Treball de Fi de Grau

Grau en Enginyeria en Tecnologies Industrials (GETI)

IMPLEMENTACIÓ HARDWARE DE L'ALGORISME DE XIFRAT ADVANCED ENCRYPTION STANDARD (AES) I AVALUACIÓ DE LA SEVA ROBUSTESA ENVERS ATACS DE CANAL LATERAL.

Memòria.

Autor: *Pau Machado Panadés*

Director: *Álvaro Gómez Pau*

Codirector: *Salvador Manich Bou*

Convocatòria: *Juny 2021*

ETSEIB

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

UPC

«*The Road goes ever on and on,*
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can»
— JRR Tolkien.

Resum.

La seguretat d'un sistema ve determinada per la seguretat de la seva baula més dèbil. És per aquest motiu que la implementació física esdevé un punt clau per tal de garantir-la. Aquest Treball de Fi de Grau se centrarà en la implementació hardware de l'algorisme de xifrat AES (Advanced Encryption Standard). Primerament es farà un estudi de l'algorisme AES que posarà de manifest les seves característiques principals i seguidament, fent ús del llenguatge de descripció de hardware VHDL, es descriurà, simularà i sintetitzarà el circuit digital que implementarà l'algorisme abans esmentat. Finalment, s'avaluarà la robustesa de la implementació física envers un side channel attack (SCA).

Proemi.

Des que vaig començar el Grau, tenia ben ficat en el cap que el meu Treball de Fi de Grau estaria enfocat en la Mecànica. Era, en definitiva, l'àmbit que més coneixia i on, al llarg del transcurs dels meus estudis, més hàbil em notava. Tanmateix, uns pocs mesos abans de la formalització de la matrícula del darrer quadrimestre del Grau, vaig fer un dràstic viratge.

Mentre cursava l'assignatura obligatòria d'Electrònica, vaig sentir un estrany interès per aquesta matèria. No l'havia tocat en cap moment, ni tan sols m'havia atansat a una placa Arduino, i els meus coneixements envers aquest camp es podria dir que eren, com a màxim, d'orella. Va ser estudiant l'examen final d'aquesta assignatura que, fatigat i saturat de tantes hores, em va donar per investigar una mica sobre circuits integrats. Vaig quedar francament astorat en veure la complexitat de les implementacions hardware actuals, sobretot de les megalòpolis microscòpiques de transistors que habiten en un processador. Aleshores, em vaig quedar amb una pregunta: i això, com es fa?

Vaig contactar aleshores amb el professor Álvaro Gómez Pau, qui de fet havia estat el meu professor de les classes online d'Electrònica, per proposar-li fer un TFG que girés al voltant d'aquesta pregunta que se m'havia quedat. Ell em va comentar que, juntament amb el professor Salvador Manich Bou, havien estat conduint un parell de treballs consecutius de criptologia. El primer fou de la Laura Casanova sobre criptoanàlisi de l'algorisme AES enfront de SCA [1] i el segon, d'en Pau Albiol sobre les contramesures a aplicar-hi [2]. Ambdós treballs havien estat centrats en una implementació software sobre un microcontrolador existent, i a mi se'm va oferir la possibilitat de donar-los continuïtat però caminant per una altra branca: la de fer una implementació hardware de l'AES pròpiament dita.

M'hi vaig llençar de cap.

Al llarg d'aquest treball he aconseguit completar el meu objectiu i motivació personal: poder respondre la pregunta que m'havia quedat. I no només això; aquest treball ha aconseguit engrandir encara més la incipient curiositat que sentia envers l'electrònica, ja que he aconseguit aprofundir-hi a nivells que el limitat temps de què disposa l'assignatura del Grau no permet i he descobert una possible nova ruta amb què emmarcar la meva especialització professional.

Pau Machado Panadés.

ÍNDEX

Acrònims i símbols d'importància.

AES → *Advanced Encryption Standard*

SCA → *Atac de Canal Lateral*

PCA → *Power Consumption Analysis*

DPA → *Differential Power Analysis*

HDL → *Hardware Description Language*

VHDL → *Very High Speed IC (VHSIC) Description Language*

FPGA → *Field-Programmable Gate Array*

MSB → *Most Significant Bit*

NIST → *National Institute of Standards and Technology*

 → *porta lògica NAND*

 → *inversor lògic o porta NOT*

 → *porta lògica XOR*

$\oplus$ → *operador lògic suma exclusiva XOR*

$\emptyset b$ → *nombre expressat en codi binari*

$\emptyset x$ → *nombre expressat en codi hexadecimal*

Agraïments.

Als meus dos directors, en especial al professor Álvaro Gómez Pau per la seva trepidant entrega, el seu flagrant interès i la seva enorme dedicació.

Als meus amics i companys, pel suport, els bons moments i l'experiència compartida.

A la meva família, per estar sempre allí, sobren les paraules.

A tots vosaltres; moltes gràcies.

ETSEIB

Introducció, objectius i abast.

Amb el pas dels anys i l'avenç exponencial que està definint l'era actual, els sistemes electrònics digitals s'han anat tornant cada vegada més i més complexos. El nombre de transistors dins dels circuits integrats transcendeix el llindar de l'imaginable o concebible per la ment humana, la recargolada i sofisticada manera amb què s'interconnecten per a fer càlculs a vertiginosa celeritat és intangible i gairebé indescriptibles. És aquí on entren en joc els anomenats *llenguatges descriptius de hardware*, els HDL, ja que és gràcies a ells que som capaços de dissenyar els sistemes digitals contemporanis sense haver-nos de deixar limitar per llur complexitat arquitectònica.

Aquest treball s'emmarca en aquest context i pretén establir una eficaç implementació hardware del Advanced Encryption Standard (AES), l'algorisme de clau simètrica per excel·lència, mitjançant la seva descripció HDL. Addicionalment, es conduirà un detallat estudi de l'arquitectura del circuit digital dissenyat per tal de conèixer, a nivell de porta lògica i RTL, el seu aspecte i comprendre de quina manera es trasllada la descripció funcional de l'algorisme a hardware.

I, per què l'AES? Avui en dia, la criptologia és una ciència molt present i puixant a les vides del dia a dia donat l'incommensurable nombre de comunicacions que es produeixen a cada instant, les quals es desitgen segures. Els algorismes que s'empren són, en la seva enorme majoria, irrompibles a la força donades les llargàries de les claus que aquests utilitzen per a encriptar els missatges. L'AES és un clar exemple, ja que des del 2001, any de la seva estandardització, ningú no ha aconseguit trencar-lo en la seva totalitat mitjançant atacs "tradicionals".

Tanmateix, no és en el seu disseny algorísmic on radica la seva vulnerabilitat. Tot dispositiu electrònic digital emet contínuament fuites que contenen traces de la seva informació interna, sovint sensible, les quals reben el nom de *canals laterals*; sistemes físics que no formen part de l'estructura funcional del dispositiu. És a dir, en altres paraules, és justament la *implementació hardware* de l'AES la que pot presentar el major risc per al propi algorisme.

En aquest treball, doncs, s'ha proposat el següent **triple objectiu**:

1. Realitzar una detallada descripció funcional de l'AES, així com una implementació software del mateix, amb objecte de maximitzar-ne el coneixement a l'hora d'abordar al segon objectiu.

2. Efectuar un disseny hardware de l'algorisme AES mitjançant un llenguatge de descripció de hardware (HDL), detallar la seva arquitectura, verificar-lo per simulació i sintetitzar-lo.

3. Atacar aquesta implementació hardware de l'AES mitjançant un *atac de canal lateral* (SCA) per tal de poder comprovar-ne la robustesa.

Val a notar que aquest treball està pensat per a **abastar** l'AES-128 d'encriptació, desestimant el desxifrat, ja que aquest procés invers és simètric. A més a més, donat el context extraordinari de la COVID-19, es maximitzaran les plataformes i els programaris virtuals i de simulació per a conduir totes les verificacions, així com per a executar el SCA plantejat. Cal afegir, finalment, que aquest treball s'ha focalitzat sobretot en el procés de disseny hardware de l'AES, tant que se li ha dotat amb un capítol específic, el Capítol 3.

CAPÍTOL 1. Estat de l'art.

En el present capítol es donarà una contextualització del marc en què s'incorpora aquest treball; es detallaran aquells conceptes teòrics d'importància i aquelles dades que constitueixen la recerca duta a terme en el transcurs de la seva realització. Caldrà, doncs, parlar sobre dos grans blocs: el de la **criptologia**, de la qual es tractarà la *criptografia* i la *criptoanàlisi*, i el del **disseny hardware de sistemes digitals**, on s'abordarà tant els *dispositius* que ho implementen com els *llenguatges HDL*.

1.1. La criptografia.

El mot criptografia prové de la unió de dues paraules gregues: *kryptós* (κρυπτός), que es traduiria com a "secret", "ocult", i *graphein* (γράφειν), que es tracta del verb "escriure". Per tant, no és agosarat definir-la com l'estudi i la pràctica que s'encarrega de la comunicació secreta o de l'intercanvi de missatges que es desitgen secrets.

El Diccionari de la llengua catalana [3] la defineix com «*art d'escriure en caràcters secrets, d'una manera xifrada o convencional*», una manera més poètica però no menys certa; i l'autor John F. Dooley ho fa com «*les tècniques per crear sistemes d'escriptura secreta*» [4]. És a dir; quan es parla de criptografia hom es refereix al fet de la *creació* (i subseqüent estudi) de sistemes per a transmetre missatges ocults.

1.1.1. Breu contextualització i evolució.

La criptografia porta convivint amb la humanitat des dels temps antics. Hi ha constància de diversos mecanismes de xifratge de les civilitzacions clàssiques que es coneixen al detall avui dia [5].

En l'antiga Grècia destaca l'*escítala*, un mètode basat a enrotllar helicoïdalment una cinta de papir al voltant d'unes vares d'un gruix determinat i escriure el missatge secret de forma longitudinal, de tal manera que un cop es desenrotllés només es podria tornar a entendre mitjançant una vara del mateix gruix. Això ho feien servir les comandàncies de les milícies espartanes per a comunicar-se entre si.

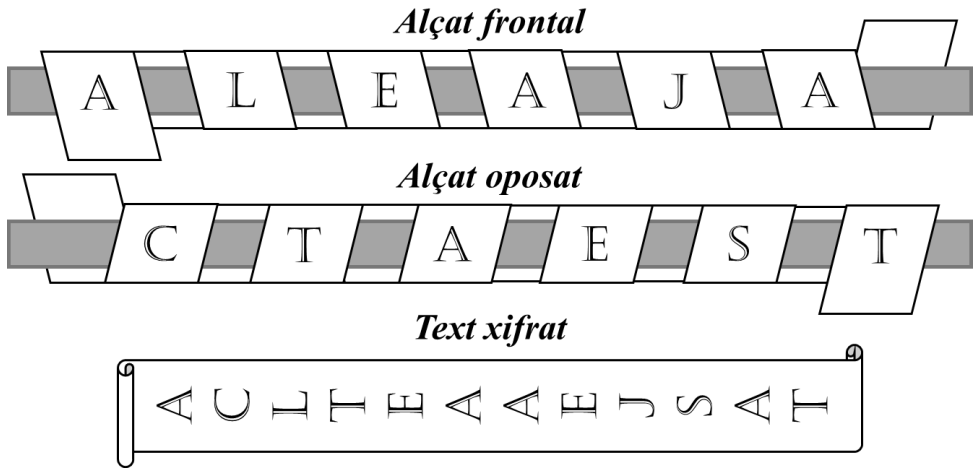


***Figura 1**. Representació gràfica d'una escítala. Els dos plans mostrats són oposats i il·lustren el sentit d'escriptura longitudinal del text secret "ALEA JACTA EST", i la imatge inferior representa el text xifrat obtingut en desenrotllar.*

Un altre mètode que s'ha fet famós en la contemporaneïtat és el de l'Emperador Romà Julius Cèsar, al qual se li ha posat el nom de "*Caesar-cipher*". Era una tècnica simple que consistia a substituir cada lletra del missatge secret per la lletra corresponent a descendir un cert nombre de posicions de l'alfabet partint de la lletra a substituir.

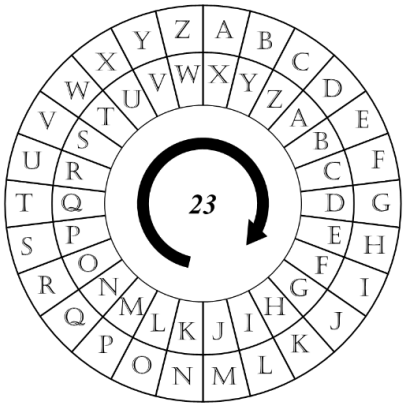


Figura 2. *Caesar-cipher amb un shifteig de 23 lletres, on les de l'anell extern són substituïdes per les de l'intern.*

Fent un salt temporal, a l'Edat Mitjana destaca el *Chiffre de Vigenère* [6], un mètode de xifratge atribuit al francès Blaise de Vigenère[1] però concebut per l'italià Giovan Battista Bellaso[2] [7]. Donat un missatge secret, es proposa una paraula qualsevol i es repeteix tantes vegades fins que es tingui la mateixa quantitat de lletres que el missatge, obtenint així una clau. Aleshores, cal emprar el que es coneix com *tabula-recta*: una taula de 26 files i 26 columnes (una per cada lletra de l'alfabet) que conté l'alfabet per files, on cada fila s'ha permutat aquest una posició. Prenent com a coordenada de fila cada lletra del missatge secret i com a coordenada de columna, l'homònima de la clau, es va obtenint el missatge xifrat (veure la *Figura 3* per a més detall).

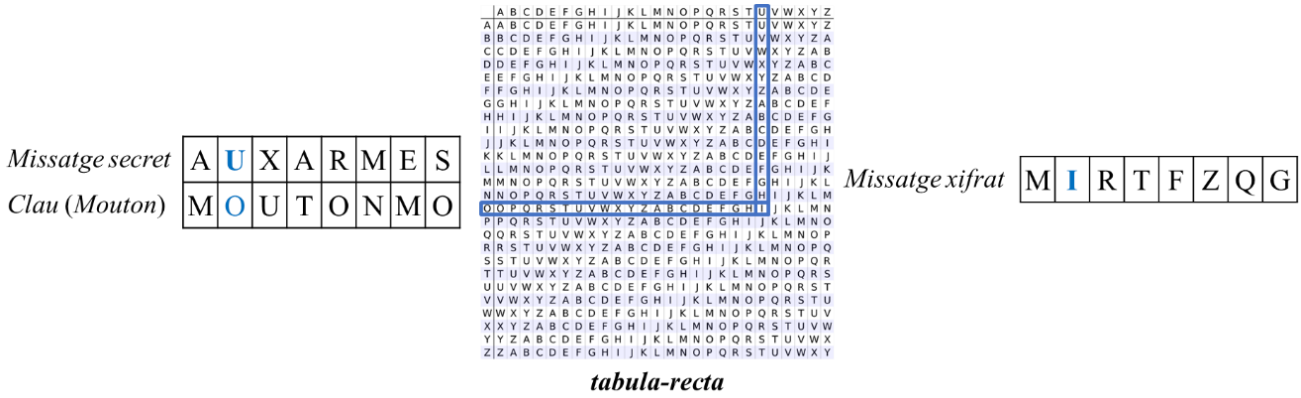


Figura 3. *Exemple del Chiffre de Vigenère, amb missatge secret "AUX ARMES" i mot de clau "MOUTON".*

Aquest mètode de xifrat és una manera elegant d'emprar el *Caesar-cipher* (en aquest cas, 26 vegades) per tal d'oferir tant la substitució de caràcters com la difusió d'aquests.

[1] Blaise de Vigenère (5 d'april del 1523 – 19 de febrer del 1596), fou un traductor, diplomàtic, astròleg, alquimista i criptògraf francès. Es diu que el 1549 féu un viatge a Roma durant el qual llegí llibres sobre criptografia i entrà en contacte amb criptògrafs italians; quelcom que l'inspirà a crear el seu mètode de xifrat.

[2] Giovan Battista Bellaso (Brescia 1505 - ?), fou un criptògraf de vida i mort quasi desconegudes, veritable inventor del *Chiffre de Vigenère*. Per a més informació, veure la referència destacada al cos del treball..

A partir d'aleshores, succeïren diferents esdeveniments d'importància en l'àmbit de la criptografia, accelerada amb l'arribada de les Grans Guerres. Cal destacar la màquina *Enigma* [8] dissenyada per les forces militars alemanyes durant la Segona Guerra Mundial (1939-1945). Es tracta d'una màquina criptogràfica electromecànica de rotors; és a dir, que es val de senyals elèctrics i rotors mecànics per a realitzar el xifratge d'un missatge donat. Simplificant, el seu funcionament es basava en 3 rotors que, mitjançant una interconnexió concreta substituïen les 26 lletres de l'alfabet (quelcom similar al *Caesar-Cipher*) i, després, es feia una reflexió dels senyals determinada per tal de reconduir la lletra substituïda una altra vegada pels 3 rotors. Addicionalment, hi havia un panell de connexions que permetia afegir una permutació addicional. El funcionament es troba, simplificat, a la *Figura 4*.

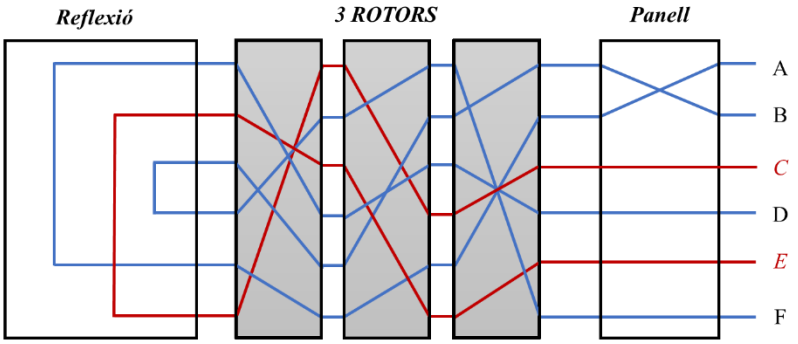


Figura 4. *Representació gràfica simplificada de les interconnexions de l'Enigma, destacant el recorregut de C.*

Finalment, en èpoques ja més contemporànies, l'electrònica començà a evolucionar amb rapidesa amb la invenció del transistor el 1947 per John Bardeen, William Shockley i Walter Houser Brattain, la creació dels primers circuits integrats entre els anys 1958 i 1959 per Jack Kilby[3] a Texas Instruments [9] [10], i amb el disseny per l'equip liderat per Federico Faggin de l'Intel 4004, el primer microprocessador amb CPU completa en un xip el 1971. A tot això se li suma la incipient naixença de la Internet, primer ARPANET als anys 60, i la declaració de la Llei de Moore[4] el 1965 [11]. Per tant, en aquest context, la necessitat d'un mètode criptogràfic eficaç era latent.

Sota aquesta necessitat es creà el *Data Encryption Standard* (DES) [12], un estàndard d'encriptació seleccionat pel NBS (*National Bureau of Standards*), ara NIST (*National Institute of Standards and Technology, Institut Nacional d'Estàndards i Tecnologia*), que permetia xifrar textos de longitud 64 bits amb claus de 56 bits. Es va aprovar el 1976 i, en aquell moment, se'l va considerar com a imbatible. Tanmateix, amb l'exponencial creixement de la computació i d'acord amb la Llei de Moore, no va costar gaire desmentir aquesta afirmació. L'any 1998 l'empresa EFF (*Electronics Frontier Foundation*) van crear la màquina "Deep Crack" amb la qual van aconseguir recuperar, a la força, la clau del DES en 22 hores i 15 minuts [13].

És en aquest context que nasqué l'AES (*Advanced Encryption Standard*), fruit del desig de trobar un substitut al DES capaç d'enfrontar-se amb els problemes de la ràpida innovació en computació, i del qual es parlarà en aquest treball.

[3] Jack Clair Kilby (8 de novembre del 1923 – 20 de juny del 2005), desenvolupador del primer circuit integrat en base de silici i guanyador del Premi Novel de Física del 2000 per aquest. Cal mencionar que Robert Noyce desenvolupà, a Fairchild Semiconductor, el circuit integrat en base de germani, tot i que el més guardonat i reconegut fou Kilby.

[4] "*El nombre de transistors es duplica aproximadament cada dos anys*".

1.1.2. Nocions bàsiques i ús en l'actualitat.

L'objectiu primordial de la criptografia és que certs missatges transmesos entre dos o més interlocutors siguin inintel·ligibles per la resta d'entitats i que ningú no pugui modificar-ne el contingut mentre és enviat. Sovint se sol anomenar *adversari* a aquell'entitat hipotètica que estigui a l'encalç de la informació que es transmet, i és comú referir-se a l'emissor i al receptor d'aquesta com a *Alice* i *Bob*[5].

Per a tal propòsit, la criptografia es val de dues eines [4]:

- **La codificació**: un seguit de transformacions de caire semàntic conegudes tan sols pels interlocutors, o públiques (com ara el codi ASCII, el binari...); el *codi*.
- **El xifrat**: un seguit de procediments de caire algorísmic que són emmascarats amb una informació que tan sols coneixen els interlocutors del missatge; la *clau*.

De les dues eines anteriors, la que més importància té actualment és el **xifrat**, sinònim del mot encriptació en català. Partint del missatge secret, que se sol anomenar com a *plaintext*, se'l sotmet a diferents operacions concretes fins a obtenir el missatge xifrat, el *ciphertext*. Atès que per desxifrar el *ciphertext* tan sols caldria conèixer l'algorisme amb què s'ha encriptat, s'inclou un pas en què el missatge és emmascarat per una *clau* que tan sols coneixen els interlocutors.

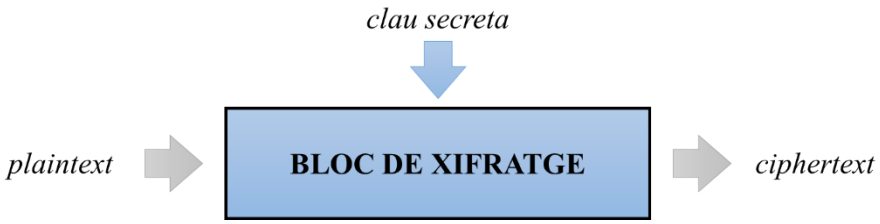


***Figura 5**. Esquema bàsic d'un algorisme de xifratge amb els tres principals actors.*

Principalment, es poden distingir dues categories de xifratges [4]:

- Substitució: es prenen els diferents caràcters d'un missatge i, un a un, se substitueixen per un altre caràcter segons un criteri predeterminat. Així doncs, dins del *ciphertext* hi haurà segurament molts caràcters que no formaven part del missatge original.
- Permutació: es roten i, de vegades, transposen els caràcters d'un missatge segons un criteri concret. Per tant, el *ciphertext* resultat contindrà només els caràcters que apareixien en l'original.

Original	D	A	D	A	I	S	M	E
Substituït	Q	E	Q	E	W	P	L	A
Permutat	I	S	M	E	D	A	D	A

***Taula 1.** Exemple de substitució monoalfabètica i permutació simple del missatge "DADAISME".*

[5] Els personatges ficticis *Alice* i *Bob* van ser inventats per Ron Rivest, Adi Shamir, i Leonard Adleman en el seu paper *A Method for Obtaining Digital Signatures and Public-key Cryptosystems* [33]. Ara es consideren els arquetips quan es parla dels interlocutors en criptografia.

L'exemple de la *Taula 1*, a nivell pragmàtic, no es podria considerar com a segur. La substitució que s'ha mostrat és l'anomenada monoalfabètica, és a dir, que canvia cada caràcter per un d'homònim, de tal manera que en el *ciphertext* s'obtindria la mateixa freqüència d'aparició del caràcter substituït que en l'original. La permutació, d'altra banda, és una simple inversió del terme per la meitat que, per raons evidents, és fàcil de descobrir.

En casos pràctics, els algorismes de substitució i de permutació solen estar presents en un mateix bloc de xifratge i s'implementen amb funcions i operacions molt complexes pensades per a què aportin el màxim grau de *confusió*[6] i *difusió*[7]; és a dir, per tal que tots els caràcters del *ciphertext* depenguin de cadascun dels caràcters del *plaintext*.

A partir de l'inici de la computació i l'inici de l'era de la informació, la criptografia ha pres una gran rellevància en el dia a dia. Cada segon es realitzen milions de comunicacions i cal assegurar-se que aquestes siguin segures enfront de la gran quantitat de potencials *adversaris*. A més a més, la natura cada vegada més impersonal d'aquestes provoca que la clau no pugui ser acordada, en el sentit més estricte de la paraula, entre els diversos interlocutors.

Els *sistemes criptogràfics* [14] apareixen com a resposta a aquestes noves necessitats, els quals estan pensats per a ser aplicats de forma pública i massiva en les diferents xarxes de comunicacions contemporànies (Internet, xarxes locals, telefonia, etc.). Destaquen les següents funcions:

- <u>*Bloc de xifratge*</u>: en la gran majoria dels casos es tracta d'algorismes *públics*; és a dir, que tothom els coneix. Això és així donada la gran robustesa dels diferents estàndards i mecanismes que s'empren, i perquè les claus tenen mides molt significatives. Inclou:
  - → *Xifrat*: s'encripta el missatge de l'emissor per obtenir el text xifrat.
  - → *Desxifrat*: es desencripta el text xifrat per obtenir l'original i que el receptor el llegeixi.
- <u>*Bloc de generació de clau*</u>: per tal que el receptor pugui desencriptar el missatge de l'emissor cal que la clau emprada en l'algorisme de desxifrat coincideixi o sigui la precisa per tal d'obtenir el mateix text original. Els dos mecanismes més importants per a poder-ho implementar es veuran en el pròxim apartat.

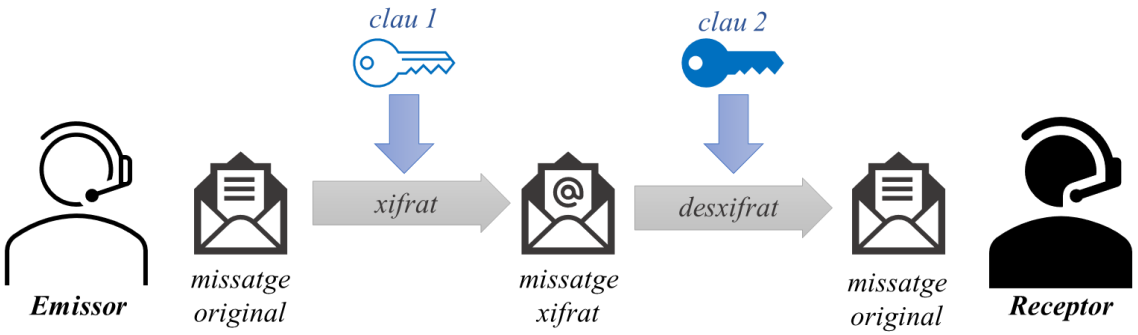


***Figura 6**. Representació d'un sistema criptogràfic (en blau: generació de clau; en gris: bloc de xifratge)*

[6] El fet de camuflar la relació entre el *ciphertext* i la *clau* o el *plaintext*.

[7] Si es canviés un caràcter del *plaintext*, la gran majoria dels caràcters del *ciphertext* haurien de canviar també.

1.1.3. Sistemes simètrics i asimètrics de generació de clau.

Com s'ha comentat en l'anterior apartat, per a poder transmetre missatges cal que el sistema criptogràfic disposi d'un algorisme de generació de claus que permeti que el receptor pugui llegir el missatge encriptat de forma correcta. En general, es distingeixen dos sistemes diferents [14].

Sistemes de clau simètrica.

En aquest tipus de sistema l'emissor i el receptor empren la mateixa clau per a xifrar i desxifrar, respectivament. És el mètode que s'ha emprat des de l'Antigor, ja que els interlocutors que volien comunicar-se en secret solien acordar sempre la clau, de tal manera que només ells la coneixien. Actualment, donada la impersonalitat de les comunicacions i el fet que els interlocutors mai solen estar físicament junts, el que es fa és emmagatzemar la clau en un lloc el més segur possible on, en cas de voler-se enviar o de rebre un missatge, ambdues parts hi poden accedir per a xifrar o desxifrar.

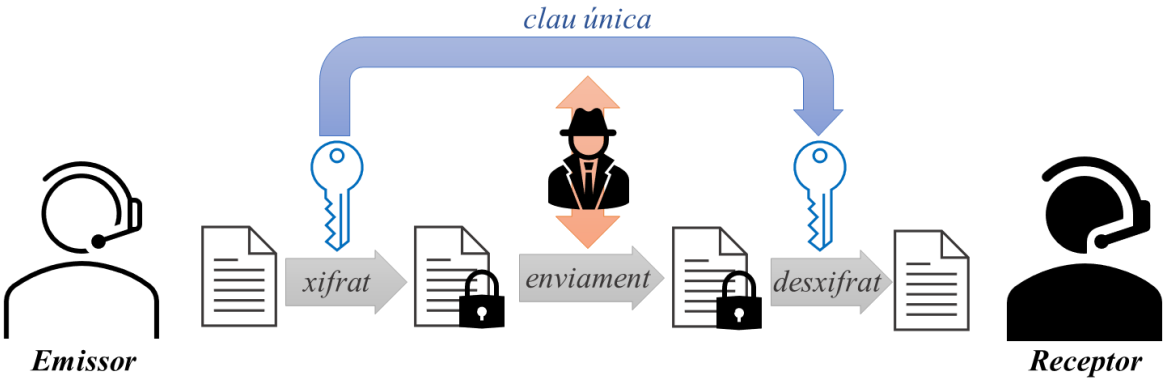


***Figura 7**. Representació gràfica d'un sistema de clau simètrica en què es mostren els punts vulnerables on un adversari podria manllevar informació.*

Val a comentar que, si bé en determinades aplicacions és un bon sistema, a un *adversari* que es trobés entremig de les comunicacions (*man-in-the-middle*[8]) tan sols li faria falta tenir accés a la clau per poder desxifrar el missatge enviat pel receptor. Aleshores, podria llegir la informació oculta o fins i tot modificar-la, xifrar-la de nou, i enviar-la al receptor. És per aquest motiu que avui en dia no se solen emprar els sistemes simètrics en casos de comunicacions on la clau només pugui ser pública (com, per exemple, Internet). A més a més, amb els avenços en computació i la rapidesa amb què es poden arribar a fer càlculs, un *adversari* que interceptés missatges podria fàcilment descobrir la clau.

Alguns algorismes d'importància que requereixen de claus simètriques són el DES, ara extint, el Triple DES[9] i l'AES. Això és així ja que les rutines d'encriptació i de desencriptació requereixen ambdues de la mateixa clau per poder passar del *plaintext* al *ciphertext* i retornar al *plaintext* original; és a dir, amb una clau diferent el procés de desxifrat donaria com a resultat un missatge completament diferent al *plaintext*.

[8] Un atac d'intermediari o *man-in-the-middle* és un atac on l'adversari es troba enmig de les comunicacions entre receptor i emissor. [14]

[9] El Triple DES fou una alternativa al DES, el qual havia estat vençut, en què s'aplica el DES tres vegades consecutives amb 2 o 3 claus diferents [14]. Tot i ser més segur que el DES, segueix sent molt lent i incomparable a l'AES.

Sistemes de clau asimètrica.

En aquests sistemes, que se solen conèixer també pel nom de *sistemes criptogràfics de clau pública*, cadascun dels interlocutors disposa de dues claus. Una d'elles és sempre pública, és a dir, que qualsevol agent extern la pot consultar; l'altra, però, és privada i només hi pot accedir l'interlocutor que en sigui propietari. Hi ha dos camins possibles per a implementar-ho:

El primer d'ells és el que intenta aconseguir el màxim de confidencialitat possible, i se sol emprar quan el que es vol fer és un *xifrat de missatge* (i. e. enviar un missatge). L'emissor procedeix a encriptar la informació que vol enviar usant la *clau pública del receptor*. Aleshores, per a poder interpretar-lo, el receptor empra la *seva pròpia clau privada* per a desencriptar el missatge.

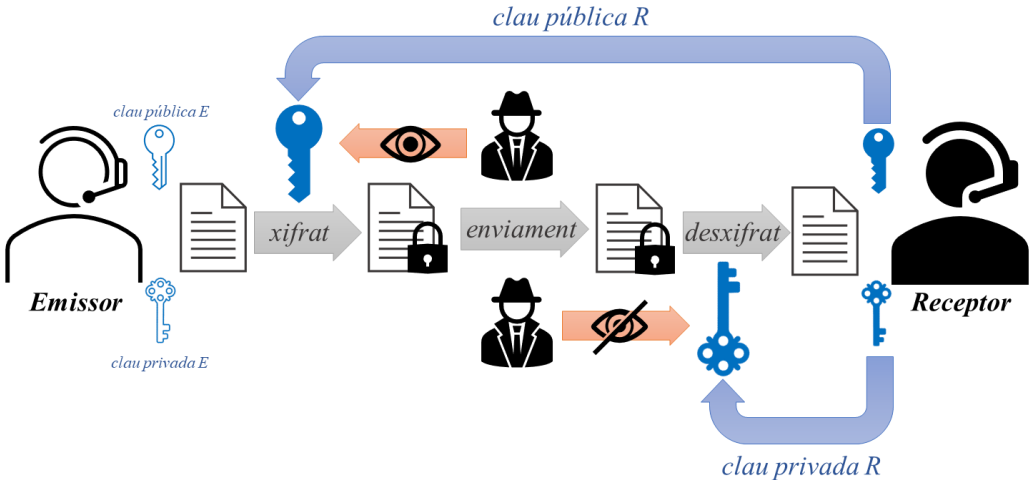


***Figura 8**. Representació gràfica d'un sistema de clau asimètrica de xifrat de missatge.*

D'altra banda, hi ha un altre mètode que permet demostrar l'autoria i l'autenticitat d'una informació a enviar, que se sol anomenar *firma digital*. Es procedeix com en el cas anterior, tot i que ara l'emissor encripta amb la *seva pròpia clau privada*. Així doncs, el receptor desxifra el missatge amb la *clau pública de l'emissor*, cosa que només podria fer si el *ciphertext* fos l'autèntic.

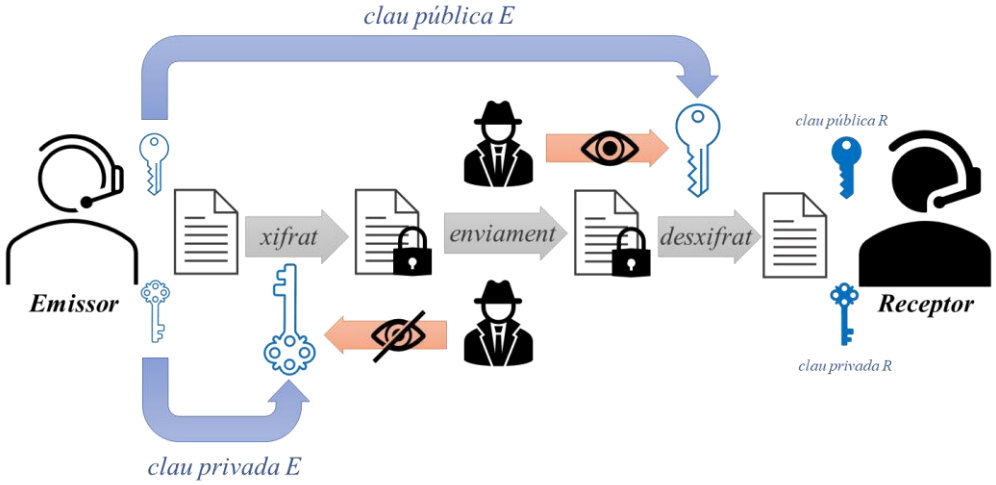


***Figura 9.** Representació gràfica d'un sistema de clau asimètrica de firma digital.*

En tots dos casos, un *adversari* coneix la clau pública i podria arribar a trobar el *ciphertext*, però per a desxifrar-lo necessitaria sempre la *clau privada*. Això fa que aquests sistemes siguin molt segurs. L'algorisme més important que ho implementa és el RSA.

1.2. La criptoanàlisi.

El mot criptoanàlisi prové, també, de la unió de dues paraules gregues: *kryptós* (κρυπτός), que com s'ha vist abans es traduiria com a "secret", "ocult", i *analýein* (αναλύειν), que es tracta del verb "analitzar". Per tant, hom podria considerar correcte de definir-la com l'anàlisi de l'ocult o de les comunicacions secretes.

El Diccionari de la llengua catalana [3] la defineix com la «*part de la criptologia que estudia els sistemes criptogràfics per a trobar debilitats i trencar la seva seguretat sense el coneixement d'informació secreta*». Altrament, l'autor John F. Dooley ho fa com «*les tècniques per trencar sistemes d'escriptura secreta*» [4]. És a dir, a diferència de la criptografia, es sol definir la criptografia com el fet de *trencar* els sistemes de transmissió de missatges ocults mitjançant la seva anàlisi.

La criptoanàlisi ha anat evolucionant, com és lògic, de la mà de la criptografia; ja que aquesta neix de la necessitat de vèncer les argúcies amb què la informació s'oculta. En aquest treball es tractaran, tan sols, els mètodes contemporanis i moderns que són de major interès.

1.2.1. Nocions bàsiques i principis de Kerckhoffs.

La criptoanàlisi, a la pràctica, s'empra per a intentar penetrar la seguretat d'un sistema criptogràfic mitjançant una acció determinada, la qual sol rebre el nom d'*atac*. Quan s'aconsegueix l'objectiu de l'atac, posant així en risc la seguretat del sistema, es sol dir que el sistema ha estat *trencat* o *crackejat*. Aquest títol no se sol posar en casos en què s'aconsegueixi obtenir alguna part de la informació, sinó més aviat quan el sistema criptogràfic no aconseguiria mantenir en secret el missatge principal que pretén emmascarar.

En la modernitat, una hipòtesi de partida que sempre es pren és a col·lació d'unes assumpcions molt fonamentals fetes per Auguste Kerckhoffs[10] l'any 1883, les quals es coneixien sota el nom de *principis de Kerckhoffs* [15]. Aquests són sis afirmacions que garanteixen el correcte disseny d'un sistema criptogràfic (traduït directament de l'original francès):

1. *El sistema ha d'ésser materialment, ans matemàticament, indesxifrable.*
2. *No cal que requereixi de ser secret, i ha de poder sens inconvenient caure en mans enemigues.*
3. *La clau ha de poder ésser comunicada i recordada sense necessitat de ser escrita, i ésser canviada o modificada a criteri dels correspondents.*
4. *Cal que sia aplicable a la correspondència telegràfica.*
5. *Cal que sia portable, i el seu manteniment o funcionament no hauria d'exigir la intervenció de masses persones.*
6. *En definitiva, vistes les condicions en què s'aplicarà, el sistema ha de ser d'ús fàcil, no hauria de demandar angoixa ni la coneixença d'una llarga sèrie de regles a observar.*

[10] Auguste Kerckhoffs von Nieuwenhof (19 de gener del 1835 – 9 d'agost del 1903) fou un lingüista i criptògraf alemany, professor de llengües a Paris, i conegut per ser un dels impulsors del llenguatge construït Volapük.

La traducció d'aquests principis a un llenguatge més contemporani seria:

1. A la pràctica, el sistema ha de ser sempre indesxifrable.
2. L'algorisme de xifratge pot ser públic i no cal que es mantingui en secret.
3. Ha de ser senzill el fet de modificar i emmagatzemar una clau de xifratge.
4. Els criptogrames haurien de donar valors alfanumèrics i no simbòlics.
5. El sistema ha de ser portable i fàcil de transportar i reparar (avui en dia trivial).
6. El sistema ha de ser senzill per a l'usuari que l'empra, sense moltes "regles d'ús" enrevessades

En criptoanàlisi, el més important és la segona assumpció, la qual és la que estrictament es coneix com *el principi de Kerckhoffs*. És més, fou replantejat per Claude Shannon[11] en el que s'anomenà com la *màxima de Shannon*: «*L'adversari coneix el sistema*» [16]; és a dir, que cal dissenyar els sistemes criptogràfics assumint que l'adversari és coneixedor de la totalitat de llur funcionament.

Aquesta idea, des de la perspectiva de la criptoanàlisi, implica que, atès que l'algorisme de xifrat serà sempre públic o susceptible de ser conegut, l'únic element que roman en secretisme és *la clau*. En conseqüència, si un conjunt d'atacs són capaços de deduir, manllevar o trobar la clau d'un sistema criptogràfic, es podrà afirmar que aquest ha estat *trencat*. En conclusió, el que els principis de Kerckhoffs impliquen, a manera de corol·lari, és que de la clau depèn la seguretat del sistema.

1.2.2. Atacs passius i atacs actius.

En funció de l'enfoc que l'*adversari* vulgui donar, es distingeixen dos tipus d'atac criptogràfic [14]:

- ***Atac passiu***: es pretén monitorar el procés de transmissió entre els interlocutors amb l'objectiu de manllevar el màxim d'informació possible sense necessitat d'alterar el contingut d'aquesta. Per tant, l'amenaça d'aquesta classe d'atac rau sobre la confidencialitat.

- ***Atac actiu***: es busca alterar la informació transmesa entre els interlocutors o modificar les operacions del sistema mitjançant més agressivitat que no pas l'atac passiu. És a dir, en aquest cas s'amenaça la integritat de les dades o de l'estructura del sistema.

D'atacs actius ja s'ha mostrat en aquest treball un exemple: el *man-in-the-middle*, on es captura informació en una transmissió entre dos interlocutors amb la potestat d'alterar-la i lliurar-la al receptor, alterant-ne la integritat.

Els *atacs de força bruta* són una classe d'atac passiu molt emprada en criptoanàlisi, els quals es basen a introduir claus de forma massiva al sistema que es vol trencar fins a arribar a la combinació correcta. A la pràctica, la *complexitat* per a aconseguir trencar o no un sistema depèn sempre de la longitud de la clau. En binari, una clau de 128 bits causaria que s'haguessin de provar $2^{128} - 1$ combinacions diferents, cosa que implicaria un temps inefable a jutjar per la velocitat de càlcul actual. Per tant, un algorisme amb una longitud de clau suficient és considerat segur enfront d'aquests atacs.

