2015•2016
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef

A random permutation-based method for secure hardware implementations

Promotor :
Prof. dr. ir. Nele MENTENS
Dhr. BOHAN YANG

Robin Schrijvers
Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

2015•2016

Faculteit Industriële ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterproef

A random permutation-based method for secure hardware implementations

Promotor :
Prof. dr. ir. Nele MENTENS
Dhr. BOHAN YANG

Robin Schrijvers

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

universiteit hasselt | KU LEUVEN

Woord vooraf

U bekijkt op dit moment de scriptie "Een random permutatie gebaseerde methode voor veilige hardware implementaties". Dit is geschreven in het kader van mijn opleiding Industriële Ingenieurswetenschappen, met specialisatie Elektronica-ICT in Diepenbeek. Vanaf de maand september 2015 tot juni 2016 heb ik dit onderzoek en het schrijven van deze thesis voortgezet.

Aangezien dat cryptografie en beveiligingen tegen hackers in deze tijden een veelbesproken onderwerp is bij onderzoekers, en soms zelfs in de media, was ik al snel zeer enthousiast om met dit onderzoek van start te gaan. Op verschillende momenten was ik onder de indruk van de complexiteit, maar na uitvoerig onderzoek en vastberadenheid heb ik dit onderzoek kunnen afsluiten door een implementatie tegen hackers te maken. Tijdens mijn onderzoek stond mijn stagebegeleider, Bohan Yang, en mijn begeleidster vanuit de opleiding, prof. dr. ir. Nele Mentens, altijd paraat om mijn vragen te beantwoorden.

Hierdoor wil ik beide zeer hartelijk bedanken voor de tijd die ze hebben vrijgemaakt en hun fijne ondersteuning tijdens dit traject. Verder bedank ik ook alle contactpersonen die klaar stonden voor mijn vragen. Zonder alle hulp kon ik dit onderzoek niet voltooien.

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Abstract

De onderzoeksgroep COSIC in het Departement ESAT van KU Leuven onderzoekt o.a. methoden om nevenkanaalaanvallen tegen te gaan in ASIC-gebaseerde systemen. Eén van deze methoden is een random permutatie om de gemeten data in vermogenanalyses te randomiseren. Om dat te realiseren onderzoekt deze masterproef enerzijds een methode om een 4x4 *S-BOX* in een *ASIC* te implementeren, om die daarna te pijplijnen en een tussenresultaat toe te voegen dat volledig random is. Dat tussenresultaat ontstaat door een random-permutatie algoritme. Anderzijds optimaliseert deze thesis deze implementatie voor een *S-BOX* uit het *PRESENT*-algoritme via getransformeerde standaardcellen.

De 4x4 *S-BOX* is gebaseerd op *Look-Up Tables* (*LUT*), waarbij de functies van de *LUT's* configureerbaar zijn. De permutatiegeneratie en zijn inverse bevatten een algoritme dat zorgt voor de ordening van random getallen. Een optimalisatie hiervan is vervolgens toegepast op een willekeurig deelblok van de *PRESENT S-BOX* netlist, waarbij de grenzen van het deelblok dienen als input en output voor de permutatiegeneratie.

De resulterende ontwerpmethode werd geëvalueerd a.d.h.v. de oppervlakte en de snelheid van de beveiligde bouwblokken in vergelijking met de originele bouwblokken. De conclusie was dat deze 4x4 *S-BOX* implementatie enkel een kost heeft in snelheid in de initialisatiefase van de permutatie generatie en dat de oppervlakte 4063,7µm² bedraagt. De P*RESENT S-BOX* implementatie neemt een oppervlakte van 7909,2 µm² in en heeft een maximale klokfrequentie van 1,72GHz.

Abstract (English)

The research group COSIC in the Department ESAT of KU Leuven is researching different methods to counter side-channel attacks in ASIC-based systems.

One of these methods is a random permutation to randomise the measured data in a power consumption measurement in order to decorrelate it from the secret information that is processed inside the chip. To achieve this, this master thesis researches a method to implement a 4x4 S-BOX in ASIC after which this can be pipelined to get a random intermediate result. This result is generated by a random permutation algorithm. Additionally, this thesis optimizes the implementation of a S-BOX of the PRESENT algorithm via transformed standard cells. The 4x4 S-BOX is based on Look-Up Tables (LUTs) with configurable functionality. The permutation generation and its inverse contain an algorithm which provides a sequence of random numbers. This is further optimized and applied to a random segment of the PRESENT S-BOX netlist. The borders of this segment act as the input and output of the permutation generation.

The resulting design method is evaluated based on the area and speed of the secured blocks in comparison with the original blocks. The conclusion was that the 4x4 S-BOX has a cost in speed in the initialisation phase of the permutation generation while the silicon area is 4063.7μm^2. The implementation of the PRESENT S-BOX' permutation generation has an area of 7909.2 μm^2 and has a maximum clock frequency of 1.72GHz.

1 Inleiding

1.1 Situering

In moderne *ASIC's* en *FPGA's* zijn er vaak encryptie-algoritmes terug te vinden om gegevens te beschermen tegen derden. De reden hiertoe is vaak om de privacy te bewaren van een bepaald persoon of bedrijf. Encryptie-algoritmes komen voor met private sleutels (symmetrische cryptografie) of met een publieke en private sleutel (asymmetrische cryptografie). Naast enkel XOR-operaties om de gegevens te vercijferen met de sleutels zijn er algoritmes die verder ook gebruik maken van grotere bewerkingen in bepaalde *rounds* in het algoritme. Enkele voorbeelden van deze bewerkingen zijn:

- Mix-columns operatie (AES): gegevens komen voor in een matrix waarin bewerkingen worden uitgevoerd op de kolommen [1].
- Sub bytes operatie (AES, PRESENT): Een waarde (n bits) wordt vervangen door een andere waarde in een of *Look-Up Table* (*LUT*, ook *S-BOX* genoemd) [2].

De werking van de bewerkingen en *LUT*-configuraties zijn publiekelijk terug te vinden, omdat het eerder niet van belang is wat er precies gebeurt, maar eerder mét wat er iets gebeurd. Hierdoor zijn het de sleutels per gebruiker die geheim blijven.

1.2 Probleemstelling en doelstellingen

Toch zijn er personen, omschreven als *hackers*, die alsnog methoden proberen te vinden om geheime sleutels en de bewerkingen erop te ontcijferen. De methode waarvoor deze thesis een oplossing wilt vinden heet een nevenkanaalaanval (*Side Channel Attack*, *SCA*). De definitie hiervan samen met enkele voorbeelden staat verder uitgelegd in hoofdstuk 2. Eén van de plaatsen om een SCA op uit te voeren zijn *S-BOX'en*, waardoor het doel van deze thesis zich bijgevolg richt op het verhelpen van nevenkanaalaanvallen in *SBOX'en* in ASIC-systemen.

1.3 Gevolgde methoden

Om nevenkanaalaanvallen te verhelpen in *S-BOX'en* ging er eerst onderzoek naar de mogelijkheid om het idee van *LUT's* in *FPGA's* te kopiëren naar *ASIC's*. De uitwerking hiervan met het uiteindelijke schema van de implementatie voor een 4x4 S-BOX staat uitgelegd in hoofdstuk 3. De tweede stap is een methode voorzien om deze *LUT's* configureerbaar te maken zodat de gemeten data van een *SCA* niet meer gecorreleerd zijn met de geheime informatie in de chip. Deze methode is gebaseerd op willekeurige permutatie generatie van een getallenrij, en staat verder uitgelegd in hoofdstuk 4.

Verder wordt de uitgewerkte implementatie van de willekeurige permutatie toegepast op een *S-BOX* van het *PRESENT*-algoritme via getransformeerde standaardcellen. Hierbij volgt ook de evaluatie a.d.h.v. snelheid en oppervlakte van de beveiligde bouwblokken in vergelijking met de originele bouwblokken. Dit gebeurt in hoofdstuk 5. Ten slotte volgt er een conclusie van alle realisaties gevolgd door voorstellen voor verder onderzoek.

2 Nevenkanaalaanvallen

In deze thesis is het fenomeen van een nevenkanalen hetgeen waar het om draait. De thesis onderzoekt een methode om nevenkanaalaanvallen tegen te gaan in ASIC- gebaseerde systemen; denk bijvoorbeeld aan bankkaarten, waarbij vertrouwelijkheid en privacy belangrijk zijn [3]. Om dit verder te kunnen volgen staat in dit hoofdstuk uitgelegd wat nevenkanaalaanvallen precies zijn en in welke vormen ze kunnen voorkomen.

2.1 Definitie

Naast de informatie die beschikbaar is via het in- en output gedrag van de chip, proberen nevenkanaalaanvallen extra informatie uit de chip halen. Wanneer er genoeg kennis is over de inwendige bouwblokken van het systeem, kunnen vele testmonsters geheime informatie vertellen over het cryptosysteem zoals private sleutels, of configuraties van bepaalde bewerkingen in algoritmes. Dit volgt uit de sterke correlatie tussen de fysieke implementatie en de patronen van de verwerkte gegevens in bepaalde algoritmes [4, 5].

2.2 Vormen

Nevenkanaalaanvallen komen voor in verschillende vormen. Terwijl de één simpeler uit te voeren is zal de ander veel effectiever zijn. Hoe dan ook moet de aanvaller over de nodige kennis over de implementatie beschikken voordat hij zijn aanval kan uitvoeren. In dit deel staan verschillende vormen van nevenkanaalaanvallen opgelijst.

2.2.1 Tijdsanalyses

Definitie

Tijdsanalyses baseren zich op vertragingen van bewerkingen in een chip, veroorzaakt door de vertraging van de transistoren [6]. Wanneer bepaalde bewerkingen in een algoritme plaatsvinden, is het mogelijk dat de ene bewerking langer zal duren dan de andere.

