Inhoudsopgave

Lijst van Figuren

Lijst van Tabellen

Lijst van Grafieken

Verklarende woorden- en symbolenlijst

XOR

"exclusieve ofpoort": dit is een logische poort met de volgende waarheidstabel die geldt voor 2 ingangen en 1 uitgang:

In1	In2	Uit
0	0	0
0	1	1
1	0	1
1	1	0

Deze bewerking wordt in de tekeningen en vergelijkingen met een "$\oplus$" aangeduid. Deze poort kan uitgebreid worden naar *n* aantal bits.

AND

"enpoort": dit is een logische poort met de volgende waarheidstabel die geldt voor 2 ingangen en 1 uitgang:

In1	In2	Uit
0	0	0
0	1	0
1	0	0
1	1	1

Deze bewerking wordt in de tekeningen met een "  " en "  " aangeduid en in de vergelijkingen met een " * ". Deze poort kan uitgebreid worden naar *n* aantal bits.

FSM

"*Finite State Machine*": deze machine biedt de VHDL-programmeur de mogelijkheid om processen te ontwerpen die op basis van zelf geformuleerde voorwaarden verschillende stappen kunnen doorlopen. Er is een terugkoppeling voorzien die de achterliggende hardware constant vernieuwt en op basis van deze vernieuwing een signaal naar de uitgang stuurt.

Samenvoegings-symbool

"◊", dit stelt een operatie voor die bits samenvoegt tot één geheel.

Schuifsymbool

"<<" en ">>", dit stelt een operatie voor die een rij bits bitsgewijs opschuift naar links of naar rechts.

Rotatiesymbool

"<<< en ">>>", dit stelt een operatie voor die een rij bits laat ronddraaien in een daarvoor voorzien register. Het aantal plaatsen dat er verschoven moet worden, wordt achter het symbool geplaatst. Dit kan zowel naar links als naar rechts.

NAND — dit is de “inverse van een logische enoperatie”. Voor deze logische poort geldt de volgende waarheidstabel voor 2 ingangen en 1 uitgang:

In1	In2	Uit
0	0	1
0	1	1
1	0	1
1	1	0

Deze poort kan uitgebreid worden naar *n* aantal bits.

Throughput — de snelheid waarmee de bits doorheen de hardware gestuurd worden, noemen we throughput. Deze wordt ook wel doorvoersnelheid genoemd en wordt uitgedrukt in gigabits/seconde. (Gbit/sec)

Radix — is het aantal uitgestuurde bits dat per klokpuls als sleutelstroom dient.

Dankwoord

Als laatstejaarsstudenten binnen de opleiding van Industrieel Ingenieur Elektronica optie Ontwerpen aan de Katholieke Hogeschool Limburg hebben we gekozen voor een meesterproef rond implementaties in hardware. Dit project werd ons aangeboden door het onderzoeksinstituut COSIC te Leuven, een referentie in haar vakgebied.

We wisten dat het een uitdaging zou worden en gedurende het jaar van de meesterproef botsten we regelmatig tegen een muur van problemen waarbij we soms teleurstellingen opliepen. We willen de mensen die ons met raad en daad bijstonden bij het tot stand komen van deze meesterproef vermelden en onze dank betuigen.

Eerst en vooral bedanken we COSIC voor de aangeboden kans om dit project te mogen realiseren. Specifiek moeten we Nele Mentens, Joe Lano en An Braeken vermelden, omdat ze ons professioneel bijgestaan hebben in onze zoektocht, ondanks hun eigen bezigheden. Door hun bijdrage konden we dit project, praktijkgericht en theoretisch onderbouwd, in de goede richting sturen.

Natuurlijk vergeten we het vakkundige taaladvies van onze docent aan de KHLim, Ivy Verbeeck, niet. Zij heeft de zware taak van wijlen Mr. Steverlinck overgenomen met de gedrevenheid die de studenten nodig hadden.

Ook onze ouders, vrienden en vriendinnen verdienen hier een plekje. Zonder hun steun en begrip zou ons dit niet gelukt zijn. We zijn hen dan ook erg dankbaar.

Ons laatste dankwoord richten we naar elkaar: door de vele maanden van samenwerking en afspraken zijn we tot een mooi eindproduct gekomen. We vonden als team de inspiratie en motivatie om verder te werken. Zonder de waardevolle discussies en het vele denkwerk zou deze meesterproef niet geworden zijn wat ze nu is.

Diepenbeek, 15 mei 2006

Voorwoord

Onze meesterproef kadert in een groot project dat de huidige manier van datacodering wil vernieuwen. Dit project is opgedeeld naar twee grote ontwerpdoelgroepen toe: meer veiligheid brengen voor de codering in software en optimalisatie qua oppervlakte en snelheid voor de codering in hardware. Als toekomstige Industrieel Ingenieurs van de optie Elektronica Ontwerpen zijn we altijd geïnteresseerd in projecten die hardwarematig ontworpen dienen te worden. De keuze om deze meesterproef te nemen was dus interessant en uitdagend.

Na enkele maanden van implementeren, simuleren en interpreteren van de testresultaten hebben we een beter inzicht gekregen in de wereld van coderingshardware. Het was een leerrijke tijd waarin we stelselmatig onze bijdrage aan het project onderdeel per onderdeel bestudeerd en uitgebouwd hebben.

We hopen deze positieve ervaring nog vaak in ons toekomstig beroepsleven tegen te komen en toe te passen, waar nodig.

Inleiding

Aanleiding en evolutie van de beveiliging van digitale communicatie

Digitale elektronica is alom tegenwoordig in ons dagelijkse leven. Digitale transmissie van data wordt steeds belangrijker, zowel voor onze private als professionele communicatie. Alle vormen van datatransmissie en dataopslag moeten beveiligd kunnen worden tegen misbruik. Het beveiligen van informatie gebeurt door ze onleesbaar te maken met behulp van coderingsalgoritmes.

Het coderen van informatie heeft een lange en fascinerende geschiedenis achter de rug: Julius Caesar liet zijn geheime documenten versleutelen door elke letter te vervangen door de letter drie plaatsen verder in het alfabet. De letter A werd voorgesteld in de D, B werd zo een E enz. Dit lijkt op zich een eenvoudige en doeltreffende methode, maar wanneer een derde persoon de versleutelde tekst onderzoekt, zal deze snel opmerken dat H (die E voorstelt) en D (die A voorstelt) vaker voorkomen dan de andere letters. De stap naar het zoeken van de statistieken van de letters in het alfabet is dan snel gezet. Eens het algoritme, de manier waarop de codering in elkaar zit, gevonden is, wordt de oorspronkelijke codering waardeloos en moet men op zoek naar een betere code. Dit proces van ontwerpen en analyseren van nieuwe algoritmen leidde ertoe dat de mens steeds vernuftigere systemen uitvond om zijn informatie te beschermen.

Door de ontwikkelingsgroei van computers en elektronische communicatiesystemen sinds de jaren ‘60 en de latere massaverspreiding van deze technologie, vormde er zich een dringende vraag naar betere informatiebescherming vanuit de privé-sector.
De ICT-revolutie verlegde de grenzen van het kopiëren en aanpassen van informatie eindeloos. Men kan tegenwoordig moeiteloos enkele duizenden kopieën van originele data maken, waardoor het onmogelijk wordt om zonder codering de authenticiteit van de data en de niet-weerlegbaarheid van de afzender te verzekeren. Ook wordt het mogelijk om opzettelijke verstoring van de transmissie te voorkomen. Denk hierbij ook aan uiterst geheime militaire en politieke communicatie. In dagelijkse toepassingen, zoals online bankieren, is dit eveneens van heel groot belang.

Het ontwerp van DES, Data Encryption Standard, heeft voor één van de moderne mijlpalen in de geschiedenis van de codering gezorgd. Om de belangrijkheid van de ontdekking aan te tonen, vermelden we dat deze manier van coderen door de U.S. als Federal Information Processing Standard gebruikt werd tot ze gebroken werd.

De opvolger AES, Advanced Encryption Standard, houdt dit tot op de dag van vandaag nog altijd vol.[1]
De ontwikkeling van datacodering en decodering heeft natuurlijk ook in de beveiliging van financiële instituten over de hele wereld een zeer belangrijke rol gespeeld.

DES heeft de zoektocht naar veilige coderingsalgoritmen in een stroomversnelling gebracht. Gezien een bewijsbaar veilig algoritme niet voorhanden is, dient beroep gedaan te worden

[1] WIKIPEDIA (a), *Advanced Encryption Standard*, 2005, online, http://nl.wikipedia.org/wiki/Advanced_Encryption_Standard, 21 september 2005.

op cryptanalyse. Hierbij kruipt men in de huid van de aanvaller en probeert men zwakheden te vinden in de algoritmen. De opgedane kennis wordt dan gebruikt om het systeem beter te beveiligen. De exponentiële groei van de rekenkracht van de computers maakt het ontwerp van veilige en efficiëntie algoritmen een uitdagende taak.

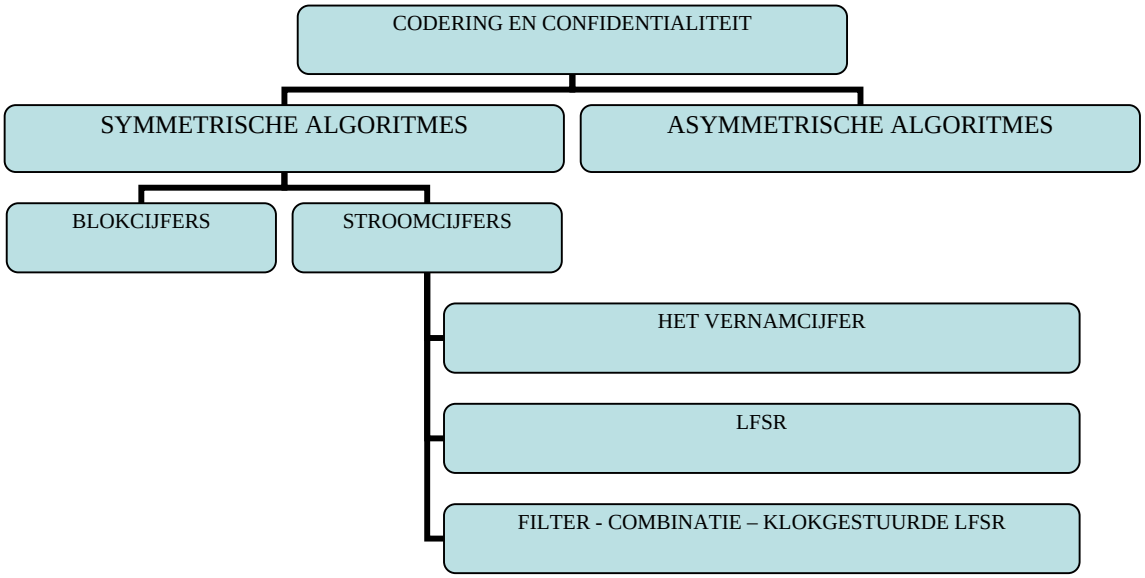
We kunnen besluiten dat men verschillende standaarden en infrastructuren constant verder moet ontwikkelen met het oog op de hoge veiligheidsnoden van deze informatiemaatschappij.

In het eerste hoofdstuk gaan we dieper in op de theoretische achtergrond van codering en decodering van data. Eerst bekijken we de twee hoofdgroepen van codering, met name de symmetrische en de asymmetrische algoritmen. Dan zullen we algemeen aantonen wat de onderlinge verschillen zijn en wanneer deze twee soorten gebruikt worden. In het tweede en derde hoofdstuk bespreken we respectievelijk onze opdrachtgever met de kadering van de opdracht en het gebruikte implementatieplatform. Vanaf het vierde hoofdstuk maken we de link met de stroomcijfers, het onderwerp van deze meesterproef, en bespreken we de resultaten.

HOOFDSTUK 1

Theoretische achtergrond

Om een goede start te nemen, geven we eerst een overzicht van de boomstructuur die we gaan volgen doorheen dit hoofdstuk. Op deze manier willen we een duidelijk beeld geven waar stroomcijfers zich situeren in de wereld van cryptografie.



Figuur 1.1: Schematisch overzicht van de coderingen

1.1 Basisgedachte achter cryptografie

Cryptografie heeft haar afkomst van twee Griekse woorden: "*kryptos*" en "*graphei*". Deze woorden betekenen respectievelijk '*verborgen*/*geheim*' en '*schrijven*', of tesamen '*geheimschrijven*'.
Dit heeft te maken met het ontwerpen van algoritmes en hardwarestructuren die een zo veilig mogelijke codering voorzien. Het omgekeerde van cryptografie noemt cryptanalyse. Men probeert de omgekeerde weg af te leggen: algoritmes breken om achter de geheime boodschap te komen.[2]

Het doel van cryptografie is een digitaal equivalent te scheppen voor een aantal begrippen uit het dagelijkse leven. De belangrijkste eigenschappen zijn geheimhouding, authenticiteit van data en van afzender en onweerlegbaarheid:

- *geheimhouding* impliceert het beveiligen tegen ongewenste lezers. Alle gecodeerde data kunnen onderschept worden, maar blijven onleesbaar als de juiste sleutel niet in

[2] WIKIPEDIA (c), *Cryptografie,* 2005, online, http://nl.wikipedia.org/wiki/Cryptografie, 21 september 2005

het bezit is van de onderschepper. Het is alsof je een auto steelt maar de contactsleutel niet in je bezit hebt.

- *authenticiteit* geeft de "echtheid" van de informatie weer waarin iedere tussenkomst van derden tegengewerkt wordt.

  Dit omvat twee aspecten:
  - authenticiteit van de *data*: dit betekent dat de data inderdaad degene zijn die verstuurd werden en dat ze niet gewijzigd werden door de aanvaller.
  - authenticiteit van de *verzender*: dit betekent dat de data wel degelijk verzonden werden door de persoon die uitgegeven wordt als verzender van de data.

  De informatie en de bron zijn zonder meer "*echt*" wanneer ze van de verwachte afzender met unieke sleutel komen.

- *onweerlegbaarheid* betekent dat de afzender van een bericht achteraf niet kan ontkennen dat hij dat bericht heeft gestuurd. Het is net zoals iemand die een contract ondertekend heeft later niet kan ontkennen dat hij dat gedaan heeft.

Stroomcijfers hebben als voornaamste doel de efficiënte *geheimhouding* van data. De bekendste situaties zijn beveiliging van zowel professionele als private communicatie.[3]
We spreken hier over communicatie in de breedste zin van het woord. Telefoonverkeer (zowel vaste lijnen als gsm), faxverkeer, bluetoothverbindingen, internetverkeer, enz… vallen allemaal onder deze noemer. Als je er niet voor kan zorgen dat je signalen onopgemerkt verzonden kunnen worden, moet je de informatie onleesbaar maken.

[3] MENEZES, A.J. (a), *Handbook of Applied Cryptography Chapter 1*, 1996, online, www.cacr.math.uwaterloo.ca/hac, 23 september 2005.

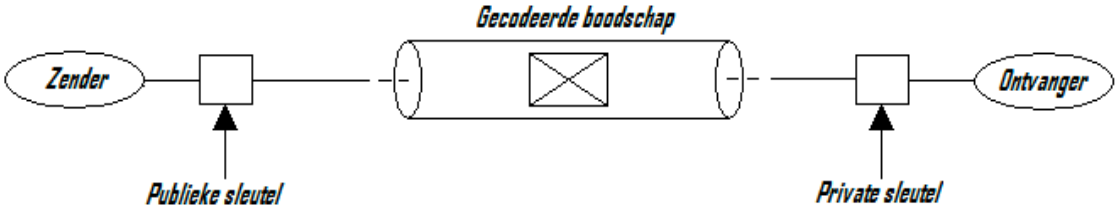
1.2 Sleuteloverdracht

1.2.1 Asymmetrische algoritmes

Het asymmetrische of publieke-sleutelalgoritme[4] *is een algoritme waarbij één sleutel gebruikt wordt voor de codering en een andere sleutel voor de decodering*

De reden waarom dit asymmetrische cryptografie genoemd wordt, zit in het feit dat de zender en de ontvanger ieder twee sleutels hebben: één persoonlijke sleutel die strikt geheim is en één publieke sleutel die toegankelijk is voor iedereen. Dit vernuftig systeem werkt als volgt: de publieke en persoonlijke sleutel zijn verschillend waardoor de ene niet uit de andere afgeleid kan worden. Ze zijn wel wiskundig aan elkaar gekoppeld waardoor berichten die met de ene sleutel vercijferd zijn, met de andere ontcijferd kunnen worden en omgekeerd.
Het belangrijkste voordeel van deze techniek is dat de zender niet kan ontkennen dat hij de data verzond met zijn unieke sleutel.

Asymmetrische cryptografie heeft ook een groot nadeel: het is een relatief zwaar proces en vraagt veel rekenkracht in tegenstelling tot de symmetrische cryptografie. Daarom wordt in veel applicaties een combinatie van symmetrische en asymmetrische cryptografie gebruikt.



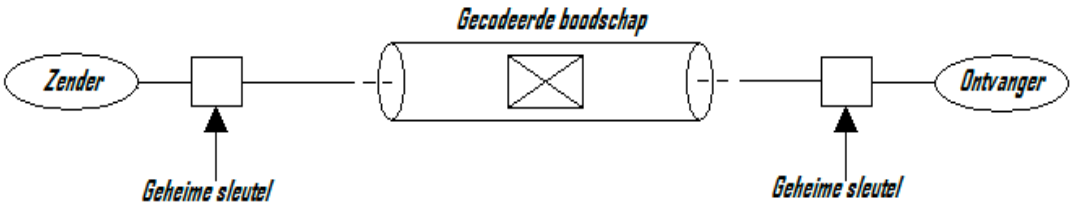
Figuur 1.2: Asymmetrisch algoritme

1.2.2 Symmetrische algoritmes

Het symmetrische of geheime-sleutelalgoritme[5] *is een algoritme waarbij dezelfde sleutel gebruikt wordt voor codering en decodering.*

De zender en ontvanger gebruiken dezelfde geheime sleutel. De zender codeert het document dat hij wil beveiligen met een geheime sleutel en de ontvanger decodeert het document met diezelfde sleutel weer naar leesbare tekst.
Er duikt natuurlijk een probleem op wanneer er een gecodeerd gesprek gevoerd wordt op verre afstand. De sleuteluitwisseling, via het internet of via fysische weg, creëert het gevaar dat de geheime sleutel voor de bedoelde ontvanger bemachtigd wordt door derden.
Een oplossing hiervoor is de geheime sleutel door te sturen via het asymmetrische sleutelalgoritme.

[4] WIKIPEDIA (b), *Asymmetrische Cryptografie*, 2005, online, http://nl.wikipedia.org/wiki/Symmetrische_cryptografie, 22 september 2005.
[5] WIKIPEDIA (f), *Symmetrische Cryptografie*, 2005, online, http://nl.wikipedia.org/wiki/Symmetrische_cryptografie, 21 september 2005.



Figuur 1.3: Symmetrisch algoritme

De symmetrische coderingsmethode wordt opgedeeld in twee subklassen: de blokcijfers en de stroomcijfers.

1.2.2.1 Blokcijfers

Doorheen de geschiedenis van de codering zijn blokcijfers meer verspreid en ontwikkeld ten opzichte van stroomcijfers. De industrie stuurde hierop aan omdat dit de meest bestudeerde, gekende en veiligste coderingstechniek was. In een later stadium startte men projecten op om deze zware en lompe manier van coderen te vervangen in de toepassingsgebieden waar het nodig bleek.

Als we de blokcijfers grondiger bestuderen, ziet men dat grote blokken van een *n* aantal bits klare tekst constant een transformatie ondergaan tot cijfertekst. Het verschil ten opzichte van de stroomcijfers zit hierin dat we niet bit per bit werken maar dat datablokken constant dezelfde transformatie ondergaan. De realisatie in hardware is complexer en de coderingssnelheid is trager dan bij stroomcijfers. In toepassingen waar snelheid, compactheid en laag vermogenverbruik belangrijk zijn, hebben stroomcijfers een groot voordeel te bieden. Een groot deel van de foutenvertraging valt weg omdat de bits individueel behandeld worden.

1.2.2.2 Stroomcijfers

Nu volgt een korte uitleg over hoe de codering met stroomcijfers te werk gaat om een leesbare boodschap, klare tekst, in een beveiligde onleesbare boodschap, cijfertekst, te transformeren. Elk karakter van de klare tekst wordt bit per bit omgezet, afhankelijk van het tijdstip waarop dit gebeurt. Daarom worden stroomcijfers ook wel eens toestandscijfers genoemd: de codering is afhankelijk van de huidige toestand, wat in dit hoofdstuk duidelijk gemaakt wordt. Om dit te verklaren moet men weten dat stroomcijfers steunen op de methode van het "*one–time-pad*"', wat de eerste keer toegepast werd in het *Vernamcijfer*.

1.2.2.2.1 Vernamcijfer

Bij het *Vernamcijfer* wordt een sleutelstroom van willekeurige bits gegenereerd die nooit opnieuw gebruikt worden. Van daar de term "*one–time–pad*". Deze sleutelstroom *k(i)* wordt vervolgens één voor één gecombineerd via een operatie met een XOR-poort met de bits van de klare tekst *m(i)*. Zo bekom je de veilige cijfertekst *c(i)*.

$$c_i = m_i \oplus k_i$$
$$(1 \leq i \leq t)$$

Het *"one-time-pad"* is theoretisch onbreekbaar wanneer je met zekerheid een willekeurige sleutelcode kan maken die maar één keer voorkomt. De lengte van de sleutel is echter gelijk aan de lengte van de klare tekst, waardoor de bruikbaarheid van het systeem erg beperkt is. Het is echter bewezen dat deze codering onbreekbaar wordt als de sleutel dezelfde lengte heeft als de te coderen boodschap. De bruikbaarheid van het systeem wordt hiermee wel gereduceerd tot zuiver theoretische proeven. Om deze praktische reden maken stroomcijfers tegenwoordig gebruik van veel kleinere en gemakkelijker te hanteren sleutels. Op basis van deze sleutels worden er willekeurig sleutelstromen gemaakt die vervolgens gecombineerd worden met de originele boodschap om zo de gecodeerde boodschap te verkrijgen. We zien nu een gelijkaardige techniek tegenover die van het *"one-time-pad"* terwijl ze sneller en minder complex is. Het veiligheidsniveau ligt natuurlijk niet zo hoog, maar dit is een bewuste keuze van de ontwikkelaars van het stroomcijfer. Men moet altijd een keuze maken tussen de efficiëntie in het gebruik, de grootte en de snelheid van de te ontwikkelen hardware. Dit varieert tevens van applicatie tot applicatie.

1.2.2.2.2 LFSR

Bij het genereren van de sleutelstroom gebruikt men vaak LFSR's of Lineair Feedback Shift Registers. Een Lineair Feeback Shift Register of doorschuifregister is een register met lengte *L*. Deze LFSR heeft *L* niveaus of vertragingselementen. Elk niveau is in staat om één bit op te slaan. Ze hebben elk één ingang en één uitgang plus een klokingang die bepaalt wanneer de data van het ene element naar het andere element geschoven wordt.(Cfr. figuur 1.4 (A))

Na elke klokpuls zullen de volgende taken gedaan worden:

- de inhoud van de laatste bit, bit 0, vormt de uitgang
- de inhoud van de bit *i* verhuist naar de bit *i – 1* voor elke *i*,

$$1 = i = L - 1$$

- de nieuwe inhoud van de bit *i -1* wordt de teruggekoppelde bit genoemd en wordt berekend op basis van de andere bits.

Deze bitblokjes van de LFSR zijn heel gemakkelijk implementeerbaar maar ze ondervinden het nadeel dat hun uitgang een lineaire sequentie van bits geeft. Lineariteit is kraakbaar omdat het voorspelbare logica bezit. De sequenties zijn vrij lang en bezitten goede statistische eigenschappen. Door hun structuur kan je onmiddellijk analyseren welke algebraïsche algoritmen gebruikt worden.
Het Berlekamp-Masseyalgoritme is een voorbeeld hoe de repeteerbaarheid uit een gegeven sequentie gehaald kan worden. Dit repeterend deel van de sequentie heet het minimale polynomiale deel en is bepaald door de eerste *2M* elementen van de sequentie.
M is de gegeven hoogste graad in het aantal bits met een M^2 complexiteit aan berekeningen. Hieruit kan de minimale lengte voor de doorschuifregisters bepaald worden.[6]

[6] MENEZES, A.J. (b), *Handbook of Applied Cryptography Chapter 6*, 1996, online, www.cacr.math.uwaterloo.ca/hac, 28 september 2005.

De moeilijkheid zit erin om deze lineariteit te doorbreken. De aanvallen om de originele klare tekst te achterhalen werken op basis van voorspelbaarheid en repeteerbaarheid in datapatronen.

1.2.2.2.3 Filter- , klok- en combinatiegestuurde LFSR's

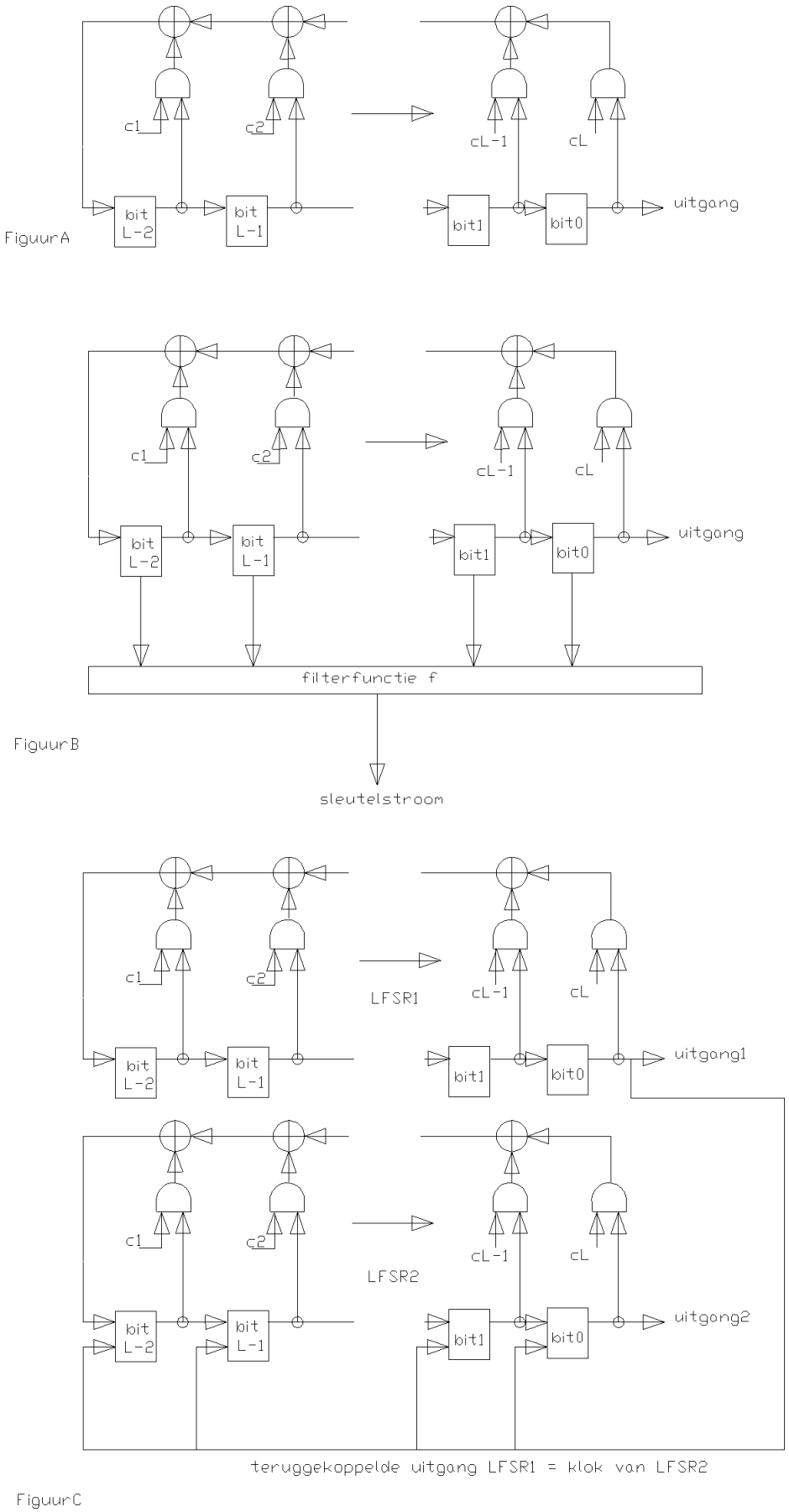
Als eerste manier om de niet-lineariteit te doorbreken kan je een niet-lineaire terugkoppeling gebruiken in plaats van de lineaire terugkoppeling in een LFSR. Dit wordt een Niet-Lineair Feedback Register genoemd, oftwel een NLFSR.

Een tweede algemene techniek voor het vernietigen van de lineariteit is het genereren van een sleutelstroom die de niet-lineaire functie is van bepaalde flipflops van een LFSR. Hier wordt *f* de filterfunctie genoemd.(Cfr. figuur 1.4 (B))

De meeste technieken klokken de LFSR's regelmatig: de beweging van alle data gebeurt door dezelfde klok. De hoofdgedachte bij de klokgestuurde generatoren laat de klok van de LFSR werken op basis van de uitgang van een andere LFSR.(Cfr. figuur 1.4 (C)) Dit is een derde methode om de lineariteit te verbreken door de LFSR op een onregelmatige manier te klokken.

Als laatste manier kan men de lineariteit van een LFSR doorbreken door een combinatie te maken van een niet-lineaire kloksturing en een filter. Dit wordt combinatiegestuurde LFSR genoemd.

De stroomcijfers A5/1, Trivium en DECIMv2, die wij geïmplementeerd hebben voor deze meesterproef, maken gebruik van een klokgestuurde LFSR of een filtergestuurde LFSR, zoals hierboven beschreven staat.