Un altre tipus d'atac passiu és el *side-channel-attack* (SCA) que es veurà a continuació.

[11] Claude Edwood Shannon (30 d'abril del 1916 – 24 de febrer del 2001) fou un matemàtic, enginyer electrònic i criptògraf americà, i és considerat el pare de la Teoria de la Informació. A més a més, fou gran pioner de les comunicacions digitals i demostrà que la informació podia ser estudiada dins de la ciència.

1.2.3. Atacs de canal lateral (SCA).

Els *side-channel attacks* (SCA) o atacs de canal lateral són un incipient tipus d'atac passiu que ha anat agafant tirada des de la dècada dels noranta i que es basa a intentar manllevar informació de l'algorisme de xifrat no de la seva funcionalitat, sinó de la seva implementació física [17]. Els SCA han demostrat al llarg del temps ser molt més eficaços que els atacs que redueixen el sistema criptogràfic al seu algorisme en lloc de tenir en compte també l'arquitectura real que els implementa.

N'hi ha de molts tipus segons el tipus de sistema que implementa el bloc de xifratge que es vol atacar. Aquest treball se centrarà només en un dels tipus de canal lateral existents tot i que n'existeixen molts d'altres possibles, per simplicitat i per continuïtat amb els treballs anteriors [1] [2].

Power consumption analysis (PCA).

La majoria de circuits integrats estan desenvolupats en tecnologia CMOS (*Metall-Oxide-Semiconductor Circuit*), que s'implementa amb transistors tipus MOSFET sobre un substrat de silici. Llur propietat principal, a nivell d'estructura, és que combinen sempre parelles de transistors tipus N i tipus P en sèrie o paral·lel segons convingui.

En circuits digitals basats en portes lògiques hi ha dues formes *rellevants*[12] de consum de corrent:

- *Consum estàtic*: és la intensitat de corrent que el circuit precisa per a funcionar quan no s'altera el seu estat però es manté un senyal concret.
- *Consum dinàmic*: és aquell que precisa el circuit per a fer un canvi (*switch*) i que prové de la càrrega o la descàrrega del condensador associat al senyal de sortida.

En tecnologia CMOS l'únic consum important és el *dinàmic*. En la *Figura 10* es mostra la natura d'aquest corrent i la relació que té amb la càrrega i la descàrrega del condensador de sortida per un circuit inversor (que es podria entendre com una porta lògica NOT). Les tensions V_{OL} i V_{OH} representen les de baix i alt nivell, respectivament, i la tensió V_{cc} és la de la font d'alimentació.

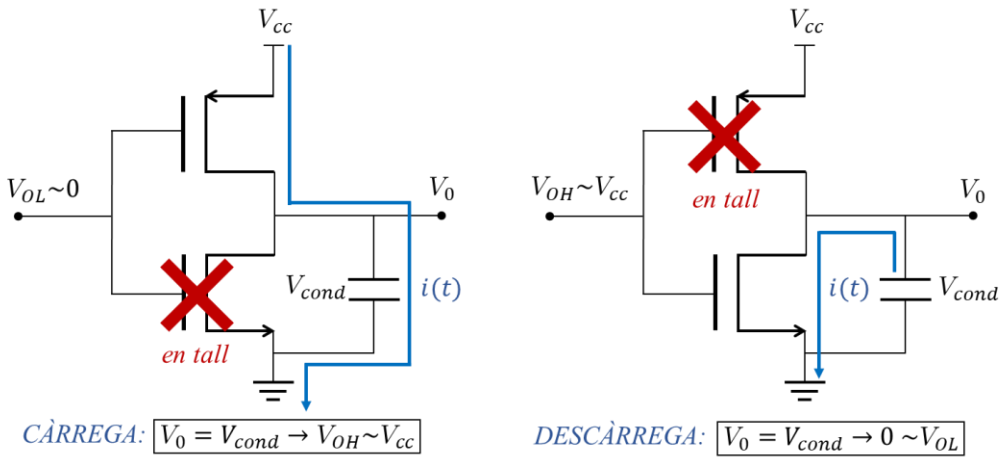


***Figura 10**. Procés de càrrega i descàrrega d'un circuit inversor de tensió CMOS (ideal) amb V_{cc} tensió d'alimentació, V_0 tensió de sortida igual a la tensió del condensador V_{cond}.*

[12] Entenent rellevant com d'importància per al SCA, ja que també existeixen el *short-circuit power consumption* i el *leakage power consumption* però son poc significatius en aquest cas [17].

En definitiva, es pot considerar que hi ha una relació entre el consum de corrent d'un circuit digital en tecnologia CMOS i els canvis (*swich* o *activat del node*) de valor dels senyals duts a terme pels diferents dispositius que l'integren. En general, els canvis de valor de nivell baix a nivell alt són els que produeixen pics de corrent més significatius, donat que si la lògica aplicada és per exemple NAND els condensadors s'estarien carregant. Val a dir que el condensador no hi és, per se, sinó que modelitza i emula la capacitància interna dels propis transistors.

A partir d'aquesta idea, cal establir un model de consum, el qual pretenen determinar aquesta relació especificada. Aquest treball posarà l'accent damunt el *Hamming weight model*, el qual assumeix que, quan es calcula un valor x_0, el corrent de consum està correlat amb el Hamming weight[13] d'aquest valor, $H_W(x_0)$ [17]. Per simplicitat, es considera que els canvis d'alt a baix nivell i de baix a alt nivell tenen el mateix pes a nivell de consum, així com que tots els bits dels senyals tenen la mateixa importància.

Per a fer la mesura del corrent, si bé hi ha diverses maneres, una bona idea és de col·locar una resistència just en connexió a terra del seu circuit d'alimentació. Aleshores, es monitoren les seves tensions i, segons la llei d'ohm, s'obtenen els valors de la intensitat del corrent de consum.

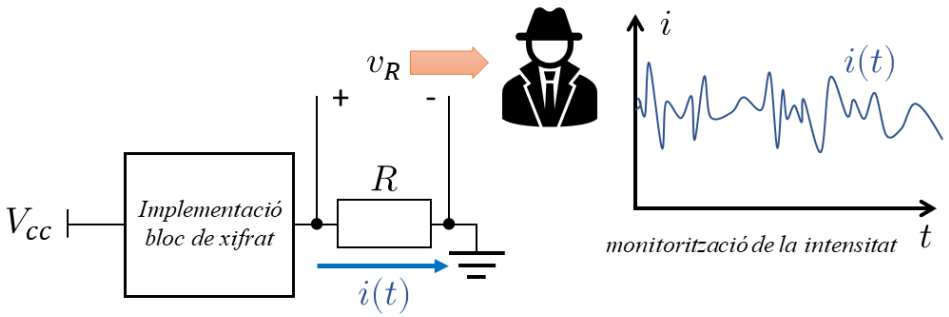


***Figura 11**. Esquematització general d'un SCA per monitorització del consum a partir d'una resistència.*

Altrament, per a l'enfoc de l'estudi es poden seguir dos tipus de metodologies clàssiques [17]:

- *Simple power analysis* (SPA): en aquest cas tan sols interessa obtenir conclusions sobre el funcionament de les operacions de l'algorisme per mitjà de la monitorització del consum. Aleshores, la informació obtinguda podria ser comparada a un "reconeixement del terreny" més que al fet de trencar el bloc de xifratge per se. Sovint es combina amb tècniques iteratives per a extreure informació complementària.

- *Differential power analysis* (DPA): per contra, aquesta classe de SCA pretén combinar la monitoratge del consum amb l'ús de tècniques d'estadística analítica per tal d'obtenir la clau de xifratge que s'està emprant. Normalment és una tècnica exitosa que permet trencar algorismes altament segurs, com l'AES, cosa que amb altres mecanismes, com ara atacs a la força bruta, es necessitaria emprar un temps quasi infinit per a aconseguir. En el cas del *Hamming weight leakage model*, el que se sol fer és comparar els valors obtinguts de forma física amb els valors estimats del *Hamming weight* per tal de trobar alguna correlació.

[13] El Hamming weight d'un senyal electrònic binari és el nombre de bits que siguin diferents al 0 lògic, en aquest cas, el nombre de valors 1 lògic.

1.3. El disseny hardware de sistemes digitals.

Els *sistemes digitals* són sistemes electrònics que treballen a un nombre discret de nivells de voltatge, que en binari és igual a dos: el nivell alt H i el nivell baix L que representen el 1 i 0 lògics, respectivament. Són la base de l'electrònica digital i generalment sempre serveixen per implementar *funcions i operacions lògiques* basades en l'àlgebra de Boole (o booleana).

Des d'un punt de vista de disseny hardware[14], avui en dia hi ha diversos camins per a implementar sistemes digitals. Es poden distingir dues grans rutes [18]:

- ***Implementació customitzada***[15]: l'element físic que implementa el sistema digital és unívoc al disseny realitzat; és a dir, tan sols serveix per a realitzar l'aplicació específica per a què s'ha dissenyat.
- ***Implementació semi-customitzada***: en aquest cas, l'element físic és versàtil; és a dir, serveix per a realitzar moltes aplicacions i és "programat" per a què realitzi aquella que es desitgi o que sigui convenient al disseny ideat.

En la *Figura 12* es mostra un esquema de les principals tecnologies que s'utilitzen per a implementar circuits digitals [18]. Al llarg d'aquest apartat s'anirà exposant en què consisteix cadascuna d'elles i en què es diferencien.

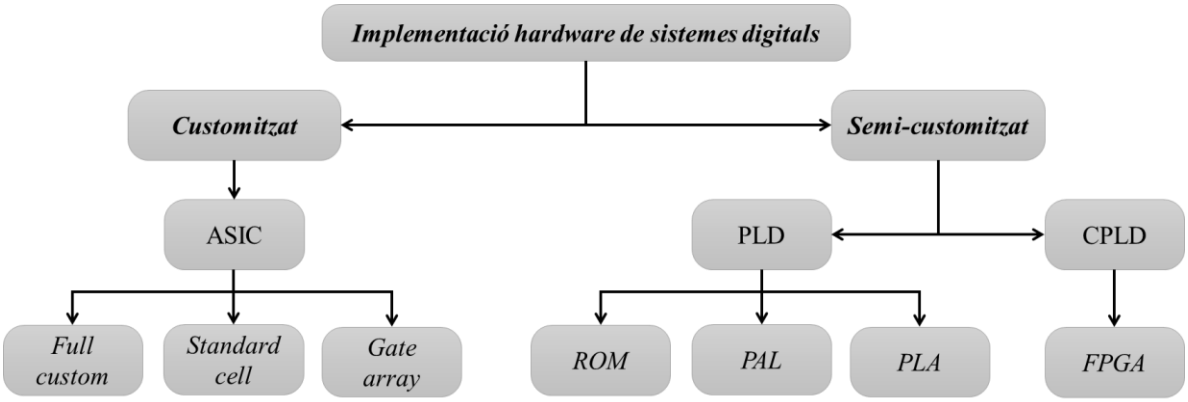


Figura 12. *Diferents tecnologies actuals per a la implementació d'un sistema digital en hardware.*

Totes aquestes tecnologies, a la pràctica, parteixen de l'ús del *circuit integrat* (IC) com a mètode de fabricació. Aquest és una placa feta a base de material semiconductor (silici) on hi ha "impresos" tots els elements lògics i electrònics necessaris per a la implementació d'una o diverses *funcions lògiques*. Actualment solen ser molt petits i s'implementen en forma d'encapsulat o xip i poden contenir centenars de milers de transistors. La tecnologia més emprada és l'anomenada CMOS comentada breument a l'apartat 1.2.3.

Sovint el disseny digital s'assisteix (o fins i tot es fa totalment) mitjançant un *hardware description language* (HDL), un llenguatge informàtic específic, del qual es parlarà en l'apartat pertinent.

[14] El hardware, que en català s'anomena *maquinari*, és el conjunt d'elements físics i materials que conformen la implementació de qualsevol sistema electrònic digital.

[15] Barbarisme que prové de la paraula anglesa "*custom"*, que en aquest context significa "*personalitzat*".

1.3.1. Circuits integrats d'aplicació específica (ASIC).

Un *application-specific integrated circuit* (ASIC) és, en termes generals, un *circuit integrat* dissenyat per a executar únicament una aplicació en particular. En altres mots, no estan pensats per a ser versàtils ni oferir la possibilitat d'un ús més generalitzat. Solen estar creats amb tecnologia MOS, i es poden distingir de diferents tipus segons el seu grau de personalització [18].

Full-custom *(completament personalitzada)*

Amb aquest enfoc, tots els aspectes del circuit imprès són escollits de forma expressa per l'aplicació amb què s'ha dissenyat. Es té un control complet del circuit a nivell de transistor, tal que es podria assegurar que s'aconsegueix una màxima optimització i s'ofereix el millor nivell de rendiment.

Malauradament, el procés necessari per a aconseguir-ho és molt complex, de tal manera que a la pràctica només és factible fer-ho amb circuits o bé molt petits, o bé analògics, donat que contenen molts menys transistors que un de gran.

Standard-cell *(cel·les estàndard)*

En aquest cas ja no es fa la implementació a nivell del transistor, sinó que es construeix el circuit a base de petites cel·les definides a priori (les *standard cells*). Sovint, els fabricants tenen llibreries de les cel·les estàndard que ofereixen, les quals solen ser elements electrònics bàsics com ara multiplexors de 2 a 1 línies, sumadors d'un bit, *flip-flops*, etc.; tot i que de vegades també es manufacturen cel·les més complexes com blocs de RAM.

Aleshores, el disseny de la implementació es fa per blocs, i cal decidir la manera amb què aquests s'interconnecten. Tanmateix, la característica principal de les ASIC es manté; el circuit resultat segueix tenint una funcionalitat unívoca a l'aplicació per a què s'ha dissenyat.

Gate-array *(matriu de portes predifoses)*

Ara, l'element mínim del circuit són matrius definides a priori formades per cel·les estàndard d'un únic tipus, conegudes com a *cel·les base*. El seu mètode de funcionament és bastant igual al cas de les *standard cells*, amb la diferència que les matrius de portes poden implementar blocs lògics més complexos. D'aquesta manera, el disseny i fabricació d'aquestes ASIC és, possiblement, el més relativament simple de tots. No obstant, el més utilitzat avui dia és l'*standard-cells*.

Les ASIC són, doncs, un mètode d'implementació hardware molt efectiu per a circuits petits, concrets, molt especialitzats i que es desitgin altament optimitzats (tot i que també es pot fer i es fa servir per a implementacions grans). No obstant això, el cost que suposa la seva fabricació és molt alt, per no dir altíssim, i cal reservar el seu ús per casos en què es fabriquin productes en massa o que es vulgui quelcom molt concret i eficaç [19].

Val a destacar que, donada l'alta complexitat de disseny de les ASIC, sovint se solen descriure mitjançant un llenguatge HDL. Després, mitjançant una eina de síntesi, es pot obtenir la distribució del xip al nivell que es desitgi. Més informació en l'apartat 1.3.3.

1.3.2. Dispositius lògics programables (PLD / CPLD).

Un *programable logic device* (PLD) és un dispositiu electrònic usat per a construir un circuit digital reconfigurable. En altres paraules, el circuit integrat que el forma permet ser "programat" per a què esdevingui el circuit digital dissenyat. D'aquí que formin part de les implementacions *semi-customized*. Sovint se les anomena com a *field-programmable*, ja que a diferència de les ASIC que es prefabriquen senceres i queden immutables, aquestes poden ser alterades "*in-the-field*"; és a dir, a "nivell de camp", sempre que calgui.

Bàsicament, un PLD està format per cel·les lògiques genèriques, conegudes com a *blocs lògics*, sempre prefabricades. Les unions entre aquestes cel·les es fan amb uns materials semiconductors específics que es poden entendre com a *fusibles*, és a dir, que segons un estímul permeten o no el pas de senyals a través d'ells (i. e. poden estar oberts o curtcircuitats). Aleshores, és possible configurar la PLD activant o desactivant aquests fusibles per tal d'obtenir el circuit desitjat [18].

Al llarg del temps hi ha hagut diverses tecnologies de dispositius PLD. Seguidament s'exposaran les més rellevants.

Simple field-programable logic devices.

Aquestes són les que, històricament, s'han anomenat com a PLD; tot i que se'ls anomena *simples* per a distingir-les del següent subtipus. Atès que la majoria estan en desús, o el seu ús s'ha vist molt reduït, es farà una breu menció a les seves diferències i característiques principals [18] [20]:

- **PROM** (*programable read-only memory*): durant bastant de temps, les memòries ROM s'aplicaven i es feien servir com a PLD (tot i que aquest terme encara no havia estat encunyat). Bàsicament, les PROM tenien fusibles per tal de crear o destruir de forma *permanent* les connexions que es desitgessin amb la resta del xip, de tal manera que es podien crear funcions lògiques. Després vingueren les EPROM, tal que els fusibles no es destruïen de forma permanent (*erasable-programable ROM*), i un llarg etcètera de variacions d'aquesta.
- **PLA** (*programable logic array*): inventada l'any 1970 a Texas Instruments, la PLA fou la primera PLD pròpiament dita. Disposava d'un pla de portes AND connectades a dues xarxes de cablejat: una procedia dels senyals d'entrada, i l'altre dels mateixos senyals d'entrada, però passats per una porta NOT. En aquestes xarxes hi havia els fusibles que permetien impedir o no el pas dels senyals a les portes AND per a "programar" la PLA. Val a destacar que aquest procediment és similar al d'obtenció de formes *product-of-sums* provinents d'una taula de la veritat; és més, sovint aquest era el mètode de disseny emprat.
- **PAL** (*programable array logic*): introduïdes l'any 1978 per Monolithic Memories, són un conjunt d'elements electrònics lògics que rodegen un nucli basat en PROM. N'hi havia de dos tipus: les *one-time programable* o OTP, que un cop "programades" no podien tornar al seu estat original; i les versions que permetien ser editades com una EPROM.

En general, aquesta tecnologia quasi ja no s'empra avui en dia, ja que les funcions lògiques que poden arribar a implementar són molt simples. Cal dir que, inicialment, les PROM i les PAL també eren considerades OTP (*one-time programable*), però després van sortir variacions "editables".

Complex field-programable logic devices.

Un *complex programable logic device* (CPLD) és un dispositiu que, com el seu nom indica, és capaç d'implementar funcions lògiques molt més complexes que el seu predecessor, el PLD. De forma genèrica, es pot pensar en CPLD com un xip que conté en el seu interior moltes PLD petites, manera amb què aconsegueix incrementar la complexitat de les operacions lògiques que implementa [20]. La *Figura 13* mostra una aproximació de la seva estructura interna.

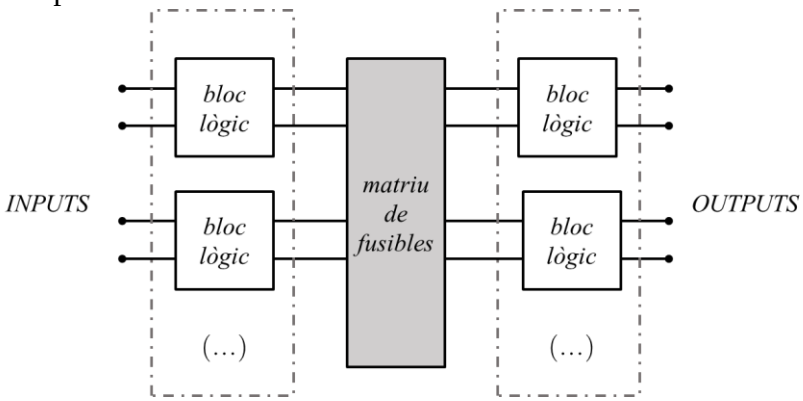


Figura 13. *Arquitectura genèrica d'un CPLD, on cada bloc lògic tendeix a ser una PLD.*

Una *field-programmable gate array* (FPGA), anomenada *matriu de portes programables in situ* en català, és un dispositiu semiconductor que disposa de diferents blocs lògics que poden ser interconnectats segons l'usuari desitgi. Si bé molt similars quant a concepte, es diferencia de les CPLD principalment per la seva arquitectura interna i pel seu quantiós abast.

En general, una FPGA disposa d'una complexa matriu de molts blocs lògics que implementen operacions senzilles interconnectats tots ells entre si; quelcom que ofereix molta llibertat de "programació". És precisament gràcies a aquesta versatilitat que aquests dispositius permeten implementar des de la porta lògica més senzilla fins a circuits molt complexos; és més, es podria afirmar que les FPGA són capaces d'implementar pràcticament qualsevol disseny hardware [20]. A la *Figura 14* es pot veure la seva arquitectura generalitzada.

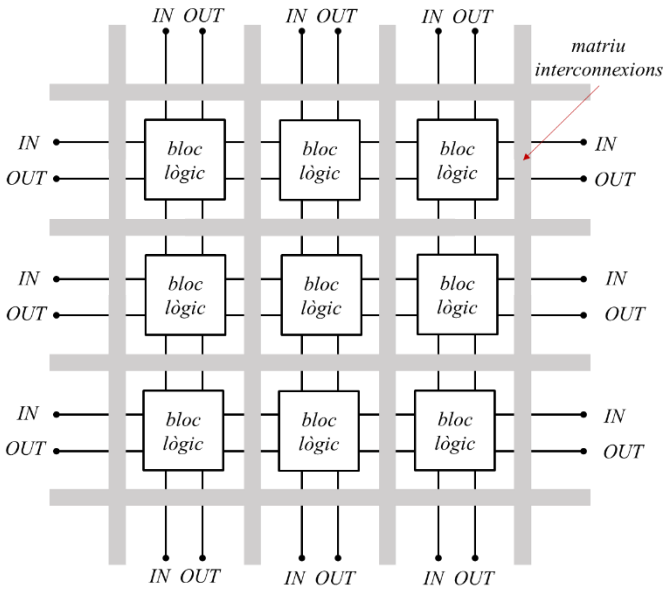


Figura 14. *Arquitectura genèrica d'una FPGA on s'ha destacat en gris la matriu d'interconnexions.*

Sovint, es considera que una FPGA és un tipus de CPLD, però com s'ha pogut veure són dispositius bastant diferents quant a arquitectura. És més, els blocs lògics de la CPLD en general solen ser més sofisticats en contrast amb els de la FPGA, que acostumen a ser més simples [18]. A nivell d'aplicació, se sol escollir abans una CPLD que una FPGA en casos en què es necessiti un alt rendiment, ja que el fet que la seva matriu de connexions sigui més senzilla i directa sol comportar retards de computació molt més petits. En canvi, atès que una FPGA té més flexibilitat, se sol preferir per a aplicacions on es requereixin més registres o una funció més complexa [20].

A nivell de preu, les FPGA són més barates. Per tant, en casos d'aplicacions experimentals, acadèmiques o de verificació de sistemes hardware solen ser les més triades. De vegades s'empren per a comprovar que el disseny d'una ASIC és funcional abans de fabricar-la.

1.3.3. Llenguatges de descripció de hardware (HDL).

Els *hardware description languages* (HDL) són llenguatges de computació que serveixen per a descriure el comportament o l'estructura de sistemes electrònics digitals. Si bé tenen molta similitud amb els llenguatges de programació, la seva semàntica i el seu ús no podrien ser més diferents. Una de les seves majors aplicacions és permetre la *simulació* dels circuits digitals que descriuen per tal de fer la seva verificació, a més a més de permetre la seva *síntesi* [21].

Moltes vegades, els HDL són vistos com una eina ECAD (*electronic computer-aided design*), les quals són eines destinades al disseny de sistemes electrònics per computador. Entre aquestes eines hi ha programes com l'*OrCAD*, del qual es parlarà amb més detall en l'apartat 4.4.

Una de les diferències fonamentals entre els HDL i els llenguatges de programació és, com s'ha advertit, la *semàntica*. El C o el Python3 estan pensats per a ser executats de forma seqüencial, és a dir, que les sentències s'executen una a una per ordre des de la línia superior del codi font fins a la darrera. Els HDL, per contra, tenen en compte que els senyals d'un sistema s'exciten a la vegada, cosa que implica que les sentències dels seus codis fonts s'executen, a no ser que s'indiqui el contrari, d'un sol cop. A més a més, com que els sistemes electrònics funcionen sovint per blocs, els HDL tenen en compte que cada bloc s'executi a la vegada però de forma independent per diferents motius, com ara per poder considerar retards de propagació entre els diferents blocs lògics [21].

Els usos més comuns dels llenguatges HDL són la simulació i la síntesi:

- ***Simulació***: els programes de simulació permeten compilar els diferents fitxers HDL per tal de crear un model de simulació amb què verificar el comportament del sistema. Aquests models serien com un "*circuit integrat virtual*" sobre el qual es poden excitar els senyals d'entrada que es desitgin. Per a fer tal cosa s'empren fitxers *testbench*, els quals indiquen el procés a seguir per a fer la simulació que es desitja.
- ***Síntesi***: el procés de síntesi d'un conjunt de fitxers HDL és la interpretació de la descripció abstracta d'aquests mitjançant les anomenades *eines de síntesi*. Aquestes procedeixen a la creació d'una *netlist*, una descripció de la connectivitat del sistema dissenyat pensat per a aplicar damunt d'un dispositiu PLD, CPLD, FPGA concret (habilitant o no els "fusibles" anteriorment descrits) o fins i tot per a poder descriure un ASIC. Aquest és l'ús més important dels HDL i el motiu pel qual són utilitzats actualment.

El procés de *disseny hardware de sistemes digitals* mitjançant HDL amb l'objectiu de ser implantats en un dels *dispositius lògics programables* (simples o complexos) comentats anteriorment es podria resumir en el diagrama de flux de la *Figura 15*.

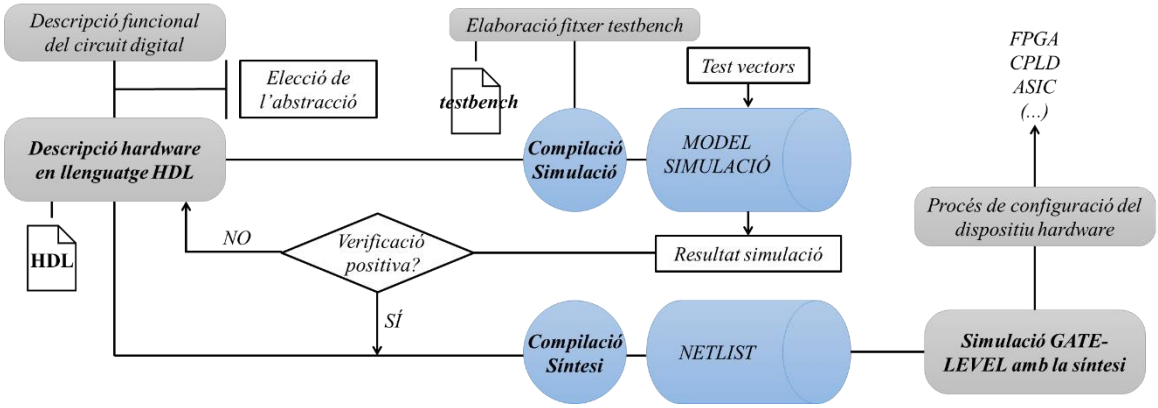


Figura 15. *Diagrama de flux que resum el procés de disseny hardware de sistemes mitjançant llenguatges HDL.*

Per a escriure els codis font que conformen la descripció amb HDL és necessari escollir un *nivell d'abstracció*. En general, per la descripció de circuits digitals solen distingir-se els següents [21]:

- ***Transistor-level***: les unitats bàsiques d'aquesta abstracció són els transistors, els condensadors i resistències, i d'altres elements del més baix nivell. Els senyals es tracten com a contínues en lloc de discretes, ja que s'empren els voltatges d'alt i baix nivell en lloc dels 1 i 0 lògics.
- ***Gate-level***: les unitats bàsiques són portes lògiques, petits multiplexors, biestables, etc. Ara sí que es tracten els senyals com a 1 i 0, de tal manera que les relacions entre les entrades i les sortides són considerades segons l'àlgebra booleana.
- ***Register-transfer-level* (RTL)**: és un nivell molt més abstracte que els anteriors, ja que els senyals esdevenen "*data types"* definits a priori i les unitats bàsiques són sistemes funcionals com sumadors complets, multiplexors de moltes línies, registres paral·lels, etc. Es descriuen mòduls separats que després es van unint. Aquest nivell és el més emprat per implementacions de baix/alt nivell arquitectònic, i permet enfocs *comportamentals* i *estructurals*.
- ***Processor-level***: les unitats bàsiques són conegudes com a IPs (*intellectual properties*), ja que solen cer processadors, blocs de memòria de gran eslora, etc.

Val a comentar, també, els dos possibles enfocs que normalment se li poden donar als dissenys amb llenguatge HDL. Depenen, normalment, del tipus de descripció que es fa:

- ***Behavioral*** (*comportamental*): la descripció que es fa és funcional; és a dir, es descriu el comportament del sistema que s'està dissenyant sense entrar en elements electrònics. Es pot enfocar de diverses maneres, des de descripció de funcions lògiques com descripcions condicionals, similars a les dels llenguatges de programació.

- ***Structural*** (*estructural*): en aquest cas sí que s'entra a parlar d'elements electrònics, portes lògiques i d'altres components per tal de tenir control sobre l'arquitectura resultant.

Hi ha molts tipus de llenguatges HDL, els més importants dels quals essent el *Verilog* i el *VHDL*. En aquest treball, per les raons que s'exposen al 3.2.1, s'ha decidit de centrar-se en el VHDL.

El llenguatge de descripció de hardware VHDL

El VHDL (*very high description language*) és un llenguatge de descripció de hardware estandarditzat per l'IEEE (*Institute of Electrical and Electronics Engineers*) l'any 1987. Va néixer a partir d'un programa desenvolupat pel Departament de Defensa dels Estats Units conegut com a VHSIC (*very high-speed integrated circuit*) que tenia per a objectiu desenvolupar un nou HDL per a poder simular ASIC. És més, el nom VHDL prové d'una fusió entre les sigles i el significat de VHSIC HDL. Quan es féu públic, l'IEEE el va estandarditzar i ara n'és el principal administrador [21]. La versió que s'ha considerat ha estat la IEEE Std 1076-2019, donat que és la més actual en data de realització d'aquest treball.

Els mòduls en VHDL s'anomenen *entity* (entitats), tal que cadascuna d'elles representa un bloc que formarà el circuit digital. Es poden imaginar com caixes negres amb entrades i sortides, tal que el seu interior ve descrit pel codi font del mateix mòdul. L'aspecte o esquema general bàsic d'una entitat es pot veure a continuació:

```
library ieee;
use ieee.std_logic_1164.all;
use -- nom altres llibreries;

entity NOM_ENTITAT is
  port(
        input:in signal nom_datatype_input;
        output:out signal nom_datatype_output
    );
end entity;

architecture NOM_ARCH of NOM_ENTITAT is
    -- part declarativa de l'arquitectura
begin
    -- sentències de l'arquitectura;
    -- aquí és on es descriu el funcionament
    -- de l'entitat en qüestió
end architecture;
```

Els objectes principals del VHDL són *constants*, *senyals* i *variables*. Els senyals són les de major importància per a aquest treball, ja que com el seu nom indica representen els diferents senyals electrònics que recorren l'entitat. Les variables i les constants, a diferència dels senyals, no tenen un homònim *directe* en el circuit electrònic físic. La característica principal de les constants són que el seu valor no pot canviar en cap moment. En canvi, les variables serien similars a una variable convencional en un llenguatge de programació. En aquest treball tan sols s'han emprat senyals.

Els *datatype* d'aquest llenguatge són diversos i depenen de la llibreria que es consulti. En aquest projecte s'han emprat els de la llibreria IEEE.std_logic_1164, a més a més d'alguns personalitzats que es veuran més endavant.

La guia que s'ha seguit, a tall de bibliografia, per a poder emprar el VHDL al llarg d'aquest treball ha estat el llibre referenciat: [22].

CAPÍTOL 2. L'Advanced Encryption Standard.

L'*Advanced Encryption Standard* (AES) és un estàndard d'encriptació en bloc del govern dels Estats Units, essent un dels algorismes més importants en criptografia de clau simètrica. Aquest treball hi està completament centrat, en particular en la seva versió de clau de 128 bits (l'AES-128), i al llarg del present capítol es tractaran els seus orígens, la seva seguretat i la seva descripció funcional.

2.1. A propòsit de l'AES.

2.1.1. Naixement.

El 2 de gener de 1997, el *National Institute of Standards and Technology* (NIST) [23], en veient que l'algorisme *Data Encryption Standard* (DES) estava quedant-se obsolet davant les exigències dels nous temps, va llançar a la comunitat criptogràfica una petició per a obtenir un nou algorisme de xifratge capaç de sobrevenir-se als reptes de seguretat del segle 21. En concret, la sol·licitud que van presentar va ser la de desenvolupar un *Federal Information Processing Standard* (FIPS), un estàndard que s'anuncia públicament amb l'objectiu de fer-lo servir en aplicacions de computació per part d'agències governamentals americanes no militars o pel sector privat.

Les condicions que el NIST imposà raïen, principalment, sobre la llargària de les claus i la mida dels blocs de xifratge. Fins ara, el DES estava fent-ne servir de 56 i 64 bits, quelcom que l'havia fet força vulnerable a atacs de força bruta donat que la capacitat de computació creixent de l'època havia tornat aquestes llargàries insegures. Particularment, es va imposar que la mida del bloc del nou estàndard fos de 128 bits i que la clau pogués acceptar mides de 128, 192 i 256 bits.

Durant el període d'entre l'anunci de la sol·licitud i l'any 1998, es van presentar una gran quantitat de candidats. El 20 d'agost del 1998, el NIST va convocar la *First AES Candidate Conference* on va anunciar que havia seleccionat una quinzena d'algorismes d'entre totes les propostes, i les va sotmetre a la revisió de la comunitat criptogràfica. El procés de selecció del finalista va durar fins l'any 2000 i va constar de dues rondes de filtratge. En la primera ronda, la qual va concloure el 15 d'abril del 1999, es van seleccionar cinc candidats finalistes: el MARS, RC6, Rijndael, Serpent, i Twofish. Durant la segona ronda es van anar fent trobades en forma de conferències entre el NIST i la comunitat criptogràfica, la qual havia estat enviant comentaris a propòsit dels cinc candidats, i el 2 d'octubre del 2000 es va anunciar que l'algorisme finalista era Rijndael, desenvolupat per dos investigadors belgues: Joan Daem i Vincent Rijmen. [24]

Finalment, l'Advanced Encryption Standard (AES) va ser anunciat com a FIPS 197 el 26 de novembre de 2001. [25]

2.1.2. Seguretat i atacs pràctics exitosos.

Al llarg de la seva vida, l'AES ha estat sotmès a criptoanàlisi per gran part de la comunitat criptogràfica amb l'objectiu de trobar-hi vulnerabilitats o, fins i tot, trencar-lo. Seguidament es farà menció dels intents que han presentat una amenaça tangible a la seguretat de l'algorisme.

L'any 2002, poc després que fos anunciat pel NIST, Nicolas Courtois i Josef Pieprzyk van desenvolupar un atac teòric que van anomenar *eXtended Sparse Linearization* (XSL) *attack* que clamava ser capaç de trencar l'AES [26]. En el seu paper, els autors afirmaven que, donat que l'àlgebra no lineal que hi ha darrere l'AES és prou simple, hi havia una manera de desxifrar dita estructura. Això va causar molta controvèrsia en el moment, ja que feia poc temps que l'estàndard havia estat anunciat. Tanmateix, al llarg del temps, diversos estudis van demostrar que l'atac teòric XSL no era plausible de forma pràctica, ja que o bé els sistemes d'equacions que plantejaven eren irresolubles o bé la *complexitat* prenia valors trepidants. [27]

L'any 2009, Alex Biryukov, Dmitry Khovratovich, i Ivica Nikolić van aconseguir amb èxit realitzar un *related key-attack* a l'AES [28]. Aquest tipus d'atacs parteixen de la hipòtesi que les diferents claus generades pel *sistema de generació de clau simètrica* solen estar connectades entre si per alguna relació matemàtica (com podria ser que els primers 15 bits són sempre els mateixos). Els resultats van ser els millors en molt de temps, obtenint una complexitat de 2^{96} per cada 2^{35} claus. No obstant això, l'escenari plantejat per un *key-attack* és bastant inversemblant perquè tan sols cal un bon protocol de selecció de clau (per exemple, mètodes pseudoaleatoris) per a combatre'ls.

L'any 2010, Vincent Rijmen, el co-desenvolupador del Rijndael, va publicar un paper petit on adreçava un atac basat en *chosen-key-relations-in-the-middle* a l'AES-128 [29]. La idea principal d'aquesta classe d'atacs és, sota la hipòtesi que es coneix la relació entre els claus, quina clau s'està fent servir o directament que es pot escollir la clau, comprometre el bloc de xifratge. En el cas plantejat per Rijmen, calia tenir la potestat d'aturar el procés d'encriptació en un punt determinat (*in-the-middle*), aplicar una modificació a l'estat (*chosen-key*) i reprendre l'encriptació a la inversa per obtenir un *plaintext* modificat amb el qual descobrir la clau real de xifratge. Aquest atac es podria considerar com a efectiu sota els supòsits plantejats per l'autor.

Finalment, l'any 2011, Andrey Bogdanov, Dmitry Khovratovich i Christian Rechberger van publicar el primer *key-recovery* atac a l'AES complet [30]. Fins aleshores, els èxits havien estat damunt de versions reduïdes de l'algorisme (menys rondes de l'habitual). L'atac va estar basat en *biclique*, que és una estructura que permet ampliar el nombre de rondes atacades, obtenint uns resultats de $2^{126,2}$ operacions per l'AES-128, $2^{190,2}$ per l'AES-192 i $2^{252,3}$ per l'AES-256. Aquests valors, tanmateix, són tan sols una quarta vegada menors que els obtinguts a la força bruta. Per tant, si bé fou considerat com un avenç en criptoanàlisi de l'AES, a la pràctica aquest atac és molt similar als resultats proveïts per força bruta i no representa una amenaça suficientment gran per a considerar que va trencar l'AES.

Com a conclusió, a data de la realització d'aquest treball no s'ha obtingut encara cap mètode pragmàtic que permeti trencar un AES complert per part d'un adversari que no tingui cap mena d'informació al voltant de la clau de xifratge. A tot això caldria afegir que els atacs de força bruta funcionen teòricament, si bé amb altes complexitats i requerits d'enorme capacitat de computació.

Tanmateix, els canals d'atac lateral o *side channel attacks* (SCA) sí que suposen una amenaça per l'AES i pràcticament per a qualsevol sistema físic de xifratge. El TFG de la Laura Casanova [1], precedent a aquest, es va fer criptoanàlisi de l'AES mitjançant un SCA. En aquest treball, més endavant, es farà l'anàlisi de la robustesa de la implementació dissenyada també amb SCA.

2.2. Funcionament i descripció tècnica.

Com ja s'ha comentat anteriorment, l'AES és un algorisme de xifratge en blocs de longitud fixa de 128 bits. Les claus, però, poden ser de llargària 128, 192 o 256 bits, tal que, a major quantitat de bits, major és la seguretat del xifrat. En funció de quina de les tres opcions s'escull, s'imposa que les operacions internes de l'algorisme s'executin en bucle un nombre de 10, 12 i 14 rondes, respectivament.

Aquest treball s'ha centrat en l'AES-128, és a dir, la versió de l'algorisme que empra claus de longitud 128 bits i, per tant, s'executa en 10 rondes. Aquesta decisió s'ha pres, principalment, per simplicitat i perquè, tot i ser l'opció de clau "menys segura", com bé s'ha estudiat en apartats anteriors cap versió de l'algorisme no es pot considerar insegura; ans al contrari. Per a abreviar, cada vegada que s'anomeni a l'AES, a no ser que es digui el contrari, s'estarà anomenant de retruc a l'AES-128.

2.2.1. Vista general de l'algorisme.

L'AES organitza el *plaintext* en una matriu 4×4 anomenada matriu d'estat o *State Matrix*, separant-lo així en 16 blocs de 1 byte (8 bits) cadascun. De vegades, se solen anomenar a les columnes de 4 bytes de la matriu com a *words*. Totes les operacions de l'algorisme es fan damunt de la *State Matrix*.

Val a dir que la clau de xifrat també se sol organitzar en una matriu 4×4 en determinades operacions, tot i que formalment se sol tractar més com una única paraula de 16 bytes.

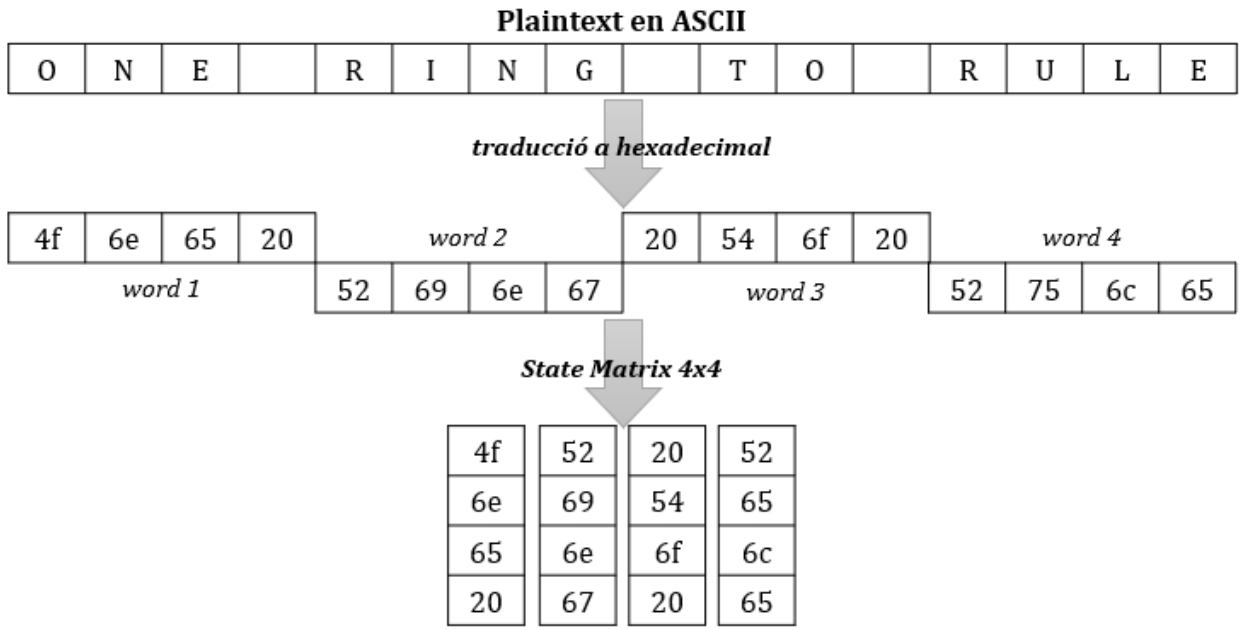


***Figura 16**. Exemple de plaintext i la seva distribució en State Matrix.*

Donada la conversió entre caràcters ASCII i el codi hexadecimal, els bytes de la matriu se solen representar en el darrer codi, tot i que algunes de les operacions de l'algorisme s'entenen millor en binari. Al llarg del treball s'emprarà i s'alternarà entre els dos codis indistintament.

