IMPLEMENTATIE EN EVALUATIE VAN SHA-3-KANDIDATEN OP FPGA

Rapport over de masterproef van

Hans NARINX en Wim RAMAKERS

kandidaten voor de graad van Academische Master
Industriële Wetenschappen, Elektronica-ICT

Promotoren:
ir. S. Indesteege
ing. J. Vliegen
dr. ir. N. Mentens

Academiejaar 2009 - 2010
Referentie: E10/ELO/11

Woord vooraf

Met dit eindwerk voltooien we onze opleiding aan de Katholieke Hogeschool Limburg departement Industriële Wetenschappen en Technologie. Daarom vinden we het belangrijk om iedereen te bedanken die bijgedragen heeft tot onze vorming aan de KHLim tot industrieel ingenieur in de elektronica.

Onze dank gaat uit naar de docenten en vooral naar onze ouders die ons alle mogelijkheden gegeven hebben en ons steeds moreel hebben gesteund om onze studies tot een goed einde te brengen.
Graag danken we ook de mensen die bijgedragen hebben tot de realisatie van dit eindwerk. Op de eerste plaats danken we dr. ir. N. Mentens en ing. J. Vliegen, de promotoren van onze school, die ons geholpen hebben gedurende dit eindwerk. Vervolgens danken we ook ir. S. Indesteege, de promotor van ESAT/COSIC en opdrachtgever van dit eindwerk, voor alle feedback en tips die we van hem kregen.
Ten slotte danken we ook nog J. Lievens, voor de begeleiding van dit eindrapport.

Gedurende ons eindwerk deden we kennis en vaardigheden op die verband houden met cryptografisch onderzoek en het optimaal implementeren van digitale schakelingen op FPGA. Dit was een zeer leuke en leerrijke ervaring.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Abstract

Door de sterke evolutie van supercomputers en cryptoanalyse wordt de beveiliging van digitale gegevens een probleem. In de huidige hashfuncties, die vooral zorgen voor de beveiliging van wachtwoorden en het plaatsen van digitale handtekeningen, zijn reeds verschillende zwakheden aan het licht gekomen. Hierdoor is de informatie op onze computers en de data die we versturen niet meer veilig. Daarom heeft het NIST een wereldwijde SHA-3-competitie gestart om een nieuwe hashfunctiestandaard te vinden die wel veilig is in de huidige en toekomstige computerwereld. Deze masterproef heeft als doel een aantal van deze hashfuncties in hardware te implementeren op FPGA en ze te evalueren en te optimaliseren op basis van hun snelheid en oppervlaktegebruik.

Om de implementaties te simuleren, wordt er gebruik gemaakt van software om enerzijds de implementatie te testen op een juiste werking (Modelsim) en anderzijds om de VHDL-code te synthetiseren (Xilinx ISE). Deze gesynthetiseerde VHDL-code kan naar een FPGA verzonden worden waardoor het ook mogelijk is de werking op werkelijke hardware te testen.

De simulaties, met behulp van software, geven een duidelijk beeld van de verschillen tussen de nieuwe hashfuncties wat betreft hun snelheid en oppervlaktegebruik. Ook tonen ze duidelijk het effect van de optimalisaties om de implementatie kleiner en/of sneller te maken.

Summary

Due to the strong improvement of supercomputers and crypto analysis the safety of digital data becomes an issue. In the current hash functions, which are used to encrypt passwords and place digital signatures, several weaknesses are found. This means that there is no guarantee that the data we send and store on our computers is safe. The NIST has started a worldwide SHA-3 competition to find a new hash standard that is safe under current and future conditions in the digital world. This masters thesis wants to implement several new hash functions on FPGA, and evaluate and optimize them with regards to speed and area.

To simulate the implementations, software is used to verify that the implementation works correctly (Modelsim) on the one hand, and on the other hand to synthesize the VHDL code (Xilinx ISE). The synthesized code can be downloaded to an FPGA to test the implementation with real hardware.

The simulations, by software, give us a clear view of the differences between the new hash functions with regards to the speed and the use of area. They also clearly show us the impact of the optimisations, which were done to improve speed and/or reduce the size of area.

Inleiding

Situering

Er wordt steeds meer en belangrijkere digitale informatie verstuurd. Dit gebeurt via kanalen waarbij het relatief gemakkelijk is om data af te luisteren of te vervalsen. Door het gebruik van digitale handtekeningen kan de ontvanger o.a. controleren of de data ook echt van de oorspronkelijke zender komt. Om zo een systeem te implementeren worden o.a. hashfuncties gebruikt.

Een hashfunctie vormt een boodschap van variabele lengte om naar een boodschap met een vaste lengte, de hashwaarde. Wanneer de boodschap verandert zal ook de hashwaarde veranderen. Hashfuncties zijn ongeveer drie ordegroottes sneller dan digitale handtekeningen. Door eerst de hashwaarde van de boodschap te berekenen en hierop de digitale handtekening toe te passen, krijgen we snellere en bovendien veiligere digitale handtekeningen.

Probleemstelling

Uit een onderzoek van het NIST (National Institute of Standards and Technology) is gebleken dat de huidige hashfuncties niet veilig genoeg zijn om in de toekomst te blijven gebruiken. Om een nieuwe, veiligere opvolger voor de huidige hashstandaard ‘SHA-2’ te vinden, organiseerde het NIST een publieke en wereldwijde wedstrijd. Deze SHA-3 competitie ging officieel van start op 11 februari 2007. Met deze masterproef willen we een bijdrage leveren aan de SHA-3 competitie door een aantal hashfunctiekandidaten te implementeren op een FPGA en deze kritisch te bekijken.

Doelstellingen

In deze masterproef worden LANE, Hamsi, ECHO en Luffa geïmplementeerd. De doelstellingen die we willen bereiken zijn:

- De oude implementatie van LANE [2] optimaliseren en herimplementen zodat ze minder oppervlakte gebruikt;
- Hamsi, ECHO en Luffa implementeren en kijken of er mogelijkheden zijn om te optimaliseren naar snelheid en/of oppervlaktegebruik;
- Voor iedere geïmplementeerde hashfunctie een gebruikersinterface maken met een buffer. Deze gebruikersinterface houdt in dat de gebruiker niet enkel de compressiefunctie ter beschikking heeft, maar ook dat de hele hashprocedure van messagepadding tot finalisatie automatisch gebeurt;
- De geïmplementeerde hashfuncties onderling vergelijken en de troeven van iedere implementatie opsommen.

Materiaal en methode

Eerst zal er een algemene literatuurstudie over cryptografie gedaan worden. Hierna zullen we de specificaties van de hashfuncties bestuderen en een hardwaremodel opstellen van de implementatie die we gaan maken. De volgende stappen zijn het implementeren, simuleren en debuggen van de ontwerpen. Daarna bekijken we de implementaties kritisch om eventuele verbeteringen te vinden. Ten slotte worden de geïmplementeerde hashfuncties onderling vergeleken op basis van snelheid en oppervlaktegebruik

1 Literatuurstudie

1.1 Cryptografie

Cryptografie bestaat al sinds er geschreven kon worden. Om geschreven teksten veilig tot bij de bestemmeling te brengen zonder dat anderen de boodschap kunnen lezen, moet er een vorm van cryptografie toegepast worden. Tegenwoordig wordt informatie heel vaak digitaal verzonden. Dit heeft als voordeel dat het veel sneller gaat, maar de mogelijkheden om de boodschap af te luisteren en te vervalsen zijn ook talrijker. De volgende paragrafen geven een aantal methoden om informatie te versleutelen, waarna het nut van hashfuncties in de cryptografie duidelijk wordt. De informatie werd gehaald uit [1],[2],[3] en [4].

1.2 Symmetrische-sleutelalgoritmen

Symmetrische-sleutelalgoritmen hebben als eigenschap dat de boodschap zowel vercijferd als ontcijferd wordt met éénzelfde sleutel. Deze sleutel mag enkel door de zender en ontvanger gekend zijn. Een veelgebruikt algoritme van deze soort is AES [5] (Advanced Encryption Standard) dat sinds 2001 in gebruik is.

1.3 Asymmetrische-sleutelalgoritmen

Asymmetrische-sleutelalgoritmen hebben in tegenstelling tot symmetrische-sleutelalgoritmen een verschillende sleutel voor het vercijferen en het ontcijferen. Een asymmetrisch-sleutelalgoritme wordt ook vaak een publieke-sleutelalgoritme genoemd. Hierbij heeft iedere gebruiker een sleutelpaar. Één van deze sleutels houdt hij/zij geheim (de private sleutel), de andere sleutel mag door iedereen gekend zijn (de publieke sleutel). Wanneer de zender zijn/haar boodschap vercijfert met de publieke sleutel van de ontvanger, kan enkel die ontvanger de boodschap terug ontcijferen. Anderen kunnen de private sleutel van iemand niet afleiden uit zijn publieke sleutel, en dus kan enkel de gewenste ontvanger de boodschap lezen.

Ook biedt deze methode de mogelijkheid om te controleren of de boodschap wel werkelijk is van wie we denken dat ze is. Dit gebeurt met digitale handtekeningen. Wanneer de zender een handtekening plaatst met zijn private sleutel, kan de ontvanger de identiteit van de zender controleren door de handtekening te verifiëren met de publieke sleutel van de zender.

1.4 Hashfuncties

Een hashfunctie vormt een boodschap van variabele lengte om naar een boodschap met een vaste lengte, de hashwaarde. Wanneer de boodschap verandert zal ook de hashwaarde veranderen. Hashfuncties zijn ongeveer drie ordegroottes sneller dan digitale handtekeningen. Door eerst de hashwaarde van de boodschap te berekenen en hierop de digitale handtekening toe te passen, krijgen we snellere en bovendien veiligere digitale handtekeningen.

De veiligheid van dit relatief simpele en snellere systeem volgt enerzijds uit de onomkeerbaarheid, en anderzijds uit de botsingsbestendigheid van hashfuncties. Wanneer een hashwaarde gekend is kan onmogelijk de originele boodschap die deze waarde als uitkomst heeft teruggevonden worden. Ook is het zeer moeilijk om twee boodschappen met dezelfde data te vinden (een botsing). Een hashfunctie wordt veilig beschouwd wanneer ze aan drie voorwaarden voldoet:

1 Preimage resistance
Voor een gegeven hashwaarde moet het moeilijk zijn om een andere boodschap te vinden die dezelfde hashwaarde heeft.

2 Second preimage resistance
Voor een gegeven boodschap moet het moeilijk zijn om een andere boodschap te vinden zodat de hashwaarde van beide boodschappen gelijk is.

3 Collision resistance
Het moet moeilijk zijn om twee verschillende boodschappen met dezelfde hashwaarde te vinden. We noemen dit paar van boodschappen een hashbotsing.
Om *collision resistant* te zijn moet de hashwaarde minstens dubbel zo zoveel bits hebben dan om *preimage resistant* te zijn. Deze stelling kan bewezen worden door de verjaardagenparadox [6] die aantoont dat we voor een n-bit hashresultaat slechts $2^{n/2}$ willekeurige inputs moeten hashen om een paar vinden, terwijl we 2^n pogingen moeten doen om een specifieke hashwaarde te vinden.

In de toekomst zal de rekenkracht steeds blijven toenemen, en zullen systemen die nu veilig zijn relatief snel te vervalsen zijn. Daarom wordt er gezocht naar nieuwe hashfuncties die meer outputbits kunnen genereren. Enkele hiervan worden in detail uitgelegd in het hoofdstuk ‘Implementatie’.

1.5 SHA-3-competitie

Na een aantal onderzoeken naar de veiligheid van de huidige hashfuncties door X. Wang, Dobbertin, Chabaud en Joux heeft het NIST [6] beslist dat de huidige hashfuncties niet veilig genoeg zijn om in de toekomst te blijven gebruiken. Om een nieuwe, veiligere opvolger voor de huidige hashstandaarden SHA-1 en SHA-2 te vinden organiseerde het NIST een publieke en wereldwijde wedstrijd. Het probleem met deze huidige standaarden is niet dat ze onveilig zijn, maar voornamelijk dat er niet veel vertrouwen in is. Deze SHA-3 competitie ging officieel van start op 31 oktober 2008. Iedereen kon toen een eigen ontwerp indienen dat aan een aantal minimale eisen moest voldoen. Een inzending moest een volledig uitgeschreven specificatie, een referentieimplementatie in C en een aantal testresultaten bevatten. Ook moest de hashfunctie 224, 256, 384 en 512 bits als output kunnen genereren en een boodschap met een lengte tot 2^{64} bits als input aanvaarden.

Op de deadline van 31 oktober 2008 waren er 64 voorstellen ingediend waarvan er uiteindelijk 51 aanvaard werden. LANE, Hamsi, ECHO en Luffa , de hashfuncties die in deze masterproef geïmplementeerd worden, waren hierbij. De evaluatiecriteria voor de nieuwe kandidaten zijn veiligheid, kost/performantie en implementatiekarakteristieken

De voornaamste factor bij het evalueren van een cryptografische functie is veiligheid. Hashfuncties kunnen voor vele toepassingen gebruikt worden zoals digitale handtekeningen, message authentication codes, pseudowillekeurige nummergeneratoren, one-way functies voor het onleesbaar maken van paswoordbestanden, ... Al deze toepassingen hebben specifieke veiligheidseisen. Daarom heeft het NIST een lijst met veiligheidseisen opgesteld (FRN-Jan07, Federal Register Notice in January 2007) waarmee de kandidaten getest en vergeleken zullen worden.

De tweede voornaamste factor volgens het NIST is de kost. Het begrip kost houdt de efficiëntie van de berekeningen en het geheugengebruik in. De efficiëntie van de berekeningen hangt zeer sterk samen met de snelheid van het algoritme. Het NIST verwacht van de nieuwe SHA-3 functie een betere performantie dan de SHA-2 familie bij dezelfde veiligheid. Geheugengebruik beperkt zich niet enkel tot RAM-geheugen bij software, ook de grootte van hardware (een bepalende factor voor de keuze van een FPGA) wordt bekeken.

De laatste factor zijn de implementatiekarakteristieken. Bij de inzendingen zitten veel algoritmen die een vernieuwend en interessant design hebben waarin unieke kenmerken zitten die niet aanwezig zijn bij het huidige SHA-2 algoritme. Ook wordt de voorkeur gegeven aan algoritmen met een grotere flexibiliteit. Dit houdt in dat het algoritme gemakkelijk op verschillende platformen geïmplementeerd kan worden. Ten slotte wordt ook de voorkeur gegeven aan simpele, elegante en goed begrijpbare algoritmen om de implementatie en analyse ervan te bevorderen.

Op 24 juli 2009 heeft het NIST op basis van voorgaande criteria veertien kandidaten geselecteerd voor de tweede ronde van de competitie [7]. Dit impliceert echter niet dat de rest slecht is. LANE is hierbij afgevallen, maar HAMSI, ECHO en Luffa zitten wel in de tweede ronde. Wereldwijd krijgen onderzoekers nu weer een jaar de tijd om de verschillende kandidaten te evalueren en te vergelijken.

Het NIST heeft geen verdere uitleg gegeven aan de afvallers. Een mogelijke reden voor het afvallen van LANE is dat het NIST net voor de aankondiging van de kandidaten voor de tweede ronde op de hoogte is gebracht van een paper [8]. Deze paper beschrijft een semi-vrije start botsingsaanval op LANE. Het besluit van de paper was: "*Although these collisions on the compression function do not imply an attack on the hash functions, they violate the reduction proofs of Merkle and Damgård, or Andreeva in the case of Lan However, due to the limited degrees of freedom, a collision attack on the hash function seems to be dicult for full round Lane.*" Hierin staat letterlijk dat de veiligheid van de hashfunctie niet in gevaar is, maar enkel het toenmalige veligheidsbewijs niet klopte. Het is zeer waarschijnlijk, maar slechts een vermoeden, dat het NIST geen tijd meer had om voor de deadline deze paper te controleren, en daarom LANE uit veiligheid heeft laten vallen. [9]

2 Implementatie

In deze masterproef worden alle hashfuncties geïmplementeerd in VHDL wat staat voor VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. Met deze taal wordt hardware beschreven, in tegenstelling tot normale programmeertalen waar we de uit te voeren instructies schrijven.

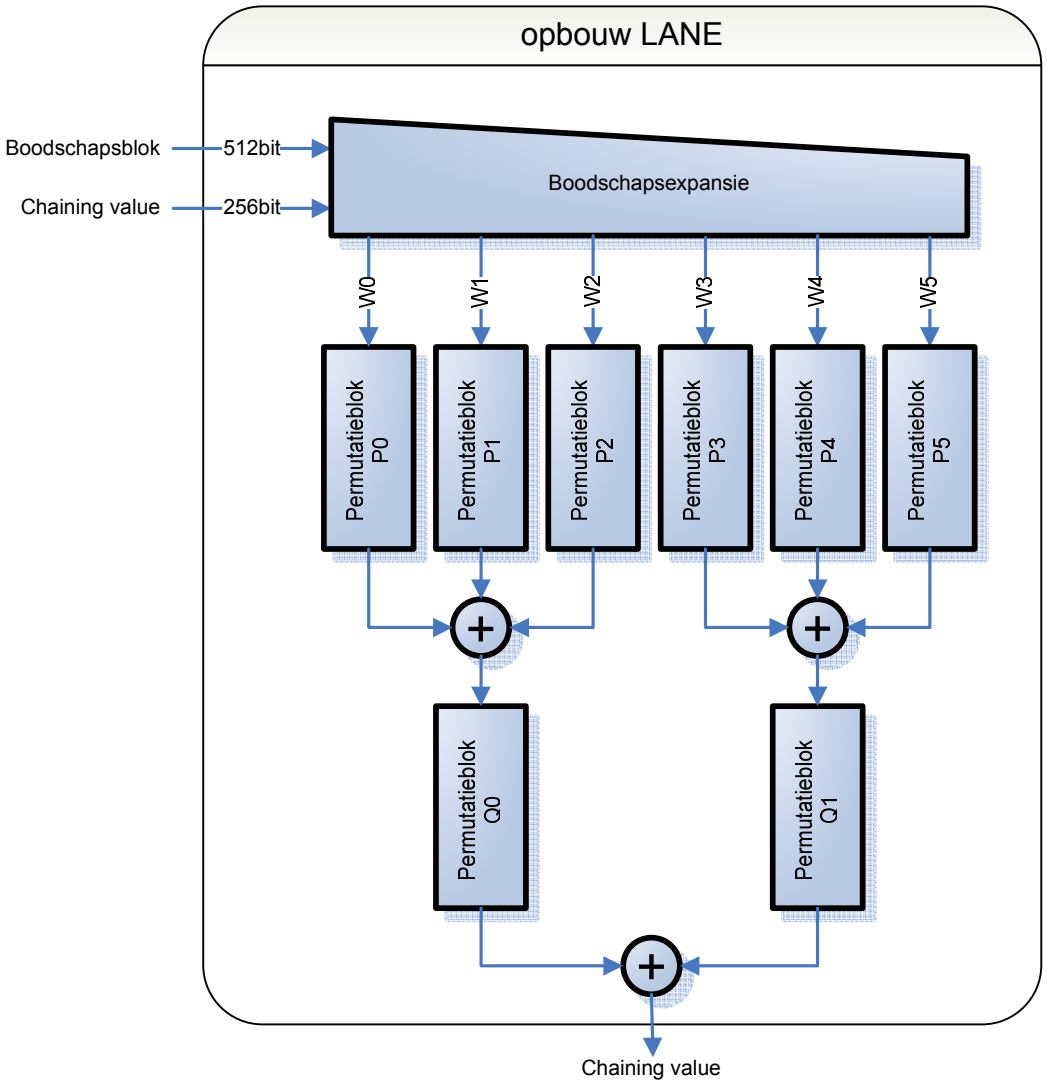
De VHDL-code wordt gesynthetiseerd voor een FPGA (Field Programmable Gate Array). Dit heeft als voordeel dat de hardware aanpasbaar is en gemakkelijk gesimuleerd kan worden op een pc.

In deze masterproef implementeren en evalueren we de hashfuncties LANE, Hamsi, ECHO en Luffa.

2.1 LANE

De hashfunctie LANE [10] is een iteratieve hashfunctie waarvan de compressiefunctie getoond wordt in Figuur 1. We zien dat de binnenkomende boodschap samen met de chaining value (= hashwaarde vorige cyclus) wordt gecombineerd tot een aantal strings w0 – w5. Deze strings ondergaan elk een P-permutatie, waarna de eerste en laatste drie uitkomsten via een exor (exclusieve or poort) van alle overeenkomende bits naar een Q-permutatie gaan. De twee uitkomsten hiervan worden weer via een exor gecombineerd tot de hash- of chainingvalue. Er bestaan verschillende varianten van LANE die een hashwaarde van 224, 256, 384 of 512 genereren. Wanneer we verder in deze masterproef de naam LANE gebruiken, bedoelen we de 256-bit versie, LANE-256.

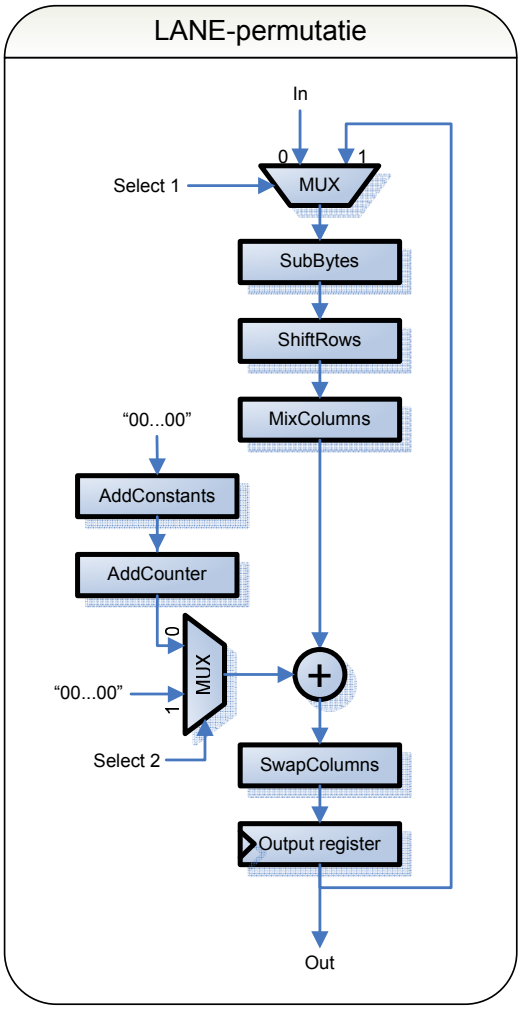
In de volgende paragrafen worden de verschillende schillen van de implementatie uitgelegd. We beginnen met het belangrijkste onderdeel, het permutatieblok, waarna we overgaan tot de volledige compressiefunctie. Hierna volgen de bovenliggende schillen die toelaten de functie praktisch te gebruiken.



Figuur 1: Opbouw van de LANE-256 compressiefunctie

2.1.1 Het permutatieblok

De permutatie van LANE voert een aantal transformaties uit op een interne toestand. LANE is opgebouwd uit een aantal hergebruikte transformaties uit het blokcijfer AES [5] (SubBytes, ShiftRows en MixColumns) en een aantal nieuwe transformaties. Ook de interne toestand is hetzelfde als die van AES, alleen gebruikt LANE twee AES-toestanden naast elkaar. De LANE-toestanden bestaan dus uit twee keer een 4x4 matrix van bytes, in totaal 256 bits. Figuur 2 geeft een grafisch overzicht van de permutatie.

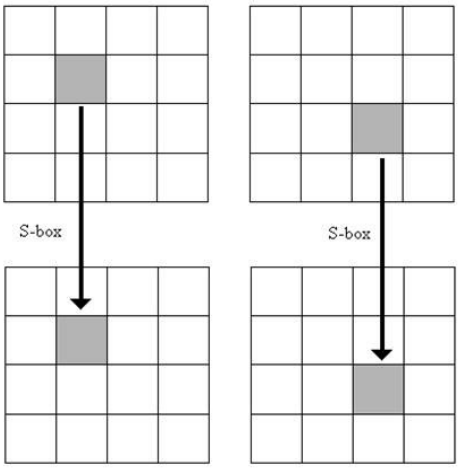


Figuur 2: De LANE-permutatie, een aaneenschakeling van transformaties

De bouwblokken zijn reeds geïmplementeerd in een ander eindwerk [2] en worden in ongewijzigde vorm gebruikt voor de nieuwe implementatie. De beschrijvingen van deze blokken komen ook uit dat eindwerk.

2.1.1.1 SubBytes

De SubBytestransformatie is identiek aan de overeenkomstige component van het AES blokcijfer, alleen wordt ze toegepast op een grotere toestand. Figuur 3 geeft de niet-lineaire substitutie (S-box) weer die onafhankelijk op elke byte wordt toegepast. Dit is dezelfde S-box als die het het AES-blokcijfer (Tabel 1).



Figuur 3: De SubBytestransformatie van LANE-256
Bron: [10]

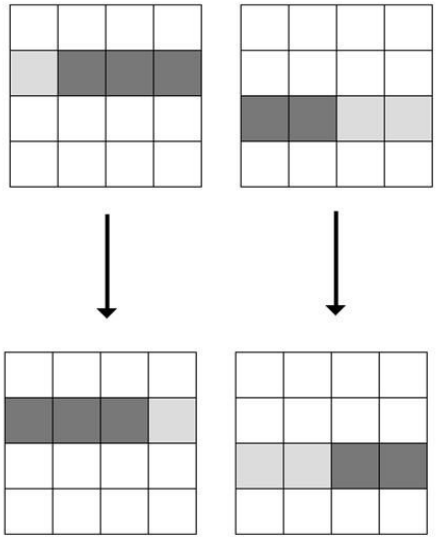
Tabel 1: De AES S-box in hexadecimale notatie

	_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

Bron: [2]

2.1.1.2 ShiftRows

De ShiftRowstransformatie schuift cyclisch met de bytes van elke AES-toestand waaruit een LANE-toestand bestaat. De bovenste rij wordt niet verschoven. De tweede, derde en vierde rij worden cyclisch repectievelijk een, twee en drie byteposities naar links geschoven. Dit is identiek aan wat er in de ShiftRowstransformatie van het AES-blokcijfer gebeurt, behalve dat het hier twee keer in parallel gebeurt. Figuur 4 illustreert ShiftRows voor LANE-256.



Figuur 4: ShiftRowstransformatie van LANE-256
Bron: [10]

2.1.1.3 MixColumns

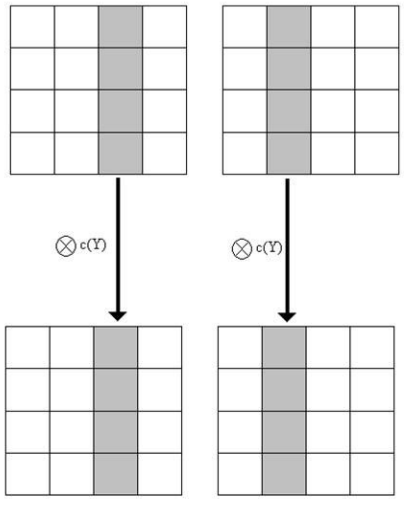
De MixColumnstransformatie beschouwt de kolommen van de interne LANE-toestand als veeltermen in GF(2^8). Als we de elementen van een kolom van boven naar onder benoemen als y_0, y_1, y_2 *en* y_3, dan krijgen we de volgende veelterm:

$$y_3 \cdot Y^3 + y_2 \cdot Y^2 + y_1 \cdot Y^1 + y_0$$

Deze veelterm wordt vervolgens, modulo $Y^4 + 1$, vermenigvuldigd met de constante veelterm $c(Y)$:

$$c(Y) = 3 \cdot Y^3 + 1 \cdot Y^2 + 1 \cdot Y^1 + 2$$

Deze bewerking wordt op elke kolom van de interne toestand toegepast, zoals weergegeven in Figuur 5.