Figuur 1.4: Gewone LFSR (A), filtergestuurde LFSR (B) en klokgestuurde LFSR (C)

HOOFDSTUK 2

COSIC en ECRYPT

2.1 COSIC: Computer Security and Industrial Cryptography

COSIC is een instituut verbonden aan de K.U.Leuven dat onderzoeksactiviteiten doet op alle domeinen van de cryptografie. Dit wil zeggen dat er digitale evenbeelden gezocht worden voor onder andere handtekeningen, identificatiemogelijkheden, notarisatie en betalingen.
Om dit doel te bereiken wordt er onderzoek gedaan naar het ontwerp, de evaluatie en de effectieve implementatie van cryptografische algoritmes, protocollen en de ontwikkeling van veiligheidsarchitecturen.
Het theoretische denkwerk rond cryptografische algoritmes en protocollen steunt hoofdzakelijk op discrete wiskunde. Daarom bestudeert men er hoofdzakelijk de volgende wiskundige takken: getallentheorie, eindige velden, Booleaanse functies, eindige geometrie en coderingstheorieën. Andere wiskundige takken, relevant aan het onderzoek, zijn statistiek en optimalisatie. Dit denkwerk is hoofdzakelijk gebaseerd op software maar moet ook als hardware geïmplementeerd kunnen worden.
Ons promotorbedrijf werkt samen met belangrijke bedrijven en onderzoekscentra voor computersystemen, telecommunicatienetwerken en betalingssystemen zoals Banksys en Alcatel. Dankzij deze connecties heeft COSIC een internationale uitstraling. Er werken 29 onderzoeksmedewerkers en 8 post-doctoraten bij COSIC, samen met nog enkele vrije onderzoekers en medeonderzoekers van andere centra. Aan het hoofd van de organisatie staan Prof. Bart Preneel, Prof. Joos Vandewalle en Prof. Ingrid Verbauwhede.[7]

Op haar beurt is COSIC verbonden aan ECRYPT, European Network of Excellence for Cryptology. Dit is een vier jaar durend project dat opgericht is in het Information Societies Programme van het Sixth Framework Programma (FP6) van de Europese Commissie.

Het ECRYPT netwerk heeft als doel om de beste onderzoeksgroepen in cryptologie te stimuleren om samen te werken. Hiermee viseert men zowel het academische als het industriële niveau dat door dit netwerk voor dringende problemen sneller en efficiënter een oplossing zal vinden.
ECRYPT organiseert onderling onderzoeksactiviteiten zoals gezamenlijke workshops, het uitwisselen van onderzoekers, de ontwikkeling van gemeenschappelijke middelen enz. Om dit doel te bereiken integreren ze hun onderzoeksmogelijkheden in vijf virtuele labs waar we hier niet verder op ingaan.[8]

[7] KALLIST, J., *Computer Security and Industrial Cryptography*, 2005, online, http://www.esat.kuleuven.be/cosic, 1 december 2005.
[8] LANO, J., *eStream*, 2005,online, http://www.ecrypt.eu.org/stream, 1 december 2005.

2.2 eSTREAM: het ECRYPT-stroomcijferproject

De standaard blokcijfers die voor codering gebruikt worden zijn de Data Encryption Standard en de Advanced Encryption Standard. Van deze twee is AES tot op de dag van vandaag ongebroken.[9]

Sinds kort is er bij verschillende wetenschappers het idee gegroeid dat stroomcijfers dezelfde veiligheid aan een lagere kostprijs kunnen bieden. Daarom heeft men het eSTREAM-project opgestart dat na enkele jaren van research en evaluatie naar één of meerdere algemene standaard stroomcijfers moet leiden. Het is een virtuele bron van informatie en resultaten van nieuwe stroomcijfervoorstellen voor cryptografische ontwikkelaars, ontwerpers en onderzoekers.

Verschillende onderzoeksinstellingen worden gestimuleerd om stroomcijfers te ontwerpen die veilig en efficiënt zijn. De stroomcijfers, die in het eSTREAM project worden geanalyseerd, moeten tot één van de volgende profielen behoren:

- profiel 1: stroomcijfers voor software applicaties met grote doorstroming van bits per klokpuls.
- profiel 2: stroomcijfers voor hardware applicaties met beperkingen zoals opslagruimte en energieverbruik. Hardwarematig stelt dit het aantal benodigde transistoren in een ASIC voor of het aantal slices op het FPGA-testplatform. In het kader van deze meesterproef kijken we voornamelijk naar de relatie tussen de efficiëntie en de benodigde hardware.

Deze oproep naar nieuwe stroomcijfers resulteerde in 35 kandidaten. Deze stroomcijfers zijn afkomstig van verschillende internationale onderzoeksinstellingen die meegedaan hebben aan het eSTREAM–project, georganiseerd door COSIC.[10] In de eerste fase van de evaluatie kon er vrij feedback gegeven worden over de veiligheid en performantie van de stroomcijfervoorstellen. In latere fasen van het project bleven de meest performante stroomcijfers over. Van de 35 stroomcijfers blijven er momenteel 10 cijfers ongebroken. Deze zijn de enige die de moeite waard zijn om in hardware te implementeren. In samenwerking met andere ECRYPT-partners zoals de RUB (Ruhr-Universitär Bochum) gaat COSIC deze 10 resterende cijfers implementeren.[11]
In het kader van deze meesterproef hebben wij *Trivium*, *Rabbit*, *DECIMv2* en *Salsa20* voor onze rekening genomen.

[9] WIKIPEDIA (d), *Data Encryption Standard*, 2005, online, http://nl.wikipedia.org/wiki/Data_Encryption_Standard, 21 september 2005.
[10] PRENEEL, B., *Network of Excellence in Cryptography*, 2005, online, http://www.ecrypt.eu.org/, 1 december 2005.
[11] RUHR-UNIVERSITÄT BOCHUM, *Ruhr-Universität Bochum*, 2005, online, http://www.ruhr-uni-bochum.de/index_en.htm, 21 september 2005.

HOOFDSTUK 3

Werkplatform: Xilinx ISE en Modelsim XE

3.1 Programmeerplatform

In dit hoofdstuk bespreken we kort de eigenschappen van het gekozen werkplatform. Het werkplatform moet de mogelijkheid bieden om de hardware te programmeren, te testen op fouten en eigenschappen, én de werking te simuleren.
Het gekozen softwarepakket wordt voorzien door Xilinx, met name het ISE-package 7.1i, en Modelsim XE 6.0.[12] Deze software werd ons aangeboden door onze promotor aan de KHLim.

Een goede functionaliteit en een hoge ontwerpsnelheid zijn de belangrijkste eigenschappen van het ISE-package. Het bezit alle mogelijkheden om een ontwerp op een snelle en goede manier met alle mogelijke bewerkingen te realiseren. De aanvullende simulatiemodule, Modelsim XE 6.0, linkt men door middel van de juiste instellingen aan het ISE-package. Met deze combinatie kan men gebruik maken van de ingebouwde bewerkingen: input- en outputtoewijzingen, tijdsgebonden ontwerpgrenzen en HDL-simulaties. Op vlak van controle en optimalisatie wordt de ontwerptijd enorm verkort vergeleken met het effectief maken en testen van de hardware.
Het ontwerpproces gaat als volgt: de code van de hardware van het stroomcijfer wordt in Xilinx geschreven en vervolgens in een testomgeving gedefiniëerd. Met deze “testbank” worden aan de ingangen van het ontwerp waarden toegekend op bepaalde tijdstippen. Vanuit Xilinx wordt Modelsim gestart: de aangelegde signalen, de interne signalen en de uitgaande signalen kan men op elk moment van de simulatie meten. Op deze manier voert men de HDL (Hardware Description Language)–simulatie uit.

3.2 Programmeertaal

Als programmeertaal hebben we VHDL (Very high speed integrated circuit Hardware Description Language) gebruikt. Met deze programmeertaal hebben we de mogelijkheid om snel complexe hardwarestructuren te ontwerpen op een hoog niveau. Elk VHDL-bestand bestaat uit entiteiten en architecturen. Entiteiten geven aan wat de ingangen en uitgangen van het ontwerp zijn. In de architecturen worden de interne signalen en processen beschreven. Zo heeft men snel een overzicht op de werking van een ontwerp en het opduiken van eventuele programmeerfouten. Toevoeging van commentaar, herbruikbaarheid van de code en het gebruik van standaardontwerpen uit de bijgevoegde bibliotheken zijn de voordelen van het programmeren op hoog niveau.

[12] XILINX, *Xilinx*, 2005, online, http://www.xilinx.com, 20 september 2005.

3.3 Hardwaresimulatie

We hebben de geprogrammeerde code op twee hardwarematige manieren benaderd: de specifieke hardware implementatie door middel van ASIC en de algemene hardware implementatie op FPGA. Met ASIC wordt het ontwerp aangepaster en efficiënter gerealiseerd. Op FPGA wordt de ontwerptijd korter, maar kan het ontwerp groter en trager worden.

We gebruiken de Virtex2Pro FPGA, een "*Field Programmable Gate Array"* (xc2vp70-6ff1704-6). Dit is een hardwareblok van open verbindingen die naargelang de geprogrammeerde code tijdelijk gesloten en weer geopend kunnen worden. Daarnaast zijn op dit model FPGA grote vermenigvuldigers, snelle "*look ahead*"-optellers, een powerPC microcontroller, … voorzien om een goede balans te maken tussen snelle hardware en een korte ontwerptijd.

Daarnaast hebben we onze code ook getest op een andere soort hardwareblok, de ASIC. Dit staat voor "*Application-Specific Integrated Circuit*". Dit is een geïntegreerd circuit dat voor specifiek gebruik ontworpen wordt. Een microprocessor ontwerpt men beter op FPGA terwijl men een mp3-spelerchip het beste in een ASIC-design beschrijft. Door de technologische vooruitgang, schaalverkleining en verbeterde instrumenten is de complexiteit van een ASIC-design gegroeid van 5000 gates naar 100 miljoen gates. Een gate stelt de grootte van de inversie van een AND-poort voor. Wanneer men naar lage productiekosten en kleinere volumes van digitale designs kijkt, wordt ASIC een minder aantrekkelijke oplossing ten opzichte van de FPGA, deze laatste groeit in snelheid en capaciteit. Een ASIC-ontwerp bestaat op verschillende niveaus: men kan de standaard bibliotheken gebruiken waarin cellen met een vaste grootte en functionaliteit gedefinieerd zijn. De programmeertaal op hoog niveau wordt omgezet naar hardware via deze voorziene cellen. Wanneer men een efficiënter gebruik van gates wil, gaat men naar een hoger ontwerpniveau, "*Full Custom*": specifieke functies worden vanaf nul ontworpen en eventueel,voor niet kritische delen, gecombineerd met standaardcellen.
De simulatie in ASIC is uitgevoerd met Designvision[13], waarbij we enkel het aantal gates gemeten hebben, zonder een kloksnelheid op te leggen. We hebben de op FPGA geteste VHDL-code rechtstreeks in deze nieuwe simulator geladen.

In de volgende hoofdstukken gaan we dieper in op de verschillende stroomcijfers die we geïmplementeerd hebben. De opgave is het maken van een hardwarematige implementatie van stroomcijfers. De informatie van een stroomcijfer halen we uit de ingezonden tekst, die voorzien is van schema's en formules, samen met de C-code die de werking van het stroomcijfer beschrijft. We kunnen de gewenste bitsequenties in software simuleren om zo onze hardware-equivalenten te vergelijken. Bij de bespreking van de resultaten gaan we vooral kijken naar de maximale snelheid per klokpuls en de ingenomen oppervlakte van de hardware op de FPGA en de ASIC. De resultaten van deze stroomcijfers worden vervolgens in het kader van het Ecrypt-project per onderzoeksgroep onderling vergeleken om het beste stroomcijfer te vinden.

[13] SYNOPSYS, *DesigVision*, 2005), online, http://www.synopsys.com, 1 mei 2006.

HOOFDSTUK 4

Stroomcijfer A5/1

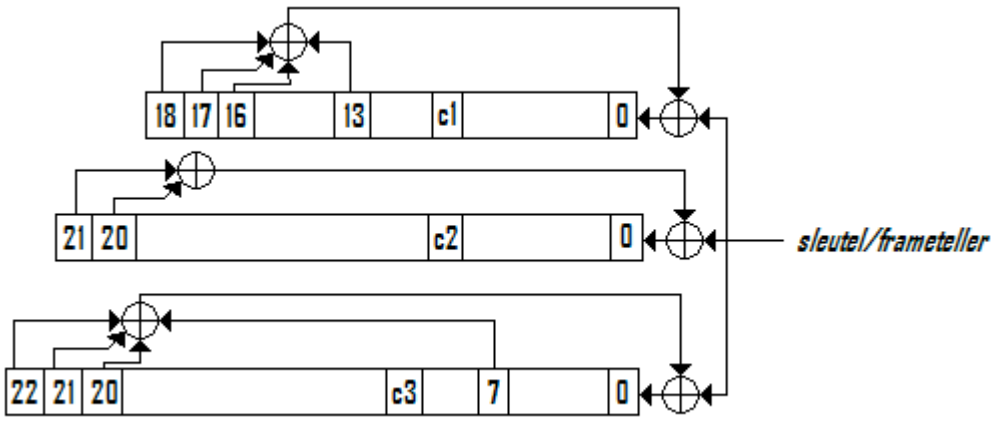
4.1 Theorie achter A5/1

Het stroomcijfer A5/1 is geen kandidaat van het ECRYPT-project. We hebben dit stroomcijfer als eerste geïmplementeerd om kennis te maken met coderingtechnieken in de praktijk. We hebben nu een goede referentie om stroomcijfers met elkaar te vergelijken. Het A5/1-stroomcijfer bestaat al enkele jaren en wordt gebruikt om gsm–gesprekken te beveiligen. Oorspronkelijk was dit stroomcijfer geheim voor het grote publiek, maar in 1994 lekte de specificaties van A5/1 uit. Sindsdien werden belangrijke zwakheden gevonden in het algoritme, waardoor het onveilig is.

4.1.1 Werking van A5/1

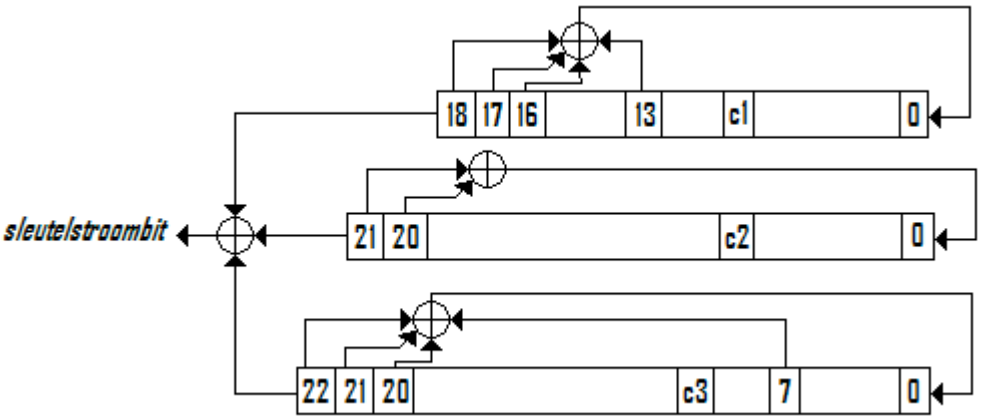
Gesprekken over een gsm worden in een sequentie van frames of "blokken" verzonden. Dit frame kan je bekijken als een "tijdsvenster" waarin er geluiden in bits omgezet worden.
Om de 4.6 milliseconden wordt er één enkel frame met een lengte van 228 bits gemaakt.
De helft van deze bits wordt dan gebruikt voor het ontvangen van het gesprek, de andere helft dient om te zenden.
Om dit geluidsframe te maken wordt er gebruik gemaakt van A5/1. Het cijfer zal steeds een sleutelstroom van 228 bits produceren. Vervolgens worden 114 bits van de klare tekst van zender A naar cijfertekst omgezet door een bewerking met een XOR-poort. De overige 114 bits van de sleutelstroom worden gebruikt om de cijfertekst van de ontvanger B om te zetten naar klare tekst. Tijdens een telefoongesprek is iedereen tegelijkertijd zender én ontvanger. Dit proces gebeurt aan beide kanten van het gsm-gesprek.

Het maken van een frame waarin een boodschap gecodeerd zit, gebeurt als volgt:
In de eerste stap van het proces wordt een geheime sleutel van 64 bits in de LFSR's geladen. Dit gebeurt door de bits van de sleutel één voor één parallel via een XOR-poort in de drie LFSR's te steken. (Cfr. figuur 4.1) Dit gebeurt 64 klokperiodes lang, zonder dat de meerderheidsregel toegepast wordt. De "*stop-and-go*"-functie van de kloksturing is dan ook uitgeschakeld.
Hierna worden de 22 bits van een frameteller op identieke wijze in de LFSR's geklokt. Dit duurt 22 klokcycli lang en gebeurt ook zonder de meerderheidsregel en de "*stop-and-go*"-functie.
Vervolgens lopen de drie LFSR's honderd klokperiodes door waarbij de "*stop-and-go*"-functie toegepast wordt zonder dat er sleutelstroombits gemaakt worden. De sleutel en frameteller zijn nu met elkaar vermengd. In dit stadium van het ontwerp hopen de ontwerpers dat hun stroomcijfer voldoende niet-lineariteit aan boord heeft opdat de initiële waarden niet ontdekt kunnen worden.



Figuur 4.1: Initialisatiefase

Op dit punt genereren de drie registers een sleutelstroom door een bewerking met een XOR-poort van de drie eindbits van de registers. Dit duurt 228 klokperiodes opdat één frame van de sleutelstroom klaar is.
Tot slot wordt de sleutelstroom samengevoegd met de klare tekst door een XOR-poort. Nu is één frame cijfertekst gegenereerd.



Figuur 4.2: Algemeen schema van A5/1

De eerste 114 bits van de cijfertekst van de eerste gsm worden naar de tweede gestuurd. De overige bits zullen dan de 114 gecodeerde bits, die afkomstig zijn van de tweede gsm, omzetten naar verstaanbaar geluid. Zo worden er om de 4.6 milliseconden tegelijkertijd frames doorgestuurd en is er communicatie mogelijk.

4.1.1.1 Meerderheidsregel van A5/1

A5/1 bestaat uit 3 verschillende LFSR's die onregelmatig geklokt worden. Ze zijn elk verschillend in lengte en worden allemaal op '0' gezet bij aanvang van een nieuw gesprek. Per LFSR zijn er een aantal taps die steeds dezelfde bits met een XOR-poort bewerken. De LFSR's hebben een *"stop-and-go"*-functie waarbij de meerderheidsregel of *"Majority Rule"* toegepast wordt.

Deze functie werkt als volgt :
Elke LFSR heeft één register dat de klokbit *c* (Cfr. figuur 4.2) bijhoudt. Deze drie bits worden constant met elkaar vergeleken. Op basis van de klokbits worden twee tot drie LFSR's geklokt waardoor alle bits van de LFSR's één plaats opschuiven. Stel dat er twee klokbits '1' en een derde klokbit '0' zijn, dan is '1' in de meerderheid en zal elke LFSR met een klokbit gelijk aan '1' geklokt worden. Zo zullen er steeds twee tot drie registers geklokt worden.

Zoals we in de theorie reeds aangehaald hebben, kunnen we de lineariteit van het stroomcijfer op verschillende manieren breken. In het geval van de A5/1 gebeurt dit dus met ongelijk geklokte registers.[14]

LFSR nummer	lengte (bits)	Karakteristieke polynomiaal	Klokbits (c)	tapbits
1	19	$x^{19} + x^5 + x^2 + x + 1$	8	13, 16, 17, 18
2	22	$x^{22} + x + 1$	10	20, 21
3	23	$x^{23} + x^{15} + x^2 + x + 1$	10	7, 20, 21, 22

Tabel 4.1: Indeling van de klokbits en de tapbits

4.1.2 Gsm als toepassing

Dit stroomcijfer vindt zijn toepassing vooral in de wereld van de communicatie. Het zit in elke gsm ingebouwd. Het vercijfert elke uitgaande boodschap en het ontcijfert alle inkomende informatie. Hoe gaat het nu precies in zijn werk?

Wanneer persoon A een persoon B wil contacteren met zijn gsm zal het stroomcijfer frames van de uitgaande boodschap coderen en naar een centrale sturen. Dit gebeurt met een unieke sleutel van persoon A. Zodra het eerste frame verstuurd is, komt dit aan in een centrale. De centrale heeft de sleutel van elk gsm-toestel en wanneer een frame van persoon A binnenkomt, zal de centrale deze decoderen met de geheime sleutel van persoon A. Vervolgens wordt dit frame weer gecodeerd maar nu met de geheime sleutel van persoon B. Als laatste stuurt de centrale het frame verder naar persoon B die het ontvangen frame kan decoderen. Zo ontstaat er een verstaanbaar gesprek tussen persoon A en B.

Ook de geheime sleutel van 64 bits is niet zonder nadenken gekozen. Deze sleutel kan men door middel van een snelle computer gemakkelijk achterhalen om zo gesprekken af te luisteren. Dit is schending van de privacy, maar mag indien nodig toegepast worden door politie voor onderzoeksdoeleinden.

[14] WIKIPEDIA (e), *A5/1*, 2005, online, http://en.wikipedia.org/wiki/A5/1, 12 september 2005.
LANO, J., Biryukov, A., Shamir, A., Wagner, D. J., *List of Stream Ciphers, Real-Time Cryptanalysis of A5/1 on a PC*, 2005, online, http://homes.esat.kuleuven.be/~jlano, 12 september 2005.

4.2 Implementatie van A5/1

4.2.1 Aanpak

Allereerst leggen we de in- en uitgangen vast. De ingangen van de entiteit bestaan uit de klare tekst van 228 bits, een sleutel van 64 bits en een frameteller van 22 bits. De overige gebruikte ingangen zijn een klok, een *reset* en een startsignaal dat dient om het stroomcijfer te starten. De uitgangen van de entiteit zijn de cijfertekst van 228 bits en een "klaar"-signaal dat aangeeft dat de codering klaar is.

Het schema is gebaseerd op een FSM samen met drie klokprocessen en een combinatorisch proces.
Een eerste klokproces zorgt ervoor dat de bits in de 3 LFSR's steeds opschuiven nadat het proces een toestandssignaal ontvangen heeft. Het combinatorisch proces stuurt de "aan"-signalen (*clk_en1, clk_en2, clk_en3*) voor de drie LFSR's aan op basis van de klokbits (*cb1, cb2, cb3*). Op deze manier is de "*stop-and-go*"-functie geïmplementeerd op basis van de meerderheidsregel. De "aan"-signalen hebben we vervolgens gebruikt als voorwaarden in het hierboven beschreven klokproces, waardoor de juiste LFSR's geklokt worden en een intern register (*keyr*) steeds ingeladen en doorgeschoven wordt.

Verder zijn er ook twee andere klokprocessen geïmplementeerd die ervoor moeten zorgen dat de interne schuifregisters (*keyreg en framereg*) de minst beduidende bit via een XOR-poort in de LFSR's laden. Dit gebeurt steeds nadat ze een toestandssignaal afkomstig van de FSM ontvangen hebben. Vlak hierna laten we alle bits van de schuifregisters één positie opschuiven en laden we via de ingang een nieuwe '0' in. Het inladen gebeurt dus op seriële wijze. Op deze manier worden de LFSR's voorzien van de sleutel en frameteller.

4.2.2 FSM

Als controleblok is er een FSM die verscheidene toestanden (*idle*, *count*, *comp*, *keylader*, *framelader*, *maj_rule* en *set*) doorloopt op basis van een interne teller (*counter*). Na een startpuls kom je in de *idle*-toestand terecht. Deze toestand kan niet meer bereikt worden, tenzij na een fout of na de toestand *set*.

In de toestand *count* laten we de interne teller met 1 verhogen en in *comp* is de teller de voorwaarde om eventueel naar de volgende toestand over te gaan. De teller houdt het aantal klokcycli bij waardoor elke toestand op het juiste moment aangeroepen wordt.
Zolang de teller onder de 64 blijft, zal *keylader* steeds de volgende toestand zijn na *comp*. Deze stuurt vervolgens een *keystatus* uit waardoor een klokproces het register *keyreg* serieel ledigt in de LFSR's. Wanneer de teller zich tussen 64 en 86 bevindt, is de toestand van de FSM *framelader* dat een *framestatus*-signaal zal uitsturen. Nadat *counter* 86 gepasseerd is en de FSM zich in *maj_rule* of *set* bevindt, stuurt de FSM een *regstatus* uit. Al deze processen worden beschreven in figuur 4.3.

Zodra de sleutel en de frameteller volledig in de LFSR's (Cfr. figuur 4.4 en 4.5) geladen zijn, zullen de LFSR's zich nog 100 klokcycli lang mengen op basis van de "*stop-and-go*"-functie. De teller houdt het aantal klokcycli bij. Hierna wordt de sleutelstroom gegenereerd.
In onze implementatie laten we een register van 228 bits vol lopen zodat er één volledig frame gebruikt kan worden voor het coderen en het decoderen. Zodra dit register vol is, zal

de FSM naar de toestand *set* springen en zal het signaal *ready* hoog uitsturen. Dit betekent dat er een frame klaar is.

Zo kunnen we constant frames vormen door steeds een startsignaal te geven op het moment dat het "klaar"-signaal uitgestuurd wordt.

4.2.3 Optimalisatie

Na enkele simulaties blijkt dat het steeds drie klokpulsen duurt vooraleer er een bit cijfertekst naar buiten komt. Daarom hebben we de FSM–toestanden vervangen door een klokproces dat alle functionaliteiten van de FSM overneemt. De andere klokprocessen moeten hierdoor niet veranderd of aangepast worden. Als uiteindelijk resultaat krijgen we één bit cijfertekst per klokpuls naar buiten. De tweede versie noemen we A5/1v2.

4.2.4 Testresultaten van A5/1

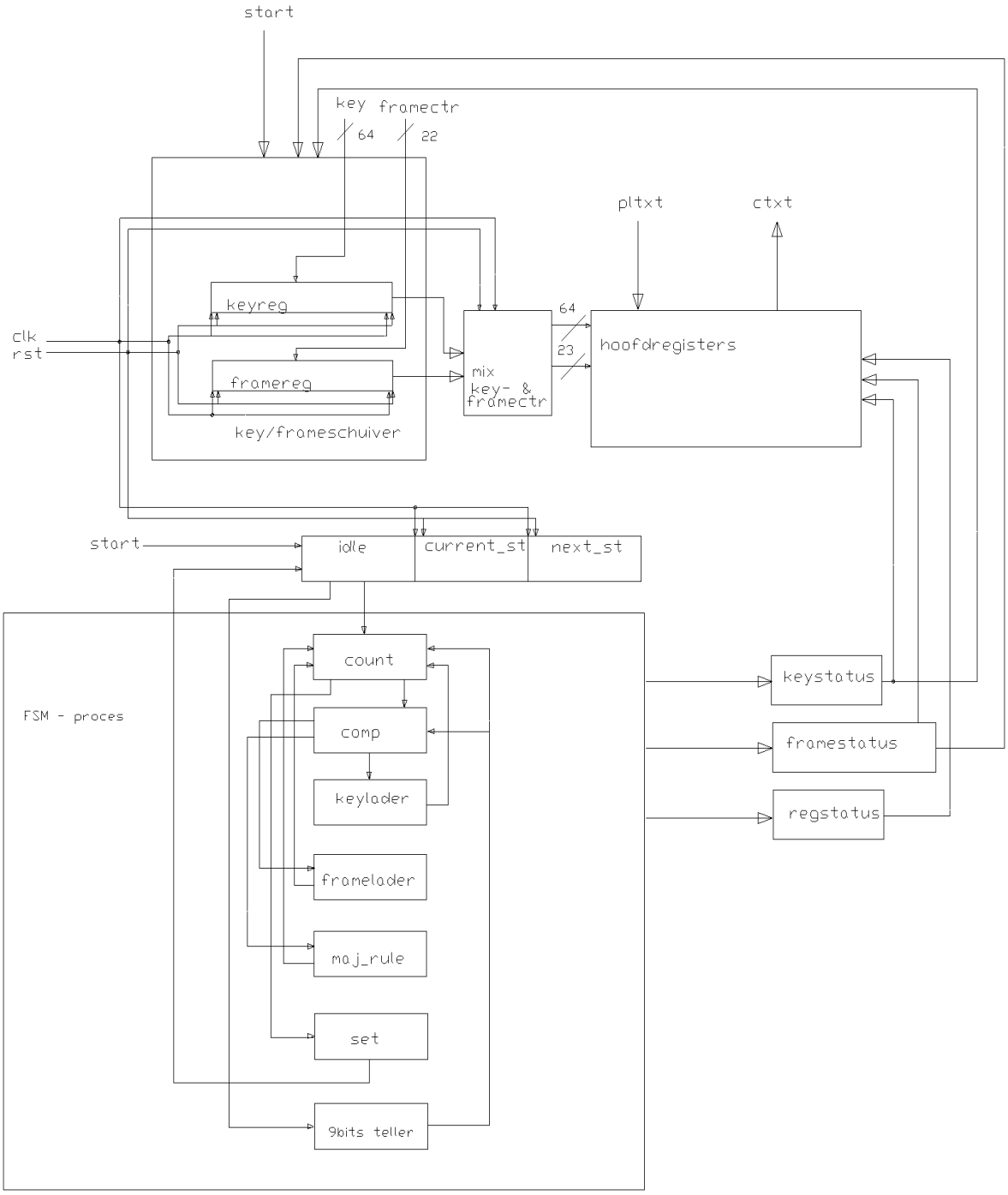
In wat volgt geven we de resultaten van onze twee implementaties van het stroomcijfer A5/1.
Het "*place and route*"–rapport op FPGA geeft de volgende resultaten:

	A5/1v1	A5/1v2
# gebruikte slices in FPGA (totaal 33088)	465	461
minimale klokperiode (op 50% duty cycle) (ns)	3,384	3,246
aantal klokpulsen nodig om 1 bit uit te sturen	3	1

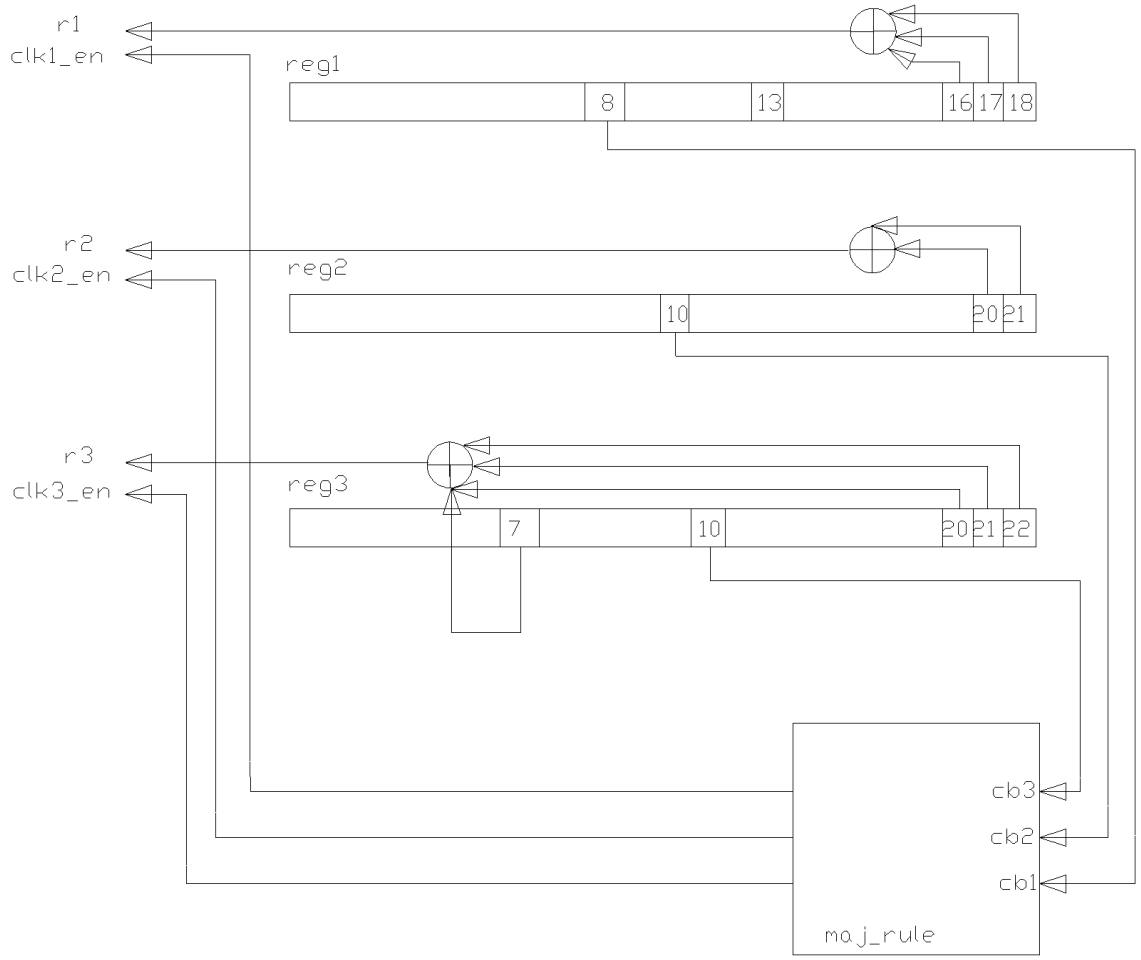
Tabel 4.2: Testresultaten A5/1

De resultaten van A5/1 geven voor A5/1v1 een verschillend resultaat qua vertraging dan A5/1v2. A5/1v2 is een stuk sneller: het kritische pad loopt bij A5/1v1 van *keyr219* naar *keyr220*. Het vullen van het sleutelstroomregister verloopt het traagst in de simulatie omdat de FSM teveel tijd in beslag neemt tijdens het vernieuwen van de variabelen en het opschuiven van toestand naar toestand. Deze tijd is vooral te wijten aan de manier waarop de baantjes op de FPGA gelegd worden: de *routing* bedraagt 79.7% tegenover 20,3 % voor de logica. Wanneer de FSM weggelaten wordt in A5/1v2, gaat het kritische pad van *counter0* naar *counter9*. Hier moet de huidige situatie altijd vergeleken worden met de teller, *counter*. Het vergelijkingsblok heeft tien bits nodig waardoor de logica voor een kleinere maar belangrijke vertraging (78.4%) zorgt.