L'AES es podria classificar com un SPN, les sigles en anglès per *substitution–permutation network*. La característica principal d'aquest tipus d'algorisme és que, prenent el *plaintext* i la clau, se'ls aplica una sèrie de rondes que contenen operacions de substitució i de permutació per a produir el *ciphertext*. Normalment, les claus s'introdueixen al final o al principi de cada ronda o bloc.

En el cas de l'AES-128 la *State Matrix* és sotmesa a 10 rondes. Durant cadascuna de les 9 primeres rondes es duen a terme les operacions pròpies de l'AES anomenades *subBytes* (substitució), *shiftRows* i *mixColumns* (permutació) i a*ddRoundKey* (introducció de la clau). En l'última ronda, però, *mixColumns* no és aplicada. Se sol considerar, a més a més, una ronda inicial on tan sols s'implementa l'addició de la clau.

Addicionalment, la clau d'entrada és sotmesa, cada ronda, a l'operació *expandKey*. L'objectiu d'aquesta és d'obtenir la clau de la ronda (*roundKey*) a ser introduïda en l'operació *addRoundKey*, de tal manera que al llarg de l'encriptació es generarien 10 claus de ronda diferents. Aquest procés es podria entendre com el *key schedule* de l'AES.

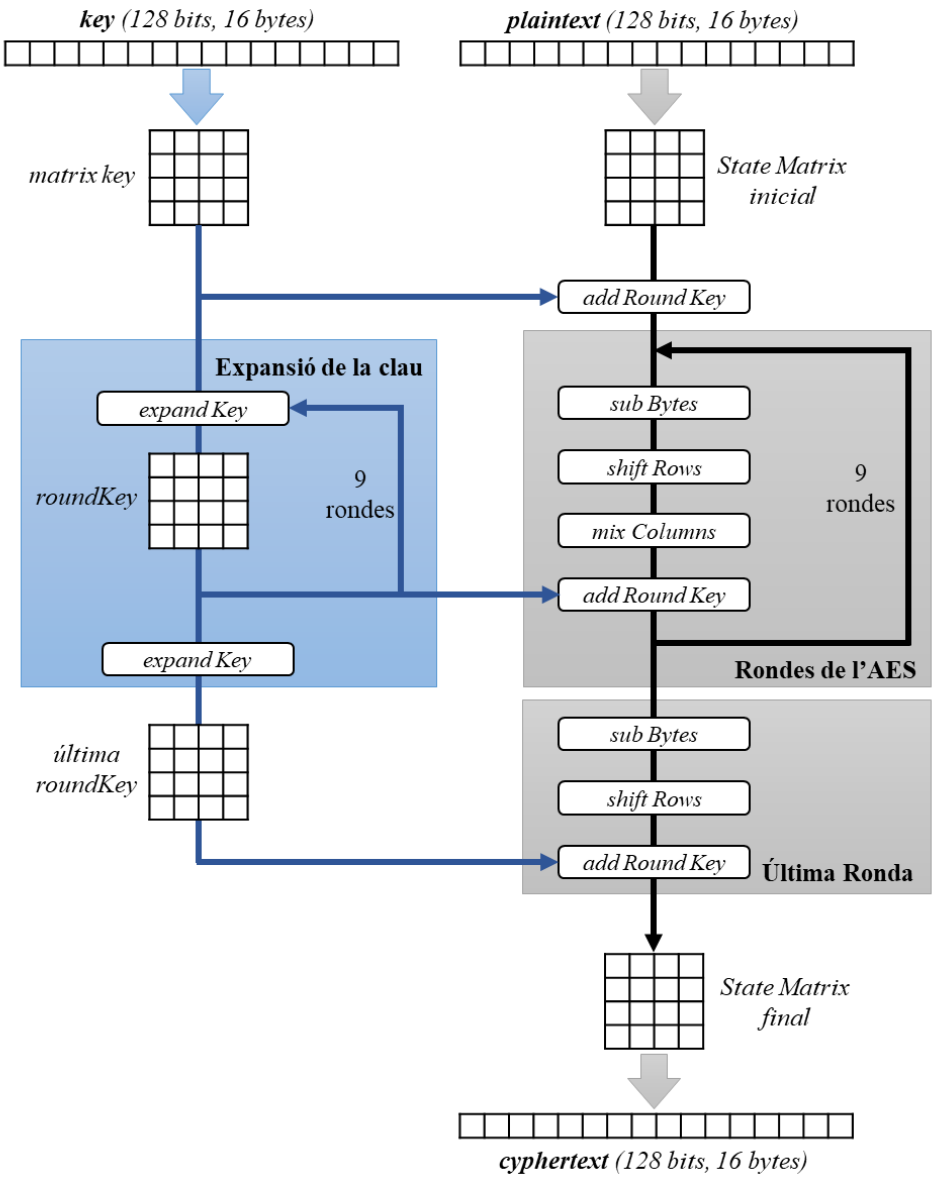


***Figura 17**. Vista general i funcional de l'AES-128, incloent les 10 rondes i l'expansió de la clau.*

2.2.2. Descripció funcional de les operacions.

De seguit es tractaran les diferents operacions que formen una ronda ordinària de l'AES i es comentaran els procediments que s'empren per a transformar la *State Matrix*.

Operació addRoundKey.

L'objectiu d'aquesta funció, com bé indica el seu nom, és afegir la *roundKey* a la *State Matrix* de la ronda actual. Per a fer-ho, cada byte de cada component d'ambdós elements és addicionat mitjançant una *suma exclusiva* o XOR bit a bit. La taula de la veritat d'aquesta darrera operació consta a la *Figura 18*. Cal afegir que l'ordre de la matriu no és alterat, és a dir, cada byte operat retorna a la seva mateixa posició a la nova *State Matrix*.

XOR	0	1
0	0	1
1	1	0

State Matrix			
4f	52	20	52
6e	69	54	65
65	6e	6f	6c
20	67	20	65

$\oplus$

roundKey			
e2	91	b1	d6
32	12	59	79
fc	91	e4	a2
f1	88	e6	93

$=$

nova State Matrix			
ad	c3	91	84
5c	7b	0d	1c
99	ff	8b	ce
d1	ef	c6	f6

***Exemple** (primer byte)*: $\emptyset x4f \oplus \emptyset xe2 = \emptyset b01001111 \oplus \emptyset b11100010 = \begin{array}{r} 01001111 \\ \oplus\ 11100010 \\ \hline 10101101 \end{array} = \emptyset b10101101 = \emptyset xab$

***Figura 18**. Exemple d'operació addRoundKey i taula de la veritat de l'operació suma exclusiva (XOR).*

Operació subBytes.

En criptografia, les funcions algebraiques de substitució solen ser complexes. És per aquest motiu que usualment s'empren matrius o taules de tipus *lookup*, les anomenades *S-box* (abreviació de *substitution box*), amb l'objectiu d'arribar als mateixos resultats emprant una simple indexació. D'aquesta manera, se li redueix molt de temps de càlcul al computador i se li evita haver de fer operacions rebuscades.

L'operació *subBytes* és, com el seu nom indica, l'operació de substitució de l'AES. El seu propòsit és reemplaçar cadascun dels components de la *State Matrix* pel seu homòleg trobat en la *Rijndael S-box*, la matriu de substitució 16×16 creada per a l'AES (per simplicitat, d'ara endavant tan sols se l'anomenarà *S-box*).

El funcionament de l'operació és senzill. Atès que cada byte de la *State Matrix* està format per dues xifres en codi hexadecimal (per exemple, $\emptyset x5c$), s'escull com a byte substitut aquell que es troba en la fila i columna que correspon, respectivament, a les dues xifres del byte de partida (és a dir, segons l'exemple, caldria buscar-lo a la fila 5, columna c).

Aquest funcionament és molt més fàcil d'assimilar de forma gràfica. La *Figura 19* pretén il·lustrar el procés de selecció del byte substitut.

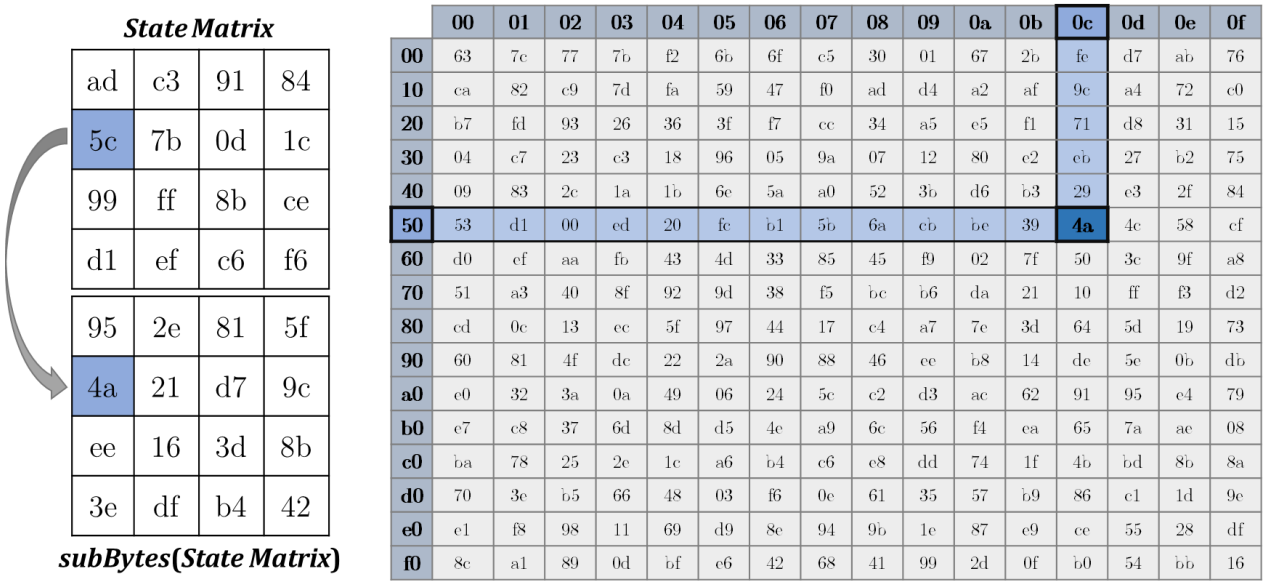


Figura 19. *La Rijndael S-Box (a la dreta) i el procés de substitució del byte $\emptyset x5c$ segons l'operació subBytes.*

Val a dir que l'operació *subBytes* és l'única no lineal de l'AES, tot i que la seva implementació no sigui en format de funció estrictament parlant.

Operació shiftRows.

Aquesta és la primera operació de permutació de la *State Matrix*, l'objectiu de la qual és aportar *difusió* al xifrat. Consisteix, com el seu nom indica, a desplaçar les files de l'algorisme amb l'objectiu de mesclar els bytes les diferents paraules entre si.

La rutina que s'implementa a l'AES és que cada byte es desplaci tantes posicions a l'esquerra (*left shifting*) com el seu nombre de fila menys u. Així doncs, la primera fila es mantindria igual, la segona rotaria una posició a l'esquerra, la tercera rotaria dues i la quarta, tres. La *Figura 20* mostra aquesta operació d'una manera molt més gràfica i entenedora.

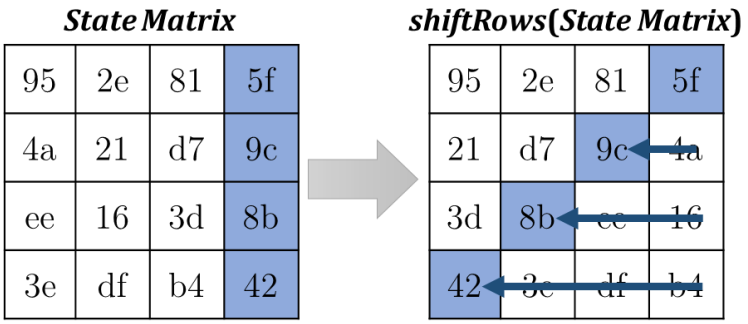


Figura 20. *Operació shiftRows destacant les posicions inicials i finals de l'última columna.*

Vegi's que s'ha emprat la paraula *rotació* en lloc de *desplaçament*. És a dir, si un byte hagués de moure's tres posicions a l'esquerra, però aquella graella de la matriu només conté dues posicions en aquesta direcció, el que farà serà "donar la volta" al voltant de la matriu i ocupar la posició de la dreta del tot.

Operació mixColumns

Aquesta és, conceptualment, l'operació més complicada de l'AES. El seu objectiu és afegir difusió al xifrat mitjançant una operació lineal invertible, en aquest cas, aplicar el producte d'una matriu constant columna a columna. Sigui a_{ij} el byte d'entrada, b_{ij} el de sortida i ij el nombre de fila i columna de la *State Matrix*, respectivament, *mixColumns* es pot escriure com:

$$\begin{bmatrix} b_{0j} \\ b_{1j} \\ b_{2j} \\ b_{3j} \end{bmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix}$$

El producte de dos nombres binaris és, segons l'Àlgebra de Boole, equivalent a aplicar l'AND lògic bit a bit. Tanmateix, el producte expressat en l'equació anterior no es tracta d'una porta AND, ja que les operacions algebraiques de l'AES es fan totes dins d'un camp de Galois.

Un camp finit, anomenat de Galois en honor a Évariste Galois[16], és un cos que conté un nombre finit d'elements. Dins d'aquest camp hi ha definides operacions d'addició, subtracció, multiplicació i divisió que tenen la peculiaritat de mai no retornar un resultat que no formi part del conjunt de nombres finits que hi estan continguts. Aquesta propietat és de gran interès per a l'encriptació binària, ja que permetria poder despreocupar-se del sobreeiximent de bits (i. e. obtenir un nombre de longitud major que la de partida).

Per a executar la *mixColumns* s'empra el $GF(2^8)$, el qual conté 256 elements; el nombre total de bytes existents (un byte té 8 bits, per tant, $2^8 = 256$ possibles combinacions de 0 i 1). En mots més planers, els elements finits del camp són "tots els bytes possibles". Les operacions que es fan servir són, solament, l'addició i la multiplicació.

Addició en el $GF(2^8)$

L'operació d'addició és, simplement, la *suma exclusiva* XOR entre els dos sumands. És lògic, ja que en binari l'únic possible instant en què pot haver-hi sobreeiximent és en sumar dos 1, cosa que és igual a 0 en una XOR (aquesta és també la raó per la qual s'empra una XOR en la *addRoundKey*).

Multiplicació en el $GF(2^8)$

Malauradament, producte no és tan directe. Sigui un byte $B = (b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$, on cada b_i és un dels seus bits, es pot representar en el camp de Galois d'ordre 8 com un polinomi d'ordre 7:

$$B(x) = b_7\ x^7 + b_6\ x^6 + b_5\ x^5 + b_4\ x^4 + b_3 x^3 + b_2\ x^2 + b_1\ x + b_0$$

Per tant, cadascun dels bits del byte esdevindria el coeficient homònim del polinomi, és a dir, aquell del terme de mateix grau que la seva posició en el byte.

[16] Évariste Galois (25 d'octubre de 1811 - 31 de maig de 1832) fou un matemàtic francès nascut a Bourg-la-Reine. A partir dels seus diversos treballs en matemàtiques es va aixecar la que es coneix com a *Teoria de Galois* i es va ajudar a desenvolupar la *teoria de grups*. Morí als 20 anys, ferit en un duel, les raons del qual rauen desconegudes.

Aleshores, la multiplicació es defineix com el producte de dos polinomis $A(x)$ i $C(x)$ representatius dels bytes A i C mòdul el polinomi irreductible de vuitè ordre $irr(x) = x^8 + x^4 + x^3 + x + 1$:

$$B(x) = A(x) \cdot C(x) \qquad \mod irr(x)$$

Cal recordar que un polinomi irreductible d'ordre n és aquell polinomi que no té cap de les seves arrels en els reals $\mathbb{R}$ (és a dir, estan totes en els complexes $\mathbb{C}$). El polinomi $irr(x)$ exerceix de *polinomi base* del camp de Galois. A $GF(2^8)$ n'hi ha molts, però en l'AES es fa servir el mostrat anteriorment.

De forma pràctica, aquesta definició implica que en cas que el resultat de la multiplicació prengués un valor no contingut en el camp finit (és a dir, en termes electrònics, que sobreeixís), caldria fer-li al polinomi resultat mòdul el polinomi $irr(x)$. Aquesta operació, simplement, és prendre com a $B(x)$ el residu de la divisió entre el polinomi $A(x) \cdot C(x)$ i el $irr(x)$.

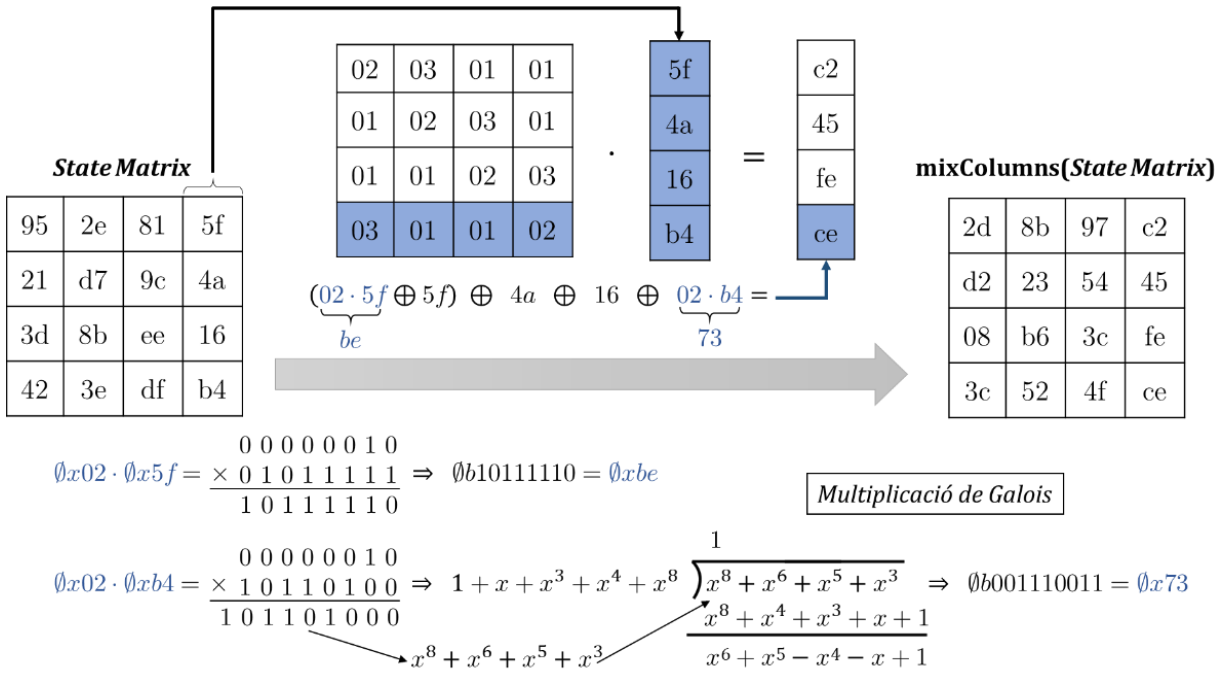
Per exemple, si $A(x) = x^7$ i $C(x) = x$, el seu producte $A(x) \cdot C(x) = x^8$; un nombre de longitud 9 bits. Aleshores, com que sobreïx, caldria aplicar-li mòdul:

$$\begin{array}{r} 1 \\ 1 + x + x^3 + x^4 + x^8 \overline{)\, x^8} \\ \underline{x^8 + x^4 + x^3 + x + 1} \\ x^4 + x^3 + x + 1 \end{array}$$

Per tant, $B(x) = x^4 + x^3 + x + 1$ que correspondria a $b = 00011011$.

Aplicació en el cas de l'AES.

En l'AES, la matriu per la qual cal multiplicar cadascuna de les columnes de la *State Matrix* conté, tan sols, els nombres 1, 2 i 3. D'aquesta manera, es pot simplificar el càlcul del producte de Galois si s'entén que aquests dos nombres es poden escriure com 1, 2 i "2+1" respectivament. Per tant, al cap i a la fi l'únic valor pel qual s'executa el producte és el 2, que en binari s'escriu com $\emptyset b00000010$.



$$\emptyset x02 \cdot \emptyset x5f = \begin{array}{r} 0\,0\,0\,0\,0\,0\,1\,0 \\ \times\ 0\,1\,0\,1\,1\,1\,1\,1 \\ \hline 1\,0\,1\,1\,1\,1\,1\,0 \end{array} \Rightarrow \emptyset b10111110 = \emptyset xbe$$

$$\emptyset x02 \cdot \emptyset xb4 = \begin{array}{r} 0\,0\,0\,0\,0\,0\,1\,0 \\ \times\ 1\,0\,1\,1\,0\,1\,0\,0 \\ \hline 1\,0\,1\,1\,0\,1\,0\,0\,0 \end{array} \Rightarrow 1 + x + x^3 + x^4 + x^8 \Rightarrow \emptyset b001110011 = \emptyset x73$$

$$\begin{array}{r} 1 \\ x^8 + x^6 + x^5 + x^3 \overline{)\, x^8 + x^6 + x^5 + x^3} \\ \underline{x^8 + x^4 + x^3 + x + 1} \\ x^6 + x^5 - x^4 - x + 1 \end{array}$$

$$x^8 + x^6 + x^5 + x^3$$

***Figura 21**. Operació mixColumns sobre una columna (superior) i les seves dues multiplicacions de Galois (inferior)*

La *Figura 21* conté una esquematització de la *mixColumns*, mostrant el procediment de càlcul d'una de les noves columnes. Val a destacar que l'operació mostra la simplificació comentada en l'anterior paràgraf (on $03 = 02 \oplus 01$) i que no s'han escrit els 01, donat que el seu producte amb un hexadecimal és ell mateix. Respecte als dos productes de Galois, cal dir que en el primer ($\emptyset x02 \cdot \emptyset x5f$) no hi ha sobreeiximent i, per tant, no s'aplica mòdul; però en el segon ($\emptyset x02 \cdot \emptyset xb4$) sí que hi ha i, per tant, cal aplicar-lo.

Operació expandKey.

Aquesta és l'única operació que es focalitza exclusivament en la clau de xifratge. L'objectiu de la mateixa és obtenir les deu diferents *roundKey* que seran incorporades en la *addRoundKey*. Normalment, es pot procedir per dos camins: o bé es generen les deu claus abans de començar les rondes i es van administrant adequadament, o bé es generen una a una durant cada ronda.

Sigui K_n la matriu 4×4 on es distribueix la *roundKey* de partida, n essent el número de ronda, i K_n la matriu de la *roundKey* de sortida, la transformació de K_n a K_{n+1} es fa com segueix:

1. Obtenció de la primera columna de K_{n+1}

Es realitzen les següents operacions seqüencialment damunt la *darrera columna de* K_n, que es nota com w_n, amb el propòsit d'obtenir la *primera columna de* K_{n+1}, que es nota com w_{n+1}:

- *RotWord*: es permuten els bytes de la columna una posició cap amunt; una lògica similar a l'operació *shiftRows*.
- *subBytes*: ídem que damunt la *State Matrix*, tot i que en aquest cas només cal substituir els quatre bytes de la columna.
- *addRoundConstant*: es fa la suma exclusiva entre la columna i la columna corresponent de la matriu constant R_{con}, que se selecciona segons el nombre de ronda actual, n.

Finalment, la columna resultat w_{n+1} es calcula com la suma XOR entre la w_n transformada i la *primera columna de* K_n. Aquest procés s'especifica gràficament en la *Figura 22*.

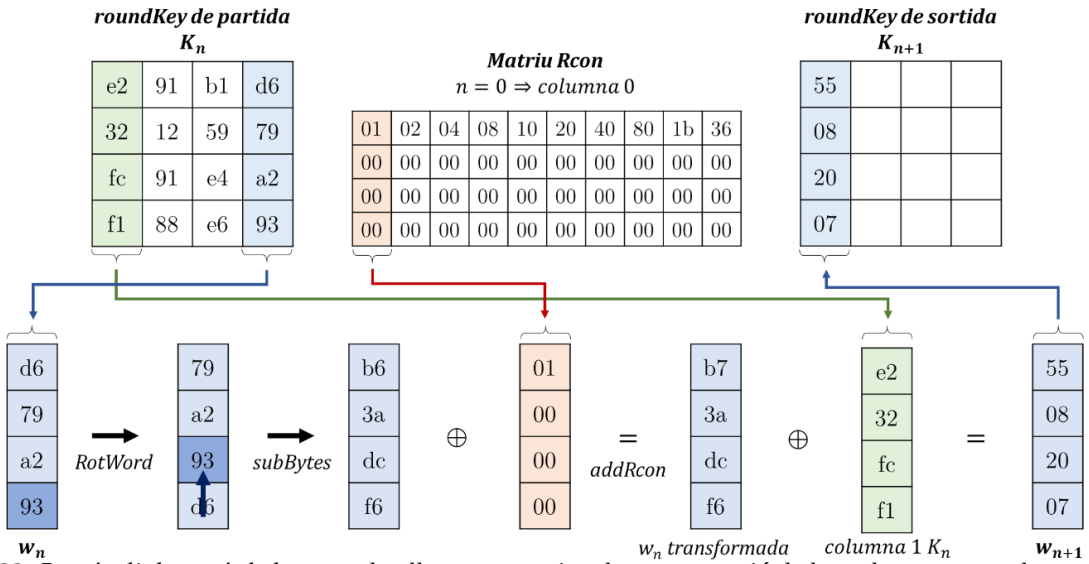


***Figura 22**. Procés d'obtenció de la w_{n+1} detallat pas a pas i amb representació de les columnes per colors coherents.*

2. *Obtenció de la resta de columnes de* K_{n+1}

La resta de les columnes s'obtenen directament a partir de les *columnes corresponents a* K_n sumades XOR amb la respectiva *columna anterior de* K_{n+1}. En la *Figura 23* es pot apreciar de forma visual el funcionament d'aquesta darrera operació.

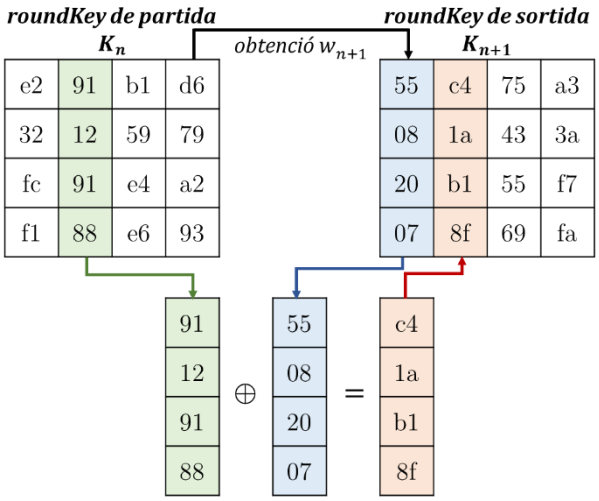


Figura 23. *Procediment d'obtenció de la segona columna de la roundKey de sortida.*

2.3. Exemple d'implementació en software.

Una bona manera de complementar la descripció funcional de l'AES i acabar de comprendre certs aspectes sobre aquesta és implementar-la en software. A més a més, en aquest treball és d'interès tenir-la a l'hora de comprovar el funcionament de la implementació hardware.

Per tant, s'ha decidit d'escriure l'AES en llenguatge Python3. En aquest apartat es farà breu menció sobre la manera amb què s'ha executat aquesta implantació, sense entrar en detalls tècnics ja que no és pas el focus d'aquest projecte, però posant l'accent en l'operació *mixColumns*. El codi és a A.1.

2.3.1. Breu comentari tècnic.

Si bé hi ha moltes maneres de fer-ho, s'ha decidit implementar l'AES en forma de *classe*. En Python3, les classes són objectes que contenen diversos atributs, que solen ser paràmetres o informació concreta, i mètodes interns, ja siguin privats o públics, amb què modificar els dits atributs o retornar resultats de càlculs amb els atributs. També poden contenir mètodes estàtics, que són mètodes que no reben cap argument de referència sobre si els ha cridat una instància de la classe o la pròpia classe.

Per a implementar les matrius *State Matrix* i *roundKey* s'ha optat per emprar objectes tipus llista. És a dir, cada matriu serà representada com una llista de quatre sub-llistes que representen les columnes. Això s'ha fet així per simplicitat, ja que la majoria de les operacions es fan columna a columna.

El codi que s'emprarà serà hexadecimal d'1 byte i, ocasionalment, binari de 8 bits. El Python3 conté la possibilitat d'operar amb aquests codis, en el seu format amb $\emptyset x$ i $\emptyset b$ al davant, però els resultats que retorna sempre són en enter. S'ha decidit d'ignorar-ho, i s'han implementat en format d'string:

$$codi\ binari \rightarrow '001110011' \qquad codi\ hexadecimal \rightarrow '73'$$

S'han escrit un seguici de funcions que permeten operar amb aquests valors i fer les conversions pertinents entre *bin-hex-int*, a l'apartat A.1.3. D'aquesta manera, la implementació de la classe esdevé molt més simple. Es poden consultar en l'apartat pertinent dels *Annexos*, el A.1.2.

L'especificació de la classe AES, també es troba la *Taula A.* **1**, als *Annexos*.

2.3.2. Simplificacions pràctiques de les operacions de l'AES.

En l'apartat 2.2 s'ha tractat la descripció funcional teòrica de l'AES, però no s'ha parlat sobre com aquesta es tradueix a la pràctica. Si bé les operacions *subBytes*, *addRoundKey* i *expandKey* se solen implementar sempre seguint al detall els seus aspectes funcionals, les operacions *shiftRows* i *mixColumns* poden ser reinterpretades en algorismes més pragmàtics.

Abans de començar, cal mencionar els mòduls importats ja que apareixeran en el codi mostrat:

```
import operacionsbinhex as op
from matrius import *
```

Són, respectivament, el mòdul que conté les funcions entre *bin-hex-int* comentat anteriorment (disponible en A.1.2) i un altre que tan sols emmagatzema les diferents matrius internes que empra l'AES (la *S-box*, la *Rcon* i la constat del *mixColumns*).

Simplificació pràctica de la shiftRows.

Aquest mètode s'ha implementat mitjançant una traducció algorísmica de l'operació *shiftRows* que s'ha considerat destacable.

El Python3 permet implementar la permutació de files de forma directa mitjançant la indexació. En aquest llenguatge els índexs es poden expressar, per a longitud $n+1$, com $0, 1, 2, 3 \dots n$ en el sentit creixent i com $-(n+1), -n, \dots, -2, -1$ en el sentit també creixent. Com que la *shiftRows* involucra una "rotació al voltant de la matriu", la combinació d'ambdós formats permet implementar la permutació:

```
def shiftRows (self):

        for i in range(4):
                self.nouestat[i] = self.estat[i-4][i], self.estat[i-3][i],
                                   self.estat[i-2][i], self.estat[i-1][i];

        self.estat = self.transpose(self.nouestat)
        self.clearnouestat()
```

Com es pot comprovar, cada byte de cada nova columna (`self.nouestat[i]`) esdevé el byte de la columna d'entrada(`self.estat[k][i]`) de la posició k, entenent que aquesta es calcula com:

$$k = i - (4 - num_{fila})$$

Així doncs, per $i = 0$, obtindríem els índex $-4,\ -3,\ -2,\ -1$; per tant, la fila no permutaria. En canvi, per $i = 2$, obtindríem els índex $-2,\ -1,\ 0,\ 1$; per tant, la fila permutaria dues posicions a l'esquerra. Això correspon amb les rotacions pròpies del shiftRows.

Simplificació pràctica de la mixColumns.

L'operació *mixColumns* és la conceptualment més complexa de l'AES. Tanmateix, és possible dissenyar-la com un algorisme molt pràctic per a la computació que prescindeix de càlculs polinòmics o algebraics. Això serà de vital importància quan s'implementi en hardware.

En Python3, s'ha implementat com un mètode, igual que la resta d'operacions pròpies de l'AES. Cal recordar que la multiplicació de Galois de l'AES es pot reduir a una multiplicació entre el byte i el valor 02 en hexadecimal. La matriu constant, anomenada *C* en el codi, s'ha definit de la mateixa manera que la *State Matrix*, però per files en lloc de columnes per a facilitar el producte.

```
C = [['02','03','01','01'], ['01','02','03','01'], ['01','01','02','03'],
     ['03','01','01','02']]
```

El codi emprat ha estat:

```
    def mixColumns (self):

        for i in range(4):
            for j in range(4):
                r = 0
                Cj = C[j] #<-----------columna de la matriu C
                Mi = self.estat[i]#<---columna actual de la matriu M
                for h in range(4):
                    if Cj[h] == '02':
                        r ^= int(self.Double(Mi[h]),16)
                    elif Cj[h] == '03':
                        r ^= int(op.XORhex(Mi[h], self.Double(Mi[h])),16)
                    elif Cj[h] == '01':
                        r ^= int(Mi[h],16)
                self.nouestat[i].append(op.int2hex(r))

        self.estat = self.nouestat
        self.clearnouestat()
```

Per a fer la multiplicació s'empra el comptador *r* que, a cada iteració, es suma XOR amb el resultat del producte byte a byte i ell mateix. Per a fer-ho, es converteix el resultat a base 10 (enter) per exigències del llenguatge Python3. Es distingeix entre tres casos:

Producte per 01 (`r ^= int(Mi[h],16)`).

En aquest cas, simplement, el resultat del producte és el byte immutat de la *State Matrix*.

Producte per 02 (`r ^= int(self.Double(Mi[h]),16)`).

Per a computar-lo es recorre al mètode estàtic *Double*, el qual implementa la multiplicació de Galois pel número 02 donat un nombre binari. Aquesta és la funció que implementa l'algorisme simplificat de què s'estava parlant. El codi és el següent:

```
    @staticmethod
    def Double(b):

        b = op.hex2bin(b)
        carry_set = b[0] == '1'
        b = b[1:] + '0'
        if carry_set: b = op.XORbin(b,op.hex2bin('1b'),n=8)
        return op.bin2hex(b)
```

El producte de Galois que s'implementa a *Double* no es fa amb la seva representació polinòmica, com s'ha explicat en l'apartat 2.2.2, sinó que s'empren propietats del codi binari per a optimitzar el càlcul. D'entrada, cal transformar el byte *b* de codi hexadecimal a binari (funció *hex2bin* del mòdul *operacionsbinhex*), i aleshores es calculen els següents passos:

1. Es comprova si el MSB de *b* és 0 o 1, i s'emmagatzema el resultat d'aquesta comprovació en la variable *carry_set*.
2. Es desplaça una posició a l'esquerra el byte *b*, ja que això és l'equivalent de calcular el producte d'un nombre binari per 2.
3. Es comprova el valor de *carry_set*. Si aquest és igual a 1, implica que el producte resulta en sobreeiximent, de tal manera que es calcula la XOR del byte *b* desplaçat i el nombre hexadecimal $\emptyset x1b$.

El tercer pas és l'equivalent a aplicar mòdul el polinomi $irr(x) = x^8 + x^4 + x^3 + x + 1$, que escrit en binari és $\emptyset b100011011$. En aquest cas, s'aplica el $\emptyset b00011011$, $\emptyset x1b$ en hexadecimal, ja que es menysté el bit numero 9 ($1 \oplus 1 = 0$).

La lògica d'aquesta implementació serà visitada de nou a l'apartat 3.2, en el disseny de la implementació hardware d'aquesta funció.

Producte per 03 (`r ^= int(op.XORhex(Mi[h], self.Double(Mi[h])),16)`).

En aquest cas es desglossa l'operació en dues parts, com es va comentar anteriorment, ja que es pot escriure que $03 = 02 \oplus 01$. Per tant, es calculen i se sumen XOR els casos de *producte per 01* i *producte per 02* per a obtenir aquest cas.

CAPÍTOL 3. Disseny d'una implementació hardware de l'AES.

En el present capítol s'exposarà com s'ha dut a terme la implementació hardware de l'algorisme AES-128. Es tractarà el circuit en alt nivell, mostrant les diferents parts que el conformen, com s'organitzen i quines funcions fan, així com una inspecció en detall dels mòduls de llenguatge descriptiu que els implementen.

3.1. Enfocament i aspectes previs.

Seguidament es parlaran de certs aspectes previs amb l'objectiu de contextualitzar la implementació realitzada, donar detalls sobre l'enfoc emprat en la seva realització i d'altres que són necessaris de comentar abans de llançar-se a l'arquitectura.

3.2.1. Sobre el llenguatge descriptiu emprat.

D'entre els diferents llenguatges descriptius HDL existents, s'ha decidit d'emprar el VHDL de l'IEEE. Entre els diferents motius pels quals s'ha escollit aquest destaca, des d'una perspectiva personal, que es tracta d'un llenguatge molt verbós. És a dir, el VHDL té una sintaxi que opera amb multitud de paraules reservades i requereix, en general, de més línies de codi que, per exemple, el *Verilog*; el qual tendeix a ser escarit i breu. Entenent que aquest treball es contextualitza en un primer contacte amb els llenguatges descriptius, aquesta verbositat del VHDL ofereix una més còmoda implementació a nivell de programador.

Donat que s'ha volgut definir l'arquitectura al detall en bona part dels mòduls escrits, s'ha decidit encarar la descripció de forma *estructural*. No obstant això, algun d'ells ha estat redactat amb un enfoc que podria considerar-se més com a *comportamental*. Això ha estat així ja que s'ha buscat la simplicitat en determinats blocs funcionals en els quals l'arquitectura de baix nivell era poc important. El nivell d'abstracció ha estat entre *gate-level* i *registre-transfer* RTL (veure 1.3.3 per a més detall)

Cal comentar, a més a més, que el disseny hardware ha estat pensat per a ser implementat en una FPGA, com es comenta en més detall en l'apartat 4.1.

3.2.2. Enfocament de la lògica dels circuits.

Un circuit electrònic es pot plantejar seguint diverses lògiques diferenciades. Per al disseny en qüestió, tanmateix, s'han considerat únicament un parell d'opcions: la lògica *combinacional* i la *seqüencial síncrona*. Des d'un punt de vista de hardware, les diferències principals entre les dues rauen, bàsicament, en l'ocupació d'àrea i el temps de computació:

- *Circuit combinacional*: les operacions són funcions lògiques que es calculen "instantàniament" atès que les seves sortides depenen només de les entrades (el temps de computació serà el retard acumulat del senyal a mesura que passa per tots els components del circuit). Per tant, si bé s'executen d'un sol cop, a nivell de hardware s'ocupa molta àrea ja que no es poden reaprofitar interfícies, havent-se d'instanciar tantes vegades com sigui necessari.

- *Circuit seqüencial síncron* (*un sol senyal de rellotge*): les sortides són no només funció de les entrades, sinó que també ho són de l'estat anterior del sistema. En termes pràctics, el càlcul de la resposta es fa per etapes tal que cadascuna d'elles es calcula en el mateix circuit físic, en bucle. Normalment, el temps de computació serà major que en el cas combinacional. L'ocupació d'àrea, però, seria molt menor ja que no caldria instanciar un mateix bloc tantes vegades com etapes hi hagués, sinó que amb un sol espai físic on anar-les calculant una a una seria suficient.

L'arquitectura de la implementació feta per a aquest treball es distingeix en dos nivells; baix i alt. Per al primer, s'ha considerat oportú emprar lògica combinacional a l'hora de dissenyar les operacions que la componen. Si bé l'ocupació d'àrea serà gran, a la pràctica aquesta és del tot minúscula donat que en l'FPGA s'empraran molt pocs recursos, com es veurà en l'apartat de síntesi. Per tant, interessa sacrificar l'àrea per tal d'optimitzar el temps de computació al màxim.

D'altra banda, l'arquitectura d'alt nivell ha estat dissenyada amb lògica seqüencial síncrona. L'AES disposa de 10 rondes, cadascuna d'elles havent-se de calcular segons les mateixes operacions (vistes al 2.2.2), fent que no sigui gaire òptim haver d'instanciar cadascuna d'elles deu vegades. Això implica que farà falta el disseny de registres paral·lels i unitats de control, de les quals es parlarà més endavant, a més a més de la inclusió d'un senyal de rellotge síncron.

3.2.3. Package d'VHDL propi.

Per tal de facilitar l'escriptura dels mòduls en VHDL que conformen la descripció de la implementació hardware de l'AES, s'ha cregut convenient crear un *Package* anomenat *my_data_type*. En ell, es defineixen tres *datatypes* de senyals que seran de gran ajuda, sobretot a l'hora d'indexar:

- *byte_word*: és una *array* de 4 bytes pensada per a ser una columna de la *State Matrix*.
- *state_matrix*: és una *array* de 16 bytes que pretén ser el *datatype* propi de les *State Matrix* i de les claus en la seva forma matricial. Serà el senyal més freqüent i el més emprat.
- *state_name*: codifica el nom de tres estats per la màquina d'estats finits que es veurà a l'apartat 3.2.1.