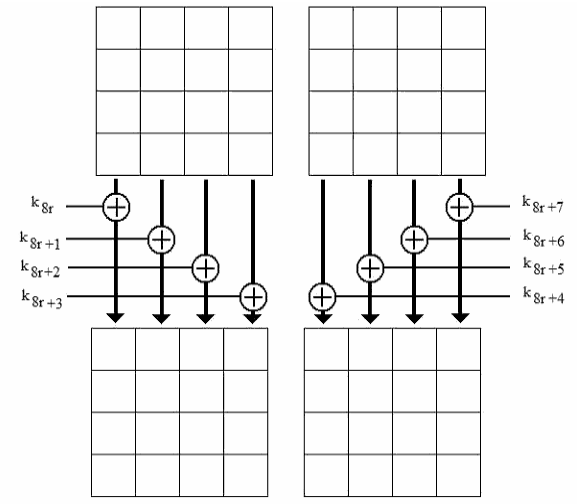
Figuur 5: De MixColumnstransformatie van LANE-256
Bron: [10]

2.1.1.4 AddConstants

De AddConstantstransformatie telt een 32-bit constante op bij elke kolom van een toestand. Deze constanten zijn gegenereerd door middel van een linear feedback shift register (LFSR), in pseudocode beschreven in Figuur 6. In Figuur 7 is de AddConstantstransformatie in formulevorm weergegeven. Bijlage A geeft de gegenereerde waarden van de constanten voor LANE 256.

$$
\begin{aligned}
&k_0 \leftarrow 0x07fc703d \\
&for\ i = 1\ to\ 272\ do \\
&\{ \\
&\quad k_i = k_{i-1} >> 1 \\
&\quad if\ k_{i-1} \wedge 0x00000001\ then \\
&\quad \{ \\
&\qquad k_i = k_i \oplus 0xd0000001 \\
&\quad \} \\
&\}
\end{aligned}
$$

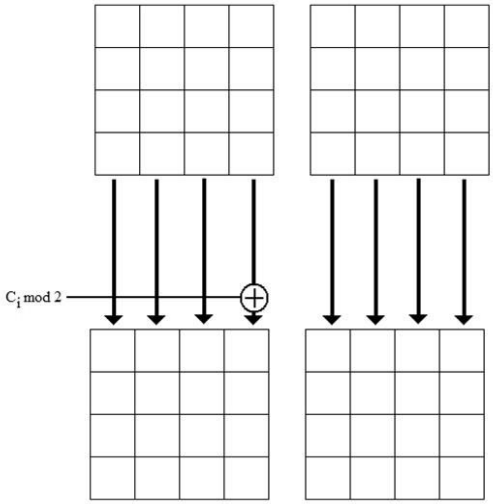
Figuur 6: De pseudocode voor het genereren van de LANE-constanten
Bron: [10]



Figuur 7: De AddConstantstransformatie
Bron: [10]

2.1.1.5 AddCounter

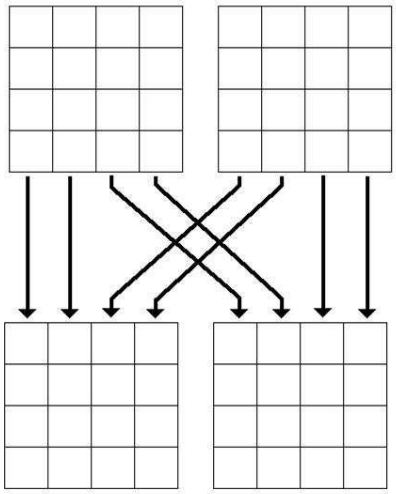
De AddCountertransformatie telt een deel van de teller C op bij de LANE-toestand. De 64 bit teller C wordt opgesplitst in twee 32-bitwoorden C_0 en C_1 , met C_0 het meest beduidende en C_1 het minst beduidende deel van C. Afhankelijk van de rondeteller r wordt dan C_0 of C_1 bij de vierde kolom van de LANE-toestand opgeteld. Als r even is, wordt C_0 bijgeteld, wanneer r oneven is wordt C_1 bijgeteld. Dit wordt grafisch weergegeven in Figuur 8.



Figuur 8: De AddCountertransformatie
Bron: [10]

2.1.1.6 SwapColumns

De SwapColumnstransformatie gaat de kolommen van de LANE-toestand opnieuw ordenen. Dit zorgt ervoor dat de AES-toestanden waaruit de LANE-toestand gemaakt is onderling gemengd worden. Figuur 9 toont hoe deze kolomverwisseling precies in zijn werk gaat.



Figuur 9: De SwapColumnstransformatie
Bron: [10]

2.1.2 De compressiefunctie

De LANE compressiefunctie bevat, naargelang de implementatie, één of meerdere permutatieblokken (zoals beschreven in de vorige paragraaf), een boodschapsexpansieblok en een aantal EXOR-poorten.

2.1.2.1 Het boodschapsexpansieblok

Ook dit blok wordt gebruikt zoals het in het eindwerk [2] ontworpen is, en bevat enkel EXOR-poorten. Het aangeboden boodschapsblok wordt opgesplitst in vier gelijke delen, de *chaining value* in twee gelijke delen. Hiermee worden zes combinaties gemaakt van elk 256 bits. In Figuur 10 zien we de juiste combinaties van 256-bit woorden.

$$W_0 = h_0 \; xor \; m_0 \; xor \; m_1 \; xor \; m_2 \; xor \; m_3 \| \; h_1 \; xor \; m_0 \; xor \; m_2$$
$$W_1 = h_0 \; xor \; h_1 \; xor \; m_0 \; xor \; m_2 \; xor \; m_3 \| \; h_0 \; xor \; m_1 \; xor \; m_2$$
$$W_2 = h_0 \; xor \; h_1 \; xor \; m_0 \; xor \; m_1 \; xor \; m_2 \| \; h_0 \; xor \; m_0 \; xor \; m_3$$
$$W_3 = h_0 \| h_1$$
$$W_4 = m_0 \| m_1$$
$$W_5 = m_2 \| m_3$$

Figuur 10: Boodschapsexpansie LANE
Bron: [8]

2.1.2.2 De permutatieblokken

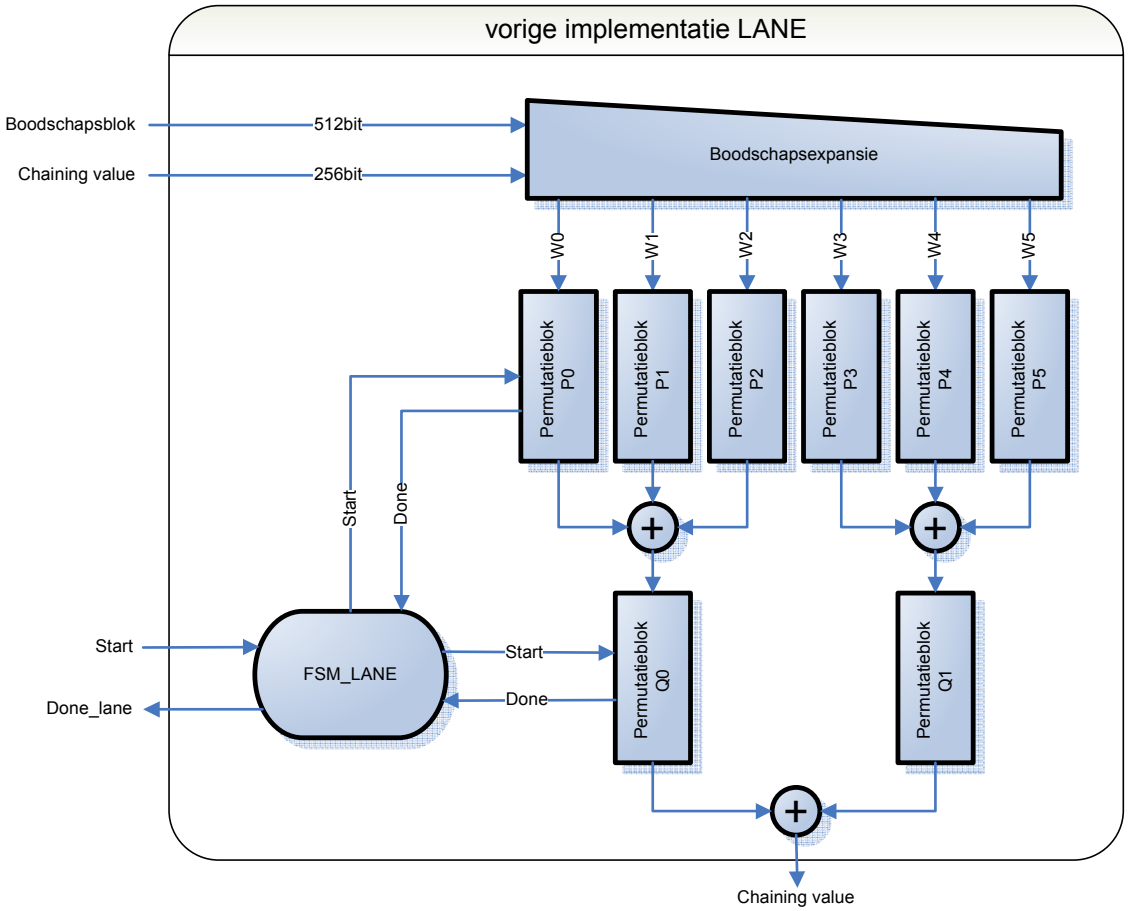
We kunnen twee types onderscheiden: het P- en Q-permutatieblok. Ze verschillen van elkaar omdat het aantal keren dat de permutatie wordt doorlopen en de waarden die moeten worden opgeteld anders zijn. Afhankelijk van de implementatie worden ze als twee verschillende blokken [2] of als één enkele, uitgebreide blok, geïmplementeerd. Hier gaan we dieper op in in de volgende alinea's.

De implementatie uit een vorig eindwerk [2] is een snelle, maar tevens zeer grote implementatie. De nieuwe implementatie in dit eindwerk is het tegengestelde. Ze is zo klein mogelijk gemaakt, wat wel tot gevolg heeft dat de snelheid een stuk lager is.

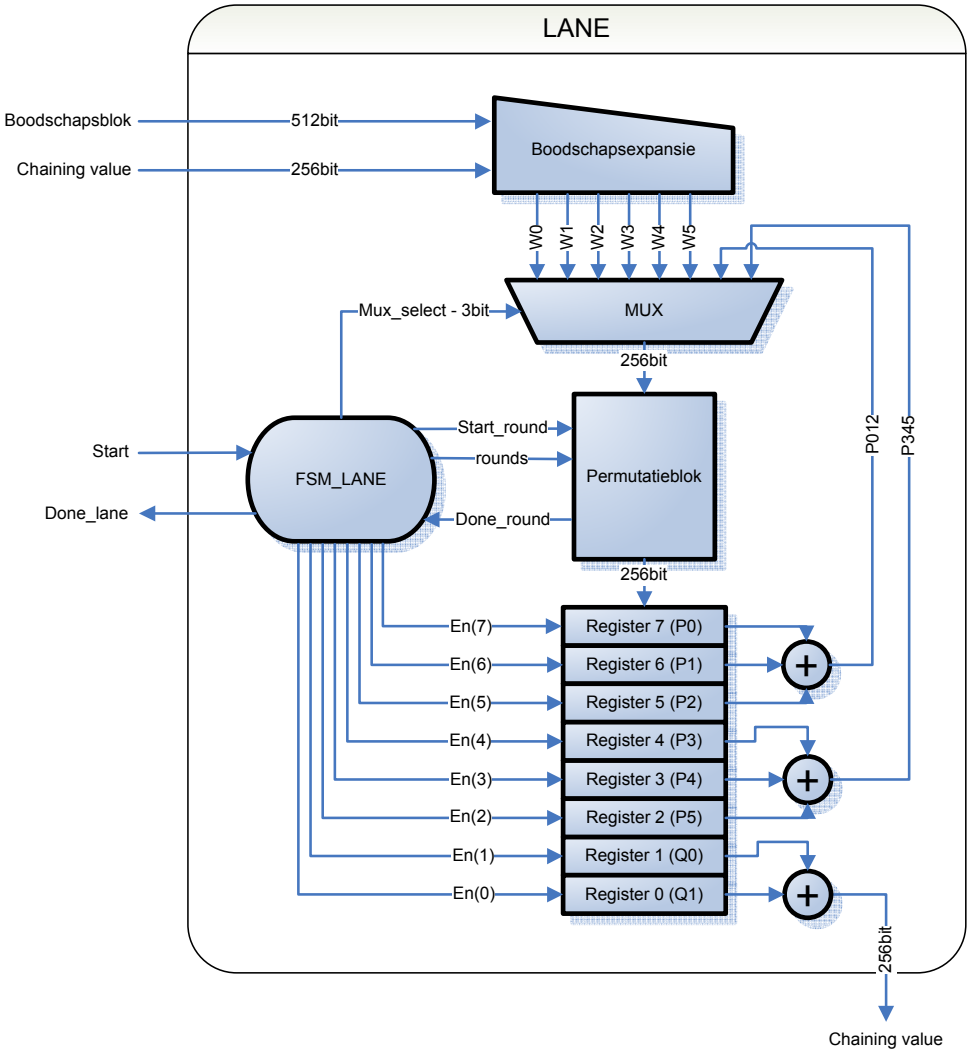
Bij de snelle versie wordt alle hardware gegenereerd zoals hij beschreven staat in de LANE-specificaties (zie Figuur 11). Zo zijn er zes blokken die dezelfde P-permutatie doen en twee blokken die exact dezelfde Q-permutatie kunnen doen. Door registers te gebruiken, om de output bij te houden, kunnen we met één P-permutatieblok, één Q-permutatieblok, acht registers en twee multiplexers de volledige compressiefunctie realiseren. Wanneer we echter nog dieper ingaan op de inhoud van de twee permutatieblokken, zien we dat ze bijna

hetzelfde doen. Het verschil zit enkel in het aantal keer dat de permutatie wordt doorlopen. Voor een P-permutatie is dat zes, voor een Q-permutatie slechts drie. Door een extra ingang te voorzien op een algemene, aangepaste permutatieblok kunnen we met slechts één blok hardware beide permutaties doorlopen. De zes P-permutatieblokken en twee Q-permutatieblokken worden dus vervangen door één permutatieblok, één multiplexer en acht registers.

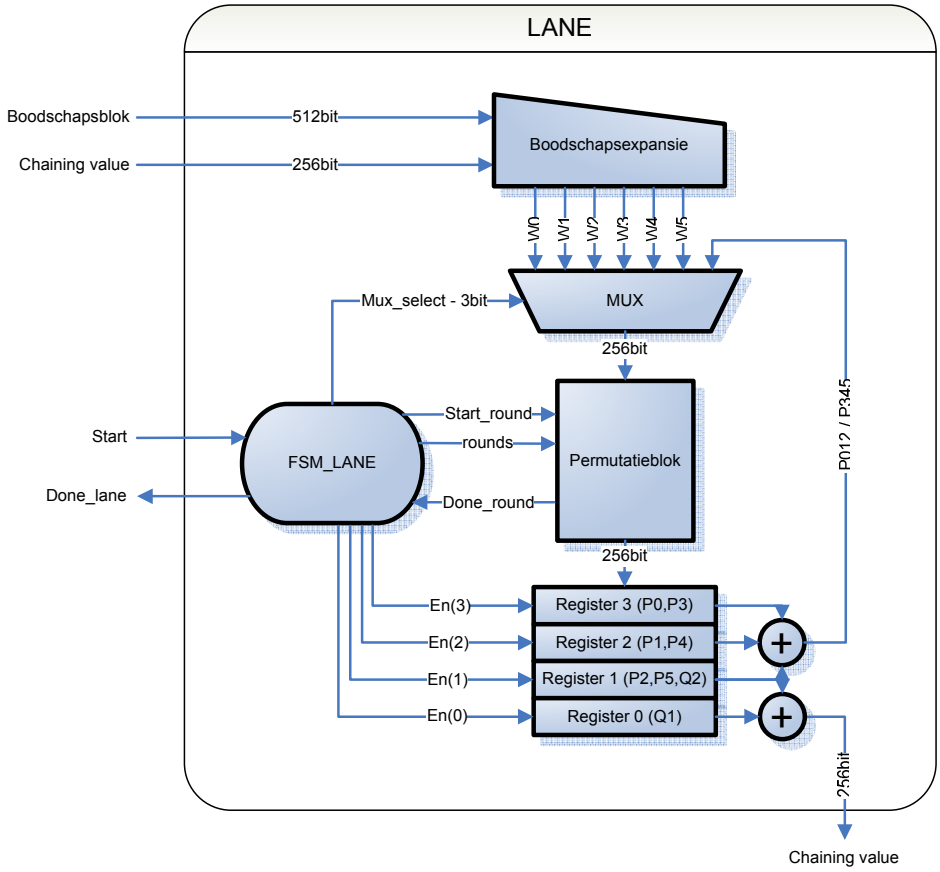
We zien nu op Figuur 12 dat we twee maal een EXOR van drie registeruitgangen hebben. De waarden die in de eerste zes registers staan zijn slechts tussenresultaten en het is dus niet nodig deze bij te houden. Wanneer we de drie eerste P-permutaties doorlopen hebben en de uitkomsten hiervan in drie registers hebben geschreven, kunnen we al direct de eerste Q-permutatie uitvoeren. Deze uitkomst moeten we in een extra, vierde, register wegschrijven. De uitkomsten van de volgende drie P-permutaties mogen nu weer in dezelfde eerste drie registers geschreven worden. De uitkomst van de tweede Q-permutatie kan nu ook in één van die drie registers geschreven worden. Zo hebben we vier registers en een EXOR -poort minder nodig, terwijl de snelheid hetzelfde blijft. Figuur 13 geeft een overzicht van de nodige bouwblokken en verbindingen voor deze laatste implementatie.



Figuur 11: De oude LANE-implementatie

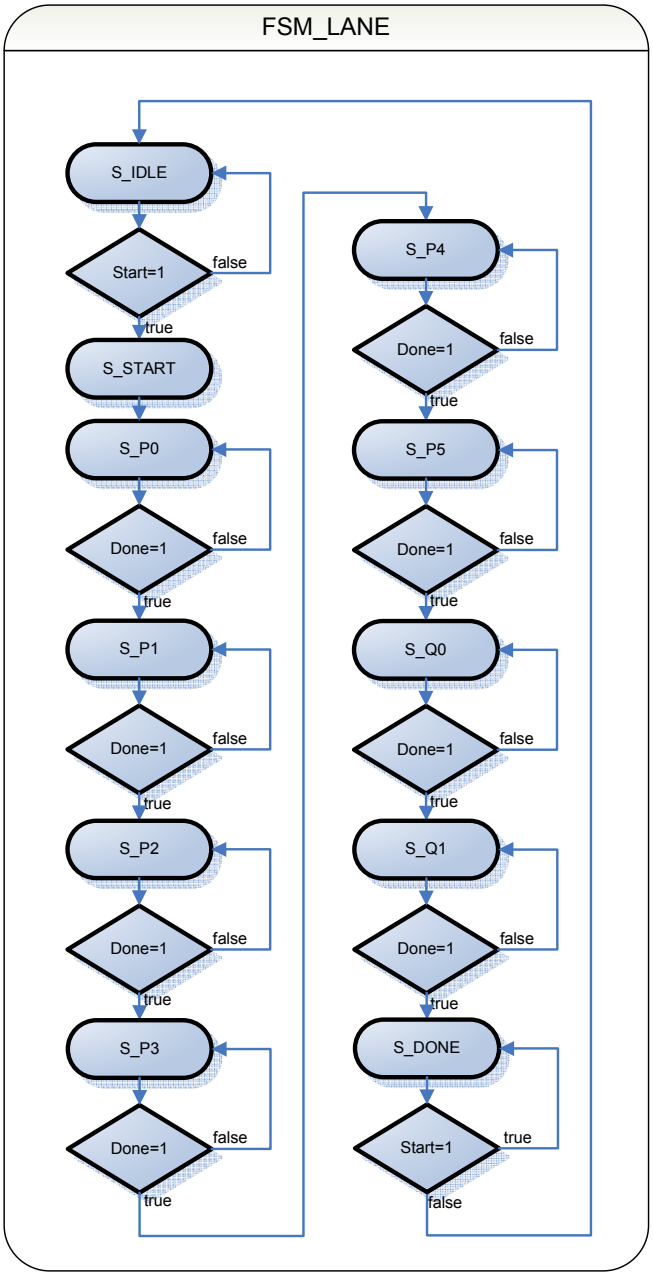


Figuur 12: De nieuwe LANE-implementatie: hergebruik van de permutatieblok



Figuur 13: De nieuwe LANE-imlementatie: zo klein mogelijk

De FSM (Finite State Machine) die de permutatieblok en de registers aanstuurt moest volledig opnieuw gemaakt worden. In Figuur 14 wordt het stroomdiagram van FSM_LANE weergegeven, waarop de verschillende stappen van de FSM gemakkelijk gevolgd kunnen worden.



Figuur 14: FSM_LANE

De FSM wacht in de idle toestand totdat het startsignaal hoog wordt. Hierna gaat hij over naar de start toestand waar alles in orde wordt gebracht om de eerste permutatie te doorlopen. Dit houdt in dat de multiplexer juist wordt aangestuurd, de enable-ingang van het juiste register hoog gemaakt wordt en het juiste sectienummer en aantal rondes wordt aangeboden aan de permutatieblok. In alle volgende toestanden gebeurt telkens hetzelfde: de permutatie wordt gestart en er wordt gewacht op het done signaal van het permutatieblok. Wanneer het done signaal hoog wordt zal de FSM niet alleen overgaan naar de volgende toestand, maar ook de signalen juist zetten voor de volgende permutatie (zoals in de start toestand). De signalen die de FSM steeds moet aansturen zijn:

- Mux_select: geeft aan welk signaal als input op het permutatieblok moet komen;
- Start_round: het signaal om de permutatie te starten;
- Rounds: signaal om aan te geven hoeveel ronden de permutatie telt, ‘1’=6 en ‘0’=3;
- En(3) – en(0): De signalen om de registers te enablen.

Het done_round signaal geeft aan wanneer het permutatieblok klaar is. In de laatste toestand, S_DONE, blijft de FSM wachten totdat het startsignaal terug laag wordt. Zolang dit niet het geval is, blijft het done signaal hoog en blijven we in dezelfde toestand. Deze toestand is enkel toegevoegd om te voorkomen dat er direct een nieuwe hashprocedure start wanneer de gebruiker het start signaal hoog laat staan. Wanneer deze extra toestand weggelaten wordt, hebben we een klokpuls minder nodig om de compressiefunctie te doorlopen.

Na synthese met ISE, een ontwikkeltool van Xilinx, krijgen we volgende gegevens die ons toelaten de compressiefuncties te vergelijken wat betreft snelheid en oppervlakte:

Tabel 2: Vergelijking tussen implementaties van de LANE-compressiefunctie

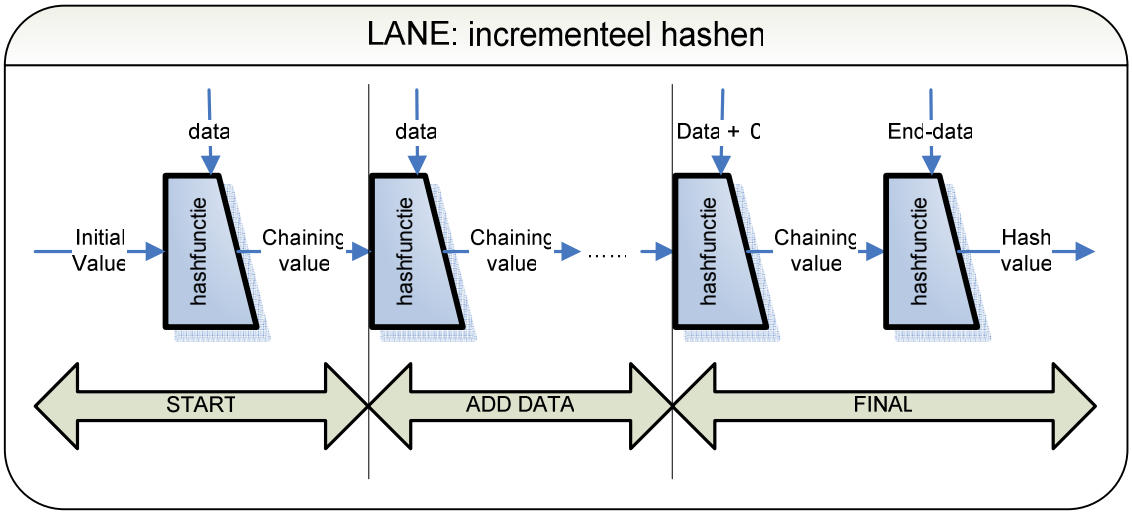
Xilinx Virtex-5 FPGA	**LANE_oud [2]**	**LANE_nieuw**	**LANE [11]**
Aantal slices	*8228*	*3389*	*3442*
Aantal Slice registers	*2111*	*1307*	*/*
Aantal slice LUT's	*16600*	*4078*	*/*
Minimale periode	*4.067 ns*	*5.622 ns*	*7.519 ns*
Geschatte maximale klokfrequentie	*245 MHz*	*177 MHz*	*133 MHz*
Aantal klokcycli per compressie	*15*	*61*	*49*
Tijd voor 1 compressie	*61 ns*	*343 ns*	*369 ns*
Doorvoercapaciteit	*8.36 Gbps*	*1.48 Gbps*	*1.38 Gbps*
Doorvoercapaciteit /oppervlakte	*1.01 Mbps/slice*	*0.43 Mbps/slice*	*0.4 Mbps/slice*

We kunnen zien dat de nieuwe implementatie van LANE duidelijk veel minder oppervlakte nodig heeft dan de oude (2.4 keer minder). De minimale periode is iets groter en de maximale klokfrequentie dus kleiner. Ook het aantal klokcycli is 4 keer groter bij de nieuwe implementatie.

De nieuwe implementatie kan ook vergeleken worden met een andere implementatie van LANE van Baldwin et al [11], die op analoge wijze opgebouwd is (ook met slechts één permutatieblok). We zien dat de doorvoercapaciteit en het aantal slices ongeveer hetzelfde zijn, maar dat de nieuwe implementatie toch iets beter is op beide vlakken. We zien wel dat de kloksnelheid van de nieuwe implementatie hoger is, wat voor een groot stuk te danken is aan een korter kritiek pad (overeenkomstig met Figuur 2). Dit komt omdat dat de constanten en de teller via een EXOR toegevoegd worden in plaats van ze als extra transformatie uit te voeren. We hebben de vertraging van de multiplexer, AddCounter- en AddConstantstransformatie dus vervangen door de vertraging van een EXOR. De reden dat het aantal klokcycli hoger is bij onze nieuwe implementatie, heeft waarschijnlijk te maken met het opsplitsen van logica, wat tot gevolg heeft dat de maximale frequentie omhoog gaat.

