2014•2015
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef
Een bibliotheek van cryptografische operaties met Lava

Promotor :
dr. Kris AERTS
Prof. dr. Nele MENTENS

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

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

2014•2015
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterproef

Een bibliotheek van cryptografische operaties met Lava

Promotor :
dr. Kris AERTS
Prof. dr. Nele MENTENS

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

universiteit hasselt | KU LEUVEN

Woord vooraf

Dit werk is gemaakt voor onderzoeksgroep ES&S. Ik heb hiervoor gekozen omdat ik zeer geïnteresseerd ben in cryptografie en de wiskunde die hier achter zit.

Met dit voorwoord wil ik ook de kans nemen om een aantal personen te bedanken. Allereerst dank ik graag mijn promotoren dr. Kris Aerts en dr. ir. Nele Mentens voor de kans om aan dit project te kunnen werken en voor de hulp tijdens het werk. Ook wil ik mijn moeder bedanken voor het ondersteunen van mijn studies. Als laatste wil ik mijn vriendin, Lene, bedanken. Zonder haar steun was dit werk nooit tot een goed einde gekomen.

Bartel Sielski
Augustus 2015

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Abstract

De onderzoeksgroep Embedded Systems & Security (ES&S) werkt momenteel aan een EDA-DSE tool waarmee cryptografische hardware gegenereerd kan worden. De tool is geschreven m.b.v. York Lava, een bibliotheek voor het ontwikkelen van digitale circuits in Haskell. Het doel van de tool is de implementatie van cryptografische functies in hardware vereenvoudigen.

In dit werk wordt een implementatie voor elliptische curve cryptografie (ECC) afgewerkt. Hiervoor is een module opgebouwd die Elliptic Curve Diffie-Hellman (ECDH) en Elliptic Curve Digital Signature Algorithm (ECDSA) afhandeld.

Hiervoor werd een elliptische kromme processor opgebouwd uit bestaande modules voor de Montgomery modulaire vermenigvuldiging, modulaire optelling/aftrekking en projectieve naar affine coördinaat omzettingen. Deze zijn uitgebreid met een EC punt vermenigvuldigingsmodule en de bestaande EC puntoptelling/-verdubbeling werd verbeterd.

Hiernaast is ook een SHA-256 hash module aangemaakt. Deze wordt samen met de EC processor aangestuurd door een ECDH/ECDSA module die de mogelijkheid biedt om digitale handtekeningen te genereren en te controleren.

Abstract in English

The research group Embedded Systems & Security (ES&S) is currently developing an EDA-DSE tool which can be used to automatically generate cryptographic hardware. The tool is made using York Lava, a library for developing digital circuits in Haskell. The development on the tool started in order to simplify the implementation of cryptographic functions in hardware.

This project continues the work done on an implementation of Elliptic Curve Cryptography (ECC). The completed module processes the calculations needed in Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA).

The completed module consists of an elliptic curve processor, which is made up of the already existing Montgomery modular multiplication, modular addition/-subtraction and projective/affine coordinate conversion modules. A new EC point multiplication module was added and the existing EC point addition/doubling module has been improved.

Message integrity is maintained with an implementation of the SHA-256 hash function. Both the hash function and EC processor are controlled by the ECDH/-ECDSA module which generates public/private key pairs and generates and verifies digital signatures.

Hoofdstuk 1

Inleiding

1.1 Doelstelling

De groeiende vraag naar robuuste cryptografische algoritmes legt een grote eis aan de ontwerpers van hardware. Om makkelijker aan deze eis te voldoen tracht dit project de implementatie van dergelijke algoritmes in hardware te vereenvoudigen. Hiervoor wordt een bibliotheek toegankelijk gesteld met uitgewerkte modules voor bepaalde cryptografische functies.

De module die in dit project uitgewerkt wordt, maakt gebruik van elliptische kromme cryptografie.[1] Het uiteindelijke geheel moet veilige communicatie mogelijk maken d.m.v. ECDH (of Elliptic Curve Diffie-Hellman) en ECDSA (of Elliptic Curve Digital Signature Algorithm).