Tegenmaatregelen

Er bestaan op dit moment verschillende methoden om tijdsanalyses te dwarsbomen. Eén ervan is om zogenaamde *dummy operations* toe te voegen in een algoritme [7]. Andere maatregelen zijn het gelijktrekken van de duur van bepaalde bewerkingen, of het gebruik van asynchrone circuits die over kenmerken beschikken die het effect van een tijdsanalyse kunnen verbergen. Dit gebeurt dankzij een duale rail technologie in asynchrone circuits. Deze zorgt ervoor dat 1 signaal wordt voorgesteld door 2 duale verbindingen [6, 8].

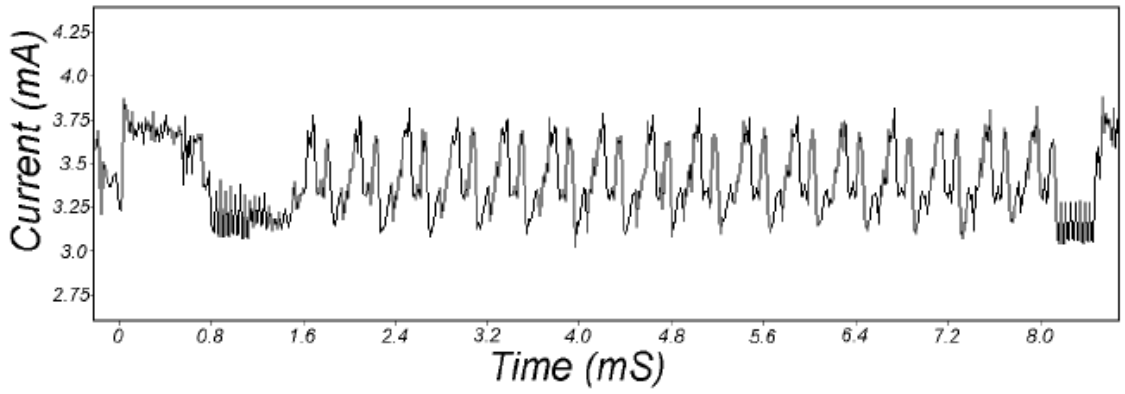
2.2.2 Vermogenanalyses

Definitie

Analyses van deze soort baseren zich op het gebruikte vermogen van de transistoren, waarbij er vooral gefocust wordt op schakelvermogen. Net zoals timing attacks variëren de patronen voor elke individuele bewerking die plaatsvindt. Vermogenanalyses komen zelf ook nog eens voor in minstens twee verschillende vormen.

Simpele vermogensanalyse

De simpele vermogensanalyse is zoals de naam het zegt een vrij simpele en straight-forward methode om een aanval uit te voeren. Er wordt een directe analyse uitgevoerd van de verbruikte stroom t.o.v. de tijd a.d.h.v. een enkele meting. Figuur 1 toont een dergelijke analyse van een volledige *Data Encryption Standard* (*DES*) operatie [5] en laat duidelijk zien hoe kwetsbaar een bepaalde chip kan zijn omwille van zijn fysieke implementatie. De 16 ronden van de *DES* operatie zijn namelijk duidelijk zichtbaar in de meting. Soortgelijke metingen kunnen al heel goedkoop uitgevoerd worden.



Figuur 1: Simpele vermogensanalyse [5]

Differentiële vermogensanalyse

Wanneer patronen moeilijker te herkennen zijn doordat ruis en vermogensvariaties in de instructiesequentie een grotere rol spelen, krijgt een differentiële vermogensanalyse de voorkeur [5].

In differentiële analyses is er een meer gestructureerde aanpak waarbij meerdere metingen gecombineerd worden en een statistische analyse wordt uitgevoerd om geheime informatie te achterhalen. Herneem het voorbeeld van *DES*. De eerste stap bestaat uit het uitvoeren van de *DES* operatie voor slechts één klokcyclus, waardoor enkel de *XOR*-operatie van de klaartekst met de sleutel plaatsvindt. Het resultaat hiervan staat direct hierna opgenomen in één van de ronde-registers. De volgende stap houdt vervolgens in om de chip in scan mode te plaatsen en de bits weer uit te scannen. Dit vele keren herhaald met verschillende klaarteksten kan aanleiding geven tot de eerste bit van de sleutel waarmee de bewerking werd uitgevoerd [4]. Na duizend tot tienduizend samples kan de gehele sleutel dus achterhaald worden.

Tegenmaatregelen

Om vermogensanalyse te hinderen zijn er verschillende wegen om uit te slaan. De chip zodanig afschermen om de toegankelijkheid van de testingangen weg te nemen is een optie, maar uiteraard de duurste waardoor de meeste fabrikanten wellicht naar andere oplossingen zoeken. Een tweede methode kan zijn om ruis te introduceren, waardoor een simpele vermogensanalyse al heel moeilijk wordt gemaakt. Er rest helaas nog de mogelijkheid om een differentiële aanval uit te voeren, waardoor dit ook niet de meest aangeraden oplossing is.

Deze thesis slaat hierdoor de weg uit van randomisatie van de gemeten gegevens, m.a.w. de sterke correlatie van de verwerkte data en de fysieke implementatie verzwakken. Herconfiguratie van bouwblokken in het algoritme is een methode die namelijk wel vaker de voorkeur heeft gekregen [9, 10, 11].

3 S-BOX in ASIC

De eerste stap van de implementatie is om het idee van een gerandomiseerde *S-BOX* te kopiëren naar *ASIC*-systemen. Doordat *S-BOX'en* voorkomen in *FPGA's* en *FPGA's* nu eenmaal andere faciliteiten hebben dan *ASIC's*, is er natuurlijk geen directe methode om dit zomaar te realiseren. Dit hoofdstuk vertelt meer over hoe *S-BOX'en* eruitzien in *FPGA's* en hoe het mogelijk is om een *S-BOX* in *ASIC* te implementeren.

3.1 Implementatie in FPGA

S-BOX'en komen zeer vaak voor in blokcijfers door hun simpele maar effectieve werking. Ze hebben een ingang en uitgang van n-bits. *LUT's* (Look-Up Tables) vertellen welke waarden overeen komen met de ingangen om daarna de ingang te substitueren (vandaar de naam *Substitution-BOX*) door de uitgang. De werking lijkt dus sterk op een directe aanpak van een waarheidstabel waarbij de ingangen direct aanleiding geven tot een bepaalde uitgang.

3.1.1 Werking

Figuur 2 geeft ten eerste de lineaire transformatie, en de substitutietabel van de AES *S-BOX*. In dit algoritme moet er een blok van 128 bits gesubstitueerd worden terwijl de *S-BOX* een ingang en uitgang heeft van 8 bits. Indien er enkel één *S-BOX* gebruikt wordt, zou de operatie pas voltooid zijn na 16 klokcycli. De eerste 4 bits van de ingang stellen een rij voor in de waarheidstabel terwijl de 4 laatste bits de kolom voorstellen. In de cel waar de aangewezen rij en kolom elkaar snijden staat de uitgang voor de *S-BOX* klaar. De inverse operatie van een *S-BOX* berekend bijgevolg terug de originele waarde

1.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

2.

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

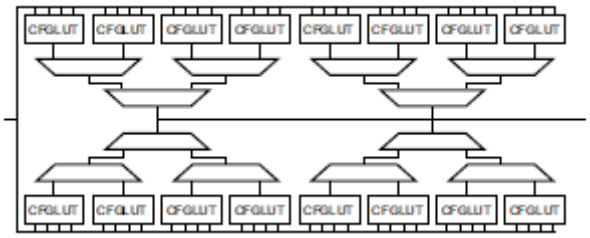
Figuur 2: *S-BOX* in *AES*. 1: Algebraïsche voorstelling; 2: Substitutietabel [1]

3.1.2 Configureerbare S-BOX

Een geavanceerdere vorm van de *S-BOX* is de configureerbare *S-BOX*. Met deze implementatie is het mogelijk om de configuratie van de *S-BOX on-the-fly* te kunnen aanpassen. Een *on-the-fly* of dynamische gebruiksmethode wilt zeggen dat de *S-BOX* continu gebruikt kan worden terwijl zijn configuratie om de zoveel tijd een update krijgt [11, 9]. Hetzelfde geldt dan voor de inverse *S-BOX*.

Deze gebruiksmethode introduceert interessante doeleinden om bijvoorbeeld nevenkanaalaanvallen tegen te gaan. Wanneer de configuraties dynamisch updates krijgen, krijgen de aanvallers na bepaalde tijdsintervallen een update waardoor ze hun tienduizenden samples niet meer kunnen correleren aan interne opslag van bepaalde registers en of *BRAM's* (dus *S-BOX* configuraties).

Figuur 3 toont hoe een *S-BOX* configureerbaar kan zijn. In dit geval werken de inputs als de *SELECT*-ingangen van de multiplexers, om zo de juiste registerwaarden te kunnen selecteren en door te verwijzen naar de uitgang. Een implementatie van deze soort maakt het zeer makkelijk om de register waarden te kunnen aanpassen, en daardoor controle te bieden over de functies van een *S-BOX*.