2.1.3 De testopstelling

Één van de doelen van deze masterproef is de interfacing naar de buitenwereld te verbeteren. Bij de testopstelling van een vorig eindwerk [2] moest de te hashen data in één groot blok aangeleverd worden om hem daarna ook als één blok in het RAM-geheugen van de FPGA te laden. Dit principe laat niet toe om incrementeel te hashen (Figuur 15), hetgeen zeer handig is als men van tevoren niet weet hoe lang de te hashen data gaat zijn (denk aan streaming van data) of wanneer er erg veel data gehashed moet worden.



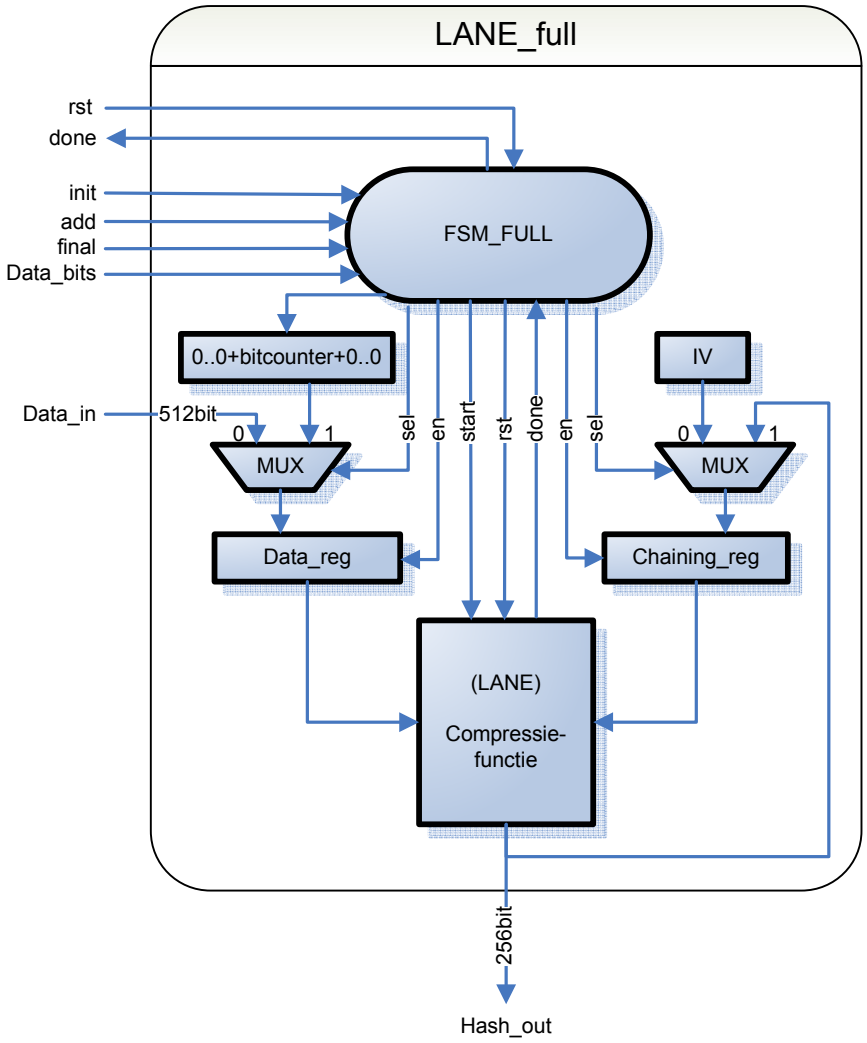
Figuur 15: Incrementele hashprocedure voor LANE

Het hardwareblok (zie Figuur 16) dat één niveau hoger dan de compressiefunctie staat gebruikt één compressieblok en handelt alle drie de fasen van het hashproces af.

Initialisatie
De gebruiker moet het systeem initialiseren alvorens hij/zij kan beginnen hashen. In plaats van de vorige hashwaarde als chaining value te gebruiken, wordt er een initialisatievector aangelegd. De hexadecimale waarde hiervan is be292e17bb541ff2fe54b6f730b1c96a7b2592688539bdf397c4bdd649763fb8.

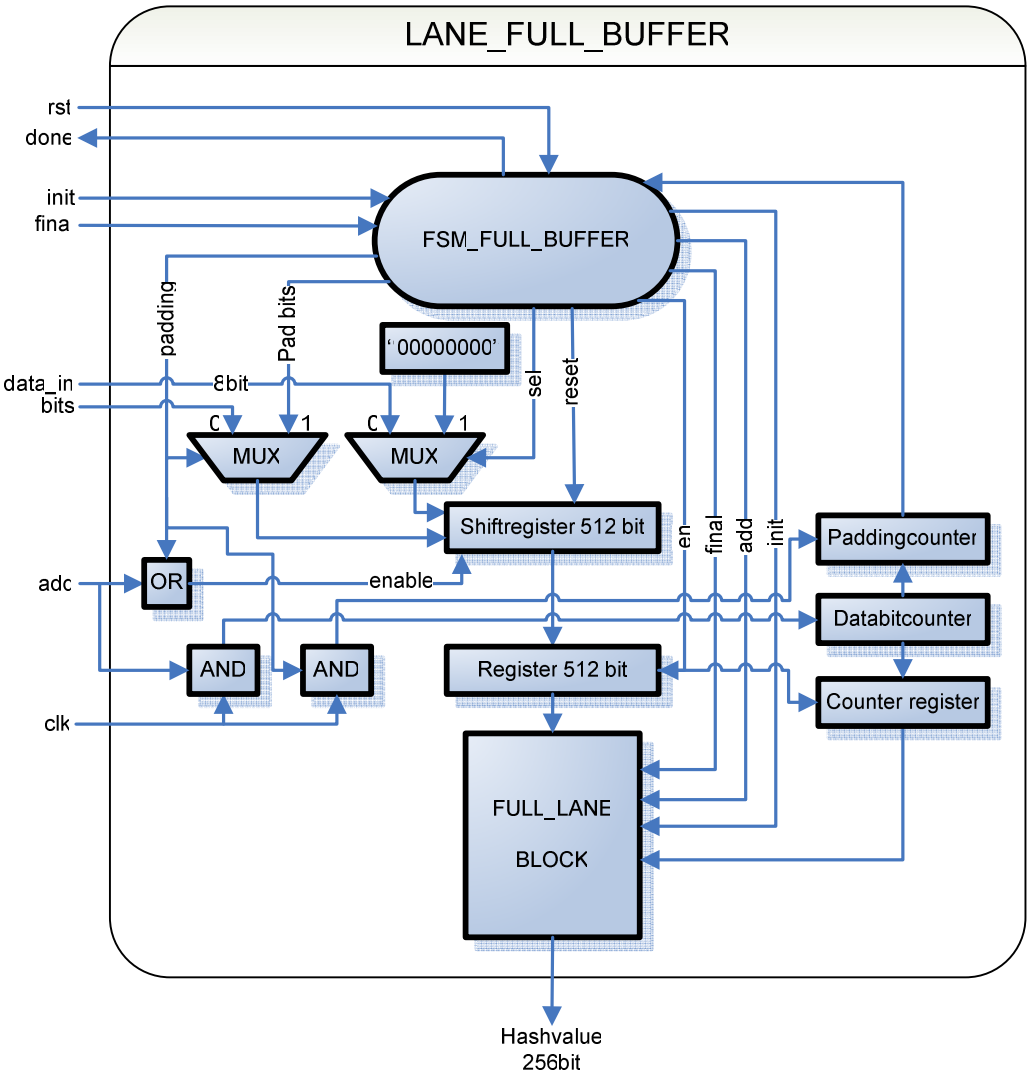
Toevoegen van data
Na de initialisatie kan de gebruiker datablokken van 512 bit toevoegen. Een interne teller houdt bij hoeveel bits er in totaal al gehasht zijn. Dit is nodig voor de volgende fase.

Message padding en output transformation
Wanneer de gebruiker klaar is met het toevoegen van data, kan hij/zij de finale hashwaarde laten berekenen. Dit gebeurt door intern een extra datablok te hashen waar als belangrijkste gegeven de lengte van de boodschap in zit. Voor onze implementatie, waarbij we geen salt gebruiken, ziet deze boodschap er als volgt uit: 00000000 || bitteller || 00.. (440x) ..00.



Figuur 16: Volledige implementatie van LANE (LANE_FULL)

In de nieuwe implementatie is er op één niveau hoger ook een databuffer voorzien (zie Figuur 17). Deze dient om het gebruik van de compressiefunctie voor de gebruiker te vergemakkelijken. De data hoeft dan niet meer in blokken van 512 bit toegevoegd te worden, maar kan per bit of per byte toegevoegd worden. De gebruiker hoeft zelf niet te controleren wanneer een volgende datablok gehashed kan worden. Bij iedere stijgende klokflank, waarbij ‘add’ hoog is, wordt de aangeboden bit of byte toegevoegd in een schuifregister. Ook wordt er voor dat schuifregister een teller bijgehouden voor het aantal bits. Wanneer deze teller op 512 komt, wordt het datablok automatisch gehashed terwijl de gebruiker bij de volgende klokpuls al nieuwe data kan toevoegen. Wanneer we de finale hashwaarde willen, maken we ‘final’ hoog en zal de buffer automatisch aangevuld worden met nullen tot 512 bit.



Figuur 17: Volledige implementatie van LANE, inclusief databuffer (LANE_FULL_BUFFER)

2.2 Hamsi

Hamsi is een familie van cryptografische hash functies [12]. Er zijn meerdere instanties van Hamsi. Hamsi-224 en Hamsi-384 zijn gelijkaardig aan respectievelijk Hamsi-256 en Hamsi-512. Het enige verschil zit in de gebruikte initial values en de uiteindelijke truncatie. Tabel 3 toont de verschillende varianten. Hierbij is P het aantal keer dat de normale niet-lineaire permutatie uitgevoerd wordt, en Pf (P final) het aantal keer dat de laatste wordt uitgevoerd. De lengte is telkens weergeven in aantal bits.

Tabel 3: De verschillen tussen de varianten van Hamsi

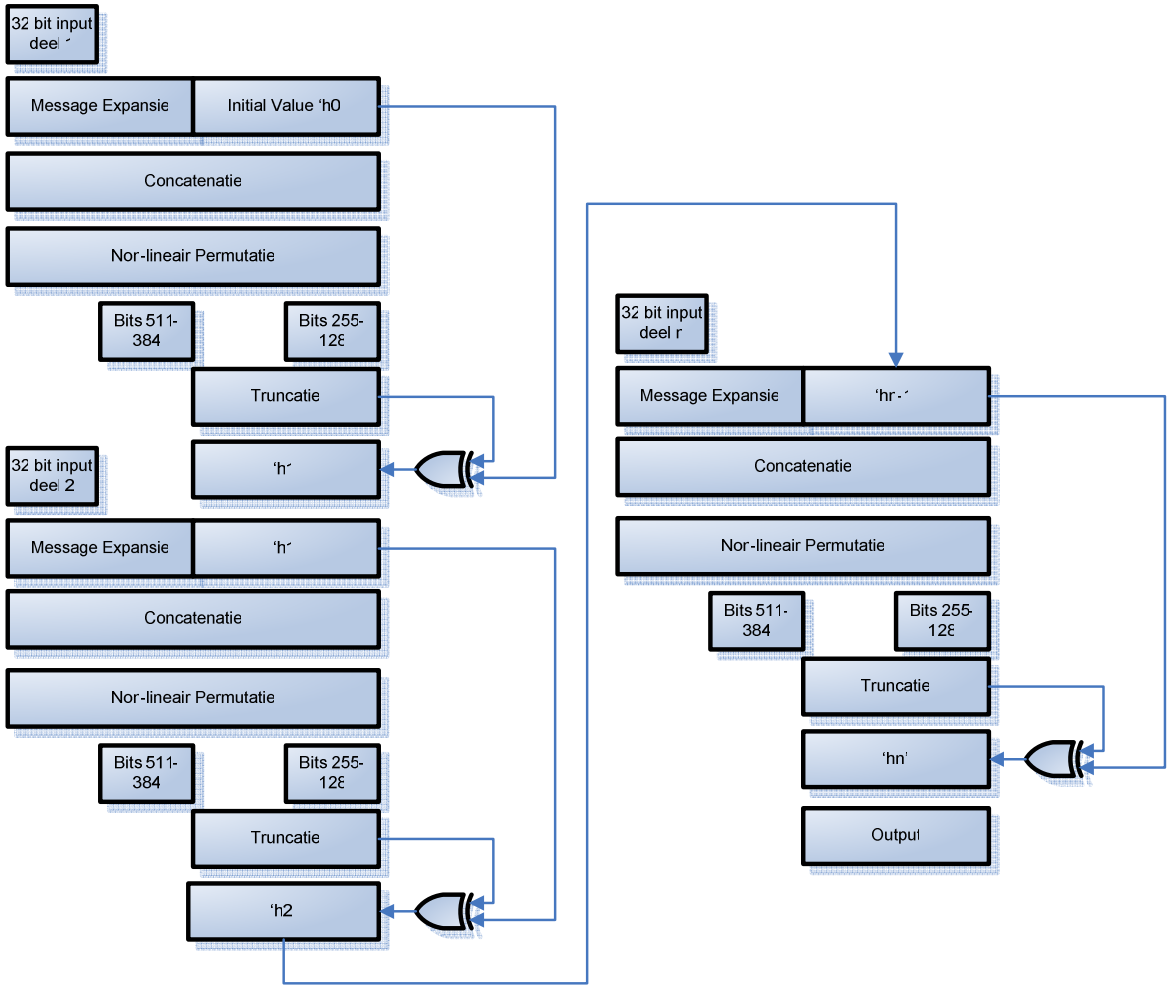
Variant	Lengte van de Hashwaarde	Lengte van de blokken aan de input	Lengte van de inital value	Aantal rondes in P,Pf
Hamsi-256	256	32	256	3, 6
Hamsi-512	512	64	512	6, 12
Hamsi-224	224	32	256	3, 6
Hamsi-384	384	64	512	6, 12

Wij gaan enkel Hamsi-256 implementeren dus is hier ook enkel de initial value van Hamsi-256 van belang.

IV256 = 0x76657273697465697420 4c657576656e2c204b6174686f6c69656b6520556e69

2.2.1 Dataflow van Hamsi-256

Om de dataflow van Hamsi-256 goed te kunnen begrijpen is er in Figuur 18 een globaal overzicht gegeven van de verschillende stappen die uitgevoerd moeten worden.



Figuur 18: Dataflow van Hamsi

In dit schema is te zien dat ieder boodschapblok op dezelfde manier wordt gecodeerd. Enkel de gebruikte waardes verschillen telkens. In wat volgt gaan we dieper in op de werking van het geheel, en elke bouwblok.

2.2.2 Bespreking van de verschillende stappen.

De verschillende stappen die uitgevoerd moeten worden zijn:

- Boodschap in de buffer zetten;
- Message padding;
- Message expansie;
- Concatenation;
- Niet-lineaire permutatie.

2.2.2.1 Boodschap in de buffer zetten

De boodschap die verwerkt moet worden door de hashfunctie 'Hamsi-256' zal verwerkt worden in stukken van 32 bits. Deze 32 bits komen over één lijn serieel, bit per bit binnen in de implementatie. Hierdoor is het gebruik van een buffer dus noodzakelijk. Dit heeft als voordeel dat er pas bij het laatste stuk , indien nodig, message padding toegepast moet worden. De lengte van de boodschap hoeft dus niet op voorhand gekend te zijn.
De boodschap stroomt binnen in de buffer. Als deze vol is, zal de inhoud van de buffer doorgestuurd worden om door de hashfunctie verwerkt te worden. Op hetzelfde moment zal de teller die telt hoeveel bits de buffer al binnengestroomd zijn gereset worden. De buffer kan nu terug bits in beginnen te lezen.

2.2.2.2 Message padding

Wanneer het laatste deel van de boodschap kleiner is dan 32 bits zal de buffer niet meer vol geraken. De boodschap zal dus ook niet meer verwerkt worden door de hashfunctie. Om dit toch mogelijk te maken zullen we bits moeten toevoegen tot de buffer vol is. We zullen dit doen door eerst een '1'-bit toe te voegen, gevolgd door '0'-bits tot de buffer gevuld is. Door het toevoegen van deze '1'-bit weten we altijd waar het einde van de eigenlijke boodschap is.
Wanneer de buffer toch vol geraakt en de boodschap dus een veelvoud van 32 bits groot is, zal er toch nog message padding toegepast worden. De message padding zal er dan voor zorgen dat de boodschap "10000…000" toegevoegd wordt.

Na de message padding zullen er nog 2 blokken data worden toegevoegd die ons vertellen hoeveel bits de volledige boodschap telt. Hierdoor is de maximale grootte van de boodschap beperkt tot 2^{64} bits.

2.2.2.3 Message expansie.

Uit de buffer komen telkens blokken van 32 bits. Door message expansion toe te passen zal zo een blok omgevormd worden tot een blok van 256 bits. Dit kan doordat iedere bit van het te bekomen 256 bit blok via een aantal xor-poorten in verbinding staat met een een aantal bits van het 32 bit blok.

2.2.2.4 Concatenatie

Nadat de message expansion gebeurd is, hebben we twee blokken van 256 bits, namelijk het blok uit de message expansion en de chaining value. Deze twee blokken zullen we dus moeten samenvoegen. De initial chaining value kunnen we in het voorgaande schema zien als 'h0'. Dit is een vaste waarde die met de eerste boodschapblok zal samengevoegd worden. De volgende chaining values zijn de xor van de vorige chaining value en de output van de niet-lineaire permutatie, die door die vorige chaining value bekomen werd.
De twee blokken worden elk als volgt opgedeeld in gelijke stukken van 32 bits.

$M -> m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7$

$C -> c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7$

Vervolgens worden deze stukken in een matrix geplaatst met grootte 4 x 4 (zie Tabel 4).

Tabel 4: Matrix na het uitvoeren van de concatenation, C is hier de chaining value.

m_0	m_1	c_0	c_1
c_2	c_3	m_2	m_3
m_4	m_5	c_4	c_5
c_6	c_7	m_6	m_7

De boodschap is nu klaar om verwerkt te worden door de niet-lineaire permutatie.

2.2.2.5 Niet-lineaire permutatie

De niet-lineaire permutatie werkt op haar beurt ook weer in verschillende stappen.

a) Xor met constanten en teller

Als eerste worden alle elementen uit de matrix ge-xor-t met een aantal constanten. Element m1 wordt nog eens extra ge-xor-t met de waarde van de teller die bijhoudt hoe vaak de permutatie al is uitgevoerd. De eerste keer staat de teller op 0, de tweede keer op 1, enz. In tabel 5 is te zien welke constanten er gebruikt worden voor de gewone niet-lineaire permutatie en in tabel 6 is te zien welke constanten dit zijn voor de laatste niet-lineaire permutatie. We gebruiken hier de waardes a0-a3, a8-a11, a16-a19 en a24-a27. De andere waardes worden gebruikt in de extra kolommen van de matrices in Hamsi-384 en Hamsi-512.

Tabel 5: Constanten bij P

$\alpha_0 = 0xff00f0f0$	$\alpha_1 = 0xccccaaaa$	$\alpha_2 = 0xf0f0cccc$	$\alpha_3 = 0xff00aaaa$
$\alpha_4 = 0xccccaaaa$	$\alpha_5 = 0xf0f0ff00$	$\alpha_6 = 0xaaaacccc$	$\alpha_7 = 0xf0f0ff00$
$\alpha_8 = 0xf0f0cccc$	$\alpha_9 = 0xaaaaff00$	$\alpha_{10} = 0xccccff00$	$\alpha_{11} = 0xaaaaf0f0$
$\alpha_{12} = 0xaaaaf0f0$	$\alpha_{13} = 0xff00cccc$	$\alpha_{14} = 0xccccf0f0$	$\alpha_{15} = 0xff00aaaa$
$\alpha_{16} = 0xccccaaaa$	$\alpha_{17} = 0xff00f0f0$	$\alpha_{18} = 0xff00aaaa$	$\alpha_{19} = 0xf0f0cccc$
$\alpha_{20} = 0xf0f0ff00$	$\alpha_{21} = 0xccccaaaa$	$\alpha_{22} = 0xf0f0ff00$	$\alpha_{23} = 0xaaaacccc$
$\alpha_{24} = 0xaaaaff00$	$\alpha_{25} = 0xf0f0cccc$	$\alpha_{26} = 0xaaaaf0f0$	$\alpha_{27} = 0xccccff00$
$\alpha_{28} = 0xff00cccc$	$\alpha_{29} = 0xaaaaf0f0$	$\alpha_{30} = 0xff00aaaa$	$\alpha_{31} = 0xccccf0f0$

Bron: [13]

Tabel 6: Constanten bij Pf

$\alpha_0 = 0xcaf9639c$	$\alpha_1 = 0x0ff0f9c0$	$\alpha_2 = 0x639c0ff0$	$\alpha_3 = 0xcaf9f9c0$
$\alpha_4 = 0x0ff0f9c0$	$\alpha_5 = 0x639ccaf9$	$\alpha_6 = 0xf9c00ff0$	$\alpha_7 = 0x639ccaf9$
$\alpha_8 = 0x639c0ff0$	$\alpha_9 = 0xf9c0caf9$	$\alpha_{10} = 0x0ff0caf9$	$\alpha_{11} = 0xf9c0639c$
$\alpha_{12} = 0xf9c0639c$	$\alpha_{13} = 0xcaf90ff0$	$\alpha_{14} = 0x0ff0639c$	$\alpha_{15} = 0xcaf9f9c0$
$\alpha_{16} = 0x0ff0f9c0$	$\alpha_{17} = 0xcaf9639c$	$\alpha_{18} = 0xcaf9f9c0$	$\alpha_{19} = 0x639c0ff0$
$\alpha_{20} = 0x639ccaf9$	$\alpha_{21} = 0x0ff0f9c0$	$\alpha_{22} = 0x639ccaf9$	$\alpha_{23} = 0xf9c00ff0$
$\alpha_{24} = 0xf9c0caf9$	$\alpha_{25} = 0x639c0ff0$	$\alpha_{26} = 0xf9c0639c$	$\alpha_{27} = 0x0ff0caf9$
$\alpha_{28} = 0xcaf90ff0$	$\alpha_{29} = 0xf9c0639c$	$\alpha_{30} = 0xcaf9f9c0$	$\alpha_{31} = 0x0ff0639c$

Bron: [13]

b) De substitutielaag

Zoals eerder vermeld bestaat elk vakje in de matrix uit 32 bits. De volledige matrix bestaat dus eigenlijk uit 128 kolommen van 4 bits. In de substitutielaag gaan we op elk van deze kolommen een substitutie toepassen. De bit in de bovenste rij is hier de minst beduidende bit. Elke combinatie van 4 bits wordt hierbij vervangen door een andere combinatie van 4 bits. Tabel 7 geeft dit weer. Hierbij is X is de oorspronkelijke combinatie en S(x) de nieuwe.

Tabel 7: Substitutie

X	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
S(x)	**8**	**6**	**7**	**9**	**3**	**C**	**A**	**F**	**D**	**1**	**E**	**4**	**0**	**B**	**5**	**2**

c) De diffusielaag

We bekijken nu de matrix weer als een 4x4 matrix. De elementen zijn dus weer 32 bits groot. De matrix wordt ook bij deze bewerking op een bepaalde manier onderverdeeld, namelijk via de diagonalen. Dit is weergegeven in tabel 8.

Tabel 8: Matrixindeling bij de diffusie

A0	B0	C0	D0
D1	A1	B1	C1
C2	D2	A2	B2
B3	C3	D3	A3

We bekomen hierbij dus 4 “diagonalen” namelijk A, B, C en D. Op elke diagonaal gaan we vervolgens de diffusie toepassen. We leggen dit uit aan de hand van een voorbeeld en gebruiken diagonaal A.

Vb.
A0 = A0 <<< 13
A2 = A2 <<< 3
A1 = A1 xor A0 xor A2
A3 = A3 xor A2 xor (A0 << 3)
A1 = A1 <<< 1
A3 = A3 <<< 7
A0 = A0 xor A1 xor A3
A2 = A2 xor A3 xor (A1 << 7)
A0 = A0 <<< 5
A2 = A2 <<< 22
Waarbij: <<< X = left rotation, << X = left shift (nullen invoegen), X= aantal posities.

Hetzelfde wordt uiteraard gedaan bij B, C en D.

Na deze 3 stappen is de niet-lineaire permutatie 1 keer doorlopen. In Hamsi 256 worden deze 3 stappen onder normale omstandigheden 3 keer doorlopen. Wanneer echter het laatste stukje boodschap (nl. de 32 minstbeduidende bits van de totale boodschaplengte in aantal bits) wordt verwerkt gaat de niet-lineaire permutatie 6 keer doorlopen worden en zullen er andere constanten gebruikt worden dan onder normale omstandigheden.

d) De truncatie

Het resultaat van de bovenstaande stappen is 512 bits lang. Van deze 512 bits hebben we er maar 256 nodig om de volgende chaining value te kunnen genereren. Daarom moeten we een deel van de code negeren. Bij Hamsi-256 zullen we de 2^e en laatste rij van de matrix weggooien om zo de 256 bit code te verkrijgen. Deze 256 bits gaan door een xor met de chaining value van deze blok. Dit resultaat wordt meegegeven naar de volgende blok.

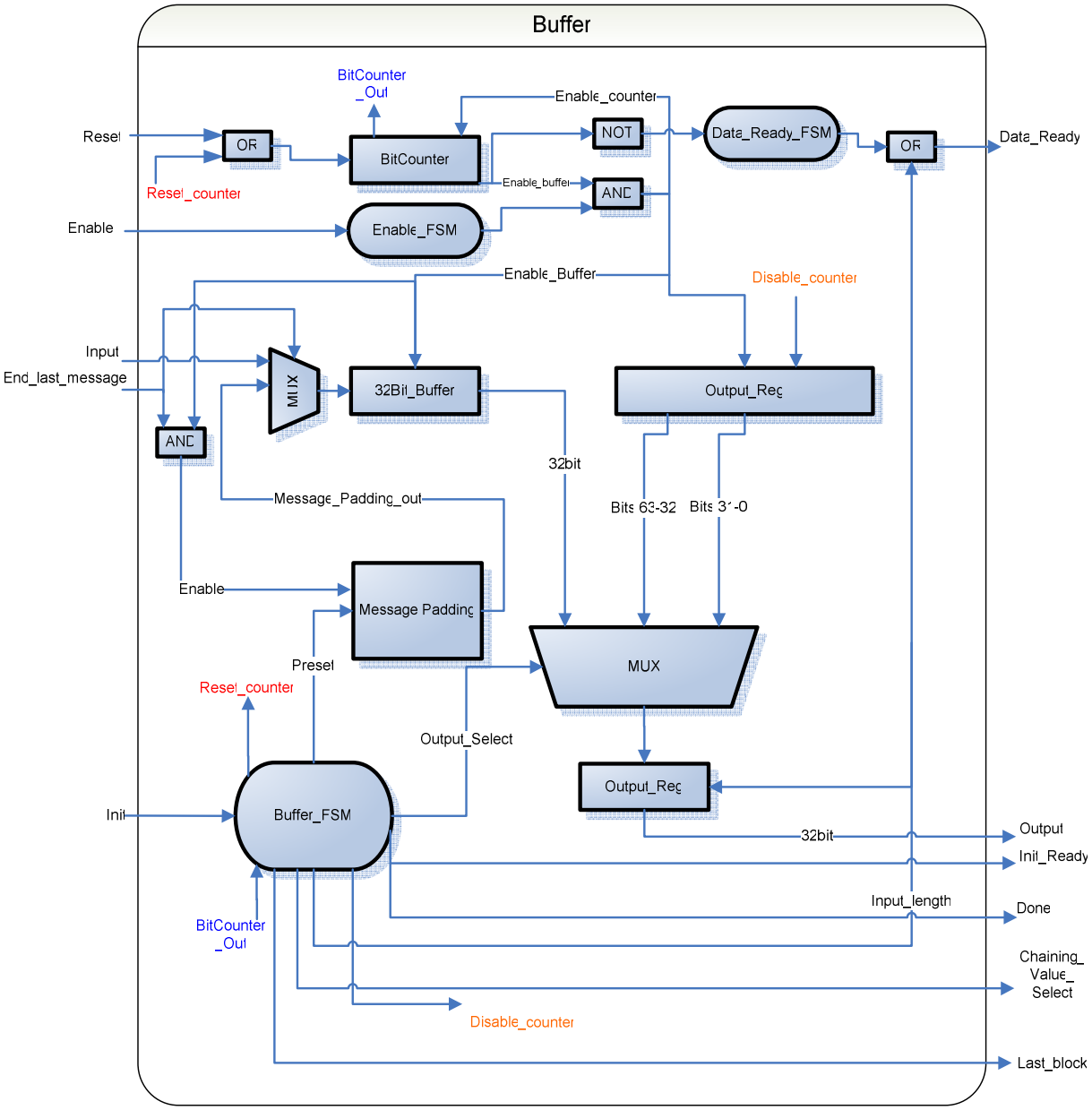
2.2.3 Uitvoering in hardware

Hier beschrijven we de hardware-implementatie van de volgende blokken:

- Buffer;
- Hamsi;
- Niet-lineaire permutatie.