Per tant, la indexació en *byte_word* i *state_matrix* serà byte a byte, tal com treballa l'AES. El fet de treballar amb aquest *datatypes* no té implicacions en la síntesi; només serveix per a la facilitació de l'escriptura del codi VHDL. Seguidament es pot observar el codi:

```
library ieee;
use ieee.std_logic_1164.all;

package my_data_type is

  type byte_word is array(3 downto 0) of std_logic_vector(7 downto 0);
  type state_matrix is array(15 downto 0) of std_logic_vector(7 downto 0);
  type state_name is (IDLE, ENCR, DELIV);

end package;
```

També es troba disponible al seu respectiu apartat dels *Annexos*, A.2.13.

3.2. Arquitectura de baix nivell.

Es presentarà i justificarà de seguit el disseny en VHDL de les diferents operacions de l'AES i altres mòduls de baix nivell que s'han dissenyat. L'objectiu serà fer una visita *bottom-up* del circuit descrit perquè, a més de ser el sentit amb què s'ha fet la descripció, permetrà un millor enteniment de l'arquitectura de nivell alt, que es veurà en el pròxim apartat.

Abans d'entrar en matèria, caldria adreçar com s'han tractat els senyals principals del sistema: la *State Matrix* i la *roundKey*. En ambdós casos, s'ha emprat el *datatype* personalitzat *state_matrix* vist al 3.2.3, tal que cadascun dels 16 bytes es representi en el codi sempre en hexadecimal. En VHDL es pot escriure un *std_logic* en hexadecimal emprant la sintaxi: $x"h_1h_2"$, on cada h_i representa un dels dos caràcters en codi hexadecimal amb què s'escriuen els bytes.

Per a estandarditzar-lo, s'ha decidit que l'ordre dels bytes dins de l'*array* del *datatype state_matrix* sigui el convencional en l'AES:

$$SM = \begin{bmatrix} b_{15} & b_{11} & b_7 & b_3 \\ b_{14} & b_{10} & b_6 & b_2 \\ b_{13} & b_9 & b_5 & b_1 \\ b_{12} & b_8 & b_4 & b_0 \end{bmatrix} \quad \Rightarrow \quad signal\ SM\colon state\_matrix \equiv (b_{15}, b_{14}, \dots, b_1, b_0)$$

És a dir, cada grup de 4 bytes del senyal serà una de les quatre columnes de la *State Matrix* algebraica.

Seguint amb aquesta idea de l'estandardització, s'ha imposat que cadascun dels senyals d'entrada i de sortida a un mòdul, en cas de ser una *State Matrix* o una *roundKey*, s'escriguin sempre de la següent manera: *in_STATE_MATRIX, out_STATE_MATRIX, in_ROUND_KEY, out_ROUND_KEY.*

Finalment, cal parar atenció damunt de la *Taula 2*. En ella hi ha especificades les diferents entitats de baix nivell que s'han dissenyat, així com els seus ports d'entrada i sortida amb el nom del senyal i especificant la longitud d'aquests en bits.

	Nom	***Inputs***	***Outputs***
OPERACIONS PRÒPIES DE L'AES	*subBytes* *shiftRows* *mixColumns*	*in_STATE_MATRIX (128 bits)*	*out_STATE_MATRIX (128 bits)*
	addRoundKey	*in_STATE_MATRIX (128 bits)* *in_ROUND_KEY(128 bits)*	*out_ROUND_KEY (128 bits)*
	expandKey	*in_ROUND_KEY, (128 bits)* *num_rond (4 bits)*	*out_ROUND_KEY (128 bits)*
SUB-BLOCS DE LES OPERACIONS	*galoisDouble Sbox*	*in_byte (1 bit)*	*out_byte (1 bit)*
REGISTRE PARAL·LEL	*reg_ff*	*E, CLR, clk (1 bit)* *D (128 bits)*	*Q (128 bits)*

Taula 2. *Entitats de baix nivell segons la seva tipologia i amb els seus senyals d'entrada i sortida.*

3.2.1. Operacions de l'AES.

Seguidament, es presentaran les diferents implementacions de les operacions pròpies de l'AES. Donat que ja s'ha vist en profunditat el funcionament descriptiu d'aquestes, s'intentarà no redundar en les explicacions. En cas que sigui necessari, es remet el lector a l'apartat 2.2.2.

Arquitectura de la subBytes.

El plantejament en VHDL d'aquesta operació ha requerit de dues entitats: *subBytes* i *Sbox*. La primera el que fa és, simplement, enviar byte a byte tots els elements de la *State Matrix* d'entrada a la *Sbox* i, un cop substituïts, els connecta amb la *State Matrix* de sortida. El seu codi complet pot trobar a l'apartat corresponent dels *Annexos*, A.2.6.

L'entitat que cal comentar amb detall és l'*Sbox*, que és l'encarregada d'implementar la substitució dels bytes per se, el codi complet de la qual es pot trobar a l'apartat dels *Annexos* A.2.12. Per a descriure'n l'arquitectura amb VHDL, s'ha emprat la sentència *when select* com segueix:

```
architecture arch of Sbox is
begin
  with in_byte select
        out_byte <=
        x"63" when x"00",
        x"7c" when x"01",
        -- (la resta de valors de la taula)
        "XXXXXXXX" when others;
end;
```

El que aquest codi està descrivint, en realitat, és un multiplexor de 256 línies a 1 amb un senyal de selecció de 8 bits (tal que $2^8 = 256$). Cadascuna de les línies seria un dels valors de la *S-box* i el senyal de selecció, el byte de la *State Matrix* a substituir.

A la *Figura 24* es pot veure l'esquema del circuit digital que implementa la *subBytes*. A l'esquerra hi ha els 16 bytes provinents de la *State Matrix* d'entrada i, a la dreta, els que van a parar a la de sortida. Cal destacar l'existència de 16 instàncies de la *Sbox*, una per cada byte.

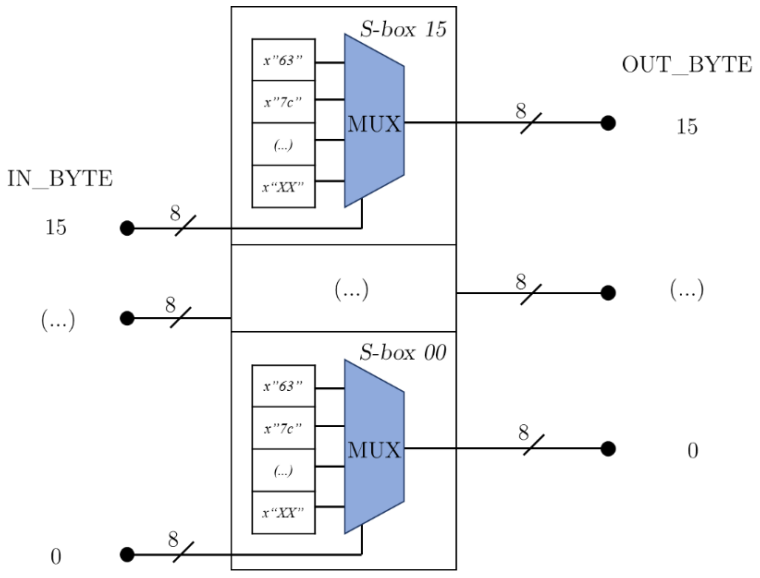


Figura 24. *Circuit esquemàtic de la implementació de la subBytes de forma combinacional amb 16 S-box.*

Com s'ha comentat a l'apartat 3.2.2, l'entitat *subBytes* està construïda amb lògica combinacional. És a dir, atès que els 16 bytes de la *State Matrix* d'entrada han de passar per la *Sbox*, és necessari que hi hagi una instància d'aquesta entitat per a cada byte. Això lliga amb el comentat sobre la gran ocupació d'àrea dels circuits combinacionals.

Per a generar les 16 instàncies de la *Sbox* s'ha emprat la sentència *generate* a l'arquitectura de l'entitat *subBytes* com segueix:

```
gen: for indx in 0 to 15 generate
        pm: entity work.Sbox port map(
          in_byte => in_STATE_MATRIX(indx),
          out_byte=> out_STATE_MATRIX(indx)
          );
```

A tall d'informació addicional, val a dir que l'entitat *Sbox* es pot entendre com una memòria ROM (*Read Only Memory*) de capacitat $2^8 \cdot 8 = 2048\ bits$ donat que implementa una funció lògica (la de la substitució) amb valors memoritzats en una *lookup table* o taula de consulta, que es pot interpretar com un conjunt de cel·les de memòria.

Arquitectura de la shiftRows.

Per a fer el seu disseny només ha fet falta una sol'entitat: *shiftRows*. La implementació d'aquesta tan sols requereix de descriure la manera amb què els diferents bytes de la *State Matrix* d'entrada i de sortida s'enllacen segons els *shiftings* propis de l'operació. La simplificació pràctica vista a l'apartat 2.3.2 no és vàlida en VHDL ja que la indexació en aquest llenguatge només és amb enters positius.

El codi complet es pot consultar a l'apartat A.2.7 dels *Annexos*. Seguidament es pot veure com quedaria l'assignació a la *State Matrix* de sortida dels 4 primers bytes (primera columna) de la d'entrada:

```
out_STATE_MATRIX(15)<= in_STATE_MATRIX(15);
out_STATE_MATRIX(14)<= in_STATE_MATRIX(10);
out_STATE_MATRIX(13)<= in_STATE_MATRIX(5);
out_STATE_MATRIX(12)<= in_STATE_MATRIX(0);
```

A la *Figura 25* s'ha representat el circuit digital que s'obtindria segons l'arquitectura proposada. S'ha distingit amb colors els bytes pertanyents a cadascuna de les files de les dues *State Matrix*.

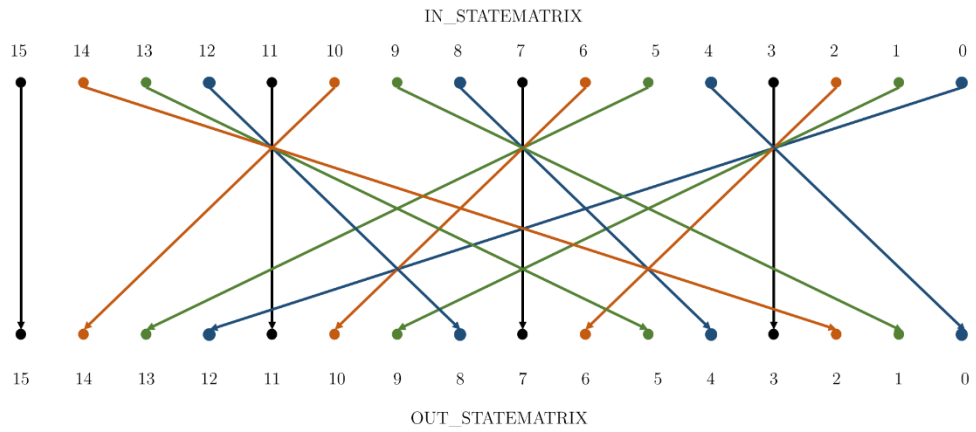


***Figura 25**. Mapejat de bytes propi de la shiftRows, on s'han distingit les diferents files mitjançant un color diferent per a ajudar a la comprensió (negre: fila 0, taronja: fila 1, verd: fila 2, blau: fila 3)*

Arquitectura de la mixColumns.

De manera similar a la *subBytes*, han calgut un parell d'entitats per a implementar aquesta operació: *mixColumns* i *galoisDouble*. Ambdues són d'importància i, en conseqüència, s'exposaran una a una.

<u>*Disseny del producte matricial*</u> (entitat *mixColumns*)

L'entitat *mixColumns* és on s'ha descrit la implementació del producte entre les diferents files de la matriu constant de l'operació i les diferents columnes de la *State Matrix* d'entrada. Des d'un punt de vista algebraic, es podria escriure que aquest producte, en forma de sistema lineal (recordant que $03 = 02 + 01$), és el següent:

$$
\begin{aligned}
b_{4j} &= 02 \cdot a_{4j} \oplus (02 \cdot a_{3j} \oplus 01) \cdot a_{3j} \oplus 01 \cdot a_{2j} \oplus 01 \cdot a_{1j} \\
b_{3j} &= 01 \cdot a_{4j} \oplus 02 \cdot a_{3j} \oplus (02 \cdot a_{2j} \oplus 01) \cdot a_{2j} \oplus 01 \cdot a_{1j} \\
b_{2j} &= 01 \cdot a_{4j} \oplus 01 \cdot a_{3j} \oplus 02 \cdot a_{2j} \oplus (02 \cdot a_{1j} \oplus 01 \cdot a_{1j}) \\
b_{1j} &= (02 \cdot a_{4j} \oplus 01 \cdot a_{4j}) \oplus 01 \cdot a_{3j} \oplus 01 \cdot a_{2j} \oplus 02 \cdot a_{1j}
\end{aligned}
$$

Fixant-se en aquest sistema, s'ha observat que tots els bytes de la columna són multiplicats per 02 en algun moment o altre i, per extensió, també ho són tots els de la matriu. Conseqüentment, s'ha estimat oportú de calcular de cop la *State Matrix* sencera multiplicada per 02. El resultat d'aquesta multiplicació s'ha definit com el senyal intern *doubled* (`signal doubled: state_matrix`).

Aleshores, el producte per a cada columna es pot descriure en VHDL tal com es mostra en el sistema d'equacions anterior, interpretant cada $02 \cdot a_{ij}$ com el byte corresponent del senyal *doubled*:

```
col_mult: for i in 0 to 3 generate
         out_STATE_MATRIX(15-4*i) <= doubled(15-4*i) xor doubled(14-4*i) xor
                                     in_STATE_MATRIX(14-4*i) xor in_STATE_MATRIX(13-4*i)
                                     xor in_STATE_MATRIX(12-4*i);

          out_STATE_MATRIX(14-4*i) <= in_STATE_MATRIX(15-4*i) xor doubled(14-4*i) xor
                                     doubled(13-4*i) xor in_STATE_MATRIX(13-4*i)
                                     xor in_STATE_MATRIX(12-4*i);

          out_STATE_MATRIX(13-4*i) <= in_STATE_MATRIX(15-4*i) xor in_STATE_MATRIX(14-4*i)
                                     xor doubled(13-4*i) xor doubled(12-4*i)
                                     xor in_STATE_MATRIX(12-4*i);

          out_STATE_MATRIX(12-4*i) <= doubled(15-4*i) xor in_STATE_MATRIX(15-4*i) xor
                                     in_STATE_MATRIX(14-4*i) xor in_STATE_MATRIX(13-4*i)
                                     xor doubled(12-4*i);
  end generate;
```

Cal notar que, en emprar la sentència *generate* i el *datatype state_matrix*, una bona manera de fer les indexacions és implementar-ho tot per la primera columna (bytes del 15 al 12) i després restar-los $4 \cdot i$ (és a dir, la distància entre ells i el byte homònim de la columna i). La resta del codi es pot consultar a l'apartat pertinent dels *Annexos*, A.2.8.

Seguidament, a la *Figura 26*, es mostra la representació en forma de circuit d'aquesta multiplicació per una sola columna. Pel mateix motiu exposat en el cas de la *S-box*, tot el bloc de la figura es generarà una vegada per columna, és a dir, un total de quatre vegades.

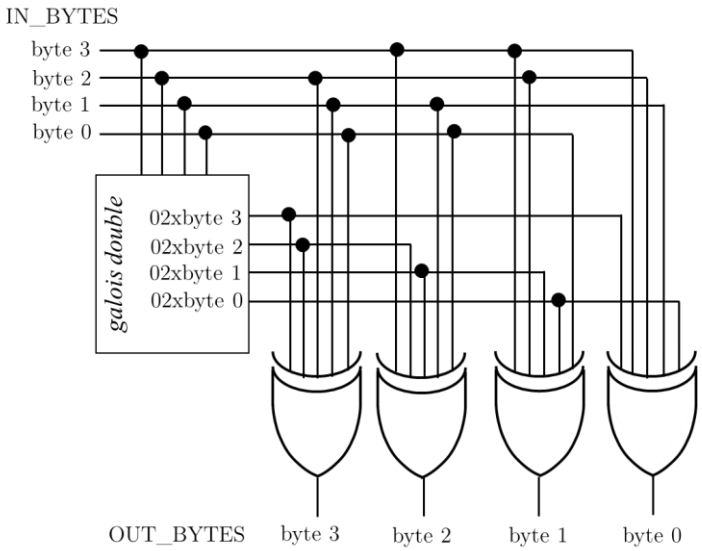


***Figura 26**. Esquemàtic de la multiplicació d'una de les quatre columnes de la State Matrix. S'instancia 4 vegades.*

<u>*Disseny del producte de Galois*</u>. (entitat *galois_double*)

El disseny en general que s'ha seguit és el de la simplificació pràctica vista en l'apartat 2.3, que en Python3 era el mètode *Double*. És a dir, considerant que la multiplicació entre un nombre binari i 02 és equivalent a un *shifteig* d'una posició a l'esquerra, en cas de sobreeiximent cal aplicar la suma XOR del nombre *shiftat* amb el valor $\emptyset x1b$ (i. e. el polinomi base del camp $GF\ 2^8$ de l'AES). La manera amb què s'ha dissenyat aquesta idea, però, té alguns matisos que cal comentar.

Per tal d'unificar el procés i simplificar, es pot considerar que una suma XOR amb el valor $\emptyset x1b$ (en binari $\emptyset b00011011$), tan sols afecta les posicions 0, 1, 3 i 4 del byte *shiftat*. Aquestes posicions són les que coincideixen amb un 1 en la seva XOR bit a bit amb $\emptyset x1b$. Aquesta afirmació és certa ja que $b \oplus 0 = b$ i $b \oplus 1 = \overline{b}$; és a dir, només la XOR amb 1 altera el bit amb què se suma.

Com que el MSB del byte d'entrada és el que indica si hi haurà sobreeiximent (si és igual a 1 n'hi ha, si és 0, no), se'l fa servir per "simular" la suma XOR amb el valor $\emptyset x1b$. En la *Figura 27* es troba el circuit lògic que ho implementa. El codi VHDL, a l'apartat A.2.9 dels *Annexos*.

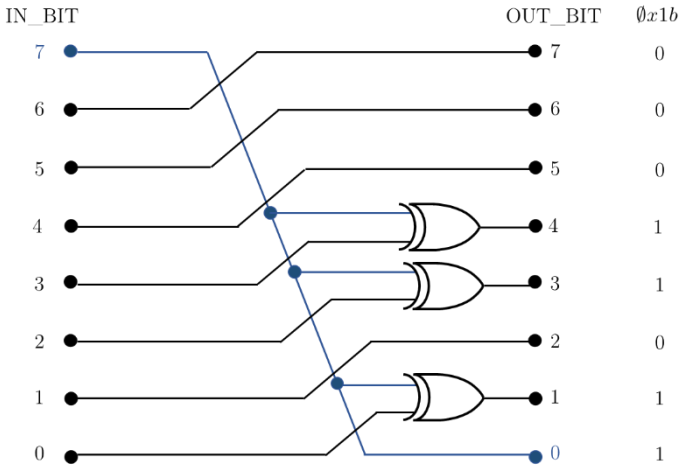


***Figura 27.** Esquemàtic de la multiplicació de Galois per 02 amb portes lògiques i shifteig. S'ha inclòs a la dreta el byte $\emptyset x1b$ en binari, de tal manera que es pot veure que els bits 1 del byte coincideixen amb les XOR amb el MSB.*

Arquitectura de la expandKey.

Per a l'operació d'expansió de la clau s'ha descrit una entitat, *expandKey*, i s'ha aprofitat una de ja escrita, la *Sbox*. En lloc de calcular prèviament totes les *roundKey*, s'ha escollit implementar aquesta operació tal que a cada ronda es generi la clau la de la pròxima ronda. El codi font en VHDL es pot consultar a l'apartat A.2.10, als *Annexos*.

Abans de computar les diferents funcions internes de la *expandKey*, cal triar la columna de la matriu *Rcon* que es correspon amb la ronda actual. Hi ha moltes maneres de fer-ho, però la que més s'adequa a aquest treball és emprar la sentència *when select* de la següent manera:

```
num_rond select
    Rcon<=
        (x"01",x"00",x"00",x"00") when "0001",
        (x"02",x"00",x"00",x"00") when "0010",
        -- La resta de les columnes
        (x"00",x"00",x"00",x"00") when others;
```

És a dir, s'ha implantat com un multiplexor de senyal de selecció *num_rond*, un nombre de 4 bits (*nibble*) que indicaria el nombre de ronda actual. Aquest senyal és matèria de discussió de l'apartat venidor 3.2.1. Per ara, només cal saber que és un input de la *expandKey*.

Seguidament, es procedirà a explicar l'arquitectura de les dues parts que conformen l'operació.

<u>*Obtenció de la primera columna de la roundKey de sortida.*</u>

Per a fer la *RotWord* i la *subBytes* (32 bits) s'ha seguit la mateixa lògica que amb la *shiftRows* i la *subBytes* (128 bits), ambdues ja mostrades. Per tant, primer s'executa el *shifteig* i després es passen els bits resultants per l'entitat *Sbox*. Donat que es treballa amb una sola columna, s'hauran d'instanciar 4 *Sbox*.

Finalment, es fa la suma XOR entre la columna *Rcon* seleccionada prèviament i la primera columna de la *roundKey* d'entrada. En la *Figura 28* es pot veure de forma gràfica aquesta implementació.

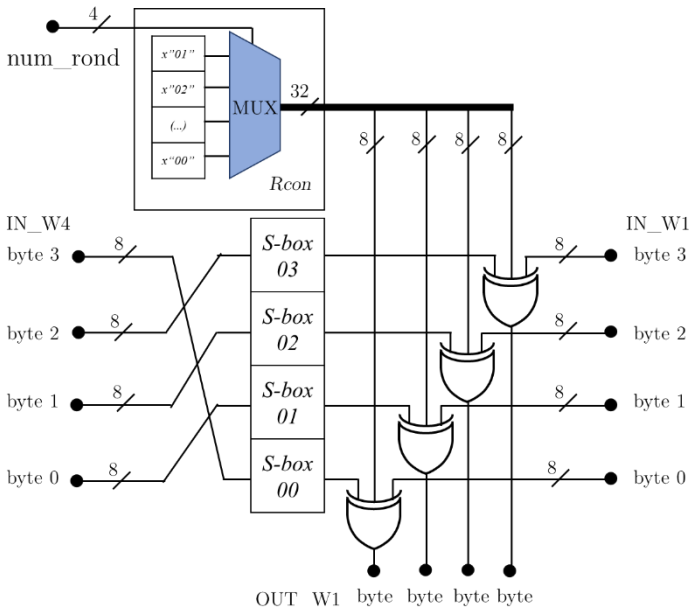


Figura 28. *Esquemàtic de l'obtenció de la primera columna de roundKey de sortida.*

Obtenció de la resta de columnes de la roundKey de sortida.

Per a tal propòsit, tan sols cal fer la suma XOR de la columna anterior amb la columna homònima de la *roundKey* d'entrada. Es parteix sempre de la primera columna de la *roundKey* de sortida calculada amb anterioritat. Per a veure-ho amb més facilitat, s'ha elaborat la *Figura 29*.

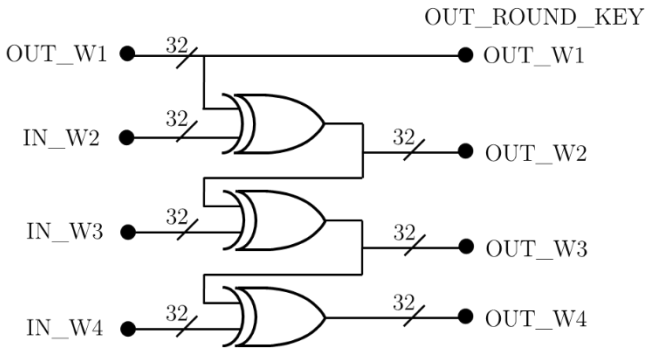


Figura 29. *Esquemàtic de la cascada de portes XOR per al càlcul de la resta de columnes de roundKey.*

Del codi font emprat per al càlcul de la *expandKey* cal destacar que, atès que el VHDL no permet operar amb els senyals de sortida, s'han creat com a senyals interns les diferents columnes (o *words*) de les *roundKey* d'entrada i sortida, s'ha operat amb elles mitjançant la sentència *generate*, i finalment s'han connectat amb les columnes pertinents de la *roundKey* de sortida:

```
rot_in_W4 <= (in_ROUND_KEY(2), in_ROUND_KEY(1), in_ROUND_KEY(0), in_ROUND_KEY (3));

  --Operació SubBytes() i generació de les columnes d'out_ROUND_KEY

  substitucio: for byte in 3 downto 0 generate
    pm: entity work.Sbox port map(
          in_byte => rot_in_W4(byte),
          out_byte=> sub_in_W4(byte));

        -- Generació primera columna
        out_W1(byte) <= sub_in_W4(byte) xor in_ROUND_KEY(12+byte) xor Rcon(byte);

        -- Generació resta de columnes
        out_W2(byte) <= out_W1(byte) xor in_ROUND_KEY (8+byte);
        out_W3(byte) <= out_W2(byte) xor in_ROUND_KEY (4+byte);
        out_W4(byte) <= out_W3(byte) xor in_ROUND_KEY (byte);
  end generate;

  out_ROUND_KEY <= (out_W1(3), out_W1(2), out_W1(1), out_W1(0), out_W2(3), out_W2(2),
out_W2(1), out_W2(0), out_W3(3), out_W3(2), out_W3(1), out_W3(0), out_W4(3), out_W4(2),
out_W4(1), out_W4(0));
```

Arquitectura de la addRoundKey.

La implementació d'aquesta operació només consta d'una entitat, *addRoundKey*, i la seva arquitectura és una sentència *generate* que suma XOR byte a byte la *State Matrix* i la *roundKey*.

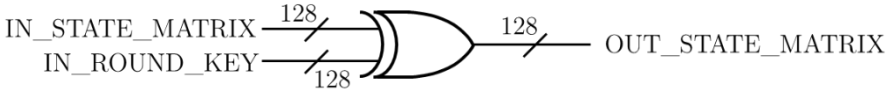


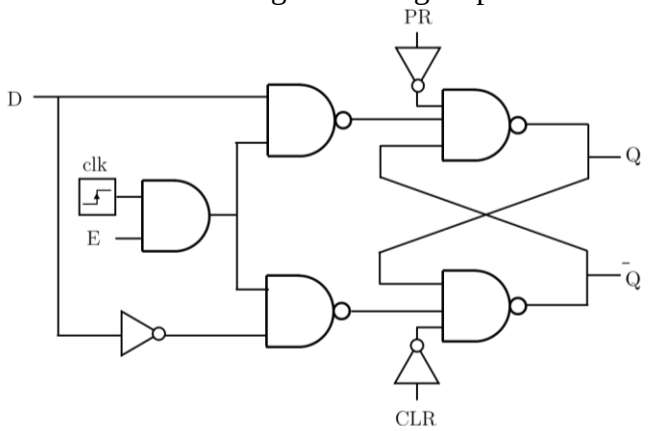
Figura 30. *Esquemàtic de la addRoundKey que implementa la suma XOR.*

El codi simple que ho descriu es pot consultar a l'apartat pertinent dels *Annexos*, A.2.11.

3.2.2. Registre paral·lel basat en biestable tipus D.

Tenint en compte que, si bé les diferents operacions de l'AES han estat implementades amb lògica combinacional, la idea pel circuit d'alt nivell és muntar un sistema seqüencial síncron. Per a tal propòsit és necessària la creació d'un registre paral·lel que emmagatzemi la informació de l'estat de cada ronda i que s'actualitzi a cada senyal de rellotge.

Un registre paral·lel és un conjunt de w–biestables que permet emmagatzemar paraules de w bits de llargària. N'hi ha de molts tipus, però donades les exigències de l'AES, el tipus que més s'adequa és el biestable tipus D síncron, també anomenat *D flip-flop*. Per a fer el disseny el més complet possible, s'ha decidit d'implementar-lo amb senyals de control E (*enable*), CLR (*clear*) i PR (*preset*) que sigui actiu per flanc ascendent de rellotge i amb lògica positiva. A la *Figura 31* hi ha el seu esquemàtic.



PR	CL	E	clk	D	Q
1	0	X	X	X	1
0	1	X	X	X	0
0	0	0	X	X	Q
0	0	1	0	X	Q
0	0	1	1	1	1
0	0	1	1	0	0
1	1	*ESTAT PROHIBIT*			

Figura 31. *Esquemàtic d'un biestable tipus D síncron amb senyals de control E, PR i CLR i la seva taula de la veritat.*

Com a consideracions, la taula de la veritat marca amb una *X* quan el valor dels senyals són indiferents i amb una *Q* quan hi ha manteniment de senyal. Cal destacar, també, el següent:

- Els senyals PR i CLR posen, independentment del rellotge, el biestable a 1 o a 0.
- El senyal E desactivat implica que el senyal de sortida Q es mantindrà.
- La combinació PR i CLR a nivell alt està prohibida.

En lloc d'implementar 128 biestables, el que s'ha fet en codi VHDL ha estat descriure el comportament d'un registre paral·lel de 128 bits. S'ha anomenat a l'entitat *reg_ff*, i l'arquitectura dissenyada es mostra a continuació:

```
process(clk)
  begin
    if rising_edge(clk) then
          if CLR = '1' then
            Qbuff <= (x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00",
                      x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00");
          else
            if E='1' then
              Qbuff <= D;
                end if;
          end if;
        end if;
  end process;
  Q <= Qbuff;
```

Atès que no es necessita el PR, només s'ha considerat el CLR. El codi complet es troba al A.2.5.

3.3. Arquitectura d'alt nivell.

Un cop vist el disseny hardware de les operacions pròpies de l'AES i els registres paral·lels de 128 bits, en aquest apartat es presentarà com s'ha fet el disseny de l'arquitectura d'alt nivell. L'entitat principal ha rebut el nom d'*AES,* tot i que per distingir-la del nom de l'algorisme, s'anomenarà *topAES* al llarg del document ("*top*" perquè és l'entitat de major nivell).

Abans d'entrar en matèria, cal parar atenció a l'organització general del *topAES* mostrada a la *Figura 32.* En ella es veu l'estructura dels macro-blocs que l'implementen, així com els diversos senyals principals que emanen d'ells i amb què es comuniquen entre si.

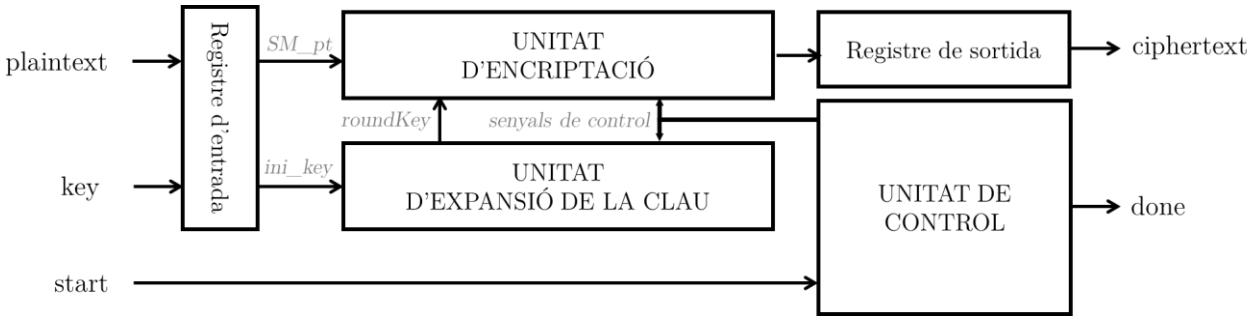


Figura 32. *Simplificació esquemàtica del nivell més alt de la implementació de l'AES.*

Els senyals d'input (entrada) i output (sortida) de la implementació són:

- *Plaintext* (input): senyal de 128 bits que conté el missatge secret a xifrar per l'algorisme.
- *Key* (input): senyal de 128 bits que conté la clau secreta per a fer el xifratge.
- *Start* (input): senyal d'1 bit que es posa a nivell alt quan es vol començar a fer el xifratge i que es posa a nivell baix quan es vol reiniciar el sistema.

- *Ciphertext* (output): senyal de 128 bits que conté el missatge secret xifrat.
- *Done* (output): senyal d'1 bit que està a nivell alt quan s'ha finalitzat l'encriptació i que està a nivell baix mentre dura el procés de xifrat.

- *CLK*: és el senyal de rellotge, *unívoc* en tot el sistema, que incideix en tots els macro-blocs del sistema. No es mostra a la *Figura 32*, però se sobreentén la seva existència.

Les diferents unitats que formen l'arquitectura de més alt nivell s'han definit de la següent manera:

- ***Unitat d'encriptació***: és el mòdul on es calculen les 10 rondes de l'AES-128, tal que a cada cop de rellotge es calcula una ronda.
- ***Unitat d'expansió de la clau***: aquí es calculen les diferents *roundKey* que s'empraran i s'administren, a cada cop de rellotge, a la unitat d'encriptació.
- ***Registres d'entrada i sortida***: emmagatzemen els senyals d'entrada i sortida per tal que es mantinguin quan convingui.
- ***Unitat de control***: d'aquesta unitat sorgeixen, en funció de l'estat del sistema, els senyals de control que reben les unitats d'encriptació i expansió amb l'objectiu de controlar-los.

Els senyals destacats de color més dèbil són els senyals interns principals del sistema.

3.2.1. Unitat de control.

Per a poder controlar i automatitzar el disseny hardware és necessari la implementació d'una unitat de control que enviï els senyals necessaris a la resta d'unitats, tant per a assegurar el bon funcionament d'aquestes com per a donar-los ordres. En particular, l'arquitectura que s'ha plantejat per al control és la mostrada a la *Figura 33*.

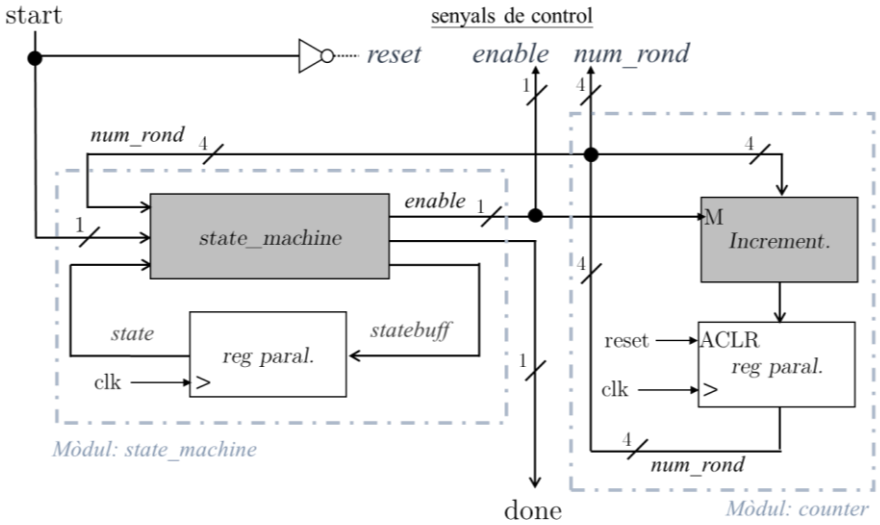


***Figura 33**. Esquemàtic de la unitat de control de la implementació.*

En l'esquemàtic es destaquen dos mòduls diferents en requadres puntejats, *state_machine* i *counter*, i els tres *senyals de control* del sistema que s'especifiquen a continuació:

- *Reset* (1 bit): es calcula com la negació de l'input principal *start*. Serveix com a senyal de posada a 0 dels diferents registres paral·lels del *topAES*.
- *Enable* (1 bit): es calcula en el mòdul *state_machine* i és el senyal d'habilitació de la majoria dels registres paral·lels de les unitats del *topAES*.
- *Num_rond* (4 bits): es calcula en el mòdul *counter* i és el que indica el nombre de la ronda actual, on $\emptyset b0000$ és l'instant inicial i $\emptyset b1010$ és la darrera ronda.

Seguidament es tractaran els dos mòduls principals i s'exposarà de quina manera es calculen els senyals de control ja esmentats.

Màquina d'estats finits (state_machine).

En electrònica, una màquina d'estats finits o autòmat síncron és un model computacional que permet permutar entre un nombre finit d'estats amb què es pot descriure el comportament d'un sistema determinat. S'estableixen uns criteris de transició concrets que defineixen, en funció d'unes entrades determinades, els criteris amb què es salta d'estat a estat.

En el context d'aquest treball s'ha decidit implementar-ne una amb l'objectiu de poder activar i desactivar, segons convingui, dos senyals concrets: el senyal *enable* i l'output principal *done*. Això es pot fer d'aquesta manera perquè ambdós senyals depenen intrínsecament de l'estat del sistema: *enable* ha d'estar actiu mentre s'encripti i *done* només quan s'hagi acabat la iteració, però no abans.

Per tant, com que les dues sortides de l'autòmat depenen només de l'estat, el disseny s'ha basat en els autòmats de Moore[17].

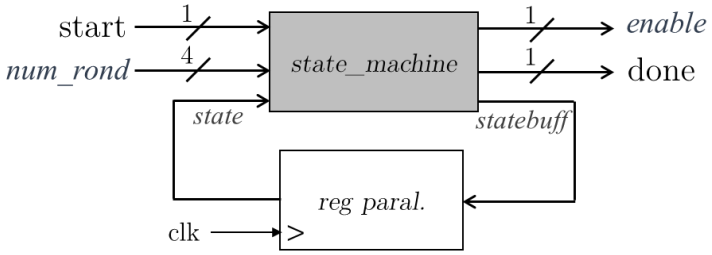


Figura 34. *Representació de la màquina d'estats finits com un sistema seqüencial síncron.*

Com es veu a la *Figura 34*, els senyals d'entrada i sortida de l'autòmat dissenyat són el *start* i *num_rond* com a inputs, i l'*enable* i el *done* com a outputs. Els senyals s*tate* i *statebuff* contenen la codificació que representa el estat actual del sistema. En concret, els possibles estats són:

- *Estat* IDLE ("*idle*"): és aquell en què l'AES encara no ha rebut cap ordre de començar l'encriptació, i és considerat com l'estat inicial. Per tant, tots els registres paral·lels estan inhabilitats i posats a 0, ja que així no es produeix l'encriptació.
  → *enable = 0* (registres inhabilitats) i *done = 0* (no ha ni començat l'encriptació)

- *Estat* ENCR ("*encrypting*"): implica que les unitats d'encriptació i d'expansió de la clau estan en funcionament ja que s'estan duent a terme les rondes de l'algorisme i els registres paral·lels estan actius.
  → *enable = 1* (registres habilitats) i *done = 0* (encara no ha acabat l'encriptació)

- *Estat* DELIV ("*delivering*"): aquest estat implica que les 10 rondes de l'AES s'han completat i que l'output *ciphertext* està donant senyal. Cal indicar-ho a l'usuari amb el senyal *done*.
  → *enable = 0* (registres inhabilitats) i *done = 1* (ha acabat l'encriptació)

Les transicions entre estats en funció de les entrades i de les sortides es troben a la *Figura 35* en forma de graf d'estats, on les entrades es mostren en les arestes i les sortides, en els vèrtexs.

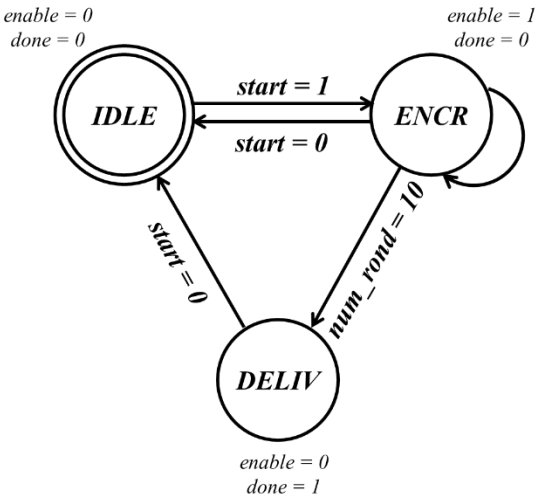


Figura 35. *Graf d'estats de l'autòmat dissenyat per al topAES.*

[17] Edward F. Moore (23 de novembre del 1925 – 12 de juny del 2003) fou un professor americà de matemàtiques i ciències de la computació a qui s'atribueix l'invent de la Màquina o Autòmat de Moore.

Traduint el graf, una encriptació qualsevol començaria per inicialitzar el sistema (*start = 0*) per tal d'estar en l'estat IDLE, introduir els *plaintext* i *key*, i indicar que l'encriptació pot començar (*start = 1*). Aleshores s'entraria en estat ENCR i l'autòmat s'hi mantindria fins arribar a darrera ronda (*num_rond = 1010*), o en cas que l'usuari poses el senyal *start* a nivell baix[18]. Finalment, s'entraria en estat DELIV i el senyal *done* i *ciphertext* donarien senyal. Per reiniciar, caldria posar *start* a 0.

La implementació VHDL d'aquest autòmat s'ha fet amb enfoc funcional. Es pot consultar a l'apartat A.2.3 dels *Annexos*, i es veurà que el codi és, bàsicament, la definició del graf d'estats. El que sí que cal comentar és que els senyals *state* i *statebuff* són del *datatype* s*tate_name* vist en el 3.2.3.

Registre comptador de les rondes (counter).

Aquest mòdul realitza el càlcul del senyal de control *num_rond*; el nombre de ronda. La manera que s'ha trobat més adient d'implementar-ho és amb un registre comptador de 4 bits, un sistema seqüencial síncron que recorre $2^4 = 16$ estats la sortida dels quals és el nombre de l'estat en binari (és a dir, si s'està a l'estat 4, la sortida serà 0100). La *Figura 36* en mostra l'esquema.

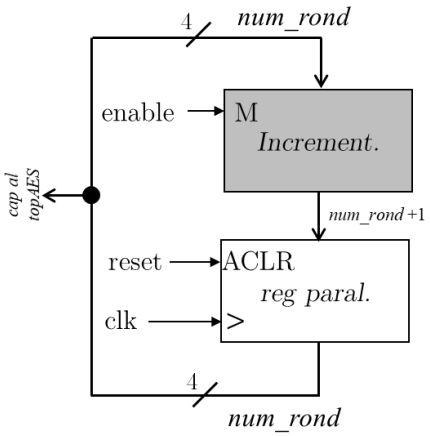


Figura 36. *Esquemàtic funcional del registre comptador del nombre de ronda actual.*

Bàsicament, el que fa aquest mòdul és incrementar en 1 bit el senyal d'entrada *num_rond* per a calcular el nou valor de *num_rond+1*, en bucle. Aquest darrer senyal és emmagatzemat en un registre paral·lel de 4 bits amb senyal de posada a zero ACLR (quan està actiu en lògica positiva, el nou nombre de ronda és sempre el 0). Addicionalment, es posseeix un senyal M (*mode*) que permet habilitar o inhabilitar el comptador, connectat al senyal de control *enable*.