Figuur 3: Schema voor configureerbare *LUT's* in *S-BOX* [11]

3.2 Implementatie in ASIC

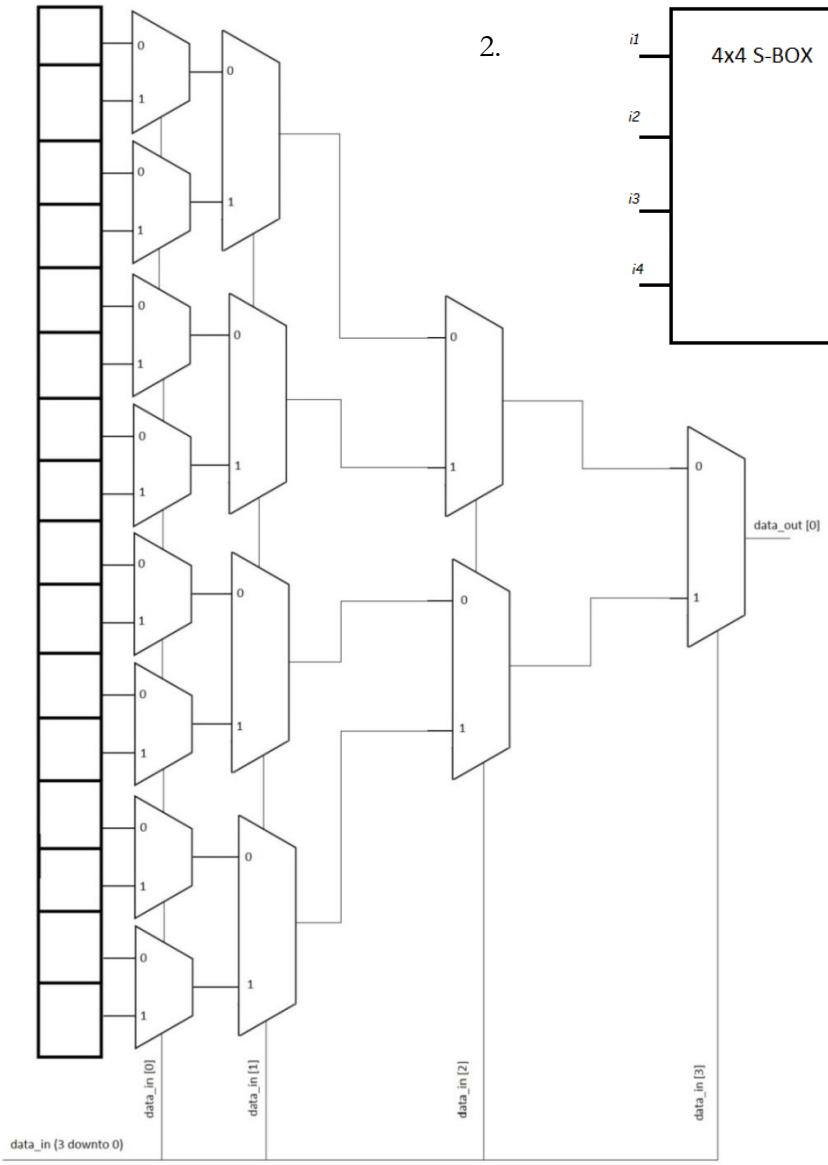
Configureerbare *S-BOX'en* maken het mogelijk om de verwerkte gegevens dynamisch te veranderen, terwijl ze toch hun functionaliteit behouden. Een variant van de configureerbare *LUT* in *FPGA* kopiëren naar *ASIC* is dus het eerste doel van deze thesis.

Omdat later een *PRESENT S-BOX* een aanpassing krijgt met de ontworpen implementatie van de permutatie-generatie staat hieronder een *S-BOX* uitgewerkt die dezelfde kenmerken heeft als de *PRESENT S-BOX*. In tegenstelling tot de 8x8 *S-BOX* uit het AES algoritme, bevindt er zich een 4x4 *S-BOX* in het *PRESENT*-algoritme.

Eender welke implementatie in *ASIC* betekent meteen dat er standaardcellen in zullen voorkomen. Rekening houdend met standaardcellen is er vervolgens de mogelijkheid om gebruik te maken van multiplexers. Zoals duidelijk in Figuur 4 bestaat de implementatie voornamelijk uit multiplexers, en een register van 16 bits.

Figuur 4 laat ook zien hoe één bit van de uitgang tot stand komt. De 4 bits van de ingangen werken elk als select-ingangen voor de 2x1 multiplexers. De eerste ingangsbit dient voor de eerste laag, de tweede bit voor de tweede laag enz. . Bijgevolg is de gehele 16-bit configuratie een register, die de inhoud bezit voor enkel de eerste bit van de uitgang. Dit schema komt dus 4x voor in de globale *S-BOX* implementatie om de 4 bits van de uitgang te kunnen bepalen. Elke eerste laag blijft bepaald door de eerste ingangsbit zoals de tweede laag ook bepaald blijft door de tweede ingangsbit. Het verschil zit enkel in de bepaling van de uitgangsbit, en de configuratie van het register (die voorlopig nog geen configuratie bevatten). Doordat de implementatie bestaat uit 4 registers waarbij elk register 16 bits groot is, zijn er zoals het moet 16 verschillende mogelijkheden voor de uitgang zodat elke waarde met een andere waarde (binnen het gebied van 16 getallen) gesubstitueerd kan worden. Verder is het ook mogelijk om elk register te initialiseren door het algoritme dat in het volgende hoofdstuk aan bod komt.

1.



2.

Figuur 4: 1: Deel-implementatie m.b.v. multiplexers; 2: 4x4 *S-BOX*

In Tabel 1 en Tabel 2 bevinden zich voorbeelden van een register configuratie met daarbij de overeenkomende waarheidstabel. Let erop dat de *LSB* in de configuraties begint aan de rechterkant. Om een getal terug te vinden moet de ingangswaarde als index dienen in de tabel van de registerconfiguratie. Vb. als de waarde 0 een 15 geeft, staat er in de 0^{de} kolom (beginnend van rechts, *LSB*) de bits 1111 af te lezen, wat dus overeenkomt met 15. Zo kan vb. de waarde 4 gesubstitueerd worden met 0001, wat overeenkomt met het getal 1 (Tabel 3).

Tabel 1: Voorbeeld van *S-BOX* configuratie

Register	Inhoud registers	Bepaalt uitgang:
reg1	0100100001101111	Data_out [0]
reg2	1000011101001101	Data_out [1]
reg3	1111001000101001	Data_out [2]
reg4	0101111001010001	Data_out [3]

Tabel 2: Overeenstemmende waarheidstabel van *S-BOX* configuratie

Ingang		Uitgang	Ingang		Uitgang
0	=>	15	8	=>	4
1	=>	8	9	=>	7
2	=>	12	10	=>	5
3	=>	14	11	=>	9
4	=>	1	12	=>	3
5	=>	10	13	=>	2
6	=>	13	14	=>	11
7	=>	0	15	=>	6

Tabel 3: Voorbeeld werking S-BOX

Index	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
reg1	0	1	0	0	1	0	0	0	0	1	1	0	1	1	1	1	MSB
reg2	1	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	
reg3	1	1	1	1	0	0	1	0	0	0	1	0	1	0	0	1	
reg4	0	1	0	1	1	1	1	0	0	1	0	1	0	0	0	1	LSB

Deze implementatie werd gesynthetiseerd in de *Design Compiler* van *Synopsys*, en het gaf als resultaat dat het een oppervlakte inneemt van 4063.7µm².

Tabel 4: Gebruikte oppervlakte van de 4X4 *S-BOX* implementatie in *ASIC*

Implementatie	Oppervlakte (µm²)
4x4 S-BOX	4063.7

4 Willekeurige permutatie algoritme

De volgende stap in deze thesis houdt in om de registerwaarden in het laatste voorbeeld van het vorige hoofdstuk willekeurig te kunnen genereren. De vereisten van een dergelijke algoritme zijn dat het enerzijds ook een omgekeerde werking moet hebben, zodat de inverse configuratie voor de *S-BOX'en* ook kan bestaan. Anderzijds moet het algoritme unieke getallen kunnen genereren zodat elk getal niet dubbel kan bestaan in de registers van de nieuwe *S-BOX*. Dit hoofdstuk vertelt in het eerste deel welk algoritme aan bod komt in de implementatie, welke in het volgende deel verder staat uitgelegd.

4.1 Algoritme

Aangezien het algoritme enkel unieke getallen mag genereren, zal een directe aanpak met willekeurige getal-generators zoals *Linear Feedback Shift Registers* (*LFSR's*) niet volstaan. Desondanks is het wel mogelijk om een getal-generator te gebruiken om getallen in een willekeurige volgorde te plaatsen. Met andere woorden zorgt het algoritme dan voor een permutatie van een bepaalde getallenrij [12]. Het algoritme is bijgevolg ook volledig gebaseerd op de generatie van een willekeurige permutatie van een getallenrij. Merk op dat de lengte van de getallenrij in het geval van de 4x4 *S-BOX* gelijk is aan 16. In de volgende hoofdstukken bevinden zich enkele methoden om willekeurige permutaties te genereren.

4.1.1 Fisher-Yates shuffle

Het algoritme ontwikkeld door Fisher en Yates (ook bekend als *Knuth shuffle* in 1938 [13]) is één van de eerste algoritmes waar de willekeurige permutatie mee begon [14]. Het stappenplan is als volgt:

- noteer alle getallen van 0 tot N;
- neem een willekeurig bestaand getal k uit de getallenrij;
- verwijder deze uit de getallenrij;
- noteer het getal op het einde van een tweede getallenrij;
- herhaal stap 2 t.e.m. 4 tot er geen getallen meer overblijven;
- het resultaat is een permutatie van de originele getallenrij.

Zoals duidelijk is dit de meest eenvoudige en voor-de-hand-liggende vorm van een permutatie. Dit in tegenstelling tot de implementatie in hardware ervan. Het probleem ontstaat doordat getallengeneratoren enkel een sequentie van bit-waarden genereren. Hierdoor is er geen controle om een reeds gegenereerd getal niet meer opnieuw te genereren, tenzij het resultaat verworpen wordt waarna een nieuw getal gegenereerd wordt. In de eerste iteraties is dit nog haalbaar, maar vanaf er vb. 3 getallen overblijven uit de 16 eerdere mogelijkheden zouden er steeds 13 niet aan de orde zijn, totdat er één gevonden is. In de volgende stap zijn de mogelijkheden beperkt tot slechts 2 keuzes. Hierdoor kan er een groot verlies aan snelheid optreden van het algoritme.

