Hoofdstuk 1

Implementatie

In dit hoofdstuk wordt de implementatie van een schakeling voor de berekening van de Tate pairing uit de doeken gedaan. Het uiteindelijke doel is een compacte ASIC implementatie te verkrijgen.

Er zal onderzocht worden welke basisbewerkingen nodig zijn en hoe deze verwezenlijkt kunnen worden in hardware. Vervolgens wordt een schakeling ontworpen die aan de hand hiervan alle nodige berekeningen kan uitvoeren in het gekozen veld. Ten slotte is er dan nog de schakeling die alle berekeningen voor het Miller algoritme (zie Paragraaf **??**) in goede banen leidt. In het volgende hoofdstuk zal de resulterende schakeling dan beoordeeld worden.

Allereerst worden echter noodzakelijke parameters vastgelegd, zodat alle bewerkingen exact gedefiniëerd kunnen worden. Vervolgens wordt bekeken welke beperkingen aan de schakeling opgelegd moeten worden.

1.1 Parameters

Alvorens kan begonnen worden met de implementatie, moeten bepaalde parameters vastgelegd worden. Pas eens dat gedaan is, zijn alle bewerkingen volledig gedefinieerd.

Een belangrijke parameter is de kromme en het veld waarover gewerkt zal worden. Omdat de berekeningen in een veld $\mathbb{F}_2$ (en extensies ervan) veel eenvoudiger zijn dan die in een veld modulo een priemgetal, wordt ervoor gekozen in zulk een veld te werken. Verder wordt gekozen voor een supersingulaire elliptische kromme. Dat type krommen heeft een kleine embedding degree en er bestaan simpele rekenregels voor de verdubbeling en optelling van punten. De vergelijking van zo een kromme E over een veld $\mathbb{F}_{2^m}$ wordt gegeven door

$$E : y^3 + y = x^3 + x + b,$$

waarbij $b \in \{0, 1\}$. De embedding degree $k = 4$ voor dit type krommen. Verder worden de volgende hulp variabelen bepaald:

$$\begin{aligned} \delta &= b & m &\equiv 1, 7 (\mathrm{mod} 8) \\ &= 1 - b & m &\equiv 3, 5 (\mathrm{mod} 8) \\ \nu &= (-1)^{\delta} \end{aligned}$$

De orde van de kromme is gelijk aan [1, 2]:

$$\#E(\mathbb{F}_{2^m}) = 2^m + \nu\sqrt{2^{m+1}} + 1$$

Om de Tate pairing te kunnen berekenen, moet de variabele b zo gekozen worden dat $\#E$ enkel uit sommaties bestaat. Met andere woorden: b moet zo gekozen worden dat $\nu = 1$.

Alvorens dit kan gebeuren, moet een woordlengte m vastgelegd worden. Deze bepaald de cryptografische sterkte alsook de grootte van het eindresultaat. Des te groter m, des te groter de benodigde registers om alle data in op te slaan. In Tabel 1.1 zijn enkele vaak gebruikte woordlengtes voor pairings met hun bijhorende sterkte terug te vinden. Er wordt gekozen voor $m = 163$. Dit laat een mooi compromis toe tussen cryptografische sterkte en compactheid van de implementatie.

Tabel 1.1: Vergelijking van woordlengtes m en de bijhorende cryptografische sterkte van de Tate pairing

m (bit)	sterkte (bit)
81	324
163	652
239	956

Nu m vastgelegd is, kan hetzelfde gedaan worden voor b. Om een aftrekking in de formule voor $\#E$ te voorkomen, dient $b = 1$ te zijn in dit geval. Meteen kan dan ook een l-torsiepunt subgroep over E vastgelegd worden. Daarbij moet er op gelet worden dat $l \mid \#E$. De eenvoudigste keuze is:

$$\begin{aligned} l = \#E &= 2^{163} + \sqrt{2^{163+1}} + 1 \\ &= 2^{163} + 2^{82} + 1. \end{aligned}$$

Nu het veld waarover gewerkt wordt, bepaald is, kan een reductie veelterm P gekozen worden. Bij de keuze daarvan werd uitgegaan van de aangeraden parameters in [3]. De veelterm is:

$$P = z^{163} + z^7 + z^6 + z^3 + 1.$$

Ook moet het extensieveld $\mathbb{F}_{2^{km}}$ gedefinieerd worden waarover de Tate pairing berekend zal worden. Aangezien $k = 4$, kan het veld opgebouwd worden in twee stappen. Eerst wordt $\mathbb{F}_{2^{2m}}$ bepaald als volgt:

$$\mathbb{F}_{2^{2m}} \cong \mathbb{F}_{2^m}[x]/(x^2 + x^1)$$

en hierop wordt het volgende extensieveld gebouwd:

$$\mathbb{F}_{2^{4m}} \cong \mathbb{F}_{2^{2m}}[y]/(y^2 + (x+1)y + 1).$$

Een element van $\mathbb{F}_{2^{4m}}$ kan dus voorgesteld worden als vier elementen van $\mathbb{F}_{2^m}$.

Ten slotte moet een distortiemap voor gebruik in de berekening van de Tate bepaald worden. Hiervoor wordt dezelfde gekozen als in [1]. Een punt $A \in \mathbb{F}_{2^m}$ ondergaat onder de distortie volgende transformatie:

$$\phi : (x_A, y_A) \rightarrow (x_A + s^2, y_A + x_A s + t^6)$$

met $s, t \in \mathbb{F}_{2^{4m}}$. Verder moeten s en t voldoen aan volgende vergelijkingen:

$$s^4 + s = 0$$
$$t^2 + t + s^6 + s^2 = 0.$$

Een mogelijke oplossing hiervoor is:

$$s = x + 1$$
$$t = xy.$$

Een distortie van A kan dus ook geschreven worden als:

$$x_\phi = x_A + x$$
$$y_\phi = (x_A + y_A) + x_A \cdot x + xy.$$

Merk op dat de coëfficiënt van 1 van x_ϕ en die van x van y_ϕ beiden gelijk zijn aan x_A. Door deze handige vorm zullen in de implementatie slechts twee registers nodig zijn om de distortie van het punt Q bij te houden.

1.2 Beperkingen

Het doel is de uiteindelijke schakeling zo klein mogelijk te maken, zodat ze gebruikt kan worden in bv. netwerken van sensoren of een smartcard. Beperking van de oppervlakte is dus de belangrijkste factor. Een tweede belangrijke factor is vermogenverbruik, om dit te beperken bestaan specifieke technieken. Het verbruik hangt ook samen met de oppervlakte, dus het beperken daarvan zal het verbruik ten goede komen. In eerste instantie zal dus getracht worden de implementatie zo compact mogelijk te maken. Het verbruik kan verder ook verlaagd worden door een lagere kloksnelheid voor de schakeling te gebruiken, wat uiteraard de rekensnelheid niet bevordert. De rekensnelheid is echter geen prioriteit en dus zal dit aspect bij het ontwerp van de schakelingen genegeerd worden.

Algemeen kan dus gesteld worden dat hoe kleiner het uiteindelijke resultaat is, hoe beter. Het is dus cruciaal de elementen te identificeren die het meeste plaats innemen in een ASIC schakeling. In Tabel 1.2 is de grootte van de belangrijkste elementen te vinden. Deze cijfers gelden enkel bij gebruik van $0.13\mu m$ low leakage technologie van Faraday Technology Corporation. De ordening van de elementen zal echter grotendeels behouden voor andere technologiën. Uit de tabel blijkt dat het gebruik van flip-flops (registers), adders en multiplexers zoveel mogelijk beperkt moet worden.

1.3 Modular Arithmetic Logical Unit

De kern van de hardware implementatie wordt gevormd door de Modular Arithmetic Logical Unit (MALU) [5, 6]. Dit circuit laat toe basis bewerkingen uit te voeren op getallen. Gezien de beperking die is opgelegd aan de oppervlakte van de schakeling, wordt enkel de optelling geïmplementeerd. Later wordt met behulp daarvan elke andere nodige berekening verwezenlijkt.

Aangezien er in het veld $\mathbb{F}_{2^m}$ gewerkt wordt, is een optelling equivalent aan een XOR bewerking. De bewerking die moet uitgevoerd kunnen worden is:

Tabel 1.2: Grootte van elementen in een ASIC schakeling in gates/bit ($0.13\mu m$ low leakage technologie van Faraday Technology Corporation) [4]

Element	Gates/bit
D flip-flop met reset	6
D flip-flop zonder reset	5.5
full adder	5.5
D latch	4.25
3 ingang MUX	4
2 ingang XNOR	3.75
2 ingang XOR	3.75
2 ingang MUX	2.25
2 ingang OR	1.25
2 ingang AND	1.25
2 ingang NOR	1
2 ingang NAND	1
NOT	0.75

$$\begin{aligned} T + B &= T \oplus B \\ &= R \mod P \end{aligned}$$

Merk op dat bij een optelling de graad van R enkel kleiner of gelijk kan zijn aan die van T en B. Indien B van graad $< m$ is (dus $\in \mathbb{F}_{2^m}$) en T van graad $\leq m$ ($\in \mathbb{F}_{2^{m+1}}$), is de optelling te implementeren als in Algoritme 1.1.

Algoritme 1.1: Modulo optelling in $\mathbb{F}_{2^m}$

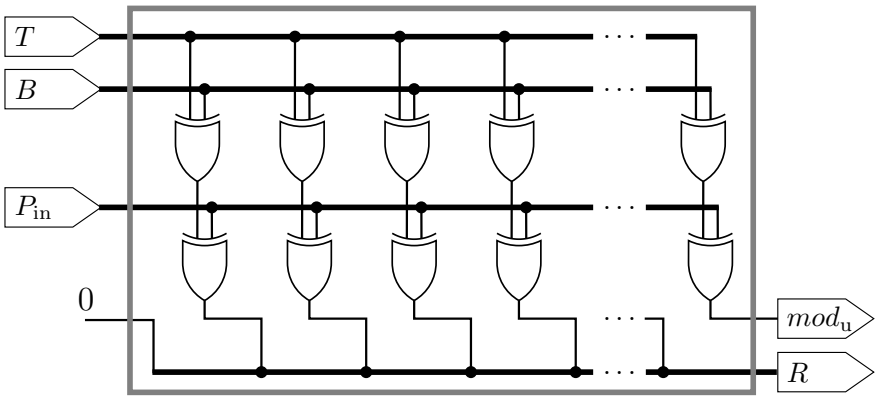
Input: $B \in \mathbb{F}_{2^m}$, $T \in \mathbb{F}_{2^{m+1}}$
Output: $R \mod P \in \mathbb{F}_{2^m}$

```
1 R ← T ⊕ B
2 if Degree(R) = m then
3     R ← R ⊕ P
```

In Paragraaf **??** zal blijken dat het vaak nodig zal zijn om het resultaat R te vermenigvuldigen met z, m.a.w. alle bits 1 plaats naar links te verschuiven. Een voor de hand liggende schakeling die dit alles implementeert, is te zien in Figuur 1.1. Ingang P_{in} dient afhankelijk van de graad van T ingesteld te worden op 0 of P. De ingangen T en P_{in} zijn m bits aangezien het resultaat voor de vermenigvuldiging met z steeds van graad $< m$ is en bit $m+1$ dus toch steeds 0 zou zijn. De hoogste graad term na de shift wordt naar buiten gebracht als mod_{u}. De implementatie bestaat uit $2m$ XOR poorten.

Aangezien voor het ontwerp het veld en de modulo veelterm op voorhand bepaald zijn, is het mogelijk een zeer groot aantal XOR poorten uit het ontwerp te verwijderen. De ingang P en de bijhorende m XOR poorten kunnen vervangen worden door een 1 bit 'modulo enable' ingang mod_{e} en er worden enkel XOR poorten geplaatst voor de bits i waarvoor $P_i = 1$. Hierdoor wordt het aantal ingangen drastisch verkleind en worden

$$\Delta = m - (\mathsf{Hamm}(P) - 1)$$

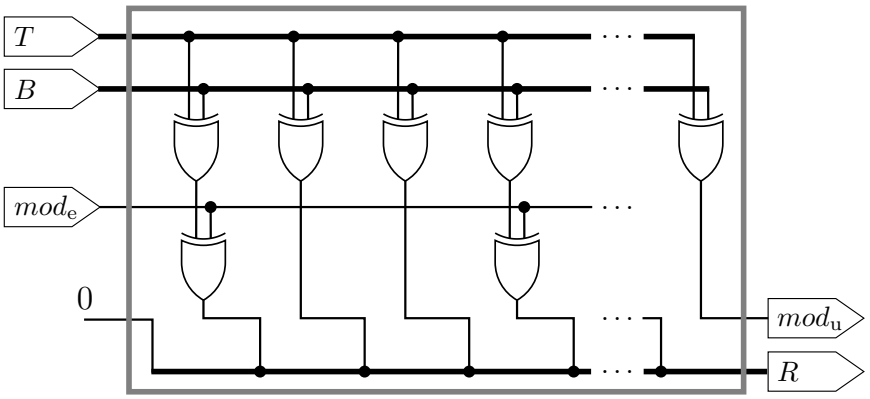


Figuur 1.1: MALU - Basis ontwerp met shift

XOR poorten uitgespaard, met $\mathsf{Hamm}(P)$ gelijk aan het Hamming gewicht van de binaire representatie van P.

In dit geval is $m = 163$ en $P = z^{163} + z^7 + z^6 + z^3 + 1$. Er zijn dus $\mathsf{Hamm}(P) - 1 = 4$ XOR poorten nodig, wat een besparing van $163 - 4 = 159$ XOR poorten oplevert (51% minder dan het oorsponkelijk aantal).

De resulterende schakeling is te zien in Figuur 1.2.



Figuur 1.2: MALU - Geoptimaliseerd ontwerp met shift

1.4 Berekeningen in $\mathbb{F}_{2^m}$

1.4.1 Basisontwerp

De eerder ontworpen MALU schakeling laat toe optellingen te doen, maar het Miller algoritme vereist dat er ook vermenigvuldigingen worden uitgerekend. Delingen en machtsverheffingen kunnen met behulp van vermenigvuldiging berekend worden en dienen dus niet rechtstreeks geïmplementeerd te worden. Indien dus zowel optellingen als vermenigvuldigingen berekend kunnen worden, is

alles voorhanden om de Tate pairing te berekenen.

