissando la tensione di alimentazione a 3.3 V [33], la documentazione ufficiale attesta un assorbimento di corrente tipico a questa frequenza a singolo *core* `M7` (con elaborazione dati e CPU a pieno carico) pari a 110 mA [33].
  Ciò determina una potenza attiva stimata di:

$$P = 3.3 \text{ V} \cdot 110 \text{ mA} \approx \mathbf{360 \text{ mW}} \tag{6.7}$$

  Che si traduce in:

$$E \approx \frac{360 \text{ mW} \cdot 135{,}8\,\mu\text{s}}{256 \text{ bit}} \approx \mathbf{190{,}9 \text{ nJ}}/\text{bit} \tag{6.8}$$

- **Tang Nano 9K (Misurazione esecuzione hardware a 27 MHz):** A differenza delle architetture *general-purpose* che compensano i ritardi incrementando la frequenza di clock, il SoC proposto delega il carico crittografico completamente al nostro coprocessore dedicato. L'acceleratore chiude il calcolo in $92{,}8\ \mu$s (rispettando il vincolo Modbus), pur mantenendo l'assorbimento di base del un chip a circa 33mW. Questo si traduce in un'ottimizzazione energetica significativa:

$$E \approx \frac{33\ \text{mW} \cdot 92{,}8\ \mu\text{s}}{256\ \text{bit}} \approx \mathbf{11{,}9\ nJ}/\text{bit} \tag{6.9}$$

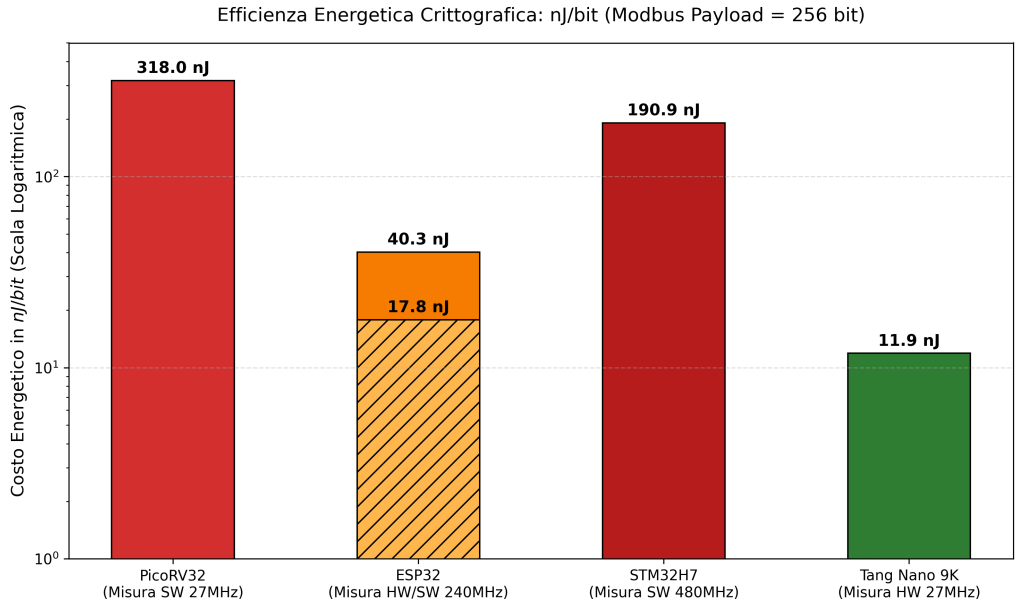

Figura 6.7: Confronto dell'efficienza energetica crittografica tra l'architettura proposta e le piattaforme commerciali (calcolo su payload da 256 bit).

Analizzando i risultati (Figura 6.7), emerge come le architetture commerciali garantiscano l'integrità temporale del bus seriale soltanto a discapito del consumo energetico del nodo. Al contrario, la *Tang Nano 9k* si conferma una tecnologia in grado di garantire una prestazione sufficiente senza un significativo aumento dei consumi elettrici. L'impiego dell'acceleratore hardware risulta quindi la soluzione più equilibrata e scalabile per l'implementazione di protocolli di *cybersecurity* nei dispositivi *edge* negli ambienti industriali critici.