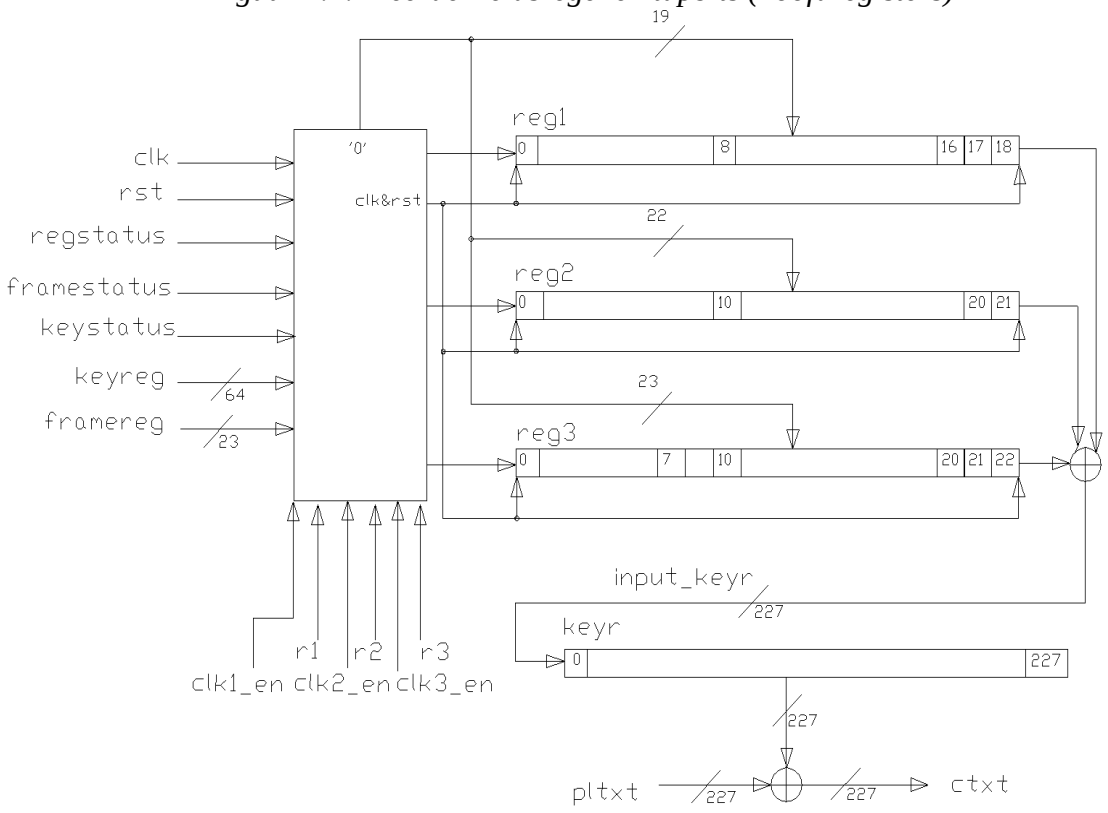
4.3 Schematische voorstelling van A5/1



Figuur 4.3: FSM



Figuur 4.4: Meerderheidsregel en tapbits (hoofdregisters)



Figuur 4.5: Interne registerwerking (hoofdregisters)

HOOFDSTUK 5

Stroomcijfer Trivium

5.1 Theorie achter Trivium

Trivium is het eerste cijfer van de eSTREAM-inzendingen dat wij geïmplementeerd hebben. De ontwerpers van Trivium hebben zich gebaseerd op het volgende: als de basis van een stroomcijfer grondig getest is, geven eenvoudige stroomcijfers na een lange overlevingstijd meer vertrouwen dan complexere ontwerpen. De werking is beter controleerbaar en aanpassingen zijn gemakkelijker aan te brengen.
Trivium is ontworpen in het kader van een onderzoek dat nagaat in hoeverre een stroomcijfer qua hardware geminimaliseerd kan worden. De ontwerpers willen niets inboeten aan veiligheid en snelheid.

5.1.1 Werking van Trivium

Trivium is een stroomcijfer dat tot 2^{64}-bits sleutelstroom kan genereren, vertrekkend van een 80-bit geheime sleutel en een 80-bit *IV* (initiële waarde). Zoals bij de meeste stroomcijfers bestaat het coderingsproces uit twee fasen: de initialisatiefase, waarin de sleutel en de initiële waarde ingeladen en gemixt worden én de eigenlijke generatiefase van de sleutelstroombits.
Er wordt één groot register van 288 bits gebruikt.
De sleutel en *IV* worden parallel ingeladen nadat alle bits, behalve de laatste 3 (286, 287 en 288), op '0' gezet werden. Dit gebeurt als volgt: [15]

$$(s_1, s_{2,....}, s_{93}) \leftarrow (K_1,....., K_{80},0,....0)$$
$$(s_{94}, s_{95,....}, s_{177}) \leftarrow (IV_1,....., IV_{80},0,....0)$$
$$(s_{178}, s_{279,....}, s_{288}) \leftarrow (0,.....,0,1,1,1)$$

[15] LANO, J., De Canniere, C., Preneel B., *Trivium*, 2005, online, http://www.ecrypt.eu.org/stream/trivium.html, 8 oktober 2005.

for i = 1 to 4 x 288 do

$$t_1 \leftarrow s_{66} \oplus (s_{91} * s_{92}) \oplus s_{93} \oplus s_{171}$$
$$t_2 \leftarrow s_{162} \oplus (s_{175} * s_{176}) + s_{177} \oplus s_{264}$$
$$t_3 \leftarrow s_{243} \oplus (s_{286} * s_{287}) \oplus s_{288} \oplus s_{69}$$
$$(s_1, s_{2,....}, s_{93}) \leftarrow (t_3, s_1, ..., s_{92})$$
$$(s_9, s_{95,....}, s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176})$$
$$(s_{178}, s_{279,....}, s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287})$$

end for

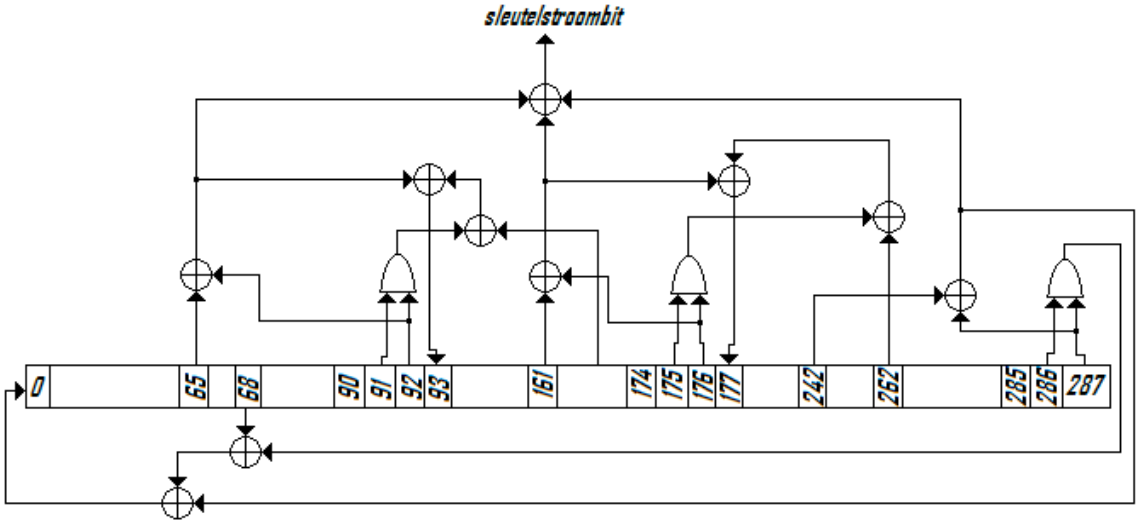
De bits worden nu gedurende 4 cycli van 288 klokpulsen geroteerd zonder een sleutelstroom te genereren. Na deze stap sturen we de sleutelstroombits naar buiten.
Dit register heeft 15 bits om 3 andere bits te vernieuwen die telkens één sleutelstroombit *z(i)* genereren. Dit blijft gebeuren totdat we het gewenste aantal sleutelstroombits naar buiten toe komt. Dit proces gaat als volgt:

for i = 0 to N do

$$t_1 \leftarrow s_{66} \oplus s_{93}$$
$$t_2 \leftarrow s_{162} \oplus s_{177}$$
$$t_3 \leftarrow s_{243} \oplus s_{288}$$
$$t_1 \leftarrow t_1 \oplus t_2 \oplus t_3$$
$$t_1 \leftarrow t_1 \oplus (s_{91} * s_{92}) \oplus s_{171}$$
$$t_2 \leftarrow t_2 \oplus (s_{175} * s_{176}) \oplus s_{264}$$
$$t_3 \leftarrow t_3 \oplus (s_{286} * s_{287}) \oplus s_{69}$$
$$(s_1, s_{2,....}, s_{93}) \leftarrow (t_3, s_1, ..., s_{92})$$
$$(s_9, s_{95,....}, s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176})$$
$$(s_{178}, s_{279,....}, s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287}$$

end for

De figuur hieronder toont het algemene schema van Trivium. In deze tekening wordt er per klokperiode 1 sleutelstroombit gegenereerd.



Figuur 5.1: Algemeen schema van Trivium

5.2 Implementatie van Trivium

5.2.1 Aanpak

We starten weer met het vastleggen van de in- en uitgangen. De ingangen van de entiteit bestaan uit de klare tekst van 228 bits, de sleutel van 80 bits, de initiële waarde van 80 bits en een vrij in te geven aantal sleutelstroombits (maximaal 64). Als in- en uitgangen van één bit groot namen we een klok, een *reset* en een startsignaal. De uitgangen van de entiteit zijn dan de cijfertekst van 228 bits en een signaal *ready* dat aangeeft wanneer de codering afgelopen is. Vervolgens creëren we één groot register van 288 bits.

Onze implementatie bestaat uit 2 klokprocessen waarrond een combinatorisch gedeelte zit.
De combinatoriek bestaat uit alle bewerkingen die de sleutelstroombits vernieuwen. Dit zijn de bewerkingen met XOR-poorten en AND-poorten. De uitkomst van al deze bewerkingen wordt uiteindelijk samengevoegd in een bewerking met een XOR-poort om zo de sleutelstroom *z(i)* te genereren.

5.2.2 Registerwerking

Door middel van een klokproces laden we de sleutel en de initiële waarde in. Dit gebeurt vanaf de eerste bit voor de sleutel en vanaf de 94ste bit voor de initiële waarde. De laatste 3 bits worden op 1 gezet. We gebruiken een teller (*counter*) om de duur van de eerste fase vast te leggen. Deze telt naar beneden vanaf 1152, de exacte lengte van de initialisatiecyclus: 4 maal 288. Zolang deze cyclus niet afgelopen is en het *start_generating*-signaal niet hoog is, worden de bits seriëel doorheen het register geschoven zonder dat het combinatorische proces werkt. Na 1152 klokpulsen wordt het *start_generating*-signaal hoog. Op dat moment worden de sleutelstroombits door het combinatorische proces gegenereerd. Dit gehele proces wordt schematisch beschreven in figuur 5.2 en 5.3.
In de C–broncode van Trivium is de cijfertekst beperkt tot 64 bits. Daarom beperken we ons ook tot de generatie van 64 sleutelstroombits per klokpuls. Terwijl de tweede cyclus gebeurt, zorgt het tweede klokproces ervoor dat de sleutelstroombits *z(i)* in een sleutelstroomregister geladen worden. In een bewerking met een XOR-poort van de klare tekst met de sleutelstroom maken we de uiteindelijke cijfertekst.

5.2.3 Testresultaten van Trivium

Hier geven we de resultaten van onze versie van het stroomcijfer Trivium. Het doel van dit stroomcijfer is compactheid: de verhouding tussen slice-count (de ingenomen oppervlakte op de FPGA) en coderingssnelheid moet zo klein mogelijk zijn.
Het "*place and route*"–rapport op FPGA geeft de volgende resultaten:

	Trivium (1 Bit)
# gebruikte slices in FPGA (totaal 33088)	323
minimale klokperiode (op 50% duty cycle) (ns)	8,493
aantal klokpulsen nodig om 1 bit uit te sturen	1

Tabel 5.1: Testresultaten Trivium1b

Het kritische en traagste pad loopt van *counter5* naar *ready* waarbij de tijdsverdeling ongeveer gelijk ligt tussen de logica en de ligging van de signaalbanen. (44.7% logica en 55.3% route)

5.2.4 Meerbitsuitbreiding van Trivium

We hebben al aangehaald dat de ontwerpers van Trivium een flexibel, compact en snel stroomcijfer voor ogen hebben. Dit wordt gerealiseerd met een bitgeoriënteerde aanpak.
De interne toestand moet wel niet–lineair zijn en de operaties moeten zo parallel mogelijk lopen om de niet-lineariteit in de sleutelstroom te behouden. Bij het specifieke geval van Trivium mogen er geen tapbits hergebruikt worden gedurende minstens 64 operaties nadat deze bits aangepast werden. Op deze manier kunnen er maximaal 64 iteraties in 1 keer naar buiten toe gegenereerd worden. De radix wordt verhoogd. De 3 AND-poorten en 11 XOR-poorten van het originele 1-bit schema worden in dit geval gekopieerd. De schematische voorstelling van figuur 5.3 kan dan maximaal 64 keer gekopieerd worden. Het aantal keer dat we ze kopiëren staat gelijk aan het aantal iteraties dat we per klokpuls willen genereren.
Kiest men voor een radix van 64 bits, dan zijn er meer AND-poorten en XOR-poorten nodig. Hierdoor zal natuurlijk de complexiteit van het geheel stijgen.

Throughput	1 bit	8 bit	16 bit	32 bit	64 bit
AND-poorten	3	24	48	96	192
XOR-poorten	11	88	176	352	704

Tabel 5.2: Trivium: aantal AND- en XOR-poorten

Het grote voordeel van deze manier van werken is dat we de frequentie van de klokpulsen met maximum 64 maal kunnen reduceren. We kunnen de frequentie 64 keer kleiner maken wanneer men geen 64 bits per klokpuls nodig heeft, maar bijvoorbeeld slechts één bit. De sleutelstroom wordt dan aan dezelfde frequentie gegenereerd als in het originele 1-bit schema van Trivium.

Na grondige simulatie van deze uitbreiding gaf het "*place and route*"-rapport op FPGA ons de volgende resultaten:

	Trivium 8 bits	Trivium 16 bits	Trivium 32 bits	Trivium 64 bits
# gebruikte slices in FPGA (totaal 33088)	285	303	355	443
minimale klokperiode (op 50% duty cycle) (ns)	5,869	6,579	5,898	6,726
aantal klokpulsen nodig om 1 bit uit te sturen	1/8	1/16	1/32	1/64

Tabel 5.3: Testresultaten Trivium8/16/32/64b

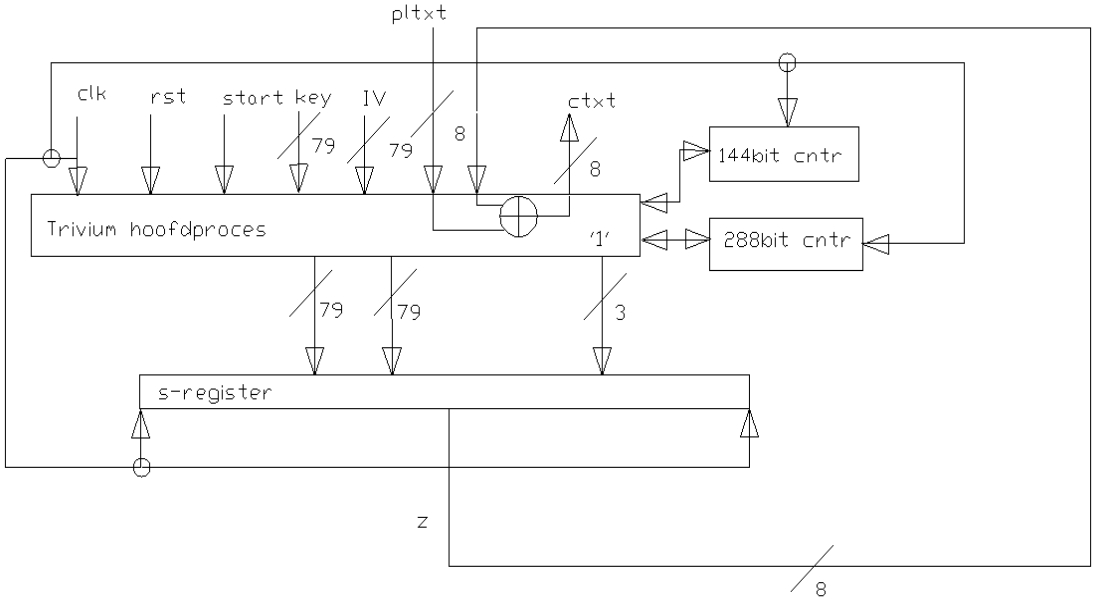
We kunnen duidelijk zien dat bij de uitbreiding naar een hogere radix de ingenomen slices op FPGA ook toenemen. De 64-bit comparator veroorzaakt in elke versie van Trivium het kritische pad: de *counter* moet telkens vergeleken worden met een grote vaste waarde. Bij de meerbitsuitbreiding wordt de hardware efficiënter gebruikt:

	Trivium 8 bits	Trivium 16 bits	Trivium 32 bits	Trivium 64 bits
het traagste pad	counter 14 – s239	init_counter12 – s261	init_counter0 – s202	init_counter20 - counter30
% vertraging van dit pad in logica	45.3	40.4	48.5	33.1
% vertraging van dit pad door routing	54.7	59.6	51.5	66.9

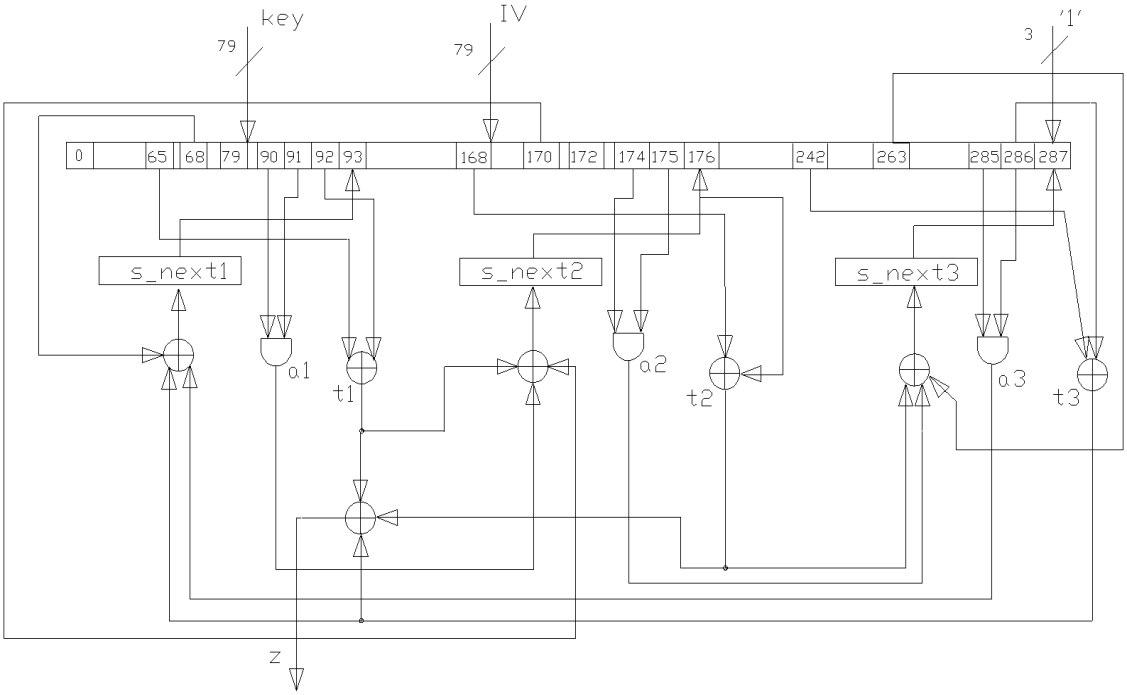
Tabel 5.4: Kritische paden in de meerbitsuitbreiding van Trivium

De routing van de paden wordt belangrijker telkens de radix verhoogd wordt.

5.3 Schematische voorstelling van Trivium



Figuur 5.2: Hoofdproces van Trivium



Figuur 5.3: Intern s-register

HOOFDSTUK 6

Stroomcijfer Rabbit

6.1 Theorie achter Rabbit

De ontwerpers van Rabbit hadden 2 doelstellingen voor ogen: hun stroomcijfer moest sneller zijn dan de meeste bestaande stroomcijfers en een 128-bit sleutel moest voldoende zijn om maximaal 2^{64} blokken klaartekst te coderen. Dit is een vrij kleine sleutel voor het grote beoogde aantal blokken klaartekst en is dus mijlenver verwijderd van de *"one-time-pad"*-methode. Zonder kennis van de sleutel is het zeer moeilijk om de sleutelstroom te onderscheiden van een "*one-time pad*". Een voorbeeld: een onderscheid maken tussen 2^{64} blokken cijfertekst, door Rabbit gecodeerd met een ongekende sleutel, én de uitgang van een echte willekeurige bitgenerator zou niet mogelijk mogen zijn. Het aantal stappen om toch lineariteit te ontdekken zou in principe meer tijd moeten kosten dan de zoektocht naar de ongekende sleutel. We spreken dan over het nagaan van 2^{128} sleutels.

6.1.1 Werking van Rabbit

Rabbit had aanvankelijk een 128-bit sleutel nodig. Later kwam er een extra 64-bit grote initiële waarde (*IV*) bij. Na elke iteratie van het systeem komt er een blok van 128 bits naar buiten. Dit blok ontstaat door het combineren van de interne toestandsbits, die constant vernieuwd worden.
De uiteindelijke codering en decodering bestaat dan uit een operatie met een XOR-poort van de klare tekst of cijfertekst met de gegenereerde 128 sleutelstroombits.

De grootte van dit stroomcijfer is 513 bits.
De verdeling gaat als volgt: acht toestandsvariabelen van 32-bit, acht tellers van 32-bit en één teller–dragerbit. De acht toestandsvariabelen worden telkens vernieuwd door acht niet-lineaire functies die afhankelijk zijn van de tellers.

6.1.1.1 Inladen van de sleutel

De initialisatie van het algoritme stopt de 128-bit sleutel in zowel de 8 toestandsvariabelen als in de 8 tellers. Het werkt als volgt :

$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} \lozenge k_j & (1) \\ k_{(j+5 \bmod 8)} \lozenge k_{(j+4 \bmod 8)} & (2) \end{cases}$$

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} \lozenge k_{(j+5 \bmod 8)} & (1) \\ k_j \lozenge k_{(j+1 \bmod 8)} & (2) \end{cases}$$

Het samenvoegen van de verschillende stukken van de sleutel gebeurt *als j even is (1) en als j oneven is (2)* voor zowel de nieuwe interne toestanden als de tellers.

Dan wordt het systeem 4 keer geïtereerd zoals het in de *next-state function* beschreven wordt. Dit moet gebeuren om de overeenkomsten tussen de sleutelbits en de bits van de begintoestand zoveel mogelijk te reduceren. Nadat ze genoeg gemengd zijn, worden de tellervariabelen opnieuw geïntialiseerd om te vermijden dat de sleutel ontdekt wordt bij omkering van het tellersysteem. Het werkt op deze manier:

$$c_{j,4} = c_{j,4} \oplus x_{(j+4 \bmod 8),4}$$

6.1.1.2 Inladen van de IV

Na het inladen van de sleutel volgt de interne hoofdtoestand. Deze toestand moet telkens doorlopen en vernieuwd worden.
Een kopie van deze hoofdtoestand wordt aangepast met behulp van de *IV*: De tellers *c* worden aangepast in functie van de *IV* via een operatie met een XOR-poort van de 64-bit *IV* met alle 256 bits van de tellertoestanden.

De *IV* wordt voorgesteld als een blok van 64 bits en gaat als volgt:

$$\begin{aligned}
c_{0,4} &= c_{0,4} \oplus IV^{[31..0]} \\
c_{1,4} &= c_{1,4} \oplus IV^{[63..48]} \Diamond IV^{[31..16]} \\
c_{2,4} &= c_{2,4} \oplus IV^{[63..32]} \\
c_{3,4} &= c_{3,4} \oplus IV^{[47..32]} \Diamond IV^{[15..0]} \\
c_{4,4} &= c_{4,4} \oplus IV^{[31..0]} \\
c_{5,4} &= c_{5,4} \oplus IV^{[63..48]} \Diamond IV^{[31..16]} \\
c_{6,4} &= c_{6,4} \oplus IV^{[63..32]} \\
c_{7,4} &= c_{7,4} \oplus IV^{[47..32]} \Diamond IV^{[15..0]}
\end{aligned}$$

Na deze aanpassing wordt alles nogmaals vier keer geïtereerd zoals in de *next-state fuction* beschreven staat. Zo maak je de toestandsbits onafhankelijk van de *IV*-bits. De verandering van de teller door de *IV* garandeert dat alle 64-bit IV blokken tot unieke sleutelstromen leiden.

6.1.1.3 Next-state-function

Dit is het hart van het Rabbit-algoritme. Deze functie wordt tijdens het hele proces uitgevoerd zodat de interne toestanden constant vernieuwd worden op basis van de tellers:

$$x_{0,i+1} = g_{0,i} + (g_{7,i} <<< 16) + (g_{6,i} <<< 16)$$
$$x_{1,i+1} = g_{1,i} + (g_{0,i} <<< 8) + g_{7,i}$$
$$x_{2,i+1} = g_{2,i} + (g_{1,i} <<< 16) + (g_{6,i} <<< 16)$$
$$x_{3,i+1} = g_{3,i} + (g_{2,i} <<< 8) + g_{1,i}$$
$$x_{4,i+1} = g_{4,i} + (g_{3,i} <<< 16) + (g_{6,i} <<< 16)$$
$$x_{5,i+1} = g_{5,i} + (g_{4,i} <<< 8) + g_{3,i}$$
$$x_{6,i+1} = g_{6,i} + (g_{5,i} <<< 16) + (g_{6,i} <<< 16)$$
$$x_{7,i+1} = g_{7,i} + (g_{6,i} <<< 8) + g_{5,i}$$
$$g_{j,i} = \left(\left(x_{j,i} + c_{j,i+1}\right)^2 \oplus \left(\left(x_{j,i} + c_{j,i+1}\right)^2 >> 32\right)\right) \bmod 2^{32}$$

6.1.1.4 Tellersysteem

Na het statische inladen van de tellerregisters worden de tellers dynamisch gemaakt door ze constant te vernieuwen:

$$c_{0,i+1} = c_{0,i} + a_0 + \phi_{7,i} \bmod 2^{32}$$
$$c_{1,i+1} = c_{1,i} + a_1 + \phi_{0,i+1} \bmod 2^{32}$$
$$c_{2,i+1} = c_{2,i} + a_2 + \phi_{1,i+1} \bmod 2^{32}$$
$$c_{3,i+1} = c_{3,i} + a_3 + \phi_{2,i+1} \bmod 2^{32}$$
$$c_{4,i+1} = c_{4,i} + a_4 + \phi_{3,i+1} \bmod 2^{32}$$
$$c_{5,i+1} = c_{5,i} + a_5 + \phi_{4,i+1} \bmod 2^{32}$$
$$c_{6,i+1} = c_{6,i} + a_6 + \phi_{5,i+1} \bmod 2^{32}$$
$$c_{7,i+1} = c_{7,i} + a_7 + \phi_{6,i+1} \bmod 2^{32}$$

Na elke iteratie wordt een nieuwe tellerwaarde berekend op basis van sleutel, *IV* en de dragerbit $\phi_{j,i+1}$. Deze waarde wordt gegeven op basis van de volgende voorwaarden:

$$\phi_{j,i+1} = \begin{cases} 1 \Leftarrow if\,(c_{0,i} + a_0 + \phi_{7,i}) \geq 2^{32} \wedge j = 0 \\ 1 \Leftarrow if\,(c_{j,i} + a_j + \phi_{j-1,i+1}) \geq 2^{32} \wedge j = 0 \\ 0 \Leftarrow if\,(otherwise) \end{cases}$$

De constanten *a* van het tellersysteem zijn de volgende:

$$a_0 = 0x4D34D34D$$
$$a_1 = 0xD34D34D3$$
$$a_2 = 0x34D34D34$$
$$a_3 = 0x4D34D34D$$
$$a_4 = 0xD34D34D3$$
$$a_5 = 0x34D34D34$$
$$a_6 = 0x4D34D34D$$
$$a_7 = 0xD34D34D3$$

Na elke iteratie krijgen we onmiddellijk een blok van 128 sleutelstroombits *s* naar buiten toe. Die bits worden op de volgende manier gegenereerd:

$$s_i^{[15..0]} = x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]}$$
$$s_i^{[31..16]} = x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]}$$
$$s_i^{[47..32]} = x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]}$$
$$s_i^{[63..48]} = x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]}$$
$$s_i^{[79..64]} = x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]}$$
$$s_i^{[95..80]} = x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]}$$
$$s_i^{[111..96]} = x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]}$$
$$s_i^{[127..112]} = x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}$$

waarbij s_i: een128-bit sleutelstroomblok per iteratie i .