2.2.3.1 Buffer

De buffer is gemaakt in 2 delen namelijk het datapad en het controlepad. In Figuur 19 is de hardware-implementatie van de buffer te zien. Hierin zijn data- en controlepad opgenomen.



Figuur 19: Hardware-implementatie van de buffer

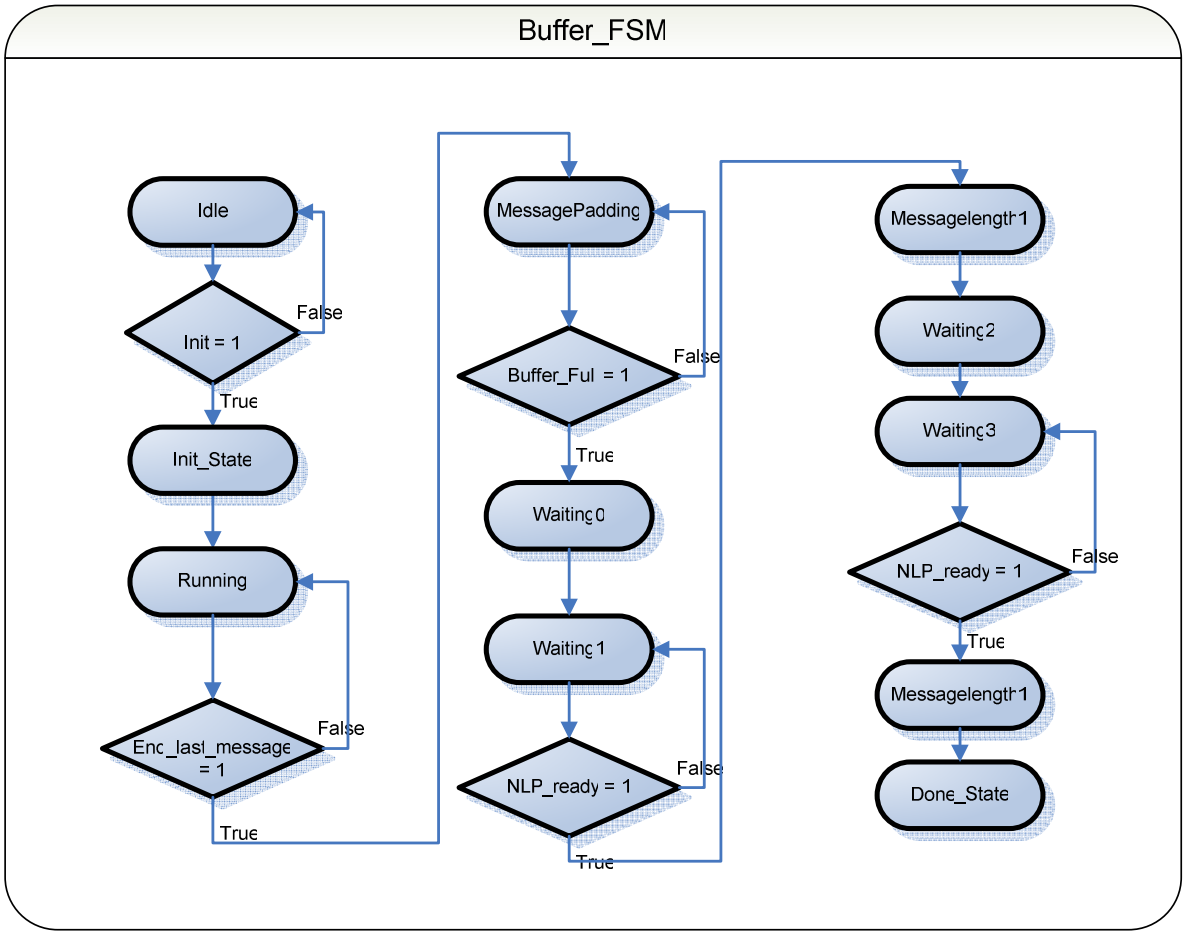
De functies van de buffer zijn enerzijds de boodschap van de input serieel,bit per bit, inladen en anderzijds de nodige messagepadding uitvoeren.
Hiervoor zijn natuurlijk een aantal controlesignalen nodig waardoor de buffer ook met andere blokken kan communiceren. Zo moet het volgende blok weten wanneer de buffer vol zit en wanneer de data gelezen mag worden.

Controlepad

In wat volgt beschrijven we de finite state machines die de werking van de buffer controleren.

Buffer_FSM

In Figuur 20 geven we de werking weer van de 'Buffer_FSM'. Vervolgens gaan we dieper in op de verschillende toestanden waarin deze FSM zich kan bevinden.



Figuur 20: Buffer_FSM

Idle: In deze toestand wacht de hardware tot er een startsignaal gegeven wordt. Wanneer dit signaal gegeven wordt zal de FSM overgaan tot de initialisatie.

Initialisatie: Het presetsignaal zal hoog worden, wat ervoor zorgt dat de flipflop van de messagepadding op 1 gezet wordt. Bij de volgende klokpuls zal de state van de FSM 'running' worden.

Running: In deze fase wordt er data aangelegd aan de buffer, met bij iedere bit een enable. Door deze enable zal ook de 6-bitcounter met 1 verhogen. Wanneer de buffer vol zit (en de

6-bitcounter dus 32 heeft bereikt) zal het signaal ‘data_Ready’ hoog worden. Het volgende blok zal dus weten dat er data klaar staat. Wanneer echter het einde van de laatste boodschap bereikt wordt zal de ingang end_last_message hoog gemaakt worden. De FSM gaat nu naar de state ‘messagepadding’.

Messagepadding: Omwille van het hoog worden van end_last_message zal de ingang van de buffer niet langer verbonden zijn met de input. Door middel van een mux zal de flipflop van de messagepadding aan de ingang gehangen worden. Deze hebben we tijdens de initialisatie op ‘1’ gezet zodat de er bij de volgende klokpuls eerst een ‘1’ aan de ingang komt te hangen. Bij deze klokpuls wordt de flipflop ook op ‘0’ gezet aangezien er een ‘0’ hangt aan de ingang. De bitcounter zal na een tijd weer melden dat de flipflop vol zit waardoor de FSM overgaat naar de state ‘waiting0’.

Waiting0 – waiting1: Om het nut van ‘waiting0’ te begrijpen moeten we kijken naar de volgende state van de FSM. In ‘waiting1’ gaat de FSM controleren of de niet-lineaire permutatie klaar is om een nieuw blok data te verwerken. Het volgend blok heeft echter een klokpuls nodig om het signaal ‘NLP_ready’ laag te maken. Hierdoor is dit signaal in de toestand ‘waiting0’ nog hoog en kunnen we hier nog niet gaan controleren op ‘NLP_ready = 1’ aangezien de FSM dan onmiddellijk zou overgaan naar de toestand ‘messagelength1’. Hierdoor zouden we dus een verkeerde werking verkrijgen.

Messagelength1: In deze fase worden de 32 meest beduidende bits van de messagecounter aan de uitgang van de buffer gehangen.

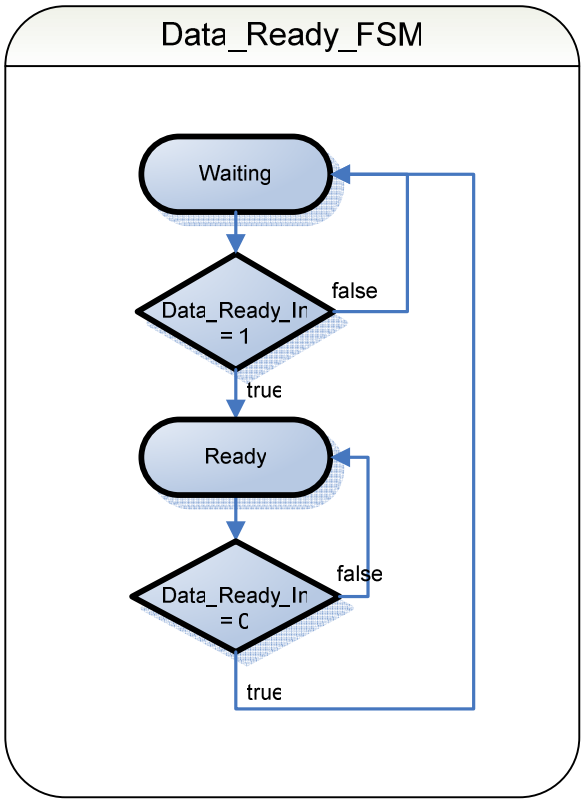
Waiting2 - waiting3: Hier geldt hetzelfde verhaal als bij waiting0 - waiting1.

Messagelength2: In deze fase worden de 32 minst beduidende bits van de messagecounter aan de uitgang van de buffer gehangen. Deze blok data is de laatste die doorgestuurd moet worden naar de niet-lineaire permutatie. We gaan daarom dus ook het signaal ‘Last_block’ hoogmaken.

Done: In deze fase zal de uitgang done op 1 gezet worden en wordt er verder niks meer uitgevoerd. De uitgang van de buffer blijft in dit geval de 32 minst beduidende bits van de messagecounter.

Data_ready_FSM en Enable_FSM

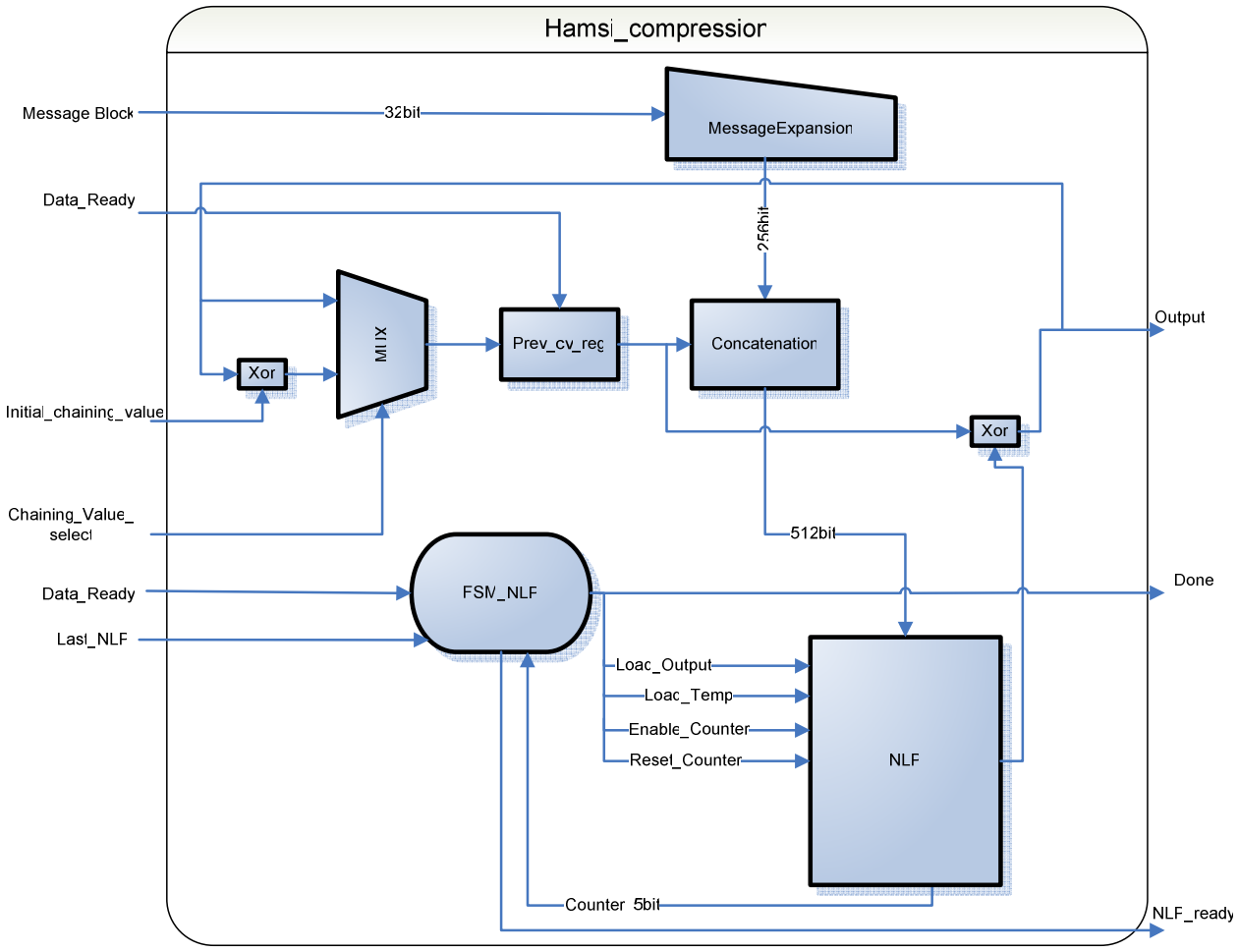
Tenslotte zijn er nog 2 FSM's die we nog niet behandeld hebben, namelijk de enable_FSM en de Data_Ready_FSM. Deze FSM's zijn ontworpen om het ingangssignaal, dat verschillende klokpulsen hoog is, te verkorten tot een uitgangssignaal dat maar 1 klokcyclus hoog is. Beide FSM's moeten exact hetzelfde werk doen dus zien ze er ook hetzelfde uit (zie Figuur 21).



Figuur 21: Data_Ready_FSM en Enable_FSM

2.2.3.2 Hamsi-256

Dit blok is ook weer gemaakt in 2 delen namelijk het datapad en het controlepad. In Figuur 22 is de hardware-implementatie te zien.



Figuur 22: Hamsi_encryption

Bespreking van de blokken bestemd voor dataverwerking

De message-expansie

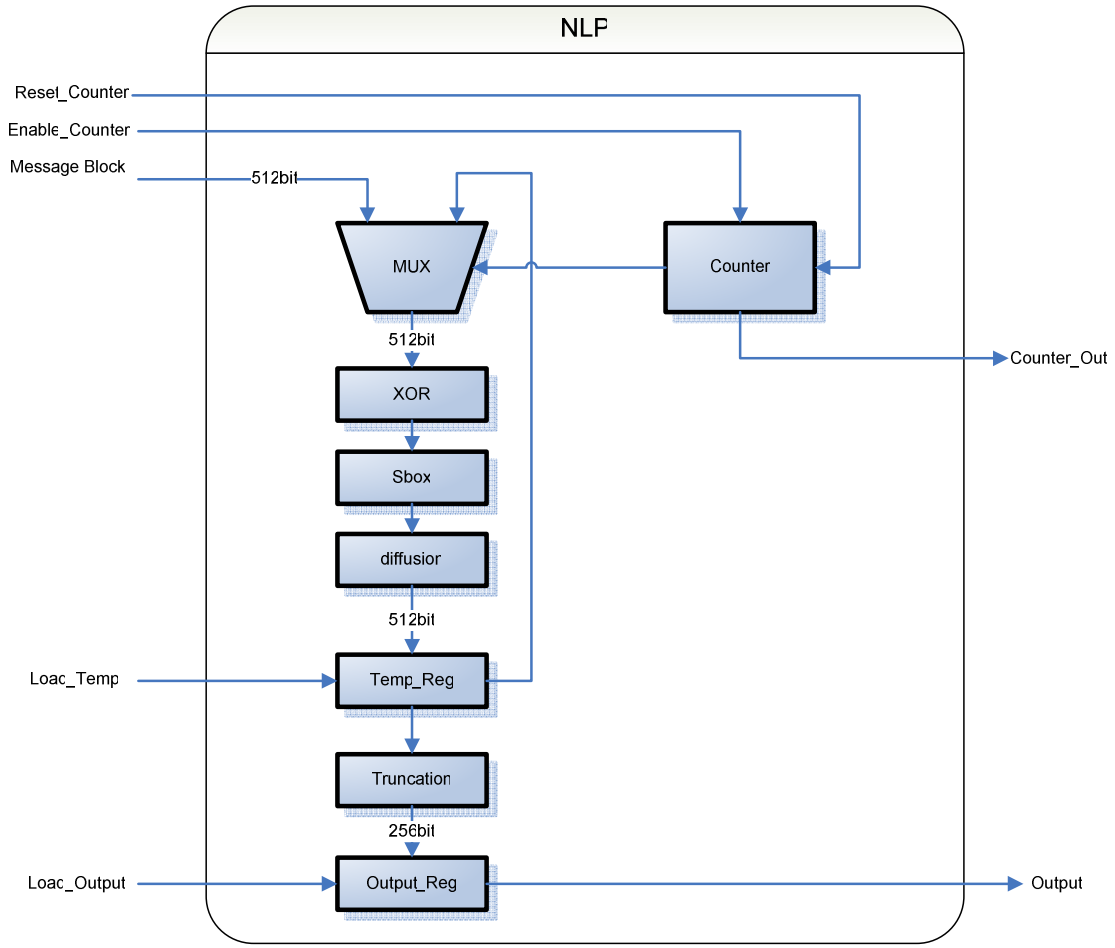
Dit blok vormt de 32 bit boodschapblok om tot een blok van 256 bits. Dit gebeurt door iedere uitgangsbit via een aantal xors te verbinden met een aantal ingangsbits.

De concatenatie

Dit blok zet de geëxpandeerde boodschap van het vorige blok om naar een nog groter blok door de boodschap aan de ingang te combineren met de chaining value. De boodschap wordt 512 bits groot.
Aan de chaining value van de concatenation hangt bij het eerste én bij het tweede boodschapblok de xor van de initial chaining value en de output van de niet lineaire permutatie . Bij de eerste boodschapblok is de output van de niet lineaire permutatie “000…000”, dus hangt er eigenlijk gewoon de initial chaining value aan de ingang, bij het tweede boodschapblok is de chaining value de xor van de output en de vorige chaining value (dus de initial chaining value) . Vanaf het derde boodschapblok is de chaining value echter de xor van de uitgang van de niet lineaire permutatie en de chaining value waarmee deze waarde werd bekomen. Deze waarde van de huidige chaining value komt altijd in ‘prev_cv_reg’ te staan.

De niet-lineaire permutatie

In Figuur 23 zien we de interne werking van de niet-lineaire permutatie.

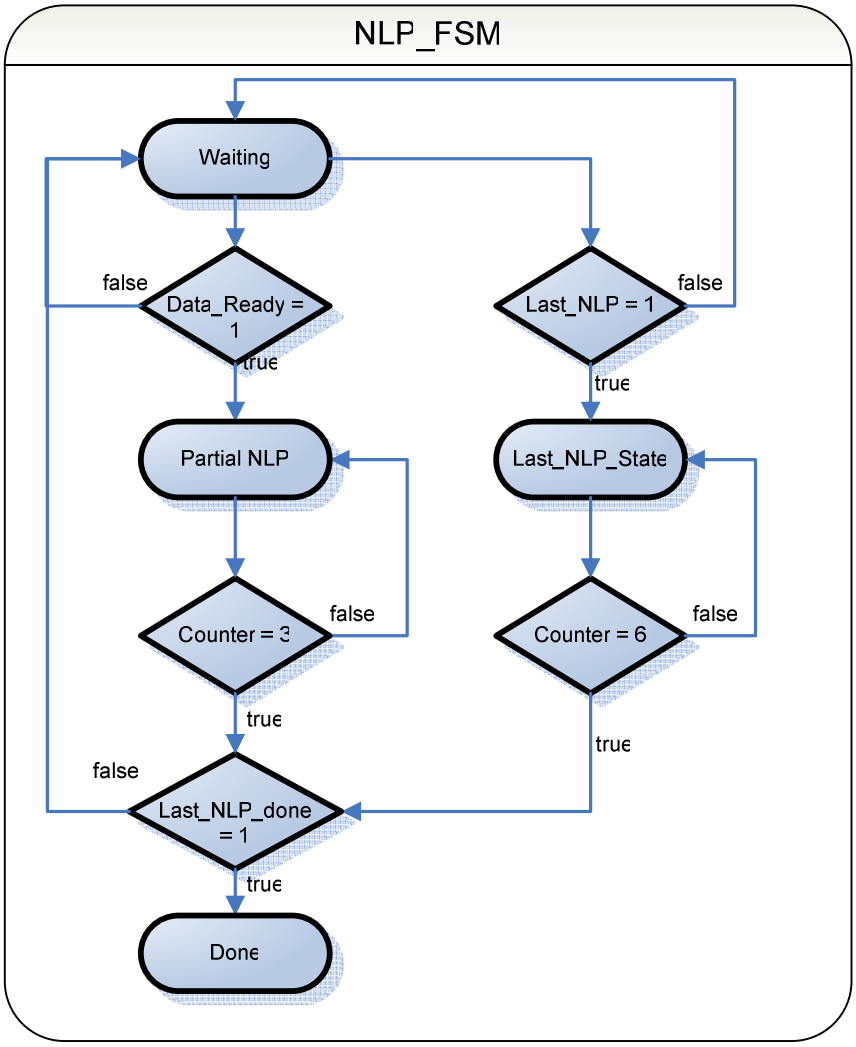


Figuur 23: Niet lineaire permutatie

De werking van de blokken ‘XOR’ , ‘Sbox’, ‘Diffusion’ en ‘Truncation’ staan uitgelegd in hoofdstuk “2.2.2.5 Niet-lineaire permutatie”. De blokken ‘Temp_Reg’ en ‘Output_Reg’ zijn registers die we nodig hebben om respectievelijk de tussenresultaten en het eindresultaat op te slaan.

De FSM van de niet-lineaire permutatie

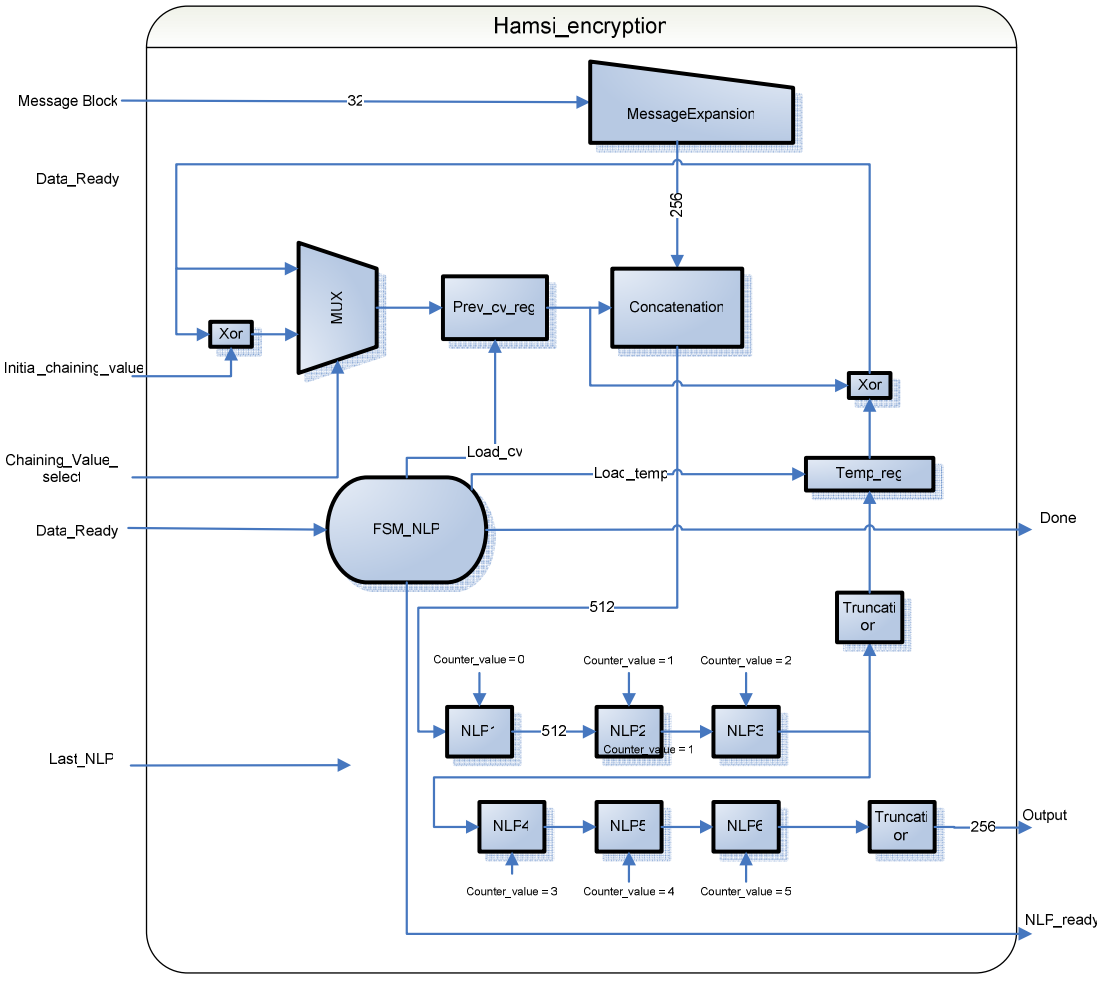
In dit geval is er maar één blok verantwoordelijk voor de goede werking van Hamsi namelijk het blok ‘FSM_NLP’. De werking van deze FSM kunnen we zien in Figuur 24.



Figuur 24: De FSM van de niet-lineaire permutatie bij de kleine implementatie

In eerste instantie zal de FSM in de toestand ‘waiting’ zijn. Wanneer er echter een signaal wordt gegeven dat er data klaar staat, zal de niet-lineaire permutatie 3 keer uitgevoerd worden, gevolgd door de truncatie. Vervolgens zal de FSM opnieuw in de toestand ‘waiting’ komen.
Er zijn echter 2 verschillende niet-lineaire permutaties. Wanneer het allerlaatste blok data wordt verwerkt, zal de niet-lineaire permutatie namelijk 6 keer uitgevoerd moeten worden in plaats van 3 keer. Om dit te verwezenlijken zal de FSM reageren op het signaal ‘Last_NLP’. Wanneer dit signaal hoog wordt terwijl de FSM in de toestand ‘waiting’ staat, zal de FSM de toestand ‘Last_NLP_State’ aannemen.
Hierdoor wordt de niet-lineaire permutatie dus 6 keer uitgevoerd en last_NLP_done op ‘1’ gezet.
Vervolgens komt de FSM nog in de toestand ‘Done’.