Capitolo 7

Discussioni e Conclusioni

7.1 Considerazioni architetturali: CPU ad alta frequenza, ASIC ed FPGA

Per una valutazione architetturale completa è necessario analizzare le possibili alternative all'utilizzo di un FPGA. Una prima obiezione potrebbe suggerire la risoluzione dei ritardi software tramite l'utilizzo di microprocessori commerciali ad altissima frequenza (es. ordine di GHz). Come dimostrato dai test, questa soluzione pur svolgendo correttamente il compito si scontra con la realtà dell'*edge computing* industriale dove i dispositivi operano sotto stringenti limiti termici, di costo e di assorbimento di potenza. Sostituire un microcontrollore periferico con un processore ad alto consumo energetico, esclusivamente per compensare l'inefficienza di una libreria crittografica, rappresenterebbe uno spreco ingiustificabile.

Una seconda più robusta alternativa consisterebbe nell'adozione di un circuito integrato per applicazioni specifiche (ASIC), strutturato come un *System-on-Chip* custom che integri sia la CPU che l'acceleratore AES esattamente come nella architettura sviluppata. Anche se un ASIC puro risulta imbattibile in termini assoluti di velocità e miniaturizzazione, il suo impiego nell'automazione industriale presenta problematiche critiche legate all'economia dei semiconduttori. I costi di *Non-Recurring Engineering* (NRE) e di *tape-out* in fonderia rendono i chip custom insostenibili per i volumi spesso medi o bassi del mercato industriale.

Inoltre, il ciclo di vita tipico di un impianto industriale (15-20 anni) espone un hardware stampato nel silicio al rischio di obsolescenza crittografica. Quindi la scelta di una board basata su FPGA come la *Tang Nano 9K* rappresenta il compromesso progettuale ideale tra prestazioni deterministiche equivalenti a un ASIC, mantiene i costi contenuti e garantisce un hardware flessibile. La logica interna può infatti essere riprogrammata sul campo per supportare futuri

aggiornamenti degli standard di sicurezza, prolungando la vita utile del nodo e mantenendo un profilo energetico perfettamente compatibile.

7.2 Limitazioni del sistema proposto

Nonostante l'architettura *System-on-Chip* sviluppata sulla scheda *Tang Nano 9K* abbia dimostrato eccellenti proprietà in termini di determinismo temporale ed efficienza energetica, l'analisi critica del sistema evidenzia alcune limitazioni. Questi vincoli derivano sia dalle scelte progettuali volte a minimizzare l'area di silicio occupata sul chip Gowin GW1NR-9, sia dalle caratteristiche geometriche e funzionali degli standard crittografici e industriali adottati in questo lavoro di tesi.

7.2.1 Modalità operativa ECB e vulnerabilità statistica

L'acceleratore hardware sviluppato implementa l'algoritmo AES-128 operando in modalità *Electronic Codebook* (ECB), cifrando ciascun blocco da 16 byte (128 bit) in modo completamente indipendente. Dal punto di vista crittografico, questa configurazione introduce una vulnerabilità: blocchi di testo in chiaro identici generano blocchi di testo cifrato identici sotto la stessa chiave.

Nel contesto delle comunicazioni industriali basate sul protocollo Modbus RTU, i pacchetti presentano un'elevata ridondanza e schemi strutturali ripetitivi (ad esempio, le interrogazioni cicliche effettuate dal *Client* sullo stato degli stessi *Holding Registers* o le risposte del *Server* contenenti letture analogiche di sensori che mantengono valori costanti nel tempo). Un attaccante in ascolto sul bus seriale RS-485, pur non potendo decifrare direttamente il contenuto informativo del *payload*, potrebbe effettuare un'analisi statistica del traffico (*traffic analysis*), identificando pattern ricorrenti associati a specifici comandi operativi o stati dell'impianto di automazione.

7.2.2 Assenza di meccanismi anti-replay e di autenticazione forte

L'aggiunta del *layer* crittografico hardware soddisfa efficacemente il vincolo di riservatezza del dato (*confidenzialità*). Tuttavia, come evidenziato dalle linee guida sulla sicurezza discusse in precedenza, dei sistemi di controllo industriale (ICS) pubblicate dal NIST[9], una protezione *inline* robusta richiede anche proprietà di *autenticazione* e *freshness*, che l'attuale implementazione non integra:

- **Vulnerabilità ai *Replay Attack*:** Poiché la *Protocol Data Unit* (PDU) del Modbus RTU non prevede indicatori temporali (*timestamp*), numeri di sequenza progressivi o valori casuali monouso (*nonce*) all'interno del *payload*

cifrato, il sistema è intrinsecamente esposto ad attacchi di riproduzione. Un attaccante potrebbe catturare un *frame* cifrato legittimo (ad esempio, un comando di attivazione o disattivazione di un attuatore) e ritrasmetterlo sul bus in un secondo momento. L'acceleratore hardware decifrerà correttamente il pacchetto e la CPU lo considererà valido, forzando un comportamento anomalo nei PLC senza che il sistema possa rilevare l'anacronismo della transazione.