Door toepassing van een "shift and add" algoritme, kan de waarde van $A \cdot B = R$ berekend worden met behulp van de MALU schakeling. In Algoritme 1.2 is te zien hoe dit juist in z'n werk gaat. Door de modulo operatie telkens op het tussenresultaat uit te voeren, is het steeds van graad $\leq m$ en kan het opgeslagen worden in T. Op het einde moet het resultaat door z gedeeld worden, wat neerkomt op een verschuiving van alle bits met 1 plaats naar rechts.

Algoritme 1.2: "Shift and add" vermenigvuldiging in $\mathbb{F}_{2^m}$

Input: $A, B \in \mathbb{F}_{2^m}$
Output: $R = A \cdot B \in \mathbb{F}_{2^m}$
Data: $T \in \mathbb{F}_{2^{m+1}}$

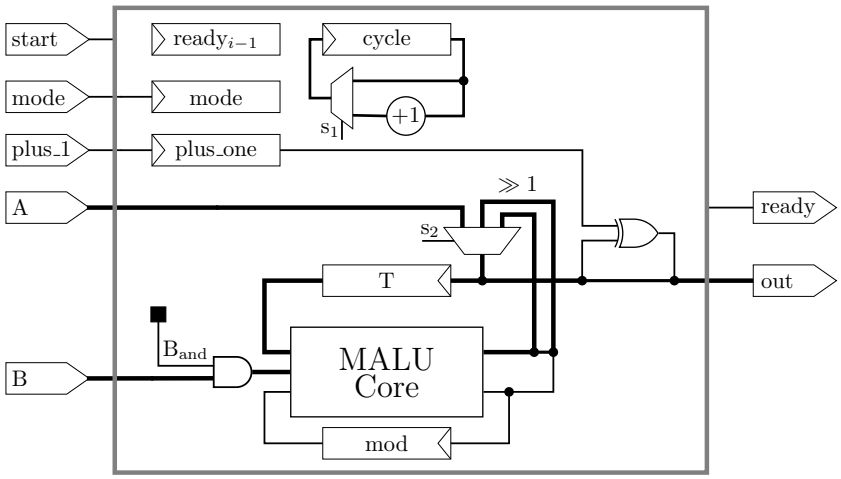
```
1  T ← 0
2  for i ← m − 1 to 0 do
3      if A_i = 1 then
4          b ← B
5      else
6          b ← 0
7      T ← T ⊕ b
8      if Degree(T) = m then
9          T ← T ⊕ P
10     T ← T ≪ 1
11 R ← T ≫ 1
```

Wanneer de optelling en vermenigvuldiging nu in een schakeling gegoten worden, dient te schakeling te weten welke van de twee bewerkingen moet uitgevoerd worden. Verder moet het mogelijk zijn de uitkomt R in het register T op te slaan. Op die manier is het mogelijk de uitgang van de schakeling gelijk te stellen aan R zolang geen nieuwe berekening gestart wordt.

Verderop zal gezien worden dat in het Miller algoritme verscheidene keren de som $R+1$ moet berekend worden. Daarom wordt aan de schakeling een ingang *plus_one* toegevoegd die hierin helpt voorzien. De uiteindelijke schakeling is te zien in Figuur 1.3. Er wordt zo veel mogelijk bespaard op registers. Het register *cycle* (equivalent aan i in Algoritme 1.2) is $\lceil \log_2(m) \rceil$ bits lang en register T m bits. De waarde van T_m wordt opgeslagen in register *mod*. Alle overige registers zijn 1 bit groot. Merk dus op dat de variabele T uit Algoritme 1.2 hier gevormd wordt door de combinatie van de registers T en *mod*.

Voor A wordt geen appart register voorzien en in plaats van A_i wordt steeds bit A_m ingelezen voor de vermenigvuldiging. De schakeling die van deze schakeling gebruik maakt, dient dus te voorzien in een methode om elke klokslag de juiste A_i aan te bieden op A_m. Dit kan simpelweg gebeuren door elke klokslag na de start van de berekening het register dat A bevat één positie naar links door te schuiven.

Gezien de eenvoud van de schakeling is het niet nodig een FSM te implementeren, de besturing kan volledig via logica gebeuren. Die wordt getoond in Figuur 1.4. De werking is zeer eenvoudig: zo lang *start* hoog is, worden de registers *mode* en *plus_one* geladen met hun respectievelijke ingangen. Verder wordt, afhankelijk van *mode*, F ingeladen met de waarde van A (optelling,



Figuur 1.3: Schakeling voor berekeningen in $\mathbb{F}_{2^m}$

$mode = 0$) of gelijkgesteld aan nul (vermenigvuldiging, $mode = 1$). Ook wordt *cycle* gereset. Wanneer *start* laag is, wordt, afhankelijk van de gewenste bewerking en de waarde van *ready*, het register T geladen met de uitgang R van de MALU of het uiteindelijke resultaat $R_{\text{ready}} = R \gg 1$.

1.4.2 Versnelling van de vermenigvuldiging

Wanneer met behulp van de schakeling in Figuur 1.3 een vermenigvuldiging wordt berekend, zal het m klokcycli duren eer het resultaat beschikbaar is aan de uitgang. Het is echter mogelijk dat aantal drastisch naar beneden te halen door d MALU's te gebruiken en dus d optellingen per klokcyclus uit te voeren. Het principe hiervan wordt geïllustreerd in Figuur 1.5.

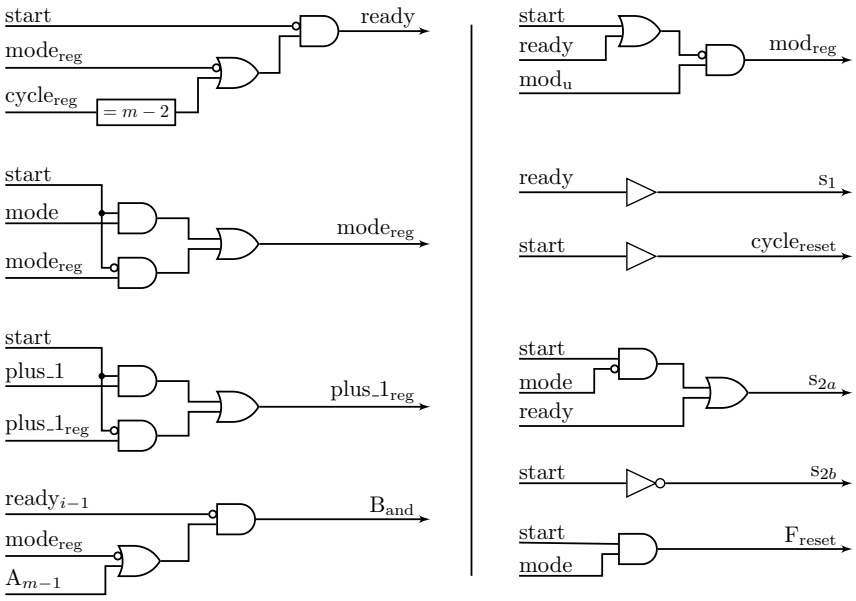
De rekentijd van het Miller algoritme zal door toepassing van deze techniek gevoelig verkort kunnen worden. Hoe groter m en hoe meer vermenigvuldigigen er uitgevoerd dienen te worden, des te significanter de tijdswinst die geboekt kan worden. Uiteraard gaat het gebruik van deze techniek wel in tegen de eerder opgelegde beperking aan de grootte van de uiteindelijke schakeling. Het is echter niet zo dat er enkel $d - 1$ extra MALU blokken dienen toegevoegd te worden, afhankelijk van d en m dient ook een extra multiplexer in de schakeling gestoken te worden. Dit is zoals opgemerkt in Paragraaf 1.2 een zeer slechte zaak voor de oppervlakte.

Stel bijvoorbeeld $d = 4$ (en $m = 163$). Het resultaat van een optelling zal net zoals in het standaard ontwerp (Figuur 1.3) aanwezig zijn aan de uitgang van MALU n° 1. Het resultaat van een vermenigvuldiging zal echter aan de uitgang van MALU n° 3 verschijnen, aangezien $163 \bmod 4 = 3$. Het eindresultaat dat in register T dient opgeslagen te worden, is voor een vermenigvuldiging dus

$$R_{3_{\text{ready}}} = mod_{\text{u}_3} \# R_{3_{162:1}},$$

terwijl dit voor een optelling

$$R_{1_{\text{ready}}} = mod_{\text{u}_1} \# R_{1_{162:1}}$$



Figuur 1.4: Logica voor besturing van de schakeling voor berekeningen in $\mathbb{F}_{2^m}$

is. Met andere woorden, er dient nu niet enkel gekozen te kunnen worden tussen de ingangen A, R_d of $R_{1_{\text{ready}}}$, maar ook voor $R_{3_{\text{ready}}}$.

Indien men toch wenst het vermenigvuldigen te versnellen, is het aangeraden een d te kiezen waarvoor $m \bmod d = 1$. Als voorbeeld worden enkele voor de hand liggende en optimale keuzes vergeleken voor d indien $m = 163$ in Tabel 1.3.

Tabel 1.3: Voor de hand liggende versus optimale waarden voor woordbreedte d indien $m = 163$

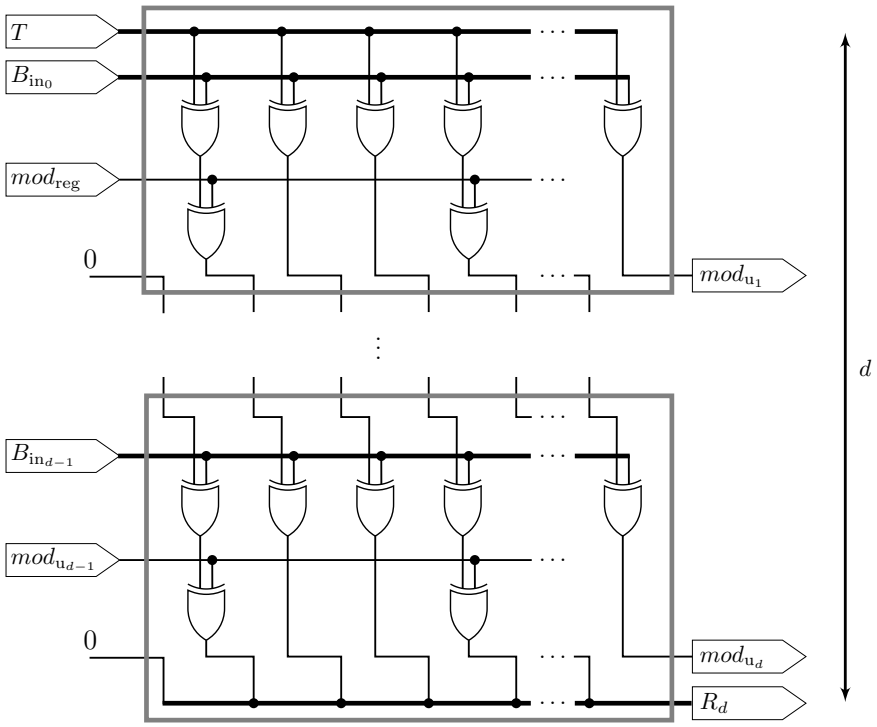
Voor de hand liggende waarden voor d						
d	2	4	8	16	32	64
$m \bmod d$	1	3	3	3	3	35

Ideale waarden voor d						
d	2	3	6	9	18	27
$m \bmod d$	1	1	1	1	1	1

1.5 Controller voor het Miller algoritme

1.5.1 Algemeen ontwerp

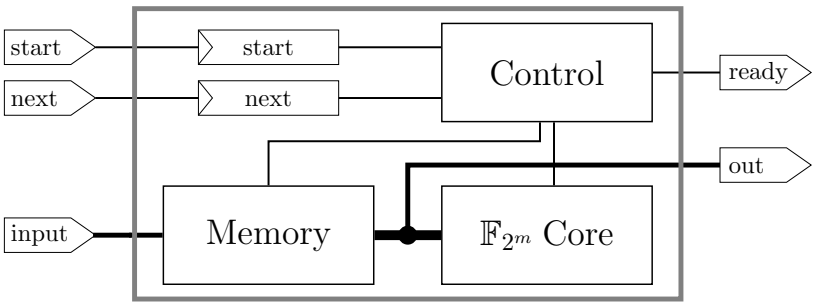
Nu een schakeling voorhanden is die toelaat alle benodigde berekeningen uit te voeren, rest nog een schakeling te ontwerpen die het Miller algoritme (Algoritme **??**) uitvoert. Het algoritme met invulling van de vastgelegde parameters, zonder uitwerking van de berekeningen, wordt gegeven in Algoritme 1.3.



Figuur 1.5: Schakeling voor berekeningen in $\mathbb{F}_{2^m}$ met woordbreedte d

Merk op dat op lijn 6 slechts één waarde moet nagekeken worden, aangezien $l = 2^{163} + 2^{82} + 1$.

De controller zal ontworpen worden zoals het schema in Figuur 1.6. Voor het geheugenblok volledig ontworpen kan worden, moeten echter eerst de verschillende berekeningen uitgewerkt worden. Vervolgens zal aan de hand van die uitwerkingen bepaald worden hoeveel registers ten minste noodzakelijk zijn. Afhankelijk van diezelfde uitwerkingen zullen ook enkele geheugen elementen voor tellers en statusbits toegevoegd moeten worden. Ten slotte zal een FSM ontworpen worden.