2.2.3.4 Snelle implementatie van Hamsi-256



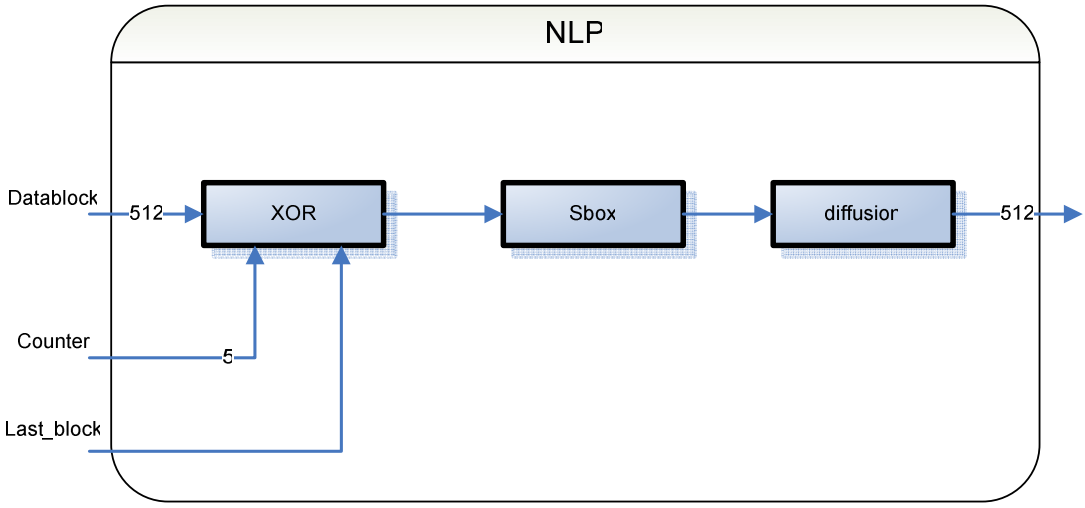
Figuur 25: Snelle implementatie van Hamsi-256

Zoals in Figuur 26 te zien is zijn de niet-lineaire permutaties hier combinatorisch uitgevoerd. De counter (die normaal aan de ingang van de XOR-blok wordt toegevoegd) uit de snelle implementatie vervangen we nu door constanten die we aan de permutaties hangen. Ook komt er in iedere permutatieblok nog eens het signaal Last_nlp binnen,

waardoor ook weer de constanten in de XOR-blok worden aangestuurd. Dit werd niet volledig getekend aangezien dit de Figuur veel te onduidelijk zou maken

In een gewone permutatie zouden er 3 rondes moeten zijn. Daarom tappen we het signaal achter de 3^{e} permutatieblok af en gaan we dit na de truncation terugsturen als volgende chaining value. Deze truncation is niet echt een blok aangezien we de helft van de signalen gewoonweg niet gebruiken aan de ingang van de cv_reg.

Wanneer we nu in de permutatie van het laatste datablok zitten staat de uiteindelijke hash-waarde automatisch aan de uitgang, zonder dat hier ook maar één klokcyclus voor nodig is.



Figuur 26: De niet-lineaire permutatie bij de snelle implementatie

2.2.3.4 Evaluatie op basis van snelheid en oppervlaktegebruik.

Na synthese met ISE, een ontwikkeltool van Xilinx, krijgen we volgende gegevens die ons toelaten de compressiefuncties te vergelijken wat betreft snelheid en oppervlakte:

Tabel 9: Vergelijking tussen implementaties van de LANE-compressiefunctie

Xilinx Virtex-5 FPGA	**Hamsi-256 fast**	**Hamsi-256 small**	**Hamsi[13]**
Aantal slices	*4,664*	*2113*	*733*
Aantal Slice registers	*514*	*781*	*/*
Aantal slice LUT's	*7216*	*3004*	*/*
Minimale periode	*4.826*	*3.242 ns*	*3.484 ns*
Geschatte maximale klokfrequentie	*207 MHz*	*308 MHz*	*287 MHz*
Aantal klokcycli per compressie	*1*	*5*	*7*
Tijd voor 1 compressie	*24 ns*	*16 ns*	*24 ns*
Doorvoercapaciteit	*6.62 Gbps*	*1.97 Gbps*	*1.48 Gbps*
Doorvoercapaciteit /oppervlakte	*1.42 Mbps/slice*	*0.93 Mbps/slice*	*2 Mbps/slice*

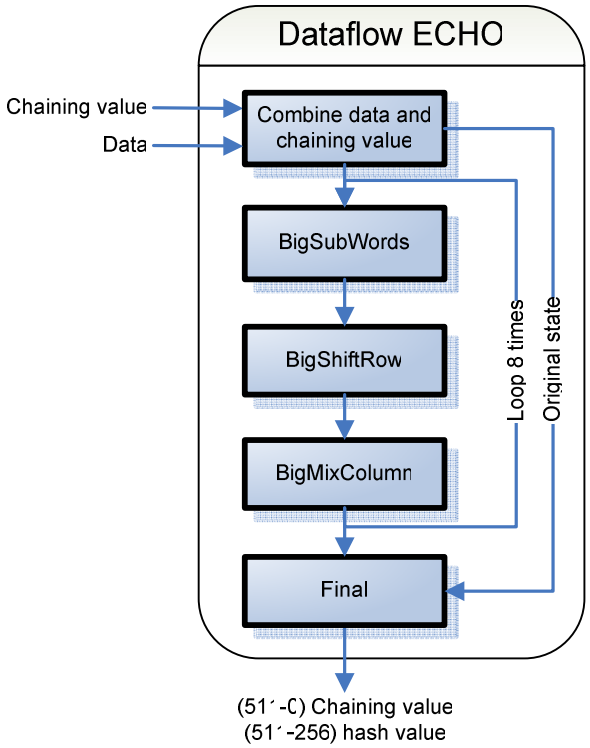
In de tabel is te zien dat de kleine versie van Hamsi-256 meer dan 2 keer zo klein is geworden en dat de kloksnelheid gestegen is. Dit is te verklaren door het feit dat we maar 1 in plaats van 6 permutatie-blokken hebben.
Uit de tabel valt ook af te leiden dat de doorvoercapaciteit meer dan 3 keer zo groot geworden is. Dit is op analoge wijze te verklaren.

2.3 ECHO

De hashfunctie ECHO [14] is een iteratieve hashfunctie waarvan de compressiefunctie schematisch getoond wordt in Figuur 27. We zien dat de binnenkomende boodschap samen met de chainingwaarde wordt gecombineerd tot een ECHO-state (zie Tabel 10). De state is een 4x4 matrix met 128 bit woorden, wat een totaal van 2048 bits is. We passen de transformaties BigSubWords, BigShiftRow en BigMixColumn een aantal keer toe op de beginstate en voeren daarna ook een Final-transformatie uit.

Er bestaan verschillende varianten van ECHO die een hashwaarde van 128 tot en met 256 of 257 tot en met 512 bits genereren. In deze masterproef werken we met een versie die een hashwaarde van 256 bits heeft.

In de volgende paragrafen worden de verschillende schillen van de implementatie uitgelegd. We beginnen met het blok dat een ECHO-state maakt van de data. Hierna volgen de beschrijvingen van de transformatieblokken en het Final-blok. Als laatste lichten we de implementatie toe van de volledige compressiefunctie en de bovenliggende schillen, die toelaten de hashfunctie praktisch te gebruiken.



Figuur 27: Dataflow ECHO-compressiefunctie

Tabel 10: De ECHO-state

w_0	w_4	w_8	w_{12}
w_1	w_5	w_9	w_{13}
w_2	w_6	w_{10}	w_{14}
w_3	w_7	w_{11}	w_{15}

Bron: [14]

2.3.1 ECHO-state aanmaken

Tabel 11 geeft een overzicht van de combinatie van chainingwaarden en data tot een ECHO-state. De letter v staat voor de chainingwaarde en de letter m voor de nieuwe data. Als we nieuwe data willen hashen moeten we uiteraard de chainingwaarde van de vorige compressie als input gebruiken. De exponent geeft aan welk 128 bit woord uit de 1536 bit lange datastring genomen moet worden

Tabel 11: Samenstelling van de ECHO-state

v_{i-1}^0	m_i^0	m_i^4	m_i^8
v_{i-1}^1	m_i^1	m_i^5	m_i^9
v_{i-1}^2	m_i^2	m_i^6	m_i^{10}
v_{i-1}^3	m_i^3	m_i^7	m_i^{11}

Bron: [14]

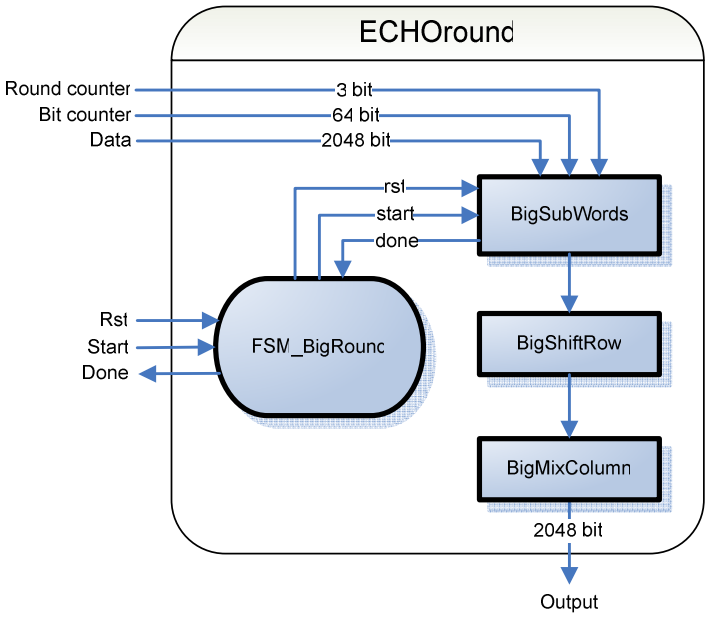
2.3.2 ECHO-ronde

Een ECHO-ronde bestaat uit drie transformaties en zal in de compressiefunctie acht keer doorlopen worden. De transformaties BigSubWords, BigShiftRow en BigMixColumn worden in volgende paragrafen besproken. Figuur 28 geeft een overzicht van de hardware-implementatie van een ECHO-ronde.

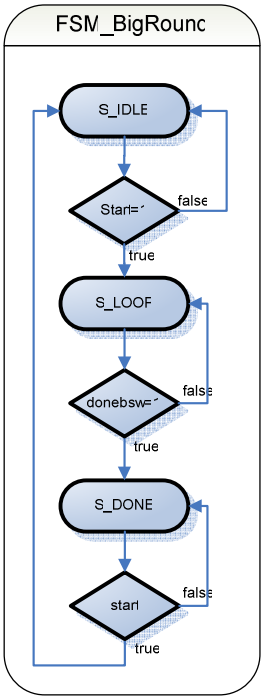
Wanneer de ronde gestart wordt, moet BigSubWords ook gestart worden. Hierna moeten we wachten tot done hoog wordt, waarna we dan de twee resterende transformaties uitvoeren. Waarom de implementatie zo is opgebouwd zal blijken uit de volgende paragrafen. Figuur 29 toont een stroomdiagram van FSM_BigRound.

FSM_BigRound begint in de state S_IDLE waar hij wacht op het hoogmaken van het startsignaal. Indien het startsignaal hoog is, wordt de BigSubWordstransformatie gestart en wachten we in de state S_LOOP tot ze klaar is. Nu dienen enkel nog de BigShiftRow- en

BigMixColumntransformatie uitgevoerd te worden in de volgende klokpuls. We eindigen in de S_DONE state waar het donesignaal van ECHOround hoog wordt gemaakt.



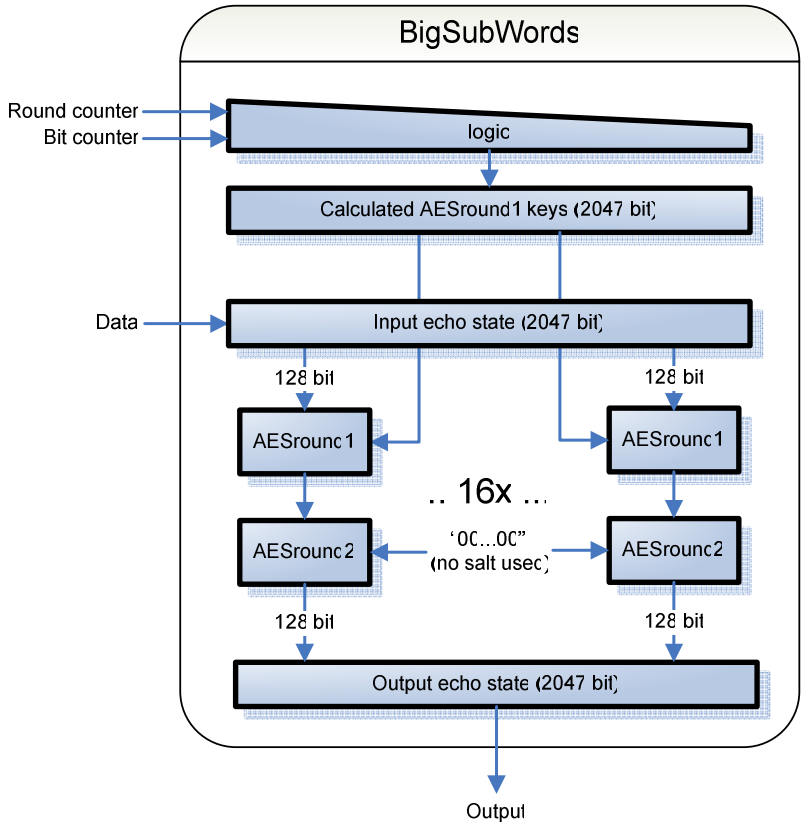
Figuur 28: De ECHO-ronde



Figuur 29: FSM_BigRound

2.3.2.1 BigSubWords

Deze operatie is eigenlijk een S-box look-up. Op ieder 128 bit woord wordt twee keer een AES-ronde toegepast (zie Figuur 30). Bij de eerste ronde hangt de sleutel af van een counter en bij de tweede ronde kunnen we een saltwaarde opgeven of gewoon nul kiezen als we geen salt wensen te gebruiken (zoals in deze masterproef). De opbouw van een AES-ronde wordt uitgelegd in de volgende paragraaf.

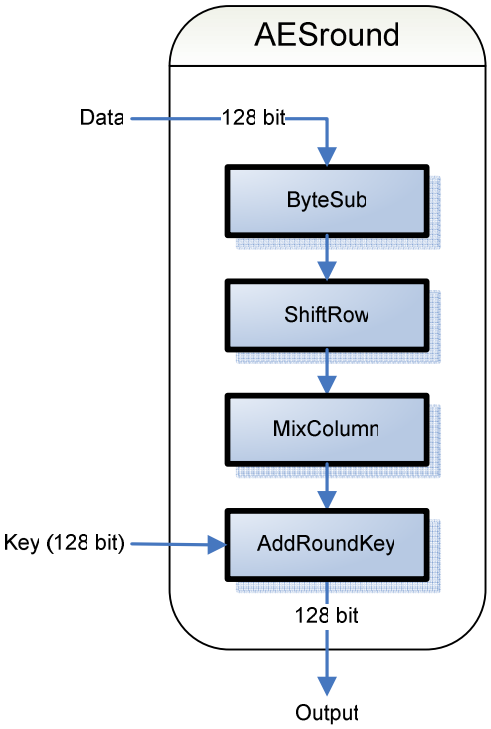


Figuur 30: De BigSubWordstransformatie van ECHO

AES-ronde

Een AES-ronde [5] bestaat uit 4 transformaties die achtereenvolgens worden uitgevoerd op een 128 bit woord (Zie Figuur 31):

- *SubBytes:*
  Look-up-table: iedere byte wordt vervangen door een andere byte, zoals beschreven onder paragraaf 2.1.1.1.
- *Shiftrows:*
  De rijen worden net als in BigShiftRows een aantal plaatsen verschoven, zoals beschreven onder paragraaf 2.1.1.2..
- *Mixcolumns:*
  Er wordt een transformatie toegepast op iedere kolom, zoals beschreven onder paragraaf 2.1.1.3.
- *AddRoundKey:*
  Er wordt door middel van EXOR's een sleutel toegevoegd.
  Zoals eerder vermeld is de sleutel de eerste keer afhankelijk van een counter en de tweede keer kan ofwel een saltwaarde opgegeven worden ofwel nul gebruikt worden als we geen salt wensen te gebruiken.



Figuur 31: De AES-ronde

Implementatiemogelijkheden

De BigSubWordstransformatie kan op verschillende manieren worden geïmplementeerd. Een eerste manier is door middel van pipelining. De verschillende 128 bit woorden worden achtereenvolgens in een pipeline van twee AES-ronden geschoven. Het aantal pipelines kan gekozen worden in functie van het gewenste oppervlaktegebruik. De snelheid zal wel dalen wanneer we minder pipelines in parallel schakelen. In Tabel 11 staan de resultaten van enkele implementaties (genoteerd als 2 x 'aantal'). Als referentiepunt starten we met de implementatie waarbij twee keer 16 AES-ronden aangemaakt worden, zonder tussenregister.

Een tweede manier is door slechts één AES-ronde te hergebruiken (wordt verder 'single mode' genoemd en genoteerd als 1 x 'aantal'). Een overzicht van de resultaten van de verschillende implementaties staat in Tabel 12. Ons referentiepunt is ook hier een implementatie met twee maal 16 blokken.

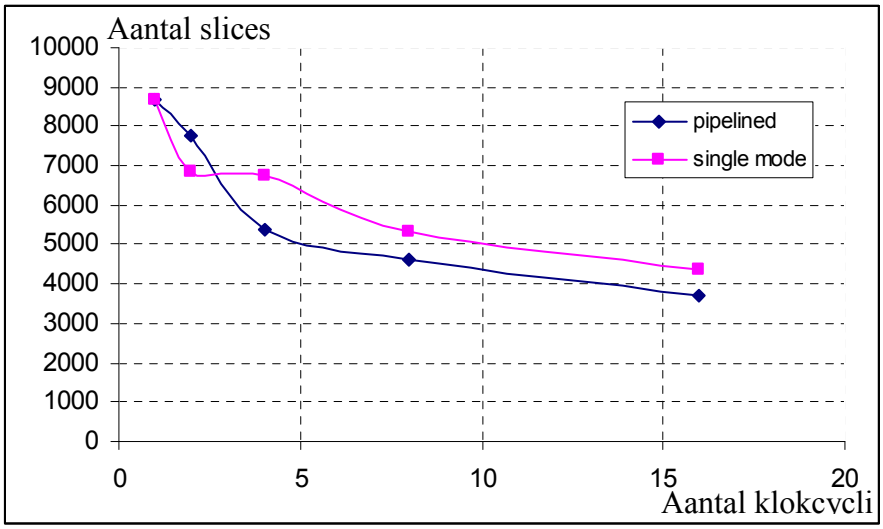
De gegevens uit Tabel 12 en 13 worden grafisch weergegeven in Figuur 32. Hieruit kunnen we een optimale implementatie kiezen.

Tabel 12: Overzicht pipelined implementaties

Virtex 5 XC5VLX155T	**ref**	**2 x 8**	**2 x 4**	**2 x 2**	**2 x 1**
Maximale frequentie	*95MHz*	*266MHz*	*247MHz*	*203MHz*	*224MHz*
Aantal klokcycli	*1*	*3*	*5*	*9*	*17*
Aantal Slices	*8659*	*7763*	*5372*	*4635*	*3702*

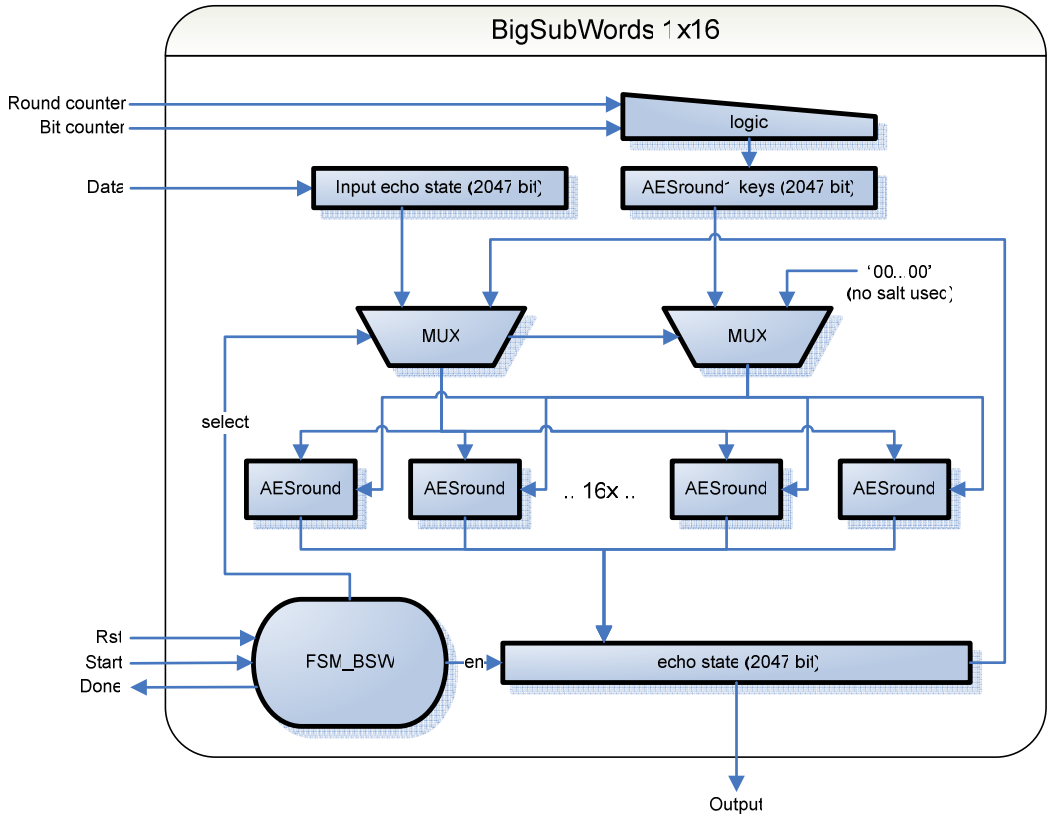
Tabel 13: Overzicht single mode implementaties

Virtex 5 XC5VLX155T	**ref**	**1 x 16**	**1 x 8**	**1 x 4**	**1 x 2**
Maximale frequentie	*95MHz*	*266MHz*	*261MHz*	*227MHz*	*200MHz*
Aantal klokcycli	*1*	*2*	*4*	*8*	*16*
Aantal Slices	*8659*	*6864*	*6743*	*5353*	*4357*

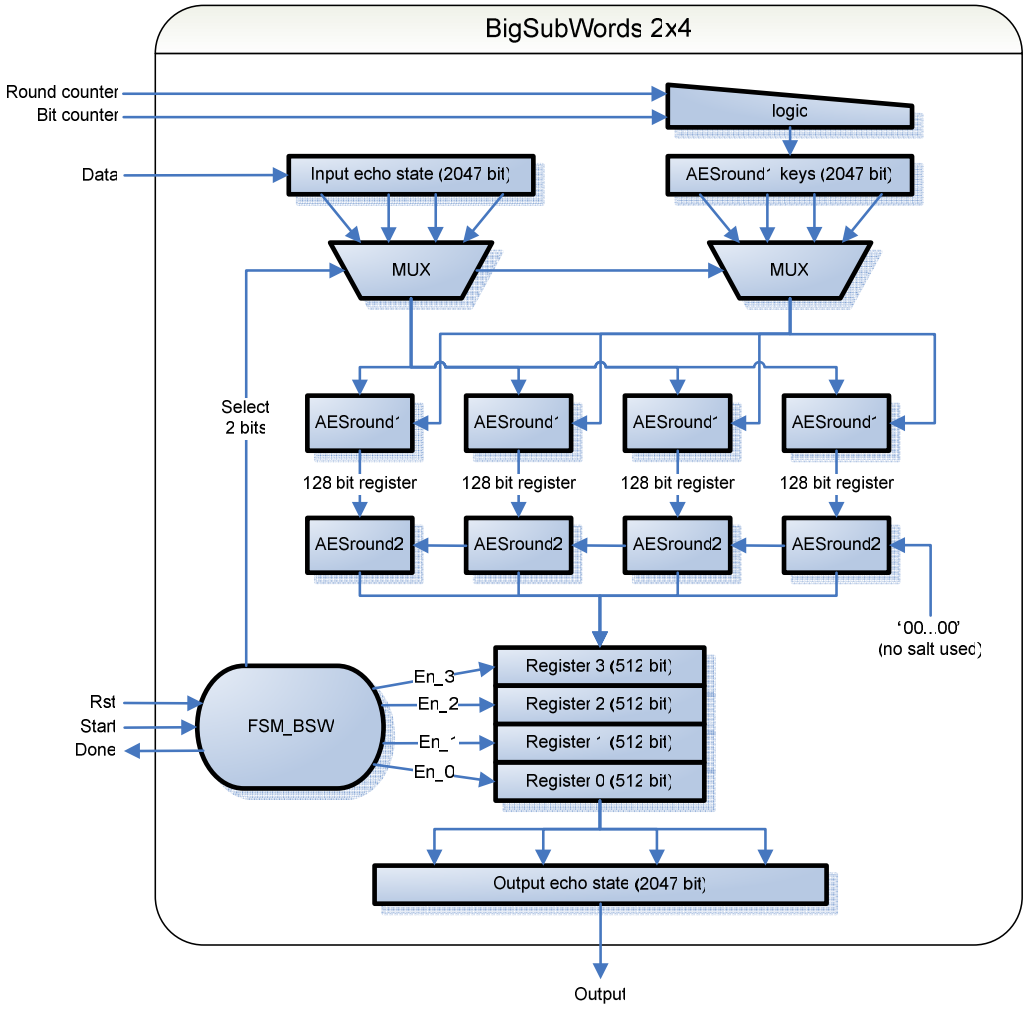


Figuur 32: Resultaten BigSubWordsimplementatie

Het is zeer belangrijk om niet enkel een zo klein mogelijk ontwerp na te streven, maar ook rekening te houden met de vertraging die hierdoor opgewekt wordt. Voor één compressie wordt immers acht keer de BigSubWordscyclus doorlopen en de totale vertraging zal dus ongeveer een factor acht groter zijn dan de vertraging van dit blok wanneer we het verschil in maximale frequentie verwaarlozen. Op Figuur 32 zien we dat er twee implementaties zijn die aan onze eisen voldoen. Willen we efficiëntie met de nadruk op snelheid, dan kiezen we best de '1x16'-implementatie. Wanneer we, zoals in deze masterproef, de nadruk op verkleining leggen kiezen we best voor de '2x4'-implementatie.
Hoe deze implementaties opgebouwd zijn zien we in respectievelijk Figuur 33 en Figuur 34.