- **Mancanza di codici di autenticazione dei messaggi (MAC):** Non essendo implementato un codice di autenticazione (come l'HMAC) o una modalità di cifratura autenticata AEAD (*Authenticated Encryption with Associated Data*) quale l'AES-GCM, il sistema verifica l'integrità formale del dato solo tramite il CRC hardware del Modbus, ma non è in grado di validare matematicamente l'identità del mittente, esponendosi parzialmente ad attacchi di tipo *spoofing* qualora la chiave venisse compromessa.

7.2.3 Gestione statica delle chiavi crittografiche (*Key Management*)

Nel *firmware bare-metal* attualmente eseguito sul processore *soft-core* PicoRV32, la chiave segreta a 128 bit viene memorizzata in modo statico all'interno del codice sorgente (*hardcoded*). Questa soluzione anche se ottima per implementazione rapida e *test* presenta criticità per il *deployment* in scenari industriali reali:

- L'assenza di un protocollo di negoziazione e scambio dinamico delle chiavi (come un algoritmo *Diffie-Hellman* basato su curve ellittiche) impedisce il *rekeying* automatico e periodico delle credenziali di sessione.
- La compromissione fisica o logica di un singolo nodo periferico (*Edge Node*) comporterebbe la perdita di sicurezza dell'intero bus industriale, violando il principio di *Forward Secrecy*, in quanto la chiave compromessa permetterebbe la decifratura retroattiva di tutto il traffico precedentemente intercettato.

7.3 Risultati ottenuti

I risultati sperimentali dimostrano che l'architettura proposta introduce un incremento di latenza di $46{,}7\ \mu$s per ogni blocco di 16 byte. Questo valore rispetta i vincoli temporali *real-time* dello standard Modbus RTU, confermando la trasparenza del *layer* crittografico rispetto all'infrastruttura di rete preesistente.

Il consumo specifico misurato è pari a $11{,}9$ nJ/bit. Questo profilo energetico ridotto rende l'architettura perfetta per l'integrazione nei nodi industriali perife-

rici dell'*edge computing*, caratterizzati in ogni campo da vincoli di alimentazione stringenti e budget di potenza limitati.

7.4 Conclusioni e sviluppi futuri

In conclusione, questo lavoro di tesi ha dimostrato con successo la fattibilità tecnica di un'integrazione crittografica completamente trasparente. Il sistema progettato agisce come un vero e proprio *layer* invisibile che si sovrappone all'infrastruttura di rete preesistente e protegge il canale operando in modo del tutto inavvertibile per il bus di campo, permettendo ai nodi *legacy* di continuare a comunicare senza necessitare di alcuna modifica strutturale o aggiornamento software. Gli obiettivi prefissati nel Capitolo 1 sono stati pienamente raggiunti: l'integrazione di un acceleratore dedicato su FPGA ha permesso di risolvere il problema della trasmissione in chiaro dei dati tipica del Modbus RTU, garantendo contemporaneamente il rispetto dei rigorosi vincoli *real-time* (limite $t_{1.5}$). In particolare, il sistema risponde alla necessità critica di efficienza energetica nell'*edge computing* industriale attraverso una proposta crittografica fortemente sostenibile, capace di abbattere l'energia spesa per singolo bit cifrato. E' stato proprio questo vincolo di sostenibilità economica e di drastica riduzione dei consumi attivi che mi ha portato a scegliere una piattaforma hardware mirata e *low-cost* come la *Tang Nano 9K*.