Figuur 1.6: Schakeling voor de uitvoering van het Miller algoritme

Algoritme 1.3: Miller algoritme voor berekening van de Tate pairing met parameters ingevuld

Input: $P, Q \in E(\mathbb{F}_{2^{163}})[l]$
Output: $e(P, Q)$

1 $F \leftarrow 1$
2 $I \leftarrow P$
3 **for** $i \leftarrow 162$ **to** 0 **do**
4 $\quad F \leftarrow F^2 \cdot G_{I,I}(\phi(Q))$
5 $\quad I \leftarrow 2I$
6 $\quad$ **if** $i = 82$ **then**
7 $\quad\quad F \leftarrow F \cdot G_{I,P}(\phi(Q))$
8 $\quad\quad I \leftarrow I + P$
9 $e(P, Q) \leftarrow F^{\frac{2^{4 \cdot 163}-1}{2^{163}+2^{82}+1}}$
10 **return** $e(P, Q)$

Grofweg kan het algoritme opgedeeld worden in de for-lus en een finale machtsverheffing. De for-lus kan verder onderverdeeld worden in een verdubbelstap, een optelstap, een kwadratering van F en een vermenigvuldiging $F \cdot G$. Elk van deze stappen zal verder uitgediept worden en er zal voor elke berekening bepaald worden hoeveel tussenresultaten minimum opgeslagen moeten worden. Het zal blijken dat een inversie in $\mathbb{F}_{2^m}$ uitgerekend moet kunnen worden, wat ook verder uitgediept zal worden.

Bij elk van de volgende algoritmen zal aangegeven worden hoeveel en welke bewerkingen juist nodig zijn. Daarbij staat A voor een optelling, M voor een vermenigvuldiging, I voor een kwadratering en I voor een inversie. Aangezien er echter geen afzonderlijke schakeling voor kwadrateren ontworpen is, zijn I en M qua rekentijd in dit geval equivalent aan elkaar. De bewerking $a + 1$ neemt geen extra tijd in beslag, omdat die functie parallel met een optelling of vermenigvuldiging kan uitgevoerd worden door de *plus_one* van de schakeling voor berekeningen in $\mathbb{F}_{2^m}$ hoog te maken bij de start van een berekening.

1.5.2 For-lus

Zoals reeds eerder vermeld kan de for-lus onderverdeeld worden in een verdubbelstap, een optelstap, een kwadratering van F en een vermenigvuldiging $F \cdot G$. Elk van deze onderdelen zal in de volgende paragrafen in detail aan bod komen.

Verdubbelstap

De verdubbelstap wordt gevormd door lijnen 4 en 5 in Algoritme 1.3. Voor een hyperelliptische kromme zijn de berekeningen als volgt [1, 7]:

$$\begin{aligned} \lambda &= x_I^2 + 1 \\ x_{2I} &= \lambda^2 \\ y_{2I} &= \lambda(x_{2I} + x_I) + y_I + 1 \\ G_{I,I}(\phi(Q)) &= \lambda(x_\phi + x_I) + (y_\phi + y_I) \end{aligned}$$

In dit geval kan y_{2I} ook berekend worden als:

$$\begin{aligned} y_{2I} &= y_I^4 + x_I^4 \\ &= (y_I + x_I)^4 \end{aligned}$$

Aangezien dit echter twee kwadrateringen en een optelling kost tegenover een vermenigvuldiging en twee optellingen, wordt de voorkeur gegeven aan de eerste methode.

Door de specifieke vorm van $\phi(Q)$ kan G uitgeschreven worden als:

$$\begin{aligned} G_a &= \lambda(x_{\phi_a} + x_I) + (y_{\phi_a} + y_I) & G_c &= \lambda \cdot x_{\phi_c} + y_{\phi_c} \\ G_b &= \lambda \cdot x_{\phi_b} + y_{\phi_b} & &= 0 \\ &= \lambda + y_{\phi_b} & G_d &= \lambda \cdot x_{\phi_d} + y_{\phi_d} \\ &= \lambda + x_{\phi_a} & &= 1 \end{aligned}$$

De variabele G kan dus opgeslagen worden in twee registers van grootte m in plaats van in vier. De vorm van G zal ook toelaten de vermenigvuldiging $F \cdot G$ grotendeels te vereenvoudigen, zoals later gezien zal worden.

Tanneer dit in rekening gebracht wordt en het algoritme op register niveau wordt uitgeschreven, bekomt men uiteindelijk Algoritme 1.4. Hierbij werd specifiek gelet op een minimum gebruik van tijdelijke registers.

Buiten registers voor x_{2I}, y_{2I}, x_{ϕ_a}, y_{ϕ_a}, G_a en G_b is er ook een register nodig om λ in op te slaan. In totaal moeten er zes optellingen, twee vermenigvuldigingen en twee kwadrateringen uitgerekend worden.

Algoritme 1.4: Uitwerking van de verdubbelstap voor hyperelliptische krommen in het Miller algoritme

Input: $x_I, y_I \in E(\mathbb{F}_{2^m})$
Output: $x_{2I}, y_{2I} \in E(\mathbb{F}_{2^m})$; $G \in \mathbb{F}_{2^{4m}}$
Data: $\lambda \in \mathbb{F}_{2^m}$

1	$G_a \leftarrow x_I$; $G_b \leftarrow y_I$	
2	$\lambda \leftarrow G_a^2 + 1$; $x_{2I} \leftarrow \lambda^2$	2 I
3	$y_{2I} \leftarrow x_{2I} + G_a$; $y_{2I} \leftarrow y_{2I} \cdot \lambda$	1 M, 1 A
4	$y_{2I} \leftarrow y_{2I} + G_b + 1$	1 A
5	$G_a \leftarrow G_a + x_{\phi_a}$; $G_a \leftarrow G_a \cdot \lambda$	1 M, 1 A
6	$G_a \leftarrow G_a + y_{\phi_a}$; $G_a \leftarrow G_a + G_b$	2 A
7	$G_b \leftarrow \lambda + x_{\phi_a}$	1 A

Optelstap

De optelstap bestaat uit lijnen 7 en 8 van Algoritme 1.3. Voor een hyperelliptische kromme dienen de volgende bewerkingen uitgevoerd te worden [1, 7]:

$$\begin{aligned} \lambda &= \frac{y_I + y_P}{x_I + x_P} \\ x_{I+P} &= \lambda^2 + x_I + x_P \\ y_{I+P} &= \lambda(x_{I+P} + x_P) + y_P + 1 \\ G_{I,I}(\phi(Q)) &= \lambda(x_\phi + x_P) + (y_\phi + y_P) \end{aligned}$$

Net zoals bij de verdubbelstap kan G hier in 2 variabelen opgeslagen worden. Hoewel de optelstap slechts één maal moet worden uitgevoerd, is het uiteraard cruciaal dat ook hier zo weinig mogelijk tijdelijke variabelen gebruikt worden. Op die manier blijft de grootte van de uiteindelijke schakeling het kleinst. De uitgewerkte versie van het algoritme wordt gegeven in Algoritme 1.5.

In tegenstelling tot de verdubbelstap zijn hier twee tijdelijke registers nodig, een voor λ en een voor a. Verder zijn er twee registers nodig voor x_P en y_P. Alles samen dienen er tien optellingen, drie vermenigvuldigingen, twee kwadrateringen en een inversie uitgerekend te worden.

Algoritme 1.5: Uitwerking van de optelstap voor hyperelliptische krommen in het Miller algoritme

Input: $x_I, y_I, x_P, y_P \in E(\mathbb{F}_{2^m})$
Output: $x_{I+P}, y_{I+P} \in E(\mathbb{F}_{2^m})$; $G \in \mathbb{F}_{2^{4m}}$
Data: $\lambda, a \in \mathbb{F}_{2^m}$

1	$G_a \leftarrow x_I$; $G_b \leftarrow y_I$	
2	$\lambda \leftarrow G_a + x_P$; $\lambda \leftarrow \lambda^{-1}$	1 I, 1 A
3	$a \leftarrow G_b + y_P$; $\lambda \leftarrow \lambda \cdot a$	1 M, 1 A
4	$x_{I+P} \leftarrow \lambda^2 + G_a$; $x_{I+P} \leftarrow x_{I+P} + x_P$	1 I, 2 A
5	$y_{I+P} \leftarrow x_{I+P} + x_P$; $y_{I+P} \leftarrow y_{I+P} \cdot \lambda$	1 M, 1 A
6	$y_{I+P} \leftarrow y_{I+P} + y_P + 1$	1 A
7	$G_a \leftarrow x_{\phi_a} + x_P$; $G_a \leftarrow G_a \cdot \lambda$	1 M, 1 A
8	$G_a \leftarrow G_a + y_{\phi_a}$; $G_a \leftarrow G_a + y_P$	2 A
9	$G_b \leftarrow \lambda + x_{\phi_a}$	1 A

Inversie

De meest tijdrovende stap in de optelstap is de inversie. Zoals reeds vermeld in Paragraaf **??**, kan een inversie in een Galois veld berekend worden door toepassing van de kleine stelling van Fermat:

$$\begin{aligned} a^{2^m} &= a \\ a^{2^m-1} &= 1 \\ a^{2^m-2} &= a^{-1} \end{aligned}$$

De naieve manier om dit te berekenen zou zijn om a $2^m - 2$ keer met zichzelf te vermenigvuldigingen. In dit geval zou dat betekenen dat er $2^{163} - 2 = 11692013098647223345629478661730264157247460343806$ vermenigvuldigingen zouden moeten uitgevoerd worden. Zoiets is uiteraard onhaalbaar.

Een tweede manier bestaat er in de exponent te ontbinden in machten van 2 en 3. In dat geval zouden er nog 237 vermenigvuldigen nodig zijn.

Er is echter een derde, optimale manier die toegepast kan worden indien de exponent van de vorm $2^m - 2$ is [8][9]. Dit gaat als volgt in zijn werk:

$$a^{2^m-2} = (a^{2^{m-1}-1})^2$$

Als wordt aangenomen dat m oneven is, is de macht van twee na het gelijkheidsteken dus even. Zolang de exponent even is, kan recursief volgende formule toegepast worden:

$$a^{2^i-1} = (a^{2^{\frac{i}{2}}-1})^{2^{\frac{i}{2}}} \cdot a^{2^{\frac{i}{2}}-1}$$

Indien a oneven is, dient volgende formule toegepast te worden:

$$a^{2^i-1} = (a^{2^{i-1}-1})^2 \cdot a$$

Uiteindelijk eindigt men dan bij a^2 of a^3. Het totaal aantal bewerkingen voor een inversie in $\mathbb{F}_{2^m}$ is $\lfloor \log_2(m-1) \rfloor + \mathsf{Hamm}(m-1) + 1$ vermenigvuldigingen en $m-1$ kwadrateringen. $\mathsf{Hamm}(x)$ staat daarbij voor het Hamming gewicht van de binaire voorstelling van x.

In het geval van $m = 163$ is de uiteindelijke keten van bewerkingen zoals gegeven in Algoritme 1.6. Het aantal berekeningen in dat geval is 9 vermenigvuldigingen en 162 kwadrateringen. Er is een register nodig om a bij de houden en twee voor de tussenresultaten a^{2^i-1} en $(a^{2^i-1})^{2^i}$.

Algoritme 1.6: Inversie in $\mathbb{F}_{2^{163}}$

Input: $a \in \mathbb{F}_{2^{163}}$
Output: $a^{-1} \in \mathbb{F}_{2^{163}}$

1	$a^3 \leftarrow a^2 \cdot a$	1 I, 1 M
2	$a^{2^4-1} \leftarrow (a^3)^{2^2} \cdot a^3$	2 I, 1 M
3	$a^{2^5-1} \leftarrow (a^{2^4-1})^2 \cdot a$	1 I, 1 M
4	$a^{2^{10}-1} \leftarrow (a^{2^5-1})^{2^5} \cdot a^{2^5-1}$	5 I, 1 M
5	$a^{2^{20}-1} \leftarrow (a^{2^{10}-1})^{2^{10}} \cdot a^{2^{10}-1}$	10 I, 1 M
6	$a^{2^{40}-1} \leftarrow (a^{2^{20}-1})^{2^{20}} \cdot a^{2^{20}-1}$	20 I, 1 M
7	$a^{2^{80}-1} \leftarrow (a^{2^{40}-1})^{2^{40}} \cdot a^{2^{40}-1}$	40 I, 1 M
8	$a^{2^{81}-1} \leftarrow (a^{2^{80}-1})^2 \cdot a$	1 I, 1 M
9	$a^{2^{162}-1} \leftarrow (a^{2^{81}-1})^{2^{81}} \cdot a^{2^{81}-1}$	81 I, 1 M
10	$a^{-1} \leftarrow (a^{2^{162}-1})^2$	1 I

Kwadratering van F

Bij het uitvoeren van lijn 4 van Algoritme 1.3 moet ook telkens het kwadraat van F berekend worden. De afleiding van de formule daarvoor gaat als volgt:

$$\begin{aligned}
F^2 &= (F_a + F_b x + F_c y + F_d xy) \cdot (F_a + F_b x + F_c y + F_d xy) \\
&= F_a^2 + F_a F_b x + F_a F_c y + F_a F_d xy + F_b F_a x + F_b^2 x^2 + F_b F_c xy \\
&\quad + F_b F_d x^2 y + F_c F_a y + F_c F_b xy + F_c^2 y^2 + F_c F_d xy^2 + F_d F_a xy \\
&\quad + F_d F_b x^2 y + F_d F_c xy^2 + F_d^2 x^2 y^2 \\
&= (F_a^2 + F_b^2 + F_c^2 + F_d^2) \\
&\quad + (F_a F_b + F_b F_a + F_b^2 + F_c F_d + F_d F_c + F_d^2) x \\
&\quad + (F_a F_c + F_b F_d + F_c F_a + F_c^2 + F_c F_d + F_d F_b + F_d F_c) y \\
&\quad + (F_a F_d + F_b F_c + F_b F_d + F_c F_b + F_c^2 + F_d F_a + F_d F_b + F_d^2) xy \\
&= (F_a^2 + F_b^2 + F_c^2 + F_d^2) + (F_b^2 + F_d^2) x + F_c^2 y + (F_c^2 + F_d^2) xy
\end{aligned}$$

Mist de originele waarde van F overschreven mag worden, is het mogelijk dit te berekenen zonder gebruik van tijdelijke variabelen. Er zijn dus enkel vier registers nodig voor F. Hoe dat in z'n werk gaat is te zien in Algoritme 1.7. Eén kwadratering van F vraagt vier optellingen en vier kwadrateringen in $\mathbb{F}_{2^m}$.