Si bé permet contar fins al nombre 16 (4 bits), el sistema només requereix que arribi fins a 10. Això el comptador no ho pot limitar; l'autòmat d'estats finits ja s'encarrega de "congelar" el comptador quan s'arriba a l'estat DELIV desactivant l'*enable*, i l'usuari el reinicia quan posa *start* a nivell baix per a inicialitzar una nova encriptació (el *reset* està connectat a ACLR).

A la pràctica, el que s'ha descrit en VHDL és molt similar a la implementació de l'entitat *reg_ff*, amb la diferència que el senyal CLR aquí s'anomena ACLR i que, en lloc d'enregistrar el valor d'entrada, s'enregistra el valor d'entrada incrementat en 1 bit. El codi es pot consultar en l'apartat A.2.4.

[18] Aquesta casuística és difícil que es doni a la realitat, ja que el temps de computació, com es veu a l'apartat de resultats, és de l'ordre del nanosegon. Tanmateix, sempre és important deixar un "*keyboard interrupt*" per seguretat.

3.2.2. Unitat d'encriptació.

Per a arribar al *ciphertext* cal sotmetre la *State Matrix* a les 10 rondes de l'AES, una darrera de l'altre. La unitat d'encriptació és la que s'encarrega de realitzar aquests càlculs, dissenyada amb un enfoc seqüencial síncron (conforme el dit a l'apartat 3.2.2) amb lògica positiva. És a dir, cadascuna de les rondes es computa, una per una, a cada flanc ascendent del rellotge intern i unívoc del sistema.

Tanmateix, a l'hora de dissenyar l'arquitectura, cal tenir en compte que no totes les rondes de l'AES són iguals. En concret, podem distingir-ne tres:

- ***Ronda inicial***: El primer que es fa en l'encriptació és aplicar la *addRoundKey* al *plaintext* i la clau d'entrada. Si bé no és estrictament una ronda, s'ha decidit que es comptabilitzarà com a tal. Se li assigna el nombre 0.
- ***Rondes normals***: Es caracteritzen per passar per totes les operacions pròpies de l'AES sense excepció. Són les rondes de la 1 a la 9.
- ***Ronda final***: L'especificació de l'AES diu que la *mixColumns* no s'ha de computar per a la ronda final. És a dir, això només passa a la número 10.

Hi ha diverses maneres per a distingir entre aquestes tres possibilitats. En aquest treball, s'ha optat per l'arquitectura mostrada a la *Figura 37*.

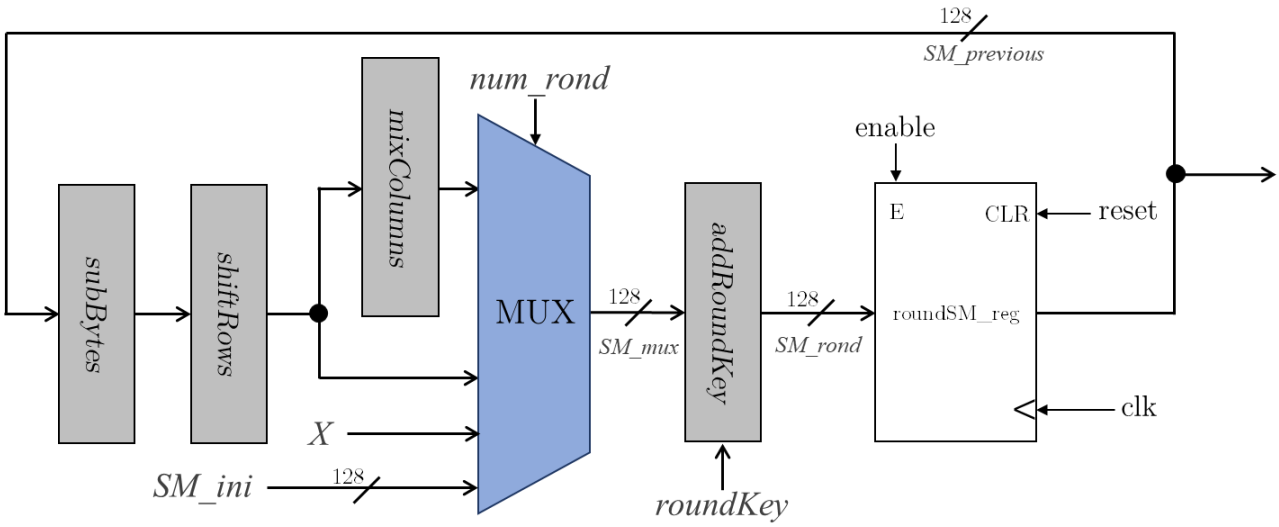


***Figura 37**. Esquemàtic de la arquitectura de la unitat d'encriptació representada com un sistema seqüencial síncron.*

És a dir, donat l'ordre d'aplicació de les operacions que s'ha estimat oportú (*subBytes*, *shiftRows*, *mixColumns* i *addRoundKey*), s'ha col·locat un multiplexor just abans de la *addRoundKey* per tal d'escollir entre les quatre línies possibles:

- La línia del senyal *SM_ini*, que és la sortida del registre inicial del *plaintext*; la ronda inicial.
- La línia que aplica l'operació *mixColumns* a la sortida del *shiftRows*; és a dir, la pertinent a les rondes normals de l'AES.
- La línia de sortida de la *shiftRows*, pròpia de la darrera ronda de l'algorisme.
- Una línia de senyal indiferent (X), ja que estrictament les línies d'un multiplexor són sempre potències de dos.

Mitjançant el senyal de control *num_rond* com a senyal de selecció, el multiplexor és capaç de disgregar entre les tres línies i aplicar amb èxit el procés estàndard d'encriptació.

El registre paral·lel *roundSM_reg* mostrat a la *Figura 37* és l'encarregat d'emmagatzemar els 128 bits de la *State Matrix* a cada ronda. A més a més, atès que està excitat pel flanc ascendent de rellotge, és el responsable que tota la unitat d'encriptació s'executi de forma síncrona cada ronda de l'AES. La seva posició en el circuit no és a l'atzar, ja que s'ha decidit que les rondes finalitzin amb l'*addRoundKey*; és a dir, que el senyal de sortida d'aquesta operació (*SM_rond*) és la matriu d'estat de la ronda que el registre ha d'emmagatzemar.

Els senyals de control del registre són *reset* (connectat a CLR) i *enable* (connectat a E). El primer d'ells permet posar a 0 la matriu d'estat de la ronda quan s'està a l'estat IDLE així com realitzar la inicialització de valors quan s'engega la font d'alimentació per primera vegada. Altrament, el segon senyal permet mantenir desactivat el procés d'encriptació fins que no es passi a l'estat ENCR i fer manteniment de senyal de l'última *SM_previous* (la seva sortida) un cop s'arribi a l'estat DELIV.

La descripció en VHDL no ha requerit d'una entitat per a la unitat d'encriptació, ja que s'ha implementat directament en el *topAES*. La rutina de la unitat s'ha instanciat com segueix:

```
--UNITAT D'ENCRIPTACIÓ
sB: entity work.subBytes port map(SM_previous, SM_subBytes);
sR: entity work.shiftRows port map(SM_subBytes, SM_shiftRows);
mC: entity work.mixColumns port map(SM_shiftRows, SM_mixColumns);
with num_rond select SM_mux <=
  SM_pt when "0000",
  SM_shiftRows when "1010",
  SM_mixColumns when others;
aK: entity work.addRoundKey port map(roundKey, SM_mux, SM_rond);
roundSM_reg: reg_ff port map(enable, RST, clk, SM_rond, SM_previous);
```

Per a poder enllaçar-les, s'ha creat un senyal intern per cada output d'entitat de baix nivell amb nom *SM_"nomentitat"*, a excepció dels tres senyals principals:

- *SM_rond*: és el que conté el valor de la matriu d'estat de la ronda actual i que emana de l'entitat *addRoundKey*. Es connecta al registre *roundSM_reg*.
- *SM_previous*: és la que conté la matriu de la ronda anterior. Això succeeix perquè en un *flip-flop* D la sortida va endarrerida un cop de rellotge.
- *SM_mux*: és el senyal que surt del multiplexor, el qual s'encamina segons el nombre de ronda.

3.2.3. Unitat d'expansió de la clau.

Com en el cas de la d'encriptació, la unitat d'expansió de la clau ha estat dissenyada com un sistema seqüencial síncron. L'única operació pròpia de l'AES que cal aplicar és la *expandKey*.

Respecte a l'arquitectura d'aquest mòdul, també és possible distingir entre més d'un tipus de ronda, tot i que de manera molt més simple que en la unitat anterior:

- ***Ronda normal***: la *roundKey* ha de ser enviada a la unitat d'encriptació per a fer-se servir en la *addRoundKey*. Són les rondes de la 1 a la 10.
- ***Ronda inicial***: la clau d'entrada, sense haver passat per *expandKey*, ha de ser enviada a la unitat d'encriptació. És la ronda número 0.

La manera de destriar de forma correcta entre aquests dos casos no és evident. Cal tenir present que, si bé no s'executa la *expandKey*, la ronda inicial *és* una ronda, i per tant el registre paral·lel de la unitat l'ha d'emmagatzemar al primer cop de rellotge. Emprant també un multiplexor i col·locant-lo entre el mòdul *expandKey* i el registre, es podrà escollir el tipus de ronda amb el senyal de control *num_rond* com a senyal de selecció.

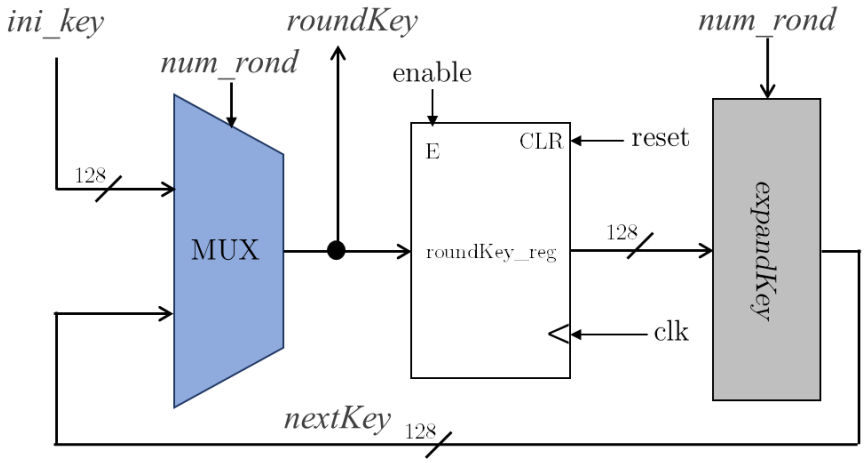


Figura 38. *Esquemàtic de la arquitectura de la unitat d'expansió de la clau com a sistema seqüencial síncron*

L'arquitectura de la unitat es pot veure a la *Figura 38*. Cal notar que la línia que es redirigeix a la unitat d'encriptació és la del senyal de sortida del multiplexor i no la del *roundKey_reg*. La raó d'això és que, si es prengués la de després del registre, s'estaria enviant la *roundKey* de la ronda anterior (ja que tot registre s'actualitza un cop de rellotge en retard). A més a més, és important destacar que l'operació *expandKey* requereix del senyal *num_rond* com a entrada per a seleccionar la *Rcon*[19].

La implementació del *roundKey_reg* segueix la mateixa lògica que la del *roundSM_reg* de la unitat d'encriptació (senyals E i CLR connectats a *enable* i *reset* respectivament, pels mateixos motius).

En VHDL tampoc s'ha creat una entitat, ja que s'ha escrit directament al *topAES*. La rutina s'ha instanciat com segueix:

```
--UNITAT D'EXPANSIÓ DE LA CLAU
with num_rond select roundKey <=
        iniKey when "0000",
        nextKey when others;
roundKey_reg: reg_ff port map(enable, reset, clk, roundKey, previousKey);
eK: entity work.expandKey port map(previousKey, nextKey, num_rond);
```

Els senyals d'interconnexió són només tres:

- *roundKey*: és la clau de la ronda que s'escull entre *nextKey* i la clau inicial *iniKey* que prové del registre del senyal *key*.
- *previousKey*: és la sortida del registre i, com a la unitat d'encriptació, es tracta de la clau de la ronda anterior.
- *nextKey*: és la clau calculada en expandKey que representa la pròxima *roundKey*.

[19] Veure la pàg. 27, on es parla de l'arquitectura de la *expandKey* i el motiu pel qual es necessita *num_rond* com a input.

3.4. TopAES complet.

L'esquema de l'arquitectura d'alt nivell de tot l'AES es pot consultar a la *Figura 39* a la següent pàgina. En ella es poden veure les diferents unitats que s'han exposat fins ara, els senyals que emanen d'elles i com es relacionen entre si. És, a la fi, la implementació hardware de l'AES pròpiament dita que aquest treball s'havia posat com a objectiu de dissenyar. El codi general de l'entitat *topAES* es troba al seu apartat pertinent en els *Annexos*, és a dir, el A.2.1.

Per a entendre'l de millor manera, cal comentar uns últims detalls que encara no s'han mencionat.

Registres d'entrada i de sortida.

Són els *plaintext_reg*, el *inputKey_reg* (d'entrada) i el *final_reg* (de sortida). En tots els casos han estat implementats com instàncies de l'entitat *reg_ff* en la descripció VHDL i el seu senyal de CLR ha estat connectat al senyal de control *reset*. Tanmateix, la connexió del seu senyal E ha estat diferent:

- En els registres d'entrada sempre estan connectats a nivell alt (1), ja que no és necessari en cap circumstància inhabilitar-los en ser tan sols d'importància a la primera ronda.
- En el registre de sortida s'ha connectat amb el senyal *done*, ja que tan sols interessa que doni senyal diferent de 0...0 un cop hagi acabat l'encriptació.

La manera amb què s'instancien en l'entitat *topAES* és la següent:

```
--REGISTRES INICIALS (Plaintext i Key)
inputKey_reg: reg_ff port map('1', reset, clk, key, iniKey);
plaintext_reg: reg_ff port map('1', reset, clk, plaintext, SM_pt);

--REGISTRE FINAL (Ciphertext)
final_reg: reg_ff port map(done_id, reset, clk, SM_previous, cyphertext);
```

Instàncies de la unitat de control.

Els dos mòduls de la unitat de control s'han instanciat en l'entitat *topAES* per separat tal com es mostra a continuació:

```
--COMPTADOR (registre comptador de 4 bit)
round_counter: entity work.counter port map(enable, reset, clk, num_rond);

--STATE_MACHINE (FSM)
FSM: entity work.state_machine port map(start, clk, num_rond, enable, done_id);
```

Val a destacar que el senyal de sortida de la *state_machine* es diu *done_id*. El motiu d'això és que, atès que el senyal *done* és un output general de l'entitat *topAES*, no es pot usar en la instància *port map*. Per tant, mitjançant el senyal intern *done_id* i assignant-lo degudament (`done <= done_id;`) és possible arreglar aquest problema.

Senyal de control reset.

Cal dir, a mode de recordatori, que el senyal *reset* prové de la negació del senyal *start*. En el codi s'ha implementat d'una manera tan simple com segueix:

```
reset <= not start;
```

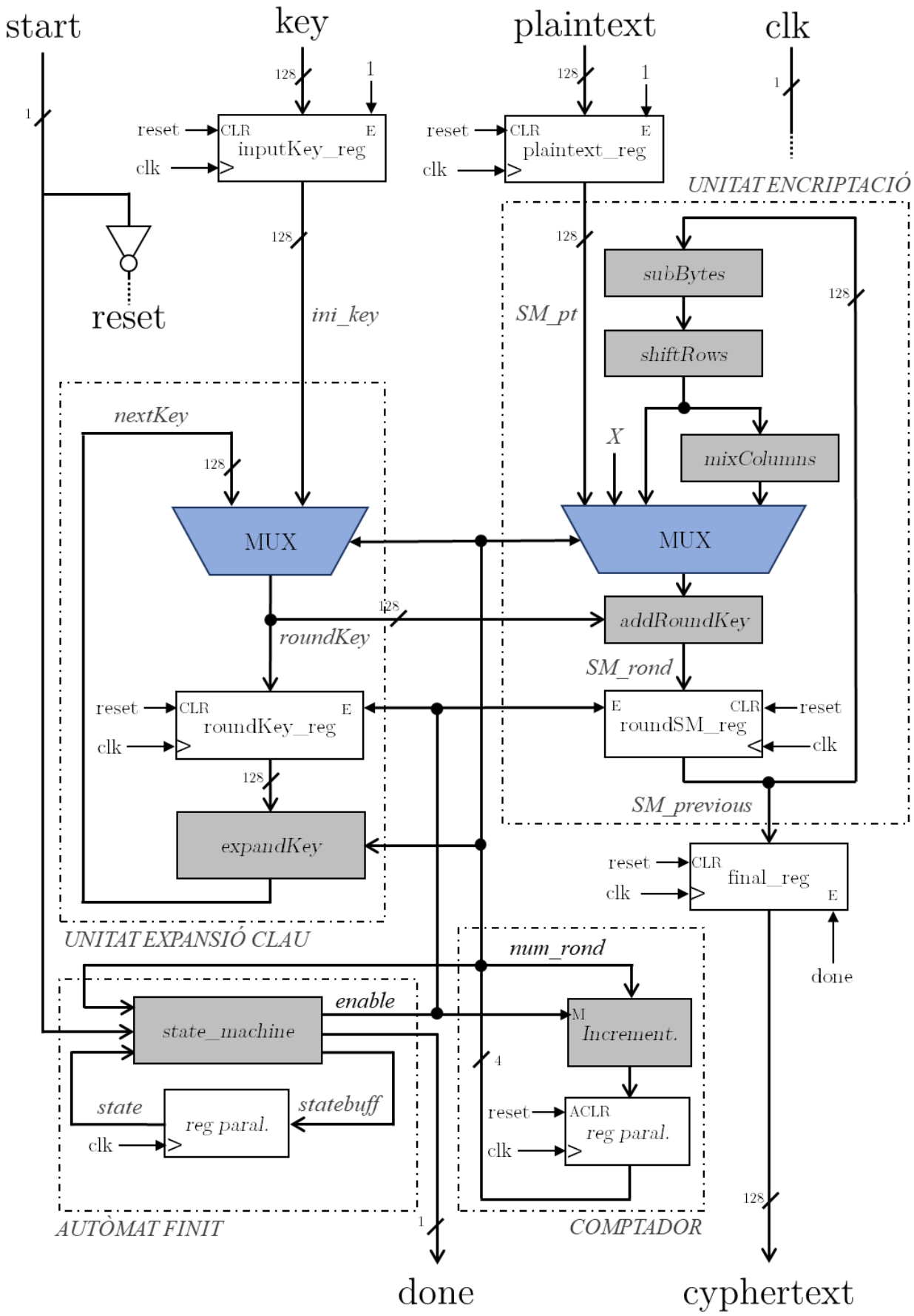


Figura 39. *Esquemàtic d'alt nivell complet de la implementació hardware de l'AES.*

CAPÍTOL 4. Resultats de la implementació hardware de l'AES.

En aquest capítol s'executarà una avaluació profunda de la implementació en hardware mitjançant el codi descriptiu VHDL, el disseny de la qual s'ha parlat en el Capítol 3, amb l'objectiu de determinar tant si funciona correctament com si la seva síntesi es correspon amb l'arquitectura plantejada. Addicionalment, es practicarà un atac SCA per a comprovar-ne la robustesa.

4.1. Aspectes tècnics: hardware i software emprats.

El disseny hardware mitjançant VHDL vist al Capítol 3 ha estat ideat per a ser implementat en una FPGA (*Field-Programmable Gate Array*). En concret, tot el procés de simulació i síntesi s'ha fet damunt la Intel / Altera Cyclone IV EP4CE30F29C6. Consultant el *Handbook* [31] d'aquesta família, és possible obtenir les especificacions d'interès per a la realització d'aquest treball, que es poden consultar resumides en la *Taula 3*.

Elements lògics	***Bits de RAM***	***Multiplic. 18 x 18***	***Xarxes CLK globals***	***I/O d'usuari màxims***	***Nombre de blocs M9K (memòries)***	***Llaços de seguiment de fase***
28.848	608.256	66	20	532	56	4
Qualificació de velocitat.			C6.			

Taula 3*. Especificació de la Altera Cyclone IV EP4CE30F29C6 d'interès per al treball.*

Per a realitzar les tasques de simulació s'ha emprat el *ModelSim – Intel FPGA Edition* desenvolupat per *Mentor Graphics*. Aquest software és un entorn que permet el desenvolupament i verificació de llenguatges descriptius de hardware, com el VHDL, donat que permet compilar fitxers escrits en aquest llenguatge, detectar-hi errors i realitzar simulacions. Totes les funcions anteriors han estat aprofitades, però cal destacar que la interfície *wave* serà la que més es veurà en aquestes pàgines, ja que és aquella que permet visualitzar gràficament els diferents senyals d'interès del sistema durant la simulació.

Altrament, la síntesi del circuit ha estat processada emprant el *Intel Quartus Prime Lite Edition*, ara pertanyent a *Intel* tot i que nasqué a mans d'*Altera*. És un entorn de disseny de PLD i CPLD que analitza i sintetitza models fets amb llenguatges descriptius de hardware, del qual la versió *Lite* permet emprar tota la família de FPGA Cyclone.

Figura 40. *Logotips dels dos programaris utilitzats.*

4.2. Resultats de la simulació.

Seguidament es mostraran els resultats obtinguts de dues simulacions fetes mitjançant el *ModelSim* de la implementació hardware de l'AES. És important i rellevant notar que, durant el procés de disseny de les diferents entitats que el conformen, s'ha anat testejant una a una per simulació mitjançant els seus propis *testbench*. Tanmateix, en aquestes pàgines tan sols es tractaran els resultats del *topAES* per simplicitat i perquè són suficients per a demostrar que tot el seu conjunt es comporta degudament.

Per a les dues simulacions, s'han fet servir plaintext diferents, però la mateixa clau.

4.2.1. Fitxer *testbench AES_TB*.

Com bé s'ha explicat en l'apartat 1.3.3, per a poder realitzar la simulació és necessari redactar un mòdul *testbench*. Per a la simulació del *topAES* s'ha fet servir el mostrat a l'apartat A.2.2 dels *Annexos*, del qual es parlarà seguidament.

En la part declarativa de l'arquitectura cal instanciar el *component* corresponent al *topAES* i declarar com a senyals interns els valors d'inicialització dels seus inputs i outputs (*plaintext, key, start, clk, ciphertext* i *done*). Seguidament, en la part de processament i sentències, després de fer el *port map* que uneix els senyals interns declarats amb els pins del *topAES*, cal obrir els processos que es duran a terme en la simulació. En aquest cas, se n'ha escrit dos.

Procés d'inici d'encriptació.

Aquest és el procés que duria a terme l'usuari de la implementació física. És a dir, és el procediment que incorpora la posada a 0 de tot el sistema, l'entrada del *plaintext* i la *key* de la ronda i la inicialització de l'encriptació. Es mostra tot seguit:

```
process
  begin
       start<='0';
       wait for 20 ns;
       plaintext <= (      ); -- introduir el plaintext
       key       <= (x"54", x"68", ... );
       wait for 20 ns;
       start <= '1';
       wait;
end process;
```

En definitiva, el que s'estaria fent és inicialitzar el sistema posant *start* a 0 (recordar que el senyal de control *reset* n'és el negat), esperar *20ns* per tal de deixar passar un cicle de rellotge i que el sistema s'actualitzi, introduir el *plaintext* (se n'ha fet servir dos) i la *key* (només se n'ha usat una), esperat 2*0ns* per tal de deixar un altre cicle de rellotge (s'assegura que els registres inicials ja donen senyal de sortida), i finalment inicialitzar l'encriptació posant *start* a 1.

Cal notar que la sentència *wait*, en aquest cas, finalitzaria amb aquest procés fins que el propi del senyal de rellotge s'acabés.

Procés del senyal de rellotge (clk).

S'ha definit el comportament del senyal *clk* com una ona quadrada de període *10ns* com segueix:

```
process
begin
  clk<='0';
  wait for 10 ns;
  clk<='1';
  wait for 10 ns;
       if done='1' then
          clk<='0';
          wait for 10 ns;
          clk<='1';
          wait for 10 ns;
          clk<='0';
          wait for 10 ns;
          clk<='1';
          wait for 10 ns;
          wait;
       end if;
end process;
```

Si només es deixessin les quatre primeres sentències dins del *process*, el senyal de rellotge es mantindria de forma perpètua (ja que no hi ha sentència *wait*, la qual el mataria a la primera oscil·lació). Per tant, s'ha decidit d'incloure una sentència *if* amb condició que el senyal de sortida *done* estigui a nivell alt; és a dir, que hagi acabat l'encriptació. Aleshores, es donen dues oscil·lacions més de marge al sistema i, finalment, es finalitza amb la sentència.

Val a dir que el període del senyal de rellotge s'ha agafat per a fer més simple i estètica la simulació mostrada, ja que en cas experimental caldria assegurar un període superior al retard de propagació.

4.2.2. Primera simulació.

Seguidament s'exposarà amb detall la primera de les simulacions. Es tractaran el comportament dels diferents senyals input/output principals així com els senyals interns del *topAES* de rellevància amb l'objectiu d'interpretar amb exactitud el funcionament de la implementació.

A la *Taula 4* es poden veure els senyals *plaintext* i *key* escollits per a aquesta simulació; *One Ring To Rule* i *Thats my Kung Fu*, respectivament. Ambdós missatges s'han escrit en codi hexadecimal entenent que estaven expressats en codi ASCII.

	Codi ASCII.	***Hexadecimal***
Plaintext	One Ring To Rule	4f 6e 65 20 52 69 6e 67 20 54 6f 20 52 75 6c 65
Key	Thats my Kung Fu	54 68 61 74 73 20 6d 79 20 4b 75 6e 67 20 46 75
Ciphertext esperat		ac 79 6e 9b b7 59 e4 82 18 cb 2a a6 1e 72 6b 90

***Taula 4**. Valors del plaintext, key i ciphertext en ASCII i hexadecimal per a la primera simulació.*

Per a determinar el *ciphertext* correcte per a aquesta combinació de *plaintext* i *key* s'ha emprat la implementació en Python3 de l'AES que s'ha presentat al punt 2.3, la qual es pot consultar als A.1.2. No es mostra la seva versió en codi ASCII perquè es tracta d'un munt de caràcters estranys sense sentit que poca importància tenen per a la verificació del *topAES*.

Amb això en ment, s'ha executat la simulació mitjançant el programari *ModelSim* i s'ha obtingut el gràfic de forma d'ones mostrat a la *Figura 41*.

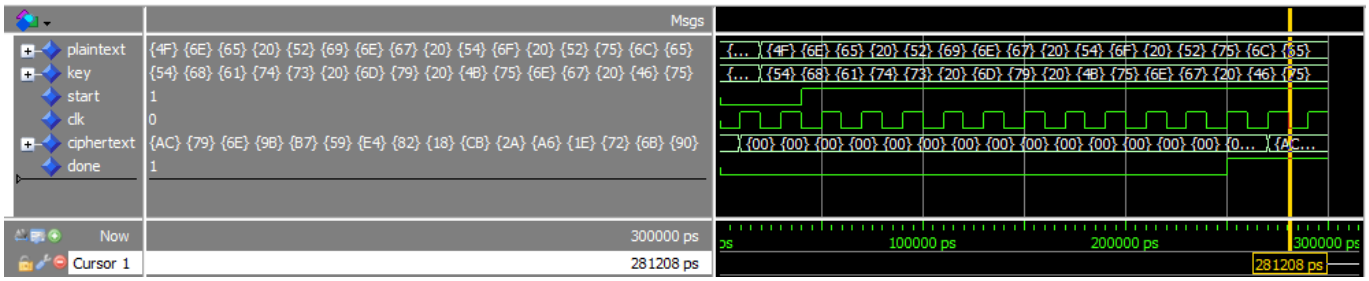


***Figura 41**. Senyals principals del sistema de la primera simulació destacant els valors finals (a l'esquerra).*

Es pot observar que el funcionament del sistema *topAES*, almenys vist com a caixa negra, és el correcte i esperable. Al principi de la simulació, fins al primer cop de rellotge, els senyals tenen els valors inicials del *testbench* (consultar A.2.2). El senyal *start* és igual a 0, de tal manera que al flanc ascendent el ciphertext dóna senyal nul ja que el seu registre paral·lel s'inicialitza, i aleshores s'entren els valors del *plaintext* i de la *key*. Seguidament, el senyal *start* pren valor 1 i, se suposa, comença l'encriptació durant els següents cicles fins que el senyal *done* es mostra igual a 1. Donat que els registres van un cicle endarrerits, al següent flanc ascendent el *ciphertext* dóna senyal. Es pot observar que, aquest, és l'esperat a la *Taula 4*.

En conclusió, aquesta simulació ha estat exitosa i ha demostrat un correcte funcionament del *topAES*.

Exploració detallada dels registres i dels senyals interns.

S'ha cregut convenient repetir la simulació, però explorant els següents senyals:

- *start*, *clk*, *done*: són els senyals principals del sistema ja coneguts.
- *enable*: senyal d'habilitació dels registres paral·lels (1 o 0).
- *state*: senyal que indica el nombre d'estat actual (IDLE, ENCR o DELIV).
- *num_rond*: nombre de la ronda actual de l'AES (0000 fins a 1011).
- *roundKey*, *SM_rond*: codificació de la clau i la *State Matrix*, respectivament, en hexadecimal que coincideix amb el nombre de la ronda i és l'entrada del registre paral·lel de la clau.
- *previousKey*, *SM_previous*: codificació de la clau i la *State Matrix*, respectivament, en hexadecimal que va una ronda endarrerida, ja que és la sortida del registre paral·lel de la clau.
- *ciphertext*: senyal de sortida amb l'output principal.

D'aquesta manera, es podrà obtenir una millor imatge del funcionament del *topAES* i de com els diferents registres paral·lels es coordinen per a arribar de forma síncrona al missatge xifrat. Els resultats de la simulació es poden veure a la *Figura 42*, la qual ocupa tota la següent pàgina per tal d'afavorir-ne la llegibilitat.

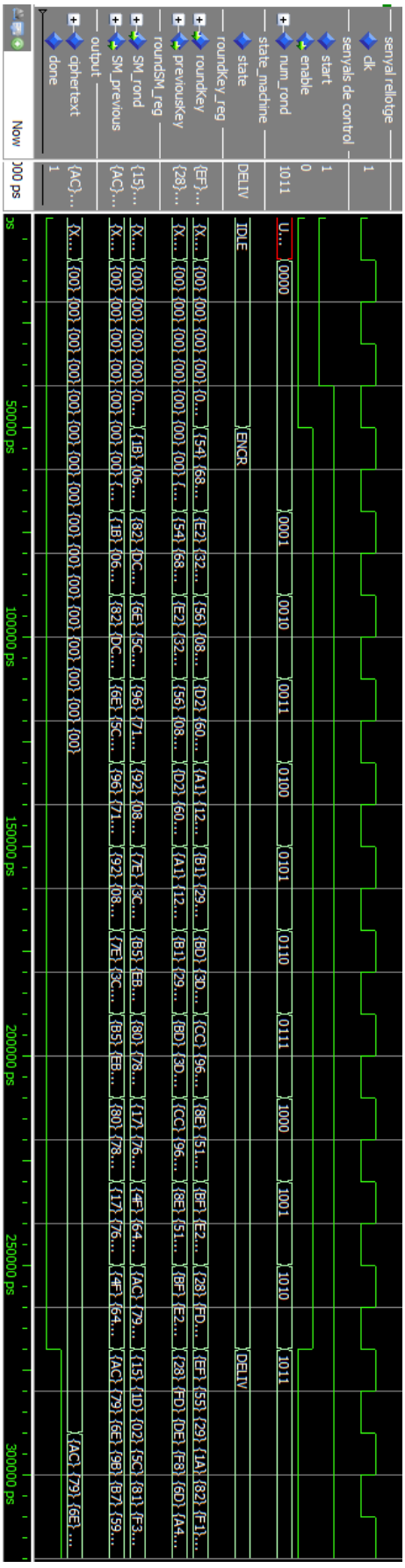


Figura 42. *Formes d'ona de la segona simulació amb els diferents senyals d'interés.*

Per a fer una anàlisi ordenada, s'aniran resseguint les formes d'ona des del temps d'inici fins al final de la simulació. Al començament, el senyal de *start* es força a nivell baix per l'usuari; per tant, els diferents registres s'inicialitzen al primer flanc ascendent de rellotge i es mantenen a 0...0 (abans d'això tenien un valor nul o *X – no importa*). Cal destacar que el sistema es troba en estat IDLE i que el nombre de ronda d'inici, després del primer flanc, és 0000.

Aleshores, un cop entrats el *plaintext* i la *key*, l'usuari posa el senyal *start* a nivell 1. Al següent cop ascendent de rellotge, el sistema s'actualitza i entra en estat ENCR alhora que el senyal *reset* (que no es mostra a la figura) passa a ser nul (negat de *start*) i s'activa l'*enable*. Com a conseqüència, el senyal *CLR* dels biestables es desactiven i els registres inicials deixen passar la informació del *plaintext* i la *key* a les seves respectives unitats de tractament (d'encriptació i d'expansió). Abans, els registres enviaven 0...0 ja que tenien el *CLR* activat.

Dins del mateix flanc de rellotge es produeix el càlcul de la ronda inicial de l'AES. Això es pot comprovar veient que *roundKey* i *SM_rond* tenen valor diferent de 0...0, ja que s'ha executat l'operació *addRoundKey* amb el *plaintext* i la *key*. Tanmateix, els senyals *previousKey* i *SM_previous* segueixen sent 0...0 ja que el senyal *enable* s'acaba d'activar i, donat que els registres tenen un retard d'un cicle de rellotge, encara no han emmagatzemat la informació de la ronda inicial.

Al següent flanc ascendent, el nombre de ronda canvia, per fi, a 0001. Es calcula així la nova ronda donat que els registres ja han emmagatzemat els valors de la seva entrada (els senyals *previousKey* i *SM_previous* ja no són 0...0). Ara, *roundKey* i *SM_rond* tenen els valors de la clau i la *State Matrix* de la ronda actual. Aquesta lògica es manté, sota l'estat ENCR, fins que s'arriba a la ronda 1010.

Quan s'hi arriba, el sistema es troba en la desena i última ronda de l'AES, tal que el senyal *SM_rond* conté la informació pròpia del *ciphertext*. No obstant, encara s'ha de transmetre fins al senyal de sortida (anomenat *ciphertext*). En la *Figura 39* del final del Capítol 3 es pot veure que el registre final (*final_reg*) que emmagatzema el senyal de sortida té com a entrada pròpia el *SM_previous*. Per tant, caldran dos flancs de rellotge addicionals per a transportar la informació fins al senyal *ciphertext*.

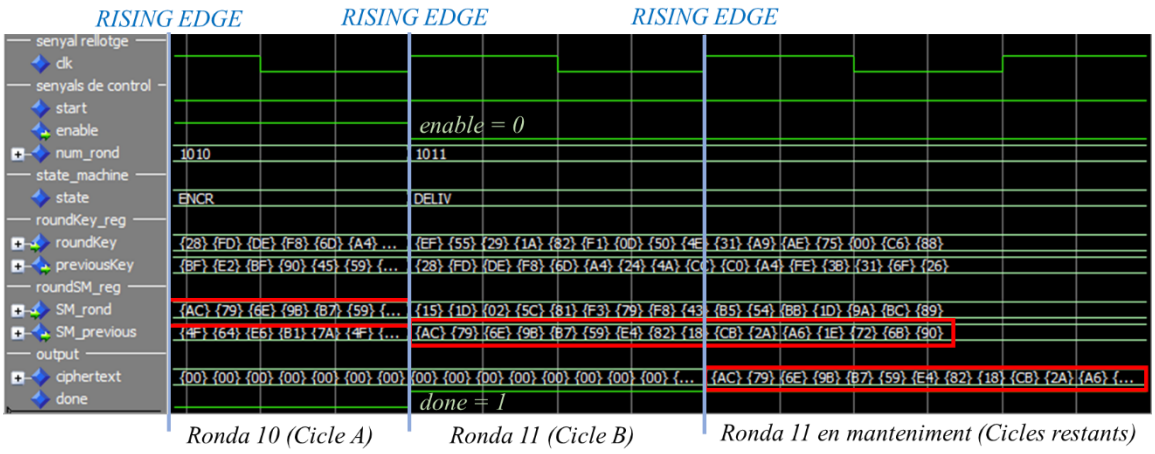


Figura 43. *Seguiment del senyal final del ciphertext durant les rondes finals 10 i 11.*

En la *Figura 43* es pot veure amb detall què ocorre en aquests dos flancs addicionals. L'estat del sistema canvia a DELIV just quan es detecta la ronda número 1010, el segon flanc de la figura, i aleshores es transmet la *State Matrix* que conté el *ciphertext* (encerclada en vermell a la figura) de

SM_rond a *SM_previous*. Per tant, durant aquest emmagatzematge, com que el senyal *SM_previous* connecta amb les operacions de l'AES, que són combinacionals, es calcula una onzena ronda extra de l'AES que s'emmagatzema a *SM_rond.* Aquest és el motiu pel qual el senyal *num_rond* és 1011. Finalment, en el tercer i últim flanc marcat a la figura, el senyal *enable* que havia estat posat a nivell baix en l'anterior cicle ara es fa efectiu, causant un manteniment de senyal en tots els registres fent que les unitats es "congelin" en la ronda 11 extra; i el *final_reg* emmagatzema la *State Matrix* corresponent al *ciphertext* i la dóna com a output ja que el senyal *done* s'ha fet efectiu.

En conclusió, el comportament intern ha estat verificat com el corresponent, esperat i congruent.

4.2.3. Segona simulació.

Per tal de demostrar l'efectivitat de l'AES, s'ha decidit fer una segona simulació canviant tan sols un caràcter del plaintext. Aleshores, en lloc d'emprar *One Ring To Rule* es prendrà *One King To Rule*. A la *Taula 5*, de forma simètrica a l'apartat 4.2.2 anterior, es pot veure els diferents valors emprats en la simulació. Cal recordar que el *ciphertext* esperat s'ha calculat amb la implementació en Python3.

	Codi ASCII.	***Hexadecimal***
Plaintext	One King To Rule	4f 6e 65 20 4b 69 6e 67 20 54 6f 20 52 75 6c 65
Key	Thats my Kung Fu	54 68 61 74 73 20 6d 79 20 4b 75 6e 67 20 46 75
Ciphertext esperat		e5 01 24 62 d7 2e 57 9f c3 0d 1b e7 9c c5 60 19
Ciphertext primera simulació.		ac 79 6e 9b b7 59 e4 82 18 cb 2a a6 1e 72 6b 90

Taula 5. *Valors del plaintext, key i ciphertext en ASCII i hexadecimal per a la segona simulació*

Val a dir que, per a poder comparar els resultats entre la primera i la segona simulació, s'ha decidit d'incorporar a la taula de damunt el *ciphertext* obtingut en la primera. Amb aquesta informació i dades de partida, es procedeix a fer la simulació.

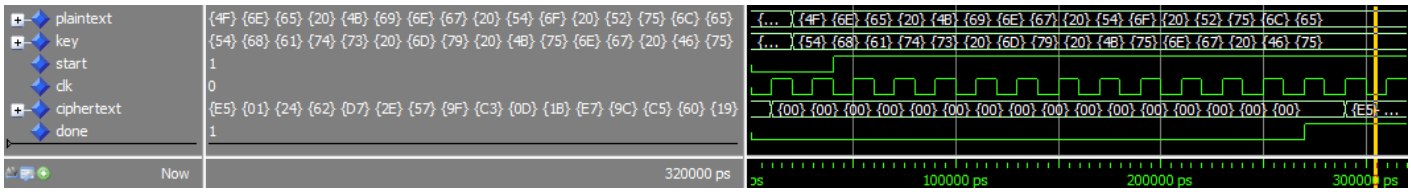


Figura 44. *Senyals principals del sistema de la primera simulació destacant els valors finals (a l'esquerra).*

Com es pot veure a la *Figura 44*, la simulació ha estat exitosa de nou donat que el *ciphertext* coincideix amb el valor esperat. A diferència de la primera, en aquest cas no es farà una inspecció profunda dels diferents senyals del sistema ja que amb una vegada és suficient per assegurar que el funcionament intern del *topAES* és l'adequat i el descrit a l'arquitectura. És interessant veure que, canviant tan sols un únic caràcter, el *ciphertext* no té res a veure. Això demostra l'efectivitat de l'algorisme AES a l'hora d'emmascarar el missatge original, ja implica que cadascun del bytes d'entrada afecten a tots els bytes de sortida (quelcom conegut pel nom d'*efecte allau*).

Havent fet dues simulacions en aquestes pàgines, i una quantitat addicional durant el transcurs del període d'elaboració del treball, es pot certificar que la implementació hardware de l'AES *topAES* dissenyada, almenys des d'un punt de vista pragmàtic i funcional, és correcte i vàlida.

4.3. Resultats de la síntesi.

La síntesi del *topAES* s'ha dut a terme mitjançant l'eina de síntesi *Intel Quartus Prime Lite Edition* damunt de la FPGA *Cyclone IV* comentades a l'apartat 4.1 de forma exitosa. El programa ha estat capaç d'executar la síntesi sense donar cap mena d'error i oferint el resum de la *Figura 45*.