1.2 Materiaal

Het project zal gerealiseerd worden in York Lava, een bibliotheek voor het beschrijven van digitale circuits in Haskell. Deze taal zal uitgebreid worden met de EDA-DSE tool ontwikkeld door ES&S. De EDA-DSE tool geeft de eindgebruiker de mogelijkheid verschillende implementaties van de ontwikkelde code te vergelijken. Hierdoor kan de meest geschikte implementatie gekozen worden voor het uiteindelijke project.

1.3 Aanpak

Het project kan opgedeeld worden in 3 delen: de elliptische kromme processor, de hashmodule en de ECC module die de 2 andere delen bestuurd.

De elliptische kromme processor verwerkt de puntvermenigvuldiging en puntoptelling. Deze bewerkingen zijn nodig bij het genereren en controleren van de digitale handtekening volgens ECDSA. Hiernaast wordt de processor gebruikt bij het genereren van de publieke sleutel voor zowel ECDSA en ECDH. De processor is opgebouwd uit de modules gemaakt in [3]. Bij controle van deze modules bleek de puntoptelling niet te werken. Om deze fout te corrigeren zal de puntoptelling, samen met de puntverdubbeling, herwerkt worden tot één module. Hierdoor zal de uiteindelijke

[1] *Elliptic Curve Cryptography* of ECC

schakeling kleiner zijn. De submodules van de EC processor worden bestuurd door de controller die tijdens dit project werd gemaakt.

De hashmodule wordt gebruikt om de input van het handtekeningalgoritme om te vormen tot een handelbare grootte. Hiervoor wordt een SHA-256 module aangemaakt, een cryptografische hash standaard. Van de hashmodule zijn bij de start van het project nog geen onderdelen aangemaakt. De volledige werking en implementatie wordt in hoofdstuk 3 besproken.

Als laatste deel wordt de ECC module gemaakt die, m.b.v. de EC processor en de hashfunctie, de sleutels genereert voor ECDSA en ECDH alsook de ECDSA handtekeningen genereert en controleert.

Het afgewerkte geheel kan nog niet omgezet worden tot een werkende netlist. Dit is te wijten aan 2 zaken. Allereerst ontbreekt er nog een random nummer generator. Ten tweede werkt de omzetting van Lava code naar VHDL nog niet afdoende. Deze problemen zullen opgelost moeten worden in een volgend project.

Hoofdstuk 2

Elliptische kromme cryptografie

Elliptische kromme cryptografie is een vorm van asymmetrische cryptografie. Ten opzichte van andere asymmetrische algoritmes is de sleutellengte van ECC veel kleiner terwijl toch dezelfde veiligheid gegarandeerd is (zie tabel 2.1). Dit maakt het mogelijk om efficiënt een boodschap digitaal te onderteken. Hierbij moet voldaan worden aan een aantal eisen:

Authenticatie De geheime sleutel van een asymmetrisch algoritme is slechts toegankelijk voor één gebruiker. Hierdoor kunnen boodschappen ondertekend met deze sleutel alleen verzonden zijn door deze gebruiker.

Integriteit Tijdens communicatie tussen twee partijen is het belangrijk dat de ontvanger weet dat de boodschap niet aangepast is tijdens de verzending. Om aan deze eis te voldoen wordt bij digitale handtekeningen gebruikt gemaakt van een cryptografische hash functie. Deze bewaart de integriteit van een boodschap terwijl de grootte van de te ondertekenen data hanteerbaar blijft.

Onweerlegbaarheid De beperkte toegang tot een gebruiker's geheime sleutel maakt het onmogelijk voor een partij om te ontkennen dat deze een bepaalde boodschap, met een geldige handtekening, verstuurd heeft. Hierdoor kan bijvoorbeeld een contract digitaal ondertekend worden zonder dat de betrokken partijen later kunnen ontkennen dat ze het contract getekend hebben.