4.1.2 Durstenfeld

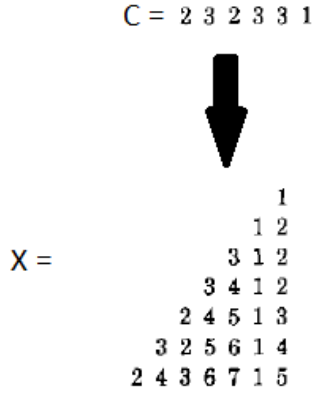
Durstenfeld ontwikkelde een modernere vorm van de Fisher-Yates *shuffle* [15, 16, 17]. Figuur 5 toont het algoritme in code-vorm. Durstenfeld stelt voor om in plaats van de willekeurige getallen in een nieuwe getallenrij te plaatsen, het willekeurig getal te wisselen met het getal op de positie van de i-de iteratie. Ook kan het willekeurig getal enkel gelegen zijn tussen 1 en de i-de iteratie. Deze methode verkleint het snelheids-probleem bij de Fisher-Yates *shuffle* aanzienlijk. Helaas is in dit algoritme het verwisselen van de twee getallen niet efficiënt te implementeren. Een mogelijkheid om dit te implementeren is een cascade van multiplexers, waar een getal telkens door moet passeren als het niet in de bijhorende register hoort. De vele multiplexers nemen ruimte in, terwijl het doorgeven van de getallen ook zijn invloed zal hebben op de snelheid.

```
for i ← n to 1 do
j ← random integer with 1 ≤ j ≤ i
exchange(S[j], S[i])
end
```

Figuur 5: Algoritme van Durstenfeld in codevorm [13]

4.1.3 D.H. Lehmer

D.H. Lehmer heeft een complexer algoritme voorgesteld dan de vorige beschreven algoritmes. Dit op basis van het algoritme van D.N. Lehmer. Zij bepalen eerst een getallenrij C dat gegenereerd is volgens een telling van faculteiten [12]. Vervolgens nemen ze een getallenrij van 1..N in rekening, waarbij ze in de eerste stap het getal 1 meteen noteren in een nieuwe getallenrij X. Vervolgens itereren ze in C en schrijven C[i] altijd links van de bestaande getallenrij. Hun input hierbij is dat alle getallen reeds in de getallenrij X, die kleiner zijn of gelijk aan C[i] met 1 geïncrementeerd moeten worden. Dit ziet eruit zoals weergegeven in Figuur 6. Het algoritme vertrekt uit de notering van het getal 1. In de eerste stap is C[i] = 1 waardoor het reeds ingevulde getal '1' met 1 geïncrementeerd wordt, waardoor de getallenrij nu eruitziet als 1 2. In de volgende stap is C[i] = 3, waardoor het getal 3 links genoteerd mag worden zonder andere getallen te incrementeren.



Figuur 6: Algoritme van Lehmer [12]

4.2 Implementatie

Deze thesis baseert zich op het algoritme van D.H. Lehmer om een variant ervan uit te werken. De bepaling van de permutatie gebeurt zonder de telling van faculteiten. Eerst genereert een getalgenerator willekeurige getallen om die daarna door te sturen naar een *Finite State Machine* (*FSM*).

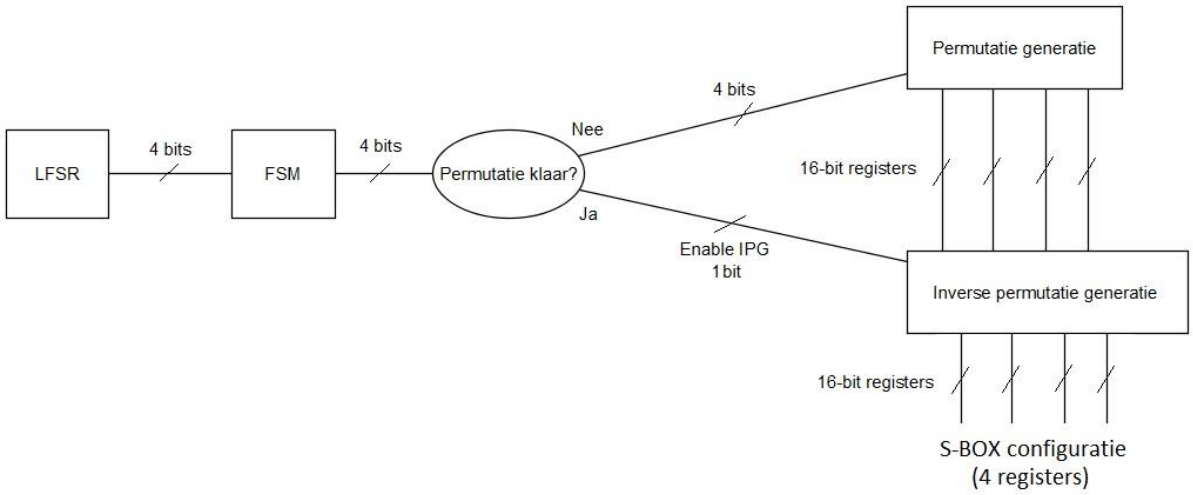
De overeenstemming met het algoritme van Lehmer zit in de regel van getallen incrementeren: Indien een gegenereerd getal zich al in de permutatie bevindt wordt dit getal links geplaatst waarna alle getallen in de permutatie die groter of gelijk zijn aan dat getal incrementeren. Figuur 7 geeft het algoritme weer in codevorm.

```
Array perm_array, numbers;
for (int i = 0; i < 16 ; i++) {
    int x = numbers[i];

    for (int j = 0; j< 16 ; j++) {
        y = perm_array[j];
        if (x >= y) {
            y = y + 1;
            perm_array[j] = y;
        }
    }
    perm_array.shift();
}
```

Figuur 7: Algoritme in codevorm

De implementatie van de willekeurige permutatie generatie is opgedeeld in 4 delen. Eerst genereert een *LFSR* willekeurige getallen van 4 bits. Om een veilig systeem te bekomen kan beter geopteerd worden voor een *True random number generator*, maar omdat de focus voornamelijk op de methode van de permutatie-generatie ligt, volstaat het gebruik van een LFSR. De gegenereerde getallen worden behandelt door de *FSM* die op zijn beurt de getallen doorstuurt naar de permutatie generatie. De permutatie generatie bepaalt dus de volgorde van alle gegenereerde getallen die voortvloeien vanuit de *FSM*. Wanneer de permutatie klaar is, is het mogelijk om de inverse permutatie te berekenen a.d.h.v. de originele configuratie van de *S-BOX*. Voor de verduidelijking toont de onderstaande figuur de logische werking van het algoritme.

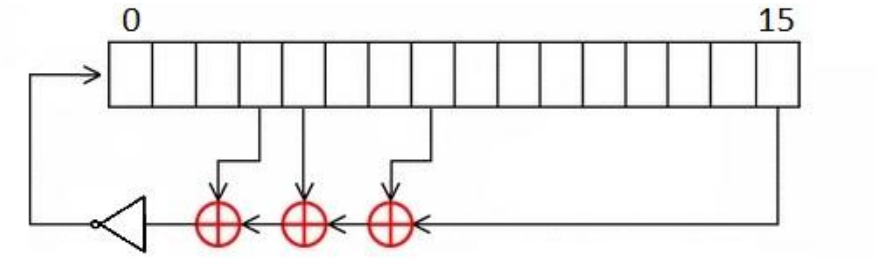


Figuur 8: Werking van het algoritme

4.2.1 Getal-generator

Het eerste element dat voorkomt in het ontwerp is een generator dat willekeurige getallen genereert. Hier bestaat de keuze tussen een pseudowillekeurige getal generator en een "echte" willekeurige getal generator. Een pseudo-generator genereert getallen die toch nog een kleine afhankelijkheid hebben van elkaar, in tegenstelling tot een "echte" willekeurige getal generator. Maar aangezien hier de generatie van een permutatie centraal staat, is de onafhankelijkheid van elk willekeurig gegenereerd getal minder van belang. Hierdoor werd er gekozen voor een *Linear Feedback Shift Register* (*LFSR*) omdat dat een veel-gekozen techniek is om getallen te genereren doordat het eenvoudig te implementeren is in hardware.

De LFSR in deze implementatie is een willekeurig ontworpen type van een *LFSR*. Er werd niet specifiek op zoek gegaan naar een *LFSR* met een hoge entropie in zijn output omdat nogmaals de permutatie generatie eerder centraal staat in deze thesis. Figuur 9 toont het schema van de gebruikte *LFSR*. Deze *LFSR* heeft een registerlengte van 16 bits, waarbij er 4 bits een *XOR*-operatie ondergaan gevolgd door een *NOT*-operatie. Het resultaat hiervan dient als de feedback-bit en wordt links in de register geplaatst, waarna de hele register naar rechts shift met 1 bit. De eerste 4 bits van de register zijn in elke klok-cyclus de representatie van het willekeurig getal dat als input dient van de *FSM*.



Figuur 9: *LFSR*

4.2.2 Finite State Machine

De *FSM* bepaalt of de gegenereerde getallen geldig zijn om in de permutatie te plaatsen. Denk eraan dat de ontworpen *S-BOX* 4 registers heeft die 16 bits lang zijn, dus 16 getallen van 4 bits. De geldigheid wordt bepaald als volgt:

- Stap 1: plaatsen van het getal 0 in de permutatie
- Stap 2: generatie mag enkel tussen 0 en i-2 (dus 0 en 1 zijn geldig)
    - Drie 0-bits doorsturen, en enkel de laatste bit overnemen van het gegenereerd getal
- Stap 3: generatie mag enkel tussen 0 en i-2 (dus 0, 1 en 2 zijn geldig)
    - Twee 0-bits doorsturen, en de laatste 2 bits overnemen van het gegenereerd getal. Indien deze 3 is, verwerp het resultaat en genereer een nieuw getal.