In questo contesto, il mio contributo sviluppato nella tesi non si concentra sulla logica matematica del blocco di calcolo crittografico, già nota in letteratura, ma sull' integrazione, organizzazione e coordinamento dell'intero sistema per renderlo compatibile con le risorse fortemente limitate della scheda selezionata facendo opportune scelte di *design*. Questo traguardo è stato raggiunto attraverso la progettazione di una macchina a stati finiti (*FSM*) *custom*, ottimizzata per gestire la sequenza temporale dei *round* massimizzando il riutilizzo dell'area logica disponibile sul chip Gowin. Infine, lo sforzo progettuale si è concentrato sulla creazione di un chiaro livello di astrazione hardware, incaricato di gestire integralmente i meccanismi di *handshake* e le risposte sul bus dedicato PCPI garantendo così una comunicazione deterministica e affidabile con il *soft-core* PicoRV32 che è stato scelto proprio per l'affinità con il progetto.

Il sistema realizzato rappresenta un valido punto di partenza. Tuttavia, per operare in scenari industriali critici e garantire la piena conformità ai più recenti standard di *cybersecurity* (come la normativa internazionale IEC 62443[34]), il progetto richiede alcune integrazioni che superino le limitazioni emerse durante l'analisi.

Gli sviluppi futuri del progetto si orienteranno quindi verso le seguenti direzioni:

- **Crittografia autenticata:** Sostituzione della modalità operativa ECB con *suite* crittografiche più avanzate, come l'*AES-GCM* o l'*AES-CCM*, per eliminare le vulnerabilità statistiche e garantire contemporaneamente riservatezza e autenticazione del *payload*.
- **Gestione della *freshness* dei dati:** Introduzione di un livello di incapsulamento intermedio che aggiunga al pacchetto un *timestamp* o un contatore di sequenza prima della cifratura, al fine di mitigare definitivamente il rischio di *Replay Attack*.
- ***Key Management* dinamico:** Integrazione di un protocollo per lo scambio sicuro delle chiavi di sessione (ad esempio tramite *ECDH*), che consenta il *rekeying* dinamico e isoli crittograficamente i nodi in caso di compromissione (*Forward Secrecy*).
- **Ottimizzazione hardware del *packing*:** Introduzione di un *flag* di controllo a livello di istruzione (per esempio nel `rs1`) per segnalare all'acceleratore l'endianness (*little-endian* o *big-endian*) dei dati in ingresso senza dover andare a modificare il codice in caso di cambiamenti. Questo permetterà di demandare l'eventuale operazione di *swap* dei byte direttamente alla logica combinatoria dell'hardware, quasi istantanea, riducendo drasticamente l'*overhead* del firmware, senza però sacrificare la standardizzazione e la flessibilità architetturale raggiunte

L'approccio ibrido *hardware/software* basato su FPGA e processore RISC-V si è confermato una scelta flessibile e scalabile. Colmando le attuali limitazioni, il prototipo potrà evolvere in una soluzione completa e robusta, idonea a proteggere le comunicazioni nei moderni contesti dell'Industria 4.0.

Bibliografia

[1] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Science & Business Media, 2009.

[2] G. S. Vernam. Cipher Printing Telegraph Systems For Secret Wire and Radio Telegraphic Communications. *Transactions of the American Institute of Electrical Engineers*, XLV:295–301, 1926.

[3] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce, 11 2001.

[4] Stephen Trimberger. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, 103:318–331, 03 2015.

[5] S. Brown and J. Rose. FPGA and CPLD architectures: a tutorial. *IEEE Design & Test of Computers*, 13(2):42–57, 1996.

[6] Sipeed. Tang Nano 9K - Sipeed Wiki. `https://wiki.sipeed.com/hardware/en/tang/Tang-Nano-9K/Nano-9K`, 2023. Hardware specification and component documentation for the Tang Nano 9K FPGA Development Board.

[7] Eric Wai and C. K. M. Lee. Seamless Industry 4.0 Integration: A Multilayered Cyber-Security Framework for Resilient SCADA Deployments in CPPS. *Applied Sciences*, 13(21), 2023.

[8] Modbus Organization, Inc. *Modbus Application Protocol Specification V1.1b3*.

[9] Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to Industrial Control Systems (ICS) Security. Technical Report Special Publication (SP) 800-82 Rev. 2, National Institute of Standards and Technology (NIST), 2015.

[10] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Science & Business Media, 2002.

[11] Claude Shannon. Communication Theory of Secrecy Systems. volume 28, 01 1948.