Tabel 2.1: Sleutel lengtes van algoritmes bij verschillende veiligheidsniveaus

Algoritme familie	Cryptosysteem	Veiligheidsniveau (bit)			
		80	128	192	256
Ontbinden in priemfactoren	RSA	1024	3072	7680	15360
Discrete logaritme probleem	DH, DSA, Elgamal	1024	3072	7680	15360
Elliptische krommen	ECDH, ECDSA	160	256	384	512
Symmetrische encryptie	AES, 3DES	80	128	192	256

2.1 Theorie

ECC is gebaseerd op de volgende vergelijking:

$$y^2 = x^3 + ax + b \mod p \tag{2.1}$$

waar p een priemgetal, groter dan 3 is en waarvoor $4a^3+27b^2 \neq 0 \mod p$. De punten die voldoen aan vergelijking (2.1), samen met een imaginair punt op oneindig en de groepsbewerkingen, vormen een cyclische groep. De extra voorwaarde voor a en b is nodig om krommen te vermijden die singulier zijn.

2.1.1 Groepsbewerkingen

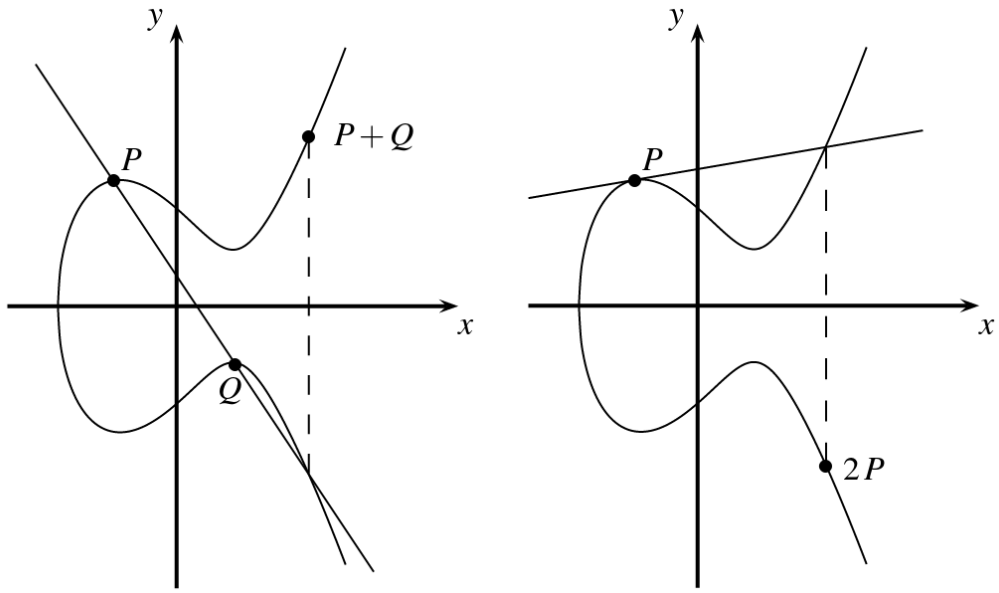
De groepsbewerking bij elliptische krommen wordt aangeduid met een plusteken. Hierdoor is, voor $P = (x_1, y_1)$ en $Q = (x_2, y_2)$:

$$P + Q = R$$
$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

Deze bewerking kan dan opgesplitst worden in twee gevallen:

Punt optelling $P + Q$: in dit geval is R het inverse van het derde snijpunt van de rechte door P en Q en de elliptische kromme. In figuur 2.1a is dit grafisch weergegeven.

Punt verdubbeling $P + P$: hier is R het inverse van het tweede snijpunt van de raaklijn aan de elliptische kromme in P. Dit is grafisch weergegeven in figuur 2.1b.



(a) Puntoptelling (b) Puntverdubbeling

Figuur 2.1: ECC groepsbewerkingen [8]

Wiskundig kan de groepsbewerking als volgt weergegeven worden:

$$x_3 = s^2 - x_1 - x_2 \mod p$$
$$y_3 = s(x_1 - x_3) - y_1 \mod p$$

met

$$s = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \mod p;\ \text{als } (x_1, y_1) \neq (x_2, y_2)\ \text{(puntoptelling)} \\ \dfrac{3x_1^2 + a}{2y_1} & \mod p;\ \text{als } (x_1, y_1) = (x_2, y_2)\ \text{(puntverdubbeling)} \end{cases}$$