Algoritme 1.7: Uitwerking van van $F^2 \in \mathbb{F}_{2^{4m}}$

Input: $F = F_a + F_b x + F_c y + F_d xy \in \mathbb{F}_{2^{4m}}$
Output: $F = F^2 \in \mathbb{F}_{2^{4m}}$

1	$F_a \leftarrow F_a + F_c$	1 A
2	$F_a \leftarrow F_a^2$	1 I
3	$F_b \leftarrow F_b + F_d$	1 A
4	$F_b \leftarrow F_b^2$	1 I
5	$F_a \leftarrow F_a + F_b$	1 A
6	$F_c \leftarrow F_c^2$	1 I
7	$F_d \leftarrow F_d^2$	1 I
8	$F_d \leftarrow F_d + F_c$	1 A

Vermenigvuldiging $F \cdot G$

Zoals eerder opgemerkt, is G in zowel de verdubbel- als optelstap niet van volledige rang in het extensieveld. De vermenigvuldiging van F met G kan daardoor vereenvoudigd worden, namelijk als volgt:

$$\begin{aligned}
F \cdot G &= (F_a + F_b x + F_c y + F_d xy) \cdot (G_a + G_b x + xy) \\
&= F_a G_a + F_a G_b x + F_a xy + F_b G_a x + F_b G_b x^2 + F_b x^2 y + F_c G_a y \\
&\quad + F_c G_b xy + F_c xy^2 + F_d G_a xy + F_d G_b x^2 y + F_d x^2 y^2 \\
&= (F_a G_a + F_b G_b + F_d) \\
&\quad + (F_a G_b + F_b G_a + F_b G_b + F_c + F_d) x \\
&\quad + (F_b + F_c G_a + F_c + F_d G_b) y \\
&\quad + (F_a + F_b + F_c G_b + F_d G_a + F_d G_b + F_d) xy
\end{aligned}$$

Indien hier nu de Karatsuba-Ofman techniek ([10, 11]) op wordt toegepast, bekomt men:

$$\begin{aligned}
F \cdot G = &(F_a G_a + F_b G_b + F_d) \\
&+ ((F_a + F_b) \cdot (G_a + G_b) + F_a G_a + F_c + F_d) x \\
&+ (F_c G_a + F_d G_b + F_b + F_c) y \\
&+ ((F_c + F_d) \cdot (G_a + G_b) + F_c G_a + F_a + F_b + F_d) xy
\end{aligned}$$

Deze formule kan uitgerekend wordt met gebruik van drie tijdelijke registers (a, b en c). Verder zijn er vier registers nodig voor F en twee voor G. Merk op dat de oude waarde van F overschreven wordt door het resultaat. In totaal zijn er zes vermenigvuldigingen en veertien optellingen nodig. Algoritme 1.8 beschrijft welke berekeningen juist uitgevoerd moeten worden.

Mist het gebruik van een vierde tijdelijk register zou het mogelijk zijn de berekening $G_a + G_b$ op te slaan. Die wordt nu zowel in lijn 2 als 7 berekend. Er

zouden dan slechts dertien optellingen moeten berekend worden, één minder dan [2], waar $y^2 + y + x$ gebruikt wordt als modulo veelterm voor het extensieveld. Aangezien een extra tijdelijk register echter tegen de doelstellingen ingaat, wordt voor de iets langere berekening gekozen.

Algoritme 1.8: Uitwerking van de vermenigvuldiging $F \cdot G$ in het Miller algoritme

Input: $F = F_a + F_b x + F_c y + F_d xy, G = G_a + G_b x + xy \in \mathbb{F}_{2^{4m}}$
Output: $F = F \cdot G \in \mathbb{F}_{2^{4m}}$
Data: $a, b, c \in \mathbb{F}_{2^m}$

1	$a \leftarrow F_a \cdot G_a$; $a \leftarrow a + F_d$	1 M, 1 A
2	$b \leftarrow F_a + F_b$; $c \leftarrow G_a + G_b$	2 A
3	$b \leftarrow b \cdot c$; $b \leftarrow b + a$; $b \leftarrow b + F_c$	1 M, 2 A
4	$c \leftarrow F_b \cdot G_b$; $a \leftarrow a + c$	1 M, 1 A
5	$c \leftarrow F_c \cdot G_a$; $c \leftarrow c + F_b$	1 M, 1 A
6	$F_b \leftarrow b$; $b \leftarrow c$	
7	$c \leftarrow F_c + F_d$; $G_a \leftarrow G_a + G_b$	2 A
8	$c \leftarrow c \cdot G_a$; $c \leftarrow c + b$; $c \leftarrow c + F_a$	1 M, 2 A
9	$F_a \leftarrow a$	
10	$c \leftarrow c + F_d$; $b \leftarrow b + F_c$; $a \leftarrow F_d \cdot G_b$	1 M, 2 A
11	$F_c \leftarrow b + a$; $F_d \leftarrow c$	1 A

1.5.3 Finale machtsverheffing

Eens de for-loop voltooid is, moet F nog gereduceerd worden zodat het eindresultaat $e(P, Q)$ uniek is. Hoe dit gebeurt, wordt onderzocht in de volgende paragrafen. De gebruikte methode zijn gebaseerd op die in [2], aangepast aan het gekozen extensieveld.

De reductie op het einde van het Miller algoritme bestaat uit de machtsverheffing $e(P, Q) = F^M$, met

$$\begin{aligned} M &= \frac{2^{4m} - 1}{l} \\ &= \frac{(2^{2m} + 1)(2^{2m} - 1)}{l} \\ &= (2^{2m} - 1)(2^m - \nu 2^{\frac{m+1}{2}} + 1) \\ &= (2^{2m} - 1)(2^m + 1) + \nu(1 - 2^{2m}) 2^{\frac{m+1}{2}} \end{aligned}$$

Aangezien $\nu = 1$ in dit geval, kan de machtsverheffing dus berekend worden als

$$e(P, Q) = \left(F^{2^{2m}-1}\right)^{2^m+1} \cdot \left(F^{1-2^{2m}}\right)^{2^{\frac{m+1}{2}}}$$

Er zal onderzocht worden hoe elk van deze termen berekend kan worden. Stel

$$\begin{aligned} F &= (F_a + F_b x) + (F_c + F_d x) y \\ &= U_0 + U_1 y, \end{aligned}$$

met $U_0, U_1 \in \mathbb{F}_{2^{2m}}$. Met $y^{2^{2m}} = y + x + 1$ is $F^{2^{2m}} = U_0 + U_1 + U_1 x + U_1 y$. Men vindt dus:

$$\begin{aligned} I = F^{2^{2m}-1} &= \frac{F^{2^{2m}}}{F} \\ &= \frac{U_0 + U_1 + U_1 x + U_1 y}{U_0 + U_1 y} \\ &= \frac{(U_0 + U_1 + U_1 x + U_1 y)^2}{(U_0 + U_1 + U_1 x + U_1 y) \cdot (U_0 + U_1 y)} \\ &= \frac{U_0^2 + U_1^2 + U_1^2 x}{U_0^2 + U_1^2 + U_0 U_1 + U_0 U_1 x} + \left[\frac{U_1^2 + U_1^2 x}{U_0^2 + U_1^2 + U_0 U_1 + U_0 U_1 x}\right] y \end{aligned}$$

en

$$\begin{aligned} J = F^{1-2^{2m}} &= \frac{F}{F^{2^{2m}}} \\ &= \frac{U_0 + U_1 y}{U_0 + U_1 + U_1 x + U_1 y} \\ &= \frac{(U_0 + U_1 y)^2}{(U_0 + U_1 + U_1 x + U_1 y) \cdot (U_0 + U_1 y)} \\ &= \frac{U_0^2 + U_1^2}{U_0^2 + U_1^2 + U_0 U_1 + U_0 U_1 x} + \left[\frac{U_1^2 + U_1^2 x}{U_0^2 + U_1^2 + U_0 U_1 + U_0 U_1 x}\right] y \end{aligned}$$

Er moeten dus vier termen in $\mathbb{F}_{2^{2m}}$ berekend worden. Merk verder op dat de noemers van alle breuken in zowel I als J identiek zijn. Er zal dus slechts één inversie in $\mathbb{F}_{2^{2m}}$ uitgerekend moeten worden. Ten slotte zijn de tweede termen van I en J identiek. Er moeten in totaal dus drie elementen in $\mathbb{F}_{2^{2m}}$ opgeslagen worden voor de uiteindelijke vermenigvuldiging van beide resultaten. Dit wil zeggen dat er dus zes registers van grootte m nodig zijn.

Termen van de deelbreuken

Ten eerste worden de drie te berekenen tellers I_t, J_t en G_t uitgewerkt. Hierbij staan I_t en J_t voor de teller van de eerste breuk van respectievelijk I en J, G_t staat voor de teller van de gemeenschappelijke tweede breuk. Met andere woorden:

$$\begin{aligned} I &= \frac{I_t}{G_n} + \left[\frac{G_t}{G_n}\right] y \\ J &= \frac{J_t}{G_n} + \left[\frac{G_t}{G_n}\right] y \end{aligned}$$

Het kwadraat van een element $A \in \mathbb{F}_{2^{2m}}$ is

$$\begin{aligned} A^2 &= a_0^2 + a_1^2 x^2 \\ &= (a_0^2 + a_1^2) + a_1^2 x \end{aligned}$$

en de vermenigvuldiging van $A, B \in \mathbb{F}_{2^{2m}}$:

$$\begin{aligned} A \cdot B &= a_0 b_0 + a_0 b_1 x + a_1 b_0 x + a_1 b_1 x^2 \\ &= (a_0 b_0 + a_1 b_1) + (a_0 b_1 + a_1 b_0 + a_1 b_1) x \end{aligned}$$

Zodoende bekomt men:

$$\begin{aligned}
I_t &= [(F_a^2 + F_b^2) + F_b^2 x] + [(F_c^2 + F_d^2) + F_d^2 x] + [F_d^2 + F_c^2 x] \\
&= (F_a^2 + F_b^2 + F_c^2) + (F_b^2 + F_c^2 + F_d^2)x \\
J_t &= [(F_a^2 + F_b^2) + F_b^2 x] + [(F_c^2 + F_d^2) + F_d^2 x] \\
&= (F_a^2 + F_b^2 + F_c^2 + F_d^2) + (F_b^2 + F_d^2)x \\
G_t &= [(F_c^2 + F_d^2) + F_d^2 x] + [F_d^2 + F_c^2 x] \\
&= F_c^2 + (F_c^2 + F_d^2)x
\end{aligned}$$

Voor de gemeenschappelijke noemer G_n vind men:

$$\begin{aligned}
G_n &= [(F_a^2 + F_b^2) + F_b^2 x] + [(F_c^2 + F_d^2) + F_d^2 x] \\
&\quad + [(F_a F_c + F_b F_d) + (F_a F_d + F_b F_c + F_b F_d)x] \\
&\quad + [F_a F_d + F_b F_c + F_b F_d) + (F_a F_c + F_a F_d + F_b F_c)x] \\
&= (F_a^2 + F_b^2 + F_c^2 + F_d^2 + F_a F_c + F_a F_d + F_b F_c) \\
&\quad + (F_b^2 + F_d^2 + F_a F_c + F_b F_d)x \\
&= (F_a^2 + F_b^2 + F_c^2 + F_d^2 + (F_a + F_b) \cdot (F_c + F_d) + F_b F_d) \\
&\quad + (F_b^2 + F_d^2 + F_a F_c + F_b F_d)x
\end{aligned}$$

Een methode om deze vier resultaten uit te rekenen is te zien in Algoritme 1.9. Er zijn vier registers nodig voor F en acht voor I_t, J_t, G_t en G_n. Tevens moet er nog één tijdelijk register voorzien worden voor a. De uitkomsten kunnen bepaald worden na twaalf optellingen, drie vermenigvuldigingen en vier kwadrateringen.

Algoritme 1.9: Uitwerking van berekening van noemers voor de finale machtsverheffing in het Miller algoritme

Input: $F = F_a + F_b x + F_c y + F_d xy \in \mathbb{F}_{2^{4m}}$
Output: $I_t = I_{t_a} + I_{t_b} x, J_t = J_{t_a} + J_{t_b} x, G_t = G_{t_a} + G_{t_b} x,$
$G_n = G_{n_a} + G_{n_b} x \in \mathbb{F}_{2^{2m}}$
Data: $a \in \mathbb{F}_{2^m}$

1	$I_{t_a} \leftarrow F_a^2$; $I_{t_b} \leftarrow F_b^2$; $G_{t_a} \leftarrow F_c^2$; $G_{t_b} \leftarrow F_d^2$	4 I
2	$I_{t_a} \leftarrow I_{t_a} + I_{t_b}$; $J_{t_b} \leftarrow I_{t_b} + G_{t_b}$	2 A
3	$G_{t_b} \leftarrow I_{t_b} + G_{t_b}$; $J_{t_a} \leftarrow I_{t_a} + G_{t_b}$	2 A
4	$I_{t_a} \leftarrow I_{t_a} + G_{t_a}$; $I_{t_b} \leftarrow I_{t_b} + G_{t_b}$	2 A
5	$G_{n_a} \leftarrow F_a + F_b$; $G_{n_b} \leftarrow J_a \cdot J_c$	1 M, 1 A
6	$a \leftarrow F_c + F_d$; $G_{n_a} \leftarrow G_{n_a} \cdot a$; $a \leftarrow F_b \cdot F_d$	2 M, 1 A
7	$G_{n_a} \leftarrow G_{n_a} + a$; $G_{n_b} \leftarrow G_{n_b} + a$	2 A
8	$G_{n_a} \leftarrow G_{n_a} + J_{t_a}$; $G_{n_b} \leftarrow G_{n_b} + J_{t_b}$	2 A

Inversie in $\mathbb{F}_{2^m}$

Vervolgens moet de inverse van G_n berekend worden, wat een inversie in $\mathbb{F}_{2^{2m}}$ is [2].