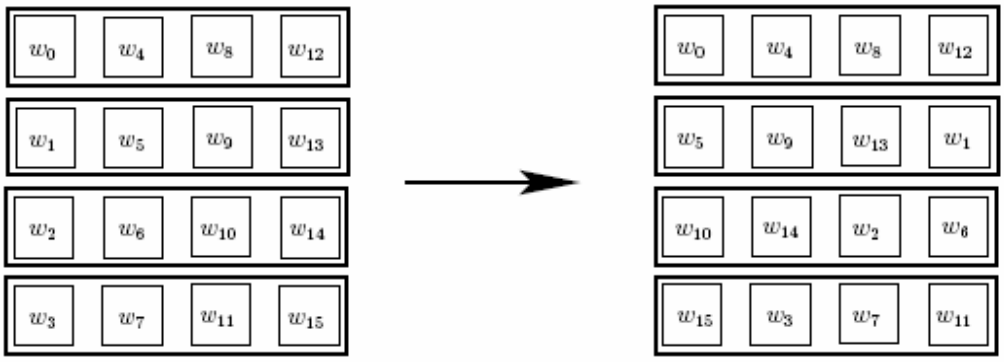
Figuur 33: BigSubWordsimplementatie 1x16 single mode



Figuur 34: BigSubWordsimplementatie 2x4 pipelined

2.3.2.2 BigShiftRows

De BigShiftRowstransformatie is gelijkaardig aan de ShiftRowstransformatie in AES. Het verschil is de lengte van de woorden in de state. De rijen worden een aantal keer naar rechts verschoven naargelang hun rijnummer. Zo zal de eerste rij, rij nul, niet verschoven worden en rij drie zal drie plaatsen naar rechts verschoven worden. De BigShiftRowtransformatie wordt grafisch weergegeven in Figuur 35.

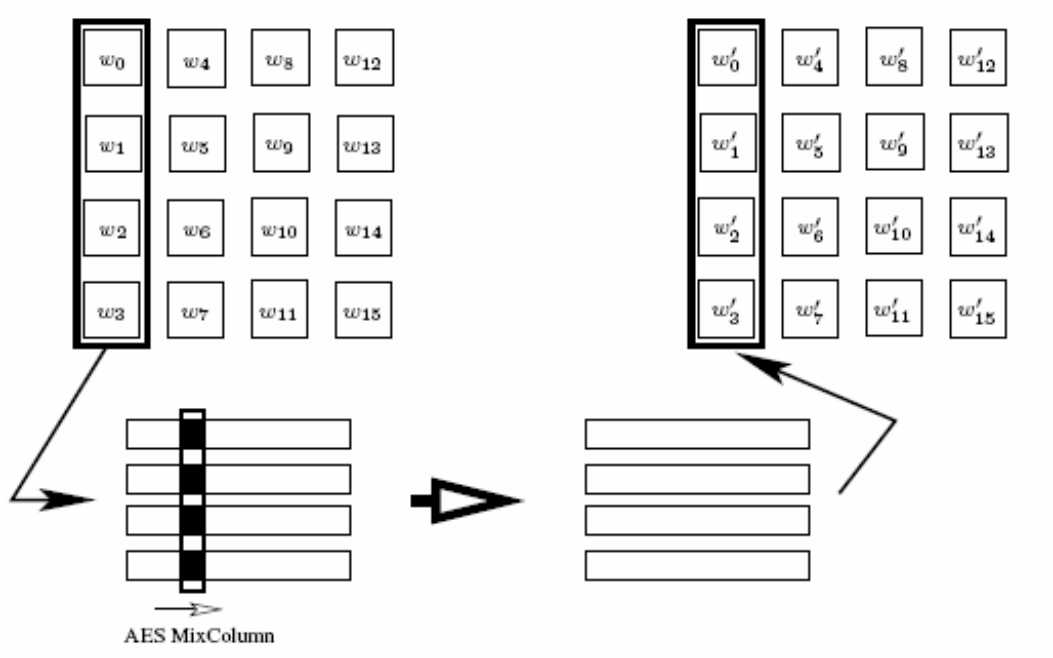


Figuur 35: De BigShiftRowstransformatie van ECHO

Bron: [14]

2.3.2.3 BigMixColumns

De BigMixColumnstransformatie is gelijkaardig aan de MixColumnstransformatie in AES. Omdat de woorden in een ECHO-state 128 bits lang zijn en de AES-transformatie slechts kolommen van acht bit woorden als input neemt moeten we 64 keren een AES-transformatie uitvoeren. Aangezien de MixColumnstransformatie enkel bestaat uit EXOR's valt er geen winst te behalen door parallellisering wat betreft oppervlaktegebruik. We zouden immers een FSM, MUX en registers nodig hebben om ieder stuk apart bij te houden. Uit simulatie blijkt dat die benadering meer plaats in neemt dan 64 keer de MixColumnstransformatie te instantiëren. De BigMixColumnstransformatie wordt grafisch weergegeven in Figuur 36.



Figuur 36: De BigMixColumnstransformatie van ECHO
Bron: [14]

2.3.3 Finalisatie ECHO

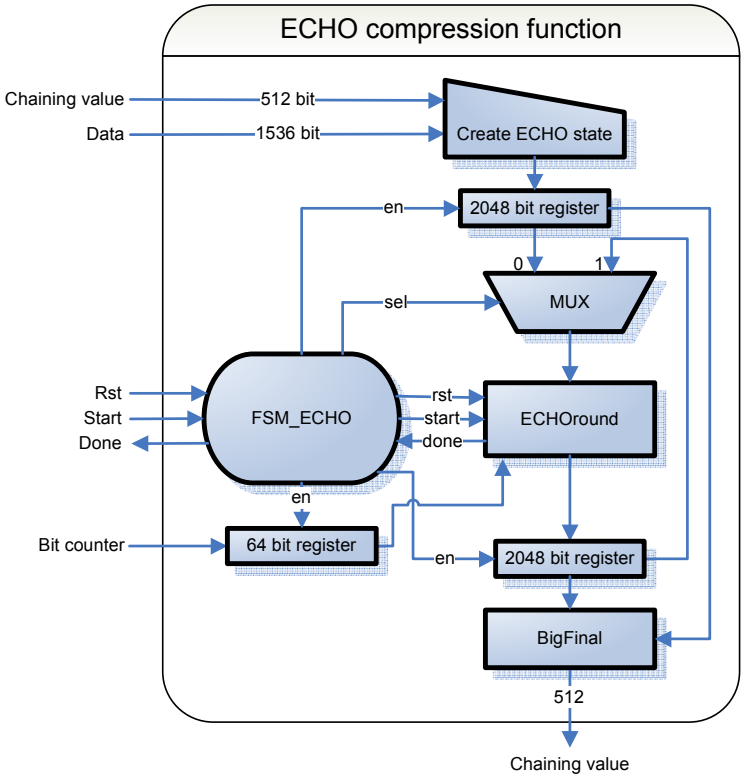
Om de compressie te finaliseren wordt de bekomen eindstate gecombineerd met de startstate op onderstaande manier:

$$
\begin{aligned}
v_i^0 &= v_{i-1}^0 \oplus m_i^0 \oplus m_i^4 \oplus m_i^8 \oplus w_0 \oplus w_4 \oplus w_8 \oplus w_{12} \\
v_i^1 &= v_{i-1}^1 \oplus m_i^1 \oplus m_i^5 \oplus m_i^9 \oplus w_1 \oplus w_5 \oplus w_9 \oplus w_{13} \\
v_i^2 &= v_{i-1}^2 \oplus m_i^2 \oplus m_i^6 \oplus m_i^{10} \oplus w_2 \oplus w_6 \oplus w_{10} \oplus w_{14} \\
v_i^3 &= v_{i-1}^3 \oplus m_i^3 \oplus m_i^7 \oplus m_i^{11} \oplus w_3 \oplus w_7 \oplus w_{11} \oplus w_{15}
\end{aligned}
$$

Hierbij is v de chainingwaarde, m de data en w de state na alle transformaties. $v_i^0 \| v_i^1 \| v_i^2 \| v_i^3$ is dan de chainingwaarde en $v_i^0 \| v_i^1$ de hashwaarde (voor een output van 128 tot en met 256 bits).

2.3.4 ECHO-Compressiefunctie

Voorgaande blokken kunnen we volgens Figuur 37 combineren tot de ECHO-compressiefunctie. Tabel 14 geeft een vergelijking tussen een versie die alle hardware aanmaakt zoals hij beschreven staat in de specificaties en een geoptimaliseerde versie waarbij de optimalisaties die voor ieder blok apart beschreven zijn allemaal zijn doorgevoerd.



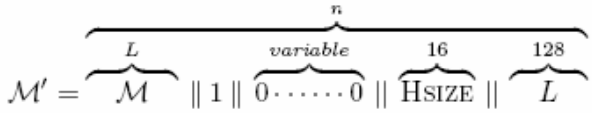
Figuur 37: De ECHO-compressiefunctie

Tabel 14: Vergelijking tussen implementaties van de ECHO-compressiefunctie

Xilinx Virtex-5 FPGA	**ECHO standaard**	**ECHO Geoptimaliseerd**
Aantal slices	*15006*	*12061*
Aantal Slice registers	*4105*	*8800*
Aantal slice LUT's	*29330*	*14407*
Minimale periode	*7.154 ns*	*5.333 ns*
Geschatte maximale klokfrequentie	*139 MHz*	*187 MHz*
Aantal klokcycli per compressie	*9*	*81*
Tijd voor 1 compressie	*64 ns*	*432 ns*
Doorvoercapaciteit	*23.86 Gbps*	*3.56 Gbps*
Doorvoercapaciteit /oppervlakte	*1.59 Mbps/slice*	*0.30 Mbps/slice*

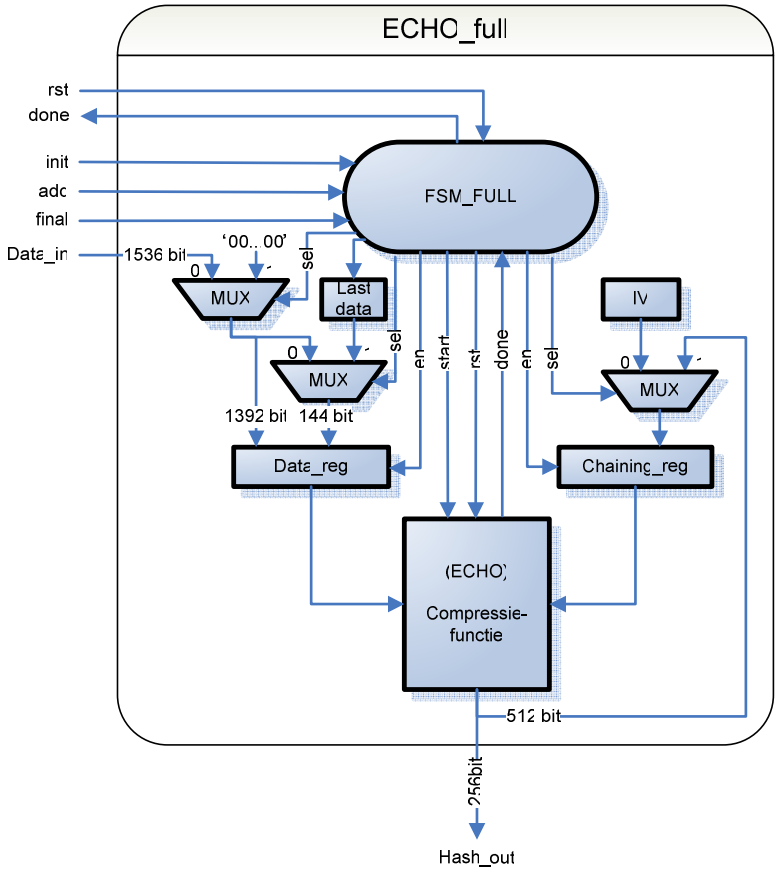
2.3.5 Testopstelling ECHO

Om de compressiefunctie gemakkelijk te kunnen gebruiken om data te hashen, is het blok ECHO_full geïmplementeerd (zie Figuur 39). Dit blok kan alle stappen in het hashproces afhandelen. Dit houdt in dat de initialisatie automatisch gebeurt, blokken data kunnen worden toegevoegd en de uiteindelijke hashwaarde kan worden berekend. De waarde van de initialisatievector is voor ECHO-256 gelijk aan de hexadecimale waarde:
0001000000000000000000000000000010000000000000000000000000000000
0001000000000000000000000000000010000000000000000000000000000000.
Indien de extra einddata niet in één keer toegevoegd kan worden, zal automatisch een tweede datablok aangemaakt en verwerkt worden. Merk hierbij op dat het laatste datablok reeds voorzien moet zijn van een toegevoegde '1' na de data. De message padding gebeurt immers zoals weergegeven in Figuur 38.



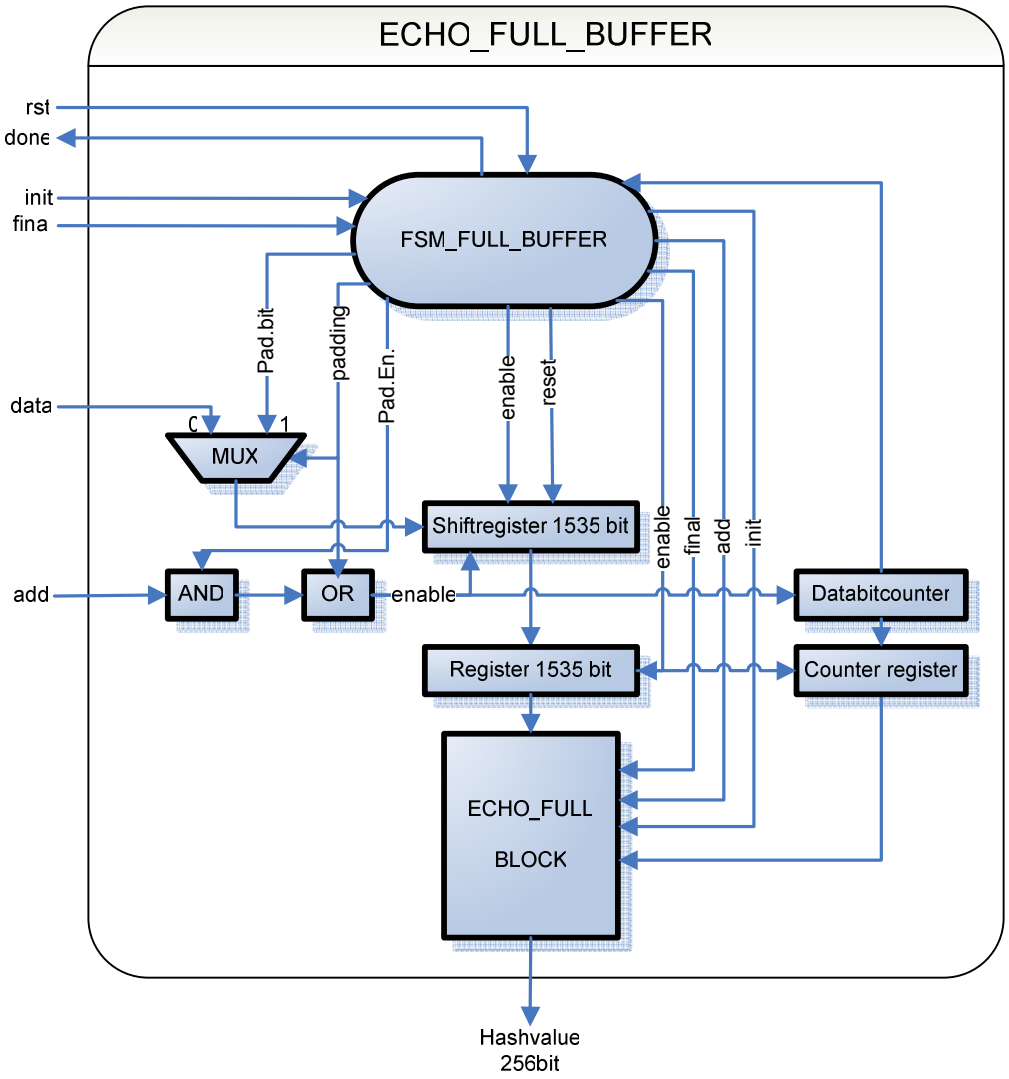
Figuur 38: Message padding in ECHO
Bron: [13]

Hierbij is Hsize hexadecimaal '0001' voor de 256-bit versie van ECHO. L stelt de interne bitcounter voor met het aantal gehashte bits. M is de originele te hashen data en M' de gepadde data.



Figuur 39: De volledige implementatie van ECHO (ECHO_full)

Nog een niveau hoger is er een buffer geïmplementeerd waardoor het mogelijk is om de data bit per bit toe te voegen en de message padding volledig automatisch te laten gebeuren. De centrale elementen zijn een schuifregister met bijhorende counter. Indien het schuifregister voldoende data bevat, wordt automatisch een blok gehashed. Een grafische voorstelling van dit blok zien we in Figuur 40.



Figuur 40: De volledige implmentatie van ECHO, inclusief databuffer (ECHO_full_buffer)

2.4 Luffa

Luffa is een familie van cryptografische hash functies [14]. Er zijn 4 varianten van Luffa namelijk Luffa-224 ,Luffa-256, Luffa-384 , Luffa-512. Tabel 15 toont de verschillende varianten.

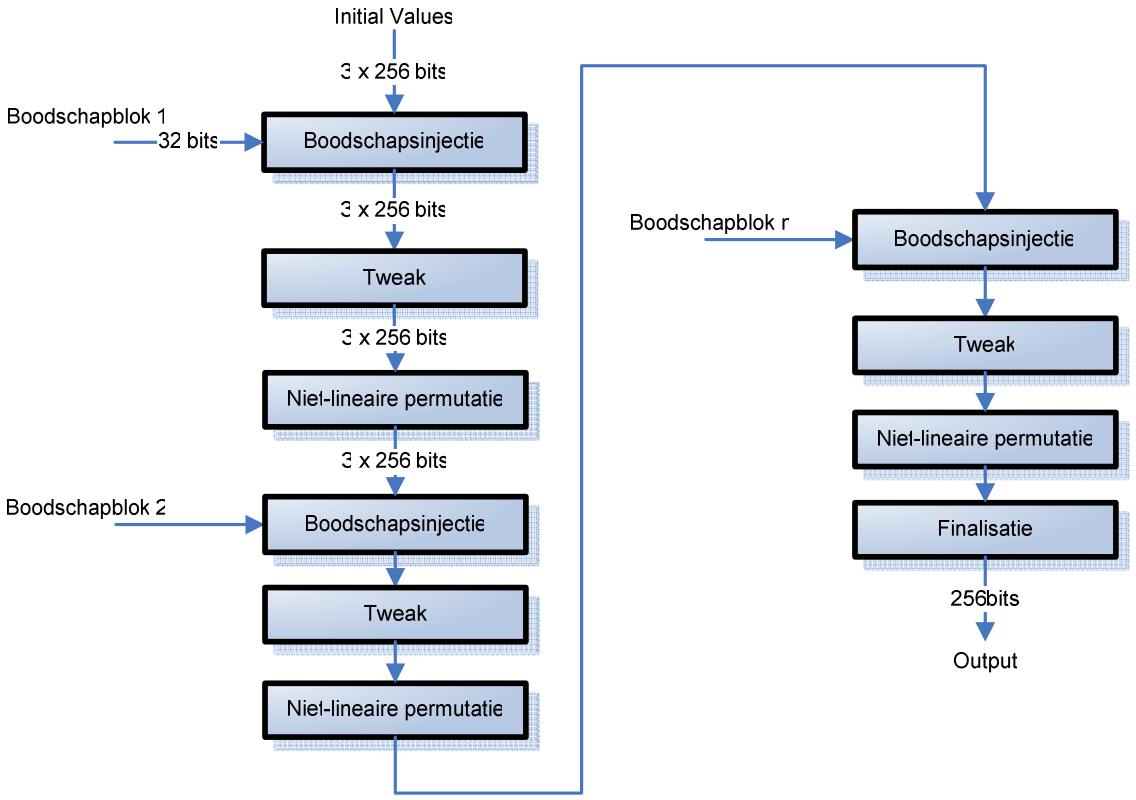
Tabel 15: De verschillen tussen de varianten van Luffa.

Variant	Lengte van de Hashwaarde	Lengte van de blokken aan de input	Aantal subpermutaties
Luffa-256	256	256	3
Luffa-512	512	512	5
Luffa-224	224	256	3
Luffa-384	384	512	4

De gebruikte initial values voor Luffa-256 zijn opgenomen in Bijlage B.

2.4.1 Dataflow van Luffa-256

Om de dataflow van Luffa-256 goed te kunnen begrijpen is er in Figuur 41 een globaal overzicht gegeven van de verschillende stappen die uitgevoerd moeten worden.



Figuur 41: Dataflow van Luffa-256

In dit schema is te zien dat iedere boodschapblok op dezelfde manier wordt gecodeerd. Enkel de gebruikte waardes veranderen telkens. In wat volgt gaan we dieper in op de werking van het geheel en van de verschillende bouwblokken.

2.4.2 Bespreking van de verschillende stappen.

De verschillende stappen die uitgevoerd moeten worden zijn:

- Boodschap in buffer zetten;
- Message padding;
- Message injectie;
- Tweak;
- Niet-lineaire permutatie;
- Finalisatie.

2.4.2.1 Boodschap in de buffer zetten

De boodschap die verwerkt moet worden door de hashfunctie 'Luffa-256' zal verwerkt worden in stukken van 256 bits. Deze 256 bits komen over één lijn binnen in de implementatie. Hierdoor is het gebruik van een buffer dus noodzakelijk. Dit heeft als voordeel dat er pas bij het laatste stuk message padding toegepast moet worden.
De boodschap stroomt binnen in de buffer. Als deze vol is, zal de inhoud van de buffer doorgestuurd worden om door de hashfunctie verwerkt te worden. Op hetzelfde moment zal de teller die telt hoeveel bits de buffer al binnengestroomd zijn gereset worden. De buffer kan nu terug bits in beginnen te lezen.

2.4.2.2 Message padding

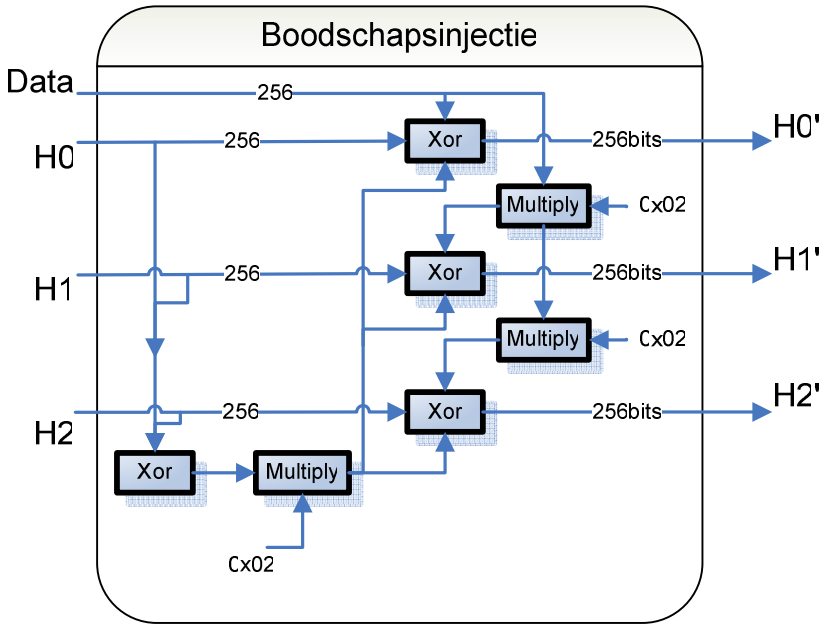
Aangezien er voor Luffa exact dezelfde message padding moet worden uitgevoerd als voor Hamsi verwijzen we hier naar paragraaf 2.2.2.2.

2.4.2.3 Message injectie

Deze blok zorgt ervoor dat de boodschap wordt geïnjecteerd in de compressiefunctie. Een multiply met 0x02 wordt gegeven door de volgende pseudo-code:

```
tmp = a[7];
a[7] = a[6];
a[6] = a[5];
a[5] = a[4];
a[4] = a[3]  xor tmp;
a[3] = a[2]  xor tmp;
a[2] = a[1];
a[1] = a[0]  xor tmp;
a[0] = tmp;
```

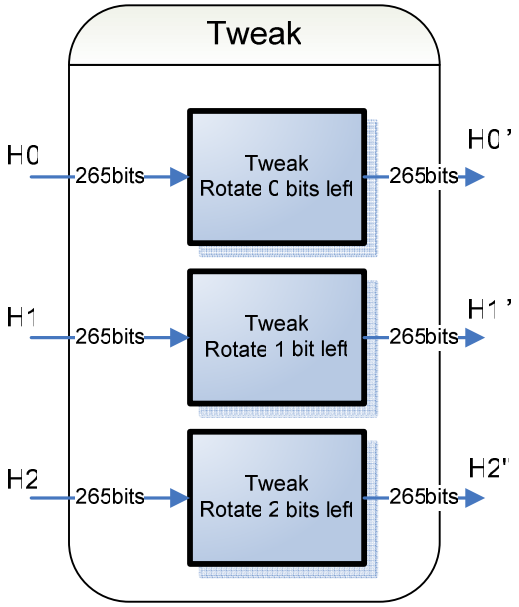
Hierbij is a[0] het meestbeduidende woord en A[7] het minstbeduidende woord.



Figuur 42: Boodschapsinjectie Luffa

2.4.2.4 Tweak

Vooraleer we de data doorheen de niet-lineaire permutatie kunnen sturen passen we eerst nog een tweak toe. De tweak gaat de 128 minst beduidende bits van elke 256 bits die uit de Message injectie komen, splitsen in blokken van 32 bits. Op elk van deze deze blokken wordt dan een rotate left uitgevoerd. Zoals eerder gezegd komen er 3 chaining values uit de message-injectie, nl H0,H1 en H2. Er wordt op deze respectievelijk een rotate left van 0,1 en 2 bits uitgevoerd.