6.1.1.5 Codering en decodering

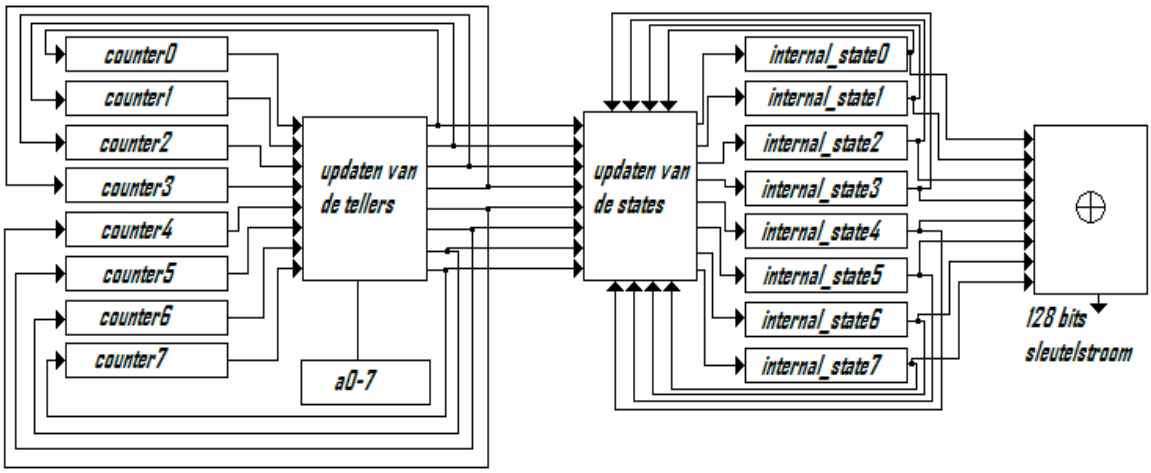
Nu worden de sleutelstroombits in een XOR-operatie met de klare tekst, p_i, gestopt wanneer we de cijfertekst, s_i, gaan genereren. Hetzelfde gebeurt met de cijfertekst wanneer we aan decodering doen.

$$c_i = p_i \oplus s_i$$
$$p_i = c_i \oplus s_i$$

waar s_i en p_i blokken van 128 bits zijn. [16]

[16]LANO, J. , Boesgaard, M., Vesterager, M., Christensen, T., Zenner, E., *Rabbit*, 2005, on line, http://www.ecrypt.eu.org/stream/rabbit.html, 20 november 2005

De volgende figuur schetst de werking van het stroomcijfer.



Figuur 6.1: Algemeen schema van Rabbit

6.2 Implementatie van Rabbit

De ingangen en de uitgangen van het stroomcijfer zijn een sleutel van 128 bits, een *IV* van 64 bits, een kloksignaal, een *reset* en een startsignaal. Nadat de codering beëindigd is, wordt de resulterende sleutelstroom van 128 bits parallel naar buiten gestuurd.

In een hoofdbouwblok beschrijven we een FSM die de verschillende fasen voorstelt. Dit controleblok moet ervoor zorgen dat de juiste bewerkingen op de juiste momenten én onder de juiste voorwaarden gebeuren. Het gekoppelde bouwblok definieert de *next-state-function*, het tellersysteem (*countersystem*) en de voorbereidende fase.

6.2.1 Controleblok

De FSM doorloopt de volgende toestanden: *idle*, *load*, *nsf*, *counterload*, *iv_load*, *iv_fase* en *ks_production*.

Op basis van het startsignaal zal de FSM overgaan van *idle* naar *load*. Hierin wordt de sleutel geladen. Vanuit deze toestand loop je door naar *nsf* waarin de *next-state-function* start. Deze toestand zal zich viermaal herhalen vooraleer ze overgaat naar *counterload*. Het systeem zal vier klokpulsen lang uitgevoerd worden.
Vanuit de toestand *counterload*, waarin de tellers vermengd worden met de *internal_state* -registers, kom je in *iv_load* en daarna in *iv_fase* terecht. In de eerste toestand zullen alle tellers met de initiële waarde gemengd worden om dan in vier klokpulsen het volledige systeem weer uit te voeren. Alles wordt gemengd opdat noch de sleutel, noch de initiële waarde achterhaald kunnen worden.
De laatste toestand heet *ks_production* waarin de sleutelstroom geproduceerd wordt. Alles is nu gereed om een boodschap te coderen.
Het uitgangsproces van de FSM geeft, op basis van de toestand waarin het *next_state*-proces zich bevindt, alle benodigde toestandsvariabelen een digitale waarde. De toestandsvariablen *key_setup*", "*iv_setup*, *counter_load*, *nx_st_en* en *ks_en* zullen de juiste registers van Rabbit aansturen tijdens de verschillende toestanden van het coderingsproces.
De toestandsvariabelen "*ksc_en*" en "*isc_en*" vernieuwen de tellerregisters.

Buiten de FSM bevindt er zich nog een ander klokproces in het controleblok. Dit proces is sequentieel en zorgt er voor dat de tellers op het juiste moment verhoogd worden. Het is beschreven in figuur 6.2.

6.2.2 Implementatie van de functies

We reserveren registers voor de interne toestanden of de *reg_internal_state (0-7)*. Dit zijn er acht, elk 32 bits groot. We doen hetzelfde met registers van dezelfde grootte om de tussenbewerkingen van deze interne toestanden op te slaan. Deze noemen we *g (0-7)*, *product (0-7)* en *p (0-7)*. Omdat het tellersysteem ook geheugenruimte nodig heeft, voorzien we 16 keer 32-bits registers voor *counter (0-7)* en *reg_counter (0-7)*. De vernieuwde tellerwaarde wordt telkens berekend op basis van de sleutel, de *IV* en de

teller-dragerbit. We voorzien de registers *phi (0-7)* en *reg_phi (0-7)* om deze waarden bij te houden. De constanten *a (0-7)* geven we hun vaste waardes in acht 32-bit registers.
Nu bespreken we het tellerproces, het *internal_state*-proces en het phi-proces zoals ze in figuur 6.3 aan elkaar gekoppeld zijn.

6.2.2.1 Tellerproces

Op basis van de variabelen *rst*, *key_setup*, *counter_load*, *iv_setup* en *next_state_en*, die vanuit het hoofdbouwblok komen, lopen we doorheen het tellerproces. Eerst laden we de sleutel in de *counter (0-7)* zoals in figuur 6.4 voorgesteld wordt. Deze toestand heet in de FSM *keysetup*. Daarna berekenen we de tellerwaardes voor de eerste keer. Voor de teller-dragerbit gebruiken we de nieuwe waardes, behalve de achtste teller-dragerbit die de oude waarde gebruikt. Dit is logisch aangezien bij elke nieuwe iteratie de laatste tellerwaarde de nieuwe wordt. Deze toestand heet in de FSM *next_state_en*. Daarna vernieuwen we de *counter (0-7)* door middel van een bewerking met een XOR-poort van de vorige tellerwaarde (*reg_counter (0-7)*) en de vorige interne toestand (*reg_internal_state (0-7)*). Deze toestand heet in de FSM *counterload*.

In de volgende stap vernieuwen we de teller voor de laatste keer door middel van een bewerking met een XOR-poort van de vorige tellerwaarde (*reg_counter (0-7)*) en de *IV*. De nieuwe tellerwaarde laden we dan door een klokproces in de registers *reg_counter (0-7)* na elke iteratie. Het vernieuwen van de tellers wordt schematisch weergegeven in figuur 6.5.

Het gelijklopende *phi-proces* zorgt ervoor dat, op basis van de 33ste bit van de huidige tellerwaarde en de vorige tellerwaarde (*counter (0-7) (32)* en *reg_counter (0-7) (32)*), de juiste *phi*-waarde in de volgende iteratie gebruikt wordt. Het is de bedoeling dat *phi (0-7)* hoog wordt als deze twee waarden, ‘0’ of ‘1’, van elkaar verschillen.

We hebben de werking van *phi* zoals in figuur 6.9 geïnterpreteerd: wanneer de iteratie loopt (als *next_st_en* hoog is), moeten we de *phi*-waarde bijhouden en opslaan in de voorziene registers (*reg_phi (0-7)*). Wanneer we de tellerwaarde dan vernieuwen (wanneer *iv_setup* hoog wordt) zetten we alle *phi*-waarden op ‘0’, behalve de waarde van *phi7*.

6.2.2.2 “Internal state”-proces

Dit proces werkt zoals het tellerproces op basis van een aantal variabelen: *rst*, *next_state_en* en *key_setup*. Daarna doorlopen we het *internal state*-proces: we laden de sleutel in de *internal_state (0-7)* zoals in figuur 6.6. Deze toestand noemen we ook *key_setup*.

De volgende belangrijke toestand voor dit proces is de *next_st_en*. Wanneer dit signaal hoog is, worden alle interne toestanden vernieuwd door middel van een samenvoeging en optelling van de *g*-registers.

We hebben de complexe bewerking van de *next-state-function* hardwarematig in verschillende stappen gesplist:

- de *p*-registers (*p (0-7*)) worden gemaakt door een optelling van de oude interne toestanden *reg_internal_state (0-7)* met de nieuwe tellerwaarden *counter (0-7)*. (Cfr. figuur 6.7)

- we vermenigvuldigen de *p*-registers met elkaar en steken dit in *product (0-7)*. Uiteindelijk bekomen we dan de *g*-waarden door de 32 minst beduidende bits van *product (0-7)* met de 32 meest beduidende bits van *product (0-7)* in een bewerking met een XOR-poort te steken. (Cfr. figuur 6.7)

6.2.2.3 Generatie van de sleutelstroom

We maken de generatie van de sleutelstroom actief wanneer de *ks_en* hoog is. Door middel van een bewerking met een XOR-poort van de oude interne toestanden (reg_*internal_state (0-7*)) worden de *s*-registers gevuld met 128 bits in het totaal.

Op dat ogenblik heeft de FSM haar volledige cyclus doorlopen en is de sleutelstroom klaar om de klare tekst tot cijfertekst te bewerken.

6.2.3 Testresultaten van Rabbit

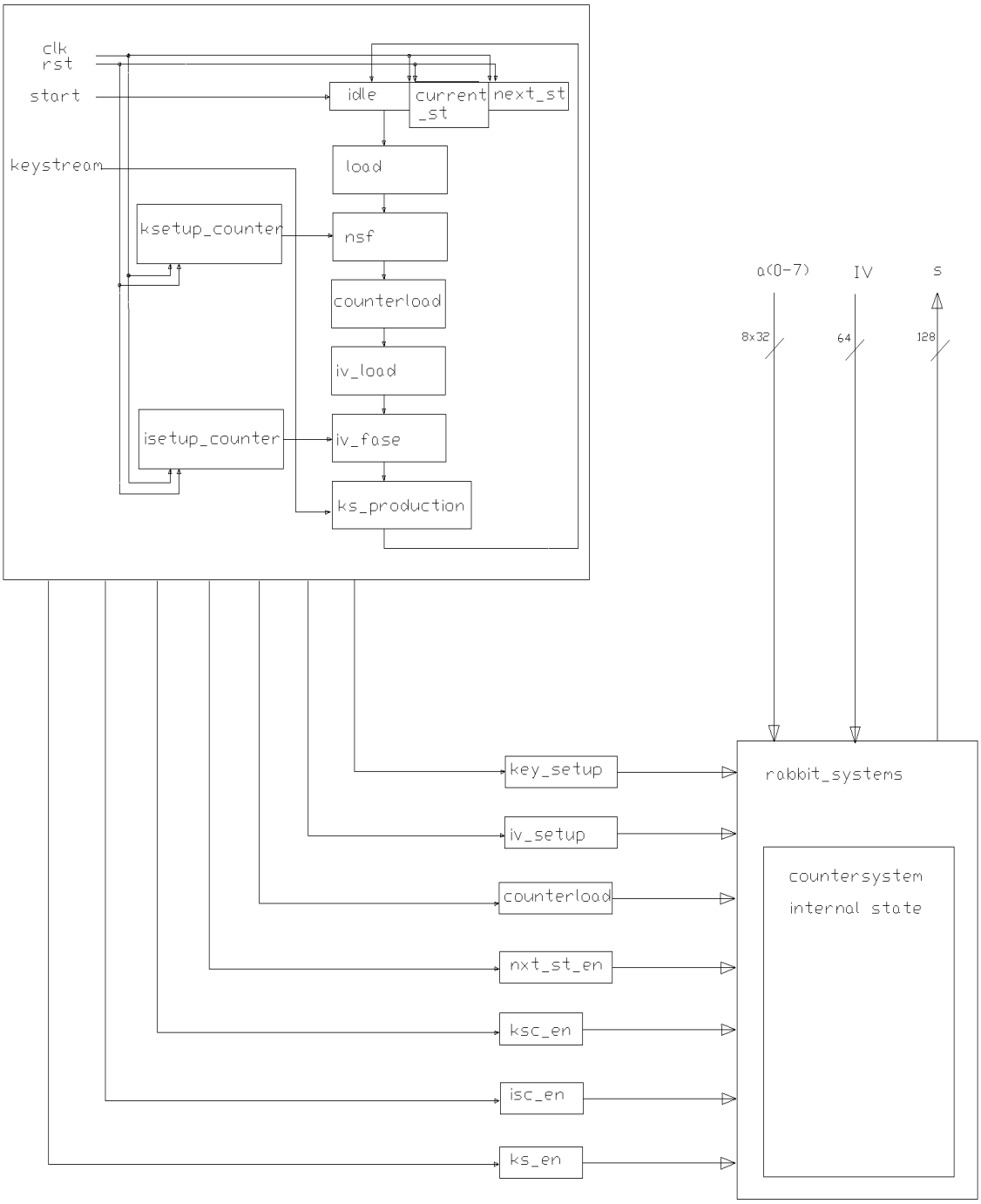
Hier geven we de resultaten van onze versie van het stroomcijfer Rabbit.
Het "*place and route*"-rapport op FPGA geeft de volgende resultaten:

	Rabbit
# gebruikte slices in FPGA (totaal 33088)	2314
minimale klokperiode (op 50% duty cycle) (ns)	1,447
aantal klokpulsen nodig om 1 bit uit te sturen	1/128

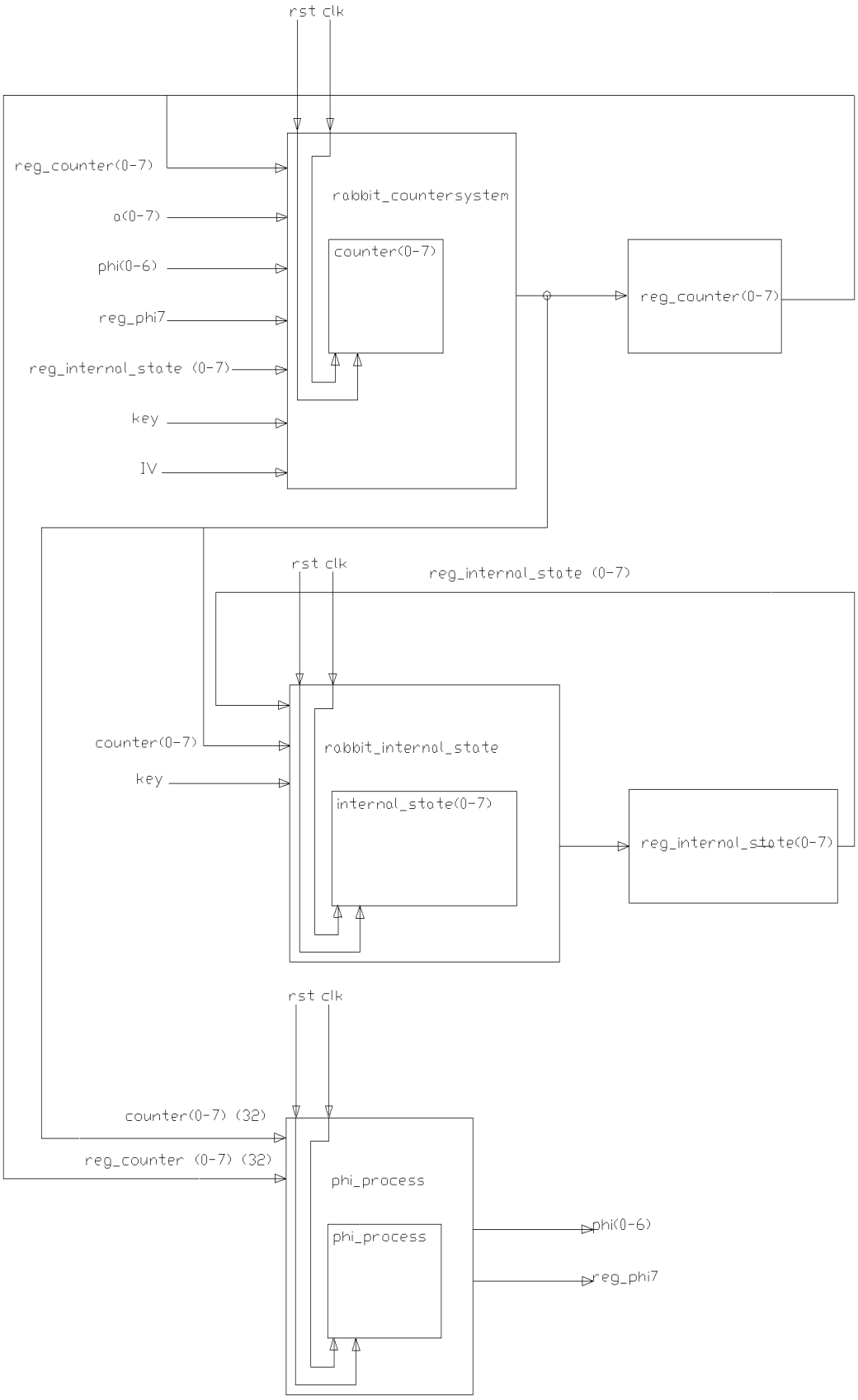
Tabel 6.1: Testresultaten Rabbit

Het grote aantal registers dat nodig is om alle tussenliggende waarden bij te houden, maakt het stroomcijfer te groot. De ontwerpers hebben hun best gedaan om de *key* en *IV* zo onherkenbaar mogelijk te maken. De grote vermenigvuldigers zijn voorzien op het gebruikte type FPGA en beperken de grootte.
Het voordeel van Rabbit is een zeer kleine maximale vertraging. De aansturing van de hardware door de klok is te traag om de combinatorische stukken op maximale snelheid te laten werken. De oorzaak van de maximale vertraging vinden we bij de klokbanen.
Het grote aantal bits per klokpuls aan de uitgang geeft een hoge throughput van 128 bits per klokpuls oftewel 88,5 Gbit/s. Hiermee hebben de ontwerpers hun doel bereikt: Rabbit is een zeer snel stroomcijfer.

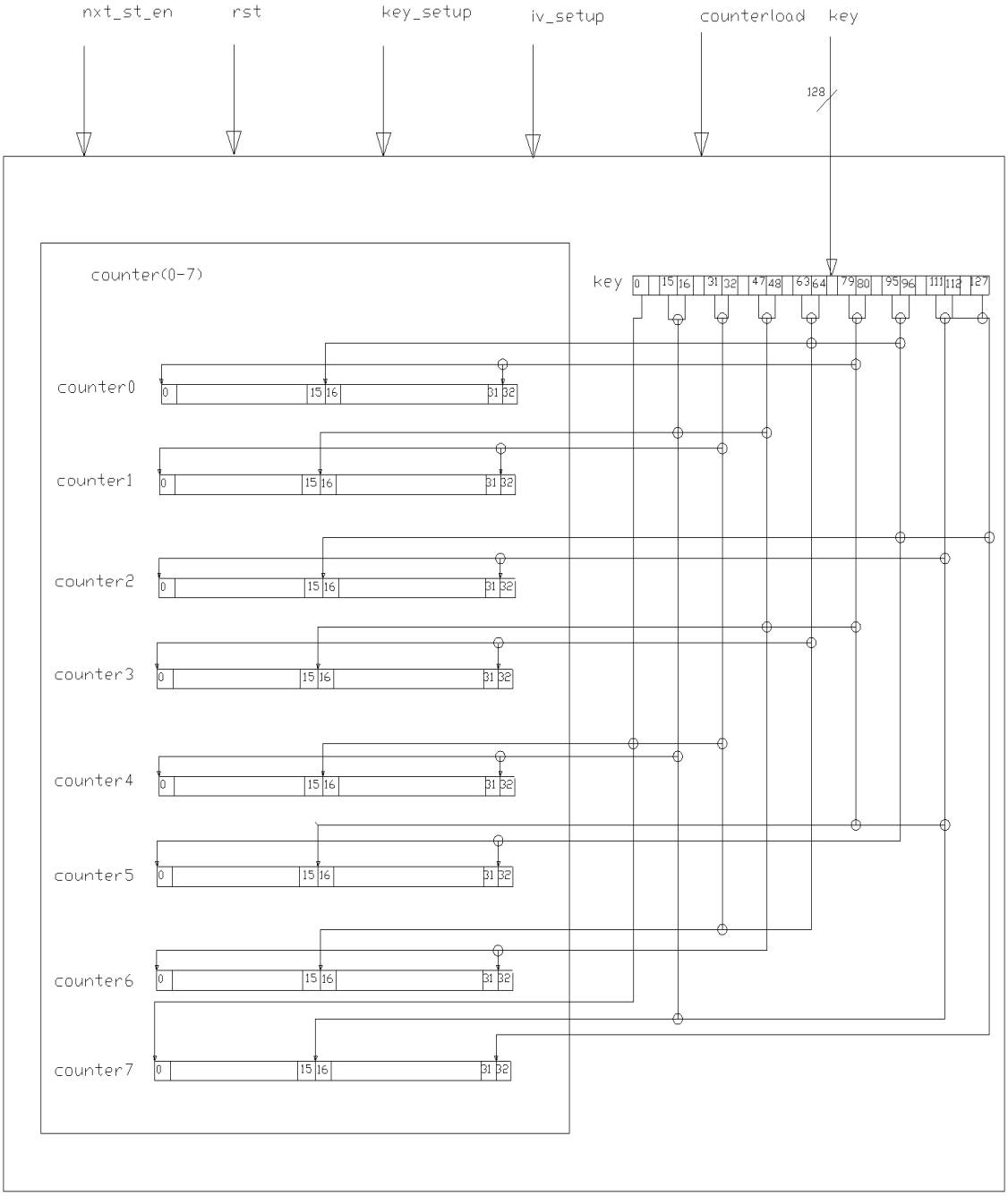
6.3 Schematische voorstelling van Rabbit



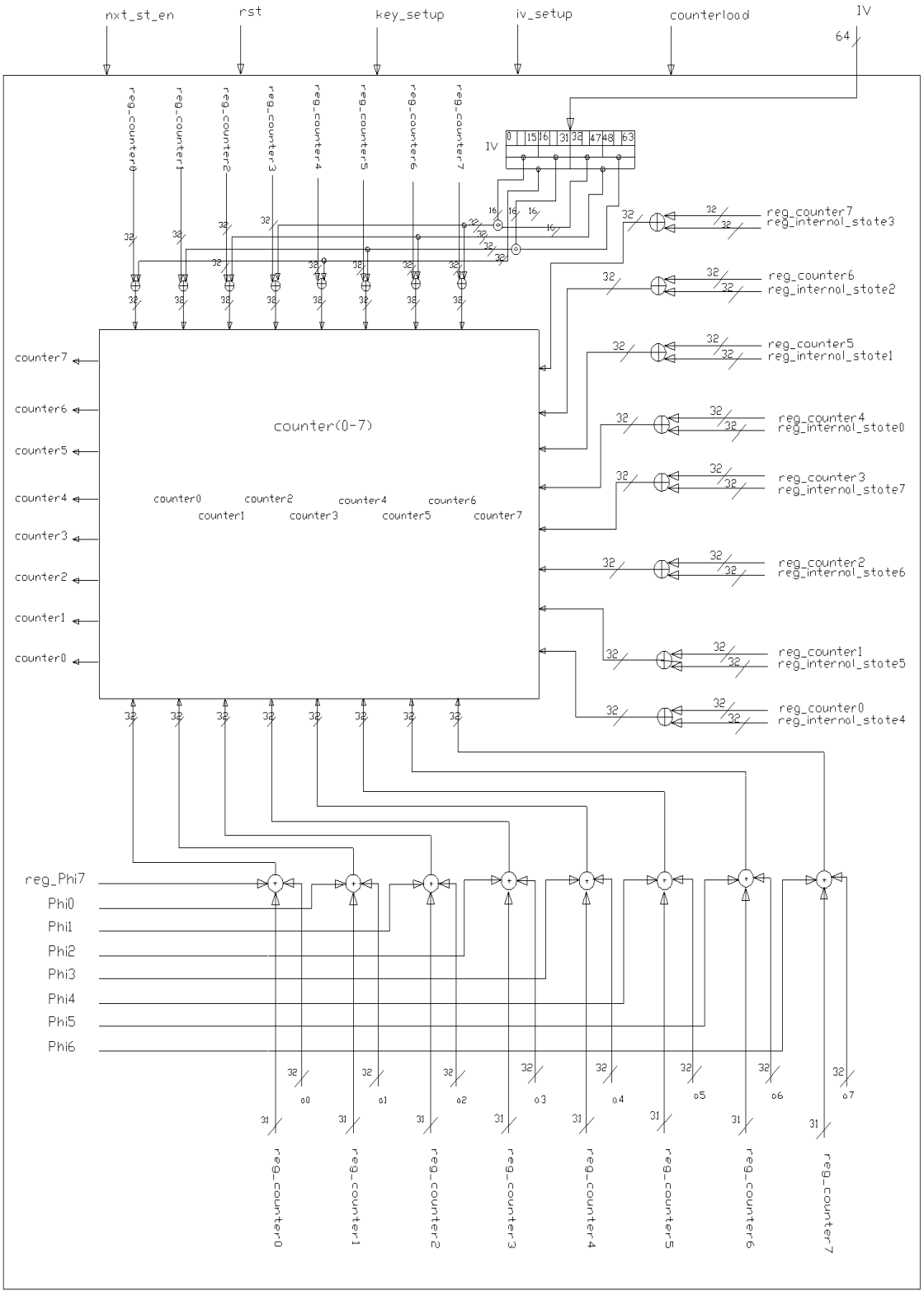
Figuur 6.2: Hoofdproces van Rabbit



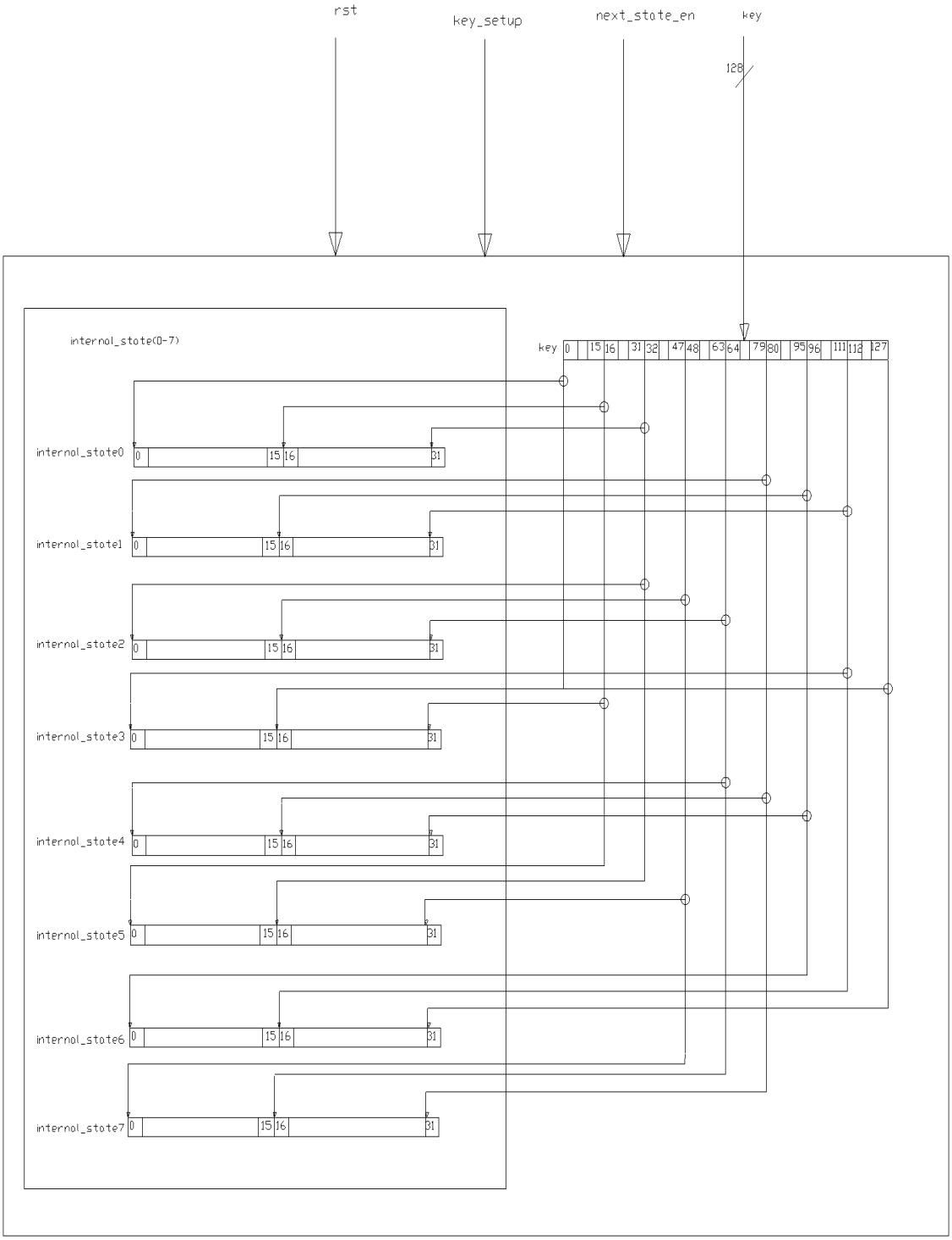
Figuur 6.3: Aansturing van de drie subprocessen (1-2-3) (rabbit_systems)



