

Universitatea Națională de Știință și Tehnologie POLITEHNICA
București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

*Optimizarea Performanței și Securității prin
Implementarea Algoritmului SM4 pe FPGA*

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de
Inginer în domeniul *IETTI*
programul de studii de licență *TST*

Conducător științific
Dr. Ing. Voichița DRAGOMIR

Absolvent
Vladimir FRUNZĂ

Anul 2025

TEMA PROIECTULUI DE DIPLOMĂ
a studentului **FRUNZĂ C. Vladimir, 441C**

1. Titlul temei: Optimizarea Performanței și Securității prin Implementarea Algoritmului Criptografic SM4 pe FPGA

2. Descrierea temei:

Algoritmul criptografic SM4, un standard recunoscut pentru securitatea datelor, este implementat folosind limbajul Verilog pe un circuit programabil FPGA. Designul include dezvoltarea logicii de criptare pe 128 de biți, utilizând transformări neliniare S-box, operații liniare și generarea cheilor pentru cele 32 de runde de procesare. Testele funcționale verifică acuratețea criptării prin compararea rezultatelor cu valori de referință, iar succesul sau eșecul procesului este indicat vizual prin două LED-uri, verde pentru succes și roșu pentru eșec. FPGA-ul controlează LED-urile prin ieșiri digitale, activând LED-ul verde sau roșu în funcție de rezultatul comparației dintre textul criptat și valoarea de referință.

3. Contribuția originală:

1. Dezvoltarea unui design digital bazat pe Algoritmul criptografic SM4, în Verilog.
2. Scrierea unei suite de teste funcționale pentru a verifica validitatea criptarilor.
3. Implementarea designului, pe un circuit programabil FPGA.
4. Conectarea ieșirilor digitale ale circuitului FPGA la 2 led-uri, unul verde și unul roșu, pentru vizualizarea criteriilor PASS/FAIL impuse de teste.

4. Resurse și bibliografie:

5. Data înregistrării temei: 2025-01-23 20:53:42

Conducător(i) lucrare,
Ş.L. dr. ing. Voichița DRAGOMIR

Student,
FRUNZĂ C. Vladimir

Director departament,
Conf. dr. ing. Șerban OBREJA

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **98d1c4b65a**

Copyright © 2025 , *Vladimir Frunză*

Toate drepturile rezervate

Autorul acordă UPB dreptul de a reproduce și de a distribui public copii pe hîrtie sau electronice ale acestei lucrări, în formă integrală sau parțială.

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul Optimizarea Performanței și Securității prin Implementarea Algoritmului SM4 pe FPGA, prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității Naționale de Știință și Tehnologie POLITEHNICA Bucureștiă cerință parțială pentru obținerea titlului de *Inginer* în domeniul *IETTI*, programul de studii *TST* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățămînt superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, *data*

27.06.2025

Absolvent *Vladimir FRUNZĂ*

(semnatură în original)

Cuprins

Lista figurilor	11
Lista tabelelor	13
Lista acronimelor	15
Introducere	17
1 Criptografie: definiții și termeni importanți	19
1.1 Fundamente matematice	19
1.2 Criptografie simetrică (cu cheie secretă) vs. criptografie cu cheie publică	23
1.3 Cifruri bloc (block cyphers) vs. cifruri flux (stream cyphers)	26
1.4 Fundamentele și obiectivele criptografiei moderne	30
2 Stadiul actual al criptografiei (SotA)	33
2.1 Evoluția cifrurilor pe bloc clasice și moderne	33
2.2 Atacuri împotriva cifrurilor pe bloc – de la tehnici clasice la provocările actuale	38
2.3 Tendințe și direcții emergente în criptografie	40
3 Algoritmul chinezesc de criptare bloc SM4	43
3.1 Istoric, origini și statutul de standard	43
3.2 Structură și funcționare	44
3.3 Analiză comparativă a SM4 cu alte cifruri bloc	48
4 Implementarea algoritmului SM4 în SystemVerilog	51
4.1 Diagrama generală a modulelor	52
4.2 Funcțiile asincrone și memoriile pentru constante	53
4.3 APB: interfața de comunicație	56
4.4 Funcția de criptare	58
4.5 Funcția de decriptare	66
4.6 Memoria pentru cheile de rundă	68
4.7 SM4 Core: mașina de stări pentru operarea criptării/decriptării	70
4.8 SM4 Top Module: implementarea modurilor de operare	71
4.9 Simulation Testbench – testarea funcționalității în simulatorul Vivado	73
5 Implementarea pe FPGA	79
5.1 Boolean Board FPGA development board - caracteristici generale	79

5.2	Necesitatea sintetizabilității codului de SystemVerilog pentru implementarea pe FPGA	81
5.3	Design Test Wrapper: implementarea unui modul de test sintetizabil	84
5.4	Wrapper Testbench pentru testarea modulului și rezultatele obținute pe placă	86
Bibliografie		97
Anexa 1 – Codul pentru SM4 Top Module		98
Anexa 2 – Codul pentru logica de primary din Simulation Testbench		107
Anexa 3 – Codul pentru Design Test Wrapper		111

Lista figurilor

1.2.1 Schemă de criptare cu cheie privată	24
1.2.2 Problema cu sistemele criptografice simetrice	25
1.2.3 Schemă de criptare cu cheie publică	25
1.3.1 Funcționarea unui cifru flux	27
1.3.2 Encriptia in modul ECB a logo-ului Linux	28
1.3.3 Schema criptării modului CBC	28
1.3.4 Schema criptării modului CFB	29
1.3.5 Schema criptării modului OFB	29
1.3.6 Schema criptării modului CTR	30
1.3.7 Criptarea in modul ECB a logo-ului Linux	30
2.1.1 Rețea Feistel cu $n+1$ runde	33
2.1.2 Rețea Substituție-Permutare (SP)	34
2.1.3 Structura Cifrului DES	35
2.1.4 Structura Cifrului AES, cu cheie de 128 bit	36
2.3.1 Bit vs. Qubit	41
3.2.1 Structura rețelei Feistel neechilibrate a algoritmului SM4	45
3.2.2 Funcția de rundă a algoritmului SM4	45
3.2.3 Exemplu rezolvat pentru aplicarea L-function în SM4	46
3.2.4 Substitution-Box pentru algoritmul SM4	46
4.1.1 Diagrama generală pentru implementarea SM4	53
4.3.1 Cum arată un transfer de scriere pe APB	56
4.4.1 FSM-ul Criptării pentru Immediate Encryption	60
4.4.2 FSM-ul Generatorului de Chei	60
4.4.3 FSM-ul criptării cu chei pregenerate	64
4.7.1 FSM-ul SM4 Core	71
4.8.1 FSM-ul SM4 Top Module	72
5.1.1 Poză cu Boolean Board de la Xilinx	79
5.3.1 Ierarhia de automate din Design Test Wrapper	84
5.3.2 FSM-ul pentru Test State	85
5.3.3 FSM-ul pentru Big State	86
5.4.1 Funcționarea unui ecran LED cu 7 segmente	87
5.4.2 Forma de undă pentru Display Wrapper	92

5.4.3 Poză cu FPGA-ul în Reset	92
5.4.4 Poză cu FPGA-ul după apăsarea butonului, cu codul corect	93
5.4.5 Poză cu FPGA-ul după apăsarea butonului, cu codul alterat	93

Lista tabelelor

3.3.1 Comparație extinsă între AES-128, SM4, 3DES și Camellia-128	48
5.1.1 Resurse relevante ale FPGA-ului <i>Spartan-7 XC7S50</i>	80

Lista acronimelor

- FPGA = *Field-Programmable Gate Array* / Matrice de porti logice programabila in teren
- SM4 = *Chinese Standard Block Cipher* / Algoritm de criptare cu bloc SM4
- RTL = *Register-Transfer Level* / Nivel de transfer intre registre
- SV = *SystemVerilog* / Limbaj de descriere hardware SystemVerilog
- FSM = *Finite State Machine* / Automat finit de stare
- ILA = *Integrated Logic Analyzer* / Analizor logic integrat
- APB = *Advanced Peripheral Bus* / Magistrala periferica avansata
- AXI = *Advanced eXtensible Interface* / Magistrala AXI
- BRAM = *Block RAM* / Memorie de tip bloc
- DSP = *Digital Signal Processing slice* / Unitate de procesare a semnalului digital
- LUT = *Look-Up Table* / Tabel de cautare
- MMCM = *Mixed-Mode Clock Manager* / Manager de ceas in regim mixt
- PLL = *Phase-Locked Loop* / Bucla inchisa de fază
- CMT = *Clock Management Tile* / Modul de management al ceasului
- IOB = *Input/Output Block* / Bloc de intrare-iesire
- JTAG = *Joint Test Action Group* / Interfață standard JTAG
- UART = *Universal Asynchronous Receiver-Transmitter* / Receptor-transmițător asincron universal
- HDMI = *High-Definition Multimedia Interface* / Interfață multimedia de înaltă definiție
- FIFO = *First-In, First-Out* / Coada „primul intrat – primul ieșit”
- ROM = *Read-Only Memory* / Memorie numai-citire
- RAM = *Random-Access Memory* / Memorie cu acces aleator
- DUT = *Device Under Test* / Dispozitiv aflat in testare
- TB = *Testbench* / Banc de test
- ECB = *Electronic Codebook* / Modul Electronic Codebook
- CBC = *Cipher Block Chaining* / Înlănțuirea blocurilor de cifru
- CFB = *Cipher Feedback* / Feedback de cifru
- OFB = *Output Feedback* / Feedback de ieșire
- CTR = *Counter Mode* / Modul contor
- M.O. = *Mode of Operation* / Mod de operare
- XADC = *Xilinx Analog-to-Digital Converter* / Convertor ADC integrat
- BRAM36 = Bloc RAM de 36 kbit / Bloc de memorie de 36 kbit

Introducere

Într-o lume modernă plină de provocări și pericole, unul dintre cele mai importante aspecte la care trebuie să fim atenți este *securitatea personală*. Aceasta se manifestă în multe forme, de la securitatea fizică la cea emotională și materială. Un element legat de securitate pe care îl omitem adesea în prezentul futurist în care coexistăm este faptul că majoritatea bunurilor noastre materiale există, sau ne sunt atribuite – în format *digital*. Cel mai evident exemplu sunt banii de pe card, dar și cărțile și documentele de identitate, actele de proprietate pentru casă și mașină și multe alte documente de mare importanță pentru noi sunt stocate electronic undeva. Cu siguranță, nimeni nu și-ar dori ca oricine să le poată accesa fără permisiune, însă deseori uităm că protejarea proprietății digitale este mult mai complicată decât cea a bunurilor fizice; soluția nu mai poate fi o simplă cameră încuiată cu cheia într-un colț întunecat al unei clădiri păzite.

„Criptografie” este un termen care își are etimologia în limba greacă, provenind de la cuvintele „κρυπτός” (kryptós), care înseamnă „ascuns”, și „γράφειν” (gráfein), care înseamnă „a scrie”. Această știință este folosită încă din Antichitate, cele mai vechi exemple cunoscute provenind din Egiptul Antic: în mormântul nobilului Khnumhotep II au fost găsite tăblițe inscripționate cu o serie de hieroglife neobișnuite, descoperite de o echipă de arheologi britanici la sfârșitul secolului al XIX-lea. Ulterior, textul a fost analizat de criptologi, ajungându-se la concluzia că modificările aveau, cel mai probabil, scop estetic, nu de secretizare; cu toate acestea, înscrisul rămâne considerat un exemplu de text „criptat”.

Deși există numeroase apariții istorice, din ce în ce mai sofisticate, ale criptografiei, definiția modernă o plasează în zona științelor exacte: este ramura matematicii care se ocupă de confidențialitatea, autenticitatea și integritatea informației. În sistemele digitale moderne, toată informația este *scrisă* sub forma de biți și este stocată sau transmisă ca fluxuri de biți (*bitstreams*), de aici fiind justificat sufixul „-grafie” (din gr. „γραφή” — „scriere”). Criptografia oferă răspunsul la problema securității datelor digitale, în timp ce *criptanaliza* caută breșe în sistemele criptografice pentru a obține, în mod neautorizat, acces la conținutul mesajelor cifrate. Cele două discipline evoluează în tandem, iar „valsul” lor matematic face ca stocarea digitală a datelor în era informațională să fie metoda preferată de majoritatea utilizatorilor, ajungând să fie mai sigură și mai facilă decât stocarea fizică.

1. Criptografie: definiții și termeni importanți

1.1. Fundamente matematice

Ramura matematicii numită *aritmetică modulară* descrie un sistem de operații asupra numerelor întregi, aplicate pe resturile împărțirii cu un modul fix n , tratând două numere drept echivalente atunci când au același rest. Aritmetica modulară oferă un cadru numeric în care adunarea, înmulțirea și exponentierea se pot efectua rapid, în timp ce operațiile inverse sunt computațional dificile; tocmai această asimetrie este exploatață de schemele criptografice moderne. Conceptele fundamentale ale aritmeticii modulare sunt folosite atât pentru proiectarea, cât și pentru analiza sistemelor criptografice, iar nevoia continuă de securitate informatică a stimulat, la rândul ei, dezvoltarea teoriei. În cele ce urmează trecem în revistă principalele idei.

Congruența modulo n și inelul \mathbb{Z}_n

Două întregi a și b sunt congruente, $a \equiv b \pmod{n}$, dacă împart același rest la împărțirea cu n . Resturile $0, 1, \dots, n - 1$ formează inelul finit \mathbb{Z}_n , unde adunarea și înmulțirea sunt *închise* și ușor de implementat. RSA, de pildă, operează în \mathbb{Z}_n cu $n = pq$, iar Diffie–Hellman folosește submultimi ale \mathbb{Z}_p^* .

Inversa modulară și Algoritmul Euclid extins

Un element $a \in \mathbb{Z}_n$ este inversabil dacă $\gcd(a, n) = 1$; inversa a^{-1} satisfacă $aa^{-1} \equiv 1 \pmod{n}$. Algoritmul Euclid extins o calculează în complexitate $O(\log(n))$ și este indispensabil la generarea cheii private RSA ($d \equiv e^{-1} \pmod{\varphi(n)}$) și la semnăturile DSA/ECDSA, unde nonce-ul trebuie inversat modulo ordinul grupului.

Exponentiere modulară rapidă

Pentru un exponent mare e , calculul direct al lui $a^e \pmod{n}$ ar necesita $e - 1$ înmulțiri modulare. Algoritmul *square-and-multiply*, ce constă în scrierea exponentului în binar urmată de alternarea operațiilor „pătrat” și „înmulțire condiționată”, reduce costul la $O(\log(e))$ pași. Pseudocodul standard este:

Algorithm 1: ExpMod (a,e,n)

Input: $a, e, n \in \mathbb{Z}$, cu $n > 0$ **Result:** $a^e \bmod n$ $R \leftarrow 1; A \leftarrow a;;$ **foreach** bit e_i al lui e de la LSB la MSB **do** **if** $e_i = 1$ **then** $R \leftarrow R \cdot A \bmod n;$ $A \leftarrow A^2 \bmod n;$ **return** $R;$

Exemplu. Calculăm $13^{2019} \bmod 77$. Scriem exponentul în binar: $2019_{10} = 11111100011_2$.

Tabelul de mai jos arată evoluția registrelor R și A (toate valorile sunt reduse modulo 77).

bit e_i	R după pas	A după pas	operații aplicate
1	13	15	$R \leftarrow RA, A \leftarrow A^2$
1	41	71	$R \leftarrow RA, A \leftarrow A^2$
0	41	36	$A \leftarrow A^2$
0	41	64	$A \leftarrow A^2$
0	41	15	$A \leftarrow A^2$
1	76	71	$R \leftarrow RA, A \leftarrow A^2$
1	6	36	$R \leftarrow RA, A \leftarrow A^2$
1	62	64	$R \leftarrow RA, A \leftarrow A^2$
1	41	15	$R \leftarrow RA, A \leftarrow A^2$
1	76	71	$R \leftarrow RA, A \leftarrow A^2$

După procesarea ultimului bit obținem $R = 6$, deci

$$13^{2019} \bmod 77 = 6.$$

Chiar și pentru exponenți de ordinul 2^{2048} , numărul de pași crește doar liniar în lungimea exponentului, ceea ce face cifrările RSA sau schimburile Diffie–Hellman practicabile pe hardware obișnuit.

Mica teoremă a lui Fermat și teorema lui Euler

Textul pentru mica teoremă a lui Fermat (FLT) spune: pentru un prim p și $a \not\equiv 0 \pmod p$, avem $a^{p-1} \equiv 1 \pmod p$. Teorema lui Euler generalizează: dacă $\gcd(a,n)=1$, atunci $a^{\varphi(n)} \equiv 1 \pmod n$. Corectitudinea criptosistemului Rivest-Shamir-Adleman (RSA), unul dintre cele mai vechi sisteme de

transmisie securizată a datelor din lume, rezultă direct din această proprietate, iar testele de primalitate tip Miller–Rabin, folosite des în criptografie, verifică deviații de la FLT pentru baze aleatoare.

Algoritmul Euclid (determinarea gcd -ului)

Algoritmul clasic al lui Euclid calculează cel mai mare divizor comun (gcd) al două întregi a și b folosind împărțiri repetitive până când restul devine zero. Complexitatea este $O(\log(\min(a,b)))$, ceea ce îl face extrem de eficient chiar și pentru numere de sute de biți. Pseudocodul standard este:

Algorithm 2: Algoritmul Euclid clasic

```
Input:  $a, b \in \mathbb{Z}$ 
Result:  $\gcd(a, b)$ 
while  $b \neq 0$  do
     $(a, b) \leftarrow (b, a \bmod b);$ 
return  $a;$ 
```

Exemplu — $\gcd(128,35)$

pas	(a, b)	$a \bmod b$
0	$(128, 35)$	$128 \bmod 35 = 23$
1	$(35, 23)$	$35 \bmod 23 = 12$
2	$(23, 12)$	$23 \bmod 12 = 11$
3	$(12, 11)$	$12 \bmod 11 = 1$
4	$(11, 1)$	$11 \bmod 1 = 0$

La pasul 4 restul devine zero, deci $\gcd(128,35)=1$.

Teorema Resturilor Chineze (CRT)

Pentru moduli coprimi n_1, \dots, n_k , sistemul $x \equiv a_i \pmod{n_i}$ are soluție unică modulo $N=n_1 \cdots n_k$. Implementările RSA folosesc CRT pentru a descompune decriptarea în două exponentieri mai mici, obținând o accelerare de circa patru ori.

Algorithm 3: Reconstrucția x prin Teorema Resturilor Chineze

Input: $a_1, n_1, a_2, n_2 \in \mathbb{Z}$ cu $\gcd(n_1, n_2)=1$

Resultat: x astfel încât $x \equiv a_1 \pmod{n_1}$ și $x \equiv a_2 \pmod{n_2}$

$N \leftarrow n_1 \cdot n_2;$

$m_1 \leftarrow N/n_1; m_2 \leftarrow N/n_2;$

$;$

(m_i este produsul celorlalți moduli);

$u_1 \leftarrow m_1^{-1} \pmod{n_1}; u_2 \leftarrow m_2^{-1} \pmod{n_2};$

$x \leftarrow (a_1 * m_1 * u_1 + a_2 * m_2 * u_2) \pmod{N};$

return $x;$

Exemplu: găsește x cu $x \equiv 2 \pmod{3}$ și $x \equiv 3 \pmod{5}$.

- $n_1=3, n_2=5 \Rightarrow N=15$
- $m_1=N/n_1=5, m_2=N/n_2=3$
- $u_1=5^{-1} \pmod{3}=2, u_2=3^{-1} \pmod{5}=2$
- $x=(2 \cdot 5 \cdot 2 + 3 \cdot 3 \cdot 2) \pmod{15} = (20+18) \pmod{15} = 38 \pmod{15} = 8$

Verificare: $8 \pmod{3} = 2$ și $8 \pmod{5} = 3$, deci soluția corectă este $x \equiv 8 \pmod{15}$.

Problema logaritmului discret și grupurile ciclice

Într-un grup finit generat de un element g , notat $\langle g \rangle$, problema logaritmului discret (DLP) constă în a determina exponentul x din relația $g^x=h$. Operația „dusă” – calculul lui g^x pentru un x dat – se face rapid prin exponentiere modulară; „întorsul”, adică găsirea lui x când cunoști doar g și h , nu are algoritm sub-exponențial cunoscut atunci când ordinul grupului este un prim mare (de pildă $\sim 2^{256}$). Această asimetrie stă la baza:

- **Protocolul Diffie–Hellman** — doi participanți publică doar g^a și g^b ; fără a rezolva DLP, un adversar nu poate afla cheia comună g^{ab} .
- **Semnăturilor DSA/Schnorr** — siguranța lor se reduce la imposibilitatea de a extrage exponentul secret din congruențe de forma $k^{-1}(h + ax) \pmod{q}$.
- **Criptografiei pe curbe eliptice (ECC)** — grupul punctelor unei curbe peste \mathbb{F}_p este ciclic de ordin prim; DLP pe curbe rămâne intractabil chiar pentru chei de 256 biți, furnizând același nivel de securitate ca RSA-2048.

Reprezentarea datelor și paddingul

În practică, un mesaj trebuie convertit într-un întreg din \mathbb{Z}_n înainte de criptare sau semnare. Această conversie este critică, deoarece mesaje „patologice” pot compromite schema. De aceea se folosesc scheme de completare controlată:

- **OAEP** (Optimal Asymmetric Encryption Padding) combină mesajul cu un vector aleator și două funcții hash, oferind aleatorizare și integritate; previne atacurile cu text ales asupra RSA.
- **PKCS #1 v1.5** definește şablonul 0x00 0x02 <Padding Random> 0x00 <Mesaj>. Deși vechi, rămâne folosit la semnături; implementările care nu verifică strict octetii sunt vulnerabile la atacul Bleichenbacher.
- **ASN.1 / DER** specifică modul exact de serializare (lungimi, tipuri de câmpuri) pentru ca aceeași semnătură să fie validată identic pe platforme diferite.
- **Base64, big-endian vs. little-endian** sunt encodări de transport; inversarea octetilor poate modifica complet valoarea numerică, deci și rezultatul criptării.

Fără un padding adecvat, un mesaj scurt precum „0” devine un rest numeric extrem de mic; criptarea sa cu RSA produce un ciphertext ușor de recunoscut și manipulat. De aceea schemele de completare și convențiile de codare sunt părți esențiale ale securității, nu simple detalii de implementare.

1.2. Criptografie simetrică (cu cheie secretă) vs. criptografie cu cheie publică

Pentru a putea începe să clasificăm sistemele criptografice, trebuie să stabilim câțiva termeni de bază. Un transfer criptografic complet implică minimum două entități: un *emisar* (cel care trimite informația) și cel puțin un *receptor* (cel care o primește). Fluxul tipic se desfășoară astfel:

1. **Alcatuirea mesajului** – emisarul compune mesajul în forma sa lizibilă, acesta având denumirea de „text în clar” (*plain text*).
2. **Criptarea**. Emisarul introduce mesajul în algoritmul de criptare împreună cu **cheia de criptare**. Cheia este un sir de biți generat (de regulă) aleator și suficient de lung pentru a nu putea fi ghicit; ea rămâne fixă pe toată durata comunicării. Algoritmul procesează cele două intrări și produce la ieșire **textul criptat** (*cipher text*), un sir de biți care pare aleator, și în care nu se mai poate regăsi niciuna dintre caracteristicile mesajului inițial.
3. **Transmiterea**. Textul criptat parcurge canalul de comunicație până la receptor(i).
4. **Decriptarea**. Fiecare receptor aplică funcția de decriptare a același algoritm, împreună cu o cheie corespunzătoare și obține **textul descifrat** (*decrypted text*). Dacă procesul a decurs fără erori, acesta este identic cu textul în clar inițial.

Cea mai simplă și evidentă modalitate prin care se pot diferenția sistemele criptografice este după metodele diferite de gestionare ale cheilor de criptare necesare pentru efectuarea unui transfer.

Atunci când majoritatea dintre noi ne gândim la un sistem criptografic, impunem automat o premiză fixă fără să ne dam seama. De la primele experiențe din copilărie, când am scris pentru prima dată un cifru criptat, folosind fără să ne știm un Cifru Cezar (cifru de înlocuire), întotdeauna ne-am gândit la sisteme de **criptografie simetrică**.

Sistemele simetrice (numite și *cu cheie privată*) se caracterizează prin faptul că aceeași cheie trebuie cunoscută atât de emițător, cât și de receptor, rămânând ascunsă publicului. Deși aceasta este cea mai veche schemă, rămâne larg utilizată deoarece:

- algoritmii sunt relativ ușor de implementat și foarte rapizi;
- variantele moderne oferă un nivel ridicat de securitate atunci când cheia este păstrată secretă.

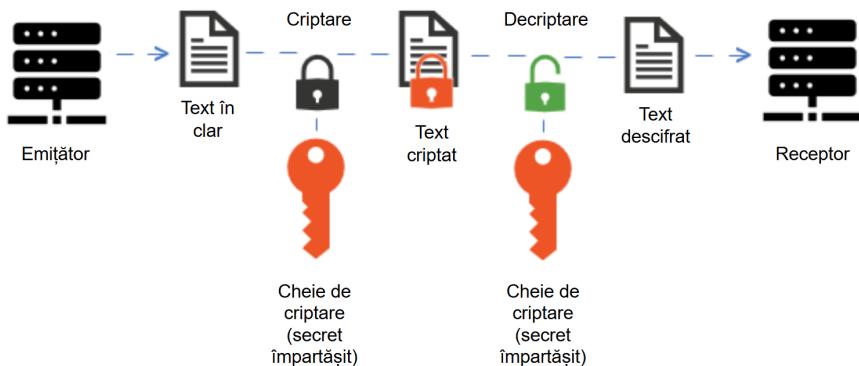


Figura 1.2.1: Schemă de criptare cu cheie privată

Cu toate că aceste sisteme sunt foarte răspândite și foarte sigure, modul în care comunicăm în epoca modernă ridică o problemă fundamentală pe care ele nu o pot rezolva. Să considerăm următorul scenariu:

Avem doi utilizatori ai internetului, Ana și Bogdan. Ei nu s-au întâlnit niciodată, deci nu au putut conveni în prealabil ce cheie privată să folosească pentru a comunica printr-un sistem criptografic simetric. Totuși, Ana dorește să-i trimită lui Bogdan un mesaj pe internet. Pentru ca această comunicare să fie sigură, Ana ar trebui să transmită, împreună cu mesajul, și cheia de criptare. Dacă însă cheia trece printr-un canal public, orice persoană ar putea să o intercepteze și ar putea descifra toate mesajele ulterioare, anulând complet efectul criptării. Prin urmare, Ana și Bogdan ar avea nevoie de un canal deja securizat pentru a împărtăși cheia; dar pentru a securiza acel canal le-ar trebui, la rândul său, o cheie de criptare, iar problema devine evident circulară.

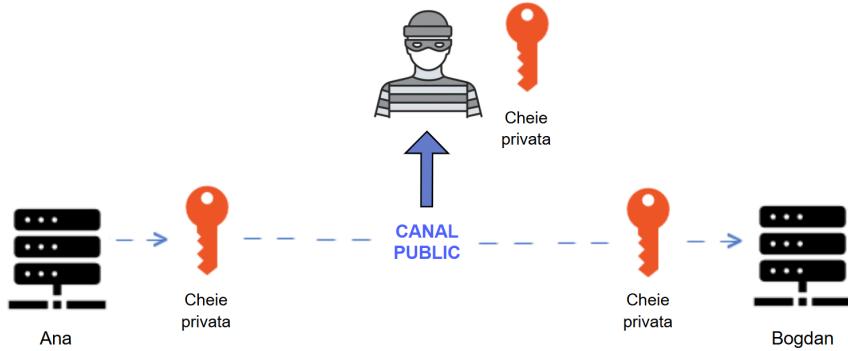


Figura 1.2.2: Problema cu sistemele criptografice simetrice

Soluția propusă de criptografia modernă o reprezintă sistemele criptografice cu cheie publică. Un sistem cu cheie publică (numit și criptografie asimetrică) folosește două chei diferite, matematic corelate între ele: o cheie publică, distribuită liber tuturor participanților, și o cheie privată, păstrată strict secret de proprietar. Cheia publică servește la criptarea mesajelor, în timp ce cheia privată permite decriptarea. Dacă, însă, pentru criptare și pentru decriptare se utilizează ambele chei, acestea capătă rolul de semnături digitale, asigurând practic ambii utilizatori că mesajele transmise pot proveni și pot fi citite doar de partenerul de comunicare.

Siguranța schemei se bazează pe existența unei funcții „cu sens unic”: operația directă — de pildă ridicarea la putere modulo un număr mare (RSA) ori înmulțirea unui punct pe o curbă eliptică (ECC) — se calculează rapid, dar operația inversă (factorizarea modulului, respectiv logaritmul discret pe curba eliptică) este considerată computațional intractabilă la dimensiuni suficiente ale cheilor. Printre algoritmii consacrați cu cheie publică se numără RSA, Diffie–Hellman, DSA și schemele pe curbe eliptice (ECDH, ECDSA, Ed25519). Sistemele cu cheie publică stau la baza protoocoalelor moderne de securitate (TLS, PGP, SSH), unde sunt folosite pentru schimbul inițial de chei simetrice, pentru certificatele X.509 și pentru semnăturile software.