Flow Status	Successful - Fri May 21 13:34:05 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	AES
Top-level Entity Name	AES
Family	Cyclone IV E
Total logic elements	5,367 / 28,848 (19 %)
Total registers	647
Total pins	387 / 533 (73 %)
Total virtual pins	0
Total memory bits	0 / 608,256 (0 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)
Device	EP4CE30F29C6
Timing Models	Final

Figura 45. *Report de la síntesi duta a terme amb l'Intel Quartus Prime.*

Cal destacar les dues dades més importants: el nombre total de blocs lògics i el nombre total de pins d'entrada i sortida que es requereixen.

El **nombre total de blocs lògics** ha estat de 5.367, un nombre força elevat, però que tan sols representa el 19% del total de blocs lògics disponibles de l'FPGA. Això implica que hi ha molta àrea de la placa que no es fa servir per a implementar el *topAES*, quelcom que té sentit donat que l'AES és un algorisme certament petit i que està pensat per a ser fàcil d'implementar. Tanmateix, un 19% d'ocupació és una dada "gran", en termes relatius. La raó d'això és que s'ha decidit d'implementar cada ronda de forma combinacional, quelcom que implica molta més ocupació d'àrea que si, per exemple, s'hagués decidit emprar més d'un senyal de rellotge i fer un *sistema seqüencial asíncron*.

Altrament, el **nombre de pins** és del 79%, una dada molt gran. Això respon al fet que el *plaintext*, la *clau* i el *ciphertext* tenen els tres una llargària de 128 bits, de tal manera que si s'han d'entrar tots els bytes d'una tongada es requeririen $128 \cdot 3 = 384$ pins. Si a això se li sumen els senyals *start*, *done* i *clk*, que ocupen 1 bit cadascun, s'obté $384 + 3 = 387$, el nombre total de pins ocupats que mostra el report.

Una bona manera de solucionar-ho seria emprant un *serialitzador* i anar entrant, per exemple, byte a byte els diferents senyals grans. Atès que no s'ha fet la implementació experimental damunt d'una FPGA física s'ha decidit de negligir aquest pas donat que, en definitiva, no aporta cap millora a la implementació arquitectònica del *topAES* (el qual és el focus del treball).

En els següents subapartats es tractaran en detall certs aspectes d'interès sobre la síntesi obtinguda.

4.3.1. Esquematització de l'arquitectura de síntesi.

En l'apartat dels *Annexos* A.3 es mostra l'esquemàtic generat per l'*Intel Quartus Prime* sobre el resultat de la *netlist* obtinguda en fer la simulació. És un esquemàtic molt aparatós i gran i s'ha decidit de partir en dues pàgines per a millorar la visibilitat d'aquest. A més a més, s'ha indicat amb requadres i textos els diferents blocs i entitats que conformen l'esquemàtic.

En aquest apartat s'observarà l'interior d'algun dels blocs i identitats per a comprovar que, efectivament, l'arquitectura de la síntesi és la que s'escau a les descripcions fetes en VHDL i el disseny arquitectònic presentat al Capítol 3.

Esquemàtic de subBytes i Sbox.

Són molt senzills i tenen molt de sentit. La *subBytes*, tal com s'havia dit a l'apartat pertinent del 3.2.1, instancia 16 vegades la *Sbox* i passa byte a byte per ella. La *Sbox* conté un grup de multiplexors que seleccionen un dels bytes del nombre *256h52379de7*(...), que seria la representació de la *lookup-table Sbox*.

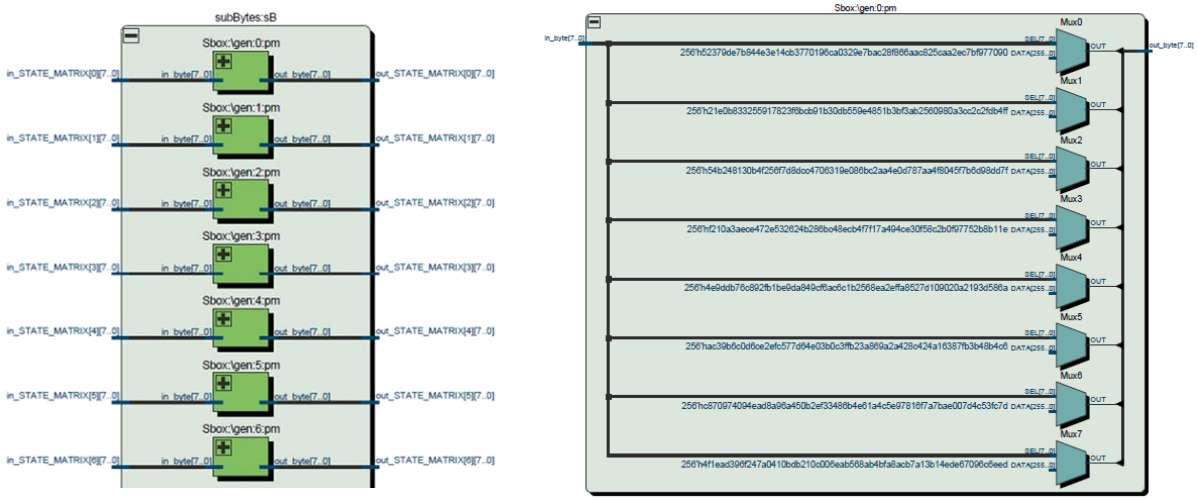


Figura 46. *Esquemàtic del netlist obtingut per simulació de les entitats subBytes (esquerra) i Sbox (dreta)*

Esquemàtic de la addRoundKey.

Veiem que, simplement, l'esquemàtic mostra una suma XOR entre la *roundKey* i la *SM_rond*.

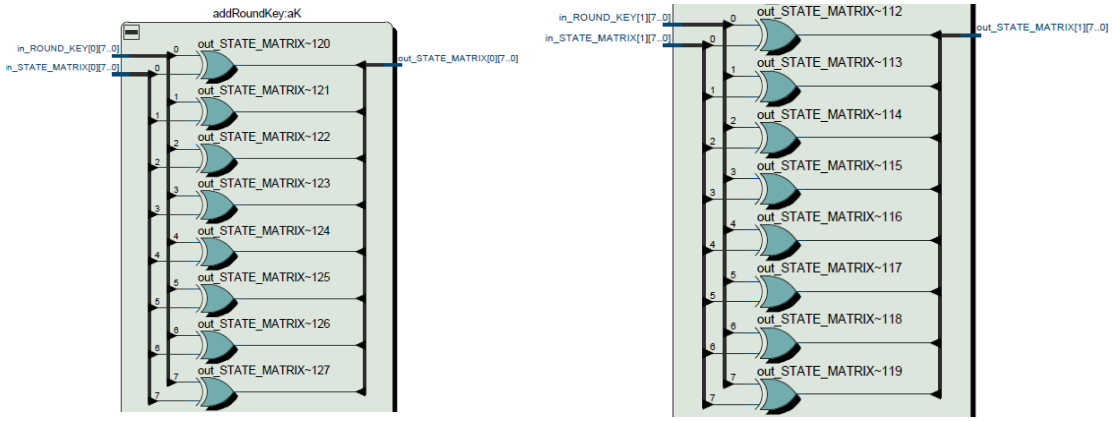


Figura 47. *Esquemàtic del netlist obtingut per simulació de l'entitat addRoundKey (es mostra el procés per 2 bytes)*

Esquemàtic de la mixColumns.

La figura mostra la transformació d'una de les columnes de la *State Matrix*. Es pot comprovar que, tal com s'havia dissenyat a l'arquitectura, cadascun dels bytes de la columna passa en primera instància per la caixa de color verd anomenada *galoisDouble*, la qual el multiplica per 02 segons les regles de la multiplicació de Galois. Aleshores, es produeix la suma XOR dels diferents bytes *doubled* i sense duplicar segons s'escaigui. A la figura és difícil d'apreciar donada la gran quantitat de línies de cablejat que mostra, però si es fa el seguiment es podrà comprovar que encaixen amb els de la *Figura 26* que s'ha mostrat a l'apartat 3.2 del Capítol 3.

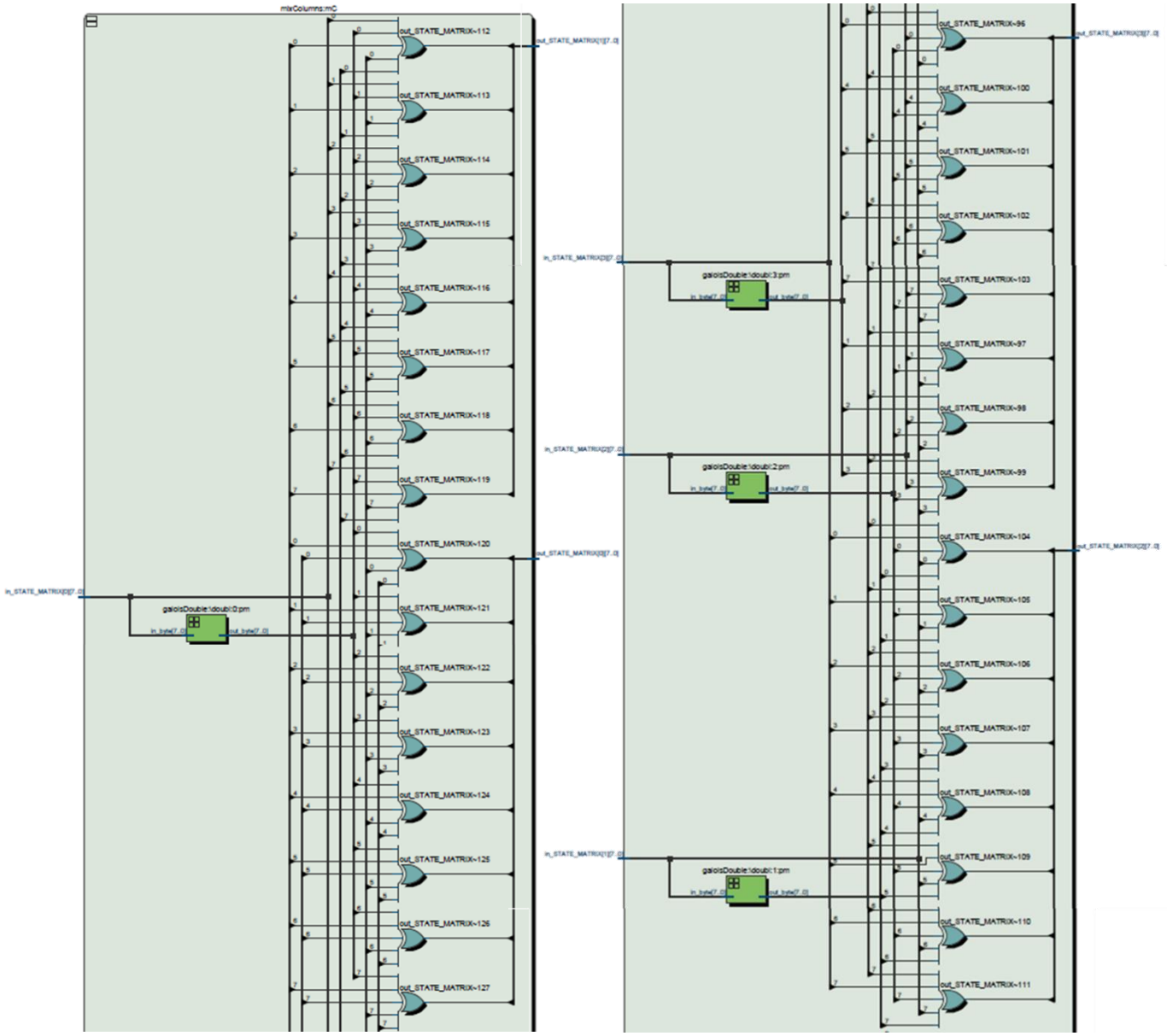


Figura 48*. Esquemàtic del netlist obtingut per simulació de l'entitat mixColumns, on es mostra el procés tan sols per una columna de la State Matrix.*

4.4. Avaluació de la robustesa envers un SCA.

Per tal de comprovar la robustesa del *topAES* dissenyat enfront d'un *side chanel attack* (SCA) s'ha decidit d'emprar la simulació per computador. Tal com s'ha mencionat en l'apartat 1.2.3, es farà un *Power Consumption Analysis* (PCA), és a dir, es traçarà el consum de corrent i se li aplicarà un *Differential Power Analysis* (DPA).

Més concretament, s'ha emprat el programa *OrCAD Capture – Lite* per tal de dissenyar i implementar el model electrònic i el *OrCAD PSpice – Lite* per a fer les captures dels corrents de consum. Aquests dos programes es consideren eines EDA (*Electronic Design Automation*) que serveixen per a editar esquemàtics de forma gràfica i per a simular-los, respectivament. Addicionalment, es farà ús del *Matlab R2020a* per als càlculs pertinents i, ocasionalment, codis en Python3.

4.4.1. Fonaments previs.

En els PCA experimentals s'enregistra la traça de consum de corrent durant tot el temps que dura el procés d'encriptació del bloc de xifratge, però tan sols es considera per a l'atac per se l'interval de temps on hi hagin les dades més bones per a arribar a bons resultats. En el cas de l'AES-128, el millor moment per a fer l'anàlisi és l'instant en què es duu a terme l'execució de la *subBytes* de la primera ronda [17]. En la *Figura 49* es mostra, sobre el disseny hardware realitzat, el moment i el punt precisos on es faria l'atac.

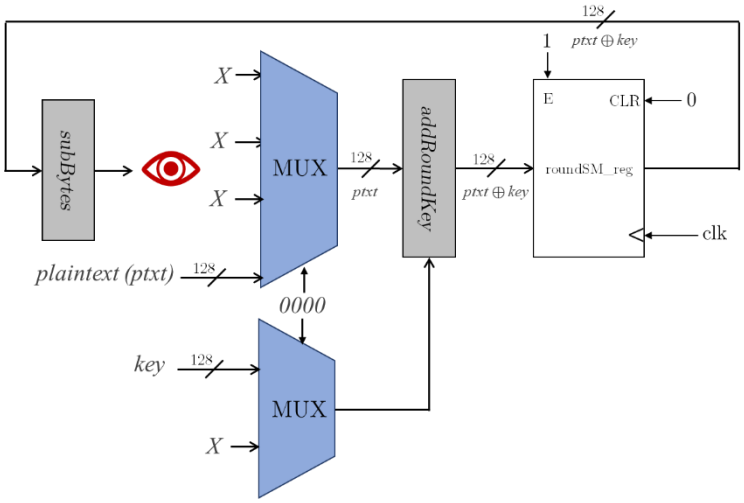


Figura 49. *Esquemàtic d'alt nivell que indica l'estat de les unitats d'encriptació i expansió de la clau del topAES en el moment de realització de l'atac (l'ull representa el punt on es monitoraria la traça de consum).*

Les raons són senzilles. Es fa al principi de la primera ronda perquè en aquest instant la *State Matrix* ha estat addicionada amb la clau original, sense ser expansionada, i perquè encara no s'ha permutat cap dels seus bytes (i. e., el byte i del *plaintext* està únicament relacionat amb el byte i de la *State Matrix* actual). La sortida de l'operació *subBytes* és escollida per dos motius: el primer, que és una operació que no permuta els bytes i manté la relació directa mencionada, i el segon perquè el seu consum de corrent sol ser major que el de la resta (i, per tant, és més fàcilment identificable en la traça de consum de corrent total) [17].

Consum de corrent en una FPGA.

Les FPGA tenen un consum de potència molt més elevat que les seves alternatives; cosa que comporta que, per exemple, les ASIC segueixin imposant-se en grans implementacions com ara processadors d'ordinadors potents. Segons l'estudi referenciat [32], un 50-70% d'aquest consum es dissipa en la *xarxa d'interconnexió* entre els diferents blocs lògics de l'FPGA, i ho atribueix a la presència d'una alta capacitància paràsita[20]. Aquesta, indica, radica en el fet que per a fer una operació cal connectar alhora una gran quantitat de blocs lògics (cosa que, per definició, fa augmentar la capacitància) i indica que una bona manera de reduir-la seria emprar eines d'estimació i optimització de la capacitat.

En el cas que ocupa a aquest treball, aquestes indicacions es tradueixen com que el consum *de corrent*, relacionat amb el *de potència*, és significatiu en la *xarxa d'interconnexió* que enllaça els diferents blocs que implementen les entitats del *topAES*. En conseqüència, una bona manera d'orientar l'atac seria modelitzar un bus d'aquesta xarxa i intentar trobar-ne la traça de corrent.

Per a entendre millor l'origen de la capacitància paràsita, es remet a l'explicació de l'apartat 1.2.3 il·lustrada a la *Figura 10* sobre el consum dinàmic de la tecnologia CMOS.

Transmissió de senyal per bus amb múltiples buffer.

Un *buffer digital* és una porta lògica que se sol emprar per a aïllar els diferents valors lògics 0 i 1 (i. e. tensions de nivell baix/alt) que al llarg del temps va prenent un senyal que es transmet a través d'un bus. En CMOS, els *buffers* solen ser dos circuits inversors (portes lògiques NOT) tal com el circuit mostrat en la *Figura 50*.

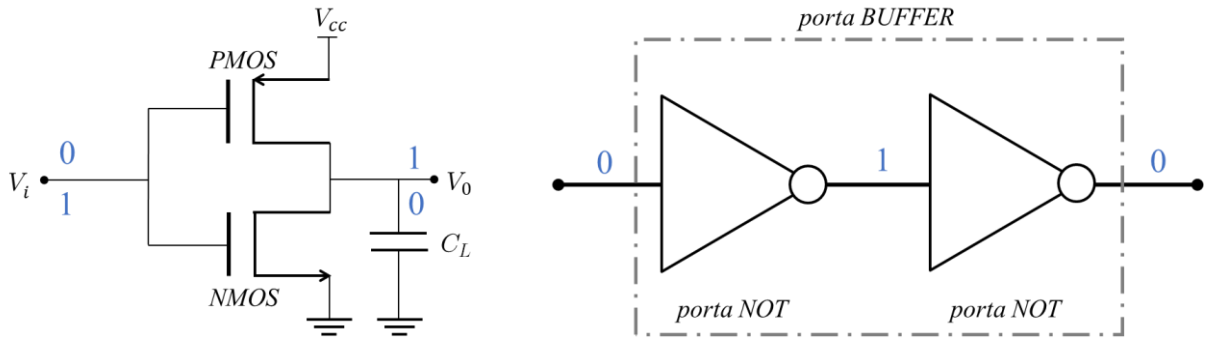


Figura 50. *Inversor en tecnologia CMOS equivalent a una porta NOT (a la esquerra) i manera d'implementar un buffer digital mitjançant dues portes NOT consecutives.*

Quan es dóna el cas que la capacitància d'un bus és significativa[21], els inversors continguts en el *buffer* tarden massa temps a establir el senyal de sortida (donat que el temps de càrrega/descàrrega de la capacitància és elevat). La solució sol raure a concatenar més d'un *buffer* per tal d'augmentar la transmissivitat i reduir el retard. Per a què això funcioni, cal assegurar-se que la conductivitat sigui major de *buffer* a *buffer* per tal de vèncer, també, les pròpies capacitàncies contingudes dins dels inversors que els formen. L'única manera de fer-ho és baixar a nivell de transistor.

[20] La *capacitat paràsita* és una capacitància no desitjable, però irremeiable, que existeix entre els diferents blocs d'un sistema electrònic digital a causa de llur proximitat.

[21] Com, per exemple, en els busos entre la CPU i les cel·les de memòria en un microcontrolador a causa dels perifèrics.

En un transistor, que la conducció a través d'ell (i, per consegüent, a través de l'inversor que implementa) sigui major o menor depèn de la seva constant de conducció, la qual s'escriu:

$$K_n = \frac{1}{2}\mu_n C_{ox}\frac{W_n}{L} \quad NMOS \qquad K_p = \frac{1}{2}\mu_p C_{ox}\frac{W_p}{L} \quad PMOS$$

On C_{ox} és la capacitat de l'òxid per unitat d'àrea, μ_n i μ_p les constants de mobilitat dels electrons i els forats, respectivament, i W i L l'amplitud del transistor i la llargària de la porta G. En una tecnologia concreta, els paràmetres C_{ox}, μ_n i μ_p són immutables perquè depenen del material i, almenys en CMOS digital, es fa servir sempre una L fixa. Per tant, per a millorar la mobilitat tan sols es pot modificar la W.

Val a dir que cal fer la distinció entre MOS tipus n i tipus p donat que la mobilitat dels electrons és major que la dels forats ($\mu_n > \mu_p$), de tal manera que se sol triar $W_n < W_p$ per tal d'equiparar els temps de pujada a 1 i de baixada a 0 (i. e. de càrrega i descàrrega de la capacitat de porta).

Seleccionant uns paràmetres W i un nombre d'inversors adients, és possible obtenir una millora considerable en la transmissió del senyal. Com es veu en la *Figura 51*, en el cas on tan sols hi ha un inversor i molta capacitància al bus existeix un retard molt gran entre l'instant en què el senyal d'entrada arriba a nivell 1 i el moment en què l'output s'estableix a 0. En canvi, en el cas de múltiples inversors (i, per tant, múltiples *buffer*) de diferent mida s'aconsegueix reduir el retard fins al punt que aquest, segurament, tan sols tingui a veure amb el retard de propagació inherent a tot sistema electrònic.

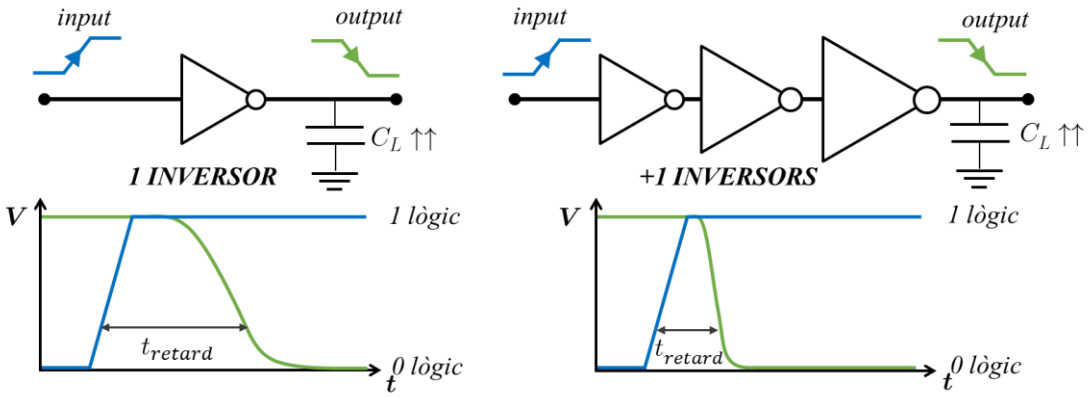


Figura 51. *Diferència en l'establiment del senyal de sortida del bus en cas d'un únic inversor o múltiples inversors amb transistors de diferent mida.*

A la pràctica, el que se sol fer és, donada una tecnologia CMOS amb una L determinada, escollir una W_n prou gran, com ara el doble de la longitud de porta G, i se selecciona la subseqüent W_p com:

$$W_p = r\ W_n\ ; \qquad r \in [2, 3]$$

Respecte al nombre de *buffers* necessaris, això depèn de cada cas en particular. El que si que sol ser usual és seleccionar la W_p de l'inversor com la W_n duplicada de l'inversor anterior, tal que la "mida" física dels *buffers* vagi augmentant al llarg del bus.

4.4.2. Model emprat per a l'atac.

Com s'ha comentat en l'apartat anterior, en l'AES-128 se sol escollir l'instant en què s'executa la *subBytes* a la primera roda per a fer-hi el PCA. En concret, en aquest treball es rastrejarà la traça de consum del bus de sortida d'aquesta operació donat tot l'exposat sobre l'alta capacitància de la *xarxa d'interconnexions* d'una FPGA.

Tal com s'ha dissenyat la implementació *topAES*, l'output de la *subBytes* s'ha de transmetre directament a la *mixColumns*[22] i també al multiplexor de la unitat d'encriptació (que té també per entrades la sortida de la *mixColumns* i el *plaintext*), el senyal de selecció del qual és *num_rond* que prové del registre comptador *counter*. A més a més, podria ser que en la implementació física en FPGA aquest senyal anés també a parar a algun perifèric d'aquesta.

En conseqüència, és possible que el bus de sortida de la *subBytes* sigui el de major capacitància paràsita d'entre tots els de les altres operacions. Tanmateix, això no es pot saber amb certesa sense la implementació física, tot i que sí que es pot assegurar amb fermesa que n'hi haurà, i molta.

Tot això s'ha modelitzat com un bus de 8 línies (on cadascuna d'elles representa un bit) que conté quatre inversors (i. e. dos buffers) de mida creixent. L'entrada a aquest bus seria un dels bytes de sortida de la *subBytes* o, més específicament, d'una de les setze instàncies de l'entitat *Sbox*. A la *Figura 52* es mostra una representació esquemàtica d'aquest model.

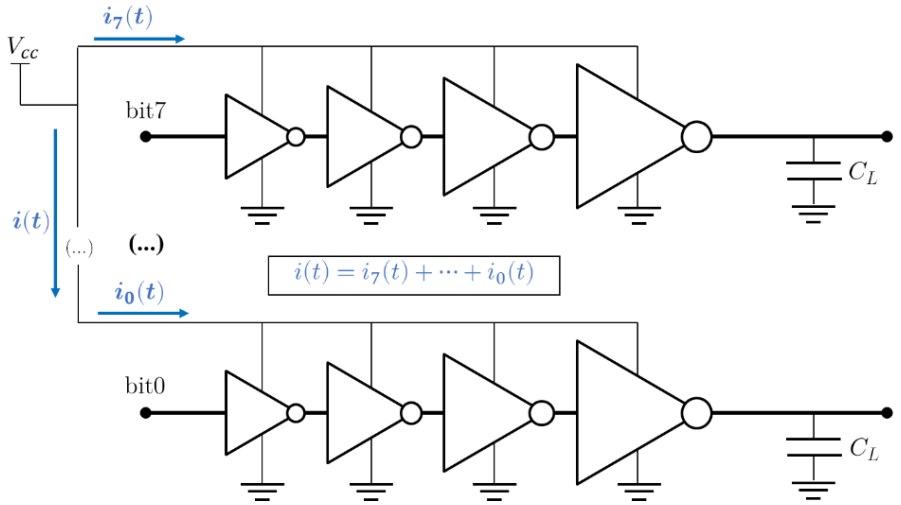


***Figura 52**. Esquemàtic del model emprat per a executar l'atac PCA per simulació.*

Cal posar l'accent damunt la xarxa d'alimentació, connectada a una font de tensió V_{cc}, ja que la intensitat del corrent $i(t)$ que hi circula és la que es vol traçar. A la pràctica, el model es pot simplificar entenent que cada línia l del bus té un corrent d'alimentació propi $i_l(t)$ connectat en paral·lel amb la resta, de tal manera que el consum $i(t)$ es pot escriure com la suma de la resta de $i_l(t)$:

$$i(t) = \sum_{l=0}^{7} i_l(t)$$

[22] La connexió amb la *mixColumns* es fa a través de la permutació de la *shiftRows*. No obstant, com ja s'ha vist en el seu subapartat d'arquitectura corresponent del 3.2.1, aquesta funció se sintetitza en la interconnexió i no en un bloc lògic.

Disseny amb l'OrCAD Capture.

El disseny s'ha fet de forma jeràrquica. És a dir, primer s'ha definit el circuit a nivell transistor, després a nivell portes i finalment a nivell *register-transfer* (RTL).

Els quatre inversors de la línia s'han dissenyat mitjançant la mateixa lògica. Es pot veure l'esquemàtic a la *Figura 53*, el qual representa el més petit de tots (i. e. el "inversor base").

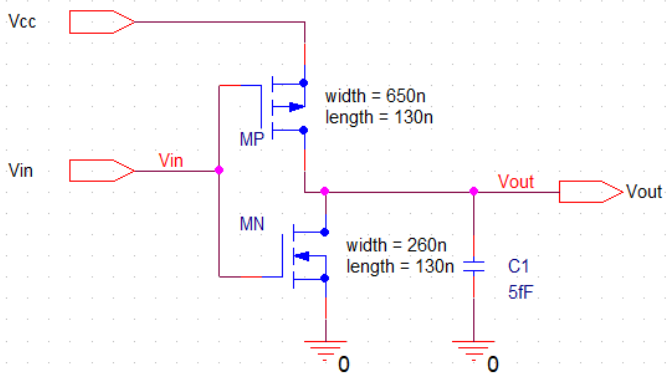


Figura 53. *Model de l'inversor base dissenyat per a simular l'atac a nivell de transistor.*

S'ha seleccionat una tecnologia CMOS tal que la llargària de porta G és $L = 130nm$ i el paràmetre $W_n = 2 \cdot L = 260nm$ per al tipus N. Emprant una raó $r = 2{,}5$ s'ha determinat que $W_p = rW_n = 650nm$. La resta d'inversors mantindran un mateix valor de L, però el paràmetre W serà sempre el doble de l'anterior inversor.

Les entrades són *Vcc*, que representa la tensió d'alimentació i *Vin*, per on passarà la informació del byte de cada línia. La sortida és *Vout*, que és el negat de *Vin*.

Respecte a la capacitància, s'ha seleccionat un valor que estigués en consonància amb els valors predits en l'estudi [32] sobre l'estimació de capacitàncies en una FPGA. S'ha decidit d'implementar-la només dins de cada inversor com a condensador atès que, a més a més de la capacitància paràsita, també s'ha de tenir en compte la pròpia de l'inversor. S'ha considerat que $C_I = 5fF$ seria suficient per a cada porta, ja que en la línia es tindria $C_L = 4 \cdot 5fF = 20fF$, cosa que implica que en tot el bus hi hagi una capacitància de $C_B = 8 \cdot 20fF = 160fF = 0{,}16pF$. Si es consideren els 16 bytes, aquesta capacitància puja a $C_T = 16 \cdot 0{,}16pF = 2{,}56pF$, valor versemblant a jutjar per [32].

La *línia de dades del bus d'1 bit* quedaria, a nivell de portes, com es veu a la *Figura 54*.

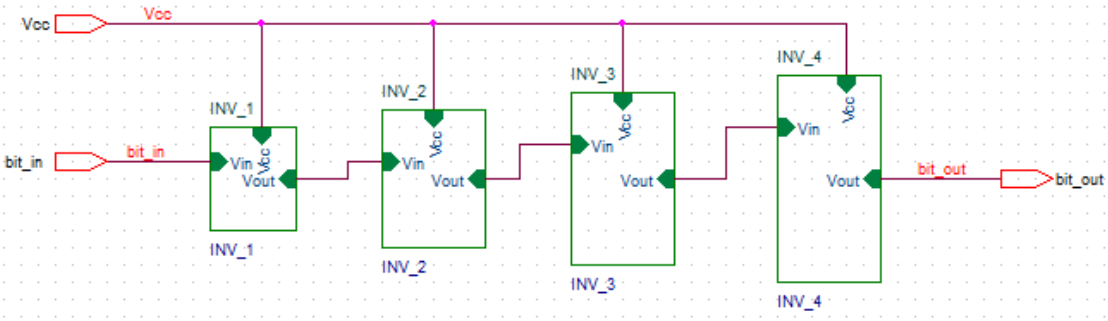


Figura 54. *Model de la línia de dades d'1 bit a nivell de portes dissenyat per a simular l'atac.*

Simulació del model amb el PSpice.

Com s'ha comentat sota la *Figura 52*, el corrent de consum total de la font d'alimentació és el paral·lel de les diferents intensitats de consum de cadascuna de les vuit línies del bus d'un byte. A més a més, el consum significatiu en CMOS és el dinàmic i, per tant, tan sols interessen els casos on es passa de tenir un 0 lògic a un 1 lògic, i quan es passa de 1 a 0.

En conseqüència, la simulació amb el *PSpice* es pot reduir a un model que contingui tan sols una d'aquestes línies d'1 bit del byte de sortida de la *subBytes*; el qual és conegut i controlable. Aleshores, obtenint la traça de consum de pujar de 0 a 1 i baixar de 1 a 0 (lògics) aquesta línia es podria, per a cada cas particular, obtenir la traça de consum total com:

$$i(t) = l_{0\to1}\ i_{0\to1}(t) + l_{1\to0}\ i_{1\to0}(t)$$

On $l_{0\to1}$ són el nombre de línies que passen de 0 a 1 lògics i $l_{1\to0}$, les que passen de 1 a 0 lògics. Cal afegir, però, que per a què aquest model funcioni cal conèixer sempre l'estat inicial del bus de dades. La implementació de tot això es veurà més endavant.

El sistema emprat per a fer-hi la simulació és el de la *Figura 55*, on cal destacar que la font de tensió és $V_{cc} = 1{,}5V$, suficient per a la tecnologia CMOS amb $L = 130nm$. Com a V_{OL} i V_{OH} s'ha considerat, per simplicitat, els valors de $0V$ i $1{,}5V$.

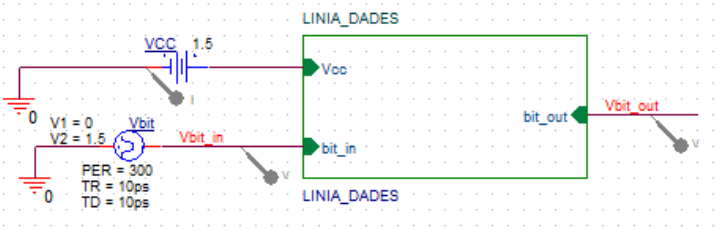


Figura 55. *Model del "set-up" experimental per a l'obtenció de les traces de corrent, amb tensió d'alimentació a 1,5V.*

S'ha fet una simulació de prova per comprovar el funcionament d'aquest model i comprovar si la propagació del senyal és bona.

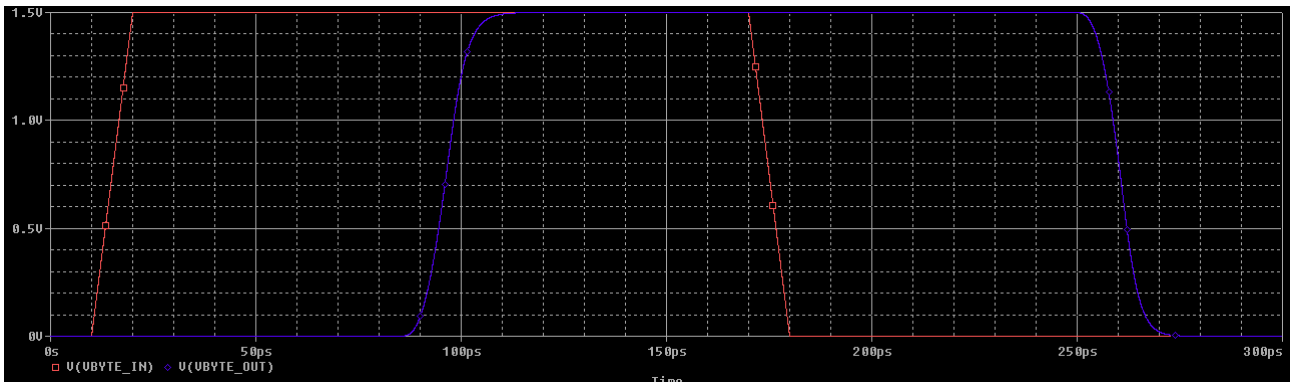


Figura 56. *Formes d'ona del senyal d'entrada, que és un VPULSE, en vermell i el senyal de sortida de la línia, en blau.*

En la *Figura 56* és possible apreciar que el model és versemblant i correcte. El retard de propagació és menor a 100ps, d'uns 92ps en la pujada i uns 95ps a la baixada. El temps d'establiment del senyal són d'uns 27ps en la pujada i uns 24ps en la baixada. És a dir, s'assoleix un ordre de magnitud de retard adequat als que es tenen com a referència en una FPGA.

Per a aquesta mateixa simulació, en la *Figura 57* es mostren els dos corrents de consum d'interès.

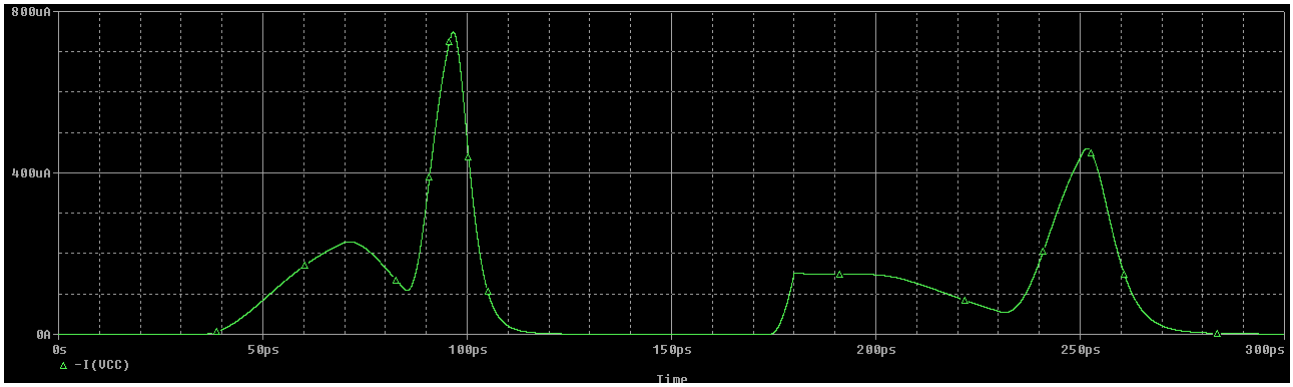


***Figura 57**. Traça de consum (en μA) del model en pujar el senyal de 0 a 1 (esquerra) i baixar de 1 a 0 (dreta)*

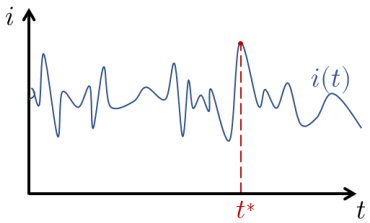
L'ordre que s'ha obtingut, considerant que és d'una única línia, és correcte. Els màxims per a la pujada de 0 a 1 és de $0{,}749mA$ i, per a la baixada, de $0{,}460mA$.

4.4.3. Realització de l'atac i DPA per simulació.

En aquest apartat es parlarà de com s'ha efectuat l'atac PCA al *topAES* partint del model dissenyat amb l'*OrCAD* anteriorment mencionat. Abans d'entrar en matèria, però, és convenient exposar de quina manera es duen a terme els *Differential Power Analysis* (DPA) amb el *Hamming weight model*.

Com a hipòtesis de partida, seguint els principis de Kerckhoffs, se suposa que es pot controlar l'entrada del *plaintext* al sistema, que l'AES és un algorisme públic i que la clau que s'està emprant ara mateix és desconeguda per a l'adversari. L'objectiu del DPA és obtenir els valors exactes de tots els bytes d'aquesta clau. Per a fer-ho, es genera un nombre considerable de *plaintext* amb què anar excitant el sistema a atacar (en aquest cas, el *topAES*) i s'obté la traça de corrents de consum.

En cas de realitzar l'atac de forma experimental, la traça de consum obtinguda és organitzada en una matriu que rep per nom Matriu de Consums Temporals (MCT per abreviar), on cadascuna de les columnes representa un dels instants temporals de captura, i cada fila conté els valors de la intensitat dels diferents *plaintext* generats. Una d'aquestes columnes representa l'instant t^* en què es produeix l'operació *subBytes* de la primera ronda i que és, a la pràctica, l'única informació d'interès d'aquesta matriu. En la *Figura 58* es veu un exemple de MTC per un byte arbitrari on es ressalta l'instant dit.



byte posició B	t_1	t_2	...	t^*	...	t_T
plaintext 1	2342	2134	...	3556	...	3556
plaintext 2	1431	2342	...	2455	...	2455
...	...	...	...	...	...	...
plaintext N	2122	1451	...	3552	...	3552

***Figura 58.** Exemple de Matriu de Consums Temporals per un byte d'una posició arbitrària B en cada plaintext on es destaca la columna en què es produeix el càlcul de la subBytes de la primera ronda (els valors són inventats).*

En total es generen setze MCT, una per cadascun dels bytes a les 16 possibles posicions dels *plaintext*. El motiu d'aquest fet radica en el fet que, per a poder executar l'anàlisi estadístic del DPA pròpiament dit, cal seguir un procediment byte a byte. Més endavant s'entendrà el perquè.

Un cop obtingudes les MCT, cal calcular les anomenades Matrius de Consums Estimats (MCE) amb què es compararan. Aquestes matrius es calculen fora del *set-up* experimental ja que són purament teòriques, i s'empra una versió simplificada de l'AES per a determinar els seus valors. Com amb les MCT, n'hi ha setze ja que es treballa byte a byte.

Partint d'un dels bytes situat a la posició B dels *plaintext* generats, se li aplica la *addRoundKey* i la *subBytes* emprant com a clau un dels 256 possibles valors d'un byte. Aleshores, atès que aquest treball usa el *Hamming weight model*, es calcula el *Hamming weight* del valor resultant. Aquest procediment s'ha de repetir per tots els 256 valors possibles del byte de la clau i s'ha de fer per tots els bytes de totes les posicions de tots els *plaintext* generats. Tot això s'ha il·lustrat a la *Figura 59*.