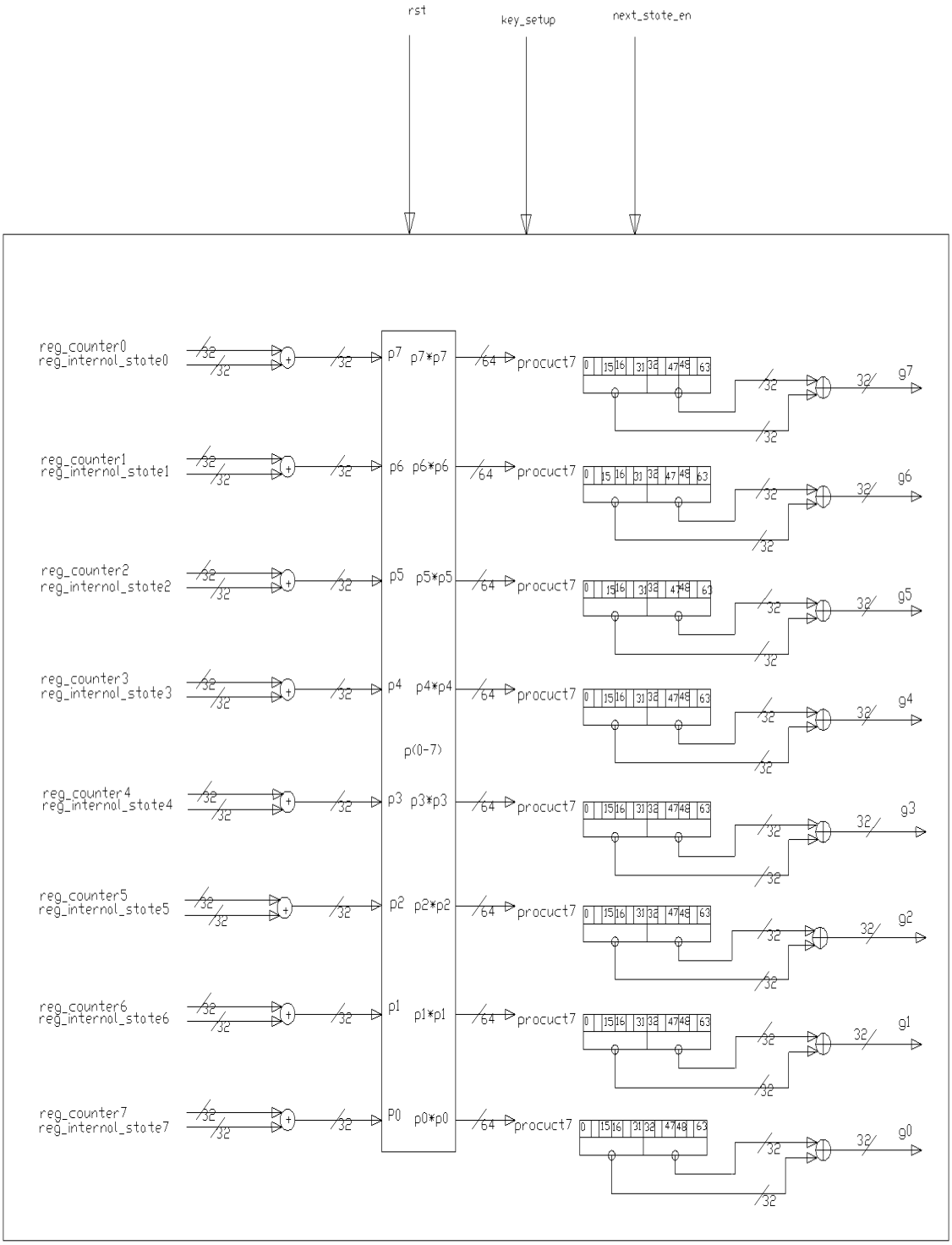
Figuur 6.4: Tellerproces (1) deel 1
(countersystem)



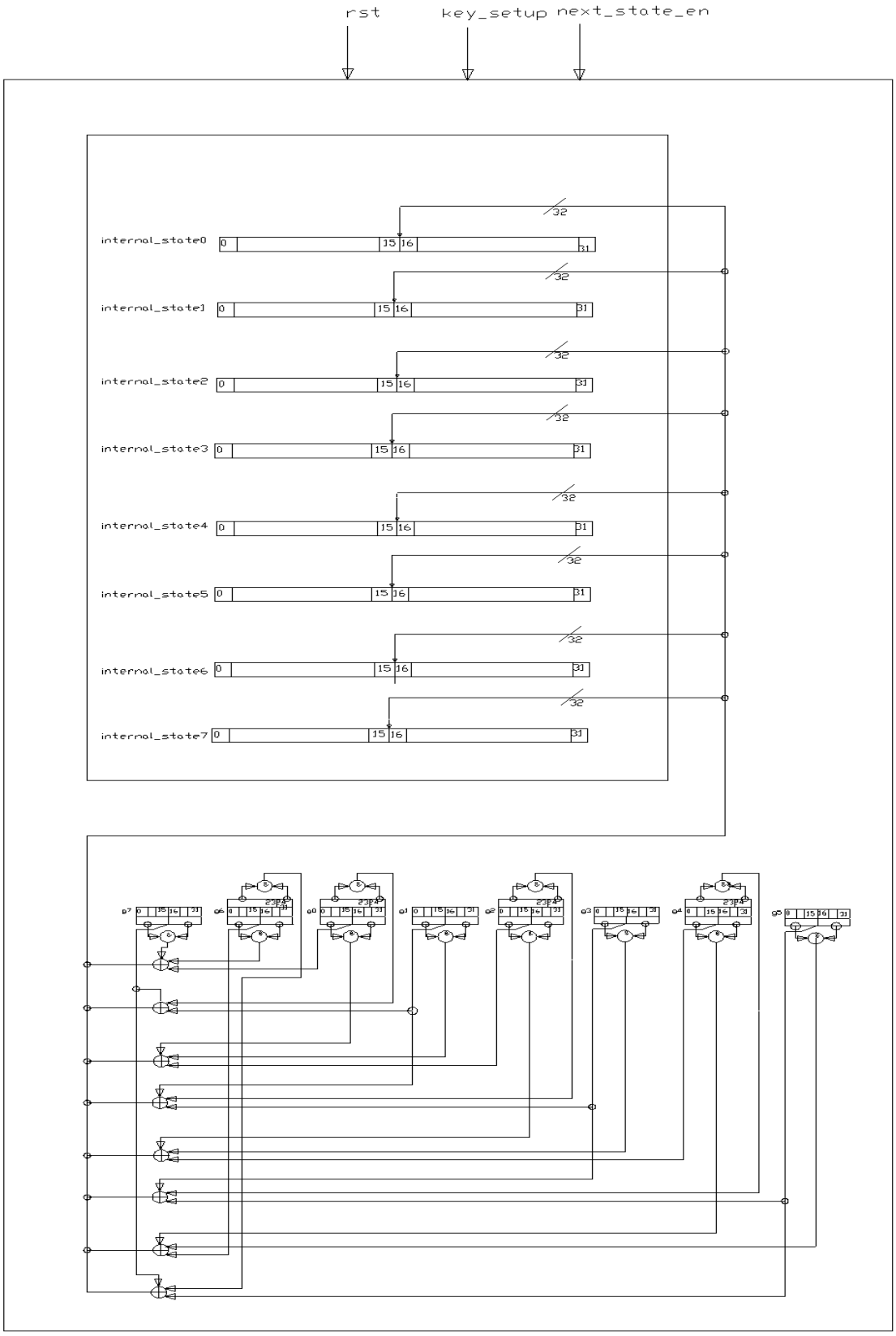
Figuur 6.5: Tellerproces (1) deel 2 (countersystem)



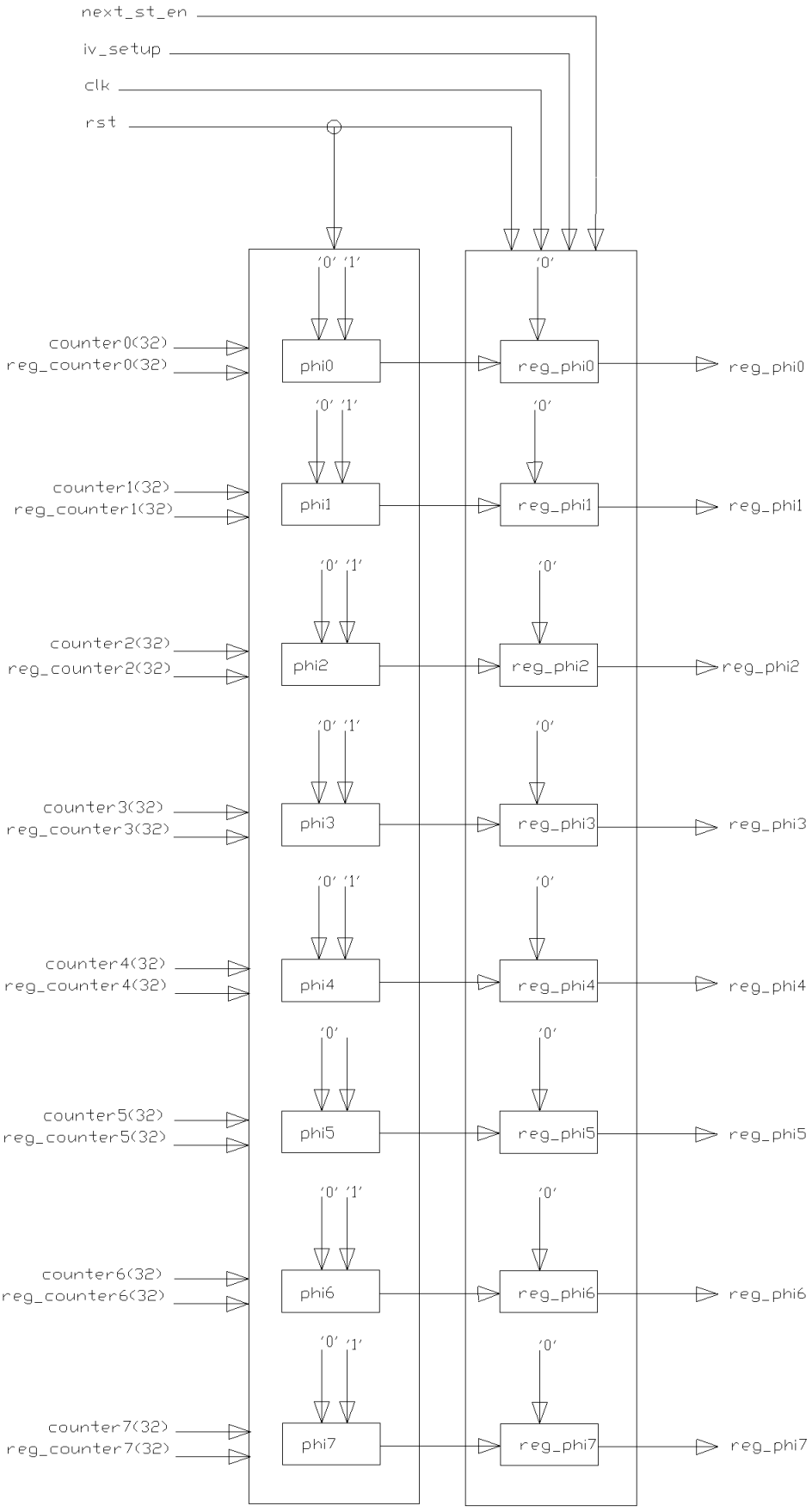
Figuur 6.6: Intern toestandsproces (2) deel 1 (internal_state)



Figuur 6.7: Intern toestandsproces (2) deel 2 (internal_state)



Figuur 6.8: Intern toestandsproces (2) deel 3 (internal_state)



Figuur 6.9: Phi-proces(3)

HOOFDSTUK 7

Stroomcijfer DECIMv2

7.1 Theorie achter DECIMv2

De ontwerpers van DECIMv2 hebben hun ontwerp, hardwarematig gezien, niet complex willen maken. Dit stroomcijfer maakt om die reden gebruik van twee specifieke structuren om de lineariteit te doorbreken.
Enerzijds gebruikt men een LFSR, gevolgd door een niet-lineaire filter, en anderzijds een mechanisme dat de sleutelstroom op een onregelmatige manier uitdunt, de *ABSG*.

DECIMv2 is een verbeterde versie van DECIM. De ontwerpers hebben in de werking van DECIM twee gebreken ontdekt: de initialisatie en de filterfunctie. In het algemeen genereert DECIM een binaire sequentie *y* vanuit een LFSR. Daarna wordt de *y*-sequentie gefilterd door een Booleaanse functie en het *ABSG*-mechanisme.

7.1.1 Werking van DECIMv2

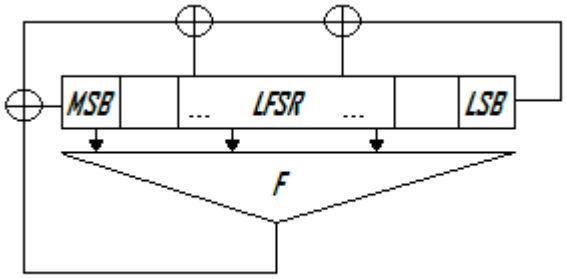
DECIM heeft een LFSR met een grootte van 192 bits. Tijdens de initialisatie worden de bits van de LFSR berekend door middel van een sleutel en een initiële waarde. Deze bits liggen aan de basis van de sleutelstroomgeneratie. De gebreken in de initialisatie zijn te wijten aan het verplaatsingseffect, die afhankelijk is van de *ABSG*-uitgang, van de elementen in de LFSR. De kans dat deze elementen nooit verplaatst en de plaats niet vernieuwd zou worden was vrij groot. Men kan de sleutel en de initiële waarde (*IV*) van DECIM met een goed uitgedachte aanval gemakkelijk terugvinden. Vandaar dat de ontwerpers een tweede versie van DECIM gemaakt hebben, met name DECIMv2.

7.1.1.1 Initialisatiefase

In DECIMv2 wordt dit probleem met een veiliger systeem aangepakt, zonder permutaties op de elementen uit te voeren. De geheime sleutel *K* van 80 bits en de initialisatievector *IV* van 64 bits worden als volgt in de LFSR geladen:

$$x_i = \begin{cases} K_i & 0 \le i \le 79 \\ K_{i-80} \oplus IV_{i-80} & 80 \le i \le 143 \\ K_{i-80} \oplus IV_{i-144} \oplus IV_{i-128} \oplus IV_{i-112} \oplus IV_{i-96} & 144 \le i \le 159 \\ IV_{i-160} \oplus IV_{i-128} \oplus 1 & 160 \le i \le 191 \end{cases}$$

De LFSR wordt nu 768 keer geklokt waarbij steeds een nieuwe bit binnenkomt op de plaats van de meest beduidende bit (MSB: most significant bit). Zo benoemen we de minst beduidende bit als LSB: least significant bit.



Figuur 7.1: Initialisatiefase van DECIMv2

De terugkoppeling van LFSR wordt als volgt beschreven:

$$P(X) = X^{192} + X^{189} + X^{188} + X^{169} + X^{156} + X^{155} + X^{132} + X^{131} + X^{94} + X^{77} + X^{46} + X^{17} + X^{16} + X^5 + 1$$

Op elke tijdstip *t* wordt het laatste register van de LFSR x_{191} vernieuwd op basis van een bewerking met een XOR-poort met de uitgang van de filterfunctie y_t en de teruggekoppelde bit lv_t.

$$x_{191} = lv_t \oplus y_t$$

7.1.1.2 Filterfunctie

De filterfunctie is een 14-variabelen grote Booleaanse functie:

$$F : F_2^{14} \to F_2; a_1,...,a_{14} \mapsto f(a_1,...,a_{13}) \oplus a_{14}$$

waar f een kwadratische symmetrische Booleaanse functie is:

$$f(a_1,...,a_{13}) = \bigoplus_{1 \le i < j \le 13} a_i a_j \bigoplus_{1 \le 13} a_i$$

De tapposities voor de filterfunctie zijn:

$$191 - 186 - 178 - 172 - 162 - 144 - 111 - 104 - 65 - 54 - 45 - 28 - 13 - 1$$

De uitgang van de filter, de *y*-sequentie, wordt dan als volgt beschreven:

$$y_t = f(s_{t+191}, s_{t+186}, s_{t+178}, s_{t+172}, s_{t+162}, s_{t+144}, s_{t+111}, s_{t+104}, s_{t+65}, s_{t+54}, s_{t+45}, s_{t+28}, s_{t+13}) \oplus s_{t+1}$$

Het grootste gebrek van DECIMv1 situeert zich in de filterfunctie waar de *y*-sequentie gegenereerd wordt. Deze sequentie is de uitgang van een symmetrische Booleaanse functie. Er bestaat een overeenkomst tussen de uitgangen en ingangen van de functie. Dit komt voor als de ingangsvectoren gezamenlijke elementen hebben. Bij DECIMv2 hebben ze ervoor gezorgd dat deze overeenkomsten niet meer kunnen voorkomen.

7.1.1.3 ABSG

ABSG staat voor *"Alternating Bit-Search Generator"* oftewel alternerende bitzoekende generator. Nadat de initialisatiefase voorbij is, zal de uitgang van de filter niet meer naar de LFSR teruggebracht worden. De uitgang dient als ingang van de *ABSG*.
De *ABSG* genereert de sleutelstroom op basis van de bitsequentie die uit de filter komt. De bitsequentie *y* wordt gesplitst in kleinere sequenties van de vorm ($\overline{b}$, b^i, $\overline{b}$), waarbij *i* = 0 en *b* {0,1} en $\overline{b}$ voor de inversie van *b* staat.
Voor elke kleinere sequentie ($\overline{b}$, b^i, $\overline{b}$) is de uitgang van de *ABSG* gelijk aan *b* indien *i* = 0, anders is de uitgang gelijk aan $\overline{b}$.

Deze sequentie wordt dan in de *ABSG* gestoken: $y = (y_t)_t \geq 0$.

De *ABSG* blijft onveranderd ten opzichte van DECIMv1. De *y*-sequentie wordt omgezet naar de *z*-sequentie op de volgende manier:

Het *ABSG*-algoritme wordt hieronder gegeven:

Ingangen: (y_0,y_1,…)
Eerst: $i \leftarrow 0$; j $\leftarrow 0$;
Doorloop de volgende stappen:

1. $e \leftarrow y_i, z_j \leftarrow y_{i+1}$;
2. $i \leftarrow i+1$;
3. $while(y_i = \overline{e}) \Rightarrow i \leftarrow i+1$;
4. $i \leftarrow i+1$;
5. $z_j naar \cdot buiten$;
6. $j \leftarrow j+1$;

Ter verduidelijking geven we een voorbeeld, stel *y* = 0101001110100100011101 als ingangssequentie, dan zal de uitgang van de *ABSG* als volgt bekomen worden:

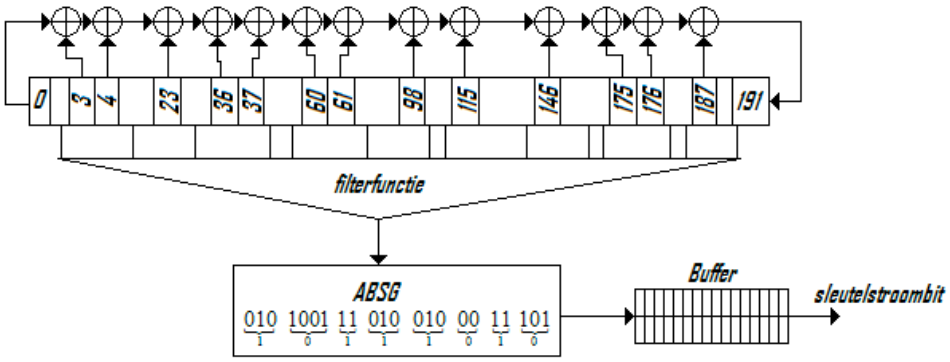
$$\underbrace{010}_{1}\ \underbrace{1001}_{0}\ \underbrace{11}_{1}\ \underbrace{010}_{1}\ \underbrace{010}_{1}\ \underbrace{00}_{0}\ \underbrace{11}_{1}\ \underbrace{101}_{0}$$

De uitgang van de *ABSG* wordt aangeduid door $z = (z_t)_t \geq 0$, in dit voorbeeld is *z* = 10111010.

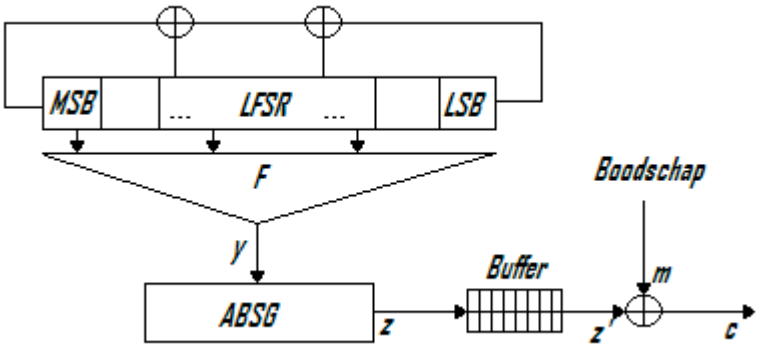
Nadat de *ABSG* doorlopen is, wordt er een buffer van 32 bits voorzien. Om een constante bitstroom te voorzien aan de uitgang van DECIMv2 is dit nodig. De uitgang van de *ABSG* geeft haar bits namelijk onregelmatig naar buiten. Per drie ingangen die de ABSG krijgt van de filterfunctie, komt er gemiddeld 1 bit naar buiten.

Deze buffer geeft telkens 1 bit naar buiten per vier bits die in de *ABSG* gestoken worden. Statistisch gezien is de mogelijkheid tot een lege buffer dan 2^{-89} zodat er geen constante bitstroom meer is. We kunnen ervan uitgaan dat deze kans onbestaande is.

In de omgekeerde situatie, wanneer de volledig gevulde buffer een bit van de *ABSG* aangeboden krijgt, zal deze bit wegvallen.
Het *ABSG*-mechanisme start wanneer de buffer leeg is en de initialisatiefase gedaan is. De sleutelstroom zal uit de buffer komen nadat ze voor de eerste maal volledig gevuld is.[17]
De twee komende figuren geven het algemene schema van het stroomcijfer en het coderingsmechanisme weer.



Figuur 7.2: Algemeen schema van DECIMv2



Figuur 7.3: DECIMv2 als coderingsmechanisme

[17] LANO, J., Wu, H., Preneel, B., *DECIMv2*, 2005, online, http://www.ecrypt.eu.org/stream/decim.html, 1 maart 2006.

7.2 Implementatie van DECIMv2

DECIMv2 is gemakkelijk op te splitsen in verschillende delen: de LFSR, de filter, de ABSG en de buffer, die stap voor stap geïmplementeerd worden.

7.2.1 LFSR

Er wordt een register van 192 bits gebruikt dat in een geklokt proces haar bits kan doorschuiven en de nieuwe bit kan opnemen. Dit register heeft de algemene naam *LFSR* gekregen. Door middel van een teller, *int_counter*, zal de LFSR, na het laden van de *key* en de *IV*, de nieuwe bit opnemen zoals in figuur 7.4. Deze bit wordt beïnvloed door *y*, de uitgang van de filter.

7.2.2 Filter

Deze filterfunctie is volledig combinatorisch en bestaat uit XOR-poorten en AND-poorten. Op basis van de tapbits zal deze filter de uitgang, *y*, genereren. (Cfr. figuur 7.5)

7.2.3 ABSG

Deze generator zal pas in werking kunnen treden nadat het *absg_en*-signaal gegeven is. In een flipflop, die *save* heet, wordt de bit, afkomstig van de filterfunctie, opgeslagen. Op deze manier vergelijken we deze flipflop constant met de uitgang *y*. Zodra de ingang van *ABSG* gelijk is aan de waarde in de flipflop zal het *absg_out*-signaal, afhankelijk van de *absg_counter*, ofwel *save* ofwel het inverse signaal van *save* worden.
Ook zal *write_en* een hoog signaal uitsturen, om aan te geven dat de uitgang van de ABSG klaar is om in de buffer geladen te worden. De uitgewerkte schematische werking wordt weergegeven in figuur 7.6.

7.2.4 Buffer

De uitgangsbuffer van 32 bits zorgt voor een constante uitgaande bitstroom. Per vier klokcycli stuurt de buffer een bit naar buiten. Er is een teller, *delay*, geïmplementeerd om het juiste aantal klokcycli bij te houden. De uitgang, *keystream*, zal om de vier klokcycli bijgevuld worden en de uitgang van het stroomcijfer vormen.

Vervolgens heeft de buffer twee klokprocessen: één proces om de leeswijzer aan te sturen en een ander proces om de schrijfwijzer aan te sturen. Door middel van deze twee wijzers zal de buffer weten welke bit naar buiten geschreven mag worden en op welke plaats een bit, afkomstig van de *ABSG*, geplaatst mag worden. De tellers, *rd_pointer* en *wr_pointer*, slaan dit op. De signalen *read_en* en *write_en* geven aan wanneer de tellers verhoogd mogen worden.

Wanneer *rd_pointer* groter is dan *wr_pointer*, kan een nieuwe bit in de buffer geschreven worden. De uitgang van de *ABSG* wordt dan opgeslagen wanneer *write_en* hoog is.

Als de twee tellers even groot zijn, zal de bit, komende van *ABSG*, niet geschreven worden. Als de *wr_pointer* groter wordt dan de *read_pointer*, zal er een *error* naar buiten gestuurd worden, wat te vermijden is.

Op deze manier wordt er de sleutelstroom, *keystream*, gegenereerd aan de uitgang, wat in figuur 7.7 beschreven wordt. Met behulp van een bewerking met een XOR-poort van de sleutelstroom en de boodschap *m* krijgt de gebruiker van het stroomcijfer de cijfertekst *c*.

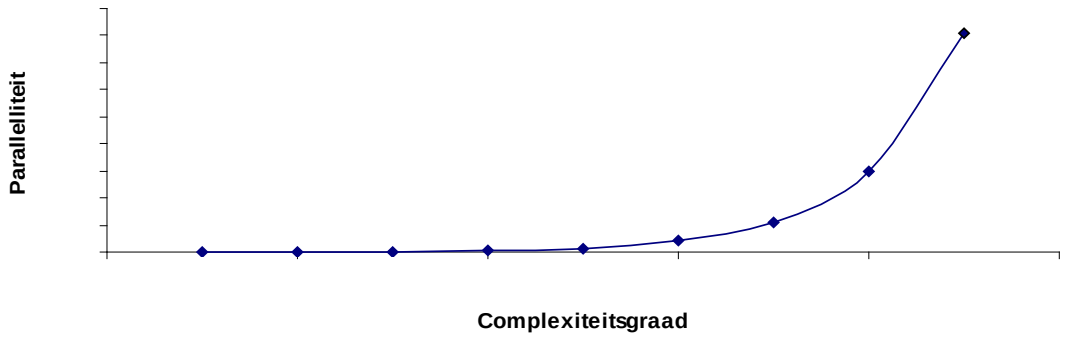
7.2.5 Testresultaten van DECIMv2

Hier geven we de resultaten van onze versie van het stroomcijfer DECIMv2.
Het "*place and route*"–rapport op FPGA geeft de volgende resultaten:

	DECIMv2
# gebruikte slices in FPGA (totaal 33088)	249
minimale klokperiode (op 50% duty cycle) (ns)	5,332
aantal klokpulsen nodig om 1 bit uit te sturen	1/4

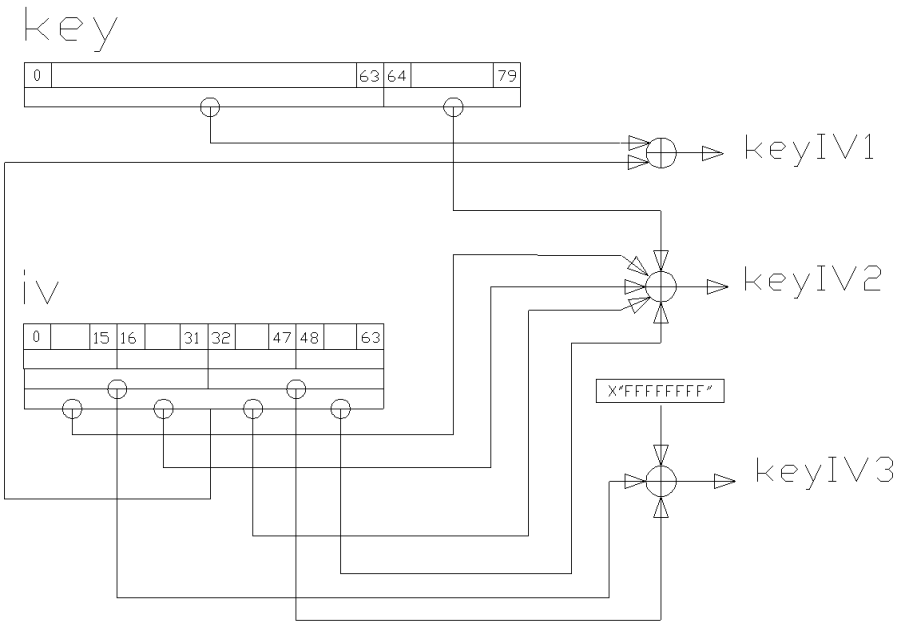
Tabel 7.1: Testresultaten DECIMv2

DECIMv2 vertoont een evenwichtige balans tussen routing (57.8%) en logica (42.2%) bij het kritische pad. Er wordt geen gebruik gemaakt van grote vergelijkingsblokken en dergelijke. De logica komt hoofdzakelijk neer op XOR-operaties en het gebruik van flip-flops, wat de kleine oppervlakte en de kleine maximale vertraging verklaart.
Een belangrijk nadeel van DECIMv2 is dat het slechts om de vier klokcycli een bit naar buiten stuurt omwille van de bufferwerking. DECIMv2 kan echter geparallelliseerd worden tot er een radix van 0,5 en 1 bit per klokpuls bereikt wordt. Dit hebben we in onze implementatie niet doorgevoerd, maar gaat theoretisch als volgt: men voorziet twee feedbackbits van de LFSR. Deze worden gebruikt om aan de twee filters, de originele en een exacte kopie, ingangsbits aan te leggen. De ABSG voorziet men van twee ingangen om de resultaten van de filters tegelijkertijd op te vangen. De LFSR moet dan op dubbele kloksnelheid aangestuurd worden. De radix heeft uiteindelijk een kwadratisch verband ten opzichte van de complexiteit van de implementatie, zoals aangegeven in de grafiek 7.1.