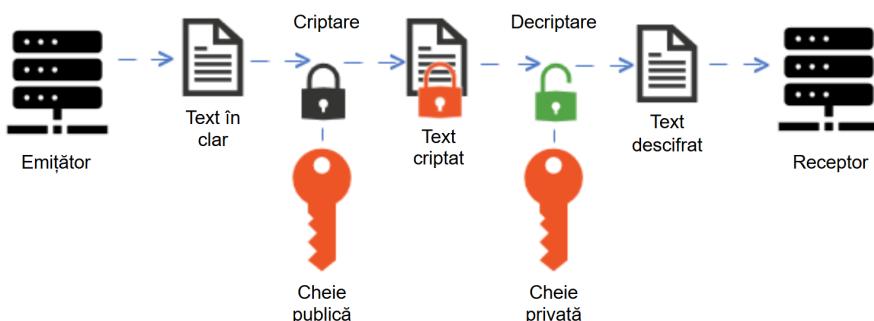


Figura 1.2.3: Schemă de criptare cu cheie publică

În continuare, această lucrare se va concentra mai mult asupra algoritmilor cu cheie privată; totuși, este important să știm că, fără algoritmii asimetrici, criptografia în sensul modern nu ar putea exista.

1.3. Cifruri bloc (block cyphers) vs. cifruri flux (stream cyphers)

După ce am analizat modul în care sistemele criptografice se deosebesc prin gestionarea cheilor, putem aborda un alt criteriu de clasificare, bazat de data aceasta pe felul în care tratează textul. Înainte de a continua, este însă esențial să introducем două concepte fundamentale: *confuzia* și *difuzia* în criptografie.

Termenii acestia au fost introdusi de Claude Shannon (cunoscut drept „părintele Teoriei Informației”), pentru a descrie cele două proprietăți esențiale pe care trebuie să le aibă un cifru modern ca să reziste analizelor statistice. **Confuzia** este definită drept capacitatea unui cifru de a ascunde legătura directă dintre cheie și textul criptat. Principiul spune că un atacator nu trebuie să poată deduce nici măcar parțial biți ai cheii observând cum se schimbă ieșirea. Aceasta se obține prin funcții nestructurante, în general neliniare, care par că schimbă „haotic” ieșirea odata cu schimbarea intrării (ex.: Substitution Boxes).

Difuzia presupune ca un singur bit modificat în textul în clar ar trebui să afecteze, aparent aleator, cât mai mulți biți din textul criptat, răspândind redundanța pe o zonă cât mai largă. În blocurile cifrului, difuzia este realizată prin permutări liniare, amestecuri de rânduri și coloane ori prin adunări modulare, astfel încât modelele statistice din mesaj să dispară rapid. Într-un algoritm bun, operațiile ar trebui să împărăște efectul fiecărui octet pe întregul mesaj.

Un cifru cu difuzie și confuzie bine echilibrate face practic imposibilă deducerea cheii prin simpla observare a textelor criptate, chiar și atunci când atacatorul cunoaște părți ale mesajelor originale. Aceste două proprietăți sunt extrem de importante în definirea și înțelegerea diferențelor dintre cifrurile de tip bloc și cele de tip flux.

Cifruri flux (stream cyphers)

În aplicațiile reale ale criptografiei, lungimea mesajului de criptat nu este, în general, cunoscută dinainte. Această situație a creat nevoie unor metode capabile să proceseze volume finite, dar imprevizibile, de date și a condus la două abordări distincte: cifrurile pe flux, care operează bit cu bit (sau octet cu octet), și cifrurile pe blocuri, care lucrează pe segmente fixe de 64, 128 de biți ori mai mult.

Primele au apărut cifrurile pe flux, datorită mecanismului lor mai intuitiv. Un exemplu celebru este mașina *Enigma*, folosită de armata germană în al Doilea Război Mondial și spartă de echipa din Bletchley Park, la care a lucrat și Alan Turing, „părintele informaticii”. Decriptarea Enigma a influențat decisiv cursul războiului și este considerată unul dintre momentele de naștere ale criptografiei moderne.

Funcționarea unui cifru pe flux este simplă: se generează un sir pseudo-aleator de biți — *keystream*

— și se combină cu textul în clar prin operația XOR, prelucrând datele secvențial. Nu este nevoie de *padding*, iar latența redusă le face ideale pentru transmisii în timp real sau pentru dispozitive cu resurse limitate. Pentru a menține securitatea, cheia secretă și vectorul de inițializare (*nonce* — starea de pornire a generatorului) nu trebuie reutilizate; dacă același keystream se aplică la două mesaje diferite, XOR-ul celor două cifruri dezvăluie XOR-ul textelor în clar. Cifrurile moderne rezolvă problema printr-un contor intern și nonce unic la fiecare mesaj. Exemple reprezentative: RC4 (în prezent depreciat), Salsa20/ChaCha20 și SNOW 3G.

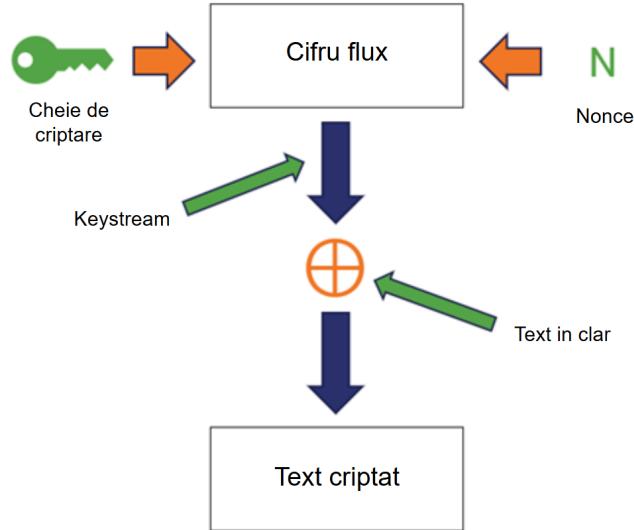


Figura 1.3.1: Funcționarea unui cifru flux

În pofida numeroaselor avantaje, cifrurile pe flux au totuși un neajuns major. Pornind de la noțiunile definite anterior, putem afirma cu certitudine că, deși un cifru pe flux excelează la capitolul *confuziei* – corelația dintre modificările din mesajul clar și cele din textul criptat fiind practic nulă – modul său de operare nu îi permite să ofere *difuzie* în sensul lui Shannon. Generatorul procesează cel mult un bit (ori un octet) pe rând, așa că nu poate răspândi o perturbare asupra mai multor biți simultan. În plus, proiectarea unui cifru pe flux compatibil cu standardele moderne este relativ complexă, ceea ce a impus căutarea unei alternative.

Cifruri bloc (block cyphers)

Apărute mai târziu decât cifrurile pe flux, *cifrurile pe bloc* procesează datele în fragmente fixe de 64, 128 biți, sau chiar mai mari. Deși, la prima vedere, acest lucru pare să complice algoritmul, în realitate oferă avantaje clare în termeni de **confuzie** și **difuzie**:

- **Confuzia** este realizată prin aceleași mecanisme ca la cifrurile pe flux, adică prin funcții neliniare (*S-box-uri*, adunări, rotații, etc.).
- **Difuzia** devine mult mai facil de atins, deoarece fiecare operație are acces la un număr mare de biți; o singură perturbare într-un bit se propagă rapid prin întregul bloc. În practică, un algoritm

pe bloc intercalează funcții liniare și neliniare (XOR-uri, rotații circulare, permutări) pentru a maximiza difuzia.

La nivel fundamental există totuși un neajuns: toate blocurile sunt tratate identic. Dacă blocurile se criptează independent, apar tipare vizibile la nivel global, chiar dacă fiecare bloc, luat separat, este perfect „amestecat”. Acest mod de operare se numește *Electronic Code Book* (ECB), iar un exemplu celebru pentru criptarea în ECB este aceea a logo-ului *Linux*, unde silueta pinguinului se distinge clar, deși datele fiecărui bloc par aleatoare.

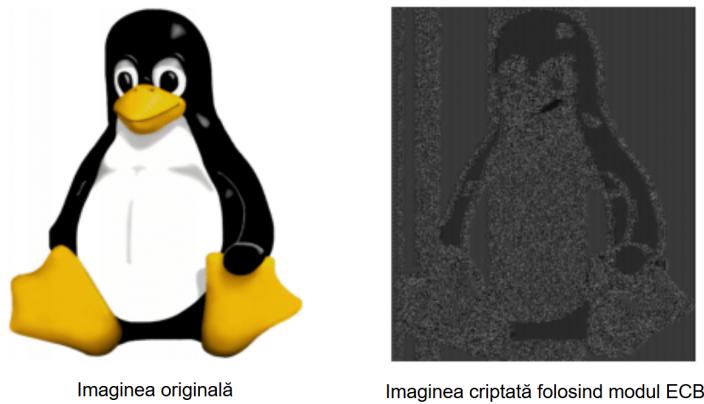


Figura 1.3.2: Encriptia in modul ECB a logo-ului Linux

Modul prin care acest neajuns este contracarat îl reprezintă implementarea a încă patru moduri de operare, aprobată și standardizată de National Institute of Standards and Technology (NIST) din SUA:

Cipher Block Chaining (CBC)

Fiecare bloc de text clar este XOR-ăt cu blocul cifrat precedent, apoi trecut prin cifrul pe bloc. Primul bloc folosește un *IV* (vector de inițializare) aleator și unic. Astfel, tiparele dintre blocuri dispar. **Limitare:** criptarea nu poate fi paralelizată, iar o eroare apărută într-un bloc afectează și decriptarea blocului următor.

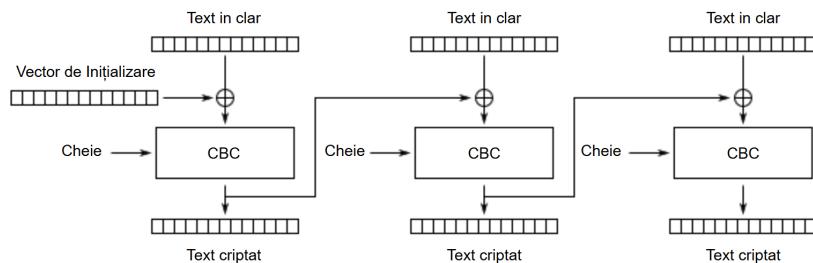


Figura 1.3.3: Schema criptării modului CBC

Cipher Feedback (CFB)

Se cifrează blocul anterior (sau IV-ul), iar rezultatul se XOR-ează cu textul clar pentru a produce textul cifrat; blocul cifrat devine feedback pentru pasul următor. Poate lucra pe segmente mai mici decât dimensiunea blocului și nu necesită *padding*. **Limitare:** procesarea rămâne serială, iar o eroare de transmisie corupe segmentul curent și pe următorul.

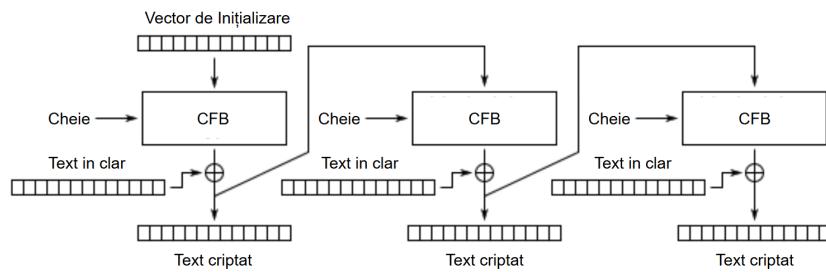


Figura 1.3.4: Schema criptării modului CFB

Output Feedback (OFB)

IV-ul trece repetat prin cifrul pe bloc, iar ieșirile successive formează un *keystream* ce se XOR-ează cu textul clar—practic transformând cifrul pe bloc într-un cifru pe flux. Erorile de comunicație nu se propagă, iar criptarea și decriptarea sunt identice. **Critic:** reutilizarea același IV sub aceeași cheie compromite securitatea (keystream reutilizat).

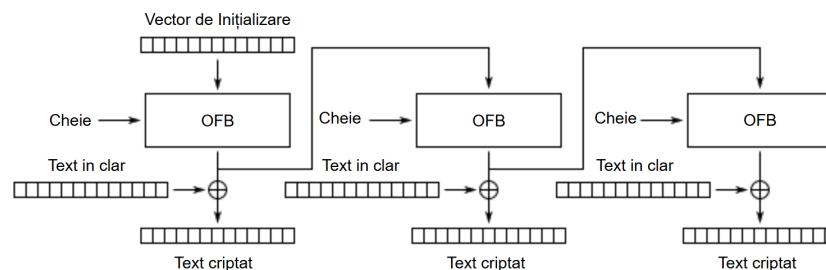


Figura 1.3.5: Schema criptării modului OFB

Counter (CTR)

Se cifrează valori monotone de tip *nonce + counter*; blocurile rezultate formează keystream-ul care se XOR-ează cu textul clar. Avantaj major: operațiile pe blocuri diferite sunt independente, deci criptarea și decriptarea sunt **complet paralelizabile** și foarte rapide (se pot pre-calcula). **Atenție:** combinația *nonce + counter* nu trebuie repetată niciodată sub aceeași cheie, altfel apare același risc ca la OFB.

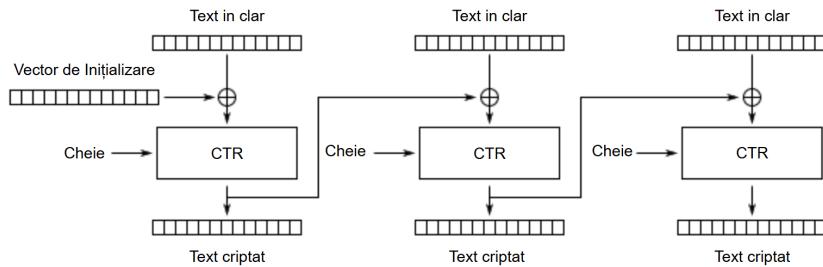


Figura 1.3.6: Schema criptării modului CTR

Acstea moduri sunt folosite *în defavoarea* modului ECB, deoarece produc rezultate pseudo-aleatoare chiar și pentru date de intrare de dimensiuni mari (formate din multe blocuri), aşa cum se poate observa pe aceeași poză: structura globală dispare complet, rămânând doar zgomot vizual.

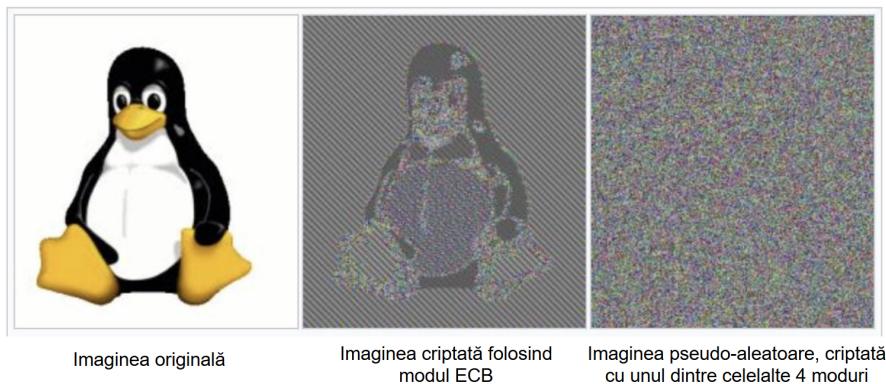


Figura 1.3.7: Criptarea în modul ECB a logo-ului Linux

1.4. Fundamentele și obiectivele criptografiei moderne

Obiective centrale

Criptografia urmărește să garanteze *confidențialitatea* mesajelor, *integritatea* și *autenticitatea* lor, precum și *non-repudierea* operațiunilor digitale.

1. **Confidențialitatea** – ascunderea conținutului față de persoane neautorizate.
2. **Integritatea** – detectarea oricărei modificări nepermise a datelor.
3. **Autenticitatea** – confirmarea faptului că mesajul provine de la entitatea declarată.
4. **Non-repudierea** – împiedicarea autorului unui mesaj valid să își poată nega ulterior acțiunea.

Metode pentru atingerea acestor obiective

Confidențialitatea se obține prin algoritmi de criptare simetrică (AES, ChaCha20) sau asimetrică (RSA, Curve25519), aşa cum urmează să fie detaliat în capitolul urmator. În mare, funcția lor este să transforme textul în clar în text cifrat folosind chei secrete, astfel încât un adversar să nu poată recupera conținutul fără cheie. Eficiența lor se bazează pe conceptele *confuzie* și *difuzie*, precum și pe moduri de operare sigure.

Integritatea este asigurată de funcții hash criptografice (SHA-3, BLAKE3) și de coduri de autentificare a mesajelor (HMAC, Poly1305). O singură modificare de bit în mesaj produce un hash complet diferit, permitând detectia imediată a oricărei alterări.

Autenticitatea se bazează pe semnăturile digitale (ECDSA, EdDSA) și pe MAC-uri. Emițătorul semnează datele cu cheia privată; destinatarul verifică semnatura cu cheia publică (sau cu cheia secretă partajată), confirmând identitatea sursei.

Non-repudierea este garantată de aceleași mecanisme de semnătură: deoarece numai deținătorul cheii private poate genera o semnătură validă, acesta nu poate contesta ulterior fără a se contrazice.

În practică, aceste primitive sunt implementate în protocoale precum *TLS*, *Signal* sau *WireGuard*, care gestionează distribuția și rotația cheilor, generarea numerelor aleatoare și protecția împotriva canalelor laterale, păstrând toate cele patru obiective pe tot parcursul ciclului de viață al datelor.

2. Stadiul actual al criptografiei (SotA)

Criptografia modernă este una dintre cele mai bine dezvoltate ramuri ale matematicii, datorită nevoii reale pe care o are societatea contemporană de mijloace sigure de protecție a informației. Tocmai de aceea, o analiză completă a *state-of-the-art-ului* în criptografie ar depăși cu mult scopul acestei lucrări. În continuare ne vom concentra, în principal, pe apariția și evoluția **cifrurilor pe bloc**, pe utilizarea lor în protocoalele criptografice larg răspândite astăzi, pe provocările cărora trebuie să le facă față (variind de la diverse tipuri de atacuri, până la limitările lor intrinseci), și pe direcțiile de dezvoltare ce conturează viitorul acestui domeniu.

2.1. Evoluția cifrurilor pe bloc clasice și moderne

Istoria cifrurilor pe bloc începe, ca multe alte povești din informatică, în Statele Unite ale anilor '70. Înainte, însă, de a putea discuta despre realizările pionierilor inginerilor de la *IBM*, trebuie să introducem scheletul teoretic pe care s-au clădit aceste realizări, începând cu *rețeaua Feistel*.

Rețele Feistel (Luby–Rackoff)

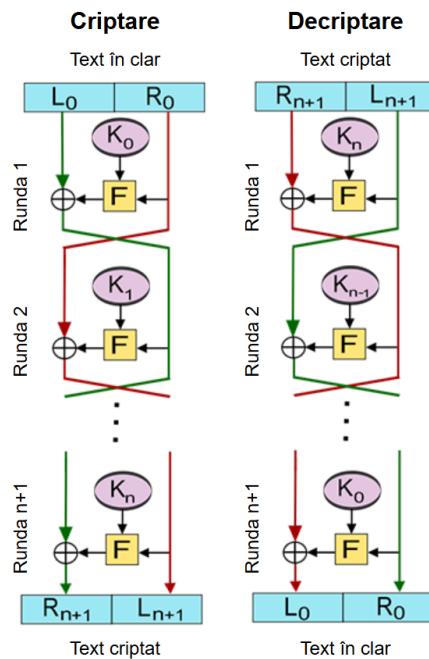


Figura 2.1.1: Rețea Feistel cu $n+1$ runde

Horst Feistel a fost un criptograf *germano-american* care a lucrat ca inginer la IBM și este principalul nume asociat cu apariția primului cifru pe bloc, *Lucifer*. Acest algoritm se bazează pe

o structură numită **rețea Feistel**, ilustrată în figura de mai sus. Ideea de bază este introducerea unei *funcții de rundă F* ce primește două intrări (o cheie de criptare și un bloc de date de dimensiune fixă), și care produce o ieșire de aceeași dimensiune.

În forma cea mai simplă, blocul de date este împărțit în două jumătăți L_0 (stânga) și R_0 (dreapta), iar prima apelare a lui F operează doar pe R_0 . La fiecare rundă i se calculează $L_i = R_{i-1}$, iar mai apoi $R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$, după care jumătățile se inversează și procesul se repetă de un număr fix de ori. Aceeași schemă, parcursă în sens invers, realizează decriptarea. Dacă numărul de runde este suficient, rezultatul devine practic pseudo-aleator.

O majoritate covârșitoare a cifrurilor pe bloc vechi (inclusiv *DES*, *3DES*), dar și cîteva cifruri moderne (ex.: *Camellia*) folosesc încă variații ale acestui tip de rețea, fapt ce stă drept dovadă pentru că de inovatori au fost cercetătorii implicați în conceperea ei.

Rețele Substituție–Permutare (SP)

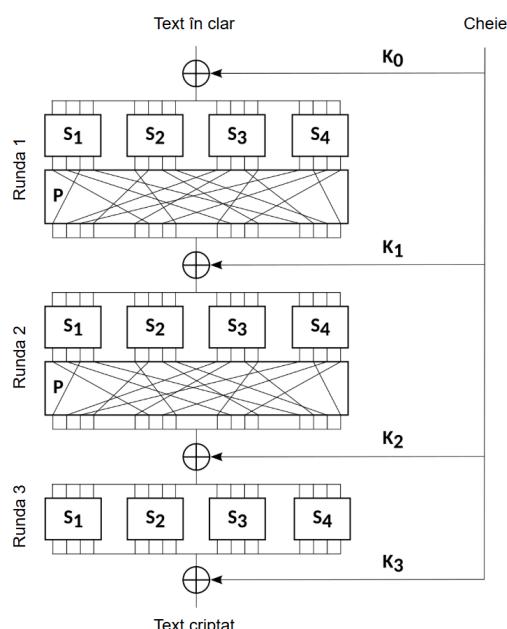


Figura 2.1.2: Rețea Substituție–Permutare (SP)

Rețeaua *Substituție–Permutare* este a două mare familie de structuri criptografice care a remodelat cifrurile pe bloc după anii '90. Conceptul, schițat încă de Claude Shannon, este surprinzător de simplu: se aplică mai întâi o **substituție** neliniară, cum este un *Substitution Box (S-Box)* bijectiv, pe fragmente mici din bloc, apoi se **permute** biții rezultatului printr-o transformare liniară fixă. Repetarea acestei secvențe, cu o subcheie diferită la fiecare pas, răspândește rapid confuzia și difuzia asupra întregului bloc.

Spre deosebire de rețelele Feistel, invertibilitatea nu se obține doar datorită proprietăților XOR ce

se fac între jumătăți, aşa că fiecare etapă a unei runde SP trebuie să fie reversibilă în sine. Costul suplimentar este răsplătit printr-un avantaj major de paralelism: toate S-boxurile pot fi evaluate simultan, iar stratul de permutare se reduce adesea la rotații sau mixări de coloane eficiente pe hardware.

Această abordare a adus un salt notabil de performanță și securitate, exemplificat mai întâi de *Advanced Encryption Standard (AES)* și continuat de cifruri „lightweight” precum *PRESENT* și *GIFT*. Astăzi, structurile SP domină standardele moderne tocmai pentru echilibrul reușit dintre rigurozitate teoretică și eficiență practică.

Data Encryption Standard (DES) și Triple DES (3DES)

Acum că am terminat cu stabilirea fundamentelor teoretice, putem aborda primele cifruri pe bloc răspândite la scară largă. **Data Encryption Standard (DES)** a fost publicat pentru prima dată în 1975, iar doi ani mai târziu, în 1977, a devenit oficial *Federal Information Processing Standard (FIPS)* al Statelor Unite, după revizuiri operate de *National Security Agency (NSA)*. Cifrul se inspiră direct din **Lucifer** al lui Feistel și utilizează o rețea Feistel *echilibrată*, adică textul de intrare este împărțit în două jumătăți egale, prelucrate alternativ pe parcursul celor 16 runde de criptare.

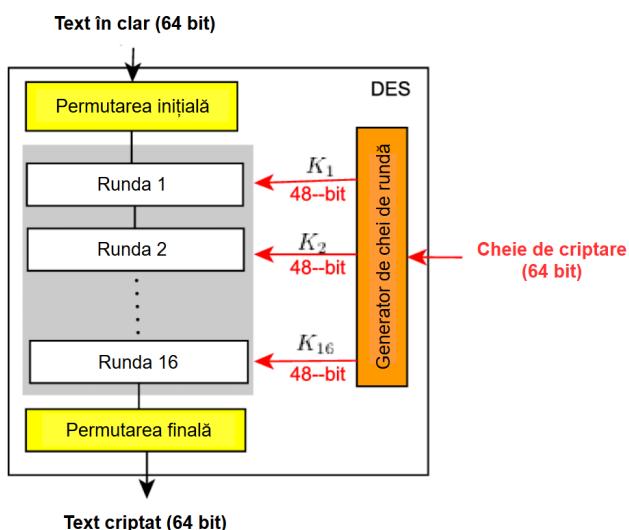


Figura 2.1.3: Structura Cifrului DES

DES este considerat astăzi *obsolete* din două motive majore: lungimea cheii de doar 56 de biți poate fi spartă prin forță brută în ore folosind hardware comercial, iar mărimea blocului de 64 de biți expune protocolul la atacul *Sweet32*, care exploatează repetarea blocurilor în fluxuri lungi. Pentru a prelungi viața standardului, s-a introdus **Triple DES (3DES)**, care aplică DES de trei ori (secvența fiind: criptare-decriptare-criptare), astfel obținându-se extinderea securității teoretice la 112/168 biți.

Totuși, **3DES** nu este o soluție definitivă: deși extinde spațiul cheilor, moștenește blocul mic de 64 bit, rămânând expus la atacul *Sweet32*. De asemenea, aplicarea triplă îi triplează și costul de calcul,

făcându-l considerabil mai lent decât cifruri moderne cu suport hardware dedicat. În plus, 3DES este încă vulnerabil la strategii *meet-in-the-middle*, care reduc securitatea efectivă la circa 112 bit, și impune grija gestionării modurilor de cheie (*keying options*) pentru a evita eșecuri criptografice subtile. Ca urmare a acestor limitări, organismul NIST a marcat 3DES drept „legacy” și a planificat retragerea sa completă din noile standardizări după anul 2024, iar protocoale contemporane precum *TLS 1.3* au eliminat deja suportul explicit pentru el.

Advanced Encryption Standard (AES)

De această dată, răspunsurile la problemele globale de criptare aveau să vină din **Europa**. Suntem în **1997**, iar *NIST*, presat de necesitatea unui nou algoritm capabil să țină pasul cu progresele în criptanaliză, lansează un concurs pentru desemnarea viitorului „*Advanced Encryption Standard*”. După mai bine de trei ani de selecție, algoritmul eponim al belgienilor *Joan Daemen* și *Vincent Rijmen* — **Rijndael** — este ales câștigător. Soluția lor, bazată pe o structură simplă, va deveni noul standard federal de securitate al Statelor Unite și își păstrează acest statut până în prezent.

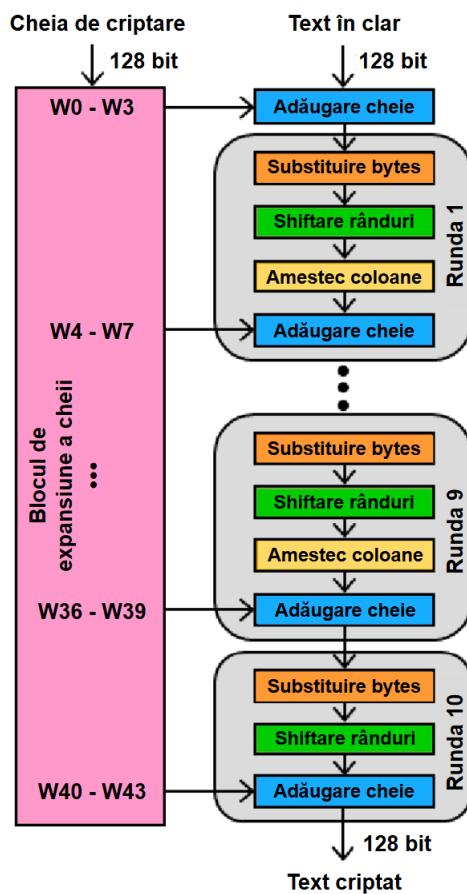


Figura 2.1.4: Structura Cifrului AES, cu cheie de 128 bit

AES (Rijndael) operează pe un *stat* de 128 biți, reprezentat ca o matrice 4×4 de octeți. Fiecare rundă (10, 12 sau 14, în funcție de lungimea cheii) aplică patru transformări reversibile:

1. **SubBytes** – fiecare octet trece printr-un S-box neliniar bijectiv (confusion).