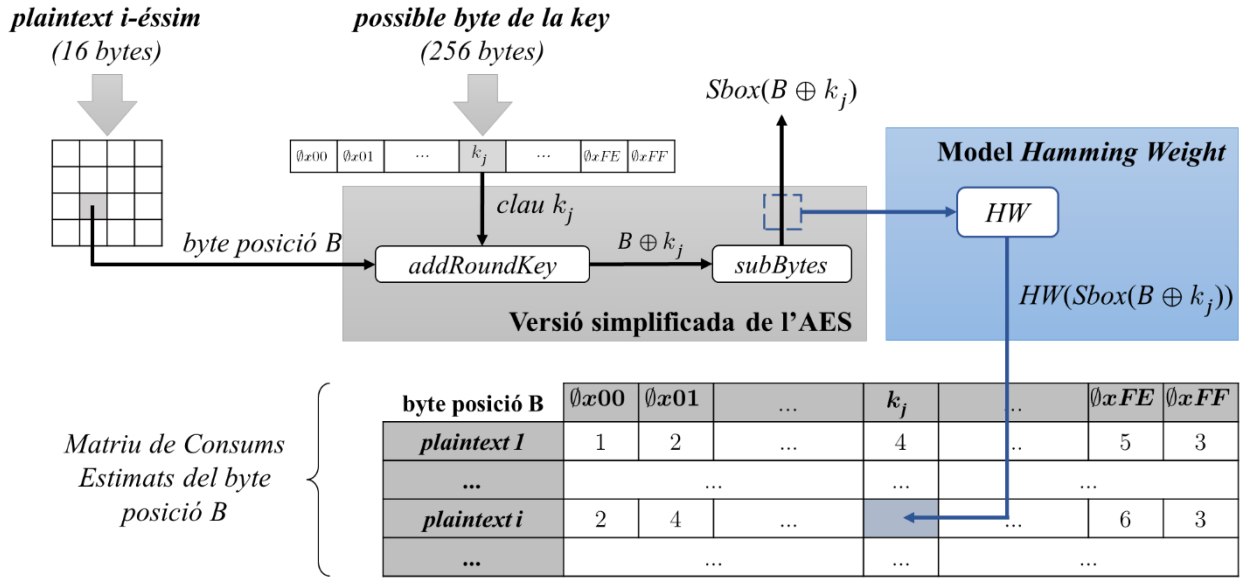


Figura 59. *Esquematització del procediment d'obtenció del consum estimat pel Hamming Weight model d'un dels plaintext pel byte situat a la posició B amb byte de clau de valor k_j*

Obtingut el seguici de 16 matrius MCT i MCE, es procedeix a fer una anàlisi de correlació entre cada columna de la MCT amb les 256 de la MCE (de bytes homònims). Atès que, com s'ha comentat abans, una de les columnes de MCT conté el valor del consum de corrent en l'instant d'interès t^*, la correlació d'aquesta amb la columna de MCE que té per byte de clau l'homònim a la *clau real* del sistema serà molt significativa.

En aquest treball, l'instant t^* és conegut ja que, en fer-se per simulació, l'interval de temps amb què es treballa és mol petit. Per tant, el que es tindrà seran 16 columnes de MCT relatives a l'instant t^*, una per cada posició de byte, que caldrà comparar amb les 256 columnes (E) de la matriu MCE de posició de byte homònima. L'útil estadístic emprat és el coeficient de correlació lineal de Pearson:

$$\rho_{t^*,E} = \frac{cov(t^*,E)}{\sigma_{t^*}\ \sigma_E} = \frac{\sum\{\ t^* - \mu_t^*\quad E - \mu_E\ \}}{\sum\{(t^* - \mu_t^{*\ 2}\}\ \sum\{(E - \mu_E{}^2\}}$$

Atès que la correlació es fa per columnes, s'obtindrà una única columna per cadascun dels bytes de la posició B del plaintext de 256 valors de $\rho_{t^*,E}$, un per cada possible valor de la clau. Aquestes columnes s'organitzen en la Matriu de Correlacions (MC) que té 16 files, una per cada byte, 256 columnes, una per cada valor de la clau, i conté totes les $\rho_{t^*,E}$.

A partir de la MC es pot fer un gràfic que mostri els diferents valors de la correlació (ordenades) en funció del valor de la clau (abscisses). Aleshores, el pic de correlació es donarà sempre en el valor de la clau correcta per a aquella posició de byte, trencant així l'AES i obtenint la clau. En la *Figura 60* s'il·lustra aquest procediment final, amb una representació del gràfic mencionat.

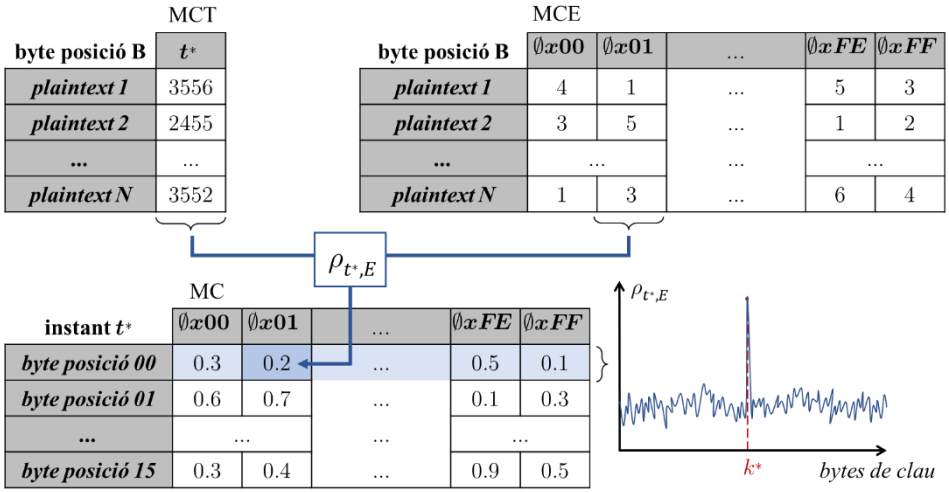


***Figura 60**. Procediment per determinar una cel·la de la Matriu de Correlacions i gràfic de correlació en funció de byte de clau (al costat inferior dret).*

Obtenció de les traces de consum i de la matriu de correlacions.

Tal com s'ha exposat en l'apartat 4.4.2, s'ha obtingut el corrent de consum d'una línia de dades quan aquesta passa de 0 a 1 lògics, $i_{0\to1}(t)$ (s'anomenarà *de pujada*), i quan passa de 1 a 0 lògics, $i_{1\to0}(t)$ (s'anomenarà *de baixada*). Aleshores, és necessari conèixer l'estat inicial del sistema per tal de poder obtenir el consum total.

En el cas del *topAES*, abans d'iniciar l'encriptació el senyal d'*start* està a 0 (és a dir, *reset* a 1) i tots els registres paral·lels tenen el senyal de CLR activat. Per tant, el registre de la *State Matrix* de la ronda està mantenint el senyal 0...0 i, en conseqüència, la *subBytes* el rep com a entrada i en calcula la sortida. Mirant la *Sbox* es pot comprovar que, donada una entrada 0...0, la sortida és $\emptyset x63$... $\emptyset x63$[23].

Això implica que, quan es faci la primera ronda, l'estat inicial del bus del model serà $\emptyset x63$, en binari $\emptyset b01100011$. Això s'ha modelitzat en Matlab com la funció *consum*:

```
function I = consum(B, Irise1, Idown0)
bits = dec2bin(B, 8);

L_1to0 = count([bits(2),bits(3),bits(7),bits(8)], '0');
L_0to1 = count([bits(1),bits(4),bits(5),bits(6)], '1');

I = Irise1.*L_0to1+Idown0.*L_1to0;
end
```

És a dir, entrat el byte B (en Matlab s'ha d'entrar en decimal) es conten els bits nuls coincidents amb les posicions on $\emptyset x63$ té uns i es multiplica pel consum *de pujada*, i viceversa. La suma de les dues casuístiques dóna el consum de corrent total I que s'estava buscant.

[23] Per més informació i detall consultar l'apartat 4.2.2, on s'exposa el funcionament intern, o la totalitat del Capítol 3.

Amb aquesta funció s'ha escrit un codi en Matlab per tal de determinar la MCT particular de l'atac que s'està duent a terme en aquest treball, el qual es pot consultar al A.4.1. Cal destacar que aquest codi pren per dades els fitxers CSV *Irise_to_1* i *Idown_to_0* que representen els consums del model d'*OrCAD* de pujada i de baixada comentats anteriorment i el fitxer *ciphertext*, que conté la sortida de la *subBytes* de la primera ronda de 500 plaintext generats aleatòriament emprant el matlab.

Com a part important es destaca el següent tros del codi, on *m* és 500 (el nombre total de *plaintext* generats), *bytes* són els 500 bytes de la posició actual, *Irise1* i *Idown0* els consums de pujada i baixada del model d'*OrCAD*. Aquesta part del codi representa l'obtenció dels 500 corrents de consum *Iconsum* de cada byte:

```
for i=1:m
    Iconsum = consum(bytes(i,:),Irise1,Idown0)'; % consum equivalent
    Iconsum = Iconsum + 100e-6 * randn(size(Iconsum)); %afegim soroll gaussià
    maxIconsum = [maxIconsum; max(Iconsum,[],'all')]; % ens interessa el màxim
end
```

Per tal d'apropar-se més a la versemblança experimental, s'ha afegit soroll gaussià de $100\mu A$ de desviació estàndard. A més a més, atès que els corrents de consum emprats per a calcular *Iconsum* provenen de simulació, l'instant t^* coincideix en tots els 500 bytes de la posició actual amb el seu punt de consum màxim, aproximadament. Així doncs, s'ha definit *maxIconsum* com la columna t^*.

En la *Figura 61* es pot contemplar un dels 500 corrents *Iconsum* aleatori sense soroll gaussià i amb soroll gaussià per tal d'entendre quin aspecte tenen.

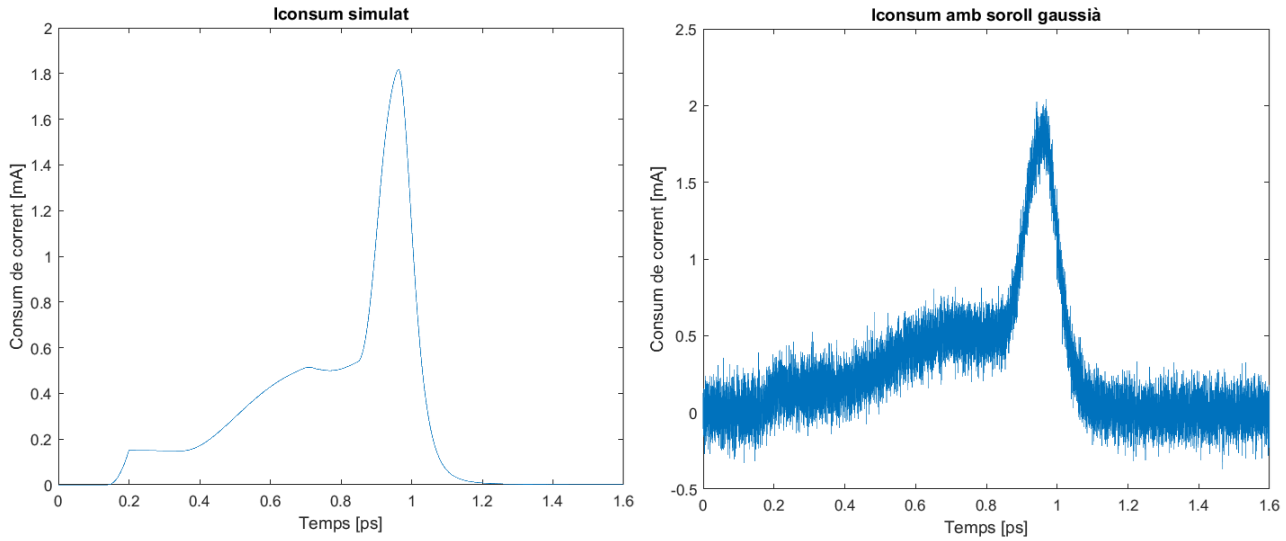


Figura 61. *Consum de corrent d'un byte d'un plaintext (esquerra) i el mateix consum però amb soroll gaussià (dreta).*

Aleshores, s'ha instanciat una *MatriuConsums* general, la qual conté totes les columnes t^* de les 16 posicions de byte possibles pels 500 plaintext (les *maxIconsum*). Aquesta matriu s'ha emprat per a fer les correlacions amb les MCE, que s'han obtingut per mitjà d'un codi en Python3 que es pot consultar als A.4.3 dels *Annexos*, les quals s'han guardat en fitxers CSV separats de nom *matriuHW#*, on # és el nombre de la posició de byte.

Mitjançant el darrer bloc del codi del A.4.1 dels *Annexos*, s'ha obtingut la matriu de correlacions sense necessitat de fer cap comentari ni que suposes cap dificultat a destacar.

4.4.4. Discussió dels resultats de l'atac.

La versió simplificada de l'AES emprada per a trobar les 500 sortides de la *subBytes* a partir dels 500 *plaintext* generats aleatòriament es troba al A.4.2 dels *Annexos.* La clau secreta que s'ha considerat que el sistema estava emprant és la que es mostra a la taula en codi hexadecimal i decimal (ja que el matlab treballa millor amb el darrer):

	clau desconeguda per l'adversari															
codi hexadecimal	54	68	61	74	73	20	6D	79	20	4B	75	6E	67	20	46	75
codi decimal	84	104	97	116	115	32	109	121	32	75	117	110	103	32	70	117

Taula 6. *Clau considerada per a la realització de l'atac en hexadecimal i decimal.*

El resultat del DPA pel *Hamming weight model* es pot veure en la *Figura 62*:

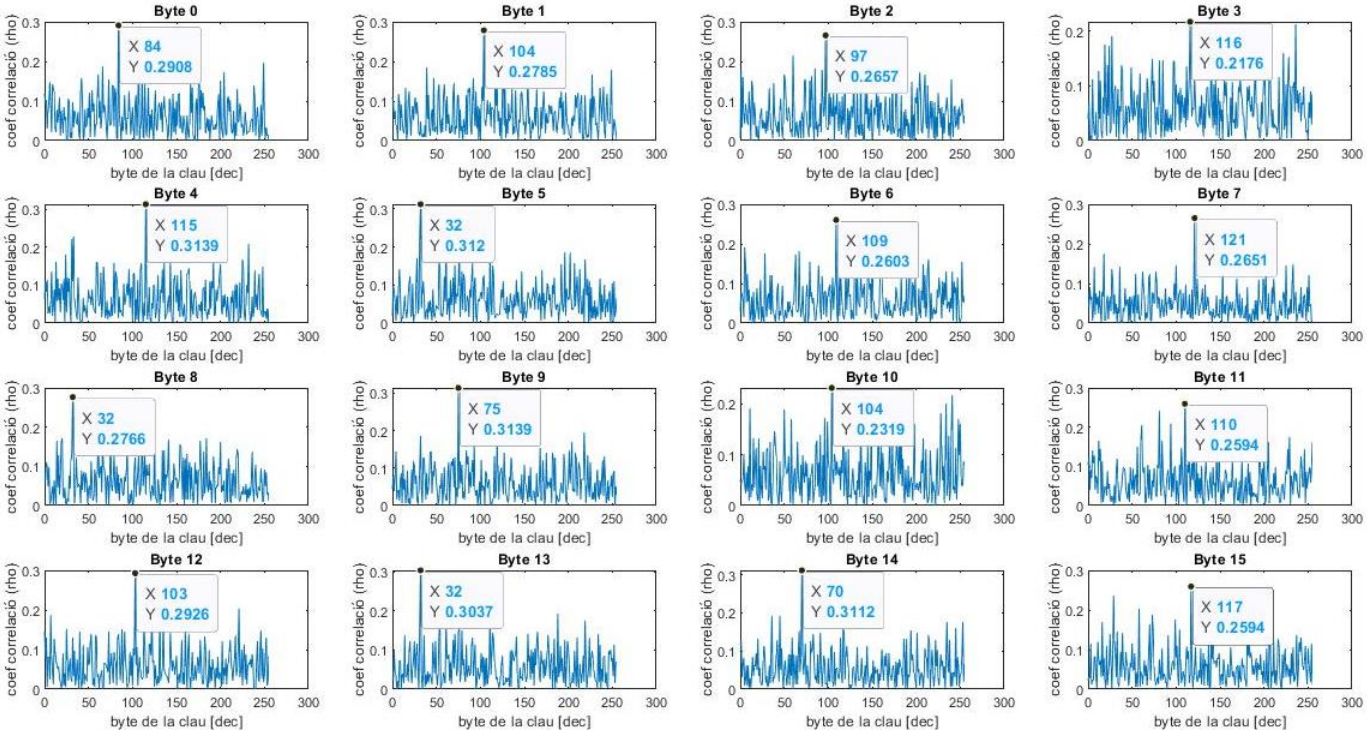


Figura 62. *Matriu de correlacions gràfica del DPA de Hamming weight model a la modelització del topAES plantejada en aquest treball*

D'entrada, destaca el fet que les correlacions obtingudes estan totes en un rang d'entre el 0 i el 0.3 del coeficient de correlació, cosa que segons la teoria explicada al principi de l'apartat no hauria de passar. Els valors de clau màxims mostrats a la figura, si s'examinen un a un amb la clau en versió decimal, es pot veure que són tots correctes a excepció d'un, el del byte de la posició 10, que hauria de ser 117 i no 104 (val a notar que el "següent més màxim" és el 117).

Això es deu principalment al *Hamming weight model*. La matriu MCE s'omple amb els *Hamming weight* dels bytes de sortida de la *subBytes*, de tal manera que el consum que s'està estimant és el corresponent a passar de 0...0 al byte de sortida (ja que els 0 romanen a 0 i el consum és de 0 a 1). Tanmateix, en el cas del *topAES*, com s'ha dit anteriorment, el valor de partida de la línia no és 0...0 sinó que $\emptyset x63$...$\emptyset x63$, el qual en binari s'escriu $\emptyset b01100011$.

És a dir, només els bits de les posicions del byte de sortida de la *subBytes* homònimes a aquelles on hi ha un 0 en el $\emptyset b01100011$ (la primera, quarta, cinquena i sisena) són les que consumeixen corrent de pujada de 0 a 1. Per tant, per a poder obtenir les correlacions idònies caldria tan sols calcular el *Hamming weight* d'aquestes posicions esmentades.

A aquest mètode se l'ha anomenat *Hamming weight model modificat*, ja que manté la mateixa lògica, però s'aplica al cas del *topAES*. La manera amb què s'ha aplicat es pot veure en Python3 als *Annexos* en el A.4.4. El resultat del DPA pel *Hamming weight model modificat* es pot veure en la *Figura 63*:

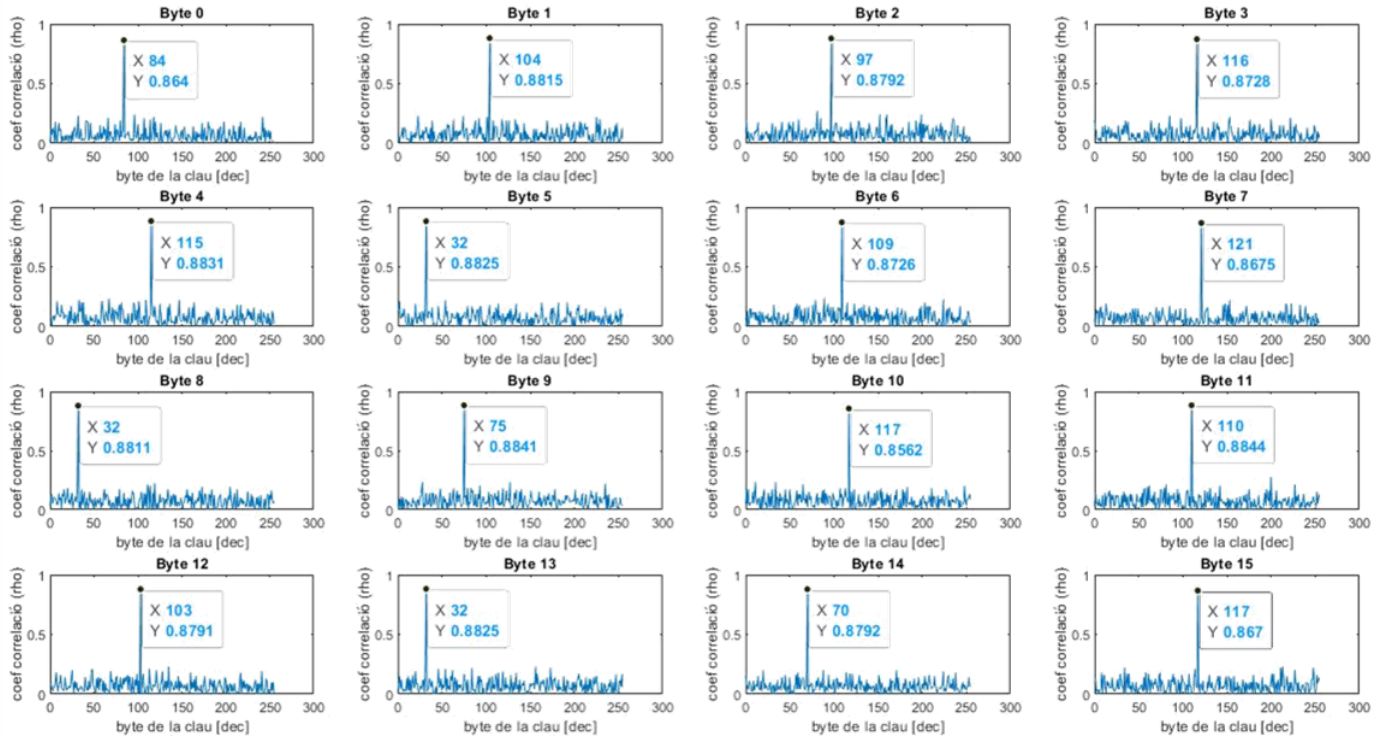


Figura 63. *Matriu de correlacions gràfica del DPA de Hamming weight model modificat a la modelització del topAES plantejada en aquest treball*

Ara sí, tots els bytes de la clau obtinguts en l'atac són els correctes i les correlacions són valors molt alts. Cap d'ells sobrepassa el 0.9, segurament a causa del soroll gaussià aplicat, però també perquè amb aquest *Hamming weight model modificat* només es tenen en compte els valors que passen de 0 a 1 i no aquells que passen de 1 a 0 en el bus de dades modelitzat. Això, pot ser, es podria considerar amb un model que combinés més d'una MCE per cada byte, però això queda fora del llindar d'aquest treball. Amb els valors obtinguts de correlació és possible obtenir conclusions molt significatives.

Queda palès, doncs, que la implementació *topAES* d'aquest treball és vulnerable a atacs de canal lateral (SCA) de consum de potència DPA, tot i que amb les hipòtesis plantejades sobre el funcionament del bus de dades hom es podria aventurar a afirmar que potser seria una mica difícil d'obtenir les claus. Això és, perquè l'adversari podria no conèixer el funcionament del hardware a tan baix nivell com per saber que les línies de dades s'inicien en $\emptyset x63$... $\emptyset x63$ en lloc de en 0...0.

No obstant, en honor al rigor, caldria afegir que tot aquest sistema modelitzat es basa en suposicions fetes per tal d'intentar fer una avaluació *per simulació* de la robustesa envers SCA i podria ser que els resultats obtinguts, en cas de fer-ho per experimentació, fossin lleugerament diferents.

Estudis complementaris.

Planificació del treball.

Aquest treball fou concebut el 10 de febrer del 2021, dia en què es va fer el seu registre, i es va començar a executar a partir del 17 de febrer del mateix any. En el diagrama de Gantt de la *Taula 7* s'ha representat el transcurs temporal de les diferents etapes amb què s'ha anat emprenent aquest treball.

	Febrer	**Març**	**Abril**	**Maig**	**Juny**	**Juliol**
Elecció del tema i registre del TFG						
Recerca teòrica sobre l'AES						
Redacció *Capítol 2* de la memòria						
Implementació en Python3 de l'AES						
Recerca i aprenentatge del VHDL						
Disseny operacions de l'AES en VHDL						
Disseny d'alt nivell de l'AES en VHDL						
Redacció *Capítol 3* i *4* de la memòria						
Síntesi del *topAES* i avaluació resultats						
Recerca d'estat de l'art complementària						
Redacció *Capítol 1* de la memòria						
Revisió i finalització de la memòria						
Atac al *topAES* per simulació						
Dipòsit digital de la memòria						
Preparació i defensa del TFG						

Taula 7. *Diagrama de Gantt del treball.*

Cal destacar que la primera etapa d'aprenentatge sobre l'AES es va combinar amb el desenvolupament del codi de la implementació Python3 d'aquest algorisme amb l'objectiu que aquest ajudés a consolidar els coneixements teòrics. Aquesta etapa finalitzà el 1 de març del 2021. La redacció del *Capítol 2* i la recerca i aprenentatge del VHDL van culminar el 22 de març, etapa important i imprescindible del projecte ja que cap d'aquests coneixements es posseïen abans d'iniciar-lo.

El desenvolupament del *topAES*, la implementació hardware que constitueix el cor d'aquest treball, va estar dividida en dues etapes: la primera consisteix a implementar les funcions de baix nivell (és a dir, les operacions de l'AES) exposades als subapartats del 3.2, i la segona, a construir l'estructura de l'alt nivell, de les diferents unitats i del sistema síncron presentats en el 3.3. Tot això culminà el 25 d'abril del 2021.

A partir d'aleshores el desenvolupament del projecte va estar centrat en l'escriptura de la memòria i la recerca complementaria per a constituir-la que encara no s'havia acabat de consolidar. La primera setmana de juny es van tenir els quatre capítols principals enllestits i revisats, es redactaren els complementaris i es va procedir a fer l'atac criptogràfic amb l'objectiu de fer l'avaluació de la robustesa.

Cost del treball.

En aquest treball, es pot desglossar el cost en dos principals components: el cost dels recursos humans i el dels recursos tecnològics emprats. Atesa la situació d'excepcionalitat per la COVID-19, les reunions del projecte s'han dut a terme totes en videotrucades, de tal manera que no hi ha hagut costos d'infraestructures associats.

Cost recursos humans.

Considerant que els salaris d'enginyer júnior ronden els $15\ €/hora$, i els d'un enginyer sènior, uns $50€/hora$, el cost es pot veure calculat a la taula. Per a fer els càlculs s'ha extret, del total d'hores de cada etapa, la columna "*hores sènior*" que representa les hores que l'enginyer sènior hi dedica.

Etapes del projecte	**Hores totals [h]**	**Hores sènior [h]**	**Cost júnior [15€/hora]**	**Cost sènior [50€/hora]**	***Cost de cadascuna de les etapes***
Elecció del tema i registre del TFG	10	1	150	50	200
Recerca teòrica sobre l'AES	25	-	375	0	375
Redacció Capítol 2 de la memòria	35	-	525	0	525
Implementació en Python3 de l'AES	20	2	300	100	400
Recerca i aprenentatge del VHDL	37	-	555	0	555
Disseny operacions de l'AES en VHDL	50	3	750	150	900
Disseny d'alt nivell de l'AES en VHDL	45	3	675	150	825
Redacció Capítol 3 i 4 de la memòria	32	2	480	100	580
Síntesi del topAES i avaluació resultats	6	2	90	100	190
Recerca d'estat de l'art complementària	30	-	450	0	450
Redacció Capítol 1 de la memòria	22	1	330	50	380
Revisió i finalització de la memòria	10	5	150	250	400
Atac al topAES per simulació	20	3	300	150	450
Dipòsit digital de la memòria	1	-	15	0	15
Preparació i defensa del TFG	21	2	315	100	415
TOTAL	364 h	24 h	5460 €	1200 €	***6660*** €

Taula 8*. Cost dels recursos humans segons les etapes del projecte i distingint els costos del enginyer júnior i sènior.*

Cost recursos tecnològics.

Per a fer el càlcul de cada cost de recurs tecnològic s'ha emprar la expressió emmarcada següent:

$$cost[€] = preu[€] \cdot \frac{t_{projecte}}{t_{amortització}}$$

On $t_{projecte}$ és la durada total del projecte, que en aquest cas s'ha estimat com a 5 mesos, i $t_{amortització}$ és el temps d'amortització del producte / recurs tecnològic del qual s'està calculant el cost.

En la *Taula 9* s'ha fet la recopilació de tots els recursos i s'ha calculat el preu de la manera dita. S'ha considerat, com es pot veure, que s'ha obtingut la FPGA per tal de fer la implementació final.

	Preu mercat	***Temps d'amortització***	***Cost total***
Computadora personal	700	5	72,92
Cyclone IV EP4CE30F29C6	100	4	10,42
Entorn hardware per l'FPGA	19	4	3,96
Spyder (entorn Python3)	0	-	0
Llicència *ModelSim – Intel FPGA Edition*	0	-	0
Llicència *Intel Quartus Prime Lite Edition*	0	-	0
Llicència *MATLAB R2020a*	0	-	0
Llicència *OrCad*	0	-	0
TOTAL	-	-	***87,3 €***

Taula 9. *Cost dels recursos tecnològics del treball.*

Per tant, el cost total del projecte s'ha estimat de **6747,3€**.

Estudi d'impacte ambiental.

Aquest treball s'ha desenvolupat íntegrament mitjançant programari i totes les seves verificacions han estat per simulació. D'aquesta manera, l'impacte ambiental ha estat el produït per la computadora sobre la qual s'han fet els diferents càlculs, recerques i resta de tasques que s'han dut a terme amb ella. Segons l'Agència de la Transició Ecològica ADEME, una computadora portàtil emet uns 60 grams de diòxid de carboni per hora de càlcul. Considerant la hipòtesi que un 80% de les hores totals d'aquest projecte corresponen a hores computacionals:

$$364\ h \cdot 0{,}8 \cdot 60\ g\ CO_2/h = 17.472\ g\ CO_2 = 17{,}472\ kg\ CO_2$$

Per a comprendre aquestes unitats, la pròpia agència destaca que 180 kg de diòxid de carboni equivalen a 1000 km en cotxe. En el cas d'aquest treball:

$$17{,}472\ kg\ CO_2 \cdot \frac{1000\ km}{180\ kg\ CO_2} = 97{,}06\ km$$

És a dir, aquest càlcul aproximat de la petjada de carboni seria equivalent a les emissions d'un viatge de 97 km en cotxe. Aquestes són unes dades molt poc significatives, de tal manera que es pot considerar que no hi ha hagut impacte ambiental transcendental en el transcurs d'aquest treball.

L'execució d'aquest treball no ha suposat un risc per a la salut o integritat de les persones que hi ha participat.

CONCLUSIONS.

Aquest Treball de Fi de Grau es concebia amb el **triple objectiu** que s'ha indicat a la introducció d'aquesta memòria, el qual s'ha assolit satisfactòriament. A grans trets, s'ha dut a terme el disseny d'una implementació hardware de l'AES-128 amb una descripció VHDL, previ estudi de la seva funcionalitat i aprenentatge del llenguatge, i s'ha verificat el seu comportament per mitjà d'eines de simulació i síntesi. És més, la seva arquitectura ha estat àmpliament discutida i tractada en una profunda anàlisi de baix i alt nivell. Addicionalment, s'ha executat un *side channel attack* (SCA) per simulació damunt d'un model de consum del sistema dissenyat per tal de conèixer la seva robustesa envers aquesta classe d'atacs.

El Capítol 2 reflecteix la **descripció funcional** de l'AES-128. Gràcies a haver observat en detall el funcionament de l'algorisme i a la implementació software en llenguatge Python3 d'aquest, s'han adquirit els coneixements necessaris de cara al disseny hardware posterior. L'enfoc del capítol ha estat teòric donat que reflecteix un estàndard públic i conegut, de manera que ha representat una etapa imprescindible en el recorregut del treball i ha permès una òptima comprensió dels coneixements associats.

El **disseny hardware de l'AES-128**, anomenat *topAES* per simplicitat al treball, i la subseqüent descripció estructural d'aquest es recull al Capítol 3. La implementació proposada difereix de les dutes a terme per la Laura Casanovas i en Pau Albiol en els seus respectius treballs [1] [2]. En ells, es programava l'AES-128 en software en un microcontrolador definit a priori (un PIC18F4520), quelcom que contrasta amb el realitzat en aquest treball, on s'ha implementat hardware ad hoc sobre una FPGA tal que únicament executa l'algorisme AES per a què s'ha dissenyat. En aquest aspecte, es podria dir que el present treball obra una ruta diferent a aquella que seguien ambdós treballs precedents, més centrada en la perspectiva del hardware *customitzat* o "a mida" que no pas la del software damunt de hardware fix.

La **verificació del *topAES*** s'ha mostrat en el Capítol 4. S'ha detallat el comportament dels senyals interns principals del sistema observant les formes d'ona obtingudes per simulació amb objecte de comprendre com es corresponen amb els aspectes arquitectònics descrits al Capítol 3. En addició, s'ha constatat l'*efecte allau* de l'algorisme, és a dir, el fet que cada valor de sortida depèn de tots i cadascun dels valors inicials, gràcies a una segona simulació. S'ha observat, d'altra banda, que la síntesi és coherent amb el disseny proposat. En conclusió, l'objectiu d'efectuar una implementació hardware ha estat complert en la seva totalitat, ja que la verificació d'aquest ha estat satisfactòria i correcta.

Addicionalment, el Capítol 4 conté un apartat on es tracta **l'atac SCA** que s'ha dut a terme, basat en un *Differential Power Analysis* (DPA) mitjançant el *Hamming weight model* per a l'estimació dels corrents de consum. Atès que ha estat executat per simulació, s'ha proposat i descrit un model basat en busos de transmissió de dades. L'objectiu era de modelitzar el més versemblant possible les línies de la *xarxa d'interconnexions* d'una FPGA, en concret les de sortida de l'operació *subBytes*. S'han exposat les diferents suposicions i hipòtesis preses, els seus fonaments, i s'ha verificat el model en detall.

Els **resultats de l'atac** han constatat que el *Hamming weight model* no estimava suficientment bé els consums de corrent del *topAES*, segons el model de busos plantejat. La raó determinada per la qual això passa té a veure amb el fet que el *Hamming weight model* parteix de la suposició que tots els valors de partida de la línia de dades són 0...0. Això és quelcom bastant freqüent en sistemes de transmissió de dades entre CPU i memòria en microcontroladors, però no en el cas del *topAES*. Donat el seu disseny, s'ha detectat que els valors de partida de la línia de sortida de la *subBytes* són $\emptyset x63...\emptyset x63$ (donats els motius exposats en el cos del treball). Això ha propugnat un replantejament del model d'estimació de consum, i s'ha plantejat una modificació en el *Hamming weight model* per tal de tenir en compte aquesta exigència física de la implementació. Els resultats d'aquest darrer model han estat els esperats i versemblants, ja que s'ha pogut obtenir la clau secreta correcta.

A tall de conclusió general, aquest treball ha reeixit en l'acompliment dels tres objectius de partida. Tanmateix, és destacable el fet que la robustesa de la implementació hardware dissenyada envers un SCA és bastant baixa. Com s'ha repetit al llarg d'aquestes pàgines, si bé l'algorisme de xifrat és irrompible matemàticament (en data de realització d'aquest treball), la seva vulnerabilitat rau en els seus canals laterals d'informació. Es podrien plantejar diferents **contramesures** per tal d'intentar millorar aquests fets, algunes de les quals es deixen indicades de seguit:

- Atès que el *topAES* té una ocupació del 19% d'elements lògics, un bon camí a seguir seria el d'implementar hardware redundant. És a dir, sistemes electrònics addicionals que o bé produeixen el dual de l'operació més vulnerable (la *subBytes*) per tal intentar camuflar el corrent de consum, o bé realitzen operacions inservibles.
- Una possibilitat seria intentar adaptar el treball d'en Pau Albiol i les seves contramesures basades en codis circulars proposades al propi *topAES* plantejat en aquest treball, i comprovar si són efectives en aplicacions *customitzades* o "a mida".

Cal constatar el fet que la robustesa del *topAES* envers el *Hamming weight model*, segons el model plantejat de simulació, sembla ser lleugerament major que en els casos d'implementació en microcontrolador. Això dóna lloc a una possible hipòtesi que es desitja deixar com a **treball futur**. Podria ser que les implementacions hardware "a mida", com el cas de les FPGA, oferissin una major seguretat quant a canals laterals a causa de la seva versatilitat en poder-se "programar". És a dir, almenys en el cas de consum de corrent, el trobat en aquest treball indica que realitzant un disseny hardware que tingués en compte els elements de què es valen els models d'estimació de consums (en el cas del *Hamming weight model*, que els valors inicials són 0...0), es podria crear alguna sort de hardware que, per se i sense necessitar de contramesures, pogués ser significativament més segur que una implementació en microcontrolador fix i definit a priori.

Per finalitzar, caldria parlar sobre l'**experiència personal**. Com bé he advertit en el proemi a aquest treball, el motiu primigeni pel qual vaig decidir d'iniciar-lo ha estat única i exclusivament un sentiment de curiositat exacerbat per un interès incipient en la matèria, el qual ha estat, també, el combustible que ha empès en tot moment l'avenç del treball. He après molts aspectes nous i interessants que no sabia d'entrada, he aprofundit en molts d'altres de què ja tenia coneixença i el meu interès envers l'electrònica no ha fet més que engrandir-se.

I és que no són altres que l'interès i la curiositat, en definitiva, els motors últims del coneixement.

Referències.

[1] L. Casanovas, «Criptonalàsisi sobre Advanced Encryption Standard (AES) mitjançant un atac de canal lateral de corrent de consum i possibles contramesures.» Treball de Final de Grau. Barcelona: Escola Tècnica Superior d'Enginyeria Industrial de Barcelona, Barcelona, 2019.

[2] P. Albiol, «Implementació d'una contramesura basada en codis circulars front a anàlisis de consum diferencial en l'algorisme de xifrat AES.» Treball de Final de Grau. Barcelona: Escola Tècnica Superior d'Enginyeria Industrial de Barcelona, Barcelona, 2020.

[3] Enciclopèdia Catalana, S.A., «Diccionari de la Llengua Catalana» Barcelona, Juliol 1995, ISBN: 84-7739-925-5.

[4] J. F. Dooley, «History of Cryptography and Cryptanalysis: Codes, Ciphers, and Their Algorithms» Springer, 2018, ISBN: 978-3-319-90443-6.

[5] E. C. Reinke, «Classical Cryptography» de *The Classical Journal,* 3 ed., vol. 58, JSTOR, 1962, p. 113–121.

[6] G. J. Simmons, «Vigenère cipher» de *Encyclopedia Britannica*, 9 Agost 2017.

[7] A. Petrucci, «BELLASO, Giovan Battista» Dizionario Biografico degli Italiani - Volume 7 (1970), [En línia]. Available: https://www.treccani.it/enciclopedia/giovan-battista-bellaso_(Dizionario-Biografico). [Últim accés: 31 Maig 2021].

[8] N. P. Smart, «The Enigma Machine» de *Cryptography Made Simple.*, Springer, Cham, 2015, ISBN: 978-3-319-21936-3, pp. 133-161.

[9] Texas Instruments, «The chip that changed the world» 15 Setembre 2020. [En línia]. Available: https://news.ti.com/blog/2020/09/15/the-chip-that-changed-world. [Últim accés: 31 Maig 2021].

[10] J. Kilby, «Invention of the Integrated Circuit» *IEEE Transactions on Electron Devices,* Vol. %1 de %2ED-23, núm. 7, p. 648–654, Juny 1976.

[11] G. E. Moore, « Cramming more components onto integrated circuits» *Electronics Magazine,* vol. 38, núm. 8, 19 Abril 1965.

[12] National Institute of Standards and Technology, «Data Encryption Standard (DES)» 25 Octubre 1999 (ratificat). [En línia]. Available: https://csrc.nist.gov/glossary/term/DES. [Últim accés: 31 Maig 2021].

[13] P. Van De Zande, «The Day DES Died» SANS Institute, 22 Juliol 2001.

[14] C. Paar i J. Pelzl, «Understanding cryptography: a textbook for students and practitioners.» Springer Science & Business Media, 2009.

[15] A. Kerckhoffs von Nieuwenhof, «La cryptographie militaire» *Journal des sciences militaires,* vol. IX, Gener 1883.

[16] C. E. Shannon, «Communication Theory of Secrecy Systems» *Bell System Technical Journal,* núm. 28, 4 Octubre 1949.

[17] F.-X. Standaert, «Introduction to Side-Channel Attacks» de *Secure Integrated Circuits and Systems*, Gener 2010, ISBN: 978-0-387-71827-9, pp. 27-42.

[18] P. P. Chu, «Introduction to Digital System Design» de *RTL HARDWARE DESIGN. Coding for Efficiency, Portability and Scalability.*, John Wiley & Sons, INC., Abril 2006, pp. 1-22.

[19] I. LANKSHEAR, «The Economics of ASICs: At What Point Does a Custom SoC Become Viable?» *Electronic Desing*, 15 Juny 2019.

[20] M. Barr, «How Programmable Logic Works» Barr Group, 1 Juny 1999. [En línia]. Available: https://barrgroup.com/embedded-systems/how-to/programmable-logic. [Últim accés: 3 Juny 2021].

[21] P. P. Chu, «Overview of Hardware Description Languages» de *RTL HARDWARE DESIGN. Coding for Efficiency, Portability and Scalability.*, Cleveland State University, John Wiley & Sons, INC., Abril 2006, pp. 23-41.

[22] O. Gazi, «A Tutorial Introduction to VHDL Programming» Singapore, Springer, 2019, ISBN 978-981-13-2309-6.

[23] National Institute of Standards and Technology, «AES development» 2018. [En línia]. Available: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development. [Últim accés: 05 05 2021].

[24] J. Daemen i V. Rijmen, «AES Proposal: Rijndael» Setembre 1999.

[25] . National Institute of Standards and Technology, «Advanced encryption standard (AES)» Gaithersburg, MD, 2001.

[26] N. Courtois i J. Pieprzyk, «Cryptanalysis of Block Ciphers with Overdefined Systems of Equations» de *Lecture Notes in Computer Science*, Novembre 2002, pp. 392–407. ISBN 978-3-540-67517-4..

[27] S. Murphy i M. Robshaw, «Comments on the Security of the AES and the XSL Technique» Information Security Group, University of London, Setembre 2002.

[28] A. Biryukov i D. Khovratovich, «Related-Key Cryptanalysis of the Full AES-192 and AES-256» de *Advances in Cryptology – ASIACRYPT 2009*, Springer, Berlin, Heidelberg, Mitsuru Matsui, 2009, pp. 1-18. ISBN: 978-3-642-10365-0.

[29] V. Rijmen, «Practical-Titled Attack on AES-128 Using Chosen-Text Relations» Juliol 2010.

[30] A. Bogdanov, D. Khovratovich i C. Rechberger, «Biclique Cryptanalysis of the Full AES», 2011.

[31] Altera Corporation;, «Cyclone IV Device Handbook» Març 2016. [En línia]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf. [Últim accés: 15 Maig 2021].