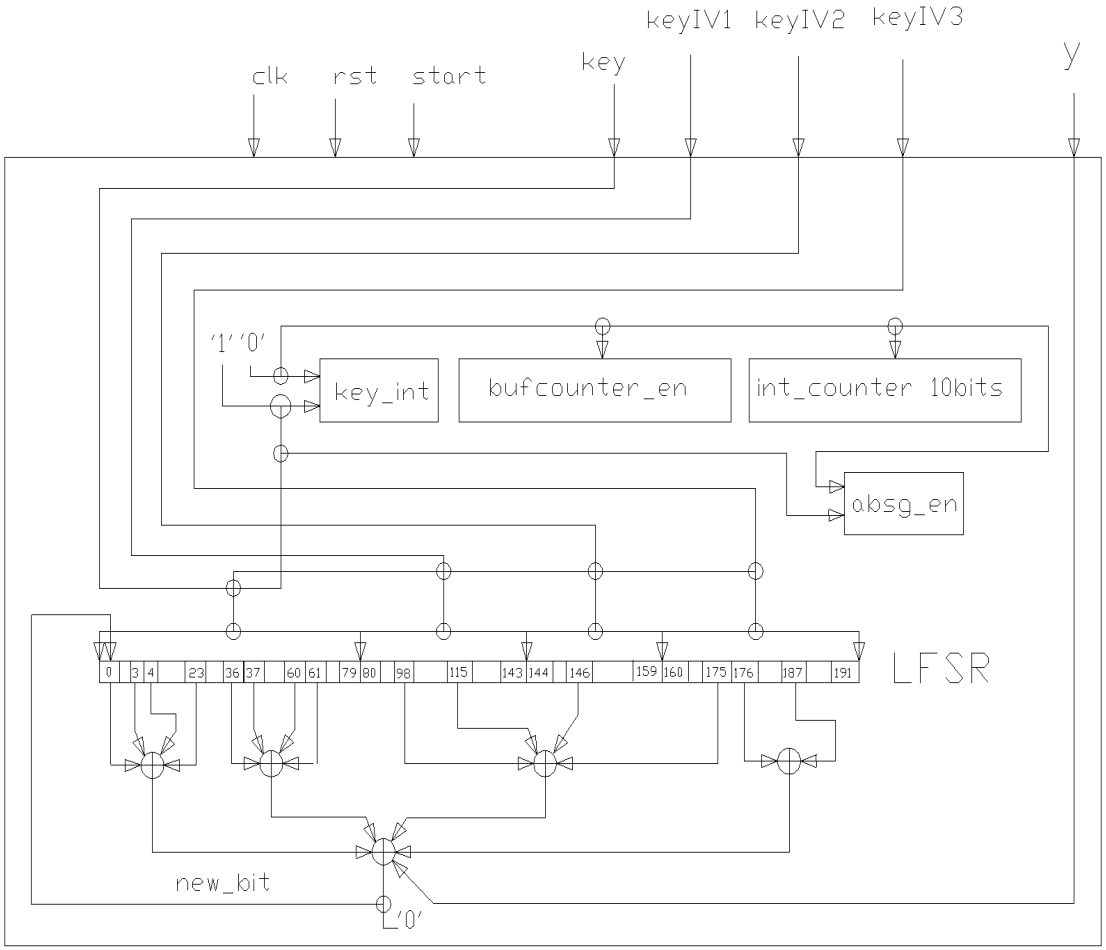


Grafiek 7.1 : DECIMv2 → parallelliteit versus complexiteit

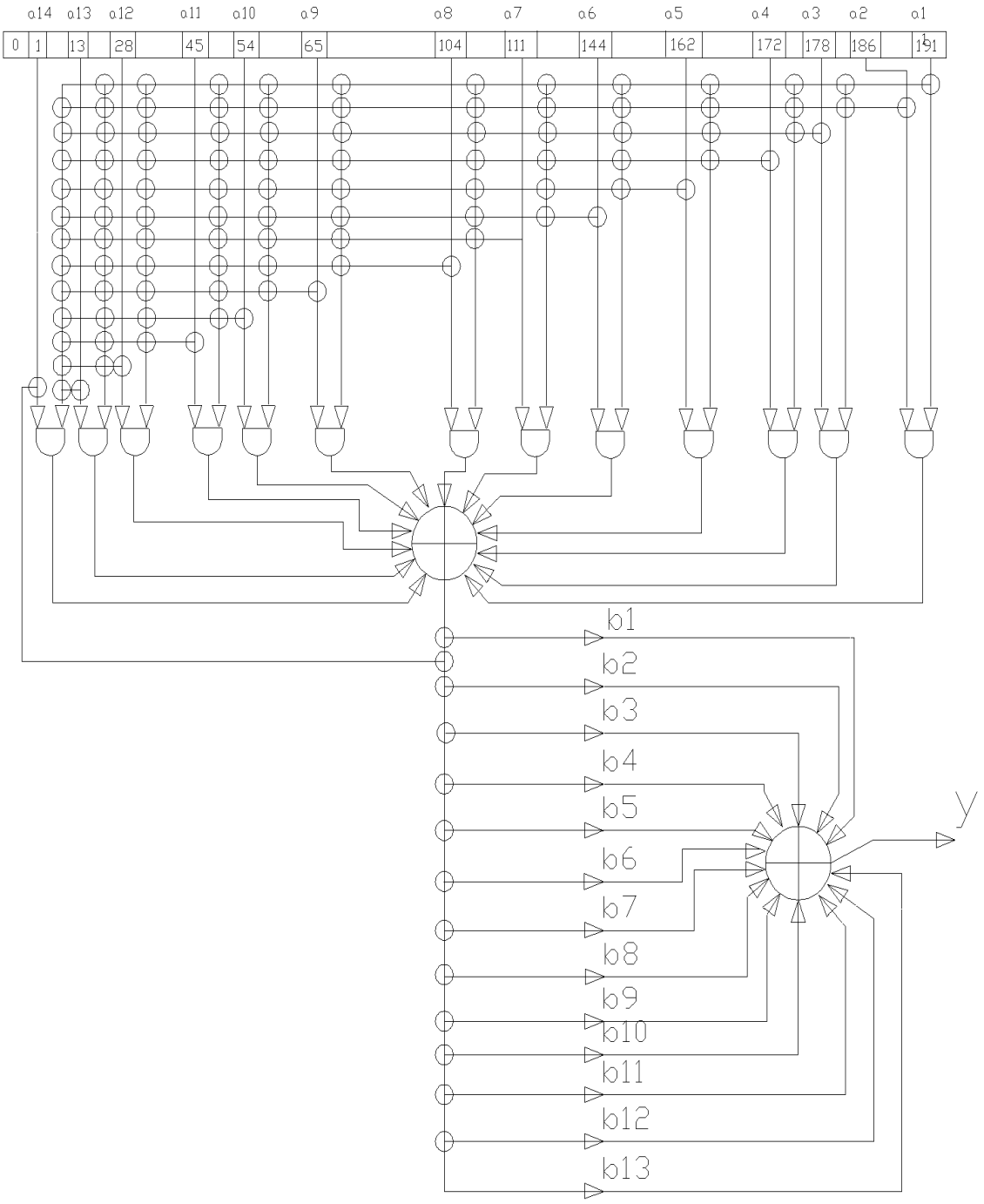
7.3 Schematische voorstelling van DECIMv2



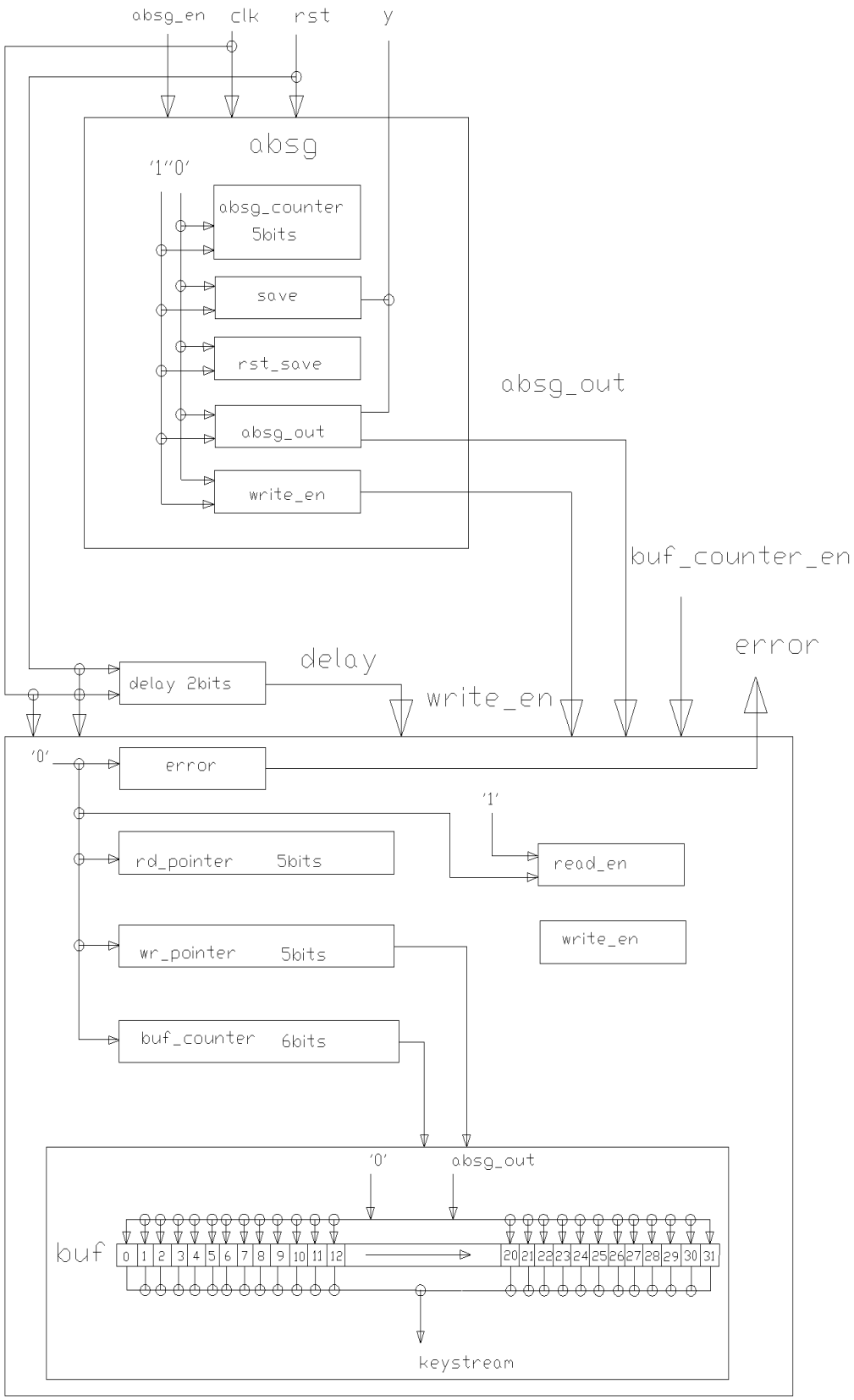
Figuur 7.4: Initialisatie van de LFSR



Figuur 7.5: LFSR



Figuur 7.6: Filterfunctie



Figuur 7.7: ABSG en buffer

HOOFDSTUK 8

Stroomcijfer Salsa20

8.1 Theorie achter Salsa20

Dit stroomcijfer is een *hashfunctie* dat een ingang en een uitgang heeft van 64 bytes. Salsa20 kan een boodschap coderen door de sleutel en het bloknummer doorheen de *hashfunctie* te laten lopen. Vervolgens worden deze uitgang en de boodschap met een XOR-poort omgezet tot de cijfertekst.

In wat nu volgt, zullen we regelmatig spreken over een "woord". Een woord is een beschrijving van een verzameling van elementen: $\{0,1,\ldots,2^{32}-1\}$. Woorden zullen we vaak schrijven in hexadecimale vorm: ze hebben "0x" in het begin van de notatie. Wanneer we een som van twee woorden *u* en *v* maken, bedoelen we: $(u + v \bmod) 2^{32}$. De optelling van twee woorden van 32 bit krijgt geen dragerbit mee.
Een voorbeeld van deze schrijfwijze:

$$0xc0a8787e + 0x9fd1161d = 0x60798e9b.$$
$$0xc0a8787e \oplus 0x9fd1161d = 0x5f796e63.$$
$$0xc0a8787e <<< 5 = 0x150f0fd8.$$

8.1.1 Werking van Salsa20

Om Salsa20 goed te kunnen begrijpen, zullen we elke functie waarvan deze *hashfunctie* gebruik maakt in detail uitleggen: de *kwartrondfunctie*, de *rijrondfunctie*, de *kolomrondfunctie*, de *dubbelrondfuncite*, de *hashfunctie* zelf en de combinerende functies achteraf.

8.1.1.1 Kwartrondfunctie

De *kwartrondfunctie* heeft een ingang en een uitgang met een lengte van vier woorden en werkt als volgt:

Stel $y = (y_0,y_1,y_2,y_3)$, dan krijgen we als resultaat: *kwartrondfunctie*$(y) = (z_0,z_1,z_2,z_3)$ waar

$$z_1 = y_1 \oplus ((y_0 + y_3) <<< 7),$$
$$z_2 = y_2 \oplus ((z_1 + y_0) <<< 9),$$
$$z_3 = y_3 \oplus ((z_2 + z_1) <<< 13),$$
$$z_0 = y_0 \oplus ((z_3 + z_2) <<< 18).$$

Opmerking
Men hoort rekening te houden dat eerst z_1 gemaakt wordt op basis van y_1, dan z_2 op basis van y_2, dan z_3 op basis van y_3 en als laatste z_0 op basis van y_0. Elke verandering is onomkeerbaar, zoals de functie zelf.

8.1.1.2 Rijrondfunctie

De *rijrondfunctie* heeft een ingang en een uitgang met een lengte van zestien woorden. Ze wordt op deze manier opgebouwd:

Stel $y = (y_0,y_1,y_2,\ldots y_{15})$, dan krijgen we als resultaat: *kwartrondfunctie*$(y) = (z_0,z_1,z_2,\ldots z_{15})$ waar

$$(z_0,z_1,z_2,z_3) = \textit{kwartrondfunctie}(y_0,y_1,y_2,y_3),$$
$$(z_5,z_6,z_7,z_4) = \textit{kwartrondfunctie}(y_5,y_6,y_7,y_4),$$
$$(z_{10},z_{11},z_8,z_9) = \textit{kwartrondfunctie}(y_{10},y_{11},y_8,y_9),$$
$$(z_{15},z_{12},z_{13},z_{14}) = \textit{kwartrondfunctie}(y_{15},y_{12},y_{13},y_{14}).$$

Opmerking
We kunnen de ingang $(y_0,y_1,y_2,\ldots y_{15})$ als een vierkante matrix bekijken:

$$\begin{bmatrix} y_0 y_1 y_2 y_3 \\ y_4 y_5 y_6 y_7 \\ y_8 y_9 y_{10} y_{11} \\ y_{12} y_3 y_{14} y_{15} \end{bmatrix}$$

8.1.1.3 Kolomrondfunctie

Identiek aan de *rijrondfunctie*, heeft de *kolomrondfunctie* een ingang en een uitgang met een lengte van zestien woorden. De bouw ervan is hetzelfde:

Stel $x = (x_0,x_1,x_2,\ldots x_{15})$ dan krijgen we als resultaat: *kwartrondfunctie*$(x) = (y_0,y_1,y_2,\ldots y_{15})$ waar

$$(y_0,y_4,y_8,y_{12}) = \textit{kwartrondfunctie}(x_0,x_4,x_8,x_{12}),$$
$$(y_5,y_9,y_{13},y_1) = \textit{kwartrondfunctie}(x_5,x_9,x_{13},x_1),$$
$$(y_{10},y_{14},y_2,y_6) = \textit{kwartrondfunctie}(x_{10},x_{14},x_2,x_6),$$
$$(y_{15},y_3,y_7,y_{11}) = \textit{kwartrondfunctie}(x_{15},x_3,x_7,x_{11}).$$

Opmerking
We kunnen de ingang $(y_0,y_1,y_2,\ldots y_{15})$ als een vierkante matrix bekijken:

$$\begin{bmatrix} x_0 x_1 x_2 x_3 \\ x_4 x_5 x_6 x_7 \\ x_8 x_9 x_{10} x_{11} \\ x_{12} x_3 x_{14} x_{15} \end{bmatrix}$$

8.1.1.4 Dubbelrondfunctie

De *dubbelrondfunctie* is een *kolomrondfunctie* gevolgd door een *rijrondfunctie*. De grootte van de ingang en de uitgang is zestien woorden:

dubbelrondfunctie(x) = *rijrondfunctie*(*kolomrondfunctie*(x))

Opmerking
Deze functie past de kolommen en vervolgens de rijen van de ingangsmatrix parallel aan. Elk woord wordt dus tweemaal aangepast.

8.1.1.5 Littleendian -functie

De eenvoudige *littleendian-functie* converteert vier bytes naar één woord:

Als $b = (b_0,b_1,b_2,b_3)$ is, dan is de *littleendian*-funtie(b) = $b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$.

Enkele voorbeelden ter verduidelijking:

littleendian-functie(86,75,30,9)= 0x091e4b56.
littleendian-functie(255,255,255,250)= 0xfaffffff.

Opmerking
De ingangen van de voorbeelden worden als decimale waarden geschreven en de functie kan niet omgekeerd worden.

8.1.1.6 Hashfunctie Salsa20

De Salsa20 *hashfunctie* maakt gebruik van de reeds beschreven functies. Zoals we eerder aanhaalden zijn de ingang en de uitgang 64 bytes groot:

Salsa20(x) = x + *dubbelrondfunctie*^{10}(x), waarbij elke 4 bytes in één woord omzet worden door de *littleendian*-functie.

We bekomen dan de waarden:

Stel $x = (x[0],x[1],x[2],\ldots,x[63])$ dan wordt

x_0 = *littleendian-functie*(x[0],x[1],x[2],x[3]),
x_1 = *littleendian-functie*(x[4],x[5],x[6],x[7]),
↓
x_{14} = *littleendian-functie*(x[56],x[57],x[58],x[59]),
x_{15} = *littleendian-functie*(x[60],x[61],x[62],x[63]).

Stel dat $(z_0,z_1,z_2,\ldots z_{15})$ = *dubbelrondfunctie*$^{10}(x_0,x_1,x_2,\ldots x_{15})$is, dan is *Salsa20*($x$) de samenvoeging van alle z-waarden in de inverse versie van de *littleendian*-functie.

$$\begin{array}{c} \textit{littleendian-functie}^{-1}(z_0 + x_0), \\ \textit{littleendian-functie}^{-1}(z_1 + x_1), \\ \downarrow \\ \textit{littleendian-functie}^{-1}(z_{14} + x_{14}), \\ \textit{littleendian-functie}^{-1}(z_{15} + x_{15}). \end{array}$$

Door het inverteren van de *littleendian-funcite* maken we van één woord terug vier bytes. Dit wordt zestien keer herhaald opdat we 64 bytes aan de uitgang krijgen.

De *hashfunctie* heeft een uitbreiding: de *uitbreidingsfunctie*:

Aan de ingang van de *hashfunctie* worden enkele constanten τ_0, τ_1, τ_2 en τ_3, telkens één byte groot, samen met een sleutel *k* en een waarde *n* aangelegd. Voor de sleutel *k* kunnen we een lengte van 32 of 16 bytes kiezen terwijl *n* altijd 16 bytes groot is.

Het aanleggen van de waarden komt erop neer dat:

Stel dat τ_0 = (101,120,112,97), τ_1 = (110,100,32,49), τ_2 = (54,45,98,121) en τ_3 = (116,101,32,107) en *k* en *n* = 16 bytes, dan is $Salsa20_k(n) = Salsa20_k(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3)$.

Op deze manier wordt de ingang van 48 bytes naar 64 bytes aan de uitgang geëxpandeerd: de *uitbreidingsfunctie*.

8.1.2 Stroomcijfer Salsa20

Salsa20 maakt als stroomcijfer gebruik van de uitbreiding van de *hashfunctie*. De sleutel *k* (32 bytes groot) en een uniek boodschapnummer (16 bytes groot) zijn de variabelen en de beschrijving gaat als volgt:

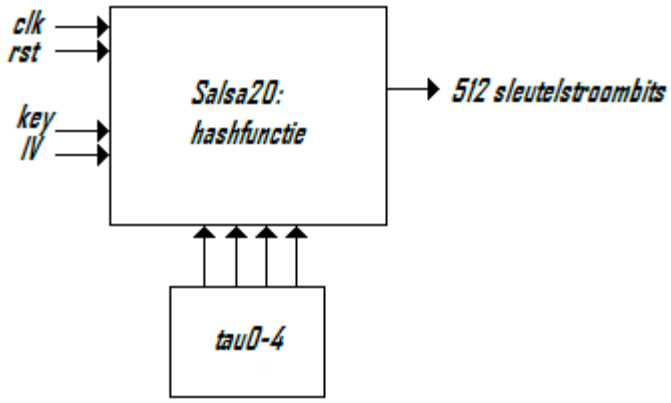
$$Salsa20_k(v,\underline{0}),\ Salsa20_k(v,\underline{1}),\ Salsa20_k(v,\underline{2}),\ldots,\ Salsa20_k(v,\underline{2\text{^}64\ -1})$$
$$Salsa20_k(v) \oplus (m[0],m[1],m[2],\ldots,m[l\text{-}1]) = (c[0],c[1],c[2],\ldots c[l\text{-}1])$$

$$\text{en } c[i] = m[i] \oplus Sasla20_k(v,i/64)[i \bmod 64]$$

Hier is $\underline{i}$ acht bytes groot: er kunnen per cyclus tot 2^{64}-1 bitblokken gecodeerd en gedecodeerd worden. De formule $Salsa20_k(v) \oplus m$ betekent dat het coderen van een boodschap *m* door het stroomcijfer $Salsa20_k(v)$ dezelfde lengte heeft als *m*.[18]

[18] LANO, J., J.Bernstein, D., *Salsa20,* 2005, on line, http://www.ecrypt.eu.org/stream/salsa20.html, 22 maart 2006.

Figuur 8.1 toont Salsa20 als stroomcijfer: de figuur laat ons duidelijk zien dat het stroomcijfer gebaseerd is op de hashfunctie.



Figuur 8.1: Algemeen schema van Salsa20

8.2 Implementatie van Salsa20

Salsa20 bestaat uit verscheidene functies die steeds gebruik maken van elkaar. We zijn logischerwijs op het laagste niveau, de *kwartrondfunctie,* beginnen met implementeren.

8.2.1 Kwartrondfunctie

Dit bouwblok heeft vier ingangen en vier uitgangen met een grootte van 4 bytes of 1 woord zoals in figuur 8.2. Ze worden respectievelijk (*y0, y1, y2, y3*) en (*z0, z1, z2, z3*) genoemd.

Intern zijn er registers aangemaakt om tussenliggende waarden bij te houden. Om een som zoals (*y0 + y3*) bij te kunnen houden hebben we (*zreg0,…,zreg4)* aangemaakt. De volgende som maakt gebruik van de uitkomst die de volledige bewerking gedaan heeft. Hiervoor voegen we nog vier registers (*z_out0,…,z_out3)* toe die de bewerking van de XOR-operatie bijhouden. De uitkomst kan nu zowel hergebruikt als naar buiten gestuurd worden.

Dit bouwblok werkt volledig combinatorisch, wat de *kwartrondfunctie* heel snel uitvoerbaar maakt.

8.2.2 Rijrondfunctie

De *rijrondfunctie* gebruikt de *kwartrondfunctie* viermaal parallel. Daarom worden er vier bouwblokken van de *kwartrondfunctie* gemaakt zodat er 64 bytes of 16 woorden op hetzelfde moment geproduceerd worden. (Cfr. figuur 8.3)

8.2.3 Kolomrondfunctie

De *kolomrondfunctie* werkt op dezelfde manier als de *rijrondfunctie.* Het enige verschil komt erop neer dat de ingangen en uitgangen anders doorgekoppeld zijn naar de *kwartrondfunctie* toe. Dit is eveneens een volledig combinatorisch blok, zoals in figuur 8.4 getoond wordt.
We bekijken de ingangen van de *rij- en kolomrondfunctie* als een vierkante matrix. We hebben ervoor gezorgd dat de *rijrondfunctie* de elementen van deze matrix rij per rij mixt via de *kwartrondfunctie*. De *kolomrondfunctie* zal dan, zoals de naam al verklapt, de elementen van de matrix kolom per kolom door elkaar schudden. Elk element van de matrix doorloopt de *kwartrondfunctie* tweemaal.

8.2.4 Dubbelrondfunctie

De *dubbelrondfunctie* maakt gebruikt van *de rij- en kolomrondfunctie* om 16 woorden om te vormen.
Eerst wordt de *kolomrondfunctie* uitgevoerd met als ingangen de 16 woorden. Zodra de uitgangen gevormd zijn, zullen deze de ingangen worden voor de *rijrondfunctie*.
De uitgangen van deze laatste functie worden onmiddellijk doorgekoppeld met de uitgangen van de *dubbelrondfunctie*, wat weergegeven wordt in figuur 8.5.

8.2.5 Littleendian-functie

Dit bouwblok voegt een aantal bytes samen tot één woord en staat los van alle vorige functies. Aan de ingang leggen we viermaal 4 bytes ofwel 1 woord aan.
Deze ingangen hebben de naam (*b0,…b3)* en worden rechtstreeks doorgekoppeld met de uitgang *b_out*. De doorkoppeling gebeurt via *de littleendian*-functie in figuur 8.6.

8.2.6 Hashfunctie Salsa20

De *hashfunctie* zoals in figuur 8.7 en 8.8 maakt gebruik van 16 bouwblokken die de *littleendian*-functie uitvoeren en één bouwblok dat de *dubbelrondfunctie* uitvoert. De formule van de *hashfunctie* komt neer op de som van de uitgangen van de 16 bouwblokken van de *littleendian*-functie met de uitkomst van diezelfde uitgangen, nadat deze tien keer de *dubbelrondfunctie* doorlopen hebben.

De registers (*hashreg0,…, hashreg15)* hebben een specifieke geheugenfunctie: wanneer de *dubbelrondfunctie* éénmaal uitgevoerd is, moeten deze uitgangen dienen als ingangen bij de volgende klokpuls. Dit wordt tien keer herhaald met behulp van een interne teller.
Intussen bewaren we de resultaten die uit de *littleendian-functie* komen in de registers (*lit_x0,…, lit_x15*) totdat de *dubbelrondfunctie* afgelopen is.
Vervolgens worden de uiteindelijke uitgangen van de *dubbelrondfunctie* opgeteld met de registers (*lit_x0,…, lit_x15)*. Dit resultaat, dat één byte groot is, wordt parallel naar buiten gestuurd.

Voor de uitbreiding van deze functie stoppen we de constanten *t*, een sleutel *k* en een waarde *n* in de *hashfunctie*.(Cfr. figuur 8.9) De *hashfunctie* is het eerste bouwblok is dat een kloksignaal nodig heeft.

8.2.7 Stroomcijfer Salsa20

Salsa20 gebruikt de *hashfunctie* als motor voor de codering en decodering. De sleutel *k* en een unieke boodschapnummer, *IV,* zijn de ingangen. Het unieke boodschapnummer wordt samengevoegd met een teller om de 2^{64}-1 blokken veilig te coderen zoals in figuur 8.10. De teller, *encryption_counter*, wordt steeds met 1 verhoogd en met het unieke boodschapnummer samengevoegd tot het *encryption_iv* dat zal dienen als ingang van de uitgebreide *hashfunctie*.
Op deze manier kan Salsa20 64 bytes of 16 woorden van de boodschap per klokpuls coderen of decoderen.