2. **ShiftRows** – permutare circulară a rândurilor matricei.
3. **MixColumns** – transformare liniară în $GF(2^8)$ care amestecă octetii fiecărei coloane (difusii).
4. **AddRoundKey** – XOR cu subcheia rundei, derivată prin *Key Schedule*.

La ultima rundă se omite pasul *MixColumns*; la decriptare se folosesc inversele acestor operații. Calitățile principale pe care le are AES, în principal prin comparație cu DES:

- **Spațiu al cheii mult mai mare** – 128/192/256 biți versus 56 biți; forța brută devine impracticabilă chiar și cu resurse guvernamentale.
- **Bloc dublu ca dimensiune** – 128 biți în loc de 64 biți; previne atacuri de tip *Sweet32* și reduce coliziunile în moduri de operare.
- **Eficiență software/hardware** – instrucțiunile dedicate AES-NI, ARMv8 CryptExt și micro-cod optimizat îl fac mai rapid decât DES chiar și într-un singur pas.

AES este foarte răspândit în Statele Unite și Europa, o parte considerabilă din criptografia ce necesită cifruri bloc fiind facută cu diverse versiuni de AES, cum ar fi:

- **AES-128/256-GCM** – standard de facto în TLS 1.3, QUIC, OpenVPN, IPsec (standardele curente pentru securizarea conexiunilor pe internet); disponibil în toate bibliotecile majore (OpenSSL, libsodium).
- **AES-256-XTS** – criptarea discurilor (BitLocker, FileVault, dm-crypt/LUKS2, hardware SED).
- **AES-128-CCM** – WPA2/WPA3 (Wi-Fi) și Bluetooth LE Secure Connections.
- **AES-CTR & AES-CBC** – încă prezente în SSH și unele implementări IPsec/legacy; recomandat doar împreună cu un MAC (HMAC-SHA2) sau într-o variantă AEAD.
- **AES-KW (Key Wrap)** – protejarea cheilor în TPM 2.0, JWE/JWK (JSON Web Encryption), HSM-uri FIPS.

Alte standarde globale

În afara tandemului DES-AES, peisajul criptografiei simetrice include câteva cifruri pe bloc remarcabile, fiecare răspunzând unor nevoi regionale sau tehnice particulare. **Camellia**, creat de NTT și Mitsubishi (Japonia), oferă o securitate comparabilă cu AES și este integrat în OpenSSL, IPsec și în unele implementări SSH, fiind preferat mai ales în Asia și în sectorul financiar. Coreea de Sud promovează propria alternativă, **ARIA**, standardizată prin *KS X 1213* și folosită în infrastructura guvernamentală locală.

Din finala concursului AES au rămas populare **Twofish** și **Serpent**, preferate în proiecte open-source precum *VeraCrypt* și *KeePass*, grație faptului că nu sunt acoperite de patente. Pentru aplicații

cu resurse reduse (IoT), cifrurile „lightweight” **PRESENT**, **GIFT** și **Xoodyak** câștigă teren, fiind candidați în programul *NIST LWC (Lightweight Cryptography)*. În China continentală, norma oficială este **SM4**, obligatorie în echipamente de rețea și în plățile *UnionPay*, cifru despre care, bineînțeles, vom vorbi mai multe în capitolele urmatoare.

Deși niciunul nu a atins omniprezența lui AES, aceste algoritmi completează peisajul prin conformitate regională, optimizări hardware dedicate sau amprentă redusă de memorie.

2.2. Atacuri împotriva cifrurilor pe bloc – de la tehnici clasice la provocările actuale

Căutarea exhaustivă a cheii (brute-force)

Definiție. Atacatorul încearcă sistematic toate valorile de cheie până găsește cea care produce textul clar corect. Efortul crește exponential cu lungimea cheii și nu exploatează nici o slăbiciune structurală a cifrului. Deși este cel mai vechi și nesofisticat tip de atac, creșterile recente exponențiale ale puterii de procesare și dorința continuă de expansiune a centrelor de date (în special datorate boom-ului Inteligenței Artificiale) fac ca acest tip de atac să ramană relevant în continuare când se pune problema testării rezistenței algoritmilor.

Nivelul de amenințare, pentru diferite cifruri:

- DES (56 biți) – *critic*; spargerea este posibilă în ore cu hardware comercial.
- 3DES (112 biți efectivi) – *fezabil pe termen mediu* pentru actori statali; uzul practic este restricționat.
- AES-128/256, Camellia, Twofish, Serpent – *neglijabil* cu resurse actuale (inclusiv centre de date).

Contramăsură: Lungimi de cheie ≥ 128 biți și politici de rotație periodică (crypto-agility).

Criptanaliza diferențială

Definiție. Analizează modul în care mici diferențe în textul clar se propagă prin runde, căutând modele care scad entropia ieșirii. Este, de asemenea, unul dintre cele mai vechi tipuri de atacuri, dar poate fi contracararat cu succes crescând cât mai mult difuzia cifrului.

Nivelul de amenințare, pentru diferite cifruri:

- DES – *mediu → înalt*; atacul teoretic a influențat proiectarea S-box-urilor.

- AES și alte cifruri moderne – *foarte scăzut*; numărul de runde și S-box-urile bijective elimină căile diferențiale eficiente.

Contramăsură: Creșterea numărului de runde și S-box-uri cu proprietăți non-liniare puternice (design „wide-trail” la AES).

Criptanaliza liniară

Definiție. Caută relații liniare aproximative între biții textului clar, cheii și textului cifrat, reducând numărul de chei candidate.

Nivelul de amenințare, pentru diferite cifruri:

- DES – *mediu*; reușită demonstrată pentru sub-versiuni cu runde reduse.
- AES, Camellia, Serpent, Twofish – *neglijabil*; cifrele actuale necesită date astronomice pentru un avantaj practic.

Contramăsură: S-box-uri cu bias liniar minim și difuzie pe toate bit-plane-urile.

Atacul *meet-in-the-middle*

Definiție. Profită de criptare multiplă (ex. 3DES) calculând în paralel toate ieșirile intermediere „de la început” și „de la sfârșit” până când se întâlnesc, reducând semnificativ complexitatea.

Nivelul de amenințare, pentru diferite cifruri:

- 3DES – *relevant*; securitatea efectivă coboară la ≈ 112 biți.
- AES (o singură criptare) – *nu se aplică*.
- Alte cifruri moderne – idem, atâtă timp cât nu se implementează în mod compus.

Contramăsură: Renunțarea la multiple-encryption în favoarea unui cifru cu cheie lungă nativă.

Atacuri pe canale laterale (timing, cache, power)

Definiție. Exploatează informații fizice *colaterale* (timp de execuție, consum electric, zgomot electromagnetic) pentru a deduce biți de cheie, fără a sparge algoritmul matematic. Sunt o tipologie de atacuri moderne, și sunt studiate din ce în ce mai mult pentru a le putea preveni cu succes.

Nivelul de amenințare, pentru diferite cifruri:

- DES, 3DES (software table-based) – *înalt*; implementări vechi nu sunt constant-time.
- AES fără instrucțiuni dedicate – *mediu*; tabelele S-box pot fi monitorizate prin cache.
- AES cu AES-NI, Camellia cu bitslice – *scăzut* când sunt scrise constant-time.

Contramăsură: Instrucțiuni hardware (AES-NI, ARM CryptExt), algoritmi bitslice, randomizare și *masking* la nivel hardware/firmware.

Injecții de fault (DFA)

Definiție. Atacatorul introduce erori controlate (temperatură, radiații, glitch de voltaj) în timpul execuției, apoi compară ieșirea greșită cu cea corectă pentru a extrage subchei. La fel ca atacurile pe canale laterale, DFA-urile sunt atacuri moderne, practic indispensabile în criptanalizele pentru smart-cards și pentru criptarea memoriilor.

Nivelul de amenințare, pentru diferite cifruri:

- DES, 3DES – *mediu*; afectează runde finale relativ ușor.
- AES – *mediu*; atacuri practice pe runde finale în smart-card-uri au fost publicate, dar necesită acces fizic și expertiză.
- Camellia, Serpent, Twofish – similar cu AES; depinde de protecțiile implementate.

Contramăsură: Detectarea dublă (redundanță), verificare MAC intern, cifre rezistente la fault (CLEFIA) și protecții fizice pe chip.

Prin combinarea cheilor suficient de lungi, a proiectării moderne a cifrurilor și a implementărilor rezistente la canale laterale, majoritatea acestor atacuri devin impracticabile în scenarii reale, însă rămân relevante în evaluările de securitate și în standardele FIPS/NIST.

2.3. Tendințe și direcții emergente în criptografie

Mulți dintre noi probabil am auzit cel puțin o dată în ultimul timp despre „quantum computing”. Chiar cu puțin timp în urma scierii acestei lucrări, Microsoft tocmai a anunțat primul prototip de cip cuantic, numit Majorana 1, care, chiar după spusele lor: "Va fi mai puternic decât toate computerele din lume care ar opera simultan." Cu siguranță că este posibil ca aceste afirmații să descrie un viitor destul de îndepărtat, însă tehnologia cu siguranță există. Ce este, deci, un calculator cuantic?

Un calculator cuantic folosește *qubiți* — elemente capabile să se afle simultan în multiple stări prin suprapunere și să fie corelate prin entanglement — pentru a executa operații care, pe hardware clasic, ar necesita explorarea secvențială a unui spațiu vast de posibilități. Această paralelism intrinsec permite algoritmilor precum *Shor* (factorizare) sau *Grover* (căutare cuadratică accelerată) să depășească fundamental performanța calculului tradițional în anumite sarcini specifice.

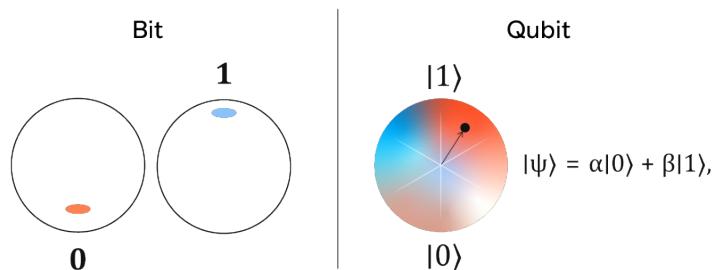


Figura 2.3.1: Bit vs. Qubit

Bineînțeles ca, având tot această cursă a înarmărilor constant prezentă, criptografia evoluează rapid sub presiunea calculului cuantic, a dispozitivelor cu resurse limitate și a cerințelor tot mai ridicate de confidențialitate. În cele ce urmează, vom prezenta foarte sumarizat câteva din actualele direcții de cercetare și standardizare din criptografie:

- **Criptografia post-cuantică (PQC).** În iulie 2022 NIST a selectat primii algoritmi finaliști (*CRYSTALS-Kyber*, *CRYSTALS-Dilithium*, *Falcon*, *SPHINCS⁺*), iar standardele oficiale vor fi publicate în 2025. În paralel, simetricele adoptă chei de ≥ 256 biți și *crypto-agility* pentru migrare rapidă.
- **Lightweight Crypto & IoT.** Programul *NIST LWC* (2023–2024) a desemnat Xoodyak, Ascon și două familii SKINNY ca noi standarde AEAD pentru microcontrolere. Obiectivul: securitate AES-echivalentă sub 20 KB de flash / 2 KB RAM.
- **Cifruri ajustabile (tweakable) și tolerate la reutilizarea nonce-ului.** Cifruri precum *SKINNY-T*, *Deoxys-II*, *HCTR2* integrează un „tweak” public, oferind rezistență la reutilizarea nonce-ului și facilitând criptarea discurilor, snapshot-urilor sau bazelor de date.
- **Zero-Knowledge și Multi-Party Computation.** ZK-SNARK/zk-STARK, *Halo2* și sisteme MPC precum *FROST* permit validarea sau calculul distribuit pe date confidențiale fără a le revela, fundament pentru roll-up-uri blockchain și vot electronic.
- **Criptografie complet omomorfă (FHE).** Schemele *CKKS*, *BFV*, *TFHE* au trecut de la demonstrații de principiu la prototipuri comerciale (Microsoft SEAL, IBM HELib), permitând procesarea sigură „în cloud” a datelor medicale sau financiare.
- **Calcul confidențial și enclave de execuție de încredere (TEE)** Extensiile hardware (*Intel TDX*, *AMD SEV-SNP*, *ARM CCA*) cripteză automat memoria mașinii virtuale; standardul de facto pentru chei rămâne AES-XTS însotit de mecanisme de atestare derivând din TPM-2.0.

În ansamblu, direcția actuală combină trecerea la algoritmi *post-quantum*, miniaturizarea pentru IoT și protocoale avansate de confidențialitate, toate sprijinite de hardware nou care protejează cheile încă din siliciu.

3. Algoritmul chinezesc de criptare bloc SM4

3.1. Istorici, origini și statutul de standard

Așa cum se întâmplă în multe domenii, standardele și practicile **euro-americane** nu se aliniază întotdeauna cu cele **est-asiatice**, aceste trei regiuni fiind, fiecare la rândul ei, centre majore de influență atât din perspectivă istorică, cât și în privința progresului științific modern. În criptografie am menționat deja variantele japoneze (algoritmul *Camellia*, dezvoltat de *Mitsubishi Electric* și *NTT*) și pe cel coreean (*ARIA* - Advanced Reliable Individual Algorithm), însă am detaliat prea puțin despre standardele adoptate de cea mai mare economie a emisferei estice.

Algoritmul **ShāngMì 4** (商密4, în limba chineză standard — mandarină), prescurtat *SM4*, este standardul și totodată cel mai răspândit cifru pe bloc de pe teritoriul Republicii Populare Chineze. Tradus mot-a-mot, numele suitei *Shāng Mì* înseamnă „secret comercial”, iar *SM4* face parte dintr-un pachet mai larg, denumit frecvent **ZUC/SMx**, format din încă cinci algoritmi:

- **SM1** — cifru pe flux cu cheie de 128 biți, utilizat în echipamente guvernamentale și în protocolul **WAPI**.
- **SM2** — schemă pe curbe eliptice (EC Diffie–Hellman + semnătură) peste o curba națională de 256 biți; înlocuiește RSA/ECC occidentale în TLS și infrastructura de semnături digitale.
- **SM3** — funcție hash de 256 biți, impusă în domeniul finanțier și în platformele de e-guvernare din China.
- **SM9** — criptografie bazată pe identitate (IBE) și semnătură cu pairing-uri bilaterale, destinată distribuției cheilor în aplicații mobile, IoT și sisteme de autentificare unde un PKI clasic ar fi costisitor.
- **ZUC** — cifru pe flux de generație nouă, folosit ca algoritm de confidențialitate și integritate în rețelele 4G LTE și 5G NR (specificațiile 128-EEA3 și 128-EIA3 din 3GPP).

Prima apariție publică a cifrului a avut loc în 2006, fiind dezvoltată în mare parte de Lü Shuwang (呂述望 în chineză standard - mandarină). Cifrul este menționat într-un document intern al Ministerului Industriei și Tehnologiei Informației din China, sub numele *SMS4*. Algoritmul fusese conceput inițial pentru a proteja protocolul național Wi-Fi **WAPI** (*WLAN Authentication and Privacy Infrastructure*), menținând compatibilitatea cu hardware modest, fără a se baza pe patente occidentale.

În 2012, după declasificarea oficială, cifrul este redenumit **SM4** și devine standard național prin *GM/T 0002–2012*. O revizie mai bogată în detalii criptanalitice apare în *GB/T 32907–2016*, iar din

2020 algoritmul este obligatoriu pentru tranzacțiile *UnionPay*, rețelele 5G locale și aplicațiile cloud certificate de administrația chineză *CAC (Cyberspace Administration China)*.

Cât despre disponibilitate, putem menționa că implementări open-source au fost integrate în ramurile *GmSSL* și *BoringSSL-GM*, iar *OpenSSL 3.x* oferă un provider dedicat *legacy_sm4*. Accelerarea hardware este prezentă pe procesoarele *Loongson 3A5000*, *Zhaoxin KX*, unele SoC *Kirin/HiSilicon* și pe FPGA-uri autohtone. De asemenea, o descriere completă a algoritmului este disponibilă online sub forma unui *Internet Draft*, cu tot cu o implementare de referință, scrisă în ANSI C.

De ce nu apare în listele NIST/FIPS?

În pofida faptului că **SM4** este considerat suficient de robust de autoritățile chineze pentru a proteja comunicațiile celei mai populate țări din lume, algoritmul nu figurează pe listele de standarde acceptate ale *National Institute of Standards and Technology* (NIST) din Statele Unite, astfel că rămâne aproape necunoscut în afara Chinei. A existat și o altă încercare importantă: varianta chineză *WAPI* (WLAN Authentication and Privacy Infrastructure) folosește SMS4/SM4, și a fost propusă în 2006 pentru „*fast-track*” ca amendament la standardul internațional ISO/IEC 8802-11 (echivalentul IEEE 802.11). Propunerea a fost însă respinsă după opoziția delegațiilor SUA și UE, care au invocat lipsa de transparență și suprapunerea cu soluția deja adoptată în Occident, *IEEE 802.11i* (bazată pe AES-CCMP).

În consecință, SM4 nu a fost preluat sub nicio formă de organisme de standardizare occidentale, chiar dacă, în practică, SM4 rămâne un standard *de facto* la nivel național chinez, folosit de furnizorii care trebuie să îndeplinească cerințele reglementărilor locale privind criptografia comercială.

3.2. Structură și funcționare

SM4 este un **cifru simetric pe bloc**, care procesează date în blocuri de 128 biți, și utilizează o cheie tot de 128 biți. Algoritmul execută **32 de runde** consecutive, fiecare bazată pe o *rețea Feistel neechilibrată*: la fiecare pas este actualizat doar unul dintre cele patru cuvinte de 32 biți ale stării interne, în timp ce celelalte trei sunt propagate nemodificate. Transformarea de rundă combină un S-box neliniar, rotații bit-wise și operații XOR, iar subcheile rundei sunt generate din cheia master printr-un *key-schedule* care folosește aceeași funcție neliniară. După parcurgerea celor 32 de runde, ordinea cuvintelor este inversată pentru a produce textul cifrat final.

Rețeaua Feistel neechilibrată utilizată

Așa cum am menționat anterior, SM4 lucrează pe un bloc de **128 biți** împărțit în patru cuvinte de 32 biți, notate X_0, X_1, X_2, X_3 . Spre deosebire de un Feistel echilibrat, numai *un* cuvânt se actualizează la fiecare rundă, ceea ce face schema *unbalanced*.

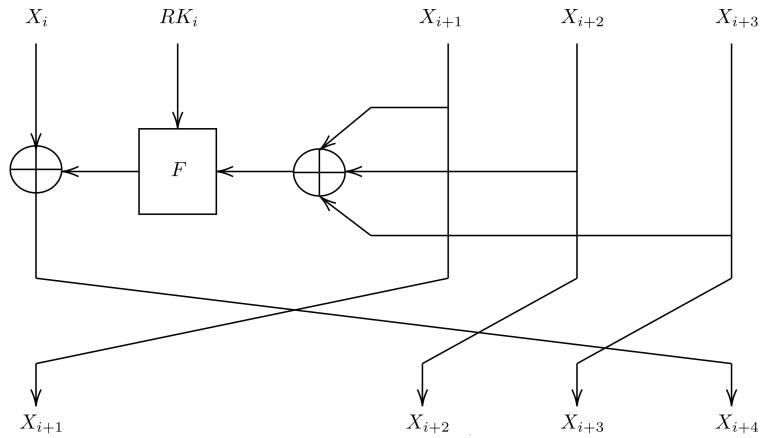


Figura 3.2.1: Structura rețelei Feistel neechilibrante a algoritmului SM4

Funcția de rundă (F function)

Funcția de rundă, denumită și *funcția F*, primește la intrare cele patru părți ale blocului, noteate X_0, X_1, X_2, X_3 (câte 32 de biți fiecare), împreună cu *cheia de rundă (round key)*. La ieșire, funcția returnează noul cuvânt X_4 , care va deveni X_0 la următoarea apelare a funcției F.

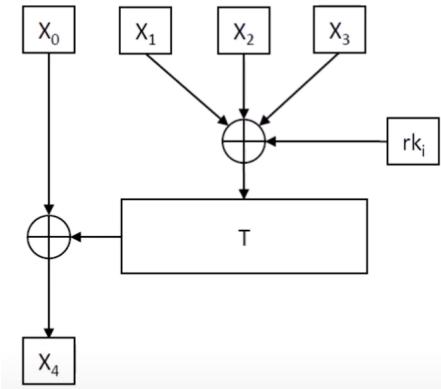


Figura 3.2.2: Funcția de rundă a algoritmului SM4

Deci, scrisă sub forma unei formule, avem:

$$F(X_i, X_{i+1}, X_{i+2}, X_{i+3}, rk_i) = X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i), \quad i=0..31$$

unde rk_i este subcheia rundei. Se observă folosirea unei funcții T, pe care o vom defini acum.

Funcția T și funcția liniară L

Acste două funcții realizează substituția liniară a cifrului. Ele sunt esențiale pentru sporirea *difuziei* cifrului, iar formulele lor sunt următoarele:

$$T(X) = L(\tau(X))$$

$$L(B) = B \oplus (B \lll 2) \oplus (B \lll 10) \oplus (B \lll 18) \oplus (B \lll 24)$$

Shiftarea este circulară (lossless), adică biții care dau overflow trec la coadă, nu sunt șterși. Putem vedea și un exemplu efectiv:

X	00111110 00010111 10111010 10010110
XOR X <<< 2	11111000 01011110 11101010 01011000
XOR X <<< 10	01011110 11101010 01011000 11111000
XOR X <<< 18	11101010 01011000 11111000 01011110
XOR X <<< 24	10010110 00111110 00010111 10111010
<hr/>	
	11100100 11000101 11100111 11010010

Figura 3.2.3: Exemplu rezolvat pentru aplicarea L-function în SM4

Funcția τ (tau) și Substitution-Box-ul (S-Box)

Acestea sunt funcțiile care adaugă *confuzie* cîfrului, prin caracterul lor neliniar. În primul rand, funcția τ :

$$\tau(X) = \text{SBox}(X) = S(x_0) \parallel S(x_1) \parallel S(x_2) \parallel S(x_3), \quad \text{unde } X = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \text{ (8 biți fiecare).}$$

Această funcție concatenează cei patru bytes obținuți după trecerea fiecărui din cei patru bytes ai intrării X prin procedeul de substituție după regula definită de S-box. Pentru SM4, S-box-ul arată așa:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	D6	90	E9	FE	CC	E1	3D	B7	16	B6	14	C2	28	FB	2C	05
1	2B	67	9A	76	2A	BE	04	C3	AA	44	13	26	49	86	06	99
2	9C	42	50	F4	91	EF	98	7A	33	54	0B	43	ED	CF	AC	62
3	E4	B3	1C	A9	C9	08	E8	95	80	DF	94	FA	75	8F	3F	A6
4	47	07	A7	FC	F3	73	17	BA	83	59	3C	19	E6	85	4F	A8
5	68	6B	81	B2	71	64	DA	8B	F8	EB	0F	4B	70	56	9D	35
6	1E	24	0E	5E	63	58	D1	A2	25	22	7C	3B	01	21	78	87
7	D4	00	46	57	9F	D3	27	52	4C	36	02	E7	A0	C4	C8	9E
8	EA	BF	8A	D2	40	C7	38	B5	A3	F7	F2	CE	F9	61	15	A1
9	E0	AE	5D	A4	9B	34	1A	55	AD	93	32	30	F5	8C	B1	E3
A	1D	F6	E2	2E	82	66	CA	60	C0	29	23	AB	0D	53	4E	6F
B	D5	DB	37	45	DE	FD	8E	2F	03	FF	6A	72	6D	6C	5B	51
C	8D	1B	AF	92	BB	DD	BC	7F	11	D9	5C	41	1F	10	5A	D8
D	0A	C1	31	88	A5	CD	7B	BD	2D	74	D0	12	B8	E5	B4	B0
E	89	69	97	4A	0C	96	77	7E	65	B9	F1	09	C5	6E	C6	84
F	18	F0	7D	EC	3A	DC	4D	20	79	EE	5F	3E	D7	CB	39	48

Figura 3.2.4: Substitution-Box pentru algoritmul SM4

Algoritmul de expansiune a cheii

Acest algoritm este modalitatea prin care SM4 generează cheile de rundă. El primește la intrare doar *master_key-ul*, care are 128 de biți iar, cu ajutorul funcției T' (foarte similară cu funcția T , folosind același τ , dar o funcție liniara L diferită), scoate la ieșire cheia de rundă de 32 de biți.

Modalitatea de calcul este următoarea: folosind un vector intern k , și doi vectori constanți *FK* (*Family Key*) și *CK* (*Constant Key*), se aplică un algoritm recurrent ce face multiple operații de XOR, similar într-un fel cu algoritmul din funcția de rundă. Prima dată, se inițializează primele patru elemente ale vectorului K , folosindu-se de valorile Master Key (MK) și ale Family Key:

$$\mathbf{MK} = (MK_0, MK_1, MK_2, MK_3), \text{ fiecare câte } 32 \text{ de biți.}$$

$$K_0 = MK_0 \oplus FK_0, \quad FK_0 = A3B1BAC6$$

$$K_1 = MK_1 \oplus FK_1, \quad FK_1 = 56AA3350$$

$$K_2 = MK_2 \oplus FK_2, \quad FK_2 = 677D9197$$

$$K_3 = MK_3 \oplus FK_3, \quad FK_3 = B27022DC.$$

După asta, se foloseste următoarea funcție recurrentă de generare a cheilor:

$$\begin{cases} K_{i+4} = K_i \oplus T'(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i), \\ rk_i = K_{i+4} \end{cases} \quad i = 0, \dots, 31.$$

CK_i este o familie de chei constante, care se obțin prin formulele:

$$CK_i = [7(4i) \parallel 7(4i+1) \parallel 7(4i+2) \parallel 7(4i+3)]_{256}, \quad i = 0, \dots, 31,$$

$$\text{echivalent } CK_i = (0x00070E15 + 0x1C1C1C1C i) \bmod 2^{32}.$$

Astfel, familia de chei arată aşa:

$CK_0 = 00070E15$	$CK_{11} = 343B4249$	$CK_{22} = 686F767D$
$CK_1 = 1C232A31$	$CK_{12} = 50575E65$	$CK_{23} = 848B9299$
$CK_2 = 383F464D$	$CK_{13} = 6C737A81$	$CK_{24} = A0A7AEB5$
$CK_3 = 545B6269$	$CK_{14} = 888F969D$	$CK_{25} = BCC3CAD1$
$CK_4 = 70777E85$	$CK_{15} = A4ABB2B9$	$CK_{26} = D8DFE6ED$
$CK_5 = 8C939AA1$	$CK_{16} = C0C7CED5$	$CK_{27} = F4FB0209$
$CK_6 = A8AFB6BD$	$CK_{17} = DCE3EAF1$	$CK_{28} = 10171E25$
$CK_7 = C4CBD2D9$	$CK_{18} = F8FF060D$	$CK_{29} = 2C333A41$
$CK_8 = E0E7EEF5$	$CK_{19} = 141B2229$	$CK_{30} = 484F565D$
$CK_9 = FC030A11$	$CK_{20} = 30373E45$	$CK_{31} = 646B7279$
$CK_{10} = 181F262D$	$CK_{21} = 4C535A61$	

Aici, aşa cum am menţionat anterior, T' este foarte similar cu T , având aceeaşi substituţie neliniara, însă schimbând funcţia liniară. Avem, deci:

$$T'(X) = L'(\tau(X))$$

$$L'(B) = B \oplus (B \lll 13) \oplus (B \lll 23)$$

unde “ $\lll n$ ” este aceeaşi rotire ciclică la stânga cu n biţi menţionată anterior.

Ca o scurtă concluzie putem afirma că, prin design, SM4 se bazează pe repetarea controlată a unui *mic set de operaţii bit-oriented*, evitând atât tabele mari în memorie, cât şi multiplicări peste câmpuri finite. Această alegere îi permite acestuia să rămână un exemplu de proiect *hardware-friendly*, însă care are şi complexitatea criptografică suficientă cât să fie numit drept standard de criptare într-o țară ce are nevoie de ea la cel mai înalt nivel, acestea fiind semnele distinctive ale unui algoritm criptografic de înaltă calitate.

3.3. Analiză comparativă a SM4 cu alte cifruri bloc

Deși SM4 este conceput inițial pentru piața chineză, el concurează direct cu AES în sisteme *embedded*, IoT și aplicații software mobile. În tabelul de mai jos sunt sintetizate diferențele esențiale dintre 3DES, Camellia-128, AES-128, și SM4, din perspectiva rezistenței criptanalitice și a costului de implementare.