.
.
.

- Stap 11: generatie mag enkel tussen 0 en i-2 (dus getallen van 0 t.e.m. 9 zijn geldig)
    - Alle bits overnemen van het gegenereerd getal. Indien deze groter is als 9, verwerp het resultaat en genereer een nieuw getal.

Het algoritme is het meest gevoelig in stap 11. Hier is er 56.25% kans dat een gegenereerd getal geldig is. Toch zal het algoritme blijven genereren tot het aan de vereiste voldoet.

4.2.3 Permutatie generatie

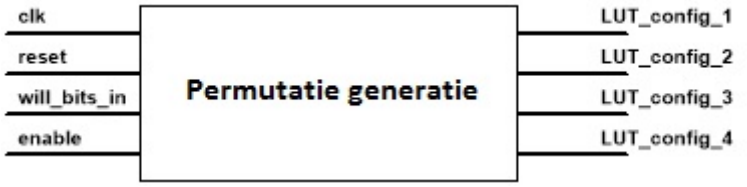
Nu er een geldig willekeurig getal beschikbaar is, is het mogelijk om dit getal te plaatsen in een nieuwe getallenrij. De implementatie hiervan bestaat uit drie delen waarbij voornamelijk de bibliotheek van *IEEE.NUMERIC_STD* en *IEEE.STD_LOGIC_UNSIGNED* aan bod komen. Merk op dat het plaatsen van één getal in de nieuwe getallenrij, inclusief het shiften en het incrementeren van de andere getallen indien nodig, overeenkomt met één klokcyclus. Onderstaande schema's beschrijven de implementatie.

De getallenrij komt voor in de *temp_reg* beschreven in Figuur 11. De lengte van dit register is 64 bits zodat 16 getallen, elk van 4 bits, in de register kunnen staan. Bij elke nieuwe klokcyclus zorgt een for-loop voor het incrementeren van een *Integer* waarde i die bovendien beschreven staat in de bibliotheek van *NUMERIC_STD*. Dit getal haalt een bepaald getal van 4 bits uit de getallenrij, waarna het gekozen getal vergeleken wordt met het gegeneerde willekeurig getal, die daarbij ook geconverteerd is naar een Integer waarde. Indien het getal uit de register groter is zal het getal geïncrementeerd worden waarna het terug op dezelfde plaats in de register binnenkomt. Hierna gebeurt de shift die overigens ook in een for-loop beschreven staat zodat er een shift van 4 bits mogelijk is.

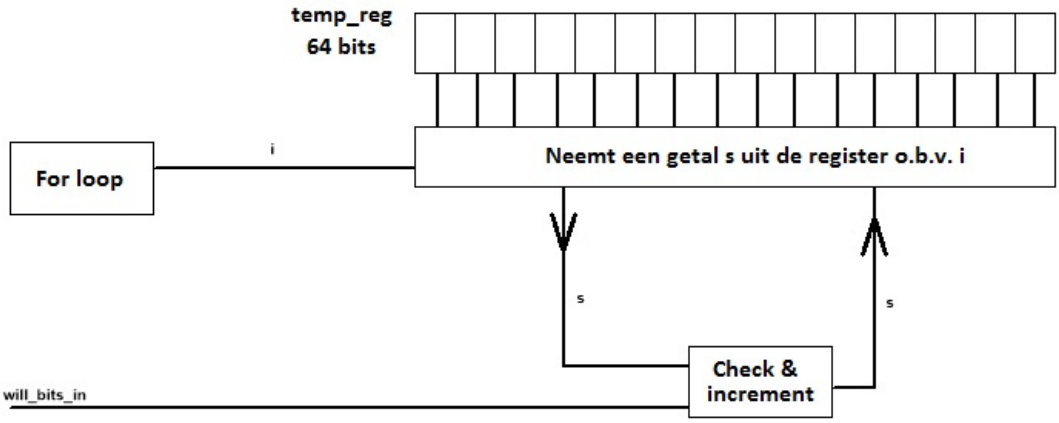
Het schema in Figuur 12 weergeeft de werking van de plaatsing in de *LUT*-registers. Ook hier zorgt een for-loop voor de verwerking van alle getallen, waarbij alle eerste bits van de getallen in de eerste *LUT*-register binnenkomen, alle tweede bits in de tweede *LUT*-register, enz…

Beide for-loops vinden dus plaats in één klokcyclus en zullen dus 16x plaatsvinden voordat de permutatie klaar is.

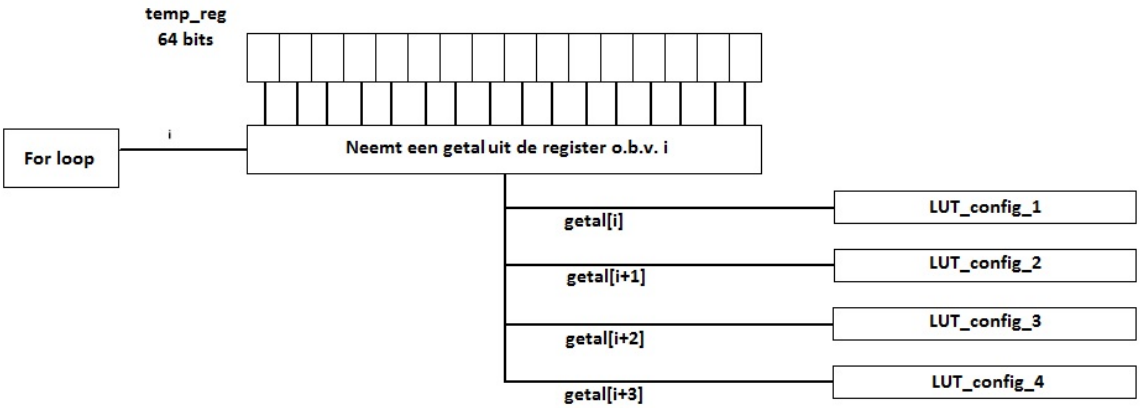
Merk op in Figuur 11 dat in de eerste fase van de loop de register nog leeg is, of m.a.w. gevuld is met nullen. Ook deze getallen komen voor in de vergelijking met het willekeurig getal, maar aangezien na elke fase een shift plaatsvindt van 4 bits, zullen deze getallen uiteindelijk uit de register vallen waardoor enkel de permutatie van de gegenereerde getallen in de register zal zitten.



Figuur 10: Bouwblok permutatie generatie



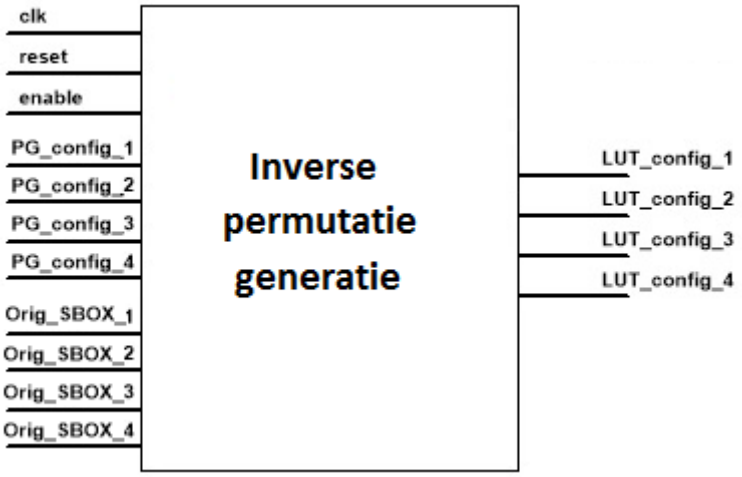
Figuur 11: Invoegen van het willekeurig getal



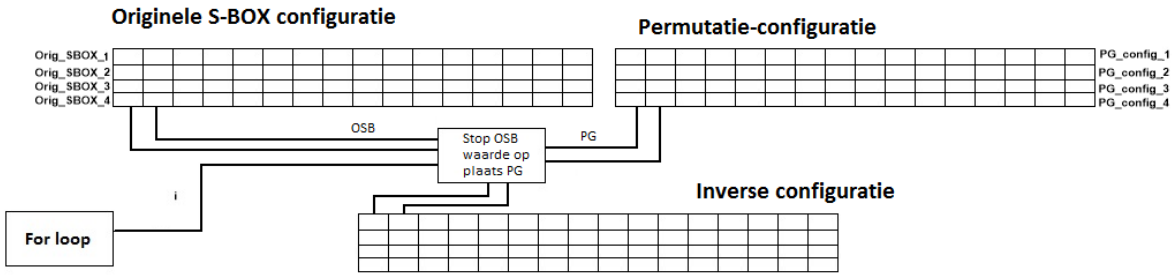
Figuur 12: Invoegen van de getallen in de LUT configuraties

4.2.4 Inverse permutatie generatie

De inverse permutatie generatie werkt tenslotte volgens adresseringen. Dit wil dus zeggen dat de generatie enkel mogelijk als de configuraties van de originele *S-BOX* en de configuraties na de permutatie generatie bekend zijn. Figuur 13 en Figuur 14 tonen het bouwblok en de werking in een schema. Opnieuw is de werking gebaseerd op een for-loop welke van 0 tot 16 telt. In deze loop staan 2 conversies van enerzijds de eerste waarde in de originele *S-BOX* configuratie (*OSB*) en anderzijds de eerste waarde in de permutatie-configuratie (*PG*). De waarde *PG* bepaalt op welke positie van de inverse configuratie de waarde *OSB* zal komen. Na 16 iteraties in slechts 1 klokcyclus is deze operatie tenslotte voltooid.