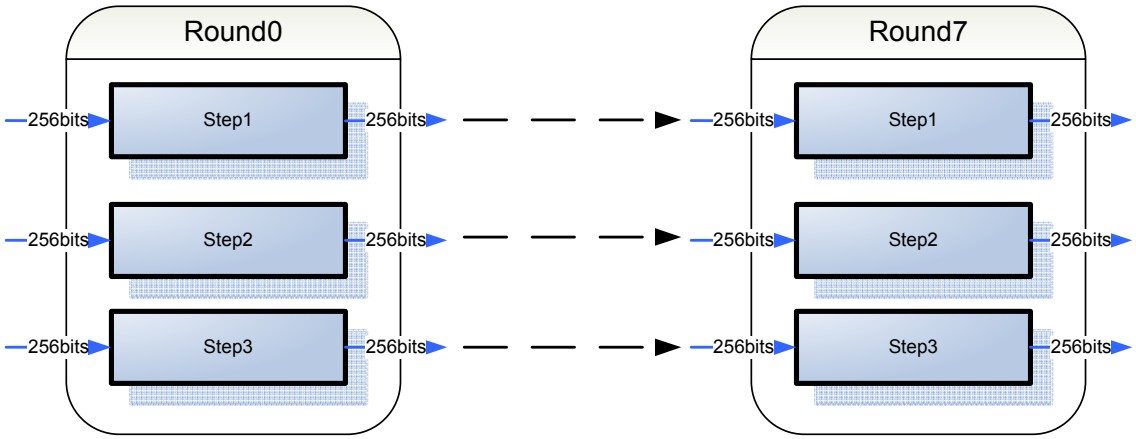


Figuur 43: Luffa Tweak

2.4.2.5 Niet-lineaire permutatie

Nadat de message injectie gebeurd is hebben we drie waardes van 256 bits die we vervolgens gaan doorsturen naar de niet-lineaire permutatie. In Figuur 44 is te zien dat deze

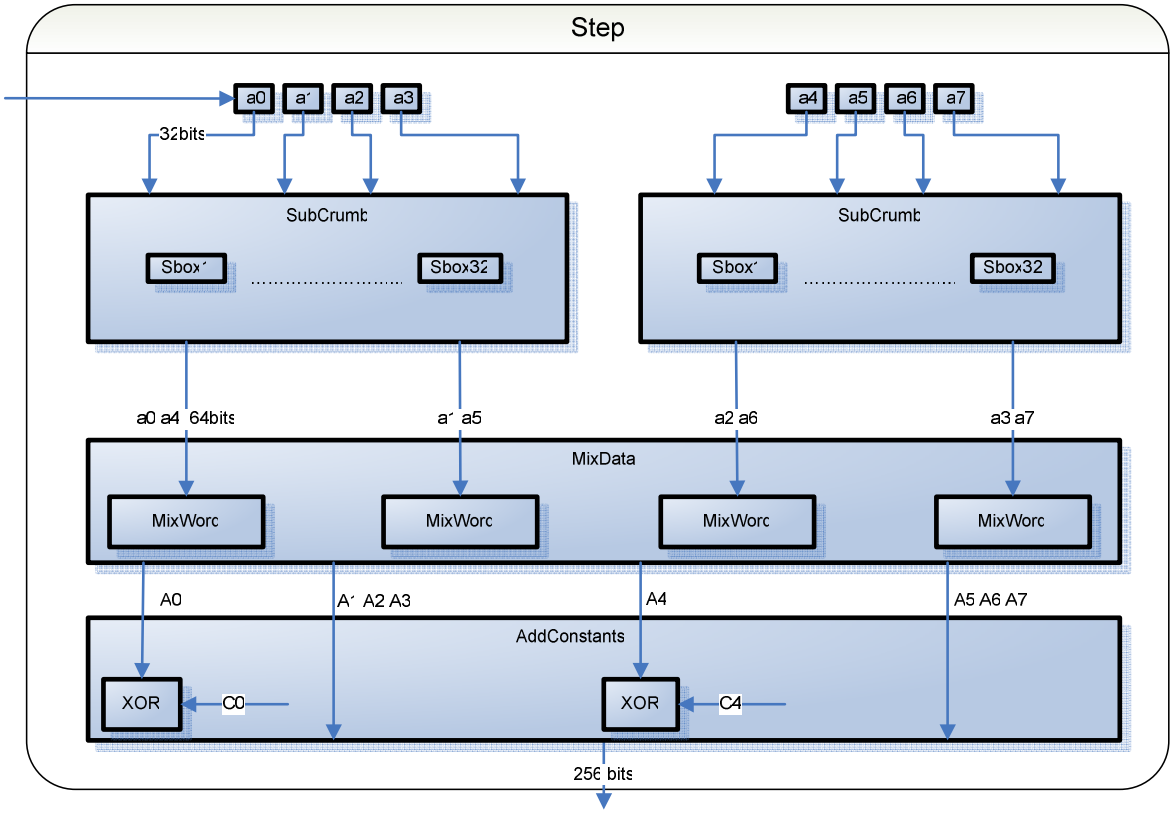
bestaat uit 8 rondes. Een ronde bestaat uit 3 subpermutatieblokken die we step-functies noemen. Deze step-functies bestaan dan weer uit een aantal blokken die we in de volgende paragrafen zullen toelichten.



Figuur 44: Niet-lineaire permutatie in Luffa

Step

In Figuur 45 zien we dat de step-functie bestaat uit een aantal blokken die de 256 bits aan de ingang zullen veranderen.



Figuur 45: Step in Luffa

We bekijken deze 256 bits als 8 woorden van 32 bits. Deze zijn in de tekening weergegeven als a0 tot en met a7 waarbij a0 het meestbeduidende woord is en a7 het minstbeduidende woord is.
In wat volgt gaan we verder in op de werking van de blokken 'SubCrumb', 'MixData' en 'AddConstants'.

Subcrumb

Subcrumb is een blok dat 128 bits aan zijn ingang heeft en bestaat uit 32 Sboxen. Aan de ingang van deze Sboxen hangen we telkens 4 bits. Deze 4 bits worden gevormd door van elk van de 4 woorden aan de ingang 1 bit te nemen. Deze bits staan telkens op dezelfde plaats in het woord. Hierdoor krijgen we een combinatie van 4 bits die de Sbox zal vervangen door een andere combinatie van 4 bits. In tabel … is te zien welke ingang tot welke uitgang leidt. Hierbij is X is de oorspronkelijke combinatie en S(x) de nieuwe.

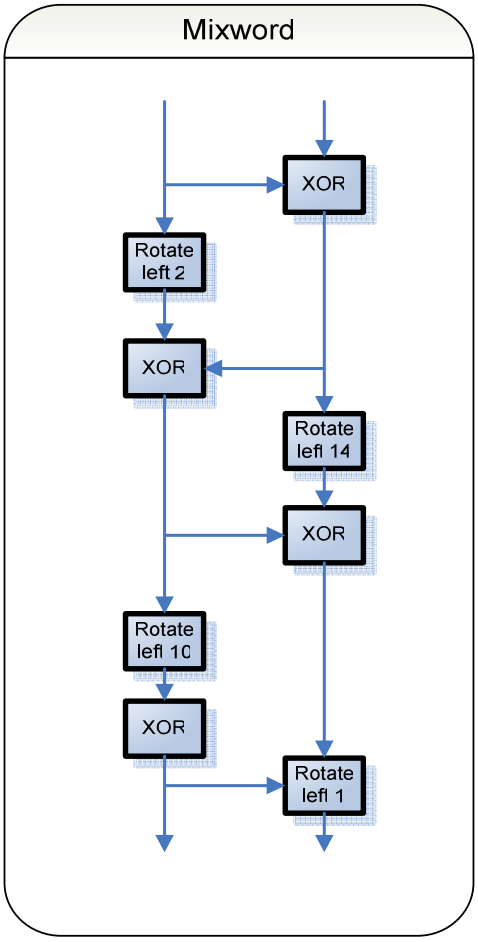
Tabel 16: Sbox in Luffa

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	D	E	0	1	5	A	7	6	B	3	9	C	F	8	2	4

Nadat de uitgang is gevormd worden de bits terug op hun oorspronkelijke plaats in elk woord gezet.We zien dat we per step-functie 2 Subcrumb-blokken nodig hebben en dus 64 Sboxen.

Mixword

Mixdata is een blok met 256 in- en uitgangen. Deze zijn verbonden met een van de vier mixword-blokken. De interne structuur dan mixword is te zien in Figuur 46



Figuur 46: Mixword in Luffa

De verbinding met de mixword-blokken gebeurt echter op een speciale manier. Wanneer we terugkijken naar Figuur 45 (step-functie) zien we dat er telkens 2 blokken van 32 bits aan de ingang van een mixwordblok hangen. Dit is enerzijds een blok van de eerste subcrumb-blok en anderzijds een blok van de tweede subcrumb-blok.

Addconstants

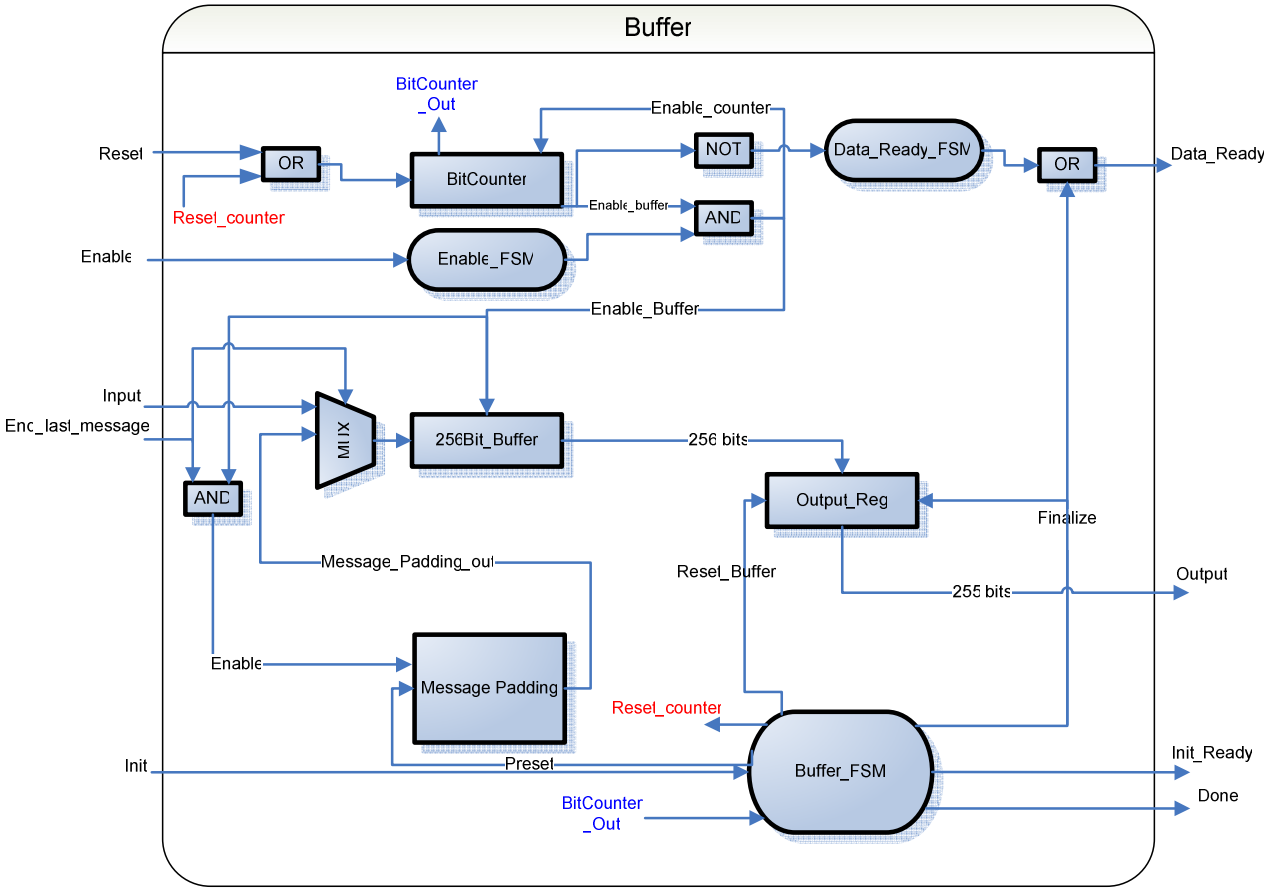
Om de werking van dit blok te kunnen begrijpen moeten we even opnieuw naar Figuur 41 gaan kijken. Hierop is te zien dat elke permutatie bestaat uit 8 rounds en iedere round bestaat op zich weer uit 3 stepfuncties. De constanten die worden toegevoegd zijn telkens 32 bits breed. Voor elke round zijn er nu 6 verschillende constanten namelijk 2 voor elke stepfunctie. In Figuur 44 zien we dat enkel A0 en A4 worden ge-xor-t met deze constanten. De rest van de data wordt dan ook onveranderd doorgestuurd. Deze constanten zijn opgenomen in Bijlage C.

2.4.2.6 Finalisatie

Wanneer de laatste boodschap door de hash-functie verwerkt is moet er nog finalisatie toegepast worden. Dit houdt in dat er nog een extra boodschapblok wordt toegevoegd. Dit is een boodschap van 256 nullen. Deze boodschap wordt dan nog eens verwerkt door de hashfunctie. Om nu nog een juiste uitgang te bekomen worden de uitgangen van de 3 subpermutatieblokken met elkaar gexored.

2.4.3 Hardware-implementatie.

2.4.3.1 Buffer



Figuur 47: Hardware-implementatie, inclusief buffer, van Luffa

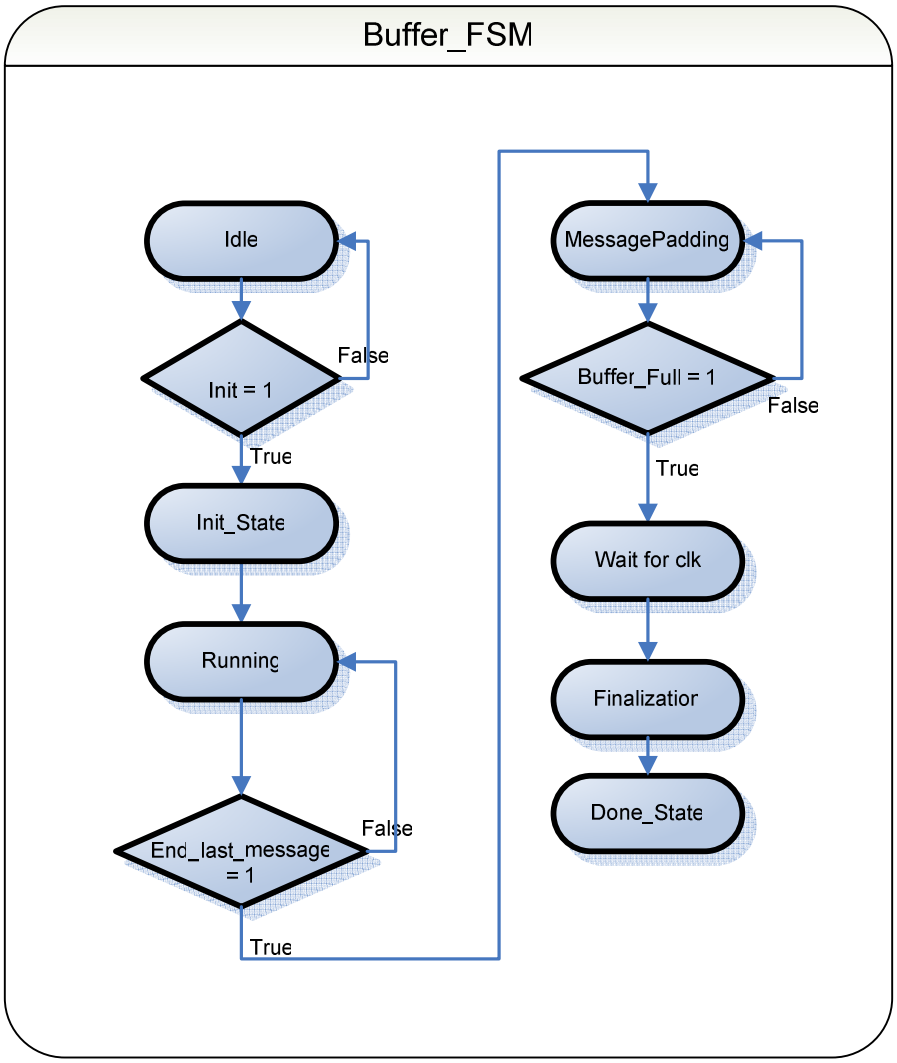
De buffer is gemaakt in 2 delen namelijk het datapad en het controlepad. In Figuur 47 is de hardware-implementatie van de buffer te zien. Hierin zijn data- en controlepad opgenomen.

De functies van de buffer zijn enerzijds de boodschap van de input serieel inladen en anderzijds de nodige messagepadding en finalisatie uitvoeren.
Hiervoor zijn natuurlijk een aantal controlesignalen nodig waardoor de buffer ook met

andere blokken kan communiceren. Zo moet het volgende blok weten wanneer de buffer vol zit en wanneer de data gelezen mag worden.

Controlepad

In wat volgt beschrijven we de finite state machines die de werking van de buffer controleren.



Figuur 48: Fsm van de buffer

In Figuur 48 geven we de werking weer van de 'Buffer_FSM'. Vervolgens gaan we dieper in op de verschillende toestanden waarin deze FSM zich kan bevinden.

Idle: In deze toestand wacht de hardware tot er een startsignaal gegeven wordt. Wanneer dit signaal gegeven wordt zal de FSM overgaan tot de initialisatie.
Initialisatie: Het presetsignaal zal hoog worden, wat ervoor zorgt dat de flipflop van de messagepadding op 1 gezet wordt. Bij de volgende klokpuls zal de toestand van de FSM 'running' worden.

Running: In deze toestand wordt er data aangelegd aan de buffer, met bij iedere bit een enable. Door deze enable zal ook de bitcounter met 1 verhogen. Wanneer de buffer vol zit

(en de bitcounter dus 256 heeft bereikt) zal het signaal ‘data_Ready’ hoog worden. Het volgende blok zal dus weten dat er data klaar staat. Wanneer echter het einde van de laatste boodschap bereikt wordt zal de ingang end_last_message hoog gemaakt worden. De FSM gaat nu naar de toestand ‘messagepadding’.

Messagepadding: Omwille van het hoog worden van end_last_message zal de ingang van de buffer niet langer verbonden zijn met de input. Door middel van een mux zal de flipflop van de messagepadding aan de ingang gehangen worden. Deze hebben we tijdens de initialisatie op ‘1’ gezet zodat de er bij de volgende klokpuls eerst een ‘1’ aan de ingang komt te hangen. Bij deze klokpuls wordt de flipflop ook op ‘0’ gezet aangezien er een ‘0’ hangt aan de ingang. De bitcounter zal na een tijd weer melden dat de flipflop vol zit waardoor de FSM overgaat naar de toestand ‘waiting0’.

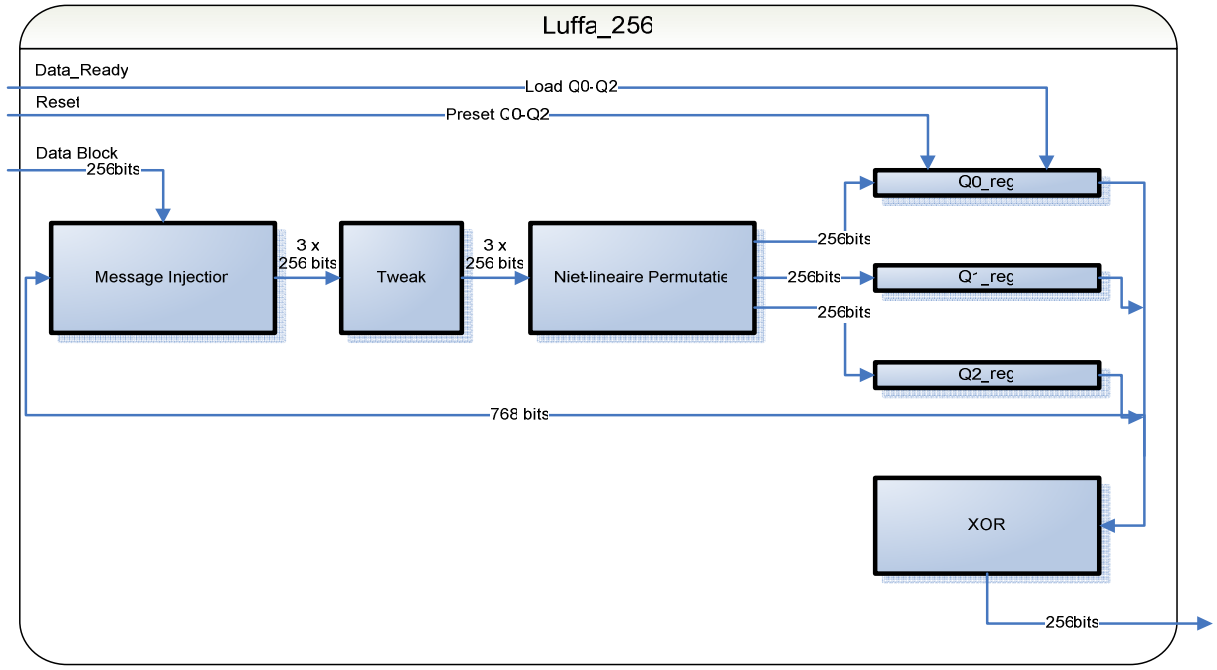
Waiting: Aangezien het data_ready signaal door de werking van de data_ready_fsm een klokpuls nodig heeft om zijn signaal laag te trekken kunnen we niet onmiddellijk naar de finalization gaan. Het data_ready signaal van deze state zou dan niet opgemerkt worden. Daarom moeten we een klokpuls wachten alvorens we naar de toestand finalization state gaan.

Finalization: In deze toestand wordt het finalize-signaal hoog waardoor enerzijds het outputregister gereset wordt en anderzijds data_ready hoog wordt. Hierdoor bekomen we een juiste finalisatie. De FSM zal hierna onmiddellijk overgaan naar de toestand done.

Done: In deze fase zal de uitgang done op ‘1’ gezet worden en wordt er verder niks meer uitgevoerd totdat er weer wordt gereset.

2.4.3.2 Hardware-equivalent geoptimaliseerd naar snelheid

Luffa_256

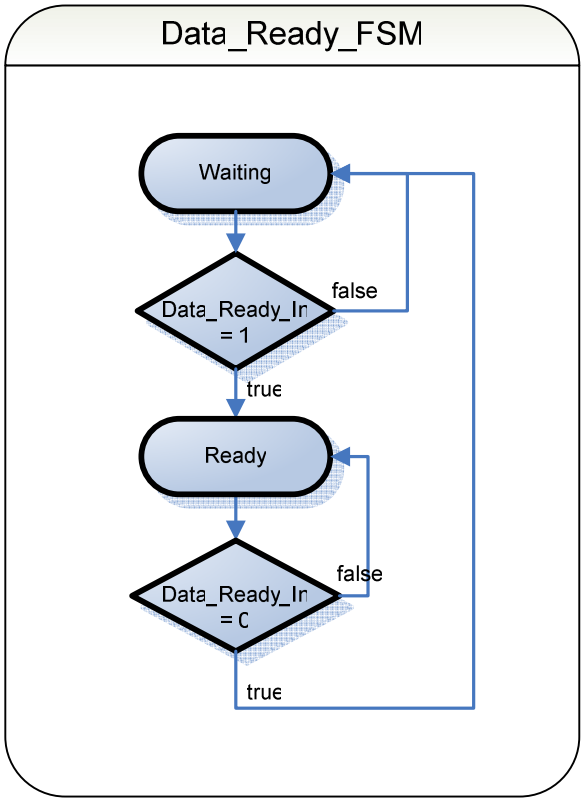


Figuur 49: Hardware-implementatie van Luffa-256 geoptimaliseerd naar snelheid

In Figuur 49 zien we de hardware-implementatie van de hashfunctie zelf. Hierin zijn 3 registers van 256 bits opgenomen. Doordat we beginnen met 3 initiële waardes H0,H1 en H2 zullen deze 3 registers eerst "gereset" moeten worden vooraleer we het eerste datablok kunnen inlezen. Deze reset is hier eerder een preset want wanneer we resetten zullen deze registers niet 0 worden maar zal hun initiële waarde geladen worden. De niet-lineaire permutatie bestaat hier uit 8 ronden die combinatorisch aan elkaar geschakeld zijn

Data_ready_FSM en Enable_FSM

Voor de werking van deze FSM's verwijzen we naar de paragraaf 'Data_ready_FSM en Enable_FSM' van Hamsi-256.



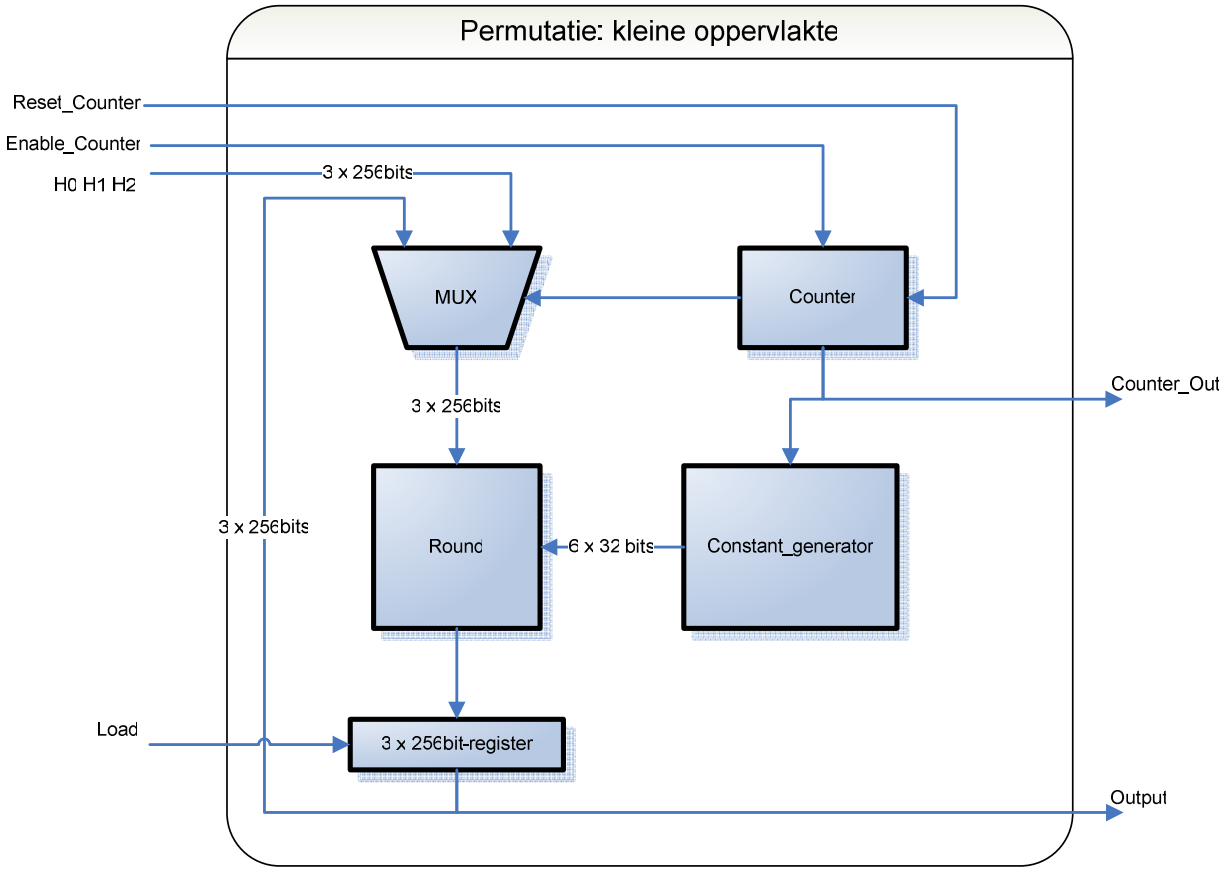
Figuur 50: Data_Ready_FSM en Enable_FSM

2.4.3.3 Hardware-implementatie geoptimaliseerd naar oppervlaktegebruik

We zullen hier eerst de implementatie van Luffa uitleggen waarna we zeer kort kunnen zijn over de aanpassingen aan de Buffer.