Caracteristică	3DES	Camellia-128	AES-128	SM4
Bloc / cheie [bit]	64 / 168	128 / 128	128 / 128	128 / 128
Runde	48 (3×16)	18	10	32
Structură internă	Feistel clasic	Feistel + SP	SP-network	Feistel neechil.
Rezistență brute-force	2^{112}	2^{128}	2^{128}	2^{128}
Diferențial ^b (runde)	16 / 16	11 / 18	9 / 10	31 / 32
Liniară ^c (runde)	16 / 16	11 / 18	8 / 10	28 / 32
Boomerang / Rectangle	–	10 r.	9 r.	24 r.
Related-key	–	14 r.	10 r.	22 r.
Biclique	–	$2^{126.1}$	$2^{126.1}$	$2^{125.1}$
Bloc mic („birthday”)	da	nu	nu	nu
Performanță SW ARM [cy/B]	160	40	28	33
Cost ASIC [kGE]	25	20	12	10
Risc SCA ^e	ridicat	mediu	scăzut	scăzut
Standardizare	NIST SP 800-67	ISO/IEC 18033-3	FIPS 197	GB/T 32907

Tabela 3.3.1: Comparaţie extinsă între AES-128, SM4, 3DES şi Camellia-128

Concluzii sustrase din comparația celor patru cifruri

- **3DES este vizibil cel mai slab.** Cheia efectivă este de 112 biți, blocul de 64 biți limitează volumul sigur de date, iar implementarea rulează de peste cinci ori mai lent (160 cicluri/byte vs 28, respectiv 33 cicluri/byte pentru AES-128, respectiv SM4) și ocupă de două ori mai multă logică pe ASIC față de celelalte opțiuni (25 kilo porti logice vs 12, respectiv 10 kilo porti logice pentru AES-128, respectiv SM4).
- **SM4 și AES sunt cele mai echilibrate.** Ambele folosesc bloc și cheie de 128 biți și nu au atacuri practice. SM4 cere mai puțină zonă hardware, în timp ce AES profită de instrucțiuni dedicate pe multe procesoare și rulează mai rapid în software, însă, aşa cum am vazut, valorile sunt comparabile.
- **Camellia se poziționează pe la mijloc.** Algoritmul japonez asigură același nivel teoretic de securitate, dar are o structură mai complexă, deci un cost hardware ușor mai mare și viteză software doar medie, comparativ cu AES sau SM4.
- **Atacurile diferențiale, liniare și biclique nu sunt critice.** Toate sparg doar porțiuni din runde complete; avantajul biclique (2^{126} operații) nu transformă niciun algoritm într-o țintă ușoară.
- **Eficiența depinde de context.** Pe microcontrolere fără accelerării, SM4 și Camellia sunt atractive datorită amprentei mici, însă pe sisteme cu AES-NI (Advanced Encryption Standard - New Instructions), AES rămâne cel mai rapid, deoarece aceste sisteme sunt optimizate special pentru AES, datorită răspândirii largi a acestuia; 3DES ar trebui păstrat doar pentru compatibilitate.
- **Standardizarea ghidează adoptia.** La finalul zilei, putem spune că avem de a face cu un concurs de popularitate: AES este standard global (FIPS, ISO), SM4 este obligatoriu în China și recunoscut internațional, dar nestandardizat în afara teritoriului chinezesc, Camellia este o alternativă ISO, iar 3DES se retrage treptat, datorită îngrijorărilor asupra rezultatelor slabe la teste de criptanalitice. (End-Of-Life 2030 la NIST).

4. Implementarea algoritmului SM4 în SystemVerilog

Cerința de a integra un algoritm criptografic într-un canal de comunicație se poate rezolva în diverse metode. Printre acestea putem enumera: software, microcontroler, GPU, ASIC sau FPGA. Când motivația proiectului este, ca în cazul nostru, testarea locală rapidă și personalizarea facilă, FPGA-ul (Field-Programmable Gate Array)iese în evidență printr-un set de avantaje clare, cum ar fi:

- **Performanță personalizabilă** – numărul de runde procesate pe fiecare tact poate fi ales prin simplă adăugare de etape de *pipeline*, până la saturarea resurselor (LUT-uri și flip-flop-uri).
- **Reconfigurabilitate facilă** – orice revizie a algoritmului, schimbare de cheie sau nou mod de operare se încarcă prin *re-flash*, fără înlocuirea circuitului.
- **Paralelism nativ** – mai multe instanțe SM4 pot funcționa în paralel (de exemplu 4×128 bit pe tact) fără competiție pe magistralele CPU.
- **Consum redus de energie** – pentru același debit, implementarea hardware dedicată necesită curenti de ordinul zecilor de miliamperi, în timp ce o rutină software pe CPU mobil ar duce la sute de miliamperi și throttle termic.

Pentru a transforma pseudocodul criptografic în logică configurabilă, este necesar ca acesta să fie tradus folosind un *hardware description language* (HDL), adică un limbaj care descrie circuite prin relații logic-temporale, nu prin instrucțiuni secvențiale clasice. Diferit de limbajele C/C++ care indică *ce să facă* procesorul pas cu pas, un HDL descrie *ce conexiuni* trebuie să existe între porți logice, lăsând sintetizatorul să genereze rețea fizică a acestor porți logice ce sunt programate direct pe cipul de FPGA.

SystemVerilog este, în prezent, cel mai răspândit HDL. Față de Verilog-2001, el adaugă numeroase îmbunătățiri, cum ar fi tipuri `typedef struct`, `enum` și `interface`, precum și cicluri `for` parametrizabile (`genvar i`), făcând posibil un modul de runda SM4 scris mult mai simplificat, într-o modalitate ce aduce pe alocuri chiar cu programarea obiect-orientată clasica.

Vivado Design Suite de la AMD/Xilinx oferă un flux complet: editor, sintaxă color, sinteză, *place and route*, analiză de temporizare și debugging *in-chip* (Integrated Logic Analyzer), totul într-un singur proiect. Suporta VHDL, Verilog, SystemVerilog și chiar HLS (C/C++ tradus automat în logică). Raportul „Utilization” indică imediat câți LUT, flip-flop, BRAM și DSP consumă design-ul SM4, iar, pentru simulare, fereastra de waveform generată permite o analiză hands-on a proceselor ce se întâmplă la rularea programului. Programatorul integrat generează bitstream-ul și îl încarcă pe orice FPGA Xilinx, de la seria Artix până la ultrascalarele Versal, fără instrumente adiționale.

Prin combinarea flexibilității *SystemVerilog* cu ecosistemul unificat *Vivado*, SM4 ajunge de la pseudocod la siliciu cu mult mai mare ușurință decât prin alte metode, păstrând, de asemenea,

libertatea de a îmbunătăți performanța sau de a adăuga măsuri de securitate (masking, counter-mode) printr-o simplă recompilare a proiectului.

4.1. Diagrama generală a modulelor

Una dintre necesitățile la care am convenit că un algoritm de criptare trebuie să răspundă pentru a fi considerat un succes este *ușurința și versatilitatea implementării*. Din acest motiv, nu există un unic „be-all, end-all solution”: fiecare metodă de proiectare are plusuri atât și cât și minusuri. Totuși, se consideră că o implementare solidă ar trebui să respecte cel puțin următoarele criterii:

- **Separarea criptării și decriptării în module distincte** — ușurează depanarea și permite reutilizarea independentă a fiecărui modul.
- **Definirea unui *core module*** care gestionează atât criptarea, cât și decriptarea și oferă o interfață clară către mediul exterior, din aceleași motive de debug și reutilizare.
- **Definirea unui *top module*** ce instanțiază modulul *core*, include toate modurile de operare ale cifrului și expune o interfață la fel de intuitivă.
- **Folosirea unei interfețe de comunicație *lightweight*** — linii de date de cel mult 32–64 biți, bazată pe un protocol bine-cunoscut și documentat.
- **Păstrarea avantajului FPGA de procesare rapidă** — convenirea asupra unor algoritmi interni cu tempi morți minimi pentru a nu compromite debitul platformei.

În cele ce urmează, prezentăm modelul convenit și folosit pentru codul final, după care vom intra și vom vorbi despre fiecare dintre module la nivel individual, comentând direct pe părți din cod.

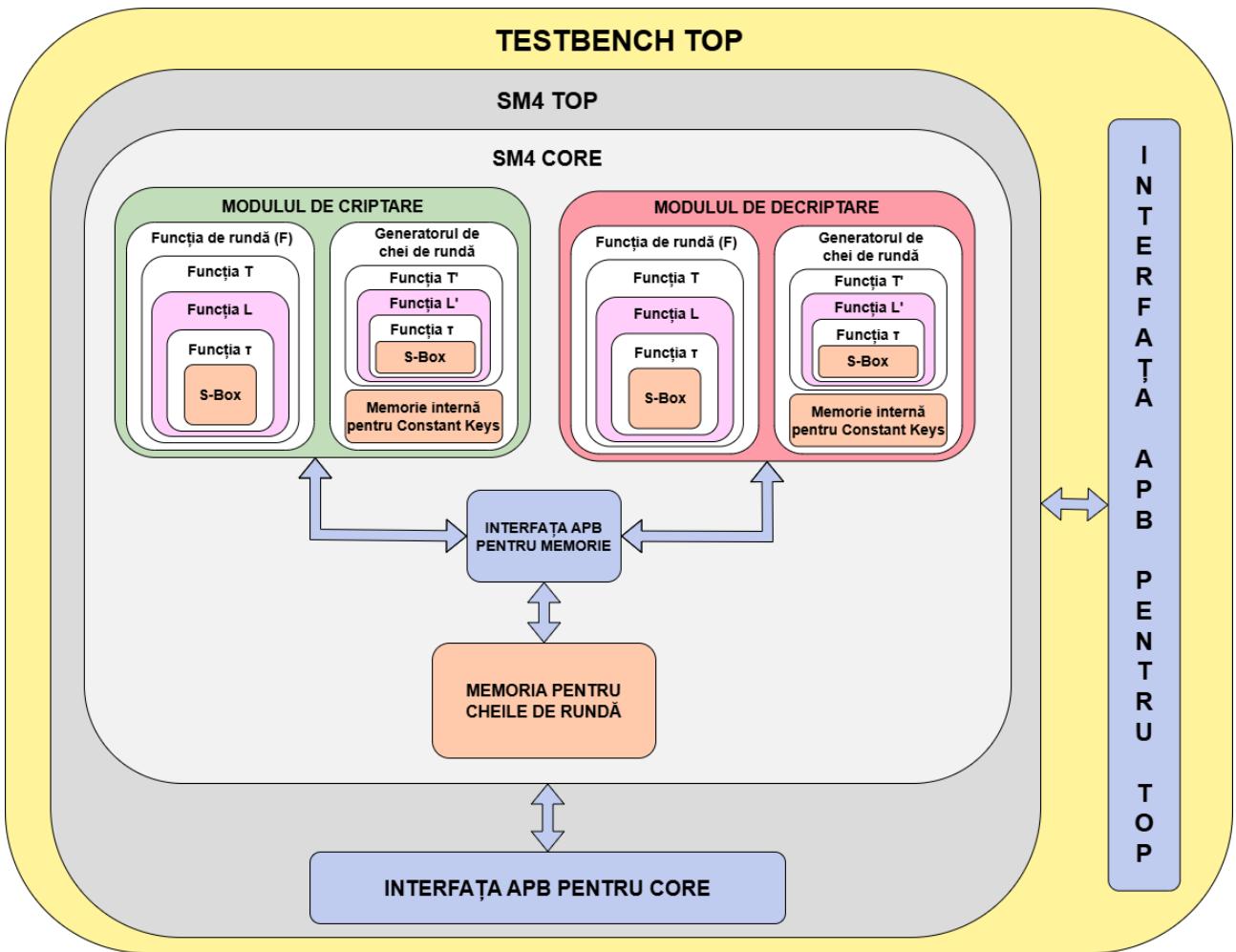


Figura 4.1.1: Diagrama generală pentru implementarea SM4

4.2. Funcțiile asincrone și memoriile pentru constante

Așa cum s-a menționat anterior, algoritmul SM4 folosește mai multe funcții, fiecare cu rol distinct. În practică, la nivel RTL se disting clar două familii:

- **Funcții asincrone** — nu necesită semnal de ceas. Exemplele tipice sunt memoriile pentru constante și transformările liniare L / L' . Aceste blocuri se implementează relativ simplu.
- **Funcții sincrone** — operează pe semnalul de tact (ceas). Aici intră funcția de rundă F și funcțiile de criptare/decriptare, care reprezintă de fapt implementările complete ale rețelelor Feistel. Ele constituie, de fapt, majoritatea dificultății proiectului, fiindcă implementările lor cer utilizarea de automate finite sincrone (FSM) pentru controlul fluxului de date.

Plecând de la cele asincrone, putem iarăși crea o clasificare, bazată de data aceasta mai direct pe modul în care acestea se implementează:

- *Memorii RAM:*
 - S-Box-ul propriu-zis;
 - memoria internă pentru cheile constante (CK_i).
- *Funcții logice pure:*
 - rotația circulară ($x \lll n$);
 - transformările liniare L și L' .
- *Module asincrone pentru accesul memoriei:*
 - funcția τ , care nu face decât să citească asincron din S-Box și să rearanjeze octetii.

În ordinea mențiunilor, arătam o moștră din implementarea *S-Boxului*:

```

1 module sbox(input [7:0] sin,output reg [7:0] sout);
2
3   always_comb
4     case (sin)
5       8'h00 : sout = 8'hd6;
6       8'h01 : sout = 8'h90;
7       8'h02 : sout = 8'he9;
8       //...
9     endcase
10
11 endmodule

```

De asemenea, implementarea foarte similară pentru memoria de *ConstantKeys*:

```

1 module constantKey(input [4:0] cin,output reg [31:0] cout);
2
3   always_comb
4     case (cin)
5       5'd0 : cout = 32'h0007_0E15;
6       5'd1 : cout = 32'h1C23_2A31;
7       5'd2 : cout = 32'h383F_464D;
8       //...
9     endcase
10
11 endmodule

```

Cât despre funcții, putem să începem cu funcția de shiftare circulară. SystemVerilog nu are această funcție built-in, așa că este nevoie să fie construită:

```

1 function automatic logic [31:0] rotate_left_32bit (input logic [31:0] v,
2                                     input int unsigned s);
3
4     int unsigned res;
5     int unsigned s_mod = s & 31; // modulo 31, a.i. sa nu avem 0 la iesire
6     rotate_left_32bit = (v << s_mod) | (v >> (32 - s_mod));
7     res = rotate_left_32bit;
8
9 endfunction

```

Practic, tot ce face această funcție este să concateneze cele două jumătăți de shiftări ale numărului introdus, făcând astfel wrap-around-ul. Ea este apelată în L și L':

```

1 function logic [31:0] Lfunction (input logic [31:0] lin);
2
3     int unsigned a;
4     Lfunction =
5         lin
6             ^ rotate_left_32bit(lin,  2)
7             ^ rotate_left_32bit(lin, 10)
8             ^ rotate_left_32bit(lin, 18)
9             ^ rotate_left_32bit(lin, 24);
10    a = Lfunction;
11
12 endfunction

```

```

1 function logic [31:0] Lprimefunction (input logic [31:0] lin);
2
3     Lprimefunction =
4         lin
5             ^ rotate_left_32bit(lin, 13)
6             ^ rotate_left_32bit(lin, 23);
7
8 endfunction

```

Și, desigur, avem și modulul asincron tau, care face patru atribuiri din S-box:

```

1 module tauFunction(input logic [31:0] tau_i,
2                     output logic [31:0] tau_o);
3
4
5     sbox sbox_i_1(tau_i[31:24],tau_o[31:24]);
6     sbox sbox_i_2(tau_i[23:16],tau_o[23:16]);
7     sbox sbox_i_3(tau_i[15:8],tau_o[15:8]);
8     sbox sbox_i_4(tau_i[7:0], tau_o[7:0]);
9
10 endmodule

```

4.3. APB: interfață de comunicație

Probabil cea mai importantă limitare a oricărui FPGA de dimensiuni medii sau mici nu este aceea a resurselor interne, majoritatea FPGA-urilor având suficiente LUT-uri cât să acopere majoritatea aplicațiilor de dimensiuni medii, ci aceea a pinilor externi de I/O. Acest obstacol, despre care vom discuta mai mult în capitolul despre sinteză, aduce nevoie ca, pentru modulele ce funcționează pe blocuri mai mari de date (ex.: 128/256 biți), modulul de top al design-ului să nu poată să transmită datele în aceste blocuri de dimensiuni mari, ci să fie nevoie să le spargă în cuvinte mai mici (de obicei multipli de 8, cum ar fi 16 sau 32 biți, în cazul nostru: 32 biți). În acest scop, modulul va folosi o interfață de comunicare pentru accesul la mediul exterior.

APB (Advanced Peripheral Bus) face parte din familia AMBA (ARM®) și este conceput drept un bus simplu, cu complexitate redusă, pentru conectarea perifericelor lente (timere, UART, GPIO) la magistrala de nivel superior (AHB/AXI). Principalele caracteristici ale APB-ului sunt:

- **Handshake pe două faze:** semnalele PSEL, PENABLE și PREADY asigură transferuri controlate, fără pipelining, răspunsul secundarului fiind întotdeauna obligatoriu pentru a realiza transferul.
- **Semnale minime:** numai PCLK, PADDR, PWDATA/PRDATA, PWRITE și selectori.
- **Performanță redusă, overhead minim:** ideal pentru registri de configurare și control, nu pentru transferuri masive de date.
- **Implementare simplă:** logică scăzută de control, latente deterministe, ușor de integrat în FPGA sau ASIC.

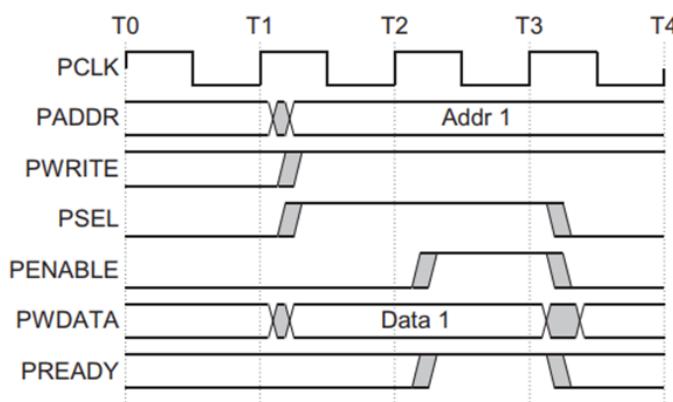


Figura 4.3.1: Cum arată un transfer de scriere pe APB

Implementarea în SystemVerilog este destul de simplă, deoarece limbajul are această clasă built-in:

```

1 interface apb_interface (input sysclk_i);
2
3     logic [4:0] paddr_i;
4     logic psel_i;
5     logic penable_i;
6     logic pwrite_i;
7     logic pready_o;
8     logic [31:0] pwdata_i;
9     logic [31:0] prdata_o;
10    logic pslverr_o;
11    logic memory_full;
12
13    modport principal (input sysclk_i, pready_o, prdata_o, pslverr_o, memory_full,
14                        output paddr_i, psel_i, penable_i, pwrite_i, pwdata_i);
15    modport secondary (input sysclk_i, paddr_i, psel_i, penable_i, pwrite_i, pwdata_i,
16                        output pready_o, prdata_o, pslverr_o, memory_full);
17
18    task write (input logic [31:0] addr, input logic [31:0] wdata);
19
20        paddr_i = addr;
21        psel_i = 1;
22        pwrite_i = 1;
23        pwdata_i = wdata;
24
25        @(posedge sysclk_i);
26        penable_i = 1;
27
28        @(posedge sysclk_i);
29        penable_i = 0;
30
31    endtask
32
33    task read (input logic [31:0] addr, output logic [31:0] rdata);
34
35        paddr_i = addr;
36        psel_i = 1;
37        pwrite_i = 0;
38
39        @(posedge sysclk_i);
40        penable_i = 1;
41
42        if (pready_o == 0)
43            @(posedge pready_o);
44
45        @(posedge sysclk_i);
46        penable_i = 0;
47        rdata = prdata_o;
48
49    endtask
50 endinterface

```

Pe lângă declararea semnalelor, s-au declarat și două modporturi, pentru primary și secondary, care practic specifică care dintre semnale sunt de input (adică *nu pot fi schimbat* de modulul curent), și care semnale sunt de output (adică *pot fi schimbat* de modulul curent). Adițional, s-au declarat și două task-uri ce definesc logica de scriere, respectiv citire, care însă pot fi folosite doar în afara design-ului, în *Testbench Top*.

Semnalul de tact (*sysclk_i*) și resetul negat ce a fost păstrat extern interfeței din motive de complexitate cauzate de folosirea a trei interfețe diferite **controlează** toată logica sincronă din modul, instrucțiunea ce dictează acest fapt fiind mereu:

```
1 always@(posedge sysclk_i or negedge rst_n_i)
```

Explicit, de fapt, toată logica acestui modul de criptare se află în aceste blocuri de *always*.

4.4. Funcția de criptare

Acum că am prezentat modulele independente de semnalul de ceas folosite în cadrul criptării, putem discuta despre funcțiile sincron T și T', precum și despre funcția F, numită generic „funcția de criptare”. Reamintim: funcțiile T și T' combină neliniaritatea (dată de funcția τ) cu liniaritatea (dată de funcțiile L și L'), și sunt apelate în funcția F respectiv în cea de generare de chei de rundă. Mai jos regăsiți implementarea funcției T, singura diferență față de T' fiind apelarea funcției L' în locul lui L.

```
1 module Tfunction(input logic sysclk_i,
2                   input logic rst_n_i,
3                   input logic [31:0] tin,
4                   output logic [31:0] tout);
5
6   logic [31:0] taux;
7   tauFunction tau1(tin,taux);
8
9   always@(posedge sysclk_i or negedge rst_n_i)
10    if(!rst_n_i)
11      tout <= 0;
12    else
13      tout <= Lfunction(taux);
14
15 endmodule
```

De asemenea, ne putem uita și la implementarea funcției F, funcția de criptare. Putem observa o logică secvențială simplă, care practic doar apelează funcția T, introducând la intrare o valoare obținută direct din textul destinat pentru criptare, reprezentată de vectorul x.

```
1 module Ffunction(input logic sysclk_i,
2                   input logic rst_n_i,
3                   input logic [31:0] x0_i,
```

```

4      input logic [31:0] x1_i,
5      input logic [31:0] x2_i,
6      input logic [31:0] x3_i,
7      input logic [31:0] rk_i,
8      output logic [31:0] fout);
9
10
10     wire [31:0] targ;
11     wire [31:0] faux;
12
13     assign targ = x1_i ^ x2_i ^ x3_i ^ rk_i;
14
15     Tfunction T1(sysclk_i, rst_n_i, targ, faux);
16
17     always@(posedge sysclk_i or negedge rst_n_i)
18         if(!rst_n_i)
19             fout <= 0;
20         else
21             fout <= x0_i ^ faux;
22
23 endmodule

```

Aşa cum am prezentat în capitolul ce explica design-ul algoritmului SM4, implementarea criptării constă în crearea FSM-ului corespunzator pentru realizarea reţelei Feistel. De asemenea, mai există problema generării de cheilor de rundă, lucru care, într-o implementare de calitate, nu trebuie să se întâmple de fiecare data cand incepe o nouă criptare, atât timp cat masterKey-ul nu s-a schimbat între timp, deoarece ar fi ineficient și ar consuma resurse inutil. Vom explora acum cum a fost facută implementarea pentru cele două variante: aceea când cheile au fost deja generate, și atunci cand nu.

Criptarea cu chei negenerate - SM4 Immediate Encryption

Incepând de la coadă, cu situația în care cheile trebuie generate, una dintre inovațiile aduse de această implementare este implementarea algoritmului autointitulat **SM4 - Immediate Encryption**.

În mod normal, dacă cheile nu sunt încă generate, o implementare clasică ar spune că prima dată trebuie să se aștepte ca blocul de generarea a cheilor să termine de umplut toată memoria cu cele 32 de chei, corespondente celor 32 de runde de criptare, iar de abia după aceasta să se dea voie algoritmului de criptare sa pornească. **Immediate Encryption** folosește două automate inteligent programate, astfel încât, când blocul de criptare trebuie să folosească cheia de rundă pentru calcul, aceasta să fie deja generată, permitînd astfel ca acesta să nu aștepte deloc, practic paralelizând două operațiuni oricum obligatorii.

În practică, se folosește aceeași variabilă de stare, numită *key_state*, care este controlată de blocul de criptare, și incrementată la fiecare ciclu de ceas. Aşa cum se poate observa în diagramele de mai jos, există un decalaj între cele două automate. În primul rând, starea de *Wait* a FSM-ului blocului de encriptie este cu un ciclu *mai devreme* decât cea a FSM-ului lui RKG.

În al doilea rând, putem face afirmația că automatul RKG se întinde pe mai multe stări decât ar fi necesar; de pildă, se poate observa că stările **GENERATE** și **UPDATE** ar fi putut fi comasate într-o singură stare. Acest aspect al design-ului trebuie obligatoriu respectat, deoarece este nevoie ca, atunci când automatul de criptare intră în starea **ENCRYPT**, cheia de rundă să fie deja generată și stabilă pentru procurare.

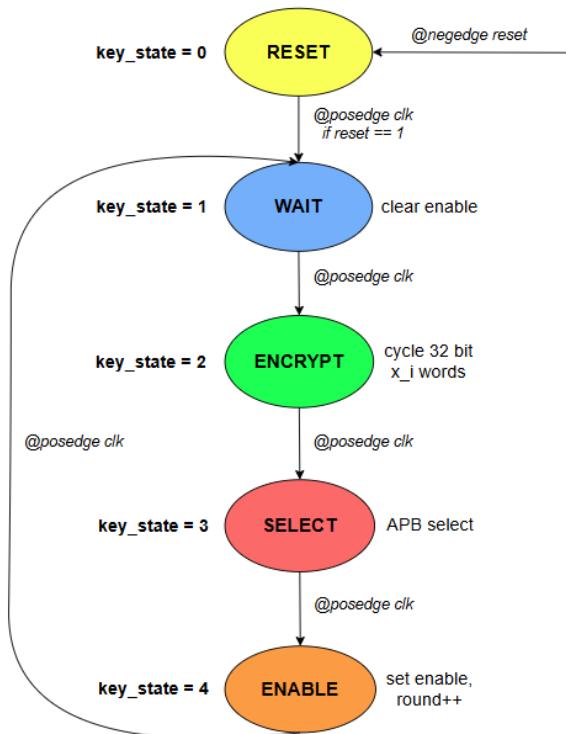


Figura 4.4.1: FSM-ul Criptării pentru Immediate Encryption

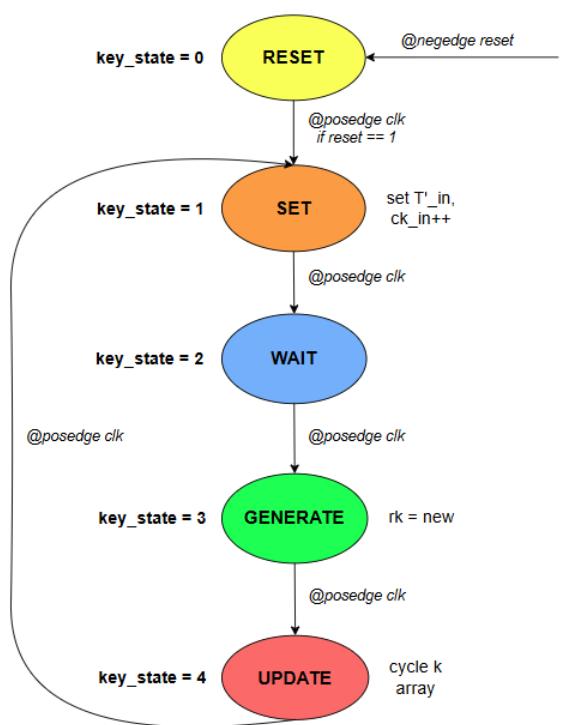


Figura 4.4.2: FSM-ul Generatorului de Chei

În cadrul acestui proiect, sunt folosite la mai multe niveluri diverse *automate*, acestea fiind gândite și scrise în stil *primary - secondary*. Ca să putem vedea cum funcționează în general această logică la nivel de cod, putem să exemplificăm chiar pe acest exemplu, unde **RKG** este *secondary* și **Encryption** este *primary*. Vom începe cu cel pentru RKG:

```

1 always @(posedge sysclk_i or negedge rst_n_i) begin
2
3     if(!rst_n_i) begin
4
5         k[0] <= k_init[0];
6         k[1] <= k_init[1];
7         k[2] <= k_init[2];
8         k[3] <= k_init[3];
9         ck_i <= 5'd0;
10        aux_key <= 128'd0;
11        tprime_i <= 32'd0;

```

```

12    rk_o <= 32'd0;
13
14  end else begin
15
16    if (master_key_i != aux_key) begin
17
18      k[0] <= k_init[0];
19      k[1] <= k_init[1];
20      k[2] <= k_init[2];
21      k[3] <= k_init[3];
22      ck_i <= 0;
23
24    end
25
26    aux_key = master_key_i;
27
28    if (key_state_i == 1 && enable_i == 1) begin
29
30      tprime_i <= k[1] ^ k[2] ^ k[3] ^ ck_o;
31      ck_i <= ck_i + 1;
32
33    end else if (key_state_i == 3 && enable_i == 1) begin
34
35      rk_o <= k[0] ^ tprime_o;
36
37    end else if (key_state_i == 4 && enable_i == 1) begin
38
39      k[0] <= k[1];
40      k[1] <= k[2];
41      k[2] <= k[3];
42      k[3] <= rk_o;
43
44    end
45
46  end
47
48 end

```

Fiind de tip *secondary*, automatul pentru **RKG** trebuie doar să verifice valoarea variabilei care determină starea (`key_state_i`) la fiecare front ascendent al ceasului, iar pentru fiecare opțiune să execute instrucțiunile corespunzătoare stării curente. Totodată, pentru fiecare stare, trebuie să se acorde atenție ordinii și condițiilor specifice instrucțiunilor implementate, aşa cum se întâmplă, de pildă, în cazul în care `key_state_i == 4`, când rotația valorilor vectorului `k` trebuie realizată în ordine, pentru a nu se pierde informația.