Figuur 13: Bouwblok inverse permutatie generatie



Figuur 14: Werking inverse permutatie

4.3 Gebruik van S-BOX implementatie

Om de *S-BOX* ten slotte de mogelijkheid te geven voor dynamisch gebruik kan een *ENABLE* signaal de permutatie-generatie starten. Wanneer de permutatie klaar is triggert er een *LUTS_GENERATED* signaal om de top laag te laten weten dat het de nieuwe registers mag inladen in de registers van de *S-BOX*. Doordat de tussenresultaten staan opgeslagen in flip-flops, gebeurt dit in twee stappen. In de eerste stap laadt de toplaag de permutatie-registers in. In de tweede stap is het de beurt aan de inverse permutatie registers. Het stappenplan om de *S-BOX* in gebruik te nemen ziet eruit zoals volgt:

- *ENABLE* = "1" maken
- Wacht tot *LUTS_GENERATED* = "1"
- Maak in dezelfde cyclus *ENABLE* = "0"
- Stuur de eerste gegevens in wanneer *LUTS_GENERATED* terug op "0" springt
- Vanaf dit punt mag *ENABLE* "1" worden op elk gewenst moment en terug "0" één klokcylus nadat *LUTS_GENERATED* "1" werd. De gegevens mogen in de tussentijd blijven stromen, maar best in een ander proces.

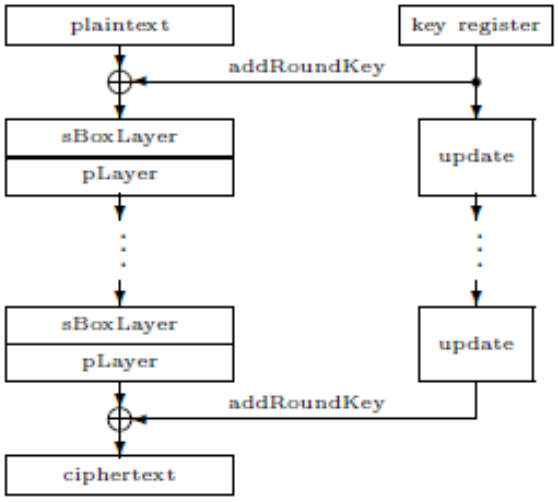
5 Implementatie permutatie algoritme in PRESENT S-BOX

De laatste stap in deze thesis bestaat uit de implementatie van het permutatie-algoritme in een *PRESENT S-BOX*. Dit hoofdstuk start met het toelichten van het *PRESENT* algoritme gevolgd door de uitwerking van de implementatie. Tenslotte eindigt dit hoofdstuk met de simulatie en evaluatie.

5.1 PRESENT algoritme

5.1.1 Principes

Het *PRESENT*-algoritme is een blokcijfer dat ontworpen is voor low-cost cryptografische systemen. De blokgrootte is 64 bits, dat 31 rondes ondergaat gebruik makend van 31 sub-sleutels. Elke sub-sleutel ontstaat uit een initiële sleutel van 80 of 128 bits groot. Hieronder bevindt zich de top laag van het encryptieschema van het *PRESENT* algoritme, welke ook duidelijk maakt dat het algoritme gebaseerd is op een substitutie-permutatie netwerk [2, 11].



Figuur 15: Top laag beschrijving van het *PRESENT* algoritme [2]

Omdat dit hoofdstuk zich baseert op een implementatie op de *S-BOX* laag van het algoritme, ligt de focus op de *S-BOX* laag en verwijst deze thesis naar [2] waarin meer informatie terug te vinden is over de permutatie-laag in dit algoritme.

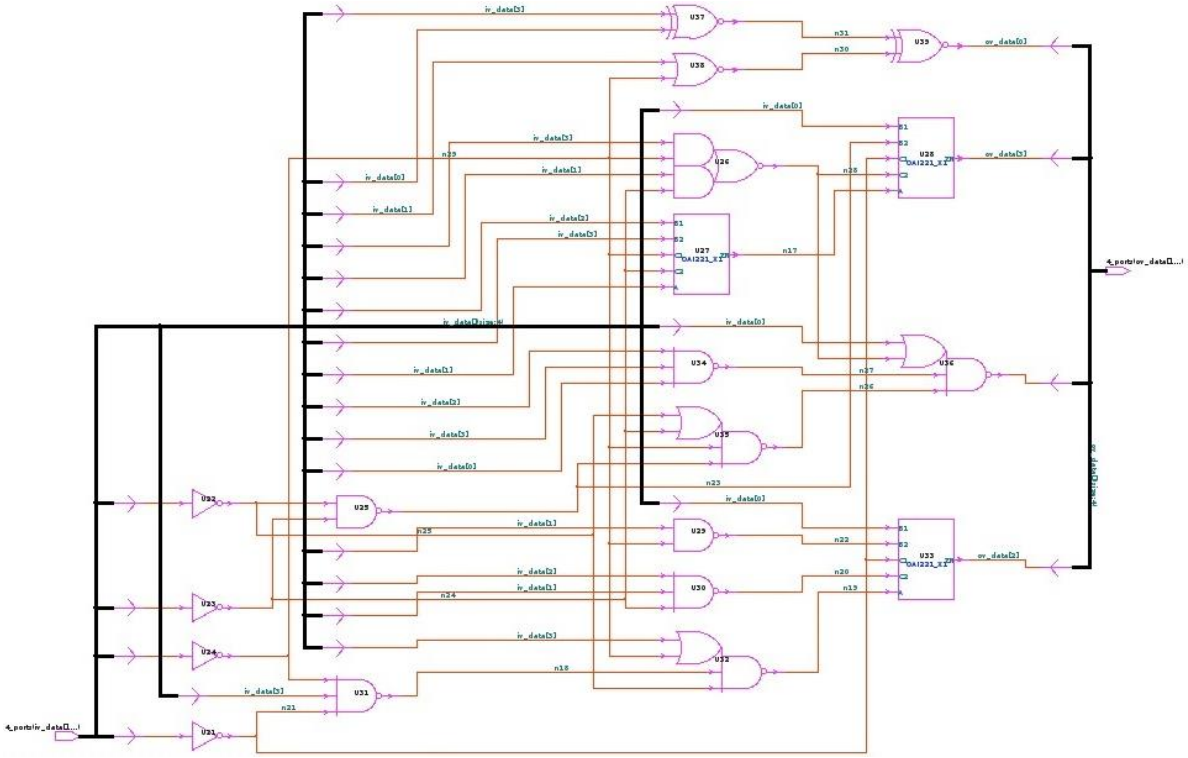
5.1.2 S-BOX laag

Zoals reeds vermeld in de uitwerking van de *S-BOX* implementatie heeft de *S-BOX* 4 in- en uitgangsbits. De *S-BOX* substitueert zoals staat weergegeven in onderstaande tabel.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Tabel 5: Substitutietabel van PRESENT S-BOX

In een *FPGA* is het opnieuw mogelijk om dit te implementeren als een *LUT*. In *ASIC* daarentegen ziet de *S-BOX* eruit als één grote logische functie. Bijgevolg zal een circuit van gates zoals *AND*, *OR*, *XOR*, enz.. de uitgangswaarde S(x) moeten geven. Desondanks kunnen enkel de klassieke standaardcellen de werking niet garanderen, waardoor er nood is aan een bibliotheek die de standaardcellen transformeert tot cellen die een *S-BOX* werking wel kunnen realiseren. De standaardcelbibliotheek die deze thesis gebruikt heet *NangateOpenCell* [18]. Na synthese van een Verilog script van de *PRESENT S-BOX* volgt de *netlist* die staat weergegeven in Figuur 17. Nevenkanaalaanvallen oefenen zich uit op een *netlist* zoals in deze figuurFiguur 17**Fout! Verwijzingsbron niet gevonden.**. De aanvallers meten de verwerkte data op bepaalde knooppunten om de data uit te kunnen lezen. De implementatie in dit hoofdstuk probeert de gegevens in deze knooppunten willekeurig te maken om de correcte representatie van de gegevens weg te nemen.