[12] Alan Konheim. Horst Feistel: the inventor of LUCIFER, the cryptographic algorithm that changed cryptology. *Journal of Cryptographic Engineering*, 9, 04 2019.

[13] Arthur Sorkin. Lucifer, a cryptographic algorithm. *CRYPTOLOGIA*, 8:22–42, 01 1984.

[14] Philippe Guillot. Auguste Kerckhoffs et la cryptographie militaire. *BibNum*, 05 2013.

[15] Pawel Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. volume 2779, pages 319–333, 09 2003.

[16] EMG2. FPGAs in 2026: The Complete Guide to High-Performance Computing Architectures and Solutions. *EMG2 Technical Insights*, 2026.

[17] AMD. Versal ACAP: Industry's First Adaptive Compute Acceleration Platform. White paper, Advanced Micro Devices, Inc., 2021.

[18] AMD. Announcing AMD Kintex UltraScale+ Gen 2 Mid-Range FPGAs. Press release, Advanced Micro Devices, Inc., 2026.

[19] Altera, an Intel Company. Altera Accelerating FPGA-based Innovations from Edge to Cloud with Scalable Portfolio. Product brief, Intel Corporation, 2024.

[20] Embedded Computing Design. Product of the Week: Lattice Semiconductor's Lattice Avant-X Family of Mid-Range, General Purpose FPGAs. *Embedded Computing Design*, 2023.

[21] Microchip Technology Inc. PolarFire SoC FPGAs Data Sheet. Datasheet, Microchip Technology Inc., 2026.

[22] Clifford Wolf and YosysHQ. Yosys Open SYnthesis Suite. `https://github.com/YosysHQ/yosys`, 2024. A framework for RTL synthesis tools and Verilog-2005 hardware compilation.

[23] YosysHQ. nextpnr – a portable FPGA place and route tool. `https://github.com/YosysHQ/nextpnr`, 2024. Vendor neutral, timing driven, FOSS FPGA place and route tool.

[24] Gwenhael Goavec-Merou and contributors. openFPGALoader. `https://github.com/trabucayre/openFPGALoader`, 2026. A C library designed for embedded 32- and 64-bit systems. GitHub repository.

[25] Free Software Foundation. *Using the GNU Compiler Collection (GCC).* Free Software Foundation, 2026. Standard GCC configured for RISC-V 32-bit cross-compilation (-march=rv32i).

[26] Keith Packard and contributors. picolibc. `https://github.com/picolibc/picolibc`, 2026. Universal utility for programming FPGA.

[27] Stephen Williams and contributors. Icarus Verilog. `https://github.com/steveicarus/iverilog`, 2026. A free compiler implementation for the IEEE 1364-2005 Verilog standard.

[28] Tony Bybell and contributors. GTKWave. `https://github.com/gtkwave/gtkwave`, 2026. A fully featured GTK+ based electronic waveform viewer.

[29] Clifford Wolf. PicoRV32 - A Size-Optimized RISC-V CPU. `https://github.com/YosysHQ/picorv32`, 2019. Open-source, silicon-validated RV32I soft-core and Pico Co-Processor Interface specification.

[30] Federico Badiali. Modulo crittografico AES in Verilog su Tang Nano 9K. `https://github.com/Sir-Feduard/Modulo-crittografico-AES-in-Verilog-su-Tang-Nano-9K`, 2026.

[31] Modbus Organization, Inc. *Modbus over Serial Line Specification and Implementation Guide V1.02*, 2006. Specifiche ufficiali per il framing RTU e i vincoli temporali t1.5 e t3.5.

[32] Espressif Systems. *ESP32 Series Datasheet*, 2024. Valori trovati in Power Consumption by Power Modes.

[33] STMicroelectronics. *STM32H755xI Datasheet*, 2023. Valori corrente trovati a Typical and maximum current consumption in Run mode, code with data processing running from flash memory, only Arm Cortex-M7 running, cache ON, LDO regulator ON.

[34] Karl-Heinz Niemann. IEC 62443 from the planner's and operator's point of view. Technical report, Fakultät I - Elektro- und Informationstechnik, 2025.

[35] Morten Jensen. tiny-AES-c - Small portable AES128/192/256 in C. `https://github.com/kokke/tiny-AES-C`, 2014. Small and portable implementation of the AES ECB, CTR and CBC encryption algorithms written in C.

[36] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer Science & Business Media, 2004.