În plus, a fost introdus semnalul `enable_i`, care controlează trecerea automatului între stări, util pentru a asigura **oprirea** generării de chei după ce toate cele 32 au fost produse. Condiția aplicată semnalului `master_key_i` are rolul de a reseta procesul de generare atunci când cheia de criptare se

schimbă, invalidând astfel toate calculele anterioare.

Acestea sunt principalele aspecte de discutat despre automatul blocului de *Generare de Chei de Runda*. Putem să vedem acum codul pentru *automatul de tip primary*, al blocului de **Criptare**:

```
1 if (encryption_dir_i == 2'b11 && encryption_done_o == 0 &&
2   fast_enc_started == 0) begin
3
4   enable_key_gen <= 1;
5   rk_source <= 0;
6   fast_enc_started <= 1;
7   key_state <= 1;
8
9 end
10
11 if (round_cnt < 32 && encryption_dir_i == 2'b11 && enable_key_gen == 1 &&
12   encryption_done_o == 0 && fast_enc_started == 1) begin
13
14   fsm_state <= IDLE;
15   encryption_done_o <= 0;
16   key_state <= key_state + 1;
17
18   if (key_state == 1) begin
19
20     vif.penable_i <= 0;
21
22   end
23
24   if (key_state == 2 && round_cnt > 0) begin
25
26     x0 <= x1;
27     x1 <= x2;
28     x2 <= x3;
29     x3 <= fout;
30
31   end
32
33   if (key_state == 3) begin
34
35     vif.paddr_i <= round_cnt;
36     vif.psel_i <= 1;
37     vif.pwrite_i <= 1;
38
39   end
40
41   if (key_state == 4) begin
42
43     vif.pwdata_i <= round_key;
44     vif.penable_i <= 1;
45     key_state <= 1;
```

```

46    round_cnt <= round_cnt + 1;
47
48 end
49
50 end else if (round_cnt == 32 && encryption_dir_i == 2'b11 && enable_key_gen == 1 &&
51     encryption_done_o == 0 && fast_enc_started == 1 && wait_clock == 0 &&
52     fast_enc_finished == 0) begin
53
54     wait_clock <= 1;
55
56 end else if (round_cnt == 32 && encryption_dir_i == 2'b11 && enable_key_gen == 1 &&
57     encryption_done_o == 0 && fast_enc_started == 1 && wait_clock == 1 &&
58     fast_enc_finished == 0) begin
59
60     wait_clock <= 0;
61     x0 <= x1;
62     x1 <= x2;
63     x2 <= x3;
64     x3 <= fout;
65     fast_enc_finished <= 1;
66
67 end else if (fast_enc_finished == 1) begin
68
69     fast_enc_finished <= 0;
70     cypher_text_o[127:0] <= {x3, x2, x1, x0};
71     encryption_done_o <= 1;
72     fast_enc_started <= 0;
73
74 end

```

În primul rând, trebuie să indicăm modulului de unde să-și preia cheile, din memorie sau pe cea generată. Acest lucru se realizează prin variabila rk_source, care, pentru generator, trebuie setată la 0; în aceeași condiție if se activează și bitul de enable pentru generarea de chei, menționat anterior.

În continuare, cât timp contorul de rundă este mai mic decât 32, parcurgem logica FSM-ului, trecând pe rând, la fel ca în automatul RKG, prin fiecare stare posibilă și aplicând seria de instrucțiuni corespunzătoare stării curente. Diferențele majore apar prin cele două contoare „de control”, key_state și round_cnt, care sunt incrementate în cadrul acestui automat. key_state se incrementează la fiecare front ascendent de ceas, făcând ca ambele FSM-uri să funcționeze sincron la frecvența ceasului, iar round_cnt crește de fiecare dată când automatul ajunge în starea 4, corespunzătoare cu „ENABLE” pentru Encryption și „UPDATE” pentru RKG.

Deoarece key_state este incrementat la începutul buclei, round_cnt va lua valori între 0 și 31 pe durata celor 32 de runde de criptare. Când valoarea 32 este atinsă, se mai așteaptă încă doi cicli de ceas cu ajutorul variabilelor wait_clock și fast_enc_finished, pentru asigurarea ultimei runde de criptare, considerând că aceasta se întâmplă în bucla principală doar când key_state are valori cuprinse între 1 și 31. În final, cifrul este stocat în memoria internă a modulului, în variabila

`cypher_text_o`, iar indicatorul `encryption_done_o` este activat.

Criptarea cu chei pregenerate

În cele mai multe cazuri, definițiile algoritmilor de criptare pe bloc se bazează strict pe calculele și aspectele matematice care fac posibilă criptarea. Desigur, nici specificația algoritmului SM4 nu tratează situația în care cheile de rundă sunt deja cunoscute, deoarece acest detaliu ține de implementare. În practică, însă, nu există niciun motiv să regenerăm cheile de fiecare dată de la zero, atâtă vreme cât cheia de criptare rămâne neschimbată; o astfel de abordare ar fi ineficientă.

În implementarea noastră, distincția este realizată printr-o memorie internă ce păstrează cheile generate—memorie pe care o vom detalia ulterior—și prin definirea și implementarea unui nou FSM care acoperă acest caz. Când memoria îi semnalează modulului de criptare că are deja stocate toate cheile de rundă, variabilei `rk_source` i se atribuie valoarea 1 (corespunzătoare citirii din memorie), iar FSM-ul dedicat pornește:

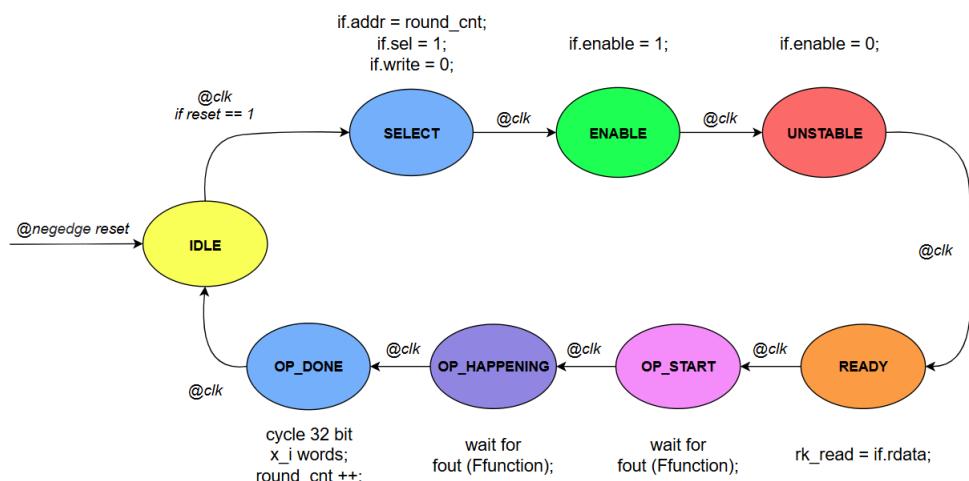


Figura 4.4.3: FSM-ul criptării cu chei pregenerate

Luând, pe rând, fiecare stare:

- **IDLE** — starea de reset; nu se execută nicio instrucțiune specifică.
- **SELECT** — corespunde fazei *SETUP* pe magistrala APB, când are loc selecția.
- **ENABLE** — corespunde primului ciclu de ceas din starea *ACCESS* pe APB, când se activează semnalul `enable` și se aşteaptă `ready`.
- **UNSTABLE** — deoarece răspunsul `ready` este întârziat, această stare coincide cu frontul ascendent al ceasului în care răspunsul sosește, iar datele citite nu sunt încă stabilite.
- **READY** — datele de pe APB sunt pregătite pentru citire; cheia de rundă este stocată local în variabila `rk.read`.

- **OP_START** — începe calculul funcției F de criptare; nu se execută instrucțiuni suplimentare.
- **OP_HAPPENING** — încă se așteaptă rezultatul funcției F de criptare; nu se execută instrucțiuni suplimentare.
- **OP_DONE** — rezultatul criptării este gata; cele patru cuvinte de 32 de biți se ciclizează, iar automatul trece la următoarea rundă de criptare.

Putem vedea mai jos cum arată o metodă mai explicită de redactare a codului pentru această situație:

```

1 if (vif.memory_full == 1 && encryption_dir_i == 2'b11 && encryption_done_o == 0) begin
2
3   enable_key_gen <= 0;
4   rk_source <= 1;
5   fast_enc_started <= 0;
6   fast_enc_finished <= 0;
7   encryption_done_o <= 0;
8   wait_clock <= 0;
9
10  if (fsm_state == IDLE)
11    fsm_state <= SELECT;
12
13  else if (fsm_state == SELECT)
14    fsm_state <= ENABLE;
15
16  else if (fsm_state == ENABLE)
17    fsm_state <= UNSTABLE;
18
19  else if (fsm_state == UNSTABLE)
20    fsm_state <= READY;
21
22  else if (fsm_state == READY)
23    fsm_state <= OP_START;
24
25  else if (fsm_state == OP_START)
26    fsm_state <= OP_HAPPENING;
27
28  else if (fsm_state == OP_HAPPENING)
29    fsm_state <= OP_DONE;
30
31  else if (fsm_state == OP_DONE)
32    fsm_state <= SELECT;
33
34
35  if (fsm_state == SELECT) begin
36
37    vif.paddr_i <= round_cnt;
38    vif.psel_i <= 1;

```

```

39      vif.pwrite_i <= 0;
40
41  end else if (fsm_state == ENABLE) begin
42
43      vif.penable_i <= 1;
44
45  end else if (fsm_state == UNSTABLE) begin
46
47      vif.penable_i <= 0;
48
49  end else if (fsm_state == READY) begin
50
51      if (vif.pready_o == 1 && vif.pslverr_o == 0)
52          rk_read <= vif.prdata_o;
53
54  end else if (fsm_state == OP_DONE) begin
55
56      round_cnt <= round_cnt + 1;
57      x0 <= x1;
58      x1 <= x2;
59      x2 <= x3;
60      x3 <= fout;
61
62  end
63
64 // Encryption done
65 if (round_cnt == 32 && fast_enc_started == 0 && encryption_dir_i == 2'b11 &&
66     encryption_done_o == 0) begin
67
68     cypher_text_o[127:0] <= {x3, x2, x1, x0};
69     encryption_done_o <= 1;
70     fast_enc_started <= 0;
71
72 end
73
74 end

```

4.5. Funcția de decriptare

Asemenea unui puzzle care poartă în sine și cheia desfacerii lui, a sosit clipa să răsucim mecanismul SM4 în sens invers și să vorbim despre *decriptare*. Funcția de decriptare din algoritmul SM4 reproduce, în linii mari, aceleași etape ca și criptarea.

În primul rând, mecanismul de substituție–permutație rămâne neschimbăt, însăciat acesta apelează în același fel funcția F. Pe lângă asta, în cadrul modulului de decriptare se implementează o nouă instanță a aceluiasi *modul de generare a cheilor de rundă*, ce folosește un automat foarte similar cu cel de la criptare, toate instrucțiunile pentru salvarea cheilor în memorie fiind identice. Singura deosebire

reală o constituie **ordinea inversă** în care sunt aplicate cheile de rundă, fapt exemplificat în cod prin trecerea de la această instrucțiune în modulul de criptare:

```

1 if (fsm_state == SELECT) begin
2
3     vif.paddr_i <= round_cnt;
4     // ....
5
6 end

```

la aceasta:

```

1 if (fsm_state == SELECT) begin
2
3     vif.paddr_i <= 31 - round_cnt;
4     // ....
5
6 end

```

Desigur, această necesitate de a accesa cheile în ordine inversă face ca orice formă de *Immediate Decryption* să fie imposibilă, generarea cheilor nefiind o operație ce să poată fi făcută în ambele sensuri. Așadar, pentru decriptare există doar două opțiuni: fie cheile sunt **generate deja** (pot să fi fost generate și după o criptare, atâtă timp cât cheia nu s-a schimbat acestea nu sunt șterse din memorie), caz în care se pune în funcțiune **exact același FSM** ca la criptare, fie avem cazul în care cheile **trebuie generate**. Pentru acesta din urmă, aşa cum este menționat în paragraful anterior, un automat bazat pe aceeași variabilă `key_state` este pus în funcțiune, automat ce se ocupă doar de **generarea cheilor**, cu o structură foarte similară cu cel de la criptare, explicitată mai jos:

```

1 if (round_cnt < 32 && encryption_dir_i == 2'b10 && enable_key_gen == 1 &&
2     decryption_done_o == 0 && gen_dec_started == 1) begin
3
4     fsm_state <= IDLE;
5     decryption_done_o <= 0;
6     key_state <= key_state + 1;
7
8     if (key_state == 1) begin
9
10        vif.penable_i <= 0;
11
12    end
13
14    if (key_state == 3) begin
15
16        vif.paddr_i <= key_gen_round;
17        vif.psel_i <= 1;
18        vif.pwrite_i <= 1;
19

```

```

20    end
21
22    if (key_state == 4) begin
23
24        vif.penable_i <= 1;
25        vif.pwdata_i <= round_key;
26        key_state <= 1;
27        key_gen_round <= key_gen_round + 1;
28
29    end
30
31 end
32
33 if (round_cnt == 32 && encryption_dir_i == 2'b10 && decryption_done_o == 0) begin
34
35     // Decryption done
36     decrypted_text_o[127:0] <= {x3, x2, x1, x0};
37     decryption_done_o <= 1;
38     gen_dec_started <= 0;
39
40 end

```

4.6. Memoria pentru cheile de rundă

Așa cum am promis în numeroasele mențiuni anterioare, a sosit momentul să descifrăm și modulul de memorie pentru cheile de rundă. Termenul de **memorie** este unul dintre cele mai vechi apărute în informatică, încrucișat orice formă de procesare are nevoie de un mecanism de stocare permanentă sau temporară a unor date. În consecință, acest termen a preluat numeroase definiții, cu aplicații reale adesea foarte diferite între ele, ceea ce impune o descriere mai explicită a modulului nostru.

Într-o încercare de a integra hardware-ul nostru în definițiile larg folosite și înțelese, putem afirma, în linii mari, că modulul de memorie pentru cheile de rundă este un *RAM (Random-Access Memory) de dimensiuni reduse*, accesat printr-o *interfață de tip APB*. În SystemVerilog, un RAM cu 32 de valori de 32 de biți se poate defini astfel:

```
1 logic [31:0] rk_mem [0:31];
```

Adițional, o memorie are nevoie și de un mecanism prin care să țină evidența gradului de umplere, precum și, mai specific, a adreselor libere. În cazul nostru, folosim un vector de apariție, numit `app_array`, format din 32 de biți corespunzători celor 32 de chei de rundă, și un bit de stare, `memory_full`, care, așa cum am văzut anterior, este transmis pe interfața APB către modulele de criptare și decriptare.

```
1 static logic memory_full = 0;
2 static logic [31:0] app_array;
```

Acest modul este unul mai simplu, având nevoie doar să implementeze **logica de secondary de APB**, care să răspundă cererilor facute de modulele primary, respectiv de cele de *criptare si decriptare*.

Mecanismul de secondary este mai simplu decât cel de primary, deoarece el trebuie doar să reacționeze atunci când este *selectat si enabled*, și trebuie doar să facă diferență dintre o scriere și o citire, aşa cum se poate vedea în modalitatea de implementare de mai jos:

```
1 begin
2
3     // Resetarea pulsurilor la valoarea de 0 după un ciclu
4     vif.pready_o <= 0;
5     vif.pslverr_o <= 0;
6
7     // Memoria se resetează la schimbarea cheii
8     if (aux_mk != master_key_i)
9         reset_memory();
10
11    aux_mk = master_key_i;
12
13    // Logica de APB
14    if (vif.psel_i == 1 && vif.penable_i == 1) begin
15
16        vif.pready_o <= 1;
17
18        // Citire
19        if (!vif.pwrite_i) begin
20
21            vif.prdata_o <= rk_mem[vif.paddr_i];
22
23            if (app_array[vif.paddr_i] == 1)
24                vif.pslverr_o <= 1'b0;
25            else
26                vif.pslverr_o <= 1'b1;
27
28        // Scriere
29        end else if (vif.pwrite_i && app_array[vif.paddr_i] == 0) begin
30
31            rk_mem[vif.paddr_i] <= vif.pwdata_i;
32            app_array[vif.paddr_i] <= 1'b1;
33
34            if (rk_mem[vif.paddr_i] == vif.pwdata_i)
35                vif.pslverr_o <= 1'b0;
36            else
37                vif.pslverr_o <= 1'b1;
38
39        end
40
41    end
42
```

```

43 // Verificarea gradului de umplere al memoriei
44 if (memory_wait == 1)
45     memory_full <= 1'b1;
46
47 if (&app_array)
48     memory_wait <= 1'b1;
49
50 end

```

În ciuda simplității acestui modul, necesitatea lui pentru acest design nu trebuie să fie subestimată. Reamintim că, pe lângă beneficiile evidente de eficiență pe care le aduce prin eliminarea necesității regenerării cheilor pentru operații cu aceeași cheie, operația de decriptare **nu poate fi realizată** fără acest modul, fapt datorat modului în care generarea de chei de rundă funcționează ca o *funcție ireversibilă*.

4.7. SM4 Core: mașina de stări pentru operarea criptării/decriptării

Modalitatea în care se abordează un proiect de design de hardware se bazează pe un concept simplu: *îmbunătățirea și ușurarea reutilizabilității și a testării*. Din acest motiv, se preferă de obicei ca design-ul să fie cât mai modular, cu o ierarhie clar definită, și cu un „ambalaj” (denumit în industrie drept **top** sau **modul de top**) ce oferă o comunicare cu exteriorul pe o interfață cunoscută, și care să funcționeze pe principiul de **"black box"**.

Pentru SM4, modularizarea ce a fost stabilită este aceea de a avea două module de top:

- la nivel inferior, avem **SM4 Core**, care este black box-ul pentru criptare și decriptare, ce primește la intrare, pe APB: cheia, un mesaj de 128 de biți și tipul operației dorite, având la ieșire rezultatul pe 128 de biți.
- iar la nivel superior, găsim modulul **SM4 Top Module**, care primește la intrare, tot pe APB: cheia, un mesaj oricât de mare, tipul operației dorite și modul de operare impus.

Continuând analiza în același sens ascendent, *SM4 Core* funcționează pe baza unui automat scurt, însă un pic mai complex decât cele de până acum, deoarece trecerea între stări nu se mai face la fiecare ciclu de ceas, ci bazat pe cerințele lui *SM4 Top Module* și pe răspunsul primit de la modulele de *Encryption* și *Decryption*. Putem observa mai jos structura automatului:

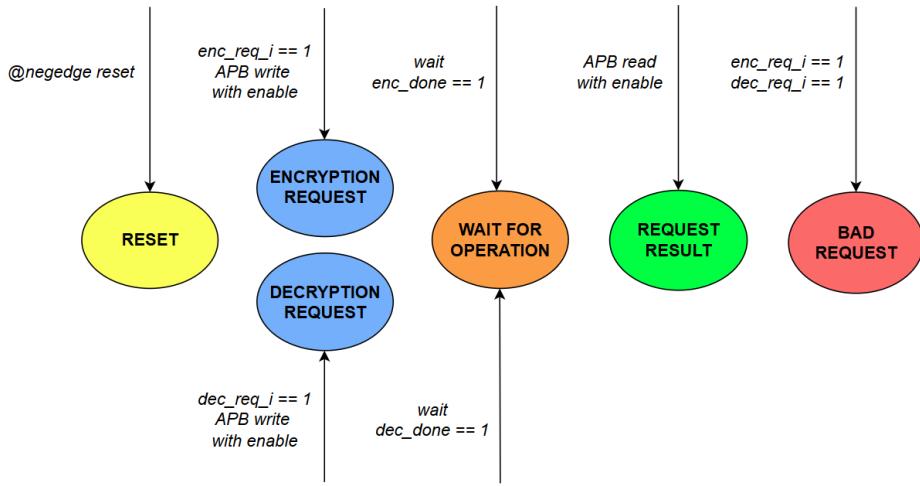


Figura 4.7.1: FSM-ul SM4 Core

În ciuda aspectului neconvențional al FSM-ului, în afara de starea de reset, toate stările sunt accesate doar pe front pozitiv de ceas, acest aspect făcând în continuare posibilă includerea acestei mașini de stări în categoria automatelor sincrone. Analizând, pe rând, cele cinci stări individuale:

- **RESET** — conține un set de instrucțiuni standard pentru orice stare de reset, setând valoarea 0 pentru toți registrii interni relevanți și toate semnalele *secondary* de pe APB.
- **ENCRYPTION/DECRYPTION REQUEST** — intrarea în aceste stări se face prin *request* de la *SM4 Top Module*; în cadrul lor se scriu în memoria modulului *SM4 Core*, pe rând, cheia și mesajul folosite pentru operația cerută.
- **WAIT FOR OPERATION** — când automatul se află în această stare, așteaptă ca unul dintre modulele de criptare sau decriptare, cel selectat pentru operație, să returneze *flag*-ul care semnifică finalizarea operației.
- **REQUEST RESULT** — cealaltă stare validă în care se intră prin *request* de la *SM4 Top Module*; tot pe interfața APB, în această stare se transmite rezultatul operației cerute către *top*.
- **BAD REQUEST** — stare de eroare în care se intră când se cer, simultan, și criptare, și decriptare, generând semnalul de eroare *pslverr_o* pe interfața APB.

Modalitatea simplă și intuitivă de utilizare a acestui modul, combinată cu faptul că acesta comunică pe interfața APB, facilitează testarea funcționalității acestuia și permite chiar folosirea lui separată într-un alt fel de proiect.

4.8. SM4 Top Module: implementarea modurilor de operare

Scopul propus prin acest proiect este crearea unui sistem, care să fie ulterior implementat pe FPGA, capabil să primească un mesaj de orice lungime și să îl poată cripta și decripta folosind logica internă.

Până acum am detaliat totul până la modulul *SM4 Core*, care conține logica algoritmului doar pentru dimensiunea standard a mesajului prevăzută de SM4, adică 128 de biți, fapt ce corespunde, în practică, la 16 simboluri de câte 8 biți, fie ele litere, caractere speciale sau spații.

Pentru a putea procesa mesaje de dimensiuni mai mari, este nevoie să implementăm un modul ce înțelege, reține în memorie și poate procesa acest tip de mesaje, și care are integrată logica pentru toate cele cinci moduri standard de operare. Reamintim că Institutul Național de Standarde și Tehnologie din SUA definește în 2001 cinci moduri de operare folosite pentru un cifru de tip bloc: *Electronic Circuit Board (ECB)*, *Cipher Block Chaining (CBC)*, *Output Feedback Mode (OFB)*, *Cipher Feedback Mode (CFB)* și *Counter Mode (CTR)*.

Modulul *SM4 Top Module* este responsabil pentru această funcționalitate, furnizând următoarele soluții pentru toate cerințele menționate:

- utilizarea interfetei *APB* pe 32 de biți: mesajul de dimensiune mare este împărțit în blocuri de 128 de biți, transmise în patru scrieri (pentru mesaj) / citiri (pentru răspuns) pe interfață;
- utilizarea unor semnale de control (*start_enc_i* și *start_dec_i*) pentru a cere o criptare sau o decriptare, respectiv *message_done_i* pentru a marca faptul că mesajul curent a ajuns la final;
- crearea unui *FSM* intern, similar cu cel din *SM4 Core*, care include un set de instrucțiuni specifice pentru diverse combinații de răspunsuri primite fie din exterior, fie de pe interfață către *SM4 Core*; acesta va fi prezentat în cele ce urmează;
- existența unor registri interni pentru stocarea tuturor informațiilor necesare: cheia, vectorul de inițializare, mesajul și răspunsul sau cifrul precedent pentru modurile de operare cu feedback;
- folosirea unor *flag-uri* (*encryption_res_valid_o* și *decryption_res_valid_o*) pentru a semnaliza către exterior că operația solicitată s-a încheiat; ele sunt setate de modul numai după ce procesarea completă a blocului de 128 de biți s-a terminat.

Codul acestui modul este ceva mai lung, aşa că este prezentat în integritate în **Anexa 1**. Putem acum să prezentăm automatul pe care funcționează acest modul:

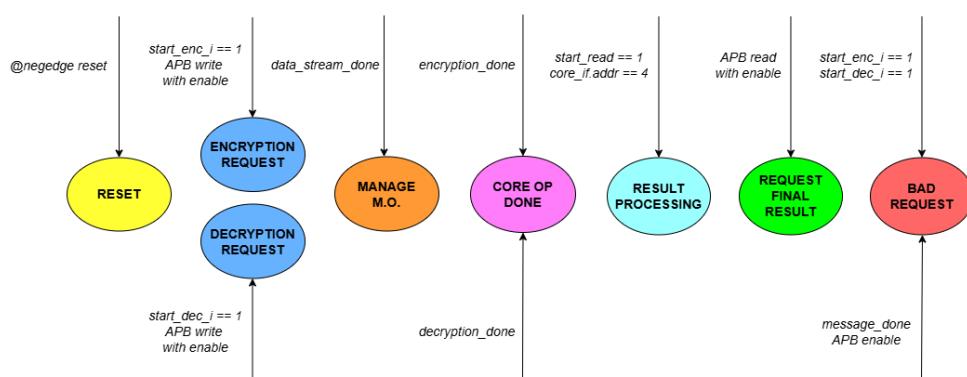


Figura 4.8.1: FSM-ul SM4 Top Module

Putem observa că avem câteva stări cu care ne-am întâlnit și la *SM4 Core*, dar și câteva stări noi. O să le considerăm pe rând:

- **RESET** — similară cu cea din *SM4 Core*; se ocupă să reseteze toți registrii interni și semnalele de interfață care îi sunt alocate prin *modports* (cele de *primary* pentru interfața cu *SM4 Core* și cele de *secondary* pentru interfața cu exteriorul).
- **ENCRYPTION/DECRYPTION REQUEST** — din nou, foarte similară cu cea din *SM4 Core*; se stochează în memoria internă toate cele necesare operației cerute (cheia și mesajul pentru *ECB*, iar, adițional, vectorul de inițializare pentru celelalte moduri de operare). În plus, această stare conține un *flag* numit *first_access*, care diferențiază între primul acces pentru un mesaj mai lung, când se transmit toate informațiile, și accesările ulterioare, în care se transmite doar textul, deoarece pe durata unei operații cheia și vectorul de inițializare rămân neschimbate.
- **Manage M.O. (Mode of Operation)** — în această stare, pe baza modului de operare cerut, se alocă în registrii interni valorile corespunzătoare operației ce va fi cerută de la *SM4 Core* (ex.: textul de criptat este mesajul pentru *ECB*, dar pentru *CTR* este vectorul de inițializare).
- **CORE OP DONE** — se intră în această stare atunci când *SM4 Core* transmite că operația cerută a luat sfârșit; rezultatul este stocat în registrii interni.
- **RESULT PROCESSING** — se atribuie rezultatului final valoarea corespunzătoare modului de operare cerut, după procesarea internă a acestuia (ex.: pentru *CTR*, se realizează operația *XOR* între rezultatul furnizat de *SM4 Core* și mesaj).
- **REQUEST FINAL RESULT** — se intră în această stare atunci când, pe interfața cu exteriorul, se solicită citirea rezultatului; acesta este transmis pe interfață în 32 de biți, după post-procesare.
- **BAD REQUEST** — similară cu cea din *SM4 Core*; stare de eroare în care se intră în două cazuri: dacă se cer simultan criptare și decriptare, sau dacă se dă *enable* pe APB-ul cu exteriorul în timp ce este ridicat flag-ul *message_done_i*.