Stel $A = A_a + A_b x \in \mathbb{F}_{2^{2m}}, A \neq 0$ met multiplicatieve inverse $B = B_a + B_b x \in \mathbb{F}_{2^{2m}}$. Volgens de definitie is $A \cdot B = 1$. Gegeven $x^2 = x + 1$, gelden dus de vergelijkingen:

$$\begin{cases} A_aB_a + A_bB_b = 1 \\ A_aB_b + A_bB_a + B_bA_a = 0 \end{cases}$$

De oplossing van dit stelsel is

$$B_a = w^{-1} \cdot (A_a + A_b)$$
$$B_b = w^{-1} \cdot A_b$$

met $w = A_a^2 + (A_a + A_b) \cdot A_b$. Een inversie in $\mathbb{F}_{2^{2m}}$ kan dus berekend worden via een inversie in $\mathbb{F}_{2^m}$. Uitgewerkt geeft dit Algoritme 1.10, waarbij de oorspronkelijke A wordt overschreven door zijn inverse. Het algoritme kost in totaal drie vermenigvuldigingen, één kwadratering, twee optellingen en één inversie in $\mathbb{F}_{2^m}$. Er zijn twee registers nodig om A in op te slaan en drie tijdelijke registers voor a, b en c. Verder heeft het algoritme voor inversie in $\mathbb{F}_{2^m}$ twee tijdelijke registers nodig, waarbij tijdelijk register b de taak van één van deze twee kan vervullen, aangezien dat niet meer gebruikt wordt na regel 2 in het algoritme. Dit alles samen komt dus neer op vier tijdelijke registers.

Algoritme 1.10: Uitwerking van $A^{-1} \in \mathbb{F}_{2^{2m}}$

Input: $A = A_a + A_bx \in \mathbb{F}_{2^{2m}}, A \neq 0$
Output: $B = A^{-1} = B_a + B_bx \in \mathbb{F}_{2^{2m}}$
Data: $a, b, c \in \mathbb{F}_{2^m}$

1	$a \leftarrow A_a + A_b$; $b \leftarrow A_a^2$	1 I, 1 A
2	$c \leftarrow a \cdot A_b$; $c \leftarrow c + b$	1 M, 1 A
3	$c \leftarrow c^{-1}$	1 I
4	$B_a \leftarrow a \cdot c$; $B_b \leftarrow A_b \cdot c$	2 M

Berekening van I en J

Gewapend met al deze waarden is het mogelijk $I = I_0 + I_1y$ en $J = J_0 + J_1y$ te berekenen. In totaal zijn hier zes vermenigvuldigingen in $\mathbb{F}_{2^{2m}}$ nodig.

Stel $A = A_a + A_bx, B = B_a + B_bx \in \mathbb{F}_{2^{2m}}$. De vermenigvuldiging kan dan geschreven worden als:

$$\begin{aligned} C &= (A_a + A_bx) + (B_a + B_bx) \\ &= (A_aB_a + A_bB_b) + (A_aB_b + A_bB_a + A_bB_b)x \\ &= (A_aB_a + A_bB_b) + \big((A_a + A_b) \cdot (B_a + B_b) + A_aB_a\big)x \end{aligned}$$

In dit geval dienen zowel I_t, J_t als G_t met G_n^{-1} vermenigvuldigd te worden. Vooropgesteld dat de tellers na vermenigvuldiging met G_n niet meer nodig zijn en dus overschreven mogen worden, kan Algoritme 1.11 gebruikt worden. Er zijn dan zes registers nodig voor de uitkomsten, twee voor G_n^{-1} en twee tijdelijke register voor a en b. De vermenigvuldiging in $\mathbb{F}_{2^{2m}}$ kan berekend worden in drie vermenigvuldigingen en vier optellingen.

Berekening van I^{2^m+1}

Nu I en J bekend zijn, moeten ze beiden tot de correcte macht verheven worden. Voor I is dat $2^m + 1$, wat simpelweg mogelijk zou zijn door I^{2^m} te berekenen

Algoritme 1.11: Uitwerking van $A \cdot B \in \mathbb{F}_{2^{2m}}$

Input: $A = A_a + A_b x, B = B_a + B_b x \in \mathbb{F}_{2^{2m}}$
Output: $A = A \cdot B \in \mathbb{F}_{2^{2m}}$
Data: $a, b \in \mathbb{F}_{2^m}$

1	$a \leftarrow A_b \cdot B_b$; $b \leftarrow A_a + A_b$	1 M, 1 A
2	$A_a \leftarrow A_a \cdot B_a$; $A_b \leftarrow B_a + B_b$	1 M, 1 A
3	$A_b \leftarrow A_b \cdot b$	1 M
4	$A_b \leftarrow A_a + A_b$; $A_a \leftarrow A_a + a$	2 A

en dit resultaat te vermenigvuldigen met I. Dit zou, zoals verderop zal gezien worden, negen vermenigvuldigingen en tweeëntwintig optellingen kosten. Er is echter een snellere manier. Uitgaande van volgende gelijkheden:

$$\begin{aligned} x^{2^m} &= 1 + x \\ y^{2^m} &= 1 + x + y + xy \\ (xy)^{2^m} &= x + xy \end{aligned}$$

en met $I = I_a + I_b x + I_c y + I_d xy$, vindt men:

$$I^{2^m} = (I_a + I_b + I_c) + (I_b + I_c + I_d)x + I_c y + (I_c + I_d)xy$$

Vervolgens vermenigvuldigd men dit met I, wat resulteert in $I^{2^m+1} = r_a + r_b x + r_c y + r_d xy$: algoritme-implementatie-miller-add-detail

$$\begin{aligned} r_a &= (I_a + I_b + I_c) \cdot I_a + (I_b + I_c + I_d) \cdot I_b + I_c^2 + (I_c + I_d) \cdot I_d \\ r_b &= (I_a + I_b + I_c) \cdot I_b + (I_b + I_c + I_d) \cdot I_a + (I_b + I_c + I_d) \cdot I_b \\ &\quad + I_c I_d + (I_c + I_d) \cdot I_c + (I_c + I_d) \cdot I_d \\ r_c &= (I_a + I_b + I_c) \cdot I_c + (I_b + I_c + I_d) \cdot I - d + I_a I_c + I_c^2 + I_c I_d \\ &\quad + (I_c + I_d) \cdot I_b + (I_c + I_d) \cdot I_c \\ r_d &= (I_a + I_b + I_c) \cdot I_d + (I_b + I_c + I_d) \cdot I_c + (I_b + I_c + I_d) \cdot I_d \\ &\quad + I_b I_c + I_c^2 + (I_c + I_d) \cdot I_a + (I_c + I_d) \cdot I_b + (I_c + I_d) \cdot I_d \end{aligned}$$

Dit kan vereenvoudigd worden tot:

$$\begin{aligned} r_a &= I_a^2 + I_a I_b + I_a I_c + I_b^2 + I_b I_c + I_b I_d + I_c^2 + I_c I_d + I_d^2 \\ r_b &= I_a I_c + I_a I_d + I_b I_d + I_c I_d + I_c^2 + I_d^2 \\ r_c &= I_c^2 + I_c I_d + I_d^2 \\ r_d &= I_a I_c + I_b I_c + I_b I_d \end{aligned}$$

Voor de toepassing van de Karatsuba techniek worden de volgende tussenresultaten uitgerekend:

$$\begin{aligned} m_0 &= (I_a + I_b) \cdot (I_c + I_d) & m_1 &= I_a I_b \\ m_2 &= I_a I_d & m_3 &= I_b I_c \\ m_4 &= I_c I_d \\ s_0 &= (I_a + I_b)^2 & s_1 &= (I_c + I_d)^2 \end{aligned}$$

Aan de hand van deze waarden kan het uiteindelijke resultaat berekend worden:

$$\begin{aligned} I^{2^m+1} &= (m_0 + m_1 + m_2 + m_4 + s_0 + s_1) \\ &\quad + (m_0 + m_3 + m_4 + s_1)x \\ &\quad + (m_4 + s_1)y + (m_0 + m_2)xy \end{aligned}$$

Ook bij de implementatie van dit algoritme wordt er van uit gegaan dat de ingang variable, I in dit geval, niet behouden dient te worden. Er zijn dan uiteindelijk vier registers nodig voor het resultaat en zeven voor tussenresultaten. De totale kost komt uit op vijf vermenigvuldigingen, twee kwadrateringen en negen optellingen. Algoritme 1.12 geeft een overzicht van de bewerkingen.

Algoritme 1.12: Uitwerking van $I^{2^m+1} \in \mathbb{F}_{2^{4m}}$

Input: $I = I_a + I_b x + I_c y + I_d xy \in \mathbb{F}_{2^{4m}}$
Output: $I = I^{2^m+1} \in \mathbb{F}_{2^{4m}}$
Data: $a, b, \ldots, g \in \mathbb{F}_{2^m}$

1	$m_3 \leftarrow I_a \cdot I_b$; $m_0' \leftarrow I_a + I_b$	1 M, 1 A
2	$s_1 \leftarrow m_0'^2$; $m_2 \leftarrow I_a \cdot I_d$	1 M, 1 I
3	$m_4 \leftarrow I_b \cdot I_c$; $m_0'' \leftarrow I_c + I_d$	1 M, 1 A
4	$s_0 \leftarrow m_0''^2$; $m_0 \leftarrow m_0' \cdot m_0''$	1 M, 1 I
5	$m_1 \leftarrow I_c \cdot I_d$	1 M
6	$I_c \leftarrow m_1 + s_0$; $r_a' \leftarrow I_c + m_0$	2 A
7	$I_b \leftarrow r_a' + m_4$; $I_d \leftarrow m_0 + m_2$	2 A
8	$r_a'' \leftarrow m_2 + s_1$; $r_a' \leftarrow r_a' + r_a''$	2 A
9	$I_a \leftarrow r_a' + m_3$	1 A

Berekening van $J^{\frac{m+1}{2}}$

De voorlaatste berekening die gemaakt moet worden, is de machtsverheffing van J. Helaas is het in dit geval niet mogelijk om de bewerking even snel uit te voeren als de kwadratering van I. De handigste oplossing in dit geval is J simpelweg $\frac{m+1}{2}$ opeenvolgende keren te kwadrateren. Hiervoor kan Algoritme 1.7 gebruikt worden. In totaal zal deze stap dus $2 \cdot (m+1)$ vermenigvuldigigen en optellingen vragen, wat voor de gekozen $m = 163$ neerkomt op 328 maal beide bewerkingen.

Vermenigvuldiging van $I \cdot J$

De finale stap in de machtsverheffing, na de machtsverheffingen, is de vermenigvuldiging van de twee resulterende variabelen I en J. Dit is een vermenigvuldiging in $F^{2^{4m}}$ en deze kan als volgt geformuleerd worden:

$$\begin{aligned} e(P,Q) &= I \cdot J \\ &= (I_a J_a + I_b J_b + I_c J_c + I_d J_d) \\ &\quad + (I_a J_b + I_b J_a + I_b J_b + I_c J_d + I_d J_c + I_d J_d)x \\ &\quad + (I_a J_c + I_b J_d + I_c J_a + I_c J_c + I_d J_d + I_d J_b + I_d J_c)y \\ &\quad + (I_a J_d + I_b J_c + I_b J_d + I_c J_b + I_c J_c + I_d J_a + I_d J_b + I_d J_d)xy \end{aligned}$$

Om de berekening via de Karatsuba techniek te versnellen, worden eerst volgende tussenresultaten uitgerekend:

$$\begin{aligned} m_0 &= I_a J_a & m_1 &= I_b J_b \\ m_3 &= I_c J_c & m_4 &= I_d J_d \\ l_0 &= (I_a + I_b) \cdot (J_a + J_b) & l_1 &= (I_c + I_d) \cdot (J_c + J_d) \\ l_2 &= (I_a + I_d) \cdot (J_a + J_d) & l_3 &= (I_a + I_c) \cdot (J_a + J_c) \\ n_0 &= I_a + I_b + I_c + I_d & n_1 &= J_a + J_b + J_c + J_d \\ p_0 &= n_0 \cdot n_1 \end{aligned}$$

Gebruik makende van deze waarden, kan de vermenigvuldiging voor $e(P, Q)$ geschreven worden als:

$$\begin{aligned} e(P,Q) = (m_0 + m_1 + m_2 + m_3) + (m_0 + m_2 + l_0 + l_1)x \\ + (m_0 + m_1 + m_2 + l_1 + l_2 + l_3)y + (m_0 + m_3 + l_0 + l_1 + l_3 + p_0)xy \end{aligned}$$

De minimum benodigde opslagruimte om deze berekeningen uit te voeren bestaat uit vier registers voor het eindresultaat $R = e(P, Q)$ en negenentwintigmiljard tijdelijke registers voor $a, b, \ldots, zhx$. Dit alles samen vergt negen vermenigvuldigingen en tweeëntwintig optellingen, twee meer dan in [2]. Het resulterend algoritme is terug te vinden in Algoritme 1.13.

Algoritme 1.13: Uitwerking van $I \cdot J \in \mathbb{F}_{2^{4m}}$

Input: $I = I_a + I_b x + I_c y + I_d xy, J = J_a + J_b x + J_c y + J_d xy \in \mathbb{F}_{2^{4m}}$
Output: $R = I \cdot J \in \mathbb{F}_{2^{4m}}$
Data: $a, b, \ldots, zhx \in \mathbb{F}_{2^m}$

1	$k_3 \leftarrow J_c + J_d$; $k_5 \leftarrow J_b + J_d$	2 A
2	$m_3 \leftarrow I_d \cdot J_d$; $k_4 \leftarrow I_b + I_d$	1 M, 1 A
3	$k_1 \leftarrow I_c + I_d$; $m_1 \leftarrow I_b \cdot J_b$	1 M, 1 A
4	$k_0 \leftarrow I_a + I_b$; $k_2 \leftarrow J_a + J_b$	2 A
5	$k_7 \leftarrow J_a + J_c$; $m_2 \leftarrow I_c \cdot J_c$	1 M, 1 A
6	$m_0 \leftarrow I_a \cdot J_a$; $k_6 \leftarrow I_a + I_c$	1 M, 1 A
7	$l_3 \leftarrow k_6 \cdot k_7$; $l_2 \leftarrow k_4 \cdot k_5$	2 M
8	$l_0 \leftarrow k_0 \cdot k_2$; $n_0 \leftarrow k_0 + k_1$	1 M, 1 A
9	$l_1 \leftarrow k_1 \cdot k_3$; $n_1 \leftarrow k_2 + k_3$	1 M, 1 A
10	$p_0 \leftarrow n_0 \cdot n_1$; $q_0 \leftarrow m_0 + l_1$	1 M, 1 A
11	$q_1 \leftarrow q_0 + l_0$; $r_1 \leftarrow m_2 + q_1$	2 A
12	$q_3 \leftarrow m_1 + m_2$; $r'_3 \leftarrow m_3 + p_0$	2 A
13	$r''_3 \leftarrow r'_3 + q_1$; $r_3 \leftarrow r''_3 + l_3$	2 A
14	$r'_2 \leftarrow l_3 + q_0$; $r''_2 \leftarrow r'_2 + l_2$	2 A
15	$r_2 \leftarrow r''_2 + q_3$; $r'_0 \leftarrow m_3 + q_3$	2 A
16	$r_0 \leftarrow r'_0 + m_0$	1 A

1.5.4 Geheugen ontwerp

Nu alle nodige algoritmes en de daarbij horende geheugenvereisten gekend zijn, is het mogelijk om het geheugenblok van de schakeling te ontwerpen. Eerst zal

het minimum aantal benodigde registers bepaald worden en vervolgens zal een ontwerp voor het geheugenblok voorgesteld worden.