Figuur 16: *Netlist* van *PRESENT S-BOX*

5.2 Implementatie

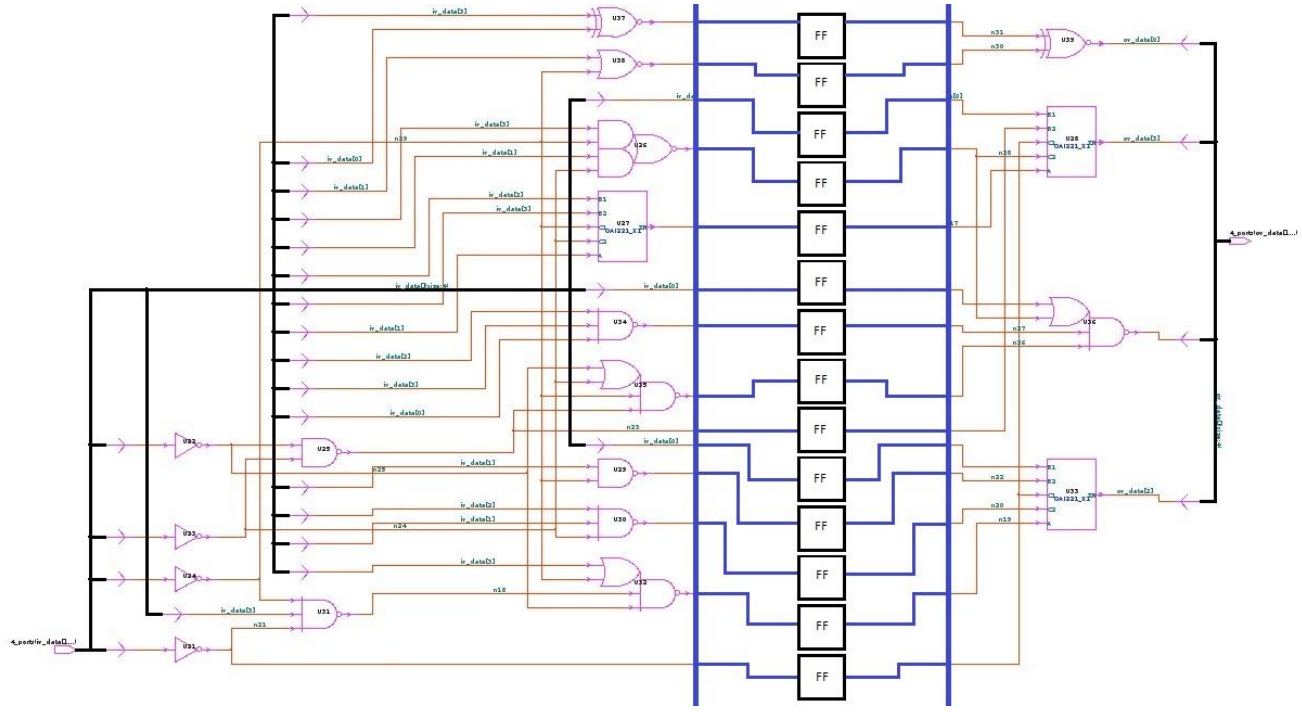
De implementatie van de permutatie generatie zal licht moeten verschillen tot de implementatie in het vorige hoofdstuk. De reeds besproken versie berekent namelijk de inverse permutatie om de gehele werking van de *S-BOX* te realiseren. Die versie probeerde de ingangen beschreven in Tabel 5 te herleiden naar de uitgangen beschreven in dezelfde tabel. Nu ligt de focus echter op de knooppunten, waardoor de data in de knooppunten uiteindelijk hetzelfde moet blijven zodat de werking van de hele *S-BOX* niet verandert. De werking van de inverse permutatie zal dus niet meer rekening houden met een originele *S-BOX* configuratie en dus enkel met de inhoud van de permutatie registers. Deze zal dus bv. een getal y uit de permutatie register op positie x nemen, en het getal x plaatsen in een nieuwe inverse permutatie register op plaats y.

Verder bestaat het doel uit een permutatie algoritme toe te passen op de besproken knooppunten in de *netlist*. De knooppunten ontstaan uit het verdelen van de *netlist* in twee deelblokken zoals in Figuur 17. Deze laat meteen ook zien wat er gebeurt met de willekeurige gegevens na het verdelen van de *netlist*. Het resultaat van de permutatie zal in flip-flops komen zodat er nieuwe knooppunten ontstaan. Aanvallers zullen nu dus de willekeurige gegevens lezen uit de flip-flops i.p.v. de directe knooppunten.

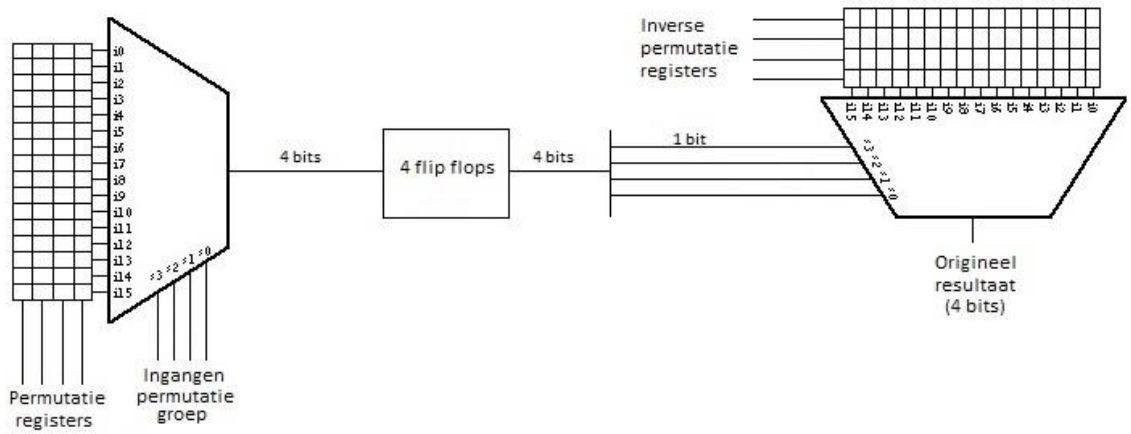
In dit voorbeeld bestaan er verder 14 links tussen het linker deelblok en het rechter deelblok. Al deze links stellen 14 bits voor die samen 16 384 permutatie-mogelijkheden hebben. Dit als één geheel te permuteren zou een groot design vragen, wat op low-cost toestellen niet echt gewenst is. Omwille van deze reden is het beter om de 14 bits te groeperen in groepen van twee keer 4 en twee keer 3 permutatiegroepen. Dit wilt dus zeggen dat er nu vier simultane berekeningen plaatsvinden waarvan twee met vier bits en twee met drie bits.

Het vorige hoofdstuk hield een implementatie in van een permutatie generatie van vier bits. Deze implementatie komt in dit voorbeeld bijgevolg twee keer voor. Bovendien zijn er ook nog eens twee implementaties die drie bits permuteren. Deze werken volledig volgens hetzelfde principe als diegene van vier bits, met uiteraard de uitzondering op de inverse permutatie generatie zoals eerder verteld in dit hoofdstuk.

Een laatste opmerking is dat er in dit geval geen *S-BOX'en* meer zijn om van de gegenereerde registers een uitgang te bepalen. De registers worden in dit voorbeeld gebruikt in multiplexers, waarbij de ingangen van permutatiegroepen als de *SELECT*-ingangen zullen werken. Eén multiplexer heeft, in het geval van een 4 bits permutatiegroep, 4 registers om uit te kiezen terwijl een 3 bits permutatiegroep 3 registers heeft om uit te kiezen (dus ook een *SELECT*-ingang van 3 bits). De werking hiervan staat weergegeven in Figuur 18.



Figuur 17: Aangepaste *netlist* van *PRESENT S-BOX*



Figuur 18: Selectie van gepermuteerde en inverse gepermuteerde waarden in een permutatiegroep van 4 bits

5.3 Simulatie en evaluatie van implementatie

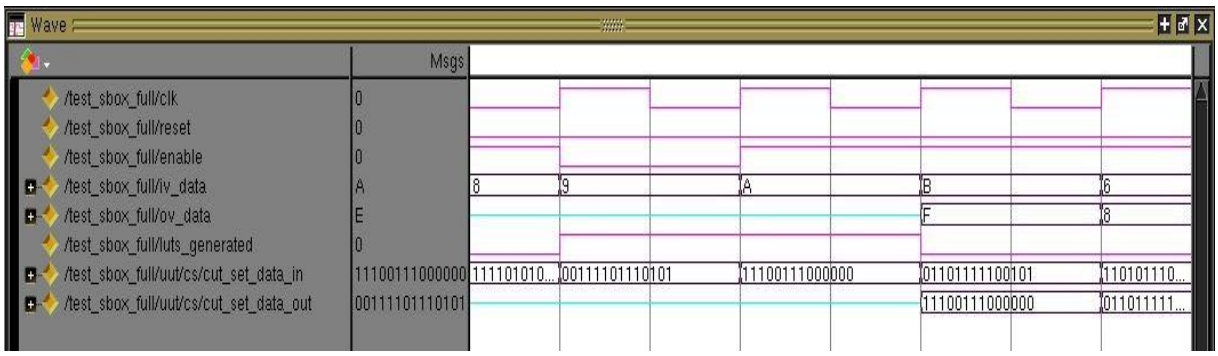
5.3.1 Simulatie

Om de werking te tonen zal het programma *ModelSim* een simulatie maken van het ontwerp. Dit met de volgende ingangen en de verwachtte uitgangen van de *PRESENT S-BOX*.

Tabel 6: Ingangen (iv_data) met de verwachtte uitgangen (ov_data) van de *PRESENT S-BOX*

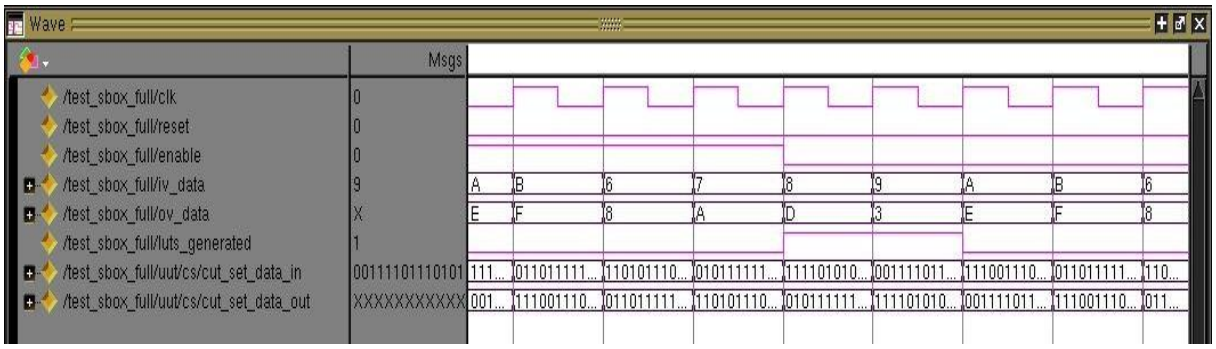
iv_data	6	7	8	9	A	B
ov_data	A	D	3	E	F	8

Figuur 19 weergeeft de simulatie. Merk op dat nog steeds hetzelfde stappenplan uit het vorige hoofdstuk geldt voor de werking van deze implementatie. Nadat *LUTS_GENERATED* triggert naar "1", hoort ENABLE op "0" te springen. Op het ogenblik dat *LUTS_GENERATED* terug op "0" springt zullen de eerste gegevens ook zichtbaar zijn aan de uitgang. Omdat er verder zich flip-flops bevinden tussen de twee deelblokken, is het resultaat van de ingang pas duidelijk in de volgende cyclus. Dit weergeeft zich naast het ingang-uitgang paar ook in de registers *"cut_set_data_in"* en *"cut_set_data_out"*, welke de verzameling van alle permutatiegroepen voorstelt aan de ingang en uitgang.