Uit de puntoptelling en puntverdubbeling kan de puntvermenigvuldiging afgeleid worden. Dit wordt voorgesteld als $Q = dP$, met P een punt op de kromme en d een natuurlijk getal. Het punt P wordt hier d keer bij zichzelf opgeteld.

$$\begin{aligned} P + P &= 2P \\ 2P + P &= 3P \\ 3P + P &= 4P \\ &\vdots \\ (q-1)P + P &= qP = \mathcal{O} \\ qP + P &= \mathcal{O} + P = P \end{aligned}$$

Uit bovenstaande vergelijkingen is te zien dat, door een punt q keer bij zichzelf op te tellen, het punt $\mathcal{O}$ op oneindig bekomen wordt. Dit punt is het neutraal element voor de puntoptelling. Alle punten die een meervoud zijn van P vormen een cyclische groep van orde q waarvan P een primitief element[1] is.

Om een efficiënte implementatie te bekomen voor de puntvermenigvuldiging is herhaaldelijk optellen met P echter niet geschikt. Dit kan verbeterd worden door gebruik te maken van het *Double-and-Add* principe. Als dan een punt P met bijvoorbeeld 26 vermenigvuldigd wordt gebeurt dit in de volgende stappen.

1. P wordt verdubbeld en het resultaat wordt opgeteld bij P. Hierdoor wordt $3P$ bekomen.
2. $3P$ wordt verdubbeld om $6P$ te bekomen.
3. $6P$ wordt verdubbeld en vervolgens opgeteld bij P. Dit is gelijk aan $13P$.
4. $13P$ wordt verdubbeld om de gewenste waarde $26P$ te bekomen.

Om te weten welke bewerkingen moeten gebeuren bij elke stap, kijken we naar de binaire voorstelling van 26 namelijk 11010. De meest beduidende bit wordt weggelaten en de overgebleven bits worden van meest beduidend naar minst beduidend afgelopen. In het geval van een 0 moet er enkel een verdubbeling gebeuren en bij een 1 gebeurt zowel een verdubbeling als een optelling.

Omdat deze stappen van de puntvermenigvuldiging makkelijk uit te voeren zijn en niet makkelijk omgekeerd kunnen worden, kan hieruit kan een cryptosysteem opgebouwd worden.

[1] Een primitief element wordt soms ook een generator genoemd.

2.1.2 ECDH en ECDSA

ECDH

ECDH is een manier om, met behulp van een elliptische kromme, Diffie-Hellman-sleuteluitwisseling te doen [5]. Met behulp van Diffie-Hellman kunnen 2 partijen via een open kanaal, dat afgeluisterd kan worden, een sleutel afspreken waar enkel zij toegang tot hebben [2]. Bij ECDH gebeurt dit in 5 stappen:

1. De partijen spreken een kromme en een primitief element P op deze kromme af.

2. Vervolgens genereren de partijen A en B elk een geheime sleutel, a en b respectievelijk.

3. Hierna berekenen de partijen de publieke sleutels $P_A = a \cdot P$ en $P_B = b \cdot P$.

4. De publieke sleutels worden uitgewisseld.

5. Uiteindelijk berekenen beide partijen $Q = a \cdot P_B$ en $Q = b \cdot P_A$.

De uiteindelijke waarde Q is hetzelfde voor beide partijen omdat

$$Q = a \cdot (b \cdot P) = b \cdot (a \cdot P) = (a \cdot b) \cdot P$$

Deze waarde is enkel gekend door partij A en B. Een derde partij C, die het kanaal afluistert, kent enkel P, P_A en P_B en kan hieruit Q niet berekenen.

ECDSA

ECDSA is een algoritme dat gebruikt wordt voor het ondertekenen van boodschappen en voor het controleren van handtekeningen [4]. Het algoritme kan opgedeeld worden in drie delen:

Sleutel generatie:

1. Gebruik een elliptische kromme E met:

   - modulus p,
   - coëfficiënten a en b,
   - een punt A dat een cyclische groep met priem orde q genereert.

2. Genereer een willekeurig natuurlijk getal d met $0 < d < q$.

3. Bereken $B = dA$

d is de geheime sleutel en de publieke sleutel is (p, a, b, q, A, B).