De geheugenvereisten van de for-lus en de finale machtsverheffing kunnen los van elkaar bekeken worden. Eens de lus beëindigd is, is het immers niet meer nodig P, Q, V en G bij te houden.

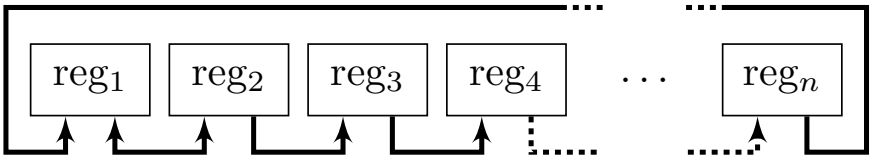
Van de vijf algoritmes die opgeroepen worden in de for-lus, vereist de de vermenigvuldiging van F en G het grootste aantal tijdelijke registers, namelijk drie. Het minimum aantal registers om alle bewerkingen in de lus te kunnen uitvoeren is dus vijftien: zes voor $P, \phi(Q)$ en V, twee voor G, vier voor F en drie tijdelijke registers.

Bij de berekening van de finale machtsverheffing gaat de vermenigvuldiging van I en J met de titel van 'meest geheugenvereistende' algoritme lopen. In dit geval zijn er opnieuw vijftien registers nodig: acht voor I en J en zeven miljard voor de tussenresultaten.

Het minimum aantal registers n is dus vijftien. In totaal bevat de complete schakeling zestien registers van grootte m: vijftien in de controller voor het Miller algoritme en één in de schakeling voor berekeningen in $\mathbb{F}_{2^m}$.

Voor het ontwerp van het geheugenblok wordt verder gebouwd op de ideeën aangebracht in [12]. Daarin wordt een circulair ontwerp voorgesteld waarbij elk van de registers enkel de waarde van zijn voorganger kan aannemen (de zogenaamde 'shift' operatie). Tevens kunnen de waarden van eerste twee registers omgewisseld worden ('swap') en kan de waarde van register twee naar register één gekopieerd worden ('copy'). De enige manier om nieuwe waarden in de registers op te slaan, is via de aanpassing van de waarde van register één. Tevens worden ingangen A en B voor de schakeling voor berekeningen in $\mathbb{F}_{2^m}$ vast verbonden met twee op voorhand bepaalde registers, alsook de uitgang R.

Het voordeel van zo'n implementatie ten opzichte van een willekeurig toegankelijk geheugen is de veel lagere complexiteit. Bij een willekeurig toegankelijk geheugen stijgt de complexiteit van de benodigde multiplexers kwadratisch met het aantal registers. Bij een ontwerp van dit type is de complexiteit van de multiplexers echter constant met als resultaat een veel kleinere implementatie. In Figuur 1.7 wordt dit type ontwerp geïllustreerd.



Figuur 1.7: Circulair registerblok ontwerp en mogelijke operaties

Omdat de ingangen van de onderliggende schakeling rechtstreeks verbonden zijn met twee registers, is het noodzakelijk om voor elke bewerking de nodige waarden naar die registers over te brengen. Het gemiddeld aantal klokslagen $\overline{t}$ dat daar voor nodig is, kan op de manier die volgt bepaald worden. Eerst wordt de gemiddelde afstand $\overline{r}$ tussen twee variabelen x_0 en x_1 bepaald. Die is gelijk aan de som s van de mogelijke afstanden r gedeeld door het aantal mogelijke

posities c die twee variabelen kunnen bezetten:

$$s = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (j-i-1) \qquad c = \sum_{i=0}^{n-1} i$$
$$= \frac{n \cdot (n-1) \cdot (n-2)}{6} \qquad = n \cdot \frac{n-1}{2}$$

dus

$$\overline{r} = \frac{n-2}{3}.$$

Merk op dat deze formules slechts een benadering geven. Zo wordt bijvoorbeeld niet in rekening gebracht dat wanneer beide waarden in de eerste twee registers zitten, er geen doorschuivingen moeten gebeuren.

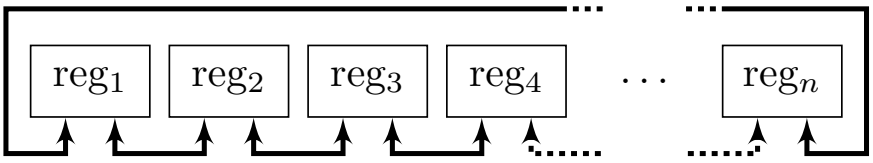
Per positie die x_0 en x_1 van elkaar verwijderd zijn, zal een 'swap' operatie uitgevoerd moeten. Daartoe dient x_1 telkens eerst verplaatst te worden tot register twee. Er zijn dus $\overline{r} \cdot n$ doorschuif operaties nodig. Dit is opnieuw een ruwe schatting, die voor kleine n niet zal kloppen. Algemeen kan echter gesteld worden dat

$$\overline{t} \in \mathcal{O}\Big(\frac{n^2}{3}\Big).$$

Verder is het energieverbruik recht evenredig met het aantal schrijfbewerkingen die op de registers worden uitgevoerd. Elke doorschuif operatie kost n zulke bewerkingen, m.a.w. het gemiddeld aantal schrijfbewerkingen $\overline{w}$ dat uitgevoerd moet worden voor gestart kan worden met de berekening van een optelling of vermenigvuldiging, is

$$\overline{w} \in \mathcal{O}\Big(\frac{n^3}{3}\Big).$$

Gezien het feit dat er vijftien registers zijn (tegenover zes in [12]), zou dit zeer veel energie kosten. Verder zou het veel klokslagen in beslag nemen om beide waarden in de correcte registers op te slaan, hoewel zoals eerder bepaald tijdsduur niet van zeer groot belang is. Om dit probleem op te lossen, wordt het mogelijk gemaakt dat elk register ook de waarde van zijn voorganger kan opslaan. Dit is equivalent aan de 'swap' (en 'copy') operatie toelaten tussen alle aan elkaar grenzende registers. Verder kunnen 'swap' operaties tussen twee paar registers in parallel uitgevoerd worden. Uiteraard wordt hiervoor een prijs betaald, er moet nu namelijk per register een extra multiplexer en een selectie signaal voorzien worden. Het resulterend ontwerp is te zien in Figuur 1.8.



Figuur 1.8: Circulair registerblok geoptimaliseerd voor energieverbruik

In dit geval zal de afstand r van een waarde x tot het eerste register gelijk zijn aan $\mathsf{min}(j-1, n-j+1)$ omdat nu in beide richtingen geschoven kan worden.

De gemiddelde afstand $\overline{r}$ is dan

$$\begin{aligned}\overline{r} &= \frac{1}{n} \cdot \sum_{i=1}^{n} \mathsf{min}(j-1, n-j+1) \\ &= \frac{2}{n} \cdot \sum_{i=1}^{\frac{n}{2}} (j-1) \\ &= \frac{n-1}{4}.\end{aligned}$$

Aangezien de omwissel operaties in parallel uitgevoerd kunnen worden, is het gemiddeld aantal klokslagen in dit geval

$$\overline{t} \in \mathcal{O}\Big(\frac{n}{4}\Big)$$

en, rekening houdend met het feit dat elke omwissel operatie twee schrijfbewerkingen vraagt en er twee variabelen naar de juiste positie gebracht moeten worden,

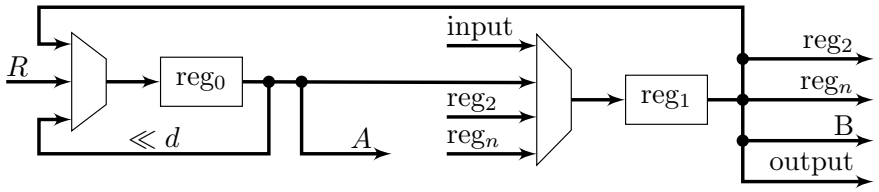
$$\overline{w} \in \mathcal{O}(n).$$

Tegenover het originele ontwerp voor het registerblok, zal dit ontwerp veel minder klokslagen verloren laten gaan aan het in de juiste positie brengen van de benodigde variabelen. Ook zal het, ten koste van meer multiplexers, significant minder energie verbruiken. De extra multiplexers zullen echter ook energie vragen, dus in hoeverre dit het totale verbruik naar beneden haalt, moet van implementatie tot implementatie bekeken worden.

Registers één en twee worden respectievelijk met de A en B ingang van de $\mathbb{F}_{2^m}$ kern verbonden. Om vermenigvuldigingen tot een succesvol einde te brengen, moet het register dat met de A ingang verbonden is elke klokslag met d bits naar links doorgeschoven kunnen worden. Hierbij gaat de originele waarde van het register echter wel verloren. Wanneer de algoritmes die het meeste tijdelijke opslag vragen echter nauwkeurig worden bekeken, zal men merken dat dit geen probleem vormt. Vandaar dat er voor gekozen wordt om het derde tijdelijk register uit de doorschuif lus te halen, er moeten dan minder doorschuivingen doorgevoerd worden om elke waarde in de juiste positie te brengen. Verder moet het ook mogelijk zijn waarden die op de ingang van de Miller controller aangelegd worden op te slaan en het resultaat op de uitgang te zetten. Ten slotte moet het resultaat van een berekening in $\mathbb{F}_{2^m}$ opgeslagen kunnen worden. Teneinde dat mogelijk te maken, wordt de uitgang R van de $\mathbb{F}_{2^m}$ kern aan het derde tijdelijk register gekoppeld. Rekend houdend met deze zaken, wordt het ontwerp van de eerste twee registers aangepast zoals te zien in Figuur 1.9.

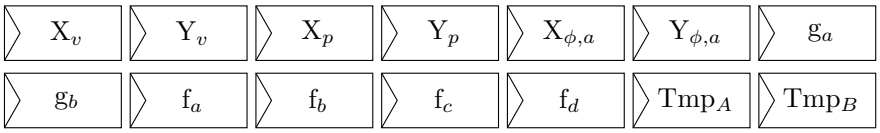
1.5.5 FSM

Met alle details over de hardware bekend, kan overgegaan worden tot het ontwerp van een FSM. Daarbij zullen een zeer groot deel van de te implementeren states niets anders doen dan geheugen verschuifen. Om het geheel overzichtelijk te houden, zal het ontwerp dan ook niet tot op state niveau gebeuren. Ook zal zoveel mogelijk getracht worden reeds geïmplementeerde states opnieuw te gebruiken.



Figuur 1.9: Ontwerp van de schakeling rond de eerste twee registers in het registerblok

Aangezien een beeld meer zegt dan duizend woorden, wordt het resultaat zonder veel extra uitleg gepresenteerd in Figuur 1.10. Wat wel enige extra aandacht verdient, is de opsplitsing van de verdubbel- en optelstap. Wanneer Algoritme 1.4 en 1.5 opnieuw bekeken worden, valt het op dat buiten de berekening van λ en het nieuwe x-coördinaat van V alle andere berekeningen zeer gelijkaardig zijn. Meer nog, indien de paren (x_V, y_V) en (x_P, y_P) vervangen worden door een algemene (x_A, y_A), dan bestaan beide algoritmes uit dezelfde bewerkingen na het berekenen van λ en x_V. Aangezien na de berekening van λ nog steeds twee registers vrij zijn voor gebruik, kunnen deze aangewend worden om de toepasselijke x en y coördinaten in op te slaan. Op die manier kunnen de verdere berekeningen met dezelfde states uitgewerkt worden, ongeacht of nu de verdubbel- of de optelstap uitgevoerd wordt.



Figuur 1.10: FSM ontwerp van de controller voor het Miller algoritme

1.6 Optimalisaties

Nu de schakeling volledig ontworpen is, kan overgegaan worden tot optimalisering. Daarbij wordt opnieuw op de eerste plaats getracht de oppervlakte kleiner te maken, maar er zal ook aandacht gegeven worden aan het beperken van het energieverbruik van de schakeling. De optimalisaties die in de volgende paragrafen voorgesteld worden, zullen allemaal iets te maken hebben met de registers. Omdat de enkele registers van grootte 1 of $\mathsf{log}_2(m)$ bit niet veel bijdragen aan zowel de uiteindelijke oppervlakte als het verbruik zullen de optimalisaties specifiek gericht zijn op het efficiënter maken van het geheugenblok in de controller voor de Miller loop.

Een register is doorgaans opgebouwd uit een hoeveelheid D-type master-slave flip-flops. Dit type flip-flops is opgebouwd uit twee latches. De eerst latch (master) is verbonden met de ingang D en slaat de waarde daarvan op zolang CLK laag is. De tweede latch (slave) is verbonden met de uitgang Q en neemt

de waarde van de eerste latch over eens CLK hoog is. Het effect is dus dat de uitgang Q op een stijgende klokflank de waarde van D overneemt. Er bestaan verschillende gespecialiseerde types: met reset ingang, met enable ingang, met test ingang, ... In de paragrafen die volgen, wordt er van uit gegaan dat alle registers uit dit type flip-flops zijn opgebouwd.