4.9. Simulation Testbench – testarea funcționalității în simulatorul Vivado

Procesul de design hardware presupune că, imediat după ce toate modulele au fost concepute și se consideră că s-a ajuns la o versiune finală, înainte ca designul să fie încărcat pe *FPGA*, acesta să fie testat mai întâi într-un *simulator*. Chiar dacă, de multe ori, pot apărea diferențe între rezultatele obținute în *simulator* și cele din *mediul real*, acest pas este esențial și reprezintă o **unealtă foarte puternică** pentru depistarea și remedierea erorilor de design.

SM4 este un algoritm cu o complexitate matematică ridicată. Din acest motiv, testarea validității calculelor poate fi greoaie, deoarece realizarea acestora manual, chiar și cu ajutorul unui calculator

digital și al unor programe precum *WolframAlpha*, durează foarte mult. În consecință, proiectanții de algoritmi de criptare oferă o alternativă mult mai simplă: **vectorii de test**. Aceștia sunt diverse tupluri de valori (pentru *SM4*, de forma: *cheie*, *vector de inițializare*, *mesaj*, *text criptat*), precalculate și deja verificate pentru validitate. *SM4* include, de asemenea, în draft-ul online de prezentare, multiple astfel de **vectori de test**, pentru toate cele cinci moduri de operare.

Asadar, în linii mari, continutul unui testbench bun, și implicit al celui folosit în cazul testării *SM4*, trebuie fie format din:

- **Inițializarea** mai multor *vectori de test*, pentru toate modurile de operare, cel puțin 2 per mod.

Pentru testarea modulelor noastre, este creată o structură de date care să stocheze acești vectori de test, în care câmpurile pentru mesaj și pentru textul criptat au dimensiunea alocată dinamic:

```

1 typedef struct { logic [127:0] key;
2                     bit [127:0] plain [];
3                     bit [127:0] cypher [];
4                     bit [127:0] iv;
5                     modes_of_op moo;
6 } test_vector;
```

care primește vectori prin următorul task:

```

1 task add_test_vector(input int idx, input logic [127:0] key, input byte plain [],
2                      input byte cypher [], input logic [127:0] iv, modes_of_op moo);
3
4     vectors[idx].key = key;
5     vectors[idx].iv = iv;
6     vectors[idx].moo = moo;
7
8     vectors[idx].plain  = new[(plain.size()+15) /16];
9     vectors[idx].cypher = new[(plain.size()+15) /16];
10
11    for (i = 0; i < plain.size()/16; i++) begin
12
13        for (j = 0; j < 16; j++) begin
14
15            aux = 'h0 + plain[16*i + j]<<8*j;
16            vectors[idx].plain[i] += aux;
17
18            aux = 'h0 + cypher[16*i + j]<<8*j;
19            vectors[idx].cypher[i] += aux;
20
21        end
22    end
23
24 endtask
```

acești vectori fiind scriși la începutul codului, în felul următor:

```

1 initial begin
2     add_test_vector(.idx(0),
3                     .key('h01_23_45_67_89_AB_CD_EF_FE_DC_BA_98_76_54_32_10),
4                     .plain({<<8{'h01_23_45_67_89_AB_CD_EF_FE_DC_BA_98_76_54_32_10}}),
5                     .cypher({<<8{'h68_1E_DF_34_D2_06_96_5E_86_B3_E9_4F_53_6E_42_46}}),
6                     .iv(0),
7                     .moo(ECB)
8     );
9
10    // ...
11 end

```

- O modalitate de a pasa *modulului de top* datele necesare pentru operațiile ce se doresc a fi executate. În Verilog, acest lucru se face, de obicei, în interiorul unui bloc `initial begin ... end`. Pentru modulul nostru, *SM4 Top Module* operează pe o interfață *APB*, aşadar pasarea datelor se face prin logica de *APB primary*, ce este prezentată în **Anexa 2**, deoarece codul este destul de lung, iar mai jos se poate găsi modul în care funcția respectivă este apelată pentru rularea testului efectiv:

```

1 initial begin
2
3     for (test_idx = 0; test_idx <= 11; test_idx++) begin
4
5         reset_module();
6
7         start_new_operation(.requested_operation(ENCRYPTION),
8                             .master_key(vectors[test_idx].key),
9                             .input_text(vectors[test_idx].plain),
10                            .init_vector(vectors[test_idx].iv),
11                            .mode_of_operation(vectors[test_idx].moo));
12
13         reset_module();
14
15         start_new_operation(.requested_operation(DECRYPTION),
16                             .master_key(vectors[test_idx].key),
17                             .input_text(vectors[test_idx].cypher),
18                             .init_vector(vectors[test_idx].iv),
19                             .mode_of_operation(vectors[test_idx].moo));
20
21         // ...
22
23     end
24
25     $stop;
26
27 end

```

- O cale de semnalizare, de obicei prin mesaje scrise în *consola* programului folosit, rezultatul

acestor operații. În cazul nostru, se folosesc *două tipuri* de mesaje, unul pentru rezultatul criptării la nivel de cuvânt de 128 de biți:

```

1 always @(posedge read_encryption_res) begin
2     #1;
3
4     if (cypher_text_enc == vectors[test_idx].cypher[i]) begin
5
6         encryption_res_wrong = 0;
7
8         $display(vectors[test_idx].moo.name(),
9             ":_Word_number_%0d/%0d_ENCRYPTION_CORRECT_at_time_%0d.", i+1,
10            vectors[test_idx].plain.size(), $time-1);
11
12    end else begin
13
14        encryption_res_wrong = 1;
15
16        $display(vectors[test_idx].moo.name(),
17             ":_Word_number_%0d/%0d_ENCRYPTION_WRONG_at_time_%0d.", i+1,
18            vectors[test_idx].plain.size(), $time-1);
19
20    end
21 end
22
23 always @(posedge read_decryption_res) begin
24     #1;
25
26     if (decrypted_text == vectors[test_idx].plain[i]) begin
27
28         decryption_res_wrong = 0;
29
30         $display(vectors[test_idx].moo.name(),
31             ":_Word_number_%0d/%0d_DECRIPTION_CORRECT_at_time_%0d.", i+1,
32            vectors[test_idx].cypher.size(), $time-1);
33
34    end else begin
35
36        decryption_res_wrong = 1;
37
38        $display(vectors[test_idx].moo.name(),
39             ":_Word_number_%0d/%0d_DECRIPTION_WRONG_at_time_%0d.", i+1,
40            vectors[test_idx].cypher.size(), $time-1);
41
42    end
43 end

```

si unul pentru contorizarea rezultatului criptării la nivel de mesaj intreg:

```

1 if (enc_wrong_flag == 0 && requested_operation == ENCRYPTION)

```

```

2   $display("DONE_", vectors[test_idx].moo.name(),
3   ":_Full_message_with_index_%0d_has_been_CORRECTLY_ENCRYPTED.", test_idx);
4
5 else if (enc_wrong_flag == 1 && requested_operation == ENCRYPTION)
6   $display("DONE_", vectors[test_idx].moo.name(),
7   ":_Full_message_with_index_%0d_has_been_WRONGLY_ENCRYPTED.", test_idx);
8
9 if (dec_wrong_flag == 0 && requested_operation == DECRYPTION)
10  $display("DONE_", vectors[test_idx].moo.name(),
11  ":_Full_message_with_index_%0d_has_been_CORRECTLY_DECRYPTED.", test_idx);
12
13 else if (dec_wrong_flag == 1 && requested_operation == DECRYPTION)
14  $display("DONE_", vectors[test_idx].moo.name(),
15  ":_Full_message_with_index_%0d_has_been_WRONGLY_DECRYPTED.", test_idx);

```

În cazul rulării acestui test, aşa cum am arătat mai sus, se vor rula toți cei 12 vectori de test inițializați, iar rezultatul din consolă arată ca mai jos, asigurându-ne astfel ca operațiile sunt efectuate corect de către modul:

```

ECB: Word number 1/1 ENCRYPTION CORRECT at time 4030.
DONE ECB: Full message with index 0 has been CORRECTLY ENCRYPTED.
ECB: Word number 1/1 DECRYPTION CORRECT at time 12650.
DONE ECB: Full message with index 0 has been CORRECTLY DECRYPTED.
ECB: Word number 1/1 DECRYPTION CORRECT at time 14010.
DONE ECB: Full message with index 0 has been CORRECTLY DECRYPTED.
ECB: Word number 1/1 ENCRYPTION CORRECT at time 19870.
DONE ECB: Full message with index 0 has been CORRECTLY ENCRYPTED.

// ...

CTR: Word number 1/4 ENCRYPTION CORRECT at time 807050.
CTR: Word number 2/4 ENCRYPTION CORRECT at time 812710.
CTR: Word number 3/4 ENCRYPTION CORRECT at time 818330.
CTR: Word number 4/4 ENCRYPTION CORRECT at time 823950.
DONE CTR: Full message with index 11 has been CORRECTLY ENCRYPTED.
CTR: Word number 1/4 ENCRYPTION CORRECT at time 829930.
CTR: Word number 2/4 ENCRYPTION CORRECT at time 835550.
CTR: Word number 3/4 ENCRYPTION CORRECT at time 841170.
CTR: Word number 4/4 ENCRYPTION CORRECT at time 846790.
DONE CTR: Full message with index 11 has been CORRECTLY ENCRYPTED.

```


5. Implementarea pe FPGA

5.1. Boolean Board FPGA development board - caracteristici generale

Ce este un development board cu FPGA?

Un *development board* (în română: *placă de dezvoltare*) aduce într-un singur produs:

1. **Circuitul programabil** – un FPGA ce poate fi configurat în mii de circuite logice diferite;
2. **Periferice minime** (oscilator, LED-uri, butoane, memorii de configurare) pentru testarea rapidă a design-urilor;
3. **Conectori de expansiune** prin care se pot conecta senzori, afișaje, servo-uri sau plăcuțe *Pmod*.

Un astfel de ansamblu permite ca descrierile hardware scrise în *Verilog/VHDL* să fie sintetizate și încărcate în timp scurt, obținând prototipuri funcționale fără cablaj suplimentare. În consecință, pe lângă utilizarea amatoare, plăcile de dezvoltare sunt folosite de către proiectanții de hardware pentru simularea practică a design-ului lor, fără să fie nevoiți să suporte costul trecerii directe în siliciu.

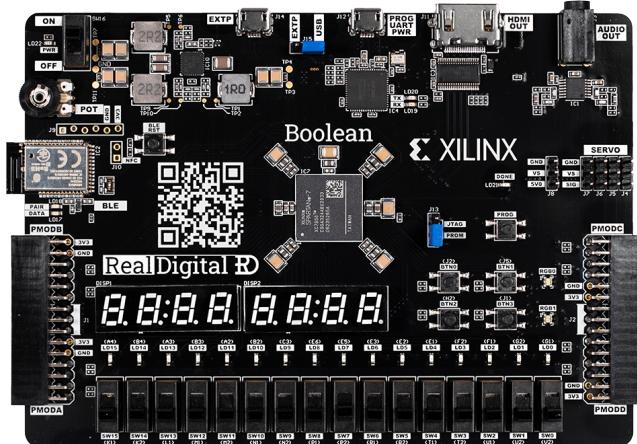


Figura 5.1.1: Poză cu Boolean Board de la Xilinx

Arhitectura FPGA Spartan-7

Placa *Boolean Board* integrează FPGA-ul XC7S50, un dispozitiv de clasă „low-cost / low-power” din familia 7-Series, deținută de *AMD*. Acestea sunt specificațiile acestuia:

Resursă	Valoare XC7S50
Celule logice	~52 000
Slice-uri (4 LUT6 + 8 FF)	~8 150
Blocuri DSP48A1	120
Block RAM	2.7 Mbit \approx 337 kB
Distributed RAM	~600 kbit
CMT (MMCM + PLL)	5
ADC intern (XADC)	12 bit @ 1 MSPS
Linii I/O maxime	~100

Tabela 5.1.1: Resurse relevante ale FPGA-ului *Spartan-7 XC7S50*

Dispozitivul este fabricat în tehnologia de 28 nm și include funcții hardware de criptare AES pentru securizarea *bitstream-ului* și detecție ECC în blocurile RAM.

Resursele plăcii *Boolean Board*

- **I/O de bază** — 16 comutatoare, 4 butoane, 16 LED-uri monocrome și 2 LED-uri *RGB*; toate conectate direct la pini FPGA pentru laboratoare vizuale rapide.
- **Afișaj** — afișaj *seven-segment* cu 8 digiți, suficient pentru a prezenta numere hexazecimale sau mesaje scurte.
- **Ceas de sistem** — oscilator MEMS de 100 MHz (± 50 ppm) conectat pe pin MRCC, gata să fie distribuit intern prin *Clock Management Tiles*.
- **Audio & Video** — amplificator stereo cu mufa 3.5 mm și ieșire *HDMI*; suficiente pentru proiecte multimedia 720p sau instrumente audio digitale.
- **Comunicație** — port micro-USB unic pentru alimentare, JTAG și UART 115 200 bps; opțional modul Bluetooth LE bazat pe *nRF52832*.
- **Control motoare** — 4 conectori servo (3-pin, PWM), alimentabili separat prin jumper J8.
- **Expansiune** — două headere de 30 pini, compatibile cu două Pmod-uri fiecare și oferind 44 I/O suplimentare, inclusiv perechi diferențiale LVDS.
- **Memorie de configurare** — flash on-board; jumperul J13 selectează pornirea automată din ROM (*PROM*) sau configurarea prin JTAG.

Fluxul de lucru cu Vivado

Placa este suportată integral de *Vivado WebPACK* (licență gratuită). Programarea se face direct via USB-JTAG, iar depanarea logicii interne se poate realiza cu *Integrated Logic Analyzer*. Pentru aplicații autonome, **bitstream-ul** poate fi scris în flash, inițializând FPGA-ul la fiecare pornire.

Aplicații practice

Datorită combinației dintre resursele FPGA și perifericele on-board, *Boolean Board* este potrivită pentru următoarele aplicații:

- laboratoare de logică digitală sau „Computer Architecture 101”;
- prototipuri multimedia (procesoare video VGA/HDMI, generatoare audio, *retro-gaming*);
- control în timp real al robotilor de mici dimensiuni, folosind servo-conectorii și LED-urile RGB ca feedback vizual;
- proiecte embedded cost-sensitive unde un microcontroler ar fi prea limitat, iar un SoC prea scump.

În concluzie, **Boolean Board** oferă un raport preț/performanță excelent: un FPGA *Spartan-7* suficient de mare pentru sisteme complexe, alături de o gamă variată de periferice ce fac posibilă trecerea de la idee la prototip în doar câteva ore.

5.2. Necesitatea sintetizabilității codului de SystemVerilog pentru implementarea pe FPGA

Procesul de proiectare nu se încheie după ce un bloc RTL trece teste de simulare. Pentru a rula pe un FPGA, descrierea hardware trebuie să poată fi **sintetizată** și apoi **implementată** (plasată și rutată). Acești doi pași transformă logica scrisă în *SystemVerilog* într-o rețea fizică de celule configurate și interconectate în siliciu.

Sinteză vs. simulare

- **Simulare** – rulează un model comportamental al designului; orice construcție permisă de limbaj este acceptată dacă simulatorul o poate interpreta; foarte similară cu *compilarea* din programarea clasică.
- **Sinteză** – convertește doar un subset „hardware-realizabil” al limbajului în elemente logice concrete (latch-uri, flip-flop-uri, LUT-uri, RAM, DSP). Prin urmare, **un cod simulabil nu este automat și sintetizabil**.

Regulă practică: înainte de trecerea la implementare, pe lângă toate erorile apărute, *toate* mesajele de tip **CRITICAL WARNING** trebuie eliminate sau justificate; ele indică adesea construcții nesuportate, pachete neconectate ori condiții de risc pentru cronometrare.

Principalele condiții pentru un cod sintetizabil

Textul de mai jos rezumă cele mai frecvente cerințe. După fiecare punct am oferit câte un exemplu practic din codul folosit pentru SM4, deoarece toate acestea au fost provocări reale pentru realizarea design-ului:

1. **Folosirea construcțiilor `always_ff`, `always_comb` și `assign`:** Se evită initial în logica de proces și utilizarea de task-uri în căile de date, deoarece acestea nu se mapează în hardware.

```
1 module SM4_Top_Module (input logic rst_n_i,
2                         input logic start_enc_i,
3                         input logic start_dec_i,
4                         input bit [2:0] mode_of_operation_i,
5                         apb_interface.secondary top_if,
6                         input logic message_done_i,
7                         output logic encryption_res_valid_o,
8                         output logic decryption_res_valid_o
9 );
10
11 // ...
12
13
14 always @(posedge top_if.sysclk_i or negedge rst_n_i) begin
15
16     if (!rst_n_i) begin
17
18         // ...
19
20     end else begin
21
22         // ...
23
24     end
25
26 end
27
28 endmodule
```

2. **Un singur bloc secvențial pe domeniu de ceas:** Toate flip-flop-urile care împart același semnal de ceas ar trebui declarate într-un singur `always_ff @(posedge clk)`; astfel se evită problemele de *mixed edge* și se simplifică generarea constrângerilor. De exemplu, **nu** este permis:

```
1 always @(posedge top_if.sysclk_i or negedge rst_n_i) begin
2
3     if (!rst_n_i) begin
4
5         // ...
6
7     end
```

```

8
9     end
10
11    always @(master_key_i) begin
12
13        // ...
14
15    end

```

3. **Stocarea valorilor în structuri statice:** FPGA-urile nu acceptă alocare dinamică (nu există heap). Dimensiunile vectorilor și ale memoriilor trebuie să fie *cunoscute la compilare*. Adică, putem avea:

```

1 typedef struct { logic [127:0] key;
2                     bit [511:0] plain;
3                     bit [511:0] cypher;
4                     bit [1:0] length;
5                     bit [127:0] iv;
6                     modes_of_op moo;
7 } test_vector;
8
9 test_vector vectors[10:0];

```

dar **nu** putem avea:

```

1 typedef struct { logic [127:0] key;
2                     bit [127:0] plain [];
3                     bit [127:0] cypher [];
4                     bit [1:0] length;
5                     bit [127:0] iv;
6                     modes_of_op moo;
7 } test_vector;
8
9 test_vector vectors[10:0];

```

4. **Evitarea semnalelor *multi-driven*:** Un singur driver combinatorial sau sevențial pe netă; altfel, sintetizatorul creează un *wired-OR* nedorit sau generează o **eroare**. Din acest motiv, o parte din semnale trebuie să fie *multiplexate* la nivel superior, în acest mod:

```

1 assign rk_if_mem.paddr_i = (encryption_dir == 2'b11) ? rk_if_ec.paddr_i :
2                                         (encryption_dir == 2'b10) ? rk_if_dc.paddr_i : 5'd0;
3 assign rk_if_mem.pwrite_i = (encryption_dir == 2'b11) ? rk_if_ec.pwrite_i :
4                                         (encryption_dir == 2'b10) ? rk_if_dc.pwrite_i : 1'd0;
5 assign rk_if_mem.psel_i = (encryption_dir == 2'b11) ? rk_if_ec.psel_i :
6                                         (encryption_dir == 2'b10) ? rk_if_dc.psel_i : 1'd0;
7 assign rk_if_mem.penable_i = (encryption_dir == 2'b11) ? rk_if_ec.penable_i :
8                                         (encryption_dir == 2'b10) ? rk_if_dc.penable_i : 1'd0;
9 assign rk_if_mem.pwdata_i = (encryption_dir == 2'b11) ? rk_if_ec.pwdata_i :
10                                         (encryption_dir == 2'b10) ? rk_if_dc.pwdata_i : 32'd0;

```

Respectarea acestor ghiduri garantează că designul RTL se poate transforma fără surprize într-un *bitstream* valid și că diferențele dintre simulare și funcționarea pe siliciu sunt reduse la minimum.

5.3. Design Test Wrapper: implementarea unui modul de test sintetizabil

Acum că am definit necesitatea unui cod sintetizabil, putem **concluziona** faptul că *testbench*-ul prezentat anterior este departe de a putea fi un *top* de test pentru design-ul nostru de **SM4**, din cauza a mai multor încălcări evidente ale regulilor tocmai enumerate. În acest caz, vom avea nevoie să scriem un **modul nou de test, pentru FPGA**, care să respecte aceste reguli, pentru a putea vedea funcționalitatea modulului de criptare pe plăcuța noastră.

Modulul **Design Test Wrapper** are drept scop să ocupe acest rol, folosind un design similar cu cel al modulelor *SM4 Top Module* și *SM4 Core*, bazat pe **automate sincrone**, însă mult mai complex, deoarece acesta se folosește de patru asemenea automate, implementate într-un mod ierarhic. Această implementare prevede că, pentru a se schimba starea unui *FSM* de pe nivel superior, trebuie să se treacă întâi prin toate stările celui de nivel inferior care, la rândul lor, se schimbă numai după trecerea prin toate stările celui de nivel inferior lui, *et cetera*. Ierarhia automatelor se poate observa mai jos:

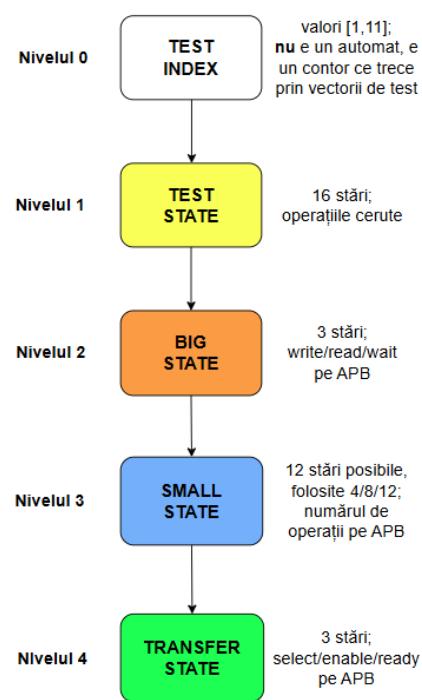


Figura 5.3.1: Ierarhia de automate din Design Test Wrapper

Însumarea acestor automate face ca, atunci când un **stimul extern** de pe placă este aplicat, toate testele definite în *wrapper* să fie rulate în ordinea stabilită, având pe parcurs și un mecanism de menținere a „scorului”, ce va fi afișat pe FPGA, explicit:

- Numărul de *criptări CORECTE* / Nr. de *criptări TOTALE*
- Numărul de *decriptări CORECTE* / Nr. de *decriptări TOTALE*

Deși codul pentru *Design Test Wrapper* este de dimensiuni foarte mari și va fi prezentat doar în **Anexa 3**, putem vizualiza, pe rând, cele patru automate, începând cu cel pentru *Test State*:

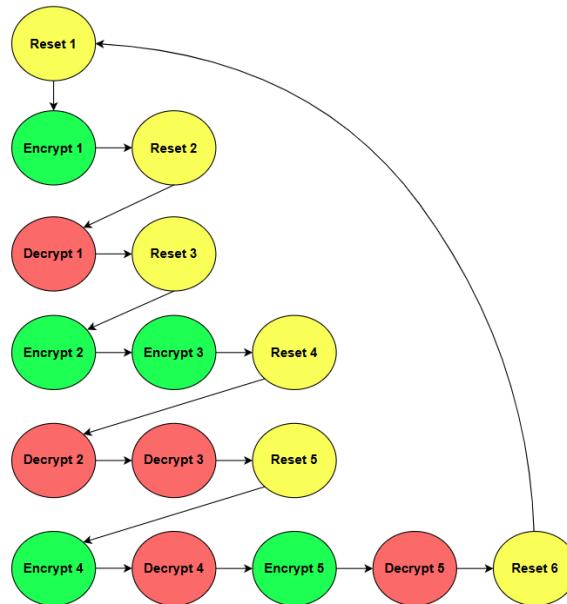


Figura 5.3.2: FSM-ul pentru Test State

Așa cum am explicitat și anterior, acest FSM acționează mai mult ca o secvență de teste. Această secvență pre-aleasă asigură că, pentru fiecare vector de test, se vor testa toate variantele relevante din punctul de vedere al diferențelor de design:

- **Criptare imediat după reset.**
- **Decriptare imediat după reset.**
- **Criptare consecutivă** – inițiată după o operație anterioară (criptare sau decriptare).
- **Decriptare consecutivă** – inițiată după o operație anterioară (criptare sau decriptare).

Automatul numit destul de generic „*Big State*”, ce diferențiază tipul operației de pe APB, poate fi vizualizat mai jos:

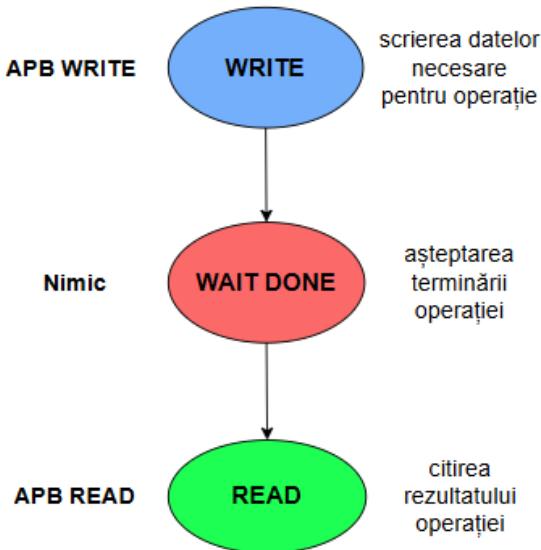


Figura 5.3.3: FSM-ul pentru Big State

Acesta este automatul ce descrie *flow-ul* unei operații de criptare/decriptare: de la scrierea ope ranzilor pe interfață către modulul de SM4 Top Module, la așteptarea flag-ului de done, la citirea rezultatului final.

Cât despre ultimele două automate:

- **Automatul *Small State*** numără câte operații consecutive de același tip trebuie făcute pe APB. Acest număr trebuie să fie:
 - **12** pentru prima serie de scieri atunci când modul de operare **nu** este *ECB* (4 pentru vectorul de inițializare, 4 pentru cheie și 4 pentru text);
 - **8** pentru prima serie de scieri în *ECB* (fără cele 4 scieri pentru vectorul de inițializare);
 - **4** pentru toate seriile ulterioare de scieri și pentru citiri (doar textul, fie el mesaj sau rezultat al operației).
- **Automatul *Transfer State*** realizează trecerea între cele trei stări non-IDLE în care se poate afla APB: *SELECT*, *ENABLE* și *READY*. Această distincție este necesară pentru a seta corect semnalele pe interfața de *top*, iar cea mai simplă implementare este prin acest mic automat.

5.4. Wrapper Testbench pentru testarea modulului și rezultatele obținute pe placă

Display Wrapper și transcodarea pentru ecranul cu 7 segmente

Așa cum am menționat anterior, placa de dezvoltare utilizată, la fel ca majoritatea *development board-urilor* cu FPGA, oferă un număr relativ restrâns de opțiuni pentru **vizualizarea rezultatelor**. Pentru scopul acestui proiect am decis să folosim cele **două afișaje LED cu șapte segmente**, pe care vom indica, respectiv:

- numărul total de *criptări* executate corect / numărul total de criptări;
- numărul total de *decriptări* executate corect / numărul total de decriptări.