Handtekening generatie:

1. Genereer een tijdelijke sleutel k_E met $0 < k_E < q$.
2. Bereken $R = k_E \cdot A$
3. Bereken $s = (m + d \cdot r) \cdot k_E^{-1} \mod q$
   met m de te ondertekenen boodschap en r de x-coördinaat van R.

De ondertekende boodschap is $(m, (r, s))$ met m de originele boodschap en (r, s) de handtekening. Het is belangrijk dat er nooit 2 verschillende boodschappen met dezelfde tijdelijke sleutel k_E ondertekend worden. Als dit wel gebeurt, is het zeer eenvoudig om, met deze handtekeningen, de geheime sleutel d te berekenen. Dit wordt bewezen in bijlage A.

Handtekening controle maakt gebruik van de publieke sleutel (p, a, b, q, A, B) uit de sleutel generatie en de ondertekende boodschap $(m, (r, s))$ uit de handtekening generatie.

1. Bereken $w = s^{-1} \mod q$
2. Bereken $u_1 = w \cdot m \mod q$
3. Bereken $u_2 = w \cdot r \mod q$
4. Bereken $P = u_1 A + u_2 B$

De handtekening is geldig als de x-coördinaat van P gelijk is aan r. Dit kan bewezen worden door middel van onderstaande vergelijkingen.

$$\begin{aligned} s &= (m + d \cdot r) \cdot k_E^{-1} \mod q \\ \Rightarrow k_E &= m \cdot s^{-1} + d \cdot r \cdot s^{-1} \mod q \\ &= u_1 + u_2 \cdot d \mod q \\ k_E \cdot A &= (u_1 + u_2 \cdot d) \cdot A \\ &= u_1 \cdot A + u_2 \cdot d \cdot A \\ &= u_1 \cdot A + u_2 \cdot B \\ &= P \end{aligned}$$

2.2 Verbetering van de puntbewerkingen

De meest tijdrovende bewerking nodig bij het verwerken van de ECDH en ECDSA algoritmes is de modulaire vermenigvuldiging. Om deze bewerking efficiënter te laten verlopen maakt dit project gebruik van een methode ontwikkeld door Montgomery in [6]. Montgomery stelt de volgende bewerking voor:

$$Mont(x, y) = x \cdot y \cdot R^{-1} \mod n$$

R is de modulus waarbinnen liefst gewerkt wordt. Voor binaire bewerkingen is dit een macht van 2 omdat het reduceren van een product dan zeer eenvoudig wordt. Deze Montgomery bewerking heeft een paar eigenschappen die het interessant maken als vervanging voor de modulaire vermenigvuldiging.

$$Mont(x, R^2) = x \cdot R^2 \cdot R^{-1} = xR \quad \text{mod } n \tag{2.2}$$
$$Mont(xR, 1) = xR \cdot R^{-1} = x \quad \text{mod } n \tag{2.3}$$
$$Mont(xR, yR) = xR \cdot yR \cdot R^{-1} = (x \cdot y)R \quad \text{mod } n \tag{2.4}$$
$$xR + yR = (x + y)R \tag{2.5}$$

Vergelijking 2.2 geeft de omzetting van een waarde naar zijn Montgomery equivalent weer. Deze Montgomery waardes kunnen dan gebruikt worden volgens 2.4 om de Montgomery waarde van het product te bekomen. Ook kunnen 2 Montgomery waardes opgeteld worden om zo de Montgomery waarde van de som te bekomen zoals getoond in vergelijking 2.5. Deze Montgomery waarden kunnen dan door middel van vergelijking 2.3 omgezet worden naar de normale waarde.

Één Montgomery bewerking is uitgebreider dan simpelweg eenmaal een modulair product berekenen omdat er een aantal voorbewerkingen moeten gebeuren. Deze voorbewerkingen zijn echter enkel afhankelijk van de modulus n. Hierdoor wordt de Montgomery bewerking efficiënter als er meerdere vermenigvuldigingen moeten gebeuren met dezelfde modulus. Dit maakt het zeer geschikt voor gebruik bij elliptisch kromme cryptografie.