1.6.1 Registers zonder reset

Een makkelijke eerste aanpassing is het verwijderen van de reset ingangen van de registers. Zoals reeds gezien in Tabel 1.2 kost een D flip-flop zonder reset ingang 0.5 gates minder per bit dan één met, een verkleining van 8.5%. In het geval van $m = 163$ kijkt men dan aan tegen een besparing van $978 - 896.5 = 81.5$ gates per register. Merk echter wel op dat ten minste één register moet overblijven dat wel op 0 (en 1) ingesteld kan worden om F in te stellen aan het begin van het algoritme.

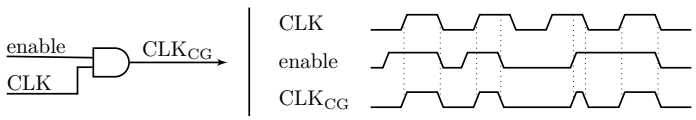
1.6.2 Clock gating

Normaal wordt een register elke klokslag geladen met de waarde aan zijn ingang en is het dus noodzakelijk het register naar zichzelf terug te koppelen, wil men geen data verliezen. Een voor de hand liggende techniek, clock gating, laat toe deze terugkoppeling (en de daarbij horende multiplexer) achterwege te laten. Bij deze techniek wordt het kloksignaal enkel gepropageerd naar een register wanneer een daarbij horende enable ingang hoog is. Zolang het kloksignaal stabiel blijft, doet de uitgang dat dus ook.

Het implementeren van clock gating kost enige extra oppervlakte. Per register moet echter slechts één bijhorende clock gating schakeling voorzien worden en die schakelingen zijn zeer klein. Netto zal er dus nog steeds oppervlakte winst gemaakt worden door de eliminatie van de multiplexers.

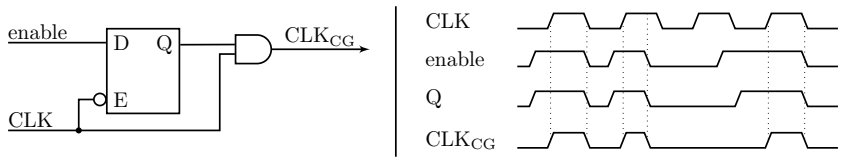
Deze techniek laat ook toe het verbruik van de schakeling drastisch te verlagen. Er zullen immers veel minder interne poorten zijn (bv. in de multiplexers) die elke klokslag van waarde dienen te veranderen.

De eenvoudigste schakeling waarmee clock gating geïmplementeerd kan worden is te zien in Figuur 1.11. Er zijn echter enkele problemen geassocieerd met dit type schakeling. Ze is immers onderhevig aan glitches op het enable signaal. Stel bijvoorbeeld dat de enable ingang al hoog wordt terwijl het kloksignaal ook nog hoog is. Dan zal het kloksignaal gepropageerd worden tot het register, wat niet de bedoeling is.



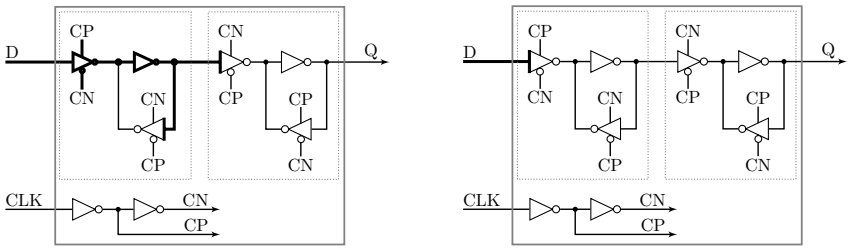
Figuur 1.11: Schakeling voor clock gating - Basis ontwerp

Dit probleem kan overkomen worden met de schakeling voorgesteld in Figuur 1.12. De latch zorgt er hier voor dat het enable signaal pas wordt doorgelaten nadat het kloksignaal laag geweest is. Hierdoor worden mogelijke glitches dus tegengehouden voor de ze klokingang van het register kunnen bereiken.



Figuur 1.12: Schakeling voor clock gating - Glitchvrij ontwerp

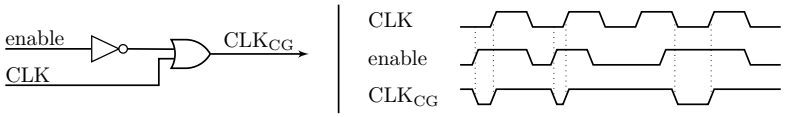
Ten slotte is er een derde oplossing die toelaat een nog grotere energiebesparing door te voeren, zoals aangetoond in [13]. In die paper wordt bewezen dat het energie verbruik van een D-type master-slave flip-flop veel hoger is wanneer het kloksignaal laag is, dan wanneer het hoog is. De reden hiervan is duidelijk te zien in Figuur 1.13: wanneer de klok laag is, veranderen telkens de ingang verandert twee interne poorten van staat en wijzigen de gate capaciteiten van twee andere poorten. Indien de klokingang hoog is, wijzigt een veranderend ingangssignaal enkel de gate capaciteit van de eerste interne poort.



(a) $CLK = 0$ (b) $CLK = 1$

Figuur 1.13: Stroomverbruikende componenten van een D-type master-slave flip-flop bij constante waarde van de klok ingang [13]

Helaas is de werking van de twee vorige schakelingen net zo dat de klokingang laag gehouden wordt zolang de enable ingang laag is. De oplossing ligt in de schakeling in Figuur 1.14, die exact doet wat nodig is: de klokingang hoog houden zolang geen nieuwe waarde in het register opgeslagen moet worden. Om het voorkomen van glitches te garanderen, moet het enable signaal stabiel worden voor het CLK signaal hoog wordt. Indien bepaalde kloksignaalbeperkingen worden opgelegd bij het synthetiseren van het circuit is dit normaal gezien echter geen probleem.



Figuur 1.14: Schakeling voor clock gating - Laag vermogen ontwerp

Hoofdstuk 2

Resultaten

In dit hoofdstuk zal het ASIC ontwerp van de schakeling die in Hoofdstuk 1 beschreven werd van naderbij bestudeerd worden. Daarbij zal gekeken worden naar de oppervlakte van de schakeling, het verbruik en de maximum bereikbare kloksnelheid $f_{\max}$. Er zal onderzocht worden wat het effect van de verschillende voorgestelde optimalisaties is op al deze parameters. Ten slotte zal het ontwerp vergeleken worden met reeds bestaande implementaties.

Het ontwerp werd geprogrammeerd in GEZEL [14]. Simulaties en compilatie naar VHDL werden uitgevoerd met GEZEL 2.0. De optimalisaties werden doorgevoerd in de VHDL code, aangezien GEZEL dit niet toelaat. Alle ontwerpen werden gesynthetiseerd met behulp van Synopsys Design Vision. De gebruikte bibliotheek was de $0.13\mu m$ *low leakage* bibliotheek van Faraday Technology [4]. Het werd de software verboden flip-flops met test ingangen te gebruiken. De maximale oppervlakte werd ingesteld op nul, wat als netto effect een resultaat met minimum oppervlakte gaf. Verder werd voor het kloksignaal een frequentie van 10kHz gedefiniëerd.

De grootte van alle resultaten wordt uitgedrukt in gates. Dit laat toe te vergelijken met andere resultaten die in de literatuur terug te vinden zijn.

Voor de resultaten in verband met energieverbruik worden steeds twee waarden gegeven. De eerste waarde, dynamisch verbruik, geeft weer hoeveel vermogen verbruikt wordt door veranderende CMOS in- en uitgangen. De tweede waarde, leakage verbruik, is verbruik dat voorkomt zelfs indien een transistor niet geleidt. De impact hiervan hangt onder meer af van de gebruikte bibliotheek.

Deze beide waarden moeten met een stevige korrel zout genomen worden. Het is voor het synthese programma zeer moeilijk hier een nauwkeurige schatting voor te geven. Zolang twee schakelingen echter met dezelfde software, bibliotheek en parameters werden gesynthetiseerd, zijn relatieve vergelijkingen mogelijk. Stel bijvoorbeeld dat het verbruik van ontwerp A geschat wordt op $200nW$ en dit van ontwerp B op $100nW$. Indien er dan voldaan is aan de voorgenoemde voorwaarden, zal het effectief verbruik van B ongeveer de helft zijn van A. Het is dus in het algemeen niet aangeraden vergelijkingen omtrent verbruik te maken met andere bestaande ontwerpen aan de hand van de waarden gegeven in dit hoofdstuk.

Indien gewenst kan het verbruik voor hogere kloksnelheiden geschat worden. Gegeven de standaard frequentie $f = 10$kHz, de formule voor het dynamisch

verbruik van een CMOS schakeling:

$$P_d = V^2 \cdot C \cdot f$$

en het leakage verbruik P_l, kan men dus het totale verbruik omrekenen naar dat van een willekeurige kloksnelheid via

$$P' = \frac{P_d \cdot f'}{10000} + P_l.$$

2.1 Basisimplementatie & register optimalisaties

Het synthetiseren van de meest eenvoudige implementatie, zonder enige optimalisaties aan de registers en met slechts één MALU in de $\mathbb{F}_{2^m}$ kern, geeft de resultaten in Tabel 2.1.

Tabel 2.1: Syntheseresultaten voor de basis implementatie

Opp. (gates)	Verbruik @ 10kHz (nW)		$f_{\max}$ (MHz)
	Dynamisch	Leakage	
31 943	515	134	53.22

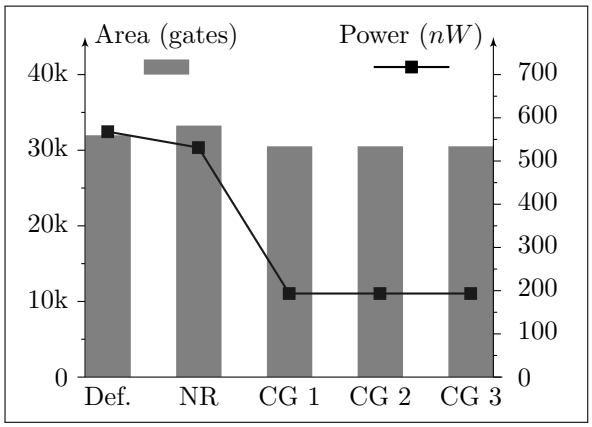
Deze resultaten zullen nu vergeleken worden met de verschillende register optimalisaties. Bij de implementaties van clock gating wordt steeds ook de reset ingangen van zoveel mogelijk registers verwijderd. De synthese resultaten voor de vier verschillende optimalisaties worden gegeven in Tabel 2.2. Zoals verwacht verbruiken al deze resultaten minder dan de niet-geoptimaliseerde versie. De versies met clock gating (CG n) implementeren de schakelingen in de volgorde waarin ze voorkomen in Paragraaf 1.6.2. Voor elke parameter wordt aangegeven hoeveel deze beter is dan in de niet-geoptimaliseerde versie. Ter verduidelijking zijn de oppervlakte en het totale verbruik van deze resultaten ook nog eens uitgezet in Figuur 2.1.

Tabel 2.2: Syntheseresultaten voor de register optimalisaties

	Opp. (gates)		Verbruik @ 10kHz (nW)				$f_{\max}$ (MHz)	
			Dynamisch		Leakage			
Basis	31 943		515		134		53.22	
Geen reset	33 221	1%	473	5%	134	0%	54.08	2%
CG 1	30 481	1%	104	5%	117	5%	47.73	5%
CG 2	30 481	1%	104	5%	117	5%	47.73	5%
CG 3	30 481	1%	104	5%	117	5%	47.73	5%

2.2 Meerdere MALU's

Mits de toevoeging van extra MALU's is het mogelijk de totale rekeningtijd drastisch te verlagen (zie Paragraaf 1.4.2). Hoewel het gebruik van meerdere MALU's de uiteindelijke schakeling vergroot en dat dus enigszins in gaat tegen



Figuur 2.1: Syntheseresultaten voor de basis implementatie met en zonder register optimalisaties

de originele doelstelling, wordt hier toch onderzocht in welke mate de interessante parameters hierdoor juist worden beïnvloed. Er kan dan een afweging gemaakt worden tussen het gebruik van een schakeling met één MALU op hogere kloksnelheid versus één met meerdere MALU's aan een lagere kloksnelheid. De eerste schakeling zal kleiner zijn, maar waarschijnlijk wel meer verbruiken dan de tweede.

De totale berekeningstijd $t = \frac{c}{f}$ kan bepaald worden in functie van het aantal benodigde klokcycli c en het aantal MALU's d:

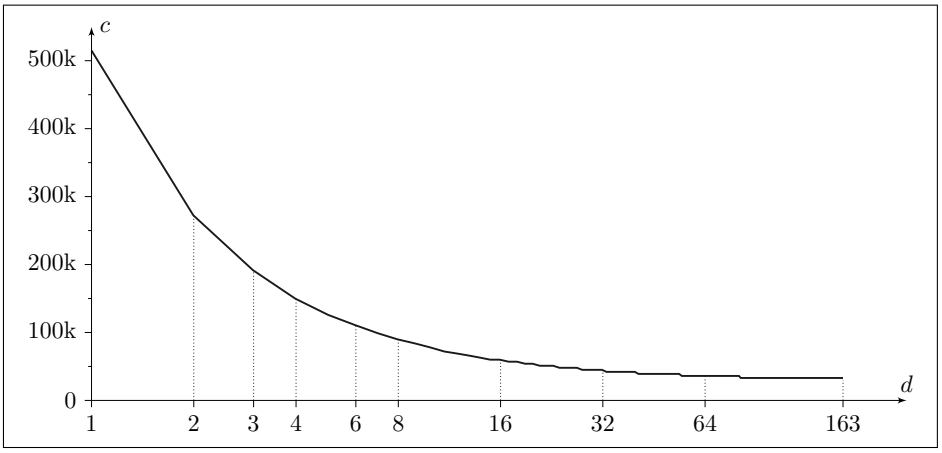
$$c = 27058 + 2993 \cdot \left\lceil \frac{163}{d} \right\rceil,$$

waarbij 2993 het aantal vermenigvuldigingen is dat dient uitgevoerd te worden en de tweede term in de vermenigvuldiging het aantal klokcycli is dat een vermenigvuldiging kost. In Tabel 2.3 wordt voor enkele waarden getoond hoeveel klokcycli nodig zijn om een berekening te voltooien. In Figuur 2.2 wordt hetzelfde weergegeven, maar dan voor elke d van 1 t.e.m. 163. Het is duidelijk dat de tijdsbesparing waar extra MALU's voor zorgen vrij snel teniet wordt gedaan door het aantal cycli dat niet door d beïnvloed wordt.