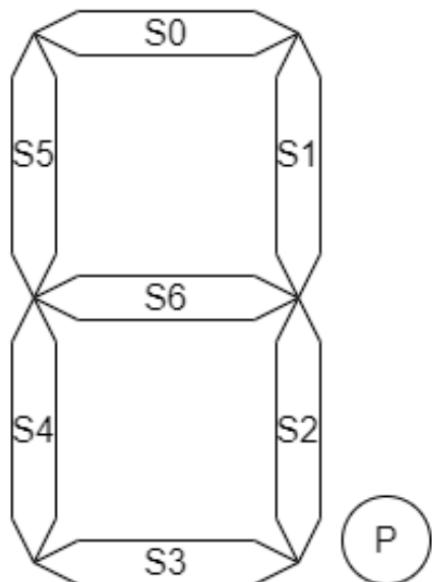


Figura 5.4.1: Funcționarea unui ecran LED cu 7 segmente

Putem observa mai sus împărțirea pe segmente pentru aceste tipuri de ecrane; fiecare dintre aceste segmente reprezintă *un bit* ce trebuie aprins individual, iar cifra/cifrele ce se doresc a fi aprinse trebuie „selectate” folosind un alt set de **4 pini pentru selecție**. Rezultă, deci, că pentru a afișa valorile dorite avem nevoie de două module auxiliare. Primul dintre ele este un **transcodor de date** dedicat numerelor de două cifre, capabil să transforme reprezentarea binară internă în codul necesar iluminării segmentelor, prezentat mai jos:

```
1 module Display_7Segx2_Transcoder(input logic [6:0] number_in,
2                                     output logic [13:0] display_out);
3
4   always_comb begin
5
6     if (number_in > 99) begin
7
8       display_out = 14'b11111111111111;
9
10    end else begin
11
```

```

12      case(number_in%10)
13
14          4'd0: display_out[6:0] = 7'b1000000;
15          4'd1: display_out[6:0] = 7'b1111001;
16          4'd2: display_out[6:0] = 7'b0100100;
17          4'd3: display_out[6:0] = 7'b0110000;
18          4'd4: display_out[6:0] = 7'b0011001;
19          4'd5: display_out[6:0] = 7'b0010010;
20          4'd6: display_out[6:0] = 7'b0000010;
21          4'd7: display_out[6:0] = 7'b1111000;
22          4'd8: display_out[6:0] = 7'b0000000;
23          4'd9: display_out[6:0] = 7'b0010000;
24      default: display_out[6:0] = 7'b1111111;
25
26  endcase
27
28  case(number_in/10)
29
30      4'd0: display_out[13:7] = 7'b1000000;
31      4'd1: display_out[13:7] = 7'b1111001;
32      4'd2: display_out[13:7] = 7'b0100100;
33      4'd3: display_out[13:7] = 7'b0110000;
34      4'd4: display_out[13:7] = 7'b0011001;
35      4'd5: display_out[13:7] = 7'b0010010;
36      4'd6: display_out[13:7] = 7'b0000010;
37      4'd7: display_out[13:7] = 7'b1111000;
38      4'd8: display_out[13:7] = 7'b0000000;
39      4'd9: display_out[13:7] = 7'b0010000;
40  default: display_out[13:7] = 7'b1111111;
41
42  endcase
43
44 end
45
46 end
47
48 endmodule

```

Al doilea modul de care avem nevoie va prelua chiar rolul de top pentru această implementare, și este necesar pentru a corela toate cele 8 cifre dorite (4 numere de 2 cifre) cu semnalele de selecție corespunzătoare lor:

```

1 module Display_Wrapper(input logic sysclk_i,
2                         input logic rst_tests_button_i,
3                         input logic run_full_button_i,
4                         output logic [3:0] sel_left_o,
5                         output logic [3:0] sel_right_o,
6                         output logic [7:0] segments_left_o,
7                         output logic [7:0] segments_right_o);

```

```

8
9
10    logic [27:0] display_left;
11    logic [27:0] display_right;
12    reg state;
13    reg [1:0] count;
14    logic [19:0] freq_divider;
15
16    assign segments_left_o[7:1] = display_left[27:21];
17    assign segments_right_o[7:1] = display_right[27:21];
18    assign segments_left_o[0] = (sel_left_o == 4'b1011) ? 0:1;
19    assign segments_right_o[0] = (sel_right_o == 4'b1011) ? 0:1;
20
21    Design_Test_Wrapper DTW1(.sysclk_i(sysclk_i),
22                            .rst_tests_button_i(rst_tests_button_i),
23                            .run_full_button_i(run_full_button_i));
24
25    ila_1 ILA2(.clk(sysclk_i),
26                .probe0(sel_left_o),
27                .probe1(sel_right_o),
28                .probe2(segments_left_o),
29                .probe3(segments_right_o),
30                .probe4(DTW1.left1_counter_value),
31                .probe5(DTW1.correct_enc_cnt),
32                .probe6(DTW1.right1_counter_value),
33                .probe7(DTW1.correct_dec_cnt));
34
35
36    always @(posedge sysclk_i) begin
37
38        case (state)
39
40            0: begin
41
42                sel_left_o <= 4'b0111;
43                sel_right_o <= 4'b0111;
44                display_left <= {DTW1.display4_o, DTW1.display3_o};
45                display_right <= {DTW1.display2_o, DTW1.display1_o};
46                count <= 0;
47                freq_divider <= 0;
48                state <= 1;
49
50            end
51
52            1: begin
53
54                if(count == 3 && freq_divider == 200000)
55                    state <= 0;
56                else if (freq_divider == 200000) begin
57                    count <= count + 1;

```

```

58         freq_divider <= 0;
59     end
60
61     if (freq_divider == 199999) begin
62
63         display_left <= display_left << 7;
64         display_right <= display_right << 7;
65         sel_left_o <= {1'b1, sel_left_o[3:1]};
66         sel_right_o <= {1'b1, sel_right_o[3:1]};
67
68     end
69
70     if (freq_divider != 200000)
71         freq_divider <= freq_divider + 1;
72
73     end
74
75     default : state <= 0;
76
77 endcase
78
79 end
80
81 endmodule

```

În această implementare, putem observa exact logica pe care am descris-o anterior. Cele **8 cifre** sunt scrise alternativ, pe rând, în timp ce toate celelalte sunt opriate. Această metodă, la frecvențe mari, cum este cazul aici, *păcălește ochiul* să vadă afişajul ca fiind aprins constant. Iese în evidență, de asemenea, folosirea unui **divizor de frecvență** pentru aceste scrieri, artificiu folosit pentru a reduce frecvența foarte mare a ceasului integrat (*100 MHz*), la o valoare mai mică și, implicit, pentru a duce la o afișare mai stabilă.

Adițional, se observă folosirea unui **Integrated Logic Analyzer** (*Analizor Integrat de Logică*), ce este folosit în scopuri de *debugging*, pentru a vedea *in real time* cum arată valorile semnalelor relevante pe FPGA.

Wrapper testbench

Urmărind aceeași motivație prezentată în **Capitolul 4.9**, înainte de implementarea pe plăcuță fizică, primul pas va fi testarea modulului în simulatorul Vivado. Pentru aceasta, se va folosi un testbench mic și simplu de implementat, ce simulează apasarea butoanelor de pe FPGA, și vom observa funcționarea corectă în fereastra de waveform:

```

1 `timescale 1ns / 1ps
2
3 module wrapper_tb();

```

```

4
5 logic sysclk_i;
6 logic rst_tests_button_i;
7 logic run_full_button_i;
8 logic [13:0] display_o [3:0];
9 logic [6:0] segments_left_o;
10 logic [6:0] segments_right_o;
11 logic [3:0] sel_left_o;
12 logic [3:0] sel_right_o;
13
14 Display_Wrapper DW1(.sysclk_i(sysclk_i),
15                         .rst_tests_button_i(rst_tests_button_i),
16                         .run_full_button_i(run_full_button_i),
17                         .run_single_button_i(run_single_button_i),
18                         .sel_left_o(sel_left_o),
19                         .sel_right_o(sel_right_o),
20                         .segments_left_o(segments_left_o),
21                         .segments_right_o(segments_right_o));
22
23 initial begin
24     sysclk_i = 0;
25     forever #10 sysclk_i = ~sysclk_i;
26 end
27
28
29 initial begin
30
31     rst_tests_button_i = 1;
32     run_full_button_i = 0;
33     run_single_button_i = 0;
34
35     #20;
36
37     rst_tests_button_i = 0;
38
39     #20;
40
41     run_full_button_i = 1;
42
43     #50000000;
44
45     $stop;
46
47 end
48
49 endmodule

```

După rularea acestui cod, vom observa pe forma de undă din Vivado:

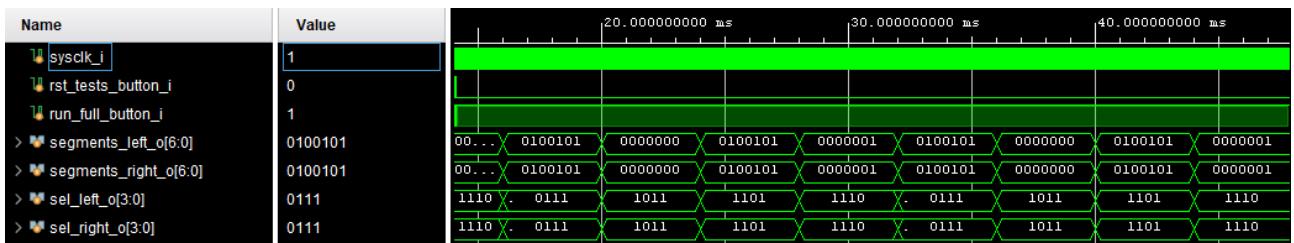


Figura 5.4.2: Forma de undă pentru Display Wrapper

Putem distinge că, după un **reset** și după o apăsare ulterioară a **butonului de începere a testului**, la un moment dat, după terminarea testului, încep aceste *scrieri alternative* despre care am vorbit anterior, iar valorile afișate sunt secvența **5 0 5 0** pe ambele ecrane, corespunzător mesajului:

50/50 criptări corecte și 50/50 decriptări corecte.

Poze cu implementarea fizică

Desigur, deoarece am avut grijă să respectăm toate cerințele de sintetizare, și codul nostru este corect, vom obține același lucru și pe plăcuță. Explicit, în timpul unui reset, vedem:

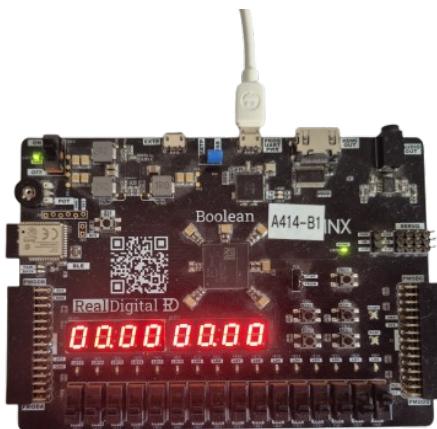


Figura 5.4.3: Poză cu FPGA-ul în Reset

Așa cum ne-am așteptă, avem valoarea 0 pe peste tot. Acum, apăsăm butonul pentru rularea testelor:

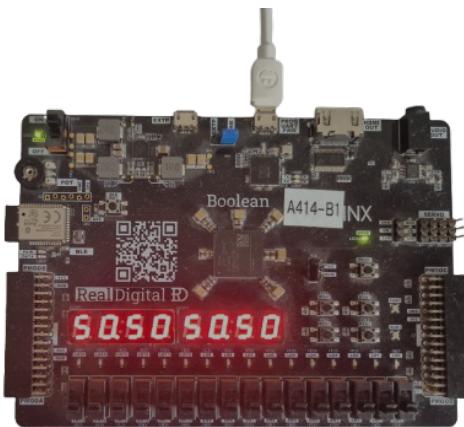


Figura 5.4.4: Poză cu FPGA-ul după apăsarea butonului, cu codul corect

Și în ultimul rând, vom verifica că nu avem de a face cu o autosugestie a faptului că codul încărcat este cel bun, schimbând un bit de la cheia primului test, pentru a vedea ce obținem:

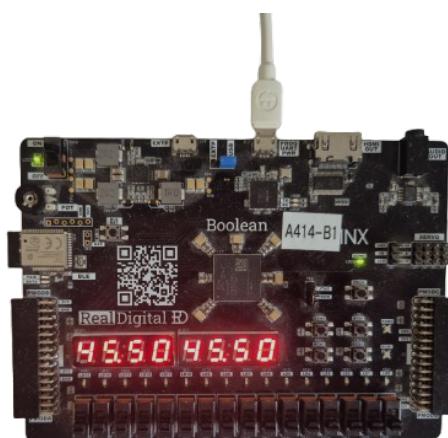


Figura 5.4.5: Poză cu FPGA-ul după apăsarea butonului, cu codul alterat

Așa cum ne-am așteptă, obținem 45/50 pentru ambele ecrane, număr obținut din faptul că toate cele 5 cripări și toate cele 5 decriptări pentru primul vector de test sunt acum greșite. Avem, deci, confirmarea faptului că am făcut cu succes implementarea modulului SM4 pe FPGA.

Concluzii

În concluzie, proiectarea și validarea unui modul de *SM4* pe FPGA s-au dovedit nu doar posibile, ci și practică pentru sisteme embedded moderne. Am demonstrat că, prin respectarea strictă a regulilor de *sintetizabilitate*: un singur bloc secvențial pe ceas, evitarea alocării dinamice, eliminarea CRITICAL WARNING-urilor și utilizarea disciplinată a `always_ff/always_comb`, logica criptografică poate fi transpusă fără surprize din simulare în siliciu. Vectorii de test oficiali, furnizați în draft-ul standardului *SM4*, au permis o verificare exhaustivă în simulator, în timp ce **Integrated Logic Analyzer**-ul on-chip a confirmat, în *real-time*, corectitudinea celor cinci moduri de operare direct pe placa *Boolean Board* cu Spartan-7.

Implementarea ierarhică, bazată pe **FSM-uri sincrone**, împreună cu interfața *APB* pe 32 de biți, conferă proiectului reutilizabilitate și compatibilitate cu o gamă largă de produse. Memoria dedicată cheilor de rundă, registrii interni pentru toate variabilele necesare operațiilor și mecanismele pentru implementarea modurilor de operare au permis criptarea/decriptarea fluxurilor de date de orice lungime, păstrând totodată un consum redus de resurse: sub 30% din LUT-uri și RAM-B pe XC7S50. Afisarea rezultatelor pe cele două display-uri cu șapte segmente (*50/50 criptări corecte, 50/50 decriptări corecte*) și controlul lor prin multiplexare rapidă au validat atât partea funcțională, cât și infrastructura de I/O a plăcii.

Direcții viitoare pot include, desigur, sporirea cunoștințelor în aria algoritmilor criptografici moderni, și implementarea SoTA-ului pe plăci cu mult mai complexe decât Boolean Board-ul folosit acum, în scopul sporirii adiționale a securității și a performanțelor sistemelor criptografice. De asemenea, de interes poate fi și testarea acestui tip de implementare într-un scenariu aplicabil în viața de zi-cu-ză, cu date transmise în timp real, pentru înțelegerea mai profundă a sistemelor criptografice curente și, ulterior, pentru creșterea capacitaților curente de securizare a datelor.

Bibliografie

- [1] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography*, ver. 0.6. [Online]. Available: https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_6.pdf.
- [2] Cryptography Academy, “*SM4 Chinese Block Cipher Explained*,” YouTube, 12 mar. 2023. [Video fișier online]. Available: https://youtu.be/GSIDS_1vRv4.
- [3] Neso Academy, “*Cryptography and Network Security – Full Course Playlist*,” 2022. [Online]. Available: <https://www.nesoacademy.org/cs/11-cryptography-and-network-security>.
- [4] Computer Science Hub, “*Cryptography Course – 40 lectures*,” YouTube playlist, 2021. [Online]. Available: <https://www.youtube.com/playlist?list=PL6N5qY2nvvJE8X75VkXg1SrVhLv1tVcfy>.
- [5] Chaitin Tech, “*SM4 Implementation on FPGA*,” YouTube, 18 apr. 2024. [Video fișier online]. Available: <https://youtu.be/xL9QWDRTzvA>.
- [6] Y. Dai, J. Guo, and S. Wu, “*SM4 Blockcipher Algorithm and Its Modes of Operation (Draft 00)*,” IETF Crypto Forum RG, Aug. 2019. [Online]. Available: <https://datatracker.ietf.org/doc/html/draft-crypto-sm4-00>.
- [7] Wikipedia, “*Advanced Encryption Standard*,” 2025. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.
- [8] Wikipedia, “*Data Encryption Standard*,” 2025. [Online]. Available: https://en.wikipedia.org/wiki/Data_Encryption_Standard.
- [9] Microsoft, “*Microsoft’s Majorana-1 chip carves new path for quantum computing*,” 20 iun. 2024. [Online]. Available: <https://news.microsoft.com/source/features/innovation/microsofts-majorana-1-chip-carves-new-path-for-quantum-computing/>.
- [10] ChipVerify, “*SystemVerilog Tutorials – Complete Guide*,” 2023. [Online]. Available: <https://www.chipverify.com/tutorials/systemverilog>.
- [11] Facultatea ETTI, UPB, “*Circuite Integrale Digitale – Curs online*,” 2022. [Online]. Available: https://wiki.dcae.pub.ro/index.php/Circuite_Integrate_Digitale.
- [12] RealDigital, “*Boolean Board Reference Manual*,” 2023. [Online]. Available: <https://www.realdigital.org/doc/02013cd17602c8af749f00561f88ae21>.
- [13] State Cryptography Administration of China, “*GM/T 0002-2012 – SM4 Block Cipher Algorithm*,” 2012. [Online]. Available: <https://www.oscca.gov.cn/sca/xxgk/2013-01/26/1002386/files/4d4f0bb637564dba9b37ad3a26f427af.pdf>.
- [14] National Institute of Standards and Technology, “*FIPS 197 – Advanced Encryption Standard (AES)*,” Nov. 2001. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>.
- [15] IEEE Standards Association, “*IEEE Std 1800-2017 – SystemVerilog Language Reference Manual*,” 2018.

- [16] AMD Xilinx, “*UG470: 7 Series FPGAs SelectIO Resources*,” v1.13, Feb. 2023. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_SelectIO.pdf.
- [17] AMD Xilinx, “*UG901: Vivado Design Suite – High-Level Synthesis*,” v2024.1, Apr. 2024. [Online]. Available: <https://docs.xilinx.com/v/u/en-US/ug901-vivado-hls>.

Anexa 1 – Codul pentru SM4 Top Module

```
1 module SM4_Top_Module (input logic rst_n_i,
2                         input logic start_enc_i,
3                         input logic start_dec_i,
4                         input bit [2:0] mode_of_operation_i,
5                         apb_interface.secondary top_if,
6                         input logic message_done_i,
7                         output logic encryption_res_valid_o,
8                         output logic decryption_res_valid_o
9 );
10
11
12
13     logic encryption_done_o;
14     logic decryption_done_o;
15     logic start_read;
16     logic enc_req_i;
17     logic dec_req_i;
18     logic enc_in_prog;
19     logic dec_in_prog;
20     logic first_access;
21     logic data_stream_done;
22     logic wait_clock;
23     logic [127:0] init_vector;
24     logic [127:0] aux_iv;
25     logic [127:0] master_key;
26     logic [127:0] plain_text;
27     logic [127:0] cypher_text_dec;
28     logic [127:0] cypher_text_enc;
29     logic [127:0] decrypted_text;
30
31     typedef enum bit [1:0] {SEL = 0, ENABLE, READY} apb_states;
32     apb_states apb_state;
33
34     apb_interface core_if (.sysclk_i(top_if.sysclk_i));
35
36     SM4_Core SM4_Core(.rst_n_i(rst_n_i),
37                       .enc_req_i(enc_req_i),
38                       .dec_req_i(dec_req_i),
39                       .core_if(core_if.secondary),
40                       .encryption_done_o(encryption_done_o),
41                       .decryption_done_o(decryption_done_o));
42
43     always @ (posedge top_if.sysclk_i or negedge rst_n_i) begin
44
45         if (!rst_n_i) begin
```

```

46
47     enc_req_i <= 0;
48     dec_req_i <= 0;
49     enc_in_prog <= 0;
50     dec_in_prog <= 0;
51     first_access <= 0;
52     data_stream_done <= 0;
53     wait_clock <= 0;
54     start_read <= 0;
55     init_vector <= 0;
56     aux_iv <= 0;
57     master_key <= 0;
58     plain_text <= 0;
59     cypher_text_dec <= 0;
60     cypher_text_enc <= 0;
61     decrypted_text <= 0;
62     top_if.pready_o <= 0;
63     top_if.pslverr_o <= 0;
64     top_if.prdata_o <= 0;
65     core_if.paddr_i <= 0;
66     core_if.pwrite_i <= 0;
67     core_if.psel_i <= 0;
68     core_if.penable_i <= 0;
69     core_if.pwdata_i <= 0;
70
71 end else begin
72
73     top_if.pready_o <= 0;
74     top_if.pslverr_o <= 0;
75     encryption_res_valid_o <= 0;
76     decryption_res_valid_o <= 0;
77
78     if (start_enc_i == 1 && enc_in_prog == 0) begin
79
80         enc_in_prog <= 1;
81         first_access <= 1;
82         data_stream_done <= 0;
83         start_read <= 0;
84
85     end
86
87     if (start_dec_i == 1 && dec_in_prog == 0) begin
88
89         dec_in_prog <= 1;
90         first_access <= 1;
91         data_stream_done <= 0;
92         start_read <= 0;
93
94     end
95

```

```

96      if ((enc_in_prog ^ dec_in_prog == 1) && top_if.psel_i == 1 && top_if.pwrite_i == 1 &&
97          top_if.enable_i == 1 && data_stream_done == 0 && message_done_i == 0) begin
98
99          top_if.pready_o <= 1;
100         top_if.pslverr_o <= 0;
101
102        if (mode_of_operation_i == 1 && first_access == 1) begin
103
104            if (top_if.paddr_i <= 3) begin
105
106                master_key[top_if.paddr_i*32 +: 32] <= top_if.pwdata_i;
107
108            end
109
110            if (top_if.paddr_i >= 4 && top_if.paddr_i <= 7) begin
111
112                if (enc_in_prog == 1)
113                    plain_text[(top_if.paddr_i-4)*32 +: 32] <= top_if.pwdata_i;
114                else
115                    cypher_text_dec[(top_if.paddr_i-4)*32 +: 32] <= top_if.pwdata_i;
116
117            end
118
119            if (top_if.paddr_i == 7) begin
120
121                wait_clock <= 1;
122
123            end
124
125        end else if (first_access == 1) begin
126
127            if (top_if.paddr_i <= 3) begin
128
129                init_vector[top_if.paddr_i*32 +: 32] <= top_if.pwdata_i;
130
131            end
132
133            if (top_if.paddr_i >= 4 && top_if.paddr_i <= 7) begin
134
135                master_key[(top_if.paddr_i-4)*32 +: 32] <= top_if.pwdata_i;
136
137            end
138
139            if (top_if.paddr_i >= 8 && top_if.paddr_i <= 11) begin
140
141                if (enc_in_prog == 1)
142                    plain_text[(top_if.paddr_i-8)*32 +: 32] <= top_if.pwdata_i;
143                else
144                    cypher_text_dec[(top_if.paddr_i-8)*32 +: 32] <= top_if.pwdata_i;
145

```

```

146      end
147
148      if (top_if.paddr_i == 11) begin
149          wait_clock <= 1;
150
151      end
152
153  end else if (first_access == 0) begin
154
155      if (top_if.paddr_i <= 3) begin
156
157          if (enc_in_prog == 1)
158              plain_text[top_if.paddr_i*32 +: 32] <= top_if.pwdata_i;
159          else
160              cypher_text_dec[top_if.paddr_i*32 +: 32] <= top_if.pwdata_i;
161
162      end
163
164      if (top_if.paddr_i == 3) begin
165
166          wait_clock <= 1;
167
168      end
169
170  end
171
172  end else if (wait_clock == 1) begin
173
174      wait_clock <= 0;
175      data_stream_done <= 1;
176      core_if.paddr_i <= 5'b11111;
177      apb_state <= SEL;
178
179  end else if (enc_in_prog == 1 && dec_in_prog == 1) begin
180
181      top_if.pslverr_o <= 1;
182
183  end else if (message_done_i == 1 && top_if.psel_i == 1 && top_if.penable_i == 1) begin
184
185      top_if.pslverr_o <= 1;
186
187  end else if (data_stream_done == 1 && (core_if.paddr_i <= 7 || core_if.paddr_i ==
188 5'b11111) && message_done_i == 0) begin
189
190      if (mode_of_operation_i inside {[1:2]}) begin
191
192          if (enc_in_prog == 1)
193              enc_req_i <= 1;
194          else if (dec_in_prog == 1)

```

```

196         dec_req_i <= 1;
197
198     end else if (mode_of_operation_i inside {[3:5]}) begin
199
200         enc_req_i <= 1;
201
202     end
203
204     if (mode_of_operation_i inside {[2:4]} && first_access == 1) begin
205
206         aux_iv <= init_vector;
207
208     end else if (mode_of_operation_i == 3 && enc_in_prog == 1) begin
209
210         aux_iv <= cypher_text_enc;
211
212     end
213
214     if (apb_state == SEL) begin
215
216         core_if.psel_i <= 1;
217         core_if.penable_i <= 0;
218         core_if.paddr_i += 1;
219         core_if.pwrite_i <= 1;
220
221         if (core_if.paddr_i <= 3) begin
222
223             core_if.pwdata_i <= master_key[core_if.paddr_i*32 +: 32];
224
225         end else if (core_if.paddr_i <= 7 && core_if.paddr_i >= 4) begin
226
227             if (mode_of_operation_i == 1) begin
228
229                 if (enc_in_prog == 1)
230                     core_if.pwdata_i <= plain_text[(core_if.paddr_i-4)*32 +: 32];
231                 else if (dec_in_prog == 1)
232                     core_if.pwdata_i <= cypher_text_dec[(core_if.paddr_i-4)*32 +: 32];
233
234             end
235
236             if (mode_of_operation_i == 2) begin
237
238                 if (enc_in_prog == 1)
239                     core_if.pwdata_i <= plain_text[(core_if.paddr_i-4)*32 +: 32] ^
240                     aux_iv[(core_if.paddr_i-4)*32 +: 32];
241                 else if (dec_in_prog == 1)
242                     core_if.pwdata_i <= cypher_text_dec[(core_if.paddr_i-4)*32 +: 32];
243
244             end
245

```

```

246         if (mode_of_operation_i == 3) begin
247             core_if.pwdata_i <= aux_iv[(core_if.paddr_i-4)*32 +: 32];
248         end
249
250         if (mode_of_operation_i == 4) begin
251             core_if.pwdata_i <= aux_iv[(core_if.paddr_i-4)*32 +: 32];
252         end
253
254         if (mode_of_operation_i == 5) begin
255             core_if.pwdata_i <= init_vector[(core_if.paddr_i-4)*32 +: 32];
256         end
257
258     end
259
260     apb_state <= ENABLE;
261
262     end
263
264     end
265
266     apb_state <= READY;
267
268 end else if (apb_state == READY) begin
269
270     core_if.penable_i <= 1;
271     apb_state <= READY;
272
273 end else if (apb_state == SEL) begin
274
275     core_if.penable_i <= 0;
276     apb_state <= SEL;
277
278 end
279
280 end else if ((encryption_done_o == 1 || decryption_done_o == 1)
281 && message_done_i == 0) begin
282
283     first_access <= 0;
284     start_read <= 1;
285     apb_state <= SEL;
286     data_stream_done <= 0;
287     dec_req_i <= 0;
288     enc_req_i <= 0;
289     core_if.paddr_i <= 5'b11111;
290
291 end else if (start_read == 1 && (core_if.paddr_i <= 3 || core_if.paddr_i == 5'b11111)
292 && message_done_i == 0) begin
293
294     if (apb_state == SEL) begin

```

```

296     if (core_if.pready_o == 1) begin
297
298         if (enc_in_prog == 1) begin
299             cypher_text_enc[core_if.paddr_i*32 +: 32] <= core_if.prdata_o;
300         end
301         else if (dec_in_prog == 1)
302             decrypted_text[core_if.paddr_i*32 +: 32] <= core_if.prdata_o;
303
304     end
305
306     core_if.psel_i <= 1;
307     core_if.penable_i <= 0;
308     core_if.paddr_i += 1;
309     core_if.pwrite_i <= 0;
310
311     apb_state = ENABLE;
312
313 end else if (apb_state == ENABLE) begin
314
315     core_if.penable_i <= 1;
316     apb_state <= READY;
317
318 end else if (apb_state == READY) begin
319
320     core_if.penable_i <= 0;
321     apb_state <= SEL;
322
323 end
324
325 end else if (start_read == 1 && (core_if.paddr_i == 4 || core_if.paddr_i == 5'b10100)
326 && message_done_i == 0) begin
327
328     if (enc_in_prog == 1)
329         encryption_res_valid_o <= 1;
330     else if (dec_in_prog == 1)
331         decryption_res_valid_o <= 1;
332
333     if (mode_of_operation_i == 2) begin
334
335         if (enc_in_prog == 1) begin
336
337             aux_iv <= cypher_text_enc;
338
339         end
340
341         else if (dec_in_prog == 1) begin
342
343             decrypted_text <= decrypted_text ^ aux_iv;
344             aux_iv <= cypher_text_dec;
345

```

```

346         end
347
348     end
349
350     if (mode_of_operation_i == 3) begin
351
352         if (enc_in_prog == 1) begin
353
354             cypher_text_enc <= cypher_text_enc ^ plain_text;
355
356         end
357
358         else if (dec_in_prog == 1) begin
359
360             decrypted_text <= decrypted_text ^ cypher_text_dec;
361             aux_iv <= cypher_text_dec;
362
363         end
364
365     end
366
367     if (mode_of_operation_i == 4) begin
368
369         if (enc_in_prog == 1) begin
370
371             aux_iv <= cypher_text_enc;
372             cypher_text_enc <= cypher_text_enc ^ plain_text;
373
374         end
375
376         else if (dec_in_prog == 1) begin
377
378             aux_iv <= decrypted_text;
379             decrypted_text <= decrypted_text ^ cypher_text_dec;
380
381         end
382
383     end
384
385
386     if (mode_of_operation_i == 5) begin
387
388         if (enc_in_prog == 1)
389             cypher_text_enc <= cypher_text_enc ^ plain_text;
390         else if (dec_in_prog == 1)
391             decrypted_text <= decrypted_text ^ cypher_text_dec;
392
393         init_vector += 1;
394
395     end