8.2.8 Testresultaten van Salsa20

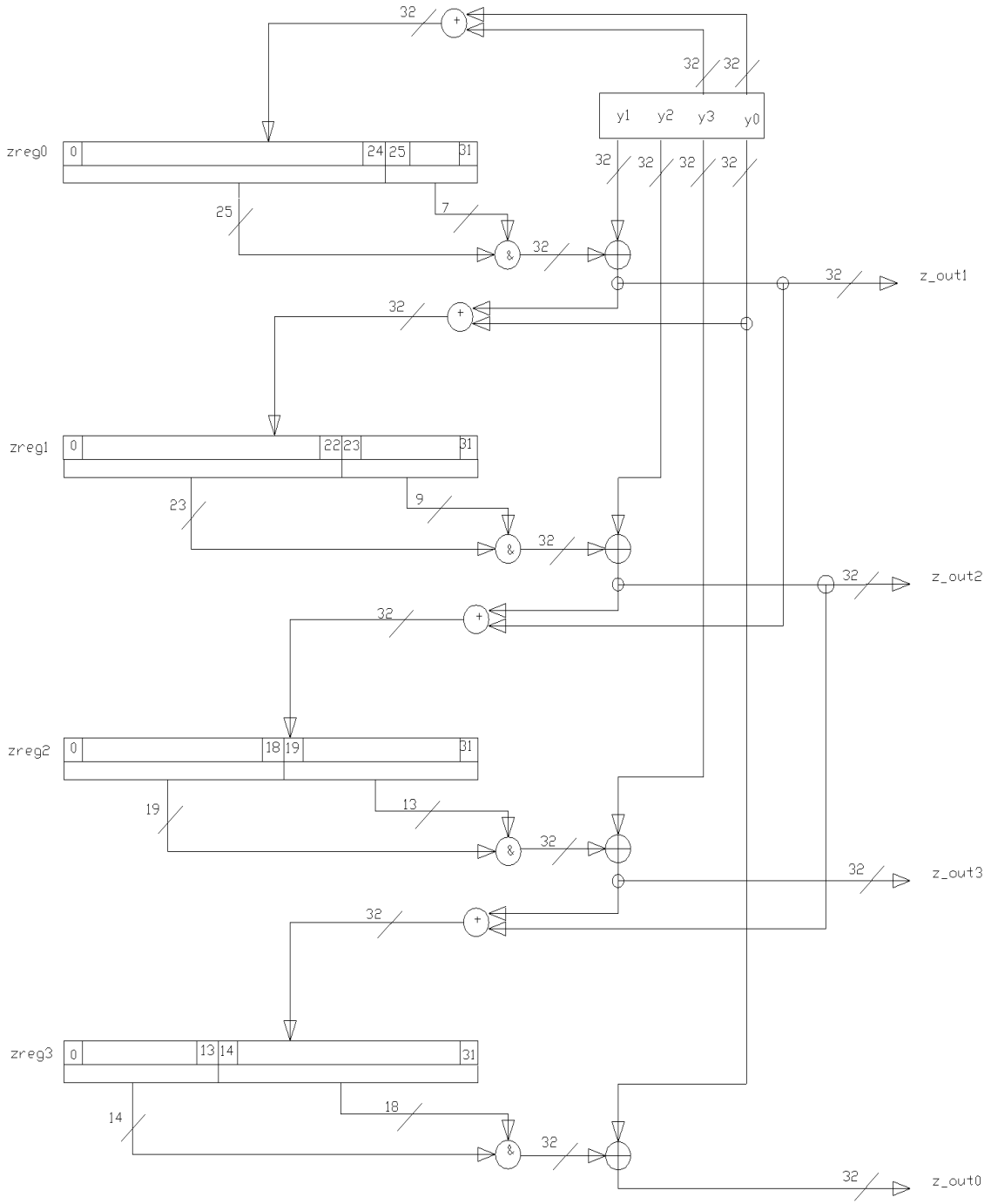
Hier geven we de resultaten van onze versie van het stroomcijfer Salsa20.
Het "*place and route*"-rapport op FPGA geeft de volgende resultaten:

	Salsa20
# gebruikte slices in FPGA (totaal 33088)	1744
minimale klokperiode (op 50% duty cycle) (ns)	35,492
aantal klokpulsen nodig om 1 bit uit te sturen	1/(51,2)

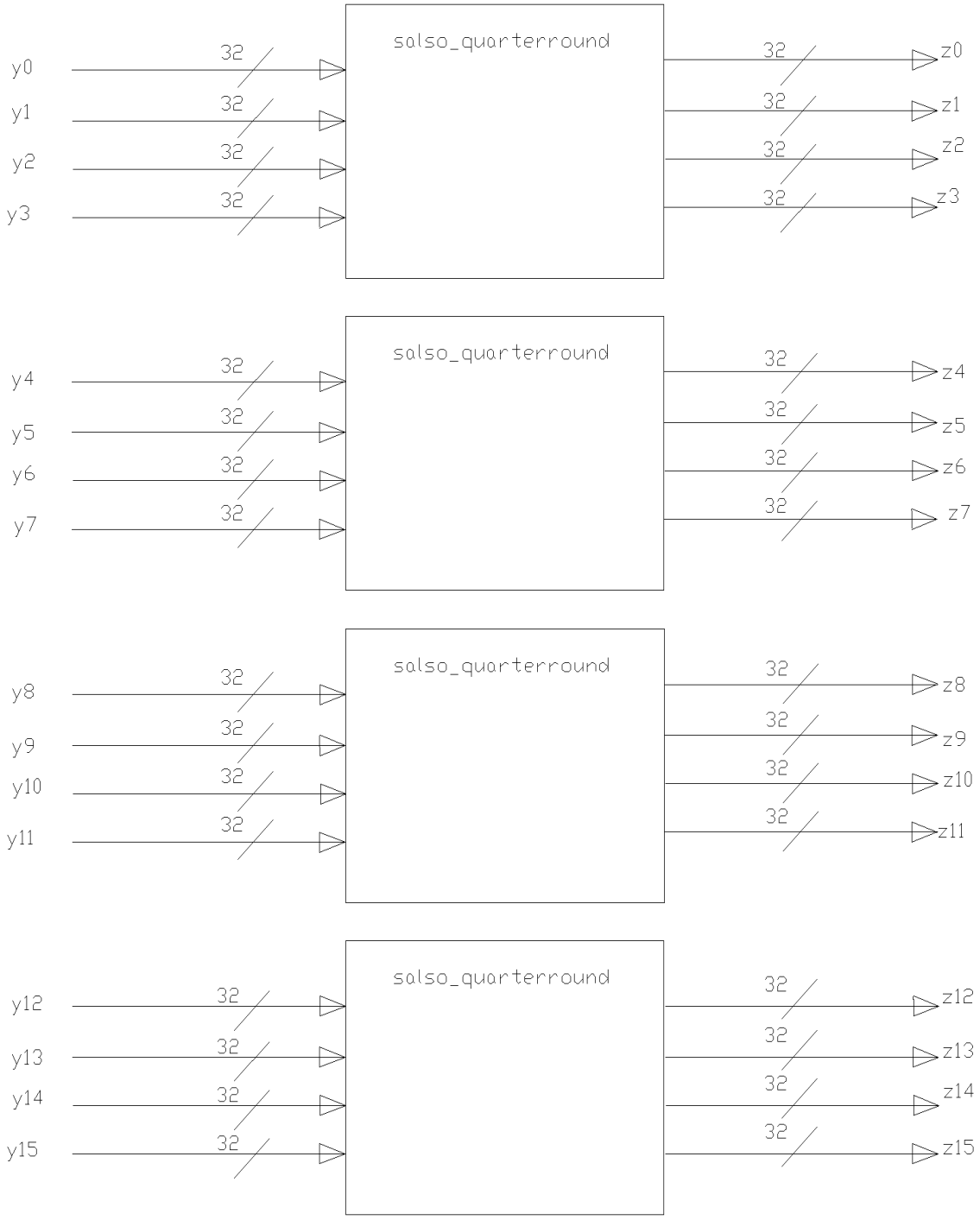
Tabel 8.1: Testresultaten Salsa20

Na een allereerste blik op Salsa20 wordt er duidelijk een hiërarchische structuur opgebouwd. Wanneer men dit in hardware realiseert, vertrekt men vanuit een kleine basisfunctie die constant hergebruikt en uitgebreid wordt. Dit herhaalt zich in de opbouw van blok tot blok totdat we het uiteindelijke stroomcijfer krijgen: de vertraging speelt zich dus meer af in de logica (60.8%) dan in de routing (39.2%). De maximale vertraging blijkt zeer groot te zijn omdat elk niveau van de structuur moet wachten totdat de uitgang van een lager niveau klaar is. De *dubbelrondfunctie* moet op haar beurt tien keer uitgevoerd worden en veroorzaakt hoofdzakelijk de vertraging. We krijgen door de werking van de d*ubbelrondfunctie* per tien klokperiodes 512 bits naar buiten. Bijgevolg komen er bij onze implementatie per klokperiode 51,2 bits uit Salsa20.

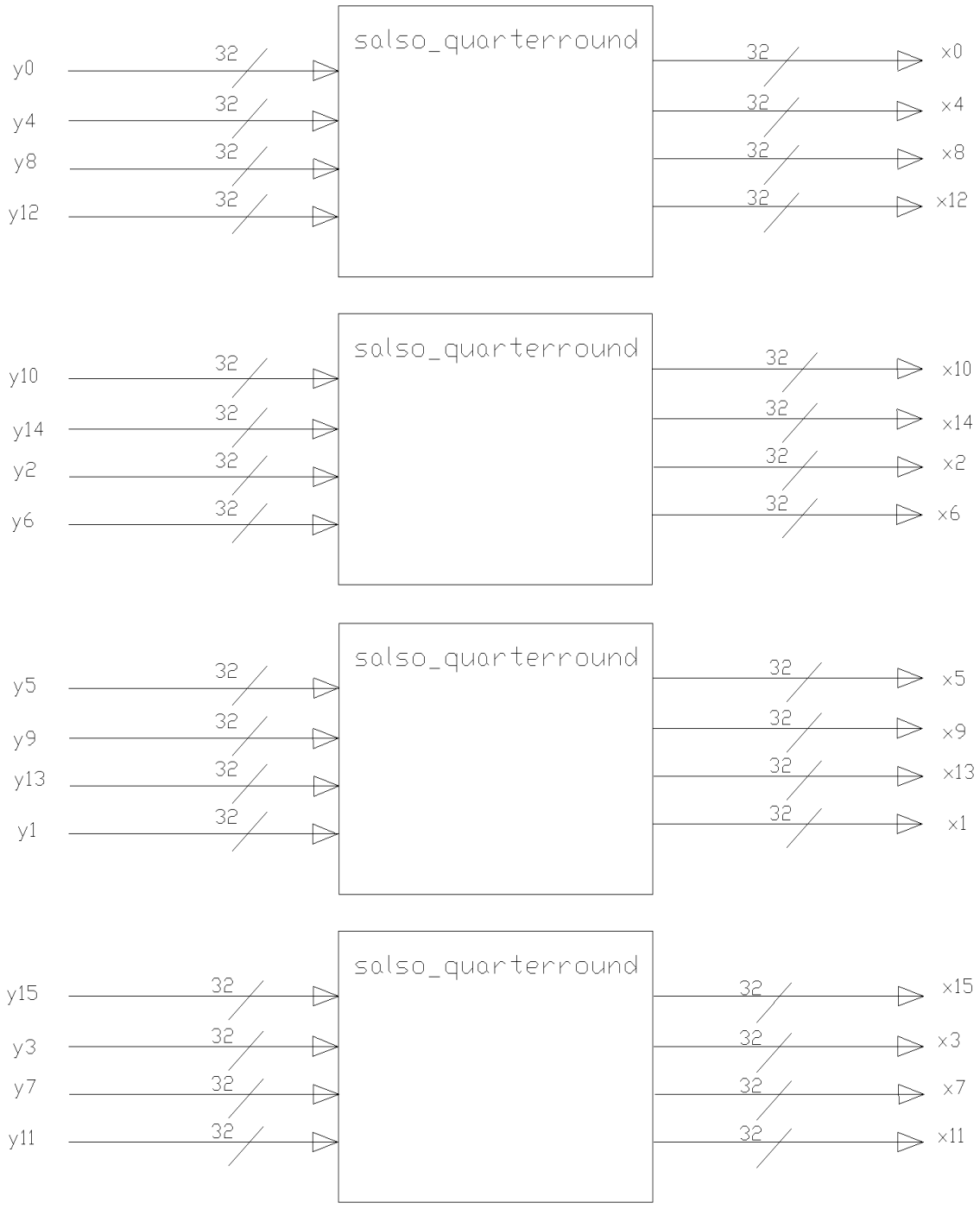
8.3 Schematische voorstelling van Salsa20



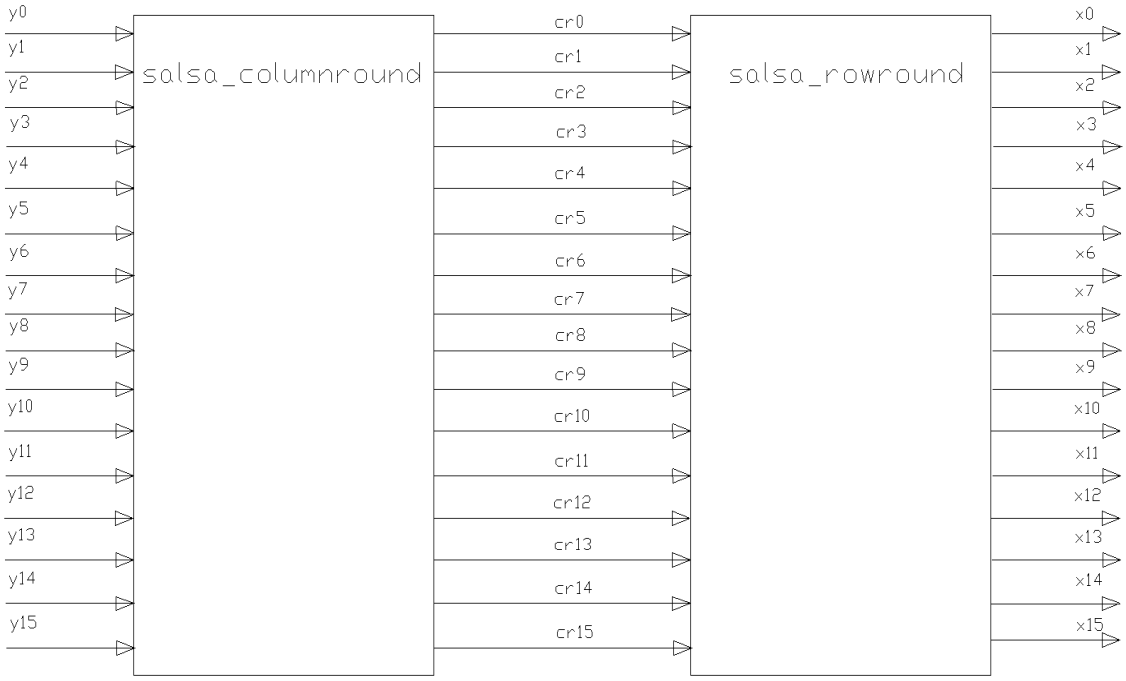
Figuur 8.2: Kwartrondfunctie



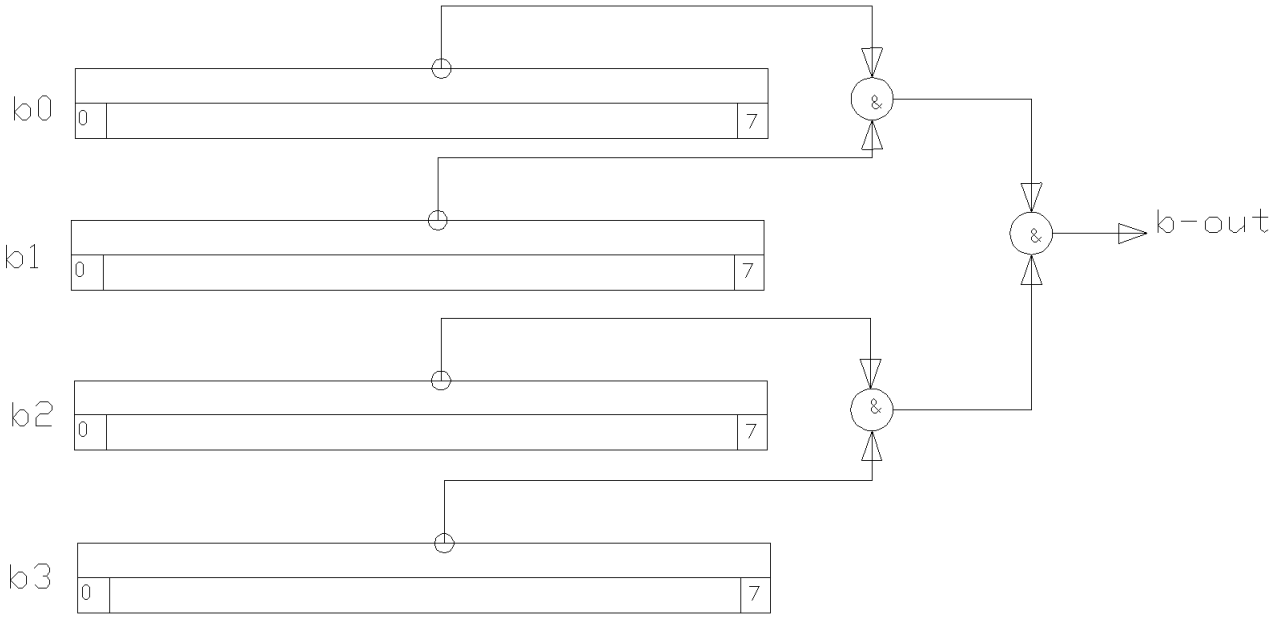
Figuur 8.3: Rijrondfunctie



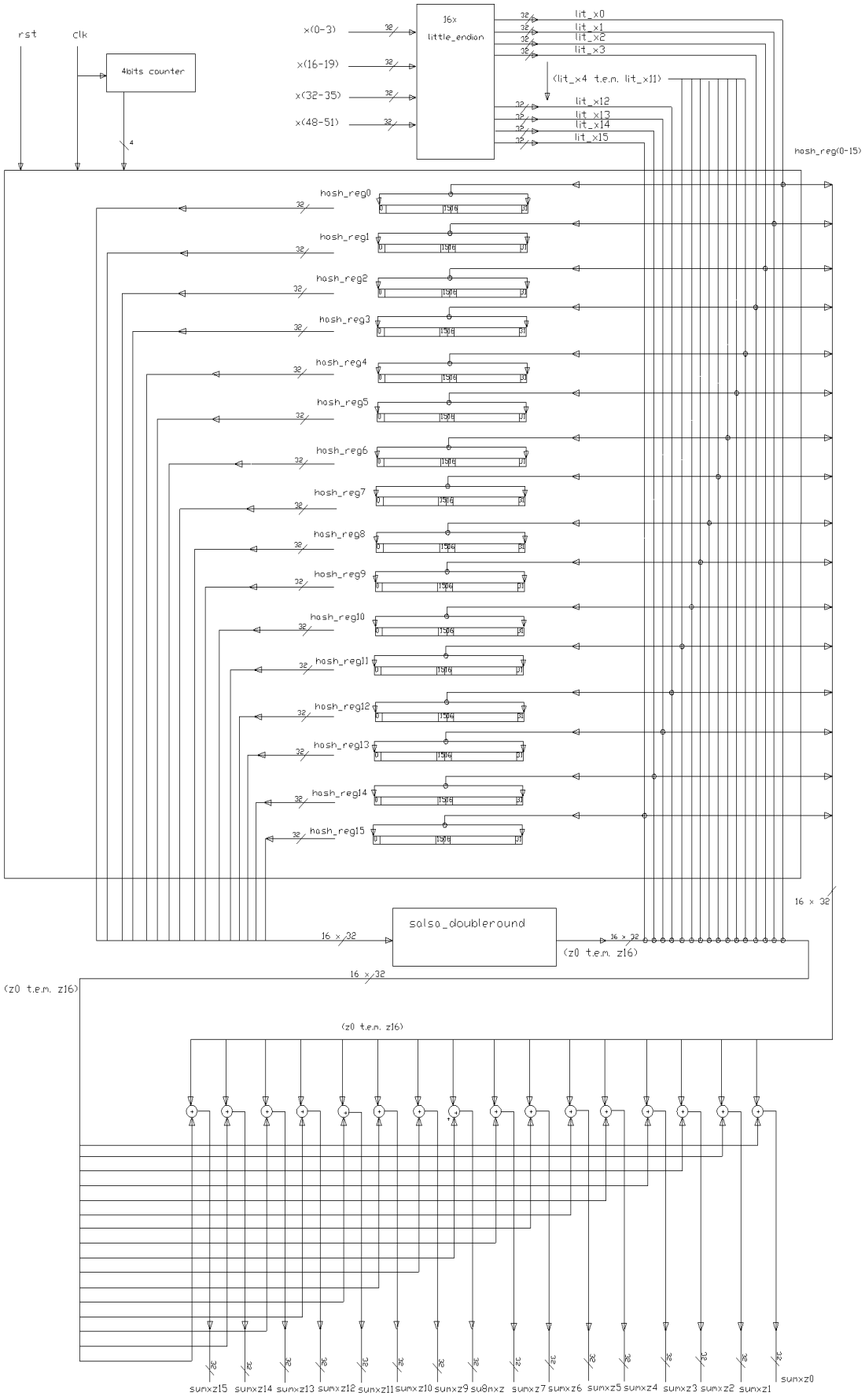
Figuur 8.4: Kolomrondfunctie



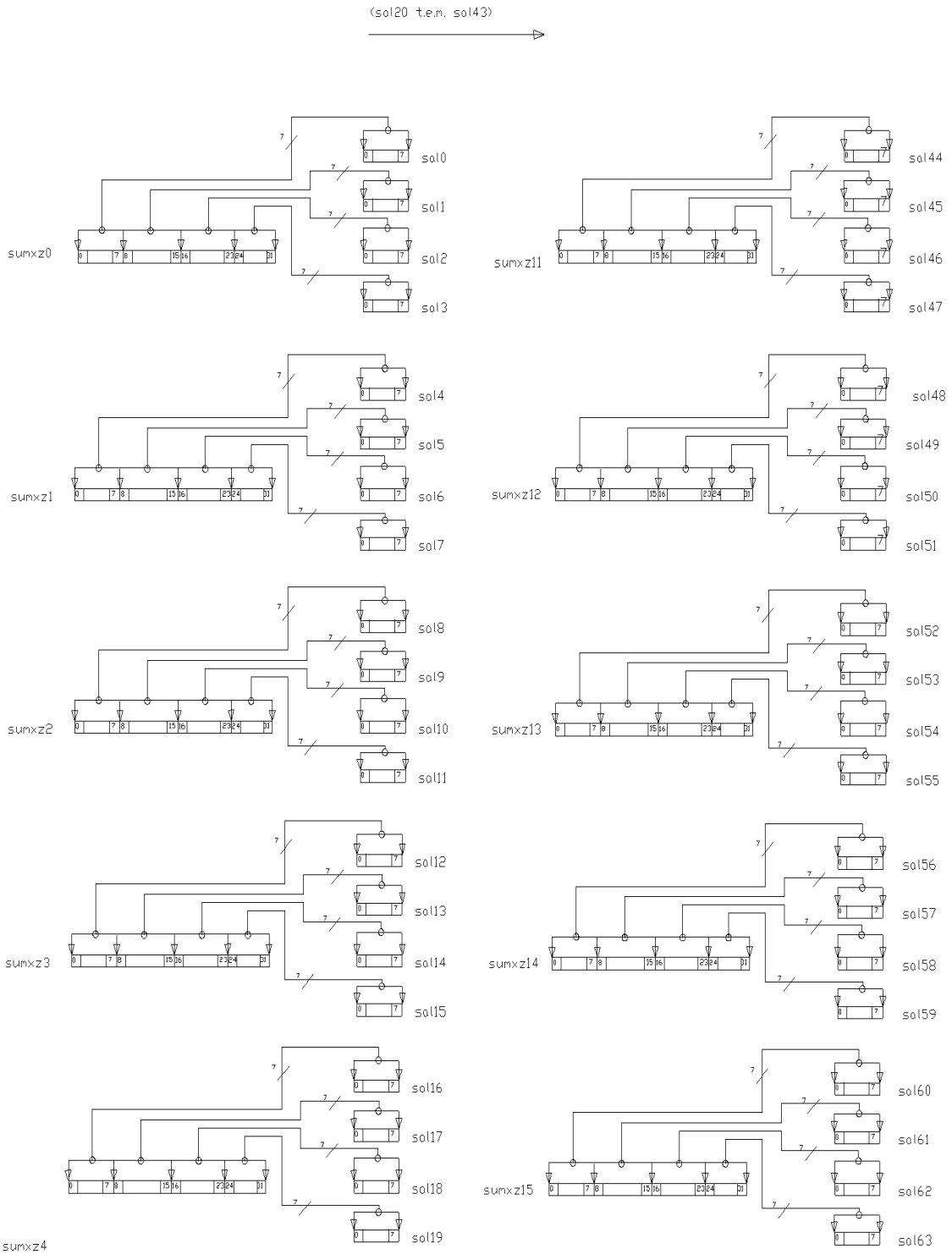
Figuur 8.5: Dubbelrondfunctie



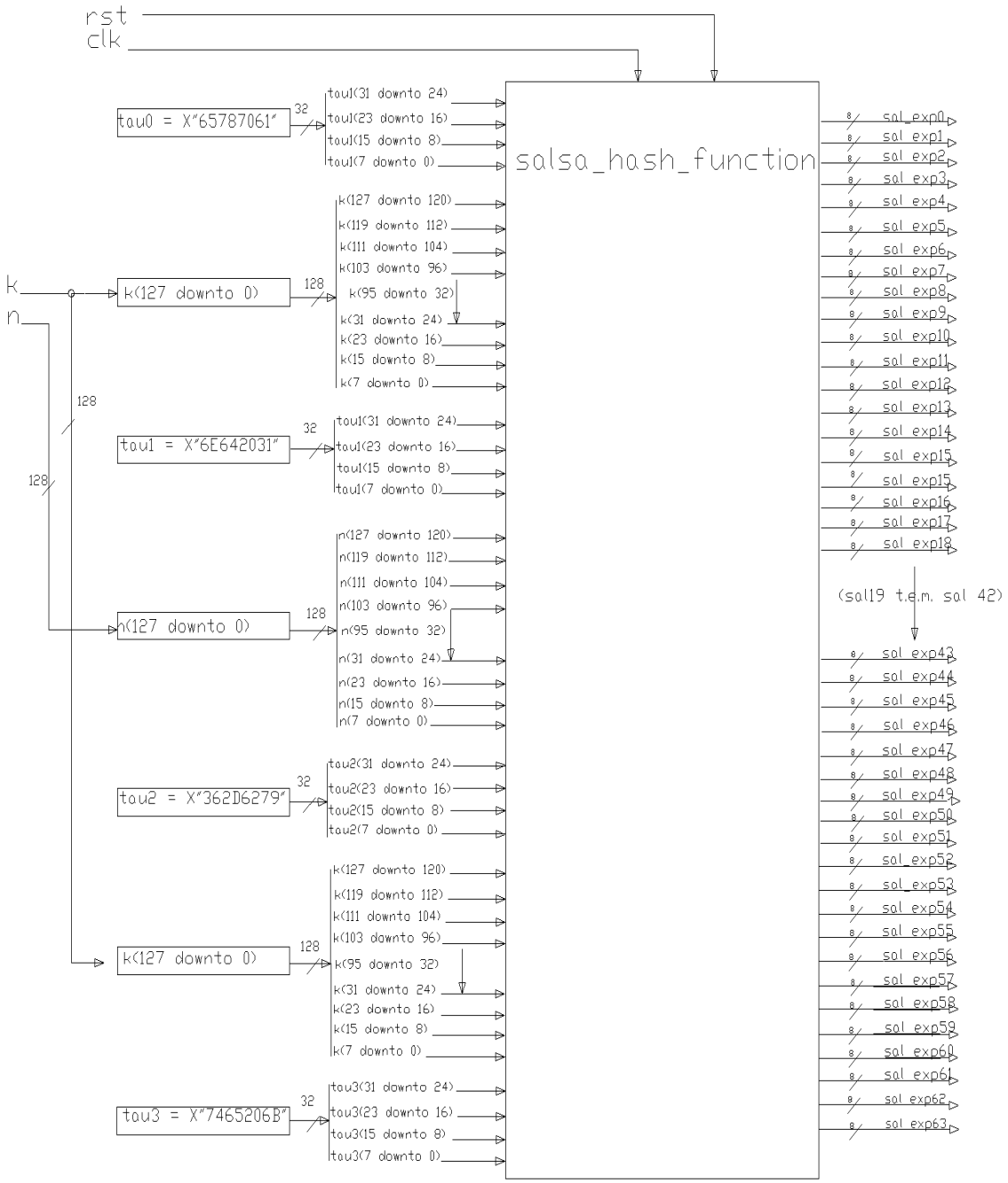
Figuur 8.6: Littleendian–functie



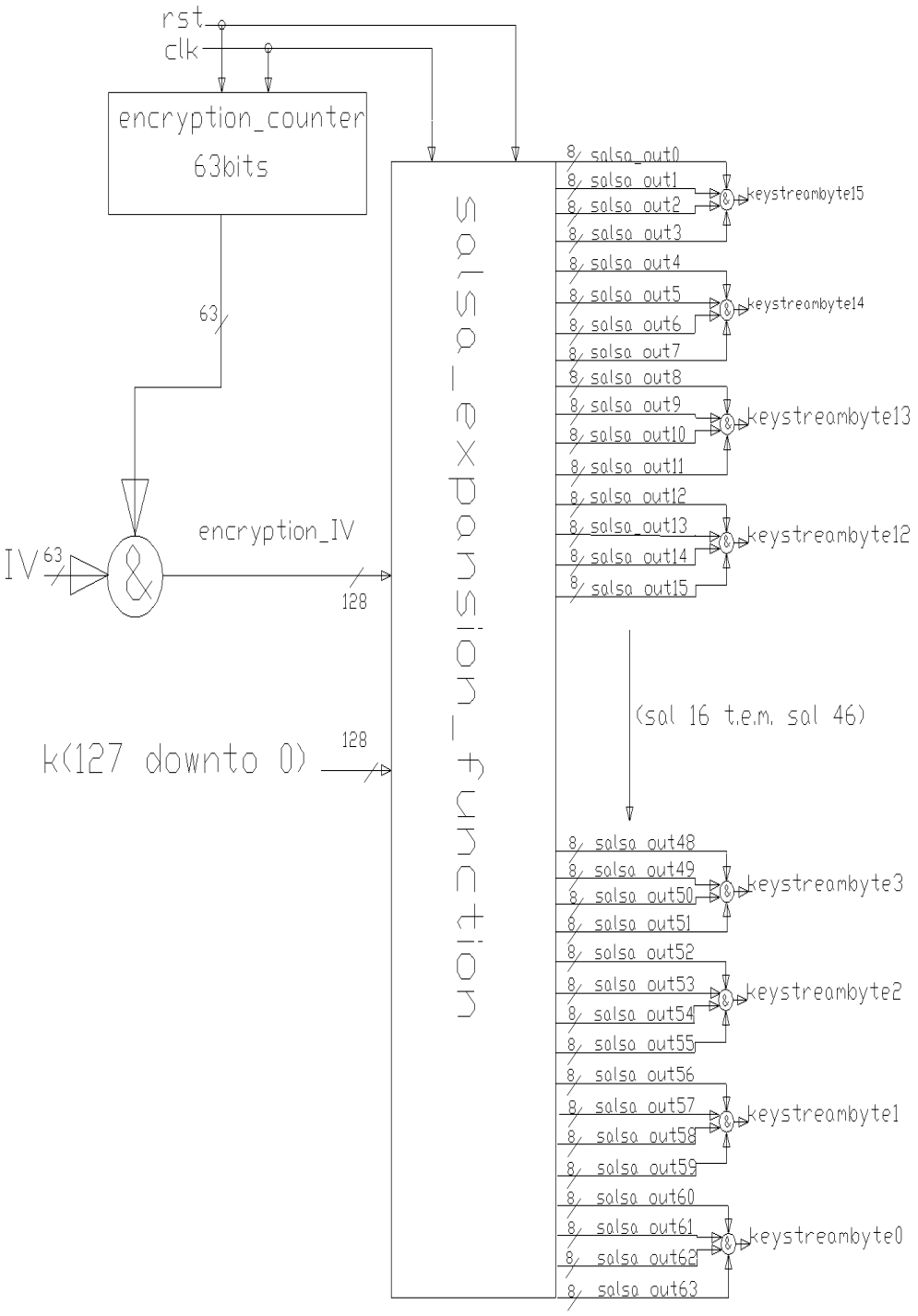
Figuur 8.7: Hashfunctie deel 1



Figuur 8.8: Hashfunctie deel 2



Figuur 8.9: Uitbreidingsfunctie



Figuur 8.10: Coderingsfunctie

HOOFDSTUK 9

Besluitvorming

9.1 Vergelijking van de stroomcijfers op FPGA

Om uit de verschillende stroomcijfers de beste te kunnen filteren, houden we rekening met bepaalde factoren: de snelheid en de oppervlakte van het stroomcijfer. We beginnen met een overzichtstabel 9.1 van onze behaalde resultaten op FPGA (Tabel 9.1).