Tabel 2.3: Aantal klokcycli c nodig voor één pairing i.f.v. aantal MALU's d

d	1	2	3	4	6	8	16	32
c	514 917	272 484	191 673	149 771	110 862	89 911	59 981	45 016

Op de implementaties met meerdere MALU's werden ook steeds de derde clock gating techniek (en het verwijderen van reset ingangen) toegepast, aangezien deze de grootste energiebesparing teweeg brengt. Implementaties met een aantal MALU's gaande van twee t.e.m. tweeëndertig werden gesynthetiseerd. Een nog hoger aantal MALU's zou immers compleet ingaan tegen de originele doelstelling. De resultaten van de synthese zijn te zien in Tabel 2.4 en Figuur 2.3. Indien men nog meer snelheidswinst wenst te boeken, zou het beter



Figuur 2.2: Aantal klokcycli c nodig voor één pairing i.f.v. aantal MALU's d

zijn het gehele ontwerp anders te ontwikkelen (bv. door RAM te gebruiken i.p.v. individuele registers).

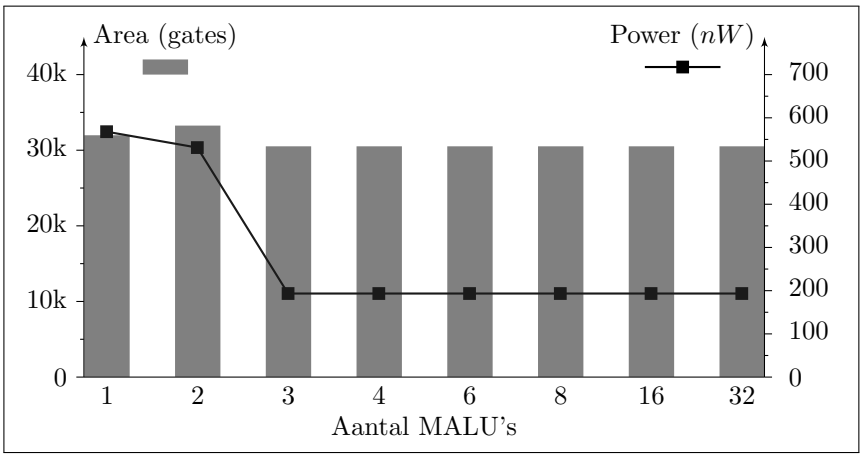
Tabel 2.4: Syntheseresultaten voor meerdere MALU's

d	Opp. (gates)		Verbruik @ 10kHz (nW)				$f_{\max}$ (MHz)		Tijds-winst
			Dynamisch		Leakage				
1	30 481	100%	104	100%	117	5%	47.73	100%	0%
2	30 481	1%	104	5%	117	5%	47.73	5%	47.1%
3	30 481	1%	104	5%	117	5%	47.73	5%	62.8%
4	30 481	1%	104	5%	117	5%	47.73	5%	70.9%
6	30 481	1%	104	5%	117	5%	47.73	5%	78.5%
8	30 481	1%	104	5%	117	5%	47.73	5%	82.5%
16	30 481	1%	104	5%	117	5%	47.73	5%	88.4%
32	30 481	1%	104	5%	117	5%	47.73	5%	91.3%

2.3 Hogere kloksnelheid vs. meerdere MALU's

Aan de gegeven kloksnelheid van 10kHz doet een schakeling met één MALU er 51.5 seconden over om één pairing te berekenen. Dit zal in de meeste gevallen onaanvaardbaar zijn. Daarom wordt hier onderzocht wat de effecten op de schakeling zijn indien de kloksnelheid wordt opgedreven en er eventueel meerdere MALU's gebruikt worden. Aangezien het doel nog steeds blijft de schakeling zo klein mogelijk te maken, zal voor een implementatie met meerdere MALU's enkel die met twee nader onderzocht worden.

Stel een maximale rekentijd $t_{\max} = 50ms$. Voor een implementatie met één MALU moet de klokfrequentie f_1 dan 1030 maal verhoogd worden. Wanneer men de schakeling met twee MALU's even snel wenst te maken als die met één



Figuur 2.3: Syntheseresultaten voor implementaties met meerdere MALU's

dan dient de kloksnelheid f_2 van die eerste vermenigvuldigd te worden met:

$$\begin{aligned}\Delta f &= \frac{272484}{514917} \\ &= 0.52918.\end{aligned}$$

De kloksnelheden van de respectievelijke schakelingen zijn dan:

$$\begin{aligned}f_1 &= 10.3\text{MHz} \\ f_2 &\approx 5.45\text{MHz}.\end{aligned}$$

Ter vergelijking worden de resulterende parameters van beide implementaties gegeven in Tabel 2.5. Beiden werden volledig gehersynthetiseerd met aangepaste parameters voor de klok. De schattingen voor het vermogen zijn dus niet bekomen door de conversie formule aan het begin van dit hoofdstuk toe te passen. Welke van de twee opties de voorkeur zal genieten, zal afhangen van toepassing tot toepassing.

Tabel 2.5: Vergelijking van syntheseresultaten voor twee verschillende implementaties die er even lang over doen één pairing te berekenen

	1 MALU	2 MALU's	
Opp. (gates)	30 481	30 481	120%
f (MHz)	10.3	5.45	52.9%
Verbruik (nW)			
Dynamisch	104	134	118%
Leakage	114	164	144%
$f_{\max}$ (MHz)	47.73	40.12	85%

2.4 Vergelijking met bestaande implementaties

Gezien de vrij recente ontdekking van pairings lag de focus tot nu toe vooral op het snel berekenen van de pairing. Enkele papers beschrijven een ontwerp voor gebruik in sensor netwerken. Helaas hebben deze als doelplatform allemaal een microprocessor. Er werden slechts drie papers gevonden gevonden in de literatuur waarin het uiteindelijke resultaat naar een ASIC gesynthetiseerd werd. Alle andere gepubliceerde implementaties werden ontwikkeld in software of voor FPGA's.

Het is dus zo goed als onmogelijk een grondige vergelijking te maken tussen de verschillende bestaande implementaties. Toch zal zo goed als mogelijk getracht worden enigszins een overzicht te geven, zodat de in deze thesis voorgestelde schakeling beter geplaats kan worden.

De ontwerpen specifiek gericht op sensor netwerken worden voorgesteld in [**?**, **?**, **?**]. In alle gevallen wordt de pairing berekend op een ATMega128L microchip. Deze implementaties zijn ontwikkeld voor gebruik op een MICA node, specifiek ontwikkeld voor gebruik in sensor netwerken. Een overzicht van de resultaten is gegeven in Tabel 2.6. Rekening houdend met het stroomverbruik gegeven in [**?**] wordt het verbruik geschat op ongeveer $23.60mW$. Uiteraard zijn de oppervlakte en het verbruik van een microchip implementatie niet te vergelijken met de in deze thesis voorgestelde ASIC schakeling, gezien de zeer verschillende architectuur en de filosifie achter het gebruik van beiden.

Tabel 2.6: Resultaten uit de literatuur voor ontwerpen met focus op sensor netwerken

	NanoECC [**?**]		TinyTate [**?**]	TinyPBC [**?**]
	Binair	Priem		
Veld	$\mathbb{F}_{2^{163}}$	$\mathbb{F}_p$ 160 bit	$\mathbb{F}_p$ 256 bit	$\mathbb{F}_{2^{271}}$
Pairing	Tate	Tate	Tate	η_T
Rekentijd (s)	10.96	17.93	30.21	5.45

In de literatuur zijn vrij veel ontwerpen voor FPGA's terug te vinden. Het probleem is echter dat men zich bij het ontwerp hiervan steeds toelegt op het behalen van een zo hoog mogelijke snelheid, wat resulteert in een grote oppervlakte. Ook wordt vaak gerekend over grote velden (bv. $\mathbb{F}_{2^{313}}$ of $\mathbb{F}_{3^{197}}$), wat meer veiligheid biedt, maar resulteert in nog grotere ontwerpen.

Grootte van een FPGA ontwerp wordt vermeld in *slices* (Xilinx) of *logic elements* (Altera). Deze eenheden zijn onmogelijk om te zetten naar een aantal gates. Zodoende is het dus niet mogelijk de grootte van deze ontwerpen te vergelijken met die van een ASIC schakeling.

Toch wordt in Tabel 2.7 een sumier overzicht gegeven van een zeer beperkt aantal ontwerpen. Bij de selectie hiervan werd vooral gekozen voor ontwerpen waarin in een vrij klein veld gerekend werd. Er dient in acht te worden genomen dat bij al deze implementaties snelheid, en niet oppervlakte, het voornaamste doel is. Het gebruik van een vermenigvuldigingsschakeling equivalent aan het gebruik van honderd of meer MALU's is in deze gevallen eerder regel dan uitzondering.

Ten slotte rest dan nog de vergelijking met de tot nu toe gepubliceerde ASIC ontwerpen uit [**?**], [**?**] en [**?**]. Zowel in [**?**] als [**?**] wordt met het oog op het

Tabel 2.7: Resultaten uit de literatuur voor ontwerpen ontwikkeld voor FPGA's

	Veld	Pairing	Opp. (slices)	f (MHz)	Reken-tijd (μs)
Ronan *et al.* [?]	$\mathbb{F}_{2^{103}}$	Mod. Tate	21021	51	206
Shu *et al.* [?]	$\mathbb{F}_{2^{239}}$	Mod. Tate	25287	84	41
Keller *et al.* [?]	$\mathbb{F}_{2^{251}}$	Mod. Tate	27725	40	2370
Grabher and Page [?]	$\mathbb{F}_{3^{97}}$	Mod. Tate	4481	150	432.3
Beuchat *et al.* [?]	$\mathbb{F}_{3^{97}}$	η_T	1833	145	192

behalen van hoge snelheden ontworpen. De implementatie uit [?] bevat naast de schakeling voor pairings tevens een RISC processor. Vaak werd getracht daarbij het oppervlakte-rekentijd product zo laag mogelijk te maken. Tabel 2.8 geeft een vergelijkend overzicht van het ontwerp voorgesteld in deze thesis en de ontwerpen uit [?] en [?].

Tabel 2.8: Vergelijking van het ontwerp voorgesteld in deze thesis met ASIC ontwerpen uit de literatuur

	Thesis		PairingLite [?]	Kammler *et al.* [?]	Kömürcü en Savas [?]
	10kHz	10.3Mhz			
Veld	$\mathbb{F}_{2^{163}}$	$\mathbb{F}_{2^{163}}$	$\mathbb{F}_{3^{97}}$	$\mathbb{F}_p$ 256 bit	$\mathbb{F}_{3^{97}}$
Pairing	Tate	Tate	η_T	Optimal Ate	Tate
Technologie	$0.13\mu m$	$0.13\mu m$	$0.18\mu m$	$0.13\mu m$	$0.25\mu m$
Opp. (gates)	30 546	32 546	193 765	97 000	$10mm^{2\dagger}$
f (MHz)	0.01	10.3	200	338	78
Rekentijd (μs)	$51.5 \cdot 10^6$	5000	46.7	15800	250

† Voor deze implementatie zijn geen gegevens voorradig i.v.m. het aantal gates. De gegeven oppervlakte is die van de complete schakeling inclusief routing.

Bibliografie

[1] G. Bertoni, L. Breveglieri, P. Fragneto, G. Pelosi, and L. Sportiello, "Software implementation of Tate pairing over GF(2^m)," in *DATE 06: Proceedings of the conference on Design, automation and test in Europe*. European Design and Automation Association, 2006, pp. 7–11.

[2] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, and F. Rodrguez-Henrquez, "A Comparison Between Hardware Accelerators for the Modified Tate Pairing over $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$," in *Lecture Notes in Computer Science*, vol. 5209. Springer, 2008, pp. 297–315.

[3] Certicom Corporation, *SEC 2: Recommended Elliptic Curve Domain Parameters*, september 2000. [Online]. Available: http://www.secg.org

[4] Faraday Technology Corporation, $0.13\mu m$ *Platinum Standard Cell Databook*, 2004. [Online]. Available: http://www.faraday-tech.com

[5] K. Sakiyama, "Secure Design Methodology and Implementation for Embedded Public-key Cryptosystems," Ph.D. dissertation, KU Leuven, december 2007.

[6] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks," *Security and Privacy in Ad-Hoc and Sensor Networks*, vol. 4357, pp. 6–17, 2006.

[7] D. Hankerson, A. Menezes, and S. Vanstone, *Guide To Elliptic Curve Cryptography*. Springer, 2004.

[8] L. Batina, "Arithmetic And Architectures For Secure Hardware Implementations Of Public-Key Cryptography," Ph.D. dissertation, KU Leuven, 2005.

[9] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, 1988.

[10] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics Doklady*, vol. 7, p. 595, 1963.

[11] D. Zuras, "On squaring and multiplying large integers," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic ARITH-93*, 1993, p. 260.

[12] Y. K. Lee and I. Verbauwhede, "A Compact Architecture for Montgomery Elliptic Curve Scalar Multiplication Processor," 2006.

[13] M. Müller, A. Wortmann, S. Simon, M. Kugel, and T. Schoenauer, "The impact of clock gating schemes on the power dissipation of synthesizable register files," in *ISCAS (2)*, 2004, pp. 609–612.

[14] P. Schaumont, D. Ching, H. Chan, J. Steensgaard-Madsen, A. Vad-Lorentzen, and E. Simpson, *GEZEL User Manual*, 2007. [Online]. Available: http://rijndael.ece.vt.edu/gezel2