Luffa_256

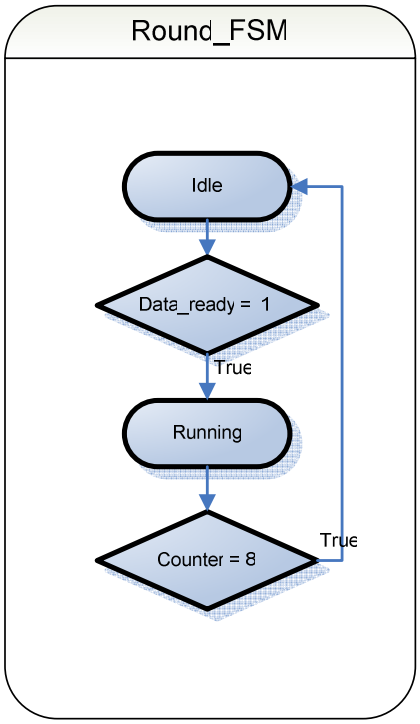
Om oppervlakte te besparen hebben we de 8 combinatorisch geschakelde ronden vervangen door één enkele ronde met 3 registers erachter. In Figuur 51 is te zien dat we een teller hebben geïmplementeerd die bijhoudt hoeveel ronden er al zijn uitgevoerd. Wanneer de eerste ronde is uitgevoerd zal deze teller ook de mux aansturen zodanig dat de uitgangen van de 3 registers verbonden zijn met de ingangen van de ronde. Als laatste stuurt deze teller ook nog het blok Constant_generator aan aangezien er bij elke ronde andere constanten nodig zijn.



Figuur 51: Hardware-implementatie van de niet-lineaire permutatie geoptimaliseerd naar oppervlaktegebruik

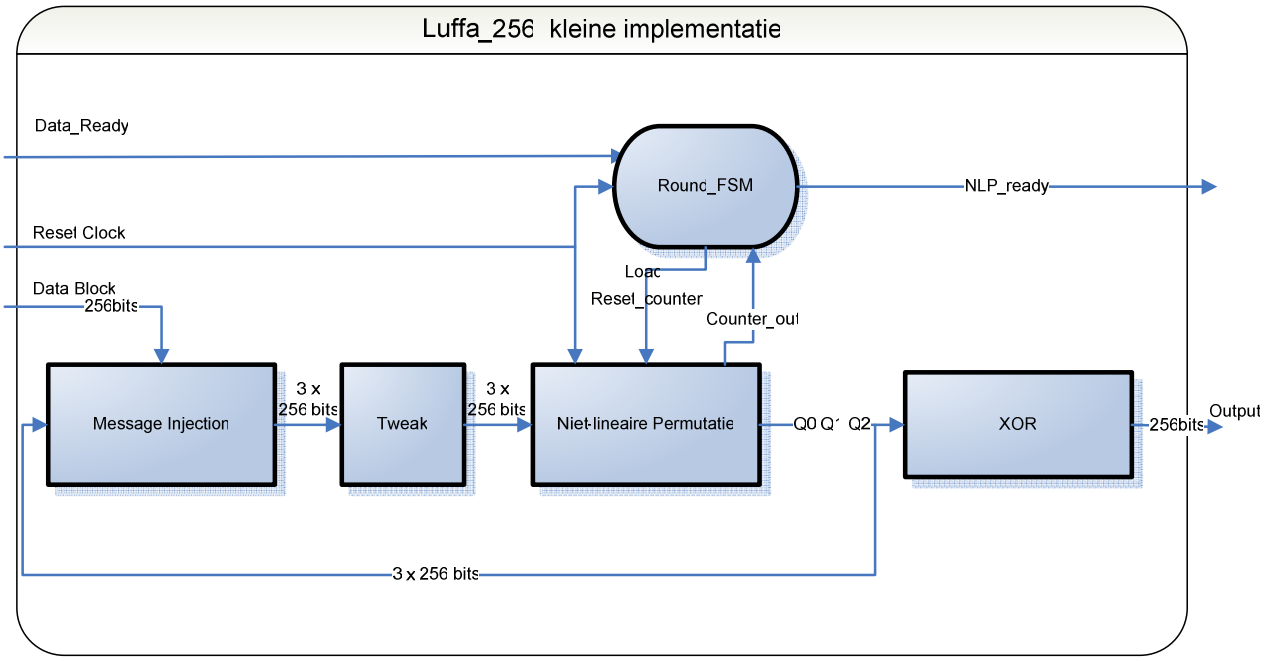
Hier is er een extra blok gedefinieerd namelijk de 'Constant_generator'. Dit is niet zozeer een echt generator maar eerder een stuk hardware waarin alle constanten zijn gedefinieerd. Aan de hand van het signaal 'counter_out' gaat deze de juiste constanten doorsturen naar het 'Round'-blok.

Wanneer we dit blok nu gaan plaatsen in Luffa-256 moeten we ook een FSM toevoegen die dit blok bestuurt. In Figuur 51 is deze FSM afgebeeld.



Figuur 52: FSM van Round

De Round_FSM heeft maar 2 toestanden. In 'Idle' is het 'load'-signaal laag dus blijven de chaining values onveranderd. Het signaal 'NLP_ready' is in deze state hoog zodat de buffer weet dat hij data mag doorsturen.
Wanneer Luffa_256 het data_Ready signaal van de buffer krijgt zal de FSM in de toestand running gaan. Nu wordt het 'load'-signaal hoog waardoor de registers elke klokpuls herladen worden en de teller optelt. Wanneer deze counter 8 heeft bereikt zal de teller gereset worden en de FSM zich terug in de toestand 'Idle' bevin



Figuur 53: Hardware-implementatie van Luffa-256 geoptimaliseerd naar oppervlakte-gebruik

Tenslotte moeten we aan de buffer nog een kleine aanpassing doen. Er is maar één klokcyclus nodig tussen de 2 pulsen van data_ready, afkomstig van de messagepadding en de finalisatie. Aangezien Luffa_256 zich in de toestand 'Running' bevindt door de eerste puls kan dit blok de tweede puls van data_ready niet opmerken. Hierdoor zouden we dus een foute werking verkrijgen. Door het signaal NLP_ready terug te koppelen naar de buffer en in de FSM van de buffer te eisen dat NLP_ready '1' is alvorens over te gaan naar de toestand 'Finalization' is dit probleem opgelost.

2.4.5 Evaluatie op basis van snelheid en oppervlaktegebruik.

Na synthese met ISE, een ontwikkeltool van Xilinx, krijgen we volgende gegevens die ons toelaten de compressiefuncties te vergelijken wat betreft snelheid en oppervlakte:

Tabel 17: Vergelijking tussen implementaties van de Luffa-compressiefunctie

Xilinx Virtex-5 FPGA	**Luffa_256 (Comb)**	**Luffa_256 (Seq)**
Aantal slices	*9611*	*2303*
Aantal Slice registers	*786*	*1165*
Aantal slice LUT's	*18918*	*4490*
Minimale periode	*20.739 ns*	*5.568 ns*
Geschatte maximale klokfrequentie	*48.2 MHz*	*179 MHz*
Aantal klokcycli per compressie	*1*	*9*
Tijd voor 1 compressie	*20.739 ns*	*44.544 ns*
Doorvoercapaciteit	*12.29 Gbps*	*5.09 Gbps*
Doorvoercapaciteit /oppervlakte	*1.28 Mbps/slice*	*2.21 Mbps/slice*

We kunnen zien dat de tweede implementatie van Luffa_256 duidelijk veel minder oppervlakte nodig heeft dan de eerste (4.2 keer minder). De minimale periode is ook ongeveer 4 keer zo klein geworden en de maximale klokfrequentie dus bijna 4 keer groter . Ook het aantal benodigde klokcycli is 9 keer groter bij de tweede implementatie.

We zien dat de doorvoercapaciteit gehalveerd is maar dat de doorvoercapaciteit per oppervlakte wel bijna verdubbeld is. Dit is voor een groot stuk te danken is aan een korter kritisch pad. Dit kritisch pad is namelijk bijna 8 keer kleiner geworden door de 8 ronden te vervangen door 1 ronde.

3 Vergelijking implementaties

De beste implementaties uit voorgaande hoofdstukken zijn samengebracht in Tabel 18 voor de kleine, en Tabel 19 voor de snelle implementaties.

Tabel 18: Vergelijking van de kleinste implementaties

Xilinx Virtex-5 FPGA	**LANE_nieuw**	**Hamsi [13]**	**ECHO Geoptimaliseerd**	**Luffa_ Seq**
Aantal slices	*3389*	*733*	*12061*	*2303*
Aantal Slice registers	*1307*	*/*	*8800*	*1165*
Aantal slice LUT's	*4078*	*/*	*14407*	*4490*
Minimale periode	*5.622 ns*	*3.484 ns*	*5.333 ns*	*5.568 ns*
Geschatte maximale klokfrequentie	*177 MHz*	*287 MHz*	*187 MHz*	*179 MHz*
Aantal klokcycli per compressie	*61*	*7*	*81*	*9*
Tijd voor 1 compressie	*343 ns*	*24 ns*	*432 ns*	*44.544 ns*
Doorvoercapaciteit	*1.48 Gbps*	*1.48 Gbps*	*3.56 Gbps*	*5.09 Gbps*
Doorvoercapaciteit /oppervlakte	*0.43 Mbps/slice*	*2 Mbps/slice*	*0.30 Mbps/slice*	*2.21 Mbps/slice*

In Tabel 18 is te zien dat Hamsi [13] met grote voorsprong de kleinste implementatie is. Wanneer we ook de snelheid in rekening gaan brengen zien we dat Luffa_seq de beste keuze is. Het valt ook direct op dat ECHO zich niet leent om zeer kleine implementaties te maken. LANE_nieuw is niet veel groter dan Luffa_seq, maar wel een stuk trager.

Tabel 19: Vergelijking van de snelste implementaties

Xilinx Virtex-5 FPGA	**LANE_oud**	**Hamsi_fast**	**ECHO standaard**	**Luffa_comb**
Aantal slices	*8228*	*4,664*	*15006*	*9611*
Aantal Slice registers	*2111*	*514*	*4105*	*786*
Aantal slice LUT's	*16600*	*7216*	*29330*	*18918*
Minimale periode	*4.067 ns*	*4.826*	*7.154 ns*	*20.739 ns*
Geschatte maximale klokfrequentie	*245 MHz*	*207 MHz*	*139 MHz*	*48.2 MHz*
Aantal klokcycli per compressie	*15*	*1*	*9*	*1*
Tijd voor 1 compressie	*61 ns*	*24 ns*	*64 ns*	*20.739 ns*
Doorvoercapaciteit	*8.36 Gbps*	*6.62 Gbps*	*23.86 Gbps*	*12.29 Gbps*
Doorvoercapaciteit /oppervlakte	*1.01 Mbps/slice*	*1.42 Mbps/slice*	*1.59 Mbps/slice*	*1.28 Mbps/slice*

Wanneer we de implementaties op basis van snelheid beoordelen, komt ECHO standaard er als de beste implementatie uit. Voor deze hoge snelheid wordt er wel een hoge prijs betaald wat betreft oppervlaktegebruik. Verder merken we op dat er voor de andere compressiefuncties een positief verband is tussen snelheid en oppervlaktegebruik.

Besluit

Uit de door ons vergeleken implementaties kunnen we verschillende besluiten trekken.

Wanneer we enkel naar oppervlaktegebruik kijken geeft Hamsi[13] de beste implementatie. Ook wanneer we kijken naar de snelle implementatie van Hamsi geeft dit qua oppervlaktegebruik nog een zeer goed resultaat. We kunnen dit ook duidelijk zien aan de hoge efficiëntie bij beide implementaties.

Op basis van snelheid kunnen we besluiten dat ECHO_standaard met grote voorsprong de beste implementatie is. Wanneer we hierbij echter ook het oppervlaktegebruik in rekening brengen merken we dat er niet zo veel verschil is tussen de verschillende implementaties. Het is duidelijk dat er een positief verband tussen snelheid en oppervlaktegebruik is.

Wanneer we globaal kijken naar de snelle en kleine implementaties, en enkel naar de doorvoercapaciteit per oppervlakte kijken, kunnen we besluiten dat Luffa_seq er als beste uikomt. Wanneer we echter de resultaten van Hamsi bekijken zien we dat zowel de kleine als de snelle implementatie een zeer goede efficiëntie heeft en het oppervlaktegebruik lager is dan bij beide implementaties van Luffa.

Het is zeer moeilijk om op basis van deze resultaten de beste functie aan te duiden aangezien dit zeer afhankelijk is van de toepassing. Dit eindwerk toont wel duidelijk de positieve en negatieve punten wat betreft de implementatie in hardware van verschillende SHA-3-kandidaten. Deze gegevens kunnen gebruikt worden voor verder onderzoek.

Literatuurlijst

[1] N. Mentens, “Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs,” PhD thesis, Katholieke Universiteit Leuven, Leuven-Heverlee, België, 2007.

[2] J. Thielen, D. Rykx, “Evaluatie van nieuwe hashfunctie kandidaten op FPGA,” M.S. thesis, Katholieke Hogeschool Limburg, Diepenbeek, België, 2009.

[3] A. Menezes, P. van Oorschot, S. Vanstone, “Handbook of Applied Cryptography,” Overview of Cryptography, CRC Press, 1996, hoofdstuk 1 + 9.

[4] M. Vandenwauver, Katholieke Universiteit Leuven, Laboratorium ESAT-Groep COSIC, “Introduction to Cryptography,” http://www.esat.kuleuven.be/cosic/intro/.

[5] J. Daemen, V. Rijmen, “The design of Rijndael: AES --- the Advanced Encryption Standard,” Springer-Verlag, 2002, ISBN 3-540-42580-2.

[6] Wikipedia, “Birthday paradox,” geraadpleegd op 20 april 2010, http://en.wikipedia.org/wiki/Birthday_paradox.

[7] A. Regenscheid, R. Perlner, S. Chang, J. Kelsey, M. Nandi, S. Paul, “Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition,” September 2009, http://csrc.nist.gov/groups/ST/hash/sha3/Round1/documents/sha3_NISTIR7620.pdf.

[8] K. Matusiewicz, M. Naya-Plasencia, I. Nikolic, Y. Sasaki, M. Schläffer, “Rebound Attack on the Full LANE Compression Function,” ASIACRYPT'09; LNCS 5912 Tokyo, Springer, 2009 (online preprint: http://eprint.iacr.org/2009/443.pdf).

[9] National Institute of Standards and Technology, Computer security division: Computer Security Resource Center, April 2005, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[10] S. Indesteege, “The LANE hash function,” Oktober 2008, http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf.

[11] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan and W. P. Marnane, “FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash,” 2009, http://eprint.iacr.org/2009/342.pdf.

[12] Ö. Küçük, “The Hash Function Hamsi,” September 2009, http://www.cosic.esat.kuleuven.be/publications/article-1203.pdf

[13] J. Fan, “Hardware Evaluation of The Hash Function Hamsi”, http://www.cosic.esat.kuleuven.be/publications/article-1322.pdf.

[14] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, Y. Seurin, “SHA-3 Proposal: ECHO,” Oktober 2008, http://crypto.rd.francetelecom.com/echo/doc/echo_description_1-5.pdf.

[15] C. De Cannière, H. Sato, D. Watanabe, “Hash Function Luffa”,Oktober 2009, http://csrc.nist.gov/groups/ST/hash/sha3/Round2/documents/Luffa_Round2_Update.zip.

Bijlagen

Bijlage A: Constanten van de AddConstantstransformatie van LANE-256

k0 = 07fc703dx, k1 = d3fe381fx, k2 = b9ff1c0ex, k3 = 5cff8e07x,
k4 = fe7fc702x, k5 = 7f3fe381x, k6 = ef9ff1c1x, k7 = a7cff8e1x,
k8 = 83e7fc71x, k9 = 91f3fe39x, k10 = 98f9ff1dx, k11 = 9c7cff8fx,
k12 = 9e3e7fc6x, k13 = 4f1f3fe3x, k14 = f78f9ff0x, k15 = 7bc7cff8x,
k16 = 3de3e7fcx, k17 = 1ef1f3fex, k18 = 0f78f9ffx, k19 = d7bc7cfex,
k20 = 6bde3e7fx, k21 = e5ef1f3ex, k22 = 72f78f9fx, k23 = e97bc7cex,
k24 = 74bde3e7x, k25 = ea5ef1f2x, k26 = 752f78f9x, k27 = ea97bc7dx,
k28 = a54bde3fx, k29 = 82a5ef1ex, k30 = 4152f78fx, k31 = f0a97bc6x,
k32 = 7854bde3x, k33 = ec2a5ef0x, k34 = 76152f78x, k35 = 3b0a97bcx,
k36 = 1d854bdex, k37 = 0ec2a5efx, k38 = d76152f6x, k39 = 6bb0a97bx,
k40 = e5d854bcx, k41 = 72ec2a5ex, k42 = 3976152fx, k43 = ccbb0a96x,
k44 = 665d854bx, k45 = e32ec2a4x, k46 = 71976152x, k47 = 38cbb0a9x,
k48 = cc65d855x, k49 = b632ec2bx, k50 = 8b197614x, k51 = 458cbb0ax,
k52 = 22c65d85x, k53 = c1632ec3x, k54 = b0b19760x, k55 = 5858cbb0x,
k56 = 2c2c65d8x, k57 = 161632ecx, k58 = 0b0b1976x, k59 = 05858cbbx,
k60 = d2c2c65cx, k61 = 6961632ex, k62 = 34b0b197x, k63 = ca5858cax,
k64 = 652c2c65x, k65 = e2961633x, k66 = a14b0b18x, k67 = 50a5858cx,
k68 = 2852c2c6x, k69 = 14296163x, k70 = da14b0b0x, k71 = 6d0a5858x,
k72 = 36852c2cx, k73 = 1b429616x, k74 = 0da14b0bx, k75 = d6d0a584x,
k76 = 6b6852c2x, k77 = 35b42961x, k78 = cada14b1x, k79 = b56d0a59x,
k80 = 8ab6852dx, k81 = 955b4297x, k82 = 9aada14ax, k83 = 4d56d0a5x,
k84 = f6ab6853x, k85 = ab55b428x, k86 = 55aada14x, k87 = 2ad56d0ax,
k88 = 156ab685x, k89 = dab55b43x, k90 = bd5aada0x, k91 = 5ead56d0x,
k92 = 2f56ab68x, k93 = 17ab55b4x, k94 = 0bd5aadax, k95 = 05ead56dx,
k96 = d2f56ab7x, k97 = b97ab55ax, k98 = 5cbd5aadx, k99 = fe5ead57x,
k100 = af2f56aax, k101 = 5797ab55x, k102 = fbcbd5abx, k103 = ade5ead4x,
k104 = 56f2f56ax, k105 = 2b797ab5x, k106 = c5bcbd5bx, k107 = b2de5eacx,
k108 = 596f2f56x, k109 = 2cb797abx, k110 = c65bcbd4x, k111 = 632de5eax,
k112 = 3196f2f5x, k113 = c8cb797bx, k114 = b465bcbcx, k115 = 5a32de5ex,
k116 = 2d196f2fx, k117 = c68cb796x, k118 = 63465bcbx, k119 = e1a32de4x,
k120 = 70d196f2x, k121 = 3868cb79x, k122 = cc3465bdx, k123 = b61a32dfx,
k124 = 8b0d196ex, k125 = 45868cb7x, k126 = f2c3465ax, k127 = 7961a32dx,
k128 = ecb0d197x, k129 = a65868cax, k130 = 532c3465x, k131 = f9961a33x,
k132 = accb0d18x, k133 = 5665868cx, k134 = 2b32c346x, k135 = 159961a3x,

k136 = daccb0d0x, k137 = 6d665868x, k138 = 36b32c34x, k139 = 1b59961ax,
k140 = 0daccb0dx, k141 = d6d66587x, k142 = bb6b32c2x, k143 = 5db59961x,
k144 = fedaccb1x, k145 = af6d6659x, k146 = 87b6b32dx, k147 = 93db5997x,
k148 = 99edaccax, k149 = 4cf6d665x, k150 = f67b6b33x, k151 = ab3db598x,
k152 = 559edaccx, k153 = 2acf6d66x, k154 = 1567b6b3x, k155 = dab3db58x,
k156 = 6d59edacx, k157 = 36acf6d6x, k158 = 1b567b6bx, k159 = ddab3db4x,
k160 = 6ed59edax, k161 = 376acf6dx, k162 = cbb567b7x, k163 = b5dab3dax,
k164 = 5aed59edx, k165 = fd76acf7x, k166 = aebb567ax, k167 = 575dab3dx,
k168 = fbaed59fx, k169 = add76acex, k170 = 56ebb567x, k171 = fb75dab2x,
k172 = 7dbaed59x, k173 = eedd76adx, k174 = a76ebb57x, k175 = 83b75daax,
k176 = 41dbaed5x, k177 = f0edd76bx, k178 = a876ebb4x, k179 = 543b75dax,
k180 = 2a1dbaedx, k181 = c50edd77x, k182 = b2876ebax, k183 = 5943b75dx,
k184 = fca1dbafx, k185 = ae50edd6x, k186 = 572876ebx, k187 = fb943b74x,
k188 = 7dca1dbax, k189 = 3ee50eddx, k190 = cf72876fx, k191 = b7b943b6x,
k192 = 5bdca1dbx, k193 = fdee50ecx, k194 = 7ef72876x, k195 = 3f7b943bx,
k196 = cfbdca1cx, k197 = 67dee50ex, k198 = 33ef7287x, k199 = c9f7b942x,
k200 = 64fbdca1x, k201 = e27dee51x, k202 = a13ef729x, k203 = 809f7b95x,
k204 = 904fbdcbx, k205 = 9827dee4x, k206 = 4c13ef72x, k207 = 2609f7b9x,
k208 = c304fbddx, k209 = b1827defx, k210 = 88c13ef6x, k211 = 44609f7bx,
k212 = f2304fbcx, k213 = 791827dex, k214 = 3c8c13efx, k215 = ce4609f6x,
k216 = 672304fbx, k217 = e391827cx, k218 = 71c8c13ex, k219 = 38e4609fx,
k220 = cc72304ex, k221 = 66391827x, k222 = e31c8c12x, k223 = 718e4609x,
k224 = e8c72305x, k225 = a4639183x, k226 = 8231c8c0x, k227 = 4118e460x,
k228 = 208c7230x, k229 = 10463918x, k230 = 08231c8cx, k231 = 04118e46x,
k232 = 0208c723x, k233 = d1046390x, k234 = 688231c8x, k235 = 344118e4x,
k236 = 1a208c72x, k237 = 0d104639x, k238 = d688231dx, k239 = bb44118fx,
k240 = 8da208c6x, k241 = 46d10463x, k242 = f3688230x, k243 = 79b44118x,
k244 = 3cda208cx, k245 = 1e6d1046x, k246 = 0f368823x, k247 = d79b4410x,
k248 = 6bcda208x, k249 = 35e6d104x, k250 = 1af36882x, k251 = 0d79b441x,
k252 = d6bcda21x, k253 = bb5e6d11x, k254 = 8daf3689x, k255 = 96d79b45x,
k256 = 9b6bcda3x, k257 = 9db5e6d0x, k258 = 4edaf368x, k259 = 276d79b4x,
k260 = 13b6bcdax, k261 = 09db5e6dx, k262 = d4edaf37x, k263 = ba76d79ax,
k264 = 5d3b6bcdx, k265 = fe9db5e7x, k266 = af4edaf2x, k267 = 57a76d79x,
k268 = fbd3b6bdx, k269 = ade9db5fx, k270 = 86f4edaex, k271 = 437a76d7x,
k272 = f1bd3b6ax

Bron: [2]

Bijlage B: Initial values voor Luffa-256

H0 = 6d251e6944b051e04eaa6fb4dbf784656e29201190152df4ee058139def610bb

H1 = c3b44b95d9d2f25670eee9a0de099fa35d9b05578fc944b3cf1ccf0e746cd581

H2 = 7efc89d5dba578104016ce5ad659c050306194f666d183624aa230a8b264ae7

Bron: [15]

Bijlage C: Constanten van de AddConstantstransformatie van Luffa-256

In $c_{(s,c)}^{(r)}$ staan de indexen en r,s en c respectievelijk voor het nummer van de round, de step-functie, en welke van de 2 constantes er bedoeld wordt.

$$
\begin{array}{ll}
c_{0,0}^{(0)} = \texttt{0x303994a6}, & c_{0,4}^{(0)} = \texttt{0xe0337818} \\
c_{0,0}^{(1)} = \texttt{0xc0e65299}, & c_{0,4}^{(1)} = \texttt{0x441ba90d} \\
c_{0,0}^{(2)} = \texttt{0x6cc33a12}, & c_{0,4}^{(2)} = \texttt{0x7f34d442} \\
c_{0,0}^{(3)} = \texttt{0xdc56983e}, & c_{0,4}^{(3)} = \texttt{0x9389217f} \\
c_{0,0}^{(4)} = \texttt{0x1e00108f}, & c_{0,4}^{(4)} = \texttt{0xe5a8bce6} \\
c_{0,0}^{(5)} = \texttt{0x7800423d}, & c_{0,4}^{(5)} = \texttt{0x5274baf4} \\
c_{0,0}^{(6)} = \texttt{0x8f5b7882}, & c_{0,4}^{(6)} = \texttt{0x26889ba7} \\
c_{0,0}^{(7)} = \texttt{0x96e1db12}, & c_{0,4}^{(7)} = \texttt{0x9a226e9d}
\end{array}
$$

$$
\begin{array}{ll}
c_{1,0}^{(0)} = \texttt{0xb6de10ed}, & c_{1,4}^{(0)} = \texttt{0x01685f3d} \\
c_{1,0}^{(1)} = \texttt{0x70f47aae}, & c_{1,4}^{(1)} = \texttt{0x05a17cf4} \\
c_{1,0}^{(2)} = \texttt{0x0707a3d4}, & c_{1,4}^{(2)} = \texttt{0xbd09caca} \\
c_{1,0}^{(3)} = \texttt{0x1c1e8f51}, & c_{1,4}^{(3)} = \texttt{0xf4272b28} \\
c_{1,0}^{(4)} = \texttt{0x707a3d45}, & c_{1,4}^{(4)} = \texttt{0x144ae5cc} \\
c_{1,0}^{(5)} = \texttt{0xaeb28562}, & c_{1,4}^{(5)} = \texttt{0xfaa7ae2b} \\
c_{1,0}^{(6)} = \texttt{0xbaca1589}, & c_{1,4}^{(6)} = \texttt{0x2e48f1c1} \\
c_{1,0}^{(7)} = \texttt{0x40a46f3e}, & c_{1,4}^{(7)} = \texttt{0xb923c704}
\end{array}
$$

$$
\begin{array}{ll}
c_{2,0}^{(0)} = \texttt{0xfc20d9d2}, & c_{2,4}^{(0)} = \texttt{0xe25e72c1} \\
c_{2,0}^{(1)} = \texttt{0x34552e25}, & c_{2,4}^{(1)} = \texttt{0xe623bb72} \\
c_{2,0}^{(2)} = \texttt{0x7ad8818f}, & c_{2,4}^{(2)} = \texttt{0x5c58a4a4} \\
c_{2,0}^{(3)} = \texttt{0x8438764a}, & c_{2,4}^{(3)} = \texttt{0x1e38e2e7} \\
c_{2,0}^{(4)} = \texttt{0xbb6de032}, & c_{2,4}^{(4)} = \texttt{0x78e38b9d} \\
c_{2,0}^{(5)} = \texttt{0xedb780c8}, & c_{2,4}^{(5)} = \texttt{0x27586719} \\
c_{2,0}^{(6)} = \texttt{0xd9847356}, & c_{2,4}^{(6)} = \texttt{0x36eda57f} \\
c_{2,0}^{(7)} = \texttt{0xa2c78434}, & c_{2,4}^{(7)} = \texttt{0x703aace7}
\end{array}
$$

Bron: [15]