	A5/1v1	A5/1v2	Rabbit	DECIMv2	Salsa20
# gebruikte slices in FPGA (totaal 33088)	465	461	2314	249	1744
minimale klokperiode (ns)	3,4	3,2	1,4	5,3	35,5
aantal klokpulsen nodig om 1 bit uit te sturen	3	1	1/128	4	1/51,2

	Trivium1	Trivium8	Trivium16	Trivium32	Trivium64
# gebruikte slices in FPGA (totaal 33088)	323	285	303	355	443
minimale klokperiode (ns)	8,5	5,9	6,6	5,9	6,7
aantal klokpulsen nodig om 1 bit uit te sturen	1	1/8	1/16	1/32	1/64

Tabel 9.1: Samenvatting van de resultaten op FPGA

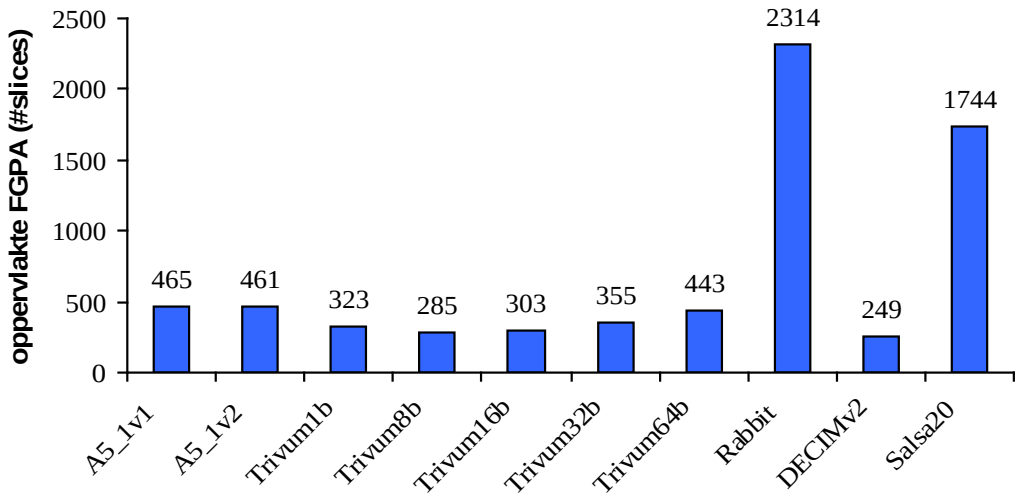
9.1.1 Oppervlakte

Allereerst vergelijken we de stroomcijfers qua oppervlakte in de FPGA. De oppervlakte benoemen we met het aantal gebruikte slices in de FPGA. Na de synthese en de place&route bekwamen we de resultaten in Tabel 9.2.

stroomcijfer	Oppervlakte (#slices op FPGA)
A5/1v1	465
A5/1v2	461
Trivum1b	323
Trivum8b	285
Trivum16b	303
Trivum32b	355
Trivum64b	443
Rabbit	2314
DECIMv2	249
Salsa20	1744

Tabel 9.2: FPGA → oppervlakte

Door het aantal ingenomen slices in de FPGA te tellen, kunnen we nagaan hoeveel oppervlakte het stroomcijfer gebruikt. De door ons gebruikte FPGA heeft een totaal van 33088 slices ter beschikking. De resultaten worden gevisualiseerd in Grafiek 9.1.



Grafiek 9.1: FPGA → oppervlakte

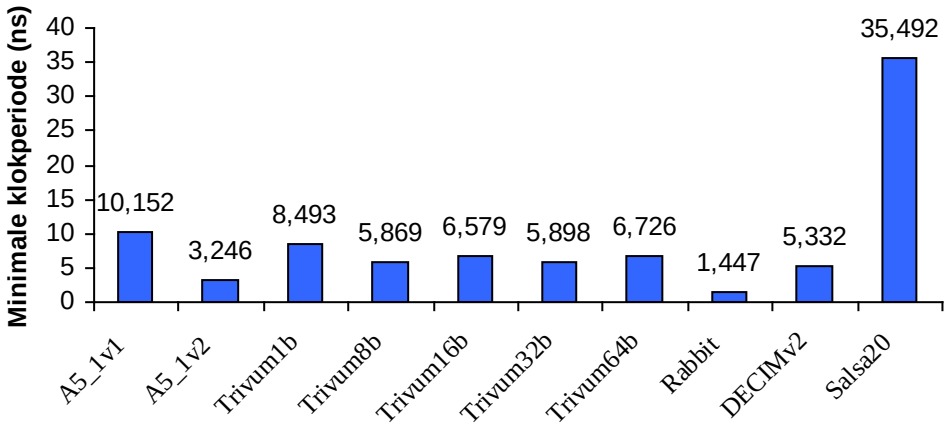
Het valt op dat Rabbit veel slices nodig heeft in vergelijking met de overige stroomcijfers. Het grootste cijfer gebruikt iets minder dan 7% van het totale aantal slices in de FPGA. Om deze reden kan Rabbit volgens ons niet in aanmerking komen als geschikt stroomcijfer in de eSTREAM-competitie. Dit geldt ook voor Salsa20, hetzij in mindere mate. We merken verder op dat de implementaties van DECIMv2 en Trivium veel minder slices in beslag nemen: ze zijn kleiner dan de twee implementaties van A5/1 en qua oppervlakte trekt DECIMv2 aan het langste eind. Dit weegt echter niet op tegen de flexibiliteit van Trivium.

9.1.2 Snelheid

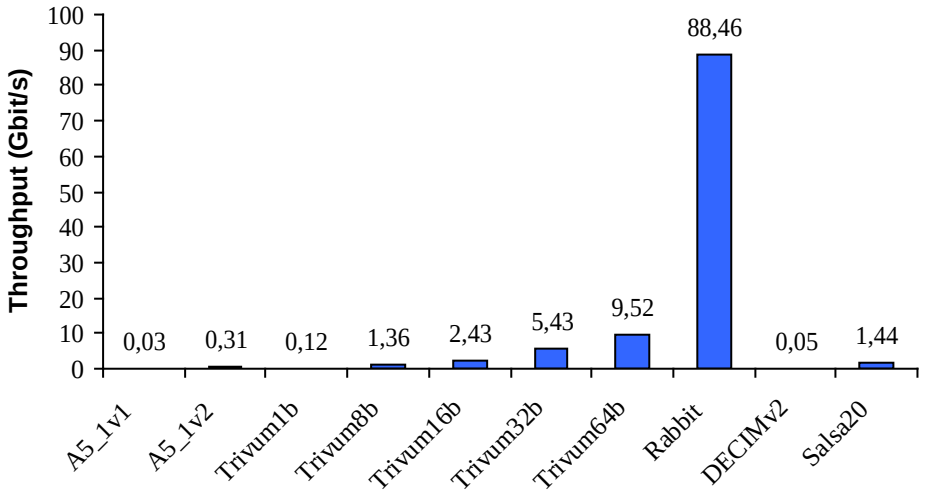
Een tweede belangrijke eigenschap van een stroomcijfer is haar maximale throughput. Op basis van de minimale klokperiode berekenen we het aantal gigabits die per seconde gegenereerd kunnen worden. De resultaten per stroomcijfer worden samengevat in Tabel 9.3 en gevisualiseerd in Figuur 9.2 en 9.3.

stroomcijfers	Gbit/s	Minimale klokperiode (ns)
A5_1v1	0,03	10,152
A5_1v2	0,31	3,246
Trivum1b	0,12	8,493
Trivum8b	1,36	5,869
Trivum16b	2,43	6,579
Trivum32b	5,43	5,898
Trivum64b	9,52	6,726
Rabbit	88,46	1,447
DECIMv2	0,05	5,332
Salsa20	1,44	35,492

Tabel 9.3: FPGA → throughput en minimale klokperiode



Grafiek 9.2: FPGA → minimale klokperiode



Grafiek 9.3: FPGA → throughput

De meest opvallende snelheden zijn die van Rabbit en die van DECIMv2:
Rabbit is enorm snel, wat te wijten is aan de vele combinatorische bouwblokken. DECIMv2 is langs de andere kant zeer traag. De verklaring is eenvoudig: de buffer van DECIMv2 stuurt slechts om de vier klokperiodes één bit naar buiten. We kunnen slechts gedurende een kwart van de tijd een boodschap coderen of decoderen.
De verschillende versies van Trivium vertonen een behoorlijke snelheid in vergelijking met de overige stroomcijfers, waar A5/1 in haar beide versies te traag blijkt.

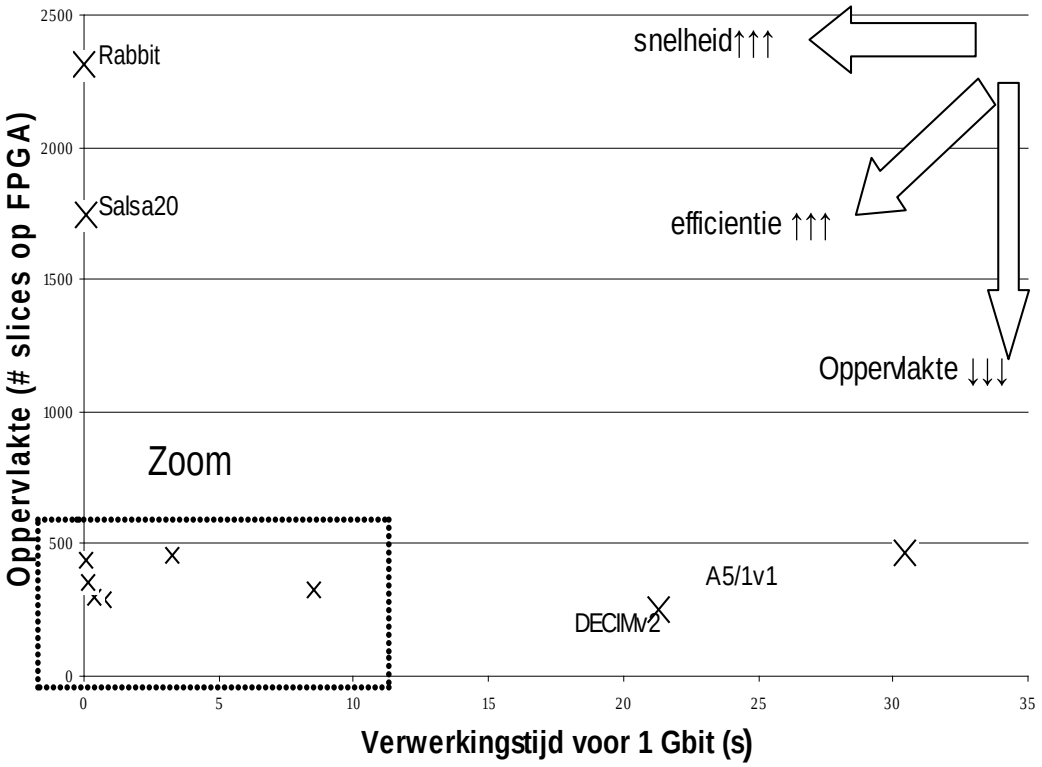
9.1.3 Oppervlakte versus snelheid

De trade-off tussen oppervlakte en snelheid wordt gevisualiseerd in Grafiek 9.4. We kiezen Trivium64b als het beste stroomcijfer uit onze geïmplementeerde reeks. Met 443 slices op de FPGA en een snelheid van 9,5 Gbits/s vertoont Trivium64b het beste evenwicht tussen minimale oppervlakte en maximale klokfrequentie.
DECIMv2 lijkt ons, wat de oppervlakteresultaten betreft, het beste. Aangezien haar snelheid de laagste is van alle stroomcijfers, vinden we het toch een ongeschikt stroomcijfer, tenzij in toepassingen waar hoge snelheden niet van cruciaal belang zijn.
Rabbit toont de tegenovergestelde balans: dit stroomcijfer kan men het beste gebruiken in toepassingen waarbij snelheid de voorkeur heeft op oppervlakte.
Salsa20 heeft volgens ons weinig toekomst als stroomcijfer in hardware omdat het geen enkele eigenschap heeft dat in het oog springt.

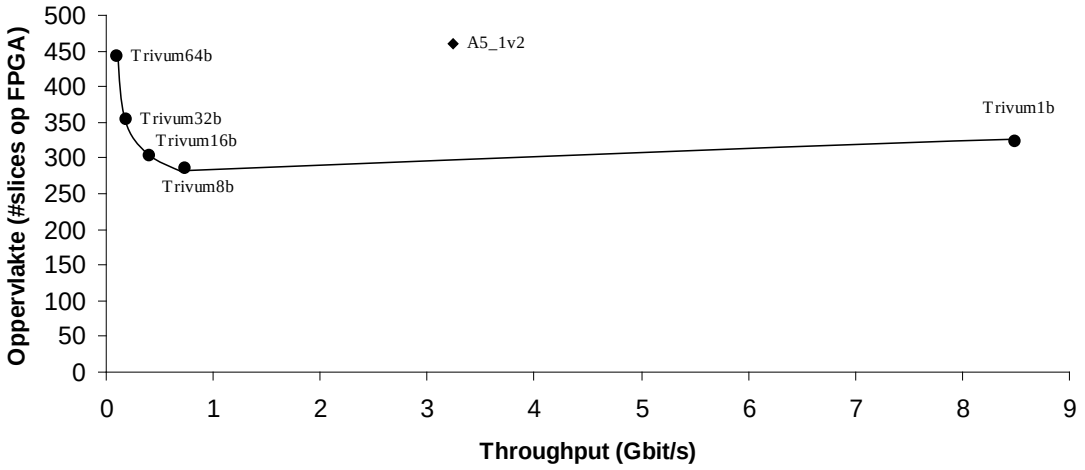
Wanneer we alle kandidaten uit onze reeks met Trivium in het algemeen, dan zien we dat de trade-off oppervlakte–snelheid van Trivium het beste is: Trivium is het efficiëntste, zowel qua oppervlakte en uitbreidmogelijkheden als gegenereerde gigabits per seconde.

stroomcijfers	verwerkingstijd 1Gbit (s)	Oppervlakte(#slices on FPGA)
A5_1v1	30,456	465
A5_1v2	3,246	461
Trivum1b	8,493	323
Trivum8b	0,734	285
Trivum16b	0,411	303
Trivum32b	0,184	355
Trivum64b	0,105	443
Rabbit	0,011	2314
DECIMv2	21,328	249
Salsa20	0,069	1744

Tabel 9.4 FPGA→ verwerkingstijd 1 Gbit en oppervlakte

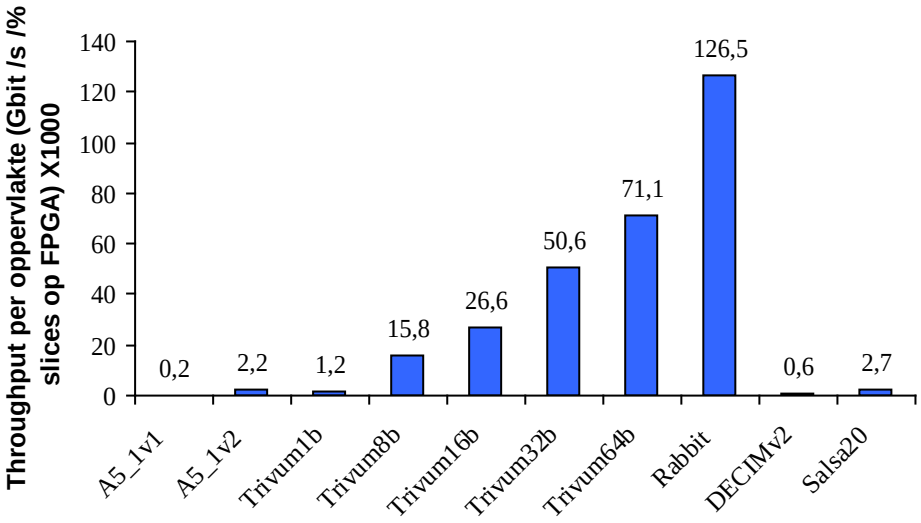


Grafiek 9.4: FPGA → verwerkingstijd versus oppervlakte



Grafiek 9.5: ZOOM: verwerkingstijd versus oppervlakte

Grafiek 9.5 toont het volgende verband: wanneer de radix van Trivium opgevoerd wordt, daalt de verwerkingstijd voor 1 Gbit. Dit wil zeggen dat er méér bits per klokpuls gegenereerd worden. Merk op dat er bij kleinere verwerkingstijden meer slices nodig zijn in de FPGA.



Grafiek 9.6: FPGA → throughput per oppervlakte

Uit grafiek 9.6 van de FPGA-resulaten kunnen we opmerken dat Rabbit het efficiënst is per slice op de FPGA: de kleine vertraging heft de grote ingenomen oppervlakte grotendeels op. We hebben een factor 1000 toegevoegd om grotere waardes te hebben.

Wanneer we nu alle voorgaande factoren in rekening brengen, kunnen we nog altijd met zekerheid besluiten dat Trivium64 het beste “allround”-stroomcijfer van onze reeks is: het is klein, snel en van alle markten thuis.

9.2 Vergelijking van de stroomcijfers op ASIC

9.2.1 Oppervlakte

We hebben onze stroomcijfers ook met Design Vision van Synopsys, een ASIC synthesetool, gesynthetiseerd. Op deze manier willen we het aantal gebruikte gates onderling vergelijken. De grootte van één gate op ASIC komt overeen met de oppervlakte van één NAND-poort. Dergelijke poort heeft een oppervlakte van 23,8 µm². De bekomen resultaten zijn de oppervlaktes van de stroomcijfers in μm^2. We hebben gebruik gemaakt van een 0,25µm CMOS-technologie en komen de volgende waarden in Tabel 9.4 uit:

A5/1 versie 1	114752 µm²
A5/1 versie 2	115600 µm²
Trivium 1 bit	96331 µm²
Trivium 8 bit	103767 µm²
Trivium 16 bit	112036 µm²
Trivium 32 bit	128391µm²
Trivium 64 bit	161227 µm²
Rabbit	1688544 µm²
DECIMv2	98097 µm²
Salsa20	409297 µm²

Tabel 9.5: ASIC → oppervlakte

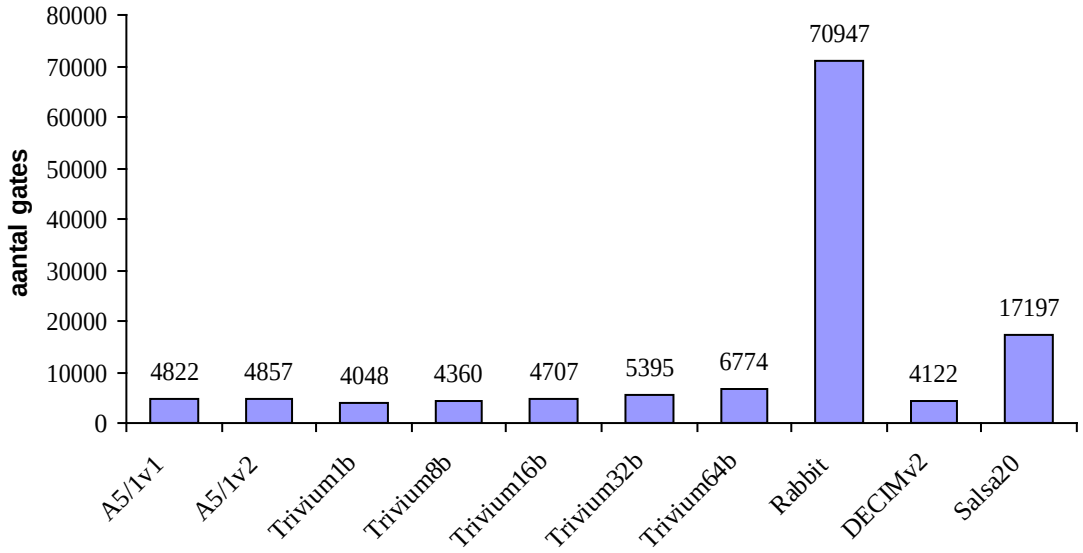
Door een kleine bewerking kunnen we vanuit de oppervlakte in μm^2 het aantal gebruikte gates berekenen. Dit wordt weergegeven in Tabel 9.6.

(oppervlakte in ASIC [µm²])/(grootte van één NAND-poort [µm²]) = # gates)

A5/1 versie 1	4821,55
A5/1 versie 2	4857,16
Trivium 1 bit	4047,52
Trivium 8 bit	4359,99
Trivium 16 bit	4707,41
Trivium 32 bit	5394,58
Trivium 64 bit	6774,26
Rabbit	70947,23
DECIMv2	4121,77
Salsa20	17197,38

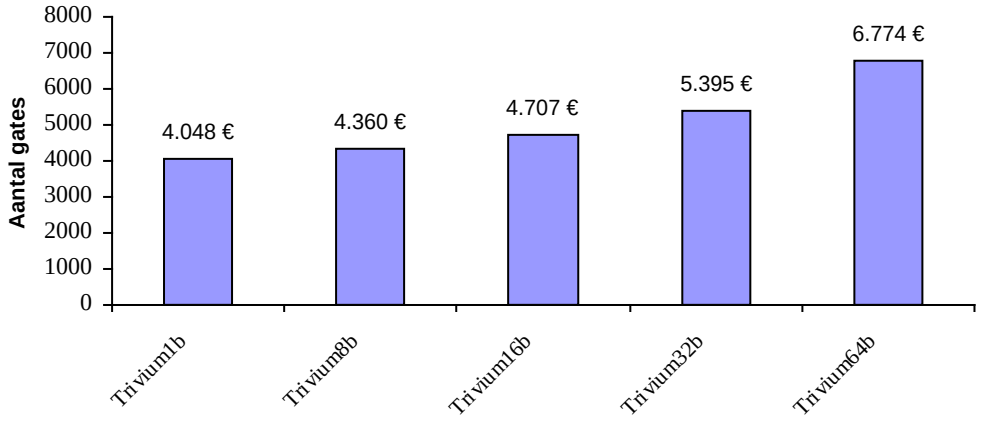
Tabel 9.6: ASIC → aantal gates

Grafiek 9.7 toont deze resultaten in grafiekvorm:



Grafiek 9.7: ASIC → aantal gates

Rabbit valt op door zijn groot aantal gates in vergelijking met de rest. Dit is te wijten aan de vele en grote vermenigvuldigers en optellers. Alleen al omwille van deze reden kunnen we nogmaals zeggen dat Rabbit geen goede toepassing is als stroomcijfer in hardware. De hoofdvereiste van het eSTREAM-project voor stroomcijfers in hardware is immers compactheid. Salsa20 is ook een uitschieter: dit stroomcijfer is net zoals Rabbit meer op softwarematig gebruik gericht dan dat het door de ontwerpers op een efficiënte manier in hardware uitgewerkt is.
Voor de overige stroomcijfers kunnen we besluiten dat ze stuk voor stuk weinig gates gebruiken, met Trivium1b als kleinste. DECIMv2 blijft goede resultaten boeken qua oppervlakte. Dit wordt getoond in Grafiek 9.8.



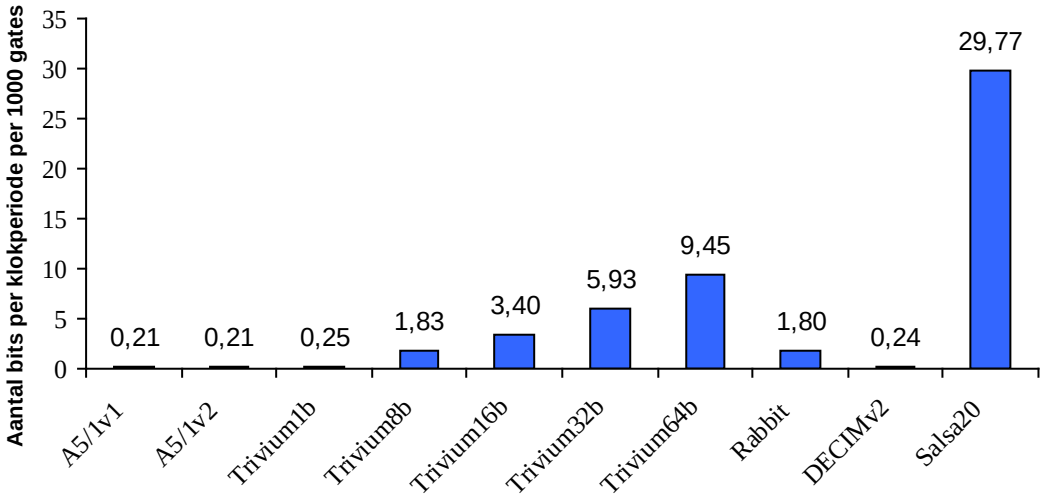
Grafiek 9.8: Trivium→ aantal gates in ASIC

9.2.2 Bits per 1000 gates

Na het berekenen van het aantal gates dat een stroomcijfer nodig heeft in ASIC, willen we het aantal bits dat per 1000 gates per klokcyclus verwerkt wordt, weergeven.
Dit wordt berekend door de verhouding van het aantal sleutelstroombits per klokperiode per 1000 gates te nemen. Zo bekomen we de volgende resultaten in tabel 9.7:

stroomcijfers	# bits/klokcyclus	# gates	bits per 1000 gates
A5/1v1	1	4821,55	0,21
A5/1v2	1	4857,16	0,21
Trivium1b	1	4047,52	0,25
Trivium8b	8	4359,99	1,83
Trivium16b	16	4707,41	3,40
Trivium32b	32	5394,58	5,93
Trivium64b	64	6774,26	9,45
Rabbit	128	70947,23	1,80
DECIMv2	1	4121,73	0,24
Salsa20	512	17197,38	29,77

Tabel 9.7: ASIC → aantal bits per klokcyclus per 1000 gates



Grafiek 9.9: ASIC → aantal bits per klokcyclus per 1000 gates

Grafiek 9.9 toont ons dat Salsa20 de beste waarden geeft. Trivium64b vertoont ook een goed resultaat in tegenstelling tot Rabbit en DECIMv2. In de zoektocht naar de meest optimale verhouding tussen aantal bits per klokcyclus blijft Trivium64b het beste scoren.

Opmerking
Het stroomcijfer A5/1 is een zeer klein stroomcijfer omdat het gebruik maakt van slechts 64 flipflops. Omdat A5/1 ons eerste stroomcijfer was, vallen onze resultaten qua oppervlakte groter uit dan verwacht. Onze implementatie bevat teveel flipflops die niets te maken hebben met de werking van het stroomcijfer zelf: de registers waarin de sleutel, de frameteller en de sleutelstroombits opgeslagen worden. Daarom willen we een kleine rechtzetting doorvoeren :

We berekenen het aantal gates zonder de overbodige flipflops. Er zijn slechts 64 flipflops nodig en elke flipflop bestaat in de gebruikte technologie uit 8 gates. De oorspronkelijke oppervlakte van A5/1 schommelt rond 115000 µm². Om een degelijke schatting uit te kunnen voeren trekken we van de oppervlakte het aantal overbodige flipflops af:

$$115000 - (314 * 23{,}8 * 8) = 55214{,}4\ \mu m^2$$

De gatecount zal bij aanpassing rond de 55214,4 µm² liggen. Dit betekent dat A5/1 het kleinste stroomcijfer is. De snelheid verandert door deze aanpassing niet noemenswaardig veel.

9.3 Algemeen besluit

Naar aanleiding van het door ECRYPT opgestarte eSTREAM-project is het onderwerp van onze meesterproef ontstaan: nieuwe stroomcijfers implementeren en vergelijken op basis van oppervlakte en snelheid. Wij hebben de stroomcijfers Trivium, Rabbit, DECIMv2 en Salsa20 toegewezen gekregen. Ter kennismaking met de hardware-implementatie van stroomcijfers kozen we A5/1: A5/1 is een goede vertrekbasis om de syntheseresultaten te toetsen aan een algemeen gekend, beproefd en reeds gebroken stroomcijfer.
Uiteindelijk luidt de conclusie dat uit onze reeks Trivium het efficiëntste stroomcijfer is. Het heeft weinig plaats nodig en stuurt per klokperiode 64 sleutelstroombits naar buiten. De overige stroomcijfers waren groter en/of sneller, zonder zoals Trivium de efficiënte combinatie te vormen in de zoektocht naar de beste trade-off tussen snelheid en oppervlakte.

Bronnen

KALLIST, J., *Computer Security and Industrial Cryptography*, 2005, online, http://www.esat.kuleuven.be/cosic, 1 december 2005.

LANO, J., *eStream*, 2005,online, http://www.ecrypt.eu.org/stream, 1 december 2005

LANO, J., Biryukov, A., Shamir, A., Wagner, D. J., *List of Stream Ciphers, Real-Time Cryptanalysis of A5/1 on a PC*, 2005, online, http://homes.esat.kuleuven.be/~jlano, 12 september 2005.

LANO, J. , Boesgaard, M., Vesterager, M., Christensen, T., Zenner, E., *Rabbit*, 2005, on line, http://www.ecrypt.eu.org/stream/rabbit.html, 20 november 2005

LANO, J., De Canniere, C., Preneel B., *Trivium*, 2005, online, http://www.ecrypt.eu.org/stream/trivium.html, 8 oktober 2005.

LANO, J., J.Bernstein, D., *Salsa20*, 2005, on line, http://www.ecrypt.eu.org/stream/salsa20.html, 22 maart 2006.

LANO, J., Wu, H., Preneel, B., *DECIMv2*, 2005, online, http://www.ecrypt.eu.org/stream/decim.html, 1 maart 2006.

MENEZES, A.J. (a), *Handbook of Applied Cryptography Chapter 1*, 1996, online, www.cacr.math.uwaterloo.ca/hac, 23 september 2005.

MENEZES, A.J. (b), *Handbook of Applied Cryptography Chapter 6*, 1996, online, www.cacr.math.uwaterloo.ca/hac, 28 september 2005.

PRENEEL, B., *Network of Excellence in Cryptography*, 2005, online, http://www.ecrypt.eu.org, 1 december 2005.

RUHR-UNIVERSITÄT BOCHUM, *Ruhr-Universität Bochum*, 2005, online, http://www.ruhr-uni-bochum.de/index_en.htm, 21 september 2005.

SYNOPSYS, *DesigVision*, 2005), online, http://www.synopsys.com, 1 mei 2006.

WIKIPEDIA (a), *Advanced Encryption Standard*, 2005, online, http://nl.wikipedia.org/wiki/Advanced_Encryption_Standard, 21 september 2005.

WIKIPEDIA (b), *Asymmetrische Cryptografie*, 2005, online, http://nl.wikipedia.org/wiki/Symmetrische_cryptografie, 22 september 2005.

WIKIPEDIA (c), *Cryptografie*, 2005, online, http://nl.wikipedia.org/wiki/Cryptografie, 21 september 2005

WIKIPEDIA (d), *Data Encryption Standard*, 2005, online, http://nl.wikipedia.org/wiki/Data_Encryption_Standard, 21 september 2005.

WIKIPEDIA (e), *A5/1*, 2005, online, http://en.wikipedia.org/wiki/A5/1, 12 september 2005.

WIKIPEDIA (f), *Symmetrische Cryptografie*, 2005, online, http://nl.wikipedia.org/wiki/Symmetrische_cryptografie, 21 september 2005.

XILINX, *Xilinx*, 2005, online, http://www.xilinx.com, 20 september 2005.

Bijlage

Op deze pagina bevindt er zich een cd-rom. Hierop is het volgende te vinden:

- de papers van elk stroomcijfer.
- de broncode van elk stroomcijfer (C-code) samen met de testvectoren.
- VHDL-files met de implementaties van de stroomcijfers met bijhorende testbenches.
- deze meesterproef in een PDF-bestand.