Een tweede bewerking die zeer tijdrovend is, is de deling modulus p die nodig is bij de puntoptelling en puntverdubbeling. Om de deling $x/y \mod p$ te berekenen, moet eerst het multiplicatief inverse van y berekend worden. Om te voorkomen dat er tijdens elke puntbewerking een deling moet gebeuren wordt het ingevoerde punt omgezet in een aangepast Jacobiaans coördinaatstelsel $\mathcal{J}^m$ zoals beschreven in [1]. Een punt (x, y) wordt in dit coördinaatstelsel voorgesteld als

$$(x, y) \equiv (X, Y, Z, aZ^4) \tag{2.6}$$
$$\text{met } x = \frac{X}{Z^2}$$
$$y = \frac{Y}{Z^3}$$
$$a = \text{de } a \text{ parameter van de gebruikte elliptische kromme}$$

Een punt kan zeer eenvoudig omgezet worden naar $\mathcal{J}^m$ door Z gelijk te stellen aan 1. Hierdoor worden X, Y en aZ^4 gelijk aan x, y en a respectievelijk.

De puntoptelling van $P = (X_1, Y_1, Z_1, aZ_1^4)$, $Q = (X_2, Y_2, Z_2, aZ_2^4)$ en $P + Q =$

(X_3, Y_3, Z_3, aZ_3^4) is dan:

$$X_3 = -H^3 - 2U_1H^2 + R^2 \qquad Y_3 = -S_1H^3 + R(U_1H^2 - X3)$$
$$Z_3 = Z_1Z_2H \qquad aZ_3^4 = a \cdot Z_3^4$$

met

$$U_1 = X_1Z_2^2 \qquad U_2 = X_2Z_1^2$$
$$S_1 = Y_1Z_2^3 \qquad S_2 = Y_2Z_1^3$$
$$H = U_2 - U_1 \qquad R = S_2 - S_1$$

Deze bewerkingen worden bij de implementatie nog verder vereenvoudigd omdat ervan uitgegaan wordt dat $Q = (x_2, y_2, 1, a)$.

De puntverdubbeling van $P = (X_1, Y_1, Z_1, aZ_1^4)$ en $2P = (X_3, Y_3, Z_3, aZ_3^4)$ is:

$$X_3 = T \qquad Y_3 = M(S - T) - U$$
$$Z_3 = 2Y_1Z_1 \qquad aZ_3^4 = 2U(aZ_1^4)$$

met

$$S = 4X_1Y_1^2 \qquad U = 8Y_1^4$$
$$M = 3x_1^2 + aZ_1^4 \qquad T = -2S + M^2$$

Voor de omzetting van de projectieve coördinaten in $\mathcal{J}^m$ naar affine coördinaten worden x en y berekend volgens vergelijking 2.6. Hiervoor moet dan eenmalig het inverse van Z berekend worden.

De 2 bovenstaande verbeteringen worden samengebracht in de EC processor. Om de puntbewerkingen uit te voeren wordt het primitief punt eerst omgezet naar het projectieve coördinaatstelsel $\mathcal{J}^m$. Vervolgens worden de 4 coördinaten elk omgezet in hun Montgomery voorstelling. Met deze waarden wordt dan de gewenste puntbewerking uitgevoerd. Hierna wordt het bekomen punt omgezet van $\mathcal{J}^m$ naar affine coördinaten. Deze omzetting gebeurt eerst omdat tijdens deze stap nog gebruik gemaakt wordt van de Montgomery bewerking. Uiteindelijk worden deze coördinaten omgezet van Montgomery voorstelling naar hun normale waarden.

2.3 Implementatie

De implementatie van ECDH en ECDSA gebeurt door middel van een elliptische kromme processor aangestuurd door een FSM. De implementatie van de EC processor is gebaseerd op [7]. Deze processor verwerkt de puntvermenigvuldiging en de puntoptelling en kan opgedeeld worden in 5 lagen:

- laag 1: de controle-unit (MC);
- laag 2:
  - normaal naar Montgomery voorstelling omzetter (NtoM),
  - EC puntbewerking (EPOp),

  - projectieve naar affine coördinaat omzetter (PtoA),
  - Montgomery naar normaal voorstelling omzetter (MtoN);
- laag