```

```

396
397     start_read <= 0;
398     data_stream_done <= 0;
399
400     end else if (top_if.psel_i == 1 && top_if.penable_i == 1 && top_if.pwrite_i == 0
401     && message_done_i == 0) begin
402
403         top_if.pready_o <= 1;
404         top_if.pslverr_o <= 0;
405
406         if (enc_in_prog == 1)
407             top_if.prdata_o <= cypher_text_enc[top_if.paddr_i*32 +: 32];
408         else if (dec_in_prog == 1)
409             top_if.prdata_o <= decrypted_text[top_if.paddr_i*32 +: 32];
410
411         end else if (message_done_i == 1) begin
412
413             init_vector <= 0;
414             enc_in_prog <= 0;
415             dec_in_prog <= 0;
416
417         end
418
419     end
420
421 end
422
423
424 endmodule

```

Anexa 2 – Codul pentru logica de primary din Simulation Testbench

```
1 task start_new_operation(req_operation_type requested_operation, input logic [127:0] master_key,
2 input bit [127:0] input_text [], input logic [127:0] init_vector, modes_of_op mode_of_operation);
3
4     enc_wrong_flag = 0;
5     dec_wrong_flag = 0;
6     read_encryption_res = 0;
7     read_decryption_res = 0;
8
9     mode_of_operation_i = mode_of_operation;
10    message_done_i = 0;
11
12    if (requested_operation == ENCRYPTION)
13        start_enc_i = 1;
14    else
15        start_dec_i = 1;
16
17    @(posedge sysclk_i);
18    start_enc_i = 0;
19    start_dec_i = 0;
20
21    j = 0;
22    i = 0;
23
24    if (mode_of_operation == ECB) begin
25
26        while (j <= 7) begin
27
28            if (j <= 3)
29                small_aux = master_key[j*32 +: 32];
30            else if (j <= 7) begin
31                small_aux = input_text[i][(j-4)*32 +: 32];
32
33            end
34
35            top_if.write(.addr(j), .wdata(small_aux));
36
37            j += 1;
38
39        end
40
41    end else begin
42
43        while (j <= 11) begin
```

```

45      if (j <= 3)
46          small_aux = init_vector[j*32 +: 32];
47      else if (j <= 7)
48          small_aux = master_key[(j-4)*32 +: 32];
49      else
50          small_aux = input_text[i][(j-8)*32 +: 32];
51
52      top_if.write(.addr(j), .wdata(small_aux));
53
54      j += 1;
55
56  end
57
58 end
59
60 if (requested_operation == ENCRYPTION)
61     @(posedge encryption_res_valid_o);
62 else if (requested_operation == DECRYPTION)
63     @(posedge decryption_res_valid_o);
64
65 @(posedge sysclk_i);
66
67 j = 0;
68
69 while (j <= 3) begin
70
71     top_if.read(.addr(j), .rdata(small_aux));
72
73     if (requested_operation == ENCRYPTION)
74         cypher_text_enc[j*32 +: 32] = small_aux;
75     else if (requested_operation == DECRYPTION)
76         decrypted_text[j*32 +: 32] = small_aux;
77
78     j += 1;
79
80 end
81
82 if (requested_operation == ENCRYPTION)
83     read_encryption_res = 1;
84 else if (requested_operation == DECRYPTION)
85     read_decryption_res = 1;
86
87 @(posedge sysclk_i);
88
89 if (encryption_res_wrong == 1)
90     enc_wrong_flag = 1;
91
92 if (decryption_res_wrong == 1)
93     dec_wrong_flag = 1;

```

```

95     read_encryption_res = 0;
96     read_decryption_res = 0;
97
98     for (i = 1; i < input_text.size(); i++) begin
99
100        j = 0;
101
102        while (j <= 3) begin
103
104            small_aux = input_text[i][j*32 +: 32];
105
106            top_if.write(.addr(j), .wdata(small_aux));
107
108            j += 1;
109
110        end
111
112        if (requested_operation == ENCRYPTION)
113            @(posedge encryption_res_valid_o);
114        else if (requested_operation == DECRYPTION)
115            @(posedge decryption_res_valid_o);
116
117            @(posedge sysclk_i);
118
119            j = 0;
120
121            while (j <= 3) begin
122
123                top_if.read(.addr(j), .rdata(small_aux));
124
125                if (requested_operation == ENCRYPTION)
126                    cypher_text_enc[j*32 +: 32] = small_aux;
127                else if (requested_operation == DECRYPTION)
128                    decrypted_text[j*32 +: 32] = small_aux;
129
130                j += 1;
131
132            end
133
134            if (requested_operation == ENCRYPTION)
135                read_encryption_res = 1;
136            else if (requested_operation == DECRYPTION)
137                read_decryption_res = 1;
138
139            @(posedge sysclk_i);
140
141            if (encryption_res_wrong == 1)
142                enc_wrong_flag = 1;
143
144            if (decryption_res_wrong == 1)

```

```

145     dec_wrong_flag = 1;
146
147     read_encryption_res = 0;
148     read_decryption_res = 0;
149
150 end
151
152 message_done_i = 1;
153
154 if (enc_wrong_flag == 0 && requested_operation == ENCRYPTION)
155     $display("DONE_", vectors[test_idx].moo.name(),
156             ":_Full_message_with_index_%0d_has_been_CORRECTLY_ENCRYPTED.", test_idx);
157 else if (enc_wrong_flag == 1 && requested_operation == ENCRYPTION)
158     $display("DONE_", vectors[test_idx].moo.name(),
159             ":_Full_message_with_index_%0d_has_been_WRONGLY_ENCRYPTED.", test_idx);
160
161 if (dec_wrong_flag == 0 && requested_operation == DECRYPTION)
162     $display("DONE_", vectors[test_idx].moo.name(),
163             ":_Full_message_with_index_%0d_has_been_CORRECTLY_DECRYPTED.", test_idx);
164 else if (dec_wrong_flag == 1 && requested_operation == DECRYPTION)
165     $display("DONE_", vectors[test_idx].moo.name(),
166             ":_Full_message_with_index_%0d_has_been_WRONGLY_DECRYPTED.", test_idx);
167
168 @(posedge sysclk_i);
169
170 endtask

```

Anexa 3 – Codul pentru Design Test Wrapper

```
1 module Design_Test_Wrapper (input logic sysclk_i,
2                               input logic rst_tests_button_i,
3                               input logic run_full_button_i,
4                               output logic [13:0] display1_o,
5                               output logic [13:0] display2_o,
6                               output logic [13:0] display3_o,
7                               output logic [13:0] display4_o);
8
9
10
11   logic rst_n_i;
12   logic start_enc_i;
13   logic start_dec_i;
14   logic [2:0] mode_of_operation_i;
15   apb_interface top_if (.sysclk_i(sysclk_i));
16   logic message_done_i;
17   logic encryption_res_valid_o;
18   logic decryption_res_valid_o;
19   typedef enum bit [2:0] {ECB = 1, CBC, CFB_128, OFB, CTR} modes_of_op;
20
21   bit [6:0] left1_counter_value;
22   bit [6:0] left2_counter_value;
23   bit [6:0] right1_counter_value;
24   bit [6:0] right2_counter_value;
25   bit [6:0] correct_enc_cnt;
26   bit [6:0] correct_dec_cnt;
27   bit [6:0] total_enc_cnt;
28   bit [6:0] total_dec_cnt;
29   bit full_req_flag;
30   bit full_btn_is_pressed;
31   bit [1:0] reset_counter;
32
33   bit [3:0] test_index;
34   bit test_result_correct;
35
36   typedef enum bit [3:0] {RST1 = 0, ENC1, RST2, DEC1, RST3,
37   ENC2, ENC3, RST4, DEC2, DEC3, RST5, ENC4, DEC4, ENC5, DEC5, RST6} test_state_machine;
38   test_state_machine test_state;
39   test_state_machine initial_ts;
40
41   typedef enum bit [1:0] {WRITE = 0, WAIT_DONE, READ} big_state_machine;
42   big_state_machine big_state;
43   bit [3:0] small_state;
44   typedef enum bit [1:0] {SEL = 0, ENABLE, READY} transfer_state_machine;
45   transfer_state_machine transfer_state;
```

```

46     bit first_clock;
47
48     bit [2:0] word_cnt;
49
50     /****** Define test vectors ******/
51
52     typedef struct { logic [127:0] key;
53                     bit [511:0] plain;
54                     bit [511:0] cypher;
55                     bit [1:0] length;
56                     bit [127:0] iv;
57                     modes_of_op moo;
58                 } test_vector;
59
60     test_vector vectors[10:0];
61
62     initial begin
63
64         left1_counter_value = 0;
65         left2_counter_value = 0;
66         right1_counter_value = 0;
67         right2_counter_value = 0;
68         full_req_flag = 0;
69         correct_enc_cnt = 0;
70         correct_dec_cnt = 0;
71         total_enc_cnt = 0;
72         total_dec_cnt = 0;
73         full_btn_is_pressed = 0;
74         test_index = 1;
75         reset_counter = 0;
76         first_clock = 0;
77
78         Reset_Module();
79
80         vectors[1].key = 'hFE_DC_BA_98_76_54_32_10_01_23_45_67_89_AB_CD_ED;
81         vectors[1].plain = 'hAA_AA_AA_AA_BB_BB_BB_BB_CC_CC_CC_DD_DD_DD_DD_EE_
82             EE_EE_EE_FF_FF_AA_AA_AA_BB_BB_BB_BB;
83         vectors[1].cypher = 'hC5_87_68_97_E4_A5_9B_BB_A7_2A_10_C8_38_72_24_5B_12_DD_
84             90_BC_2D_20_06_92_B5_29_A4_15_5A_C9_E6_00;
85         vectors[1].length = 1;
86         vectors[1].iv = 0;
87         vectors[1].moo = ECB;
88
89         vectors[2].key = 'h01_23_45_67_89_AB_CD_EF_FE_DC_BA_98_76_54_32_10;
90         vectors[2].plain = 'hAA_AA_AA_AA_BB_BB_BB_BB_CC_CC_CC_DD_DD_DD_DD_
91             EE_EE_EE_FF_FF_AA_AA_AA_BB_BB_BB_BB;
92         vectors[2].cypher = 'h5E_C8_14_3D_E5_09_CF_F7_B5_17_9F_8F_47_4B_86_19_2F_
93             1D_30_5A_7F_B1_7D_F9_85_F8_1C_84_82_19_23_04;
94         vectors[2].length = 1;
95         vectors[2].iv = 0;

```

```

96     vectors[2].moo = ECB;
97
98     vectors[3].key = 'h01_23_45_67_89_AB_CD_EF_FE_DC_BA_98_76_54_32_10;
99     vectors[3].plain = 'hEE_EE_EE_EE_FF_FF_FF_AA_AA_AA_BB_BB_BB_BB_AA_
100    AA_AA_AA_BB_BB_BB_CC_CC_CC_DD_DD_DD_DD;
101    vectors[3].cypher = 'h4C_B7_01_69_51_90_92_26_97_9B_0D_15_DC_6A_8F_6D_78_
102    EB_B1_1C_C4_0B_0A_48_31_2A_AE_B2_04_02_44_CB;
103    vectors[3].length = 1;
104    vectors[3].iv = 'h00_01_02_03_04_05_06_07_08_09_0A_0B_0C_0D_0E_0F;
105    vectors[3].moo = CBC;
106
107    vectors[4].key = 'hFE_DC_BA_98_76_54_32_10_01_23_45_67_89_AB_CD_EF;
108    vectors[4].plain = 'hEE_EE_EE_EE_FF_FF_FF_AA_AA_AA_BB_BB_BB_BB_AA_AA_
109    AA_AA_BB_BB_BB_CC_CC_CC_DD_DD_DD;
110    vectors[4].cypher = 'h91_F2_C1_47_91_1A_41_44_66_5E_1F_A1_D4_0B_AE_38_0D_3A_
111    6D_DC_2D_21_C6_98_85_72_15_58_7B_7B_B5_9A;
112    vectors[4].length = 1;
113    vectors[4].iv = 'h00_01_02_03_04_05_06_07_08_09_0A_0B_0C_0D_0E_0F;
114    vectors[4].moo = CBC;
115
116    vectors[5].key = 'h01_23_45_67_89_AB_CD_EF_FE_DC_BA_98_76_54_32_10;
117    vectors[5].plain = 'hEE_EE_EE_EE_FF_FF_FF_AA_AA_AA_BB_BB_BB_BB_AA_AA_
118    AA_AA_BB_BB_BB_CC_CC_CC_DD_DD_DD;
119    vectors[5].cypher = 'h69_D4_C5_4E_D4_33_B9_A0_34_60_09_BE_B3_7B_2B_3F_AC_32_
120    36_CB_86_1D_D3_16_E6_41_3B_4E_3C_75_24_B7;
121    vectors[5].length = 1;
122    vectors[5].iv = 'h00_01_02_03_04_05_06_07_08_09_0A_0B_0C_0D_0E_0F;
123    vectors[5].moo = CFB_128;
124
125    vectors[6].key = 'hFE_DC_BA_98_76_54_32_10_01_23_45_67_89_AB_CD_EF;
126    vectors[6].plain = 'hEE_EE_EE_EE_FF_FF_FF_AA_AA_AA_BB_BB_BB_BB_AA_AA_AA_
127    AA_BB_BB_BB_CC_CC_CC_DD_DD_DD;
128    vectors[6].cypher = 'h0D_9B_86_FF_20_C3_BF_E1_15_FF_A0_2C_A6_19_2C_C5_5D_CC_CD_
129    25_A8_4B_A1_65_60_D7_F2_65_88_70_68_49;
130    vectors[6].length = 1;
131    vectors[6].iv = 'h00_01_02_03_04_05_06_07_08_09_0A_0B_0C_0D_0E_0F;
132    vectors[6].moo = CFB_128;
133
134    vectors[7].key = 'h01_23_45_67_89_AB_CD_EF_FE_DC_BA_98_76_54_32_10;
135    vectors[7].plain = 'hEE_EE_EE_EE_FF_FF_FF_AA_AA_AA_BB_BB_BB_BB_AA_AA_AA_
136    AA_BB_BB_BB_CC_CC_CC_DD_DD_DD;
137    vectors[7].cypher = 'h1D_01_AC_A2_48_7C_A5_82_CB_F5_46_3E_66_98_53_9B_AC_32_36_
138    CB_86_1D_D3_16_E6_41_3B_4E_3C_75_24_B7;
139    vectors[7].length = 1;
140    vectors[7].iv = 'h00_01_02_03_04_05_06_07_08_09_0A_0B_0C_0D_0E_0F;
141    vectors[7].moo = OFB;
142
143    vectors[8].key = 'hFE_DC_BA_98_76_54_32_10_01_23_45_67_89_AB_CD_EF;
144    vectors[8].plain = 'hEE_EE_EE_EE_FF_FF_FF_AA_AA_AA_BB_BB_BB_BB_AA_AA_AA_
145    AA_BB_BB_BB_CC_CC_CC_DD_DD_DD;

```



```

196         .display_out(display2_o));
197
198     Display_7Segx2_Transcoder TR4(.number_in(right2_counter_value),
199                               .display_out(display1_o));
200
201     ila_0 ILA1(.clk(sysclk_i),
202                 .probe0(left1_counter_value),
203                 .probe1(run_full_button_i),
204                 .probe2(top_if.sysclk_i),
205                 .probe3(top_if.paddr_i),
206                 .probe4(top_if.pwrite_i),
207                 .probe5(top_if.psel_i),
208                 .probe6(top_if.penable_i),
209                 .probe7(top_if.pready_o),
210                 .probe8(top_if.pwdata_i),
211                 .probe9(top_if.prdata_o),
212                 .probe10(top_if.ps1verr_o),
213                 .probe11(rst_n_i),
214                 .probe12(SM4_Top_Module.SM4_Core.EC1.round_cnt),
215                 .probe13(SM4_Top_Module.SM4_Core.EC1.key_state),
216                 .probe14(SM4_Top_Module.SM4_Core.EC1.vif.pwdata_i),
217                 .probe15(SM4_Top_Module.SM4_Core.EC1.x3),
218                 .probe16(SM4_Top_Module.SM4_Core.EC1.RKG1.tprime_o),
219                 .probe17(SM4_Top_Module.SM4_Core.EC1.RKG1.tprime_i),
220                 .probe18(SM4_Top_Module.SM4_Core.EC1.RKG1.ck_i),
221                 .probe19(SM4_Top_Module.SM4_Core.EC1.RKG1.ck_o),
222                 .probe20(SM4_Top_Module.SM4_Core.DC1.RKG2.tprime_o),
223                 .probe21(SM4_Top_Module.SM4_Core.DC1.RKG2.tprime_i),
224                 .probe22(SM4_Top_Module.SM4_Core.DC1.RKG2.ck_i),
225                 .probe23(SM4_Top_Module.SM4_Core.DC1.RKG2.ck_o)
226             );
227
228     //*****
229
230     //***** Write master logic *****
231
232     always @(posedge sysclk_i or posedge rst_tests_button_i) begin
233
234         if (rst_tests_button_i == 1) begin
235
236             left1_counter_value <= 0;
237             left2_counter_value <= 0;
238             right1_counter_value <= 0;
239             right2_counter_value <= 0;
240             correct_enc_cnt <= 0;
241             correct_dec_cnt <= 0;
242             total_enc_cnt <= 0;
243             total_dec_cnt <= 0;
244             full_req_flag <= 0;
245             full_btn_is_pressed <= 0;

```

```

246     test_index <= 1;
247     reset_counter <= 0;
248     first_clock <= 0;
249
250     Reset_Module();
251
252 end else begin
253
254     start_enc_i <= 0;
255     start_dec_i <= 0;
256     rst_n_i <= 1;
257
258 //      // Button logic
259
260     if (run_full_button_i == 1 && full_btn_is_pressed == 0
261     && full_req_flag == 0) begin
262
263         full_btn_is_pressed <= 1;
264         full_req_flag <= 1;
265
266         left1_counter_value <= 0;
267         left2_counter_value <= 0;
268         right1_counter_value <= 0;
269         right2_counter_value <= 0;
270         correct_enc_cnt <= 0;
271         correct_dec_cnt <= 0;
272         total_enc_cnt <= 0;
273         total_dec_cnt <= 0;
274         test_index <= 1;
275
276         test_state <= RST1;
277         big_state <= WRITE;
278         small_state <= 0;
279         transfer_state <= SEL;
280         word_cnt <= 0;
281         test_result_correct <= 1;
282         first_clock <= 1;
283
284     end
285
286
287     if (run_full_button_i == 0 && full_btn_is_pressed == 1)
288         full_btn_is_pressed <= 0;
289
290 //      Test requested
291
292     if (full_req_flag == 1) begin
293
294         if (test_index == 11) begin

```

```

296     full_req_flag <= 0;
297     left1_counter_value <= correct_enc_cnt;
298     left2_counter_value <= total_enc_cnt;
299     right1_counter_value <= correct_dec_cnt;
300     right2_counter_value <= total_dec_cnt;
301     test_index <= 1;
302
303 end else begin
304
305     if (test_state inside {RST1, RST2, RST3, RST4, RST5, RST6}) begin
306
307         reset_counter <= reset_counter + 1;
308
309         if (reset_counter == 2'b11) begin
310
311             if (test_state == RST6)
312                 test_index <= test_index + 1;
313
314             test_state <= test_state.next;
315             reset_counter <= 0;
316             rst_n_i <= 1;
317
318         end else begin
319
320             Reset_Module();
321
322         end
323
324     end else if (test_state inside {ENC1, ENC2, ENC3, ENC4, ENC5}) begin
325
326         if (word_cnt <= vectors[test_index].length) begin
327
328             if (first_clock == 1) begin
329
330                 start_enc_i <= 1;
331                 mode_of_operation_i <= vectors[test_index].moo;
332                 message_done_i <= 0;
333                 first_clock <= 0;
334
335         end
336
337         if (big_state == WRITE) begin
338
339             if (transfer_state == ENABLE) begin
340
341                 top_if.penable_i <= 1;
342
343             end else if (transfer_state == READY) begin
344
345                 top_if.penable_i <= 0;

```

```

346
347         end else if (transfer_state == SEL) begin
348
349             top_if.psel_i <= 1;
350             top_if.penable_i <= 0;
351             top_if.paddr_i <= small_state;
352             top_if.pwrite_i <= 1;
353
354             if (word_cnt == 0) begin
355
356                 if (vectors[test_index].moo == ECB) begin
357
358                     if (small_state <= 3)
359                         top_if.pwdata_i <=
360                         vectors[test_index].key[small_state*32 +: 32];
361                     else if (small_state <= 7)
362                         top_if.pwdata_i <=
363                         vectors[test_index].plain[(small_state-4)*32 +: 32];
364
365                 end else begin
366
367                     if (small_state <= 3)
368                         top_if.pwdata_i <=
369                         vectors[test_index].iv[small_state*32 +: 32];
370                     else if (small_state <= 7)
371                         top_if.pwdata_i <=
372                         vectors[test_index].key[(small_state-4)*32 +: 32];
373                     else if (small_state <= 11)
374                         top_if.pwdata_i <=
375                         vectors[test_index].plain[(small_state-8)*32 +: 32];
376
377                 end
378
379             end else if (word_cnt > 0) begin
380
381                 if (small_state <= 3)
382                     top_if.pwdata_i <=
383                     vectors[test_index].plain[128*word_cnt +
384                     small_state*32+: 32];
385
386             end
387
388         end
389
390         if (transfer_state == SEL) begin
391
392             transfer_state <= ENABLE;
393
394         end else if (transfer_state == ENABLE) begin
395

```

```

396         transfer_state <= READY;
397
398     end else if (transfer_state == READY) begin
399
400         transfer_state <= SEL;
401         small_state <= small_state + 1;
402
403     end
404
405     if ((word_cnt == 0 && vectors[test_index].moo == ECB
406       && small_state == 8) ||
407       (word_cnt == 0 && vectors[test_index].moo != ECB
408       && small_state == 12) ||
409       (word_cnt > 0 && small_state == 4)) begin
410
411         small_state <= 0;
412         big_state <= WAIT_DONE;
413
414     end
415
416     end else if (big_state == WAIT_DONE) begin
417
418         top_if.penable_i <= 0;
419
420         if (encryption_res_valid_o == 1)
421             big_state <= READ;
422
423         transfer_state <= SEL;
424
425     end else if (big_state == READ) begin
426
427         if (transfer_state == READY) begin
428
429             top_if.penable_i <= 0;
430
431             if (small_state != 0) begin
432
433                 if (top_if.prdata_o !=
434                   vectors[test_index].cypher[128*word_cnt +
435                   32*(small_state-1)+: 32])
436                     test_result_correct <= 0;
437
438             end
439
440         end else if (transfer_state == ENABLE) begin
441
442             if (small_state < 4)
443                 top_if.penable_i <= 1;
444
445         end else if (transfer_state == SEL) begin

```

```

446
447         if (small_state <= 4) begin
448
449             top_if.psel_i <= 1;
450             top_if.penable_i <= 0;
451             top_if.paddr_i <= small_state;
452             top_if.pwrite_i <= 0;
453
454         end
455
456     end
457
458     if (small_state == 5 && transfer_state == SEL) begin
459
460         small_state <= 0;
461         big_state <= WRITE;
462         word_cnt <= word_cnt + 1;
463
464     end else
465
466     if (transfer_state == SEL) begin
467
468         transfer_state <= ENABLE;
469
470     end else if (transfer_state == ENABLE) begin
471
472         transfer_state <= READY;
473
474     end else if (transfer_state == READY) begin
475
476         transfer_state <= SEL;
477         small_state <= small_state + 1;
478
479     end
480
481     end
482
483 end else if (word_cnt > vectors[test_index].length) begin
484
485     message_done_i <= 1;
486     small_state <= 0;
487     big_state <= WRITE;
488     transfer_state <= SEL;
489     first_clock <= 1;
490     word_cnt <= 0;
491
492     test_state <= test_state.next;
493
494     if (test_result_correct == 1)
495         correct_enc_cnt <= correct_enc_cnt + 1;

```

```

496
497         total_enc_cnt <= total_enc_cnt + 1;
498
499         test_result_correct <= 1;
500
501     end
502
503 end else if (test_state inside {DEC1, DEC2, DEC3, DEC4, DEC5}) begin
504
505     if (word_cnt <= vectors[test_index].length) begin
506
507         if (first_clock == 1) begin
508
509             start_dec_i <= 1;
510             mode_of_operation_i <= vectors[test_index].moo;
511             message_done_i <= 0;
512             first_clock <= 0;
513
514         end
515
516         if (big_state == WRITE) begin
517
518             if (transfer_state == ENABLE) begin
519
520                 top_if.penable_i <= 1;
521
522             end else if (transfer_state == READY) begin
523
524                 top_if.penable_i <= 0;
525
526             end else if (transfer_state == SEL) begin
527
528                 top_if.psel_i <= 1;
529                 top_if.penable_i <= 0;
530                 top_if.paddr_i <= small_state;
531                 top_if.pwrite_i <= 1;
532
533             if (word_cnt == 0) begin
534
535                 if (vectors[test_index].moo == ECB) begin
536
537                     if (small_state <= 3)
538                         top_if.pwdata_i <=
539                         vectors[test_index].key[small_state*32 +: 32];
540                     else if (small_state <= 7)
541                         top_if.pwdata_i <=
542                         vectors[test_index].cypher[(small_state-4)*32 +: 32];
543
544             end else begin
545

```

```

546         if (small_state <= 3)
547             top_if.pwdata_i <=
548                 vectors[test_index].iv[small_state*32 +: 32];
549         else if (small_state <= 7)
550             top_if.pwdata_i <=
551                 vectors[test_index].key[(small_state-4)*32 +: 32];
552         else if (small_state <= 11)
553             top_if.pwdata_i <=
554                 vectors[test_index].cypher[(small_state-8)*32 +: 32];
555
556     end
557
558 end else if (word_cnt > 0) begin
559
560     if (small_state <= 3)
561         top_if.pwdata_i <=
562             vectors[test_index].cypher[128*word_cnt +
563                 small_state*32 +: 32];
564
565     end
566
567 end
568
569 if (transfer_state == SEL) begin
570
571     transfer_state <= ENABLE;
572
573 end else if (transfer_state == ENABLE) begin
574
575     transfer_state <= READY;
576
577 end else if (transfer_state == READY) begin
578
579     transfer_state <= SEL;
580     small_state <= small_state + 1;
581
582 end
583
584 if ((word_cnt == 0 && vectors[test_index].moo == ECB &&
585     small_state == 8) ||
586     (word_cnt == 0 && vectors[test_index].moo != ECB &&
587     small_state == 12) ||
588     (word_cnt > 0 && small_state == 4)) begin
589
590     small_state <= 0;
591     big_state <= WAIT_DONE;
592
593 end
594
595 end else if (big_state == WAIT_DONE) begin

```

```

596
597         top_if.penable_i <= 0;
598
599         if (decryption_res_valid_o == 1)
600             big_state <= READ;
601
602         transfer_state <= SEL;
603
604     end else if (big_state == READ) begin
605
606         if (transfer_state == READY) begin
607
608             top_if.penable_i <= 0;
609
610             if (small_state != 0) begin
611
612                 if (top_if.prdata_o != vectors[test_index].plain[128*word_cnt +
613
614                     32*(small_state-1) +: 32])
615                     test_result_correct <= 0;
616
617             end
618
619         end else if (transfer_state == ENABLE) begin
620
621             if (small_state < 4)
622                 top_if.penable_i <= 1;
623
624         end else if (transfer_state == SEL) begin
625
626             if (small_state <= 4) begin
627
628                 top_if.psel_i <= 1;
629                 top_if.penable_i <= 0;
630                 top_if.paddr_i <= small_state;
631                 top_if.pwrite_i <= 0;
632
633             end
634
635         end
636
637         if (small_state == 5 && transfer_state == SEL) begin
638
639             small_state <= 0;
640             big_state <= WRITE;
641             word_cnt <= word_cnt + 1;
642
643         end else
644
645             if (transfer_state == SEL) begin

```

```

646
647             transfer_state <= ENABLE;
648
649         end else if (transfer_state == ENABLE) begin
650
651             transfer_state <= READY;
652
653         end else if (transfer_state == READY) begin
654
655             transfer_state <= SEL;
656             small_state <= small_state + 1;
657
658         end
659
660     end
661
662     end else if (word_cnt > vectors[test_index].length) begin
663
664         message_done_i <= 1;
665         small_state <= 0;
666         big_state <= WRITE;
667         transfer_state <= SEL;
668         first_clock <= 1;
669         word_cnt <= 0;
670
671         test_state <= test_state.next;
672
673         if (test_result_correct == 1)
674             correct_dec_cnt <= correct_dec_cnt + 1;
675
676         total_dec_cnt <= total_dec_cnt + 1;
677
678         test_result_correct <= 1;
679
680     end
681
682 end
683
684 end
685
686 end
687
688 end
689
690
691 end
692
693 //*****
694
695 //***** Reset SM4 Module Function *****/

```

```
696
697     function void Reset_Module();
698
699         rst_n_i <= 0;
700         start_enc_i <= 0;
701         start_dec_i <= 0;
702         message_done_i <= 0;
703
704         top_if.paddr_i <= 0;
705         top_if.psel_i <= 0;
706         top_if.penable_i <= 0;
707         top_if.pwrite_i <= 0;
708         top_if.pwdata_i <= 0;
709
710     endfunction
711
712     //*****
713
714 endmodule
```