Figuur 19: Simulatie PRESENT S-BOX

Figuur 20 weergeeft vervolgens de simulatieresultaten waarin de dynamische werking aan bod komt. Zoals duidelijk is in het begin van het simulatie worden de gegevens nog steeds verwerkt. Na een aantal cyclussen springt *LUTS_GENERATED* opnieuw op "1", wat dus wilt zeggen dat een nieuwe configuratie klaar is voor in te laden (in dit geval in de registers van de multiplexers). Zoals reeds verteld gebeurt dit in 2 stappen (inladen van permutatie-registers gevolgd door de inverse permutatie registers). De gegevens blijven gedurende het inladen nog steeds de juiste uitgang behouden.



Figuur 20: Weergave van dynamische werking van de *PRESENT S-BOX*

5.3.2 Evaluatie met Nangate45 OpenCell bibliotheek

De synthese van deze implementatie werd zoals eerder vermeld ook gedaan in de *Design Compiler* van *Synopsys*. Dit deel bespreekt de evaluatie van de oppervlakte en de maximale klokfrequentie die kan worden ingesteld voor het ontwerp. De maximale klokfrequentie kan bepaald worden in het kritisch pad van het design. De compiler gaf hierbij aan dat het kritisch pad tussen de registers iv_data[3] (dus de laatste ingangsbit) en int_res_reg_7 (7^{de} flip-flop van het tussenresultaat) zit. Dat het kritisch pad niet van ingang tot uitgang loopt is logisch, aangezien het ontwerp nu twee klokcyclussen duurt voordat de data aan de uitgang zichtbaar is.

Dezelfde evaluatie is terug te vinden voor de originele PRESENT S-BOX, om de vergelijking tussen beide mogelijk te maken. In onderstaande tabel bevinden zich de resultaten van de compilatie.

Tabel 7: Evaluatie van oppervlakte en maximale klokfrequentie van het ontwerp vergeleken met de originele PRESENT S-BOX

Implementatie	Oppervlakte (μm^2)	Maximale klokfrequentie (GHz)
Originele *PRESENT S-BOX*	20,5	2,92
Implementatie	7909,2	1,72

Het is duidelijk en logisch dat er een groot verschil is in oppervlakte tussen beide designs. Het ontwerp uit deze thesis omvat een permutatie methode om een veiligheid tegen vermogenaanvallen aan te bieden, wat de originele S-BOX niet heeft. Ook is er een kost van de maximale klokfrequentie in het kritisch pad. Ook dit is logisch doordat er zich bv. grote multiplexers bevinden in het ontwerp om een tussenresultaat te kunnen opslaan in de flip-flops. Deze multiplexers worden gerealiseerd door vele verschillende gates die bijgevolg vele vertragingen met zich meebrengen.

6 Besluit

Deze thesis startte met het beschrijven van de principes van nevenkanaalaanvallen. Het gestelde doel omving bijgevolg het willekeurig maken van de verwerkte gegevens in een *ASIC* systeem, zodat de gegevens die aanvallers zouden meten hun representativiteit zouden verliezen.

In het daaropvolgende hoofdstuk werd daarom gestart met het uitwerken van een implementatie van een 4x4 *S-BOX* in een *ASIC*. Het idee van een een configureerbare *LUT* werd hierbij in rekening genomen om het te kunnen implementeren in een *ASIC* systeem. Het resultaat omvatte dus een cascade van multiplexers die met hun *SELECT* signalen de juiste data uit de registers haalde. De synthese van dit ontwerp gaf een oppervlakte van 4063,7µm².

Hoofdstuk 4 vertelde daarop de methode om de registers te vullen, hoe deze dynamisch updatete terwijl de *S-BOX* toch zijn originele functionaliteit behield. De methode is daarbij gebaseerd op het permutatie-algoritme van D.H. Lehmer beschreven in bron [12] maar kreeg wel nog enkele aanpassingen om een snel en compact design te realiseren.

Het idee van deze implementatie stond ook centraal in hoofdstuk 5, om een variant uit te werken om een toepassing in een *PRESENT S-BOX* te behalen. De *Design Compiler* van *Synopsys* synthetiseerde een Verilog beschrijving van de *PRESENT S-BOX*. In deze *netlist* konden aanvallers bepaalde knooppunten uitlezen, waardoor de *netlist* werd verdeeld in twee stukken zodat de gegevens in de knooppunten willekeurige betekenissen kregen via het permutatie-algoritme. De connecties tussen de twee deelstukken werden verder ook nog verzameld tot permutatiegroepen om het design compact te houden. In tegenstelling tot hoofdstuk 3 verzorgden hier multiplexers de selectie van de juiste gegevens uit de registers, omdat niet de gehele S-BOX maar slechts knooppunten in deze *S-BOX* bekeken werden. De grootte van enkel het bijgevoegde tussenstuk bedroeg 7281,2 µm².

Deze thesis kan tenslotte verder onderzoek inleiden door het verbeteren van de snelheid en grootte van de implementatie. Anderzijds heeft het, gebruik makend van de bibliotheek *NanGateOpenCell*, ook een basis gelegd om meerdere toepassingen van *LUT's* in *FPGA* te kopiëren naar *ASIC*.

Literatuurlijst

[1] F. PUBS, „Advanced Encryption Standard (AES),” National Institute of Standards and Technology, 2001.

[2] L. K. G. L. C. P. A. P. M. R. Y. S. a. C. V. A. Bogdanov, „PRESENT: An Ultra-Lightweight Block Cipher,” Horst-Görtz-Institute for IT-Security, Ruhr-University Bochum, Germany, Technical University Denmark, DK-2800 Kgs. Lyngby, Denmark, France Telecom R&D, Issy les Moulineaux, France.

[3] J. F. T. Souvignet, „Differential Power Analysis as a digital forensic tool,” Forensic Science International, French Gendarmerie National Forensics Lab (IRCGN), Digital Forensics Department (INL), 1 boulevard The´ophile Sueur, 93110 Rosny-Sous-Bois, France, 2013.

[4] A. Das, „Differentiel Scan-Based Side-Channel Attacks and Countermeasures,” KU Leuven, Kasteelpark Arenberg 10 (Belgium), 2013.

[5] J. J. B. J. Paul Kocher, „Power Analysis Attacks,” Advances in Cryptology, CRYPTO '99, Santa Barbara, California, 1999.

[6] M. L. C. P. J. D. D. R. T. S. C. S. Washington Cilio, „Mitigating power- and timing-based side-channel attacks using dual-spacer dual-rail delay-insensitive asynchronous logic,” in *Microelectronics Journal*, Department of Computer Science & Computer Engineering, University of Arkansas, ENGR311, CSCEDept., Fayetteville, AR72701, UnitedStates ; Department of Electrical Engineering, University of Arkansas, JBHT-CSCE504, Fayetteville, AR72701, UnitedStates, Elsevier, 2013, pp. 258-269.

[7] H. L. F. Y. Keke Wu, „Retrieving Lost Efficiency of Scalar Multiplications for Resisting against Side-Channel Attacks,” in *Journal of Computers*, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 2010, pp. 1878-1884.

[8] J. M. A. B. A. Y. Danil Sokolov, „Design and Analysis of Dual-Rail Circuits for Security Applications,” in *IEEE Transaction on computers*, 2005, pp. 449-460.

[9] B. G. I. V. Nele Mentens, „Power and Fault Analysis Resistance in Hardware through Dynamic Reconfiguration,” International Association for Cryptologic Research, KU Leuven, ESAT/SCD-COSIC and IBBT - Kasteelpark Areberg 10, B-3001 Leuven-Heverlee, Belgium, 2008.

[10] V. L. a. K. K. Thomas Roche, „Combined Fault and Side-Channel Attack on Protected Implementations of AES,” in *Smart card Research and Advanced Applications*, Leuven, Springer, 2011, pp. 65-83.

[11] A. M. O. M. T. G. Pascal Sasdrich, „Achieving Side-Channel Protection with Dynamic Logic Reconfiguration on Modern FPGA's,” Horst Görtz Institute for IT-security , Ruhr-Universität Bochum, Germany.

[12] R. Sedgewick, „Permutation Generation Methods," in *Computer Surveys*, Brown University, Providence, rhode Island 02912, 1977, pp. 137-164.

[13] M. S. U. M. A. A. A. Swaleha Seed, „Fisher-Yates Chaotic Shuffling Based Image Encryption," Department of Computer Engineering, ZH College of Engineering and Technology, Aligarh Muslim University, Aligarh 202 002, India, Department of Computer Engineering, Faculty of Engineering and Technology, Jamia Millia Islamia, New Delhi 110 025, India.

[14] F. A. Fisher, in *Statistic tables for biological, agricultural and medical research*, London, Oliver & Boyd, 1938, pp. 26-27.

[15] R. Durstenfeld, „Algorithm 235: Random Permutation," in *Communications of the ACM*, 1964, p. 420.

[16] M. C. Wilson, „Random and Exhaustive Generation of Permutations and Cycles," 2007.

[17] J. Arndt, „Generating Random Permutations," Australian National University, 2009.

[18] „NanGate Open Cell Library," Nangate Inc, [Online]. Available: http://www.nangate.com/?page_id=22. [Geopend 01 03 2016].

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
A random permutation-based method for secure hardware implementations

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2016**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Schrijvers, Robin

Datum: **5/06/2016**