[32] J. H. Anderson i F. N. Najm, «Interconnect Capacitance Estimation for FPGAs» de *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004*, University of Toronto, IEEE, 2004, pp. 713-718.

[33] R. Rivest, A. Shamir i L. Adleman, «A Method for Obtaining Digital Signatures and Public-key Cryptosystems» Communications of the ACM, 1 Febrer 1978.

ANNEXOS

A.1. Implementació de l'AES en Python3.

A.1.1. Especificació de la classe AES.

class AES (*plaintext, key, tipusPT = 'txt'*)

ATRIBUTS	Nom	Objecte	Descripció
Públics	`estat`	`list`	*Emmagatzema la matriu d'estat de la ronda actual.*
	`key`	`list`	*Emmagatzema la clau inicial en forma de matriu.*
	`roundkey`	`list`	*Emmagatzema la clau de la ronda actual en forma de matriu.*
	`round`	`int`	*Indica, amb un enter del 0 al 10, el nombre de ronda actual.*
Privats	`num_round`	`int`	*Indica el nombre de rondes. Com que es treballa amb claus de 128 bits, el seu valor és 10.*
	`nouestat`	`list`	*Matriu d'estat buida. Serveix per a realitzar les diferents operacions de l'AES al seu damunt, i després tornar-la a buidar.*
MÈTODES	**Nom**		**Descripció**
Públics	`addRoundKey()`		*Realitza l'operació addRoundKey de l'AES*
	`subBytes()`		*Realitza l'operació subBytes de l'AES*
	`shiftRows()`		*Realitza l'operació shiftRows de l'AES*
	`mixColumns()`		*Realitza l'operació mixColumns de l'AES*
	`expandKey()`		*Realitza l'operació expandKey de l'AES*
	`encrypt()`		*Calcula les rondes d'encriptació de l'AES i retorna el ciphertext.*
Interns	`clearnouestat()`		*Es crida internament per fer que l'atribut nouestat torni a ser una matriu buida.*
	`tractar(M, tipusM)`		*Es crida a l'inici per traduir el plaintext a hexadecimal en cas d'entrar-lo en ASCII.*
Estàtics	`Double(b)`		*Executa la multiplicació de Galois per 02 donat un nombre de 8 bits.*
	`subRotWord(M)`		*Executa les operacions RotWord i subBytes internes de la expandKey.*
	`transpose(M)`		*Retorna la matriu d'estat donada transposada.*
	`encodeASCII(col)`		*Tradueix la columna donada component a component del codi ASCII a l'hexadecimal.*

Taula A. 1. *Especificació de la classe AES implementada per a aquest treball*

A.1.2. Codi de la classe AES.

```
import operacionsbinhex as op
from matrius import *


class AES:

    def __init__ (self, M, k, tipusM = 'hex'):

        self.estat     = self.tractar(M, tipusM)
        self.key       = k
        self.roundkey  = k
        self.round     = 0
        self.num_rounds = 10
        self.nouestat = [[],[],[],[]]

#   FUNCIONS DE L'ALGORISME AES

    def addRoundKey(self):

        for i in range(4):
            self.nouestat[i] = op.XORcol(self.estat[i],self.roundkey[i])

        self.estat = self.nouestat
        self.clearnouestat()


    def subBytes (self):

        for i in range(4):
            for j in range(4):
                # S[fila][columna] <- manté l'ordre normal

self.nouestat[i].append(Sbox[int(self.estat[i][j][0],16)][int(self.estat[i][j][1],16)])

        self.estat = self.nouestat
        self.clearnouestat()


    def shiftRows (self):

        for i in range(4):
            self.nouestat[i] = self.estat[i-4][i], self.estat[i-3][i], self.estat[i-
2][i], self.estat[i-1][i];

        self.estat = self.transpose(self.nouestat)
        self.clearnouestat()


    def mixColumns (self):

        for i in range(4):
            for j in range(4):
                r = 0
                Cj = C[j] #<-----------columna de la matriu C
                Mi = self.estat[i]#<---columna actual de la matriu H
                for h in range(4):
                    if Cj[h] == '02':
                        r ^= int(self.Double(Mi[h]),16)
                    elif Cj[h] == '03':
                        r ^= int(op.XORhex(Mi[h], self.Double(Mi[h])),16)
                    elif Cj[h] == '01':
                        r ^= int(Mi[h],16)
                self.nouestat[i].append(op.int2hex(r))
```

```
        self.estat = self.nouestat
        self.clearnouestat()


    def expandKey (self):

        key = self.roundkey

        # obtenció de la primera columna
        col0 = self.subRotWord([key[-1][1], key[-1][2], key[-1][3], key[-1][0]])
        for i in range(4):
            self.nouestat[0].append(op.XORhex(col0[i],key[0][i],Rcon[self.round][i]))

        # obtenció de la resta de columnes
        for i in range(1,4):
          self.nouestat[i] = op.XORcol(self.nouestat[i-1],key[i])

        self.roundkey = self.nouestat
        self.clearnouestat()

#   MÈTODE QUE IMPLEMENTA L'ENCRIPTACIÓ ESTÀNDARD

    def encrypt (self):

        self.addRoundKey()

        for r in range(self.num_rounds):

            if r == self.num_rounds-1:
                self.subBytes()
                self.shiftRows()
                self.expandKey()
                self.addRoundKey()
            else:
                self.subBytes()
                self.shiftRows()
                self.mixColumns()
                self.expandKey()
                self.addRoundKey()
            self.round += 1

#   MÈTODES DE TRACTAMENT DE LA CLASSE

    def clearnouestat (self):
        self.nouestat = [[],[],[],[]]

    def tractar (self, M, tipusM):
        if tipusM == 'hex':
            return M
        elif tipusM == 'txt':
            nouestat = [];
            for col in M: nouestat.append(self.encodeASCII(col))
            return nouestat
        else:
            raise Exception("Format d'entrada no compatible: 'hex' o 'txt'.")

#   MÈTODE ESTÀTIC PRODUCTE DE GALOIS

    @staticmethod
    def Double(b):

        b = op.hex2bin(b)
        carry_set = b[0] == '1'
        b = b[1:] + '0'
        if carry_set: b = op.XORbin(b,op.hex2bin('1b'),n=8)
        return op.bin2hex(b)
```

```
#   MÈTODES ESTÀTICS OPERACIONALS

    @staticmethod
    def subRotWord (M):
        outM = []
        for j in range(len(M)):
            # S[fila][columna] <- manté l'ordre normal
            outM.append(Sbox[int(M[j][0],16)][int(M[j][1],16)])
        return outM

    @staticmethod
    def transpose(M):
        outM = []
        for i in range(4):
            outM.append([])
            for j in range(4):
                outM[i].append(M[j][i])
        return outM

#   MÈTODES ESTÀTICS ASCII

    @staticmethod
    def encodeASCII (col):
        newcol = []
        for l in col:
            newcol.append(hex(ord(l))[2:])
        return newcol
```

A.1.3. Operacions del mòdul operacionsbinhex

```
def standarizebin (b, nbits = 4):
    return b[2:].zfill(nbits)
def standarizehex (h, nbytes = 2):
    return h[2:].zfill(nbytes)

def hex2bin(h, standarize = True):
    '''
    Transforma un nombre hexadecimal de 4*n bits a un nombre binari.
    '''
    n = 4
    if not standarize: n = 1
    return standarizebin(bin(int(h,16)),len(h)*n)

def bin2hex(b, standarize = True):
    '''
    Transforma un nombre binari a un nombre hexadecimal 4*n bits.
    '''
    n = 4
    if not standarize: n = 1
    return standarizehex(hex(int(b,2)),len(b)//n)

def int2hex(i, standarize = True):
    '''
    Transforma un nombre enter a un nombre binari 4*n bits.
    '''
    return standarizehex(hex(i), nbytes = 2)

def XORcol(col1, col2, nbytes = 2):
    '''
    Suma XOR per files de columnes de n nombres hexadecimals. Retorna llista
    '''
    r =  []
    for i in range(len(col1)):
        r.append(XORhex(col1[i],col2[i], n = nbytes))

    return r
```

A.2. Descripció VHDL de la implementació hardware de l'AES.

A.2.1. Entitat AES (topAES).

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity AES is
  port(
    plaintext, key: in state_matrix;
         start, clk: in std_logic;
         ciphertext:out state_matrix;
         done: out std_logic);
end entity;

architecture encryption of AES is

  signal num_rond: std_logic_vector(3 downto 0);
  signal enable, done_id, reset: std_logic;
  signal SM_pt, SM_subBytes, SM_shiftRows, SM_mixColumns, SM_mux, SM_rond: state_matrix;
  signal iniKey, previousKey, roundKey, nextKey, SM_previous: state_matrix;

  component reg_ff is -- Registre paral·lel 128 bits
    port(
      E, CLR, clk: in std_logic; D: in state_matrix; Q: out state_matrix);
  end component;

 begin

    reset <= not start;

        --COMPTADOR (registre comptador de 4 bit)
        round_counter: entity work.counter port map(enable, reset, clk, num_rond);

        --STATE_MACHINE (FSM)
        FSM: entity work.state_machine port map(start, clk, num_rond, enable, done_id);

        --REGISTRES INICIALS (Plaintext i Key)
        inputKey_reg: reg_ff port map('1', reset, clk, key, iniKey);
        plaintext_reg: reg_ff port map('1', reset, clk, plaintext, SM_pt);

        --UNITAT D'EXPANSIÓ DE LA CLAU
        with num_rond select roundKey <=
          iniKey when "0000",
          nextKey when others;
          roundKey_reg: reg_ff port map(enable, reset, clk, roundKey, previousKey);
          eK: entity work.expandKey port map(previousKey, nextKey, num_rond);

        --UNITAT D'ENCRIPTACIÓ
        sB: entity work.subBytes port map(SM_previous, SM_subBytes);
        sR: entity work.shiftRows port map(SM_subBytes, SM_shiftRows);
        mC: entity work.mixColumns port map(SM_shiftRows, SM_mixColumns);
        with num_rond select SM_mux <=
          SM_pt when "0000",
          SM_shiftRows when "1010",
          SM_mixColumns when others;
        aK: entity work.addRoundKey port map(roundKey, SM_mux, SM_rond);
        roundSM_reg: reg_ff port map(enable, reset, clk, SM_rond, SM_previous);

        --REGISTRE FINAL (Ciphertext)
        final_reg: reg_ff port map(done_id, reset, clk, SM_previous, ciphertext);

        done <= done_id;
end;
```

A.2.2. Entitat AES_TB.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity AES_TB is
end;

architecture TB of AES_TB is

  component AES is
    port(
    plaintext, key: in state_matrix;
         start, clk: in std_logic;
         ciphertext:out state_matrix;
         done: out std_logic
    );
  end component;

  signal plaintext, key: state_matrix:= (x"00", x"00", x"00", x"00", x"00", x"00", x"00",
x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00");
  signal start, clk: std_logic;
  signal ciphertext: state_matrix;
  signal done: std_logic;

  begin
  pm: AES port map(plaintext, key, start, clk, ciphertext, done);

  process
   begin
    clk<='0';
    wait for 10 ns;
    clk<='1';
    wait for 10 ns;
         if done='1' then
            clk<='0';
       wait for 10 ns;
       clk<='1';
            wait for 10 ns;
            clk<='0';
       wait for 10 ns;
       clk<='1';
            wait for 10 ns;
       wait;
        end if;
  end process;

  process
    begin
         start<='0';
         wait for 20 ns;
         plaintext <= (x"4f", x"6e", x"65", x"20", x"52", x"69", x"6e", x"67", x"20",
x"54", x"6f", x"20", x"52", x"75", x"6c", x"65");
         key       <= (x"54", x"68", x"61", x"74", x"73", x"20", x"6d", x"79", x"20",
x"4b", x"75", x"6e", x"67", x"20", x"46", x"75");
         wait for 20 ns;
         start <= '1';
         wait;
  end process;
 end;
```

A.2.3. Entitat state_machine.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity state_machine is
  port(
    start, clk: in std_logic;
         num_ronda: in std_logic_vector(3 downto 0);
         enable, done: out std_logic
         );
end;


architecture sm of state_machine is
  signal state, statebuff: state_name;
  -- states: IDLE, ENCR, DELIV

begin

   --State changing function
   process(clk)
   begin
     if rising_edge(clk) then
            if start = '0' then
              statebuff <= IDLE;
            else
              case state is
                  when IDLE =>
                    statebuff <= ENCR;
                  when ENCR =>
                    if num_ronda = "1010" then
                       statebuff <= DELIV;
                    else
                       statebuff <= ENCR;
                    end if;
                  when DELIV =>
                    statebuff <= DELIV;
              end case;
                  end if;
     end if;
    end process;

         -- Registre paral·lel
         state <= statebuff;

         --Outputs selection function
         with state select enable<=
           '1' when ENCR,
           '0' when others;

         with state select done<=
           '1' when DELIV,
           '0' when others;
end;
```

A.2.4. Entitat counter.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port(
    E: in std_logic;     -- enable
         ACLR: in std_logic; -- posada a 0
         clk: in std_logic;
         Q: out std_logic_vector(3 downto 0));
end;

architecture increm_reg of counter is
  signal Qbuff: std_logic_vector(3 downto 0);

begin
  process(clk)
  begin
    if rising_edge(clk) then
         -- Incrementador de 4 bits amb senyals ACLR i E
           if ACLR = '1' then
             Qbuff <= "0000";
           else
             if E='1' then
               Qbuff <= Qbuff + 1;
                 end if;
           end if;
         end if;
  end process;
  Q <= Qbuff; -- Registre paral·lel 4 biestables
end;
```

A.2.5. Entitat reg_ff.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity reg_ff is -- Registre paral·lel de 128 flipflops D amb senyal E i CLR
  port(
    E, CLR, clk: in std_logic; -- E: Senyal Enable. CLR: Senyal Reset.
         D: in state_matrix;
         Q: out state_matrix);
end;

architecture flipflop128 of reg_ff is
  signal Qbuff: state_matrix;
begin
  process(clk)
  begin
    if rising_edge(clk) then
           if CLR = '1' then
             Qbuff <= (x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00", x"00",
x"00", x"00", x"00", x"00", x"00", x"00", x"00");
           else
             if E='1' then
               Qbuff <= D;
                 end if;
           end if;
         end if;
  end process;
  Q <= Qbuff;
end;
```

A.2.6. Entitat subBytes.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity subBytes is
  port(
        in_STATE_MATRIX:in state_matrix;
        out_STATE_MATRIX:out state_matrix
    );
end entity;

architecture arch of subBytes is

begin
  gen: for indx in 0 to 15 generate
        pm: entity work.Sbox port map(
          in_byte => in_STATE_MATRIX(indx),
          out_byte=> out_STATE_MATRIX(indx)
          );
  end generate;
end;
```

A.2.7. Entitat shiftRows.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity shiftRows is
  port(
        in_STATE_MATRIX:in state_matrix;
        out_STATE_MATRIX:out state_matrix
    );
end entity;

architecture sR of shiftRows is

begin
  out_STATE_MATRIX(15)<= in_STATE_MATRIX(15);
  out_STATE_MATRIX(14)<= in_STATE_MATRIX(10);
  out_STATE_MATRIX(13)<= in_STATE_MATRIX(5);
  out_STATE_MATRIX(12)<= in_STATE_MATRIX(0);

  out_STATE_MATRIX(11)<= in_STATE_MATRIX(11);
  out_STATE_MATRIX(10)<= in_STATE_MATRIX(6);
  out_STATE_MATRIX(9) <= in_STATE_MATRIX(1);
  out_STATE_MATRIX(8) <= in_STATE_MATRIX(12);

  out_STATE_MATRIX(7) <= in_STATE_MATRIX(7);
  out_STATE_MATRIX(6) <= in_STATE_MATRIX(2);
  out_STATE_MATRIX(5) <= in_STATE_MATRIX(13);
  out_STATE_MATRIX(4) <= in_STATE_MATRIX(8);

  out_STATE_MATRIX(3) <= in_STATE_MATRIX(3);
  out_STATE_MATRIX(2) <= in_STATE_MATRIX(14);
  out_STATE_MATRIX(1) <= in_STATE_MATRIX(9);
  out_STATE_MATRIX(0) <= in_STATE_MATRIX(4);

end;
```

A.2.8. Entitat mixColumns.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity mixColumns is
  port(
        in_STATE_MATRIX:in state_matrix;
        out_STATE_MATRIX:out state_matrix);
end entity;


architecture mC of mixColumns is
  signal doubled: state_matrix;
begin

  doubl: for j in 15 downto 0 generate
    pm: entity work.galoisDouble port map(     -- Calculo el resultat de galoisDouble de
          in_byte => in_STATE_MATRIX(j),          TOTS els bytes de la matriu d'estat.
          out_byte=> doubled(j)
          );
  end generate;

  out_STATE_MATRIX(15-4*i) <= doubled(15-4*i) xor doubled(14-4*i) xor
                              in_STATE_MATRIX(14-4*i) xor in_STATE_MATRIX(13-4*i)
                              xor in_STATE_MATRIX(12-4*i);

  out_STATE_MATRIX(14-4*i) <= in_STATE_MATRIX(15-4*i) xor doubled(14-4*i) xor
                              doubled(13-4*i) xor in_STATE_MATRIX(13-4*i)
                              xor in_STATE_MATRIX(12-4*i);

  out_STATE_MATRIX(13-4*i) <= in_STATE_MATRIX(15-4*i) xor in_STATE_MATRIX(14-4*i)
                              xor doubled(13-4*i) xor doubled(12-4*i)
                              xor in_STATE_MATRIX(12-4*i);

  out_STATE_MATRIX(12-4*i) <= doubled(15-4*i) xor in_STATE_MATRIX(15-4*i) xor
                              in_STATE_MATRIX(14-4*i) xor in_STATE_MATRIX(13-4*i)
                              xor doubled(12-4*i);
  end generate;
  end;
```

A.2.9. Entitat galoisDouble.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity galoisDouble is
  port( in_byte:in std_logic_vector (7 downto 0);
        out_byte:out std_logic_vector (7 downto 0));
end entity;

architecture duplicar of galoisDouble is

begin                                          -- x"1b":
  out_byte(7) <= in_byte(6);                   -- 0
  out_byte(6) <= in_byte(5);                   -- 0
  out_byte(5) <= in_byte(4);                   -- 0
  out_byte(4) <= in_byte(3) xor in_byte(7);    -- 1
  out_byte(3) <= in_byte(2) xor in_byte(7);    -- 1
  out_byte(2) <= in_byte(1);                   -- 0
  out_byte(1) <= in_byte(0) xor in_byte(7);    -- 1
  out_byte(0) <= in_byte(7);                   -- 1
end;
```

A.2.10. Entitat expandKey.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity expandKey is
  port(
    in_ROUND_KEY: in state_matrix;
         out_ROUND_KEY:out state_matrix;
         num_rond: in std_logic_vector(3 downto 0)
    );
end entity;

architecture eK of expandKey is
  signal rot_in_W4, sub_in_W4: byte_word;
  signal Rcon: byte_word;
  signal out_W1, out_W2, out_W3, out_W4:byte_word;
begin
  -- Selecció de la Rcon de la ronda

  with num_rond select
    Rcon<=
         (x"01",x"00",x"00",x"00") when "0001",
         (x"02",x"00",x"00",x"00") when "0010",
         (x"04",x"00",x"00",x"00") when "0011",
         (x"08",x"00",x"00",x"00") when "0100",
         (x"10",x"00",x"00",x"00") when "0101",
         (x"20",x"00",x"00",x"00") when "0110",
         (x"40",x"00",x"00",x"00") when "0111",
         (x"80",x"00",x"00",x"00") when "1000",
         (x"1b",x"00",x"00",x"00") when "1001",
         (x"36",x"00",x"00",x"00") when "1010",
         (x"00",x"00",x"00",x"00") when others;

  -- Operació RotWord()

  rot_in_W4 <= (in_ROUND_KEY(2), in_ROUND_KEY(1), in_ROUND_KEY(0), in_ROUND_KEY (3));

  --Operació SubBytes() i generació de les columnes d'out_ROUND_KEY

  substitucio: for byte in 3 downto 0 generate
    pm: entity work.Sbox port map(
           in_byte => rot_in_W4(byte),
           out_byte=> sub_in_W4(byte));

         out_W1(byte) <= sub_in_W4(byte) xor in_ROUND_KEY(12+byte) xor Rcon(byte); --
Generació primera columna

         out_W2(byte) <= out_W1(byte) xor in_ROUND_KEY (8+byte);
         out_W3(byte) <= out_W2(byte) xor in_ROUND_KEY (4+byte); -- Generació resta de
columnes
         out_W4(byte) <= out_W3(byte) xor in_ROUND_KEY (byte);
  end generate;

out_ROUND_KEY <= (out_W1(3), out_W1(2), out_W1(1), out_W1(0), out_W2(3), out_W2(2),
out_W2(1), out_W2(0), out_W3(3), out_W3(2), out_W3(1), out_W3(0), out_W4(3), out_W4(2),
out_W4(1), out_W4(0));

end;
```

A.2.11. Entitat addRoundKey.

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity addRoundKey is
  port(
    in_ROUND_KEY: in state_matrix;
         in_STATE_MATRIX:in state_matrix;
         out_STATE_MATRIX:out state_matrix
    );
end entity;

architecture aRK of addRoundKey is
begin
  gen: for indx in 15 downto 0 generate
     out_STATE_MATRIX(indx) <= in_ROUND_KEY(indx) xor in_STATE_MATRIX(indx);
  end generate;
end;
```

A.2.12. Entitat Sbox.

Nota: per no ocupar moltes planes, s'ha decidit posar la sentencia when select en tres columnes.

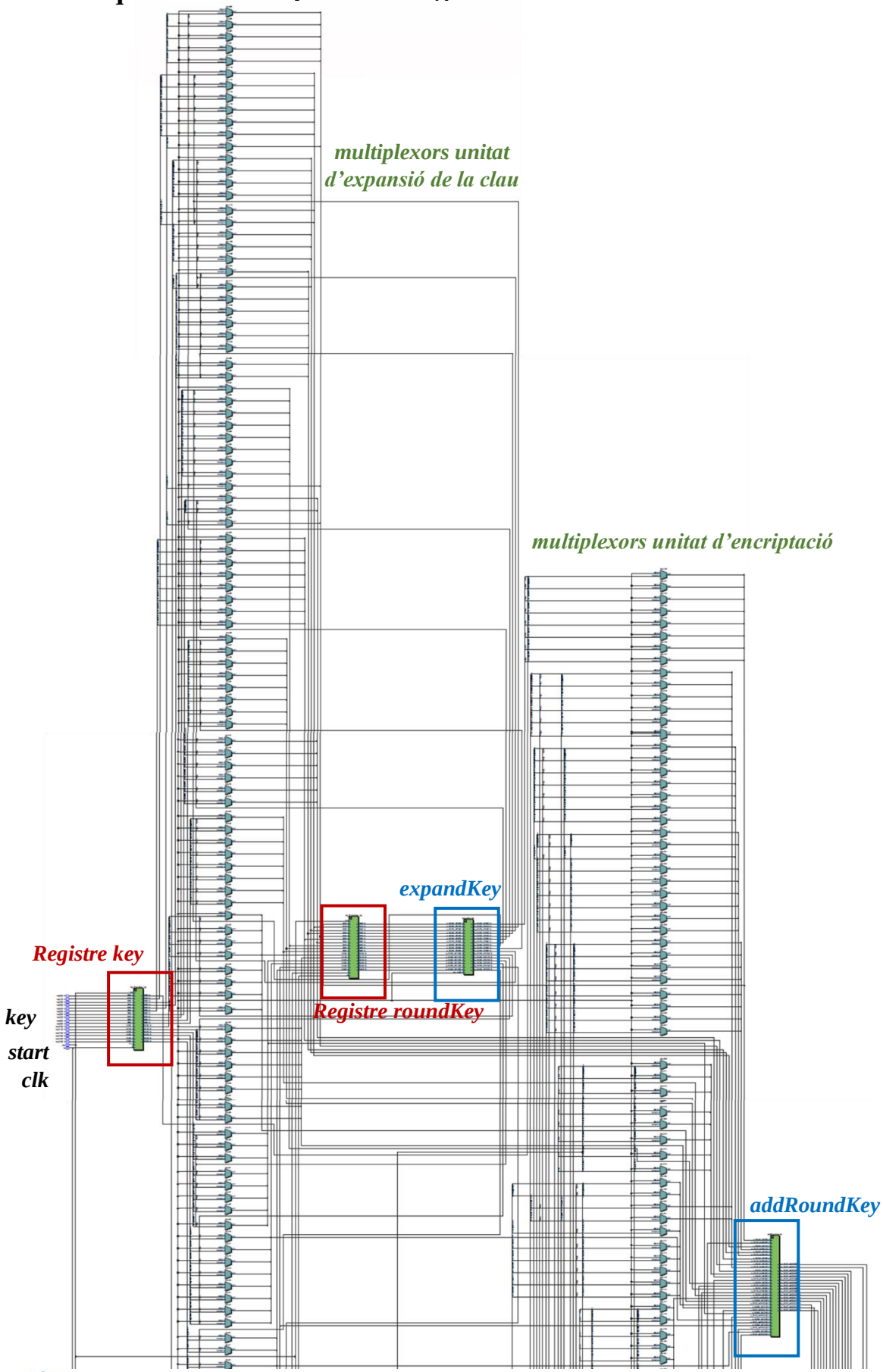
```
library ieee;
use ieee.std_logic_1164.all;
use work.my_data_type.all;

entity Sbox is
  port(
    in_byte:in std_logic_vector(7 downto 0);
         out_byte:out std_logic_vector(7 downto 0)
         );
end;

architecture arch of Sbox is

begin

  with in_byte select
         out_byte <=
         x"63" when x"00",
         x"7c" when x"01",
         x"77" when x"02",
         x"7b" when x"03",
         x"f2" when x"04",
         x"6b" when x"05",
         x"6f" when x"06",
         x"c5" when x"07",
         x"30" when x"08",
         x"01" when x"09",
         x"67" when x"0a",
         x"2b" when x"0b",
         x"fe" when x"0c",
         x"d7" when x"0d",
         x"ab" when x"0e",
         x"76" when x"0f",
         x"ca" when x"10",
         x"82" when x"11",
         x"c9" when x"12",
         x"7d" when x"13",
         x"fa" when x"14",
         x"59" when x"15",
         x"47" when x"16",
         x"f0" when x"17",
         x"ad" when x"18",
         x"d4" when x"19",
         x"a2" when x"1a",
         x"af" when x"1b",
         x"9c" when x"1c",
         x"a4" when x"1d",
         x"72" when x"1e",
         x"c0" when x"1f",
         x"b7" when x"20",
         x"fd" when x"21",
         x"93" when x"22",
         x"26" when x"23",
         x"36" when x"24",
         x"3f" when x"25",
         x"f7" when x"26",
         x"cc" when x"27",
         x"34" when x"28",
         x"a5" when x"29",
         x"e5" when x"2a",
         x"f1" when x"2b",
         x"71" when x"2c",
         x"d8" when x"2d",
         x"31" when x"2e",
         x"15" when x"2f",
         x"04" when x"30",
         x"c7" when x"31",
         x"23" when x"32",
         x"c3" when x"33",
         x"18" when x"34",
         x"96" when x"35",
         x"05" when x"36",
         x"9a" when x"37",
         x"07" when x"38",
         x"12" when x"39",
         x"80" when x"3a",
         x"e2" when x"3b",
         x"eb" when x"3c",
         x"27" when x"3d",
         x"b2" when x"3e",
         x"75" when x"3f",
         x"09" when x"40",
         x"83" when x"41",
         x"2c" when x"42",
         x"1a" when x"43",
         x"1b" when x"44",
         x"6e" when x"45",
         x"5a" when x"46",
         x"a0" when x"47",
         x"52" when x"48",
         x"3b" when x"49",
         x"d6" when x"4a",
         x"b3" when x"4b",
         x"29" when x"4c",
         x"e3" when x"4d",
         x"2f" when x"4e",
         x"84" when x"4f",
         x"53" when x"50",
         x"d1" when x"51",
         x"00" when x"52",
         x"ed" when x"53",
         x"20" when x"54",
         x"fc" when x"55",
```

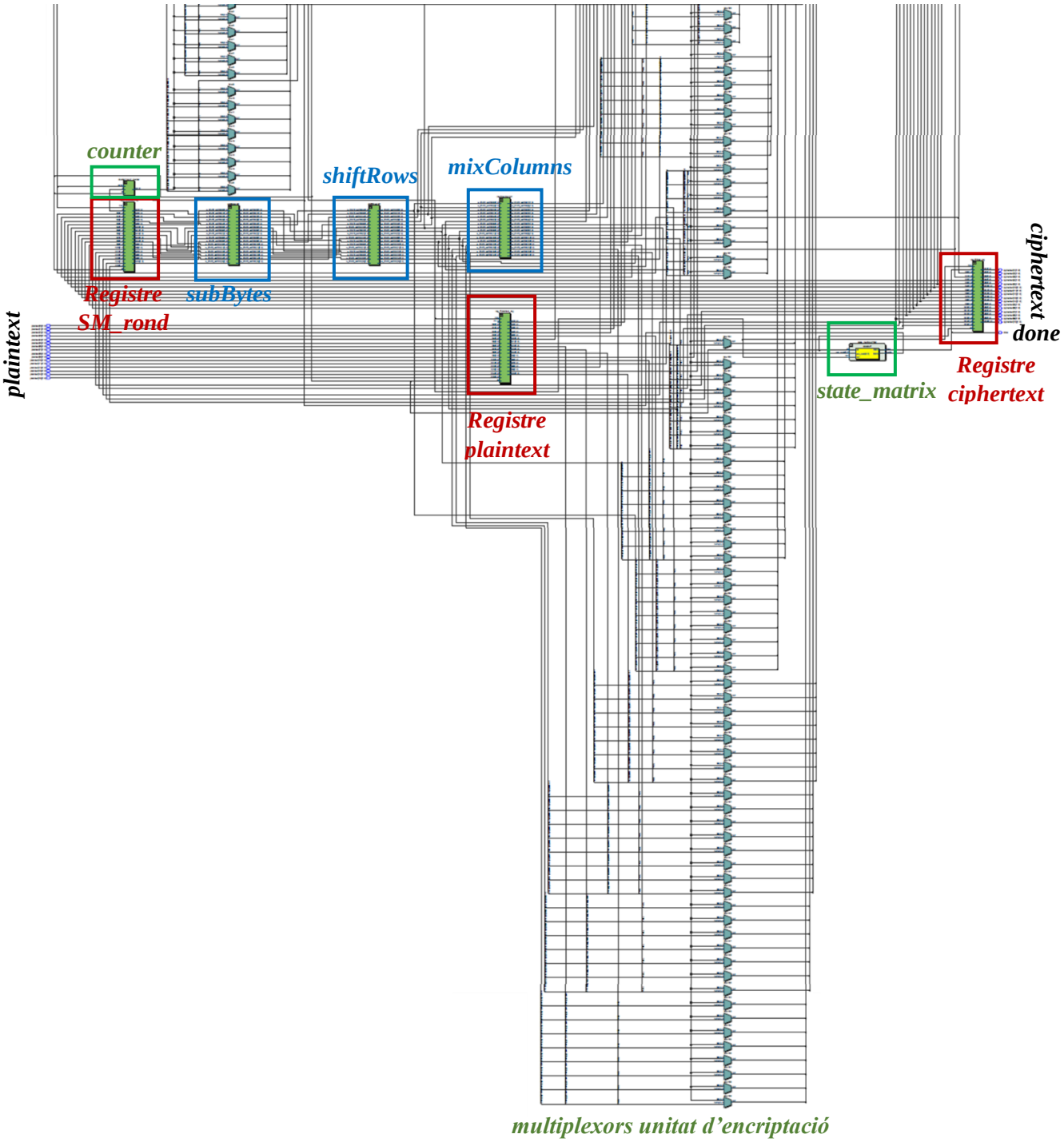
```
        x"b1" when x"56",
        x"5b" when x"57",
        x"6a" when x"58",
        x"cb" when x"59",
        x"be" when x"5a",
        x"39" when x"5b",
        x"4a" when x"5c",
        x"4c" when x"5d",
        x"58" when x"5e",
        x"cf" when x"5f",
        x"d0" when x"60",
        x"ef" when x"61",
        x"aa" when x"62",
        x"fb" when x"63",
        x"43" when x"64",
        x"4d" when x"65",
        x"33" when x"66",
        x"85" when x"67",
        x"45" when x"68",
        x"f9" when x"69",
        x"02" when x"6a",
        x"7f" when x"6b",
        x"50" when x"6c",
        x"3c" when x"6d",
        x"9f" when x"6e",
        x"a8" when x"6f",
        x"51" when x"70",
        x"a3" when x"71",
        x"40" when x"72",
        x"8f" when x"73",
        x"92" when x"74",
        x"9d" when x"75",
        x"38" when x"76",
        x"f5" when x"77",
        x"bc" when x"78",
        x"b6" when x"79",
        x"da" when x"7a",
        x"21" when x"7b",
        x"10" when x"7c",
        x"ff" when x"7d",
        x"f3" when x"7e",
        x"d2" when x"7f",
        x"cd" when x"80",
        x"0c" when x"81",
        x"13" when x"82",
        x"ec" when x"83",
        x"5f" when x"84",
        x"97" when x"85",
        x"44" when x"86",
        x"17" when x"87",
        x"c4" when x"88",
        x"a7" when x"89",
        x"7e" when x"8a",
        x"3d" when x"8b",
        x"64" when x"8c",
        x"5d" when x"8d",
        x"19" when x"8e",
        x"73" when x"8f",
        x"60" when x"90",
        x"81" when x"91",
        x"4f" when x"92",
        x"dc" when x"93",
        x"22" when x"94",
        x"2a" when x"95",
        x"90" when x"96",
        x"88" when x"97",
        x"46" when x"98",
        x"ee" when x"99",
        x"b8" when x"9a",
        x"14" when x"9b",
        x"de" when x"9c",
        x"5e" when x"9d",
        x"0b" when x"9e",
        x"db" when x"9f",
        x"e0" when x"a0",
        x"32" when x"a1",
        x"3a" when x"a2",
        x"0a" when x"a3",
        x"49" when x"a4",
        x"06" when x"a5",
        x"24" when x"a6",
        x"5c" when x"a7",
        x"c2" when x"a8",
        x"d3" when x"a9",
        x"ac" when x"aa",
        x"62" when x"ab",
        x"91" when x"ac",
        x"95" when x"ad",
        x"e4" when x"ae",
        x"79" when x"af",
        x"e7" when x"b0",
        x"c8" when x"b1",
        x"37" when x"b2",
        x"6d" when x"b3",
        x"8d" when x"b4",
        x"d5" when x"b5",
        x"4e" when x"b6",
        x"a9" when x"b7",
        x"6c" when x"b8",
        x"56" when x"b9",
        x"f4" when x"ba",
        x"ea" when x"bb",
        x"65" when x"bc",
        x"7a" when x"bd",
        x"ae" when x"be",
        x"08" when x"bf",
        x"ba" when x"c0",
        x"78" when x"c1",
        x"25" when x"c2",
        x"2e" when x"c3",
        x"1c" when x"c4",
        x"a6" when x"c5",
        x"b4" when x"c6",
        x"c6" when x"c7",
        x"e8" when x"c8",
        x"dd" when x"c9",
        x"74" when x"ca",
        x"1f" when x"cb",
        x"4b" when x"cc",
        x"bd" when x"cd",
        x"8b" when x"ce",
        x"8a" when x"cf",
        x"70" when x"d0",
        x"3e" when x"d1",
        x"b5" when x"d2",
        x"66" when x"d3",
        x"48" when x"d4",
        x"03" when x"d5",
        x"f6" when x"d6",
        x"0e" when x"d7",
        x"61" when x"d8",
        x"35" when x"d9",
        x"57" when x"da",
        x"b9" when x"db",
        x"86" when x"dc",
        x"c1" when x"dd",
        x"1d" when x"de",
        x"9e" when x"df",
        x"e1" when x"e0",
        x"f8" when x"e1",
        x"98" when x"e2",
        x"11" when x"e3",
        x"69" when x"e4",
        x"d9" when x"e5",
        x"8e" when x"e6",
        x"94" when x"e7",
        x"9b" when x"e8",
        x"1e" when x"e9",
        x"87" when x"ea",
        x"e9" when x"eb",
        x"ce" when x"ec",
        x"55" when x"ed",
        x"28" when x"ee",
        x"df" when x"ef",
        x"8c" when x"f0",
        x"a1" when x"f1",
        x"89" when x"f2",
        x"0d" when x"f3",
        x"bf" when x"f4",
        x"e6" when x"f5",
        x"42" when x"f6",
        x"68" when x"f7",
        x"41" when x"f8",
        x"99" when x"f9",
        x"2d" when x"fa",
        x"0f" when x"fb",
        x"b0" when x"fc",
        x"54" when x"fd",
        x"bb" when x"fe",
        x"16" when x"ff",
        "XXXXXXXX" when
    others;
    end;
```

A.2.13. Package customitzat my_data_type.

```
library ieee;
use ieee.std_logic_1164.all;

package my_data_type is

  type byte_word is array(3 downto 0) of std_logic_vector(7 downto 0);
  type state_matrix is array(15 downto 0) of std_logic_vector(7 downto 0);
  type state_name is (IDLE, ENCR, DELIV);

end package;
```

A.3. Esquemàtic del topAES obtingut en la síntesi.




counter
shiftRows
mixColumns
ciphertext
done
Registre
SM_rond
subBytes
plaintext
Registre
plaintext
state_matrix
Registre
ciphertext
multiplexors unitat d'encriptació

A.4. Codis emprats per a fer l'atac SCA al topAES.

A.4.1. Codi Matlab que implementa l'atac.

```
%% ----- DADES CONSUM (OrCad) I SORTIDA DE LA subBytes -------
dataOrCad = table2array(readtable ('dades\Irise_to_1.csv','NumHeaderLines',2));
dataIdown = table2array(readtable ('dades\Idown_to_0.csv','NumHeaderLines',2));
databytes = table2array(readtable('dades\ciphertext.csv')); % sortida subBytes

Irise1 = dataOrCad(:,2); % consum bus de dades en pujar de 0V a 1.5V
Idown0 = dataIdown(:,2); % consum bus de dades en baixar de 1.5V a 0V

%% ---------------- CÀLCUL MATRIU CONSUMS ----------------
MatriuConsums = [];

for nbyte = 1:16
    bytes = databytes(:,nbyte); % bytes actuals
    [m,n] = size(bytes);

    Iconsum = [];
    maxIconsum = [];
    for i=1:m
        Iconsum = consum(bytes(i,:),Irise1,Idown0)'; % consum equivalent
        Iconsum = Iconsum + 100e-6 * randn(size(Iconsum)); %afegim soroll gaussià
        maxIconsum = [maxIconsum; max(Iconsum,[],'all')]; % ens interessa el màxim
    end
   MatriuConsums = [MatriuConsums, maxIconsum];
end

%% ----------------OBTENCIÓ MATRIU CORRELACIONS HW model ----------------
[m,n] = size(MatriuConsums);
rhosMatrix = [];

for b=1:16
    filename = ['HW\matriuHW', int2str(b-1), '.csv'];
    HWbyte = table2array(readtable (filename));

    byte = MatriuConsums(:,b);
    rhos = [];
    for key = 1:256
        MatrCorr = corrcoef(HWbyte(:,key), byte);
        rhos = [rhos, abs(MatrCorr(2,1))];
    end
    rhosMatrix = [rhosMatrix; rhos];
end
writematrix(rhosMatrix, 'resultats\rho_63.xlsx')
%% ----- OBTENCIÓ MATRIU CORRELACIONS HW model modificat --------
[m,n] = size(MatriuConsums);
rhosMatrix_mod = [];

for b=1:16
    filename = ['HW_modificat\matriuHW', int2str(b-1), '.csv'];
    HWbyte = table2array(readtable (filename));

    byte = MatriuConsums(:,b);
    rhos = [];
    for key = 1:256
        MatrCorr = corrcoef(HWbyte(:,key), byte);
        rhos = [rhos, abs(MatrCorr(2,1))];
    end
    rhosMatrix_mod = [rhosMatrix_mod; rhos];
end
writematrix(rhosMatrix_mod, 'resultats\rho_63_mod.xlsx')
```

A.4.2. Codi Python3 de l'AES simplificat per l'atac.

```
import operacionsbinhex as op
from matrius import Sbox

def AES_1byte (Bptxt, Bkey):

    ## addRoundKey
    addRoundKey = op.XORhex(Bptxt,Bkey)

    ## subBytes
    subBytes = Sbox[int(addRoundKey[0],16)][int(addRoundKey[1],16)]

    return subBytes
```

A.4.3. Codi Python3 que calcula les MCE pel model HW.

```
import operacionsbinhex as op
from matrius import pyH ##matriu que conté els 256 possibles bytes.
import random
import csv
from AES_1byte import *

fptxt = open('dades\plaintext.csv', 'r') ##plaintext generats aleatòriament.

reader = csv.reader(fptxt, delimiter= ',')

plaintext = []
for ptxt in reader:
    newptxt = []
        for b in ptxt:
            newptxt.append(b[2:])
        plaintext.append(newptxt)
fptxt.close()

##Matrius per cada byte

for nbyte in range(16):
    MatriuHW = []
    for ptxt in plaintext:
        keyHW = []

         for key in pyH:
            keyHW.append(op.hex2bin(AES_1byte(ptxt[nbyte], key)).count('1'))
        MatriuHW.append(keyHW)

    filename = 'HW_modificat\matriuHW' + str(nbyte)+'.csv'
    fHW = open(filename, 'w')
    W = csv.writer(fHW)

    for line in MatriuHW:
        W.writerow(line)
    fHW.close()
```

A.4.4. Codi Python3 que calcula les MCE pel model HW modificat.

Tan sols cal canviar la següent part del codi anterior:

```
        for key in pyH:
            byteout = op.hex2bin(AES_1byte(ptxt[nbyte], key))
            keyHW.append((byteout[0]+byteout[3:6]).count('1'))
        MatriuHW.append(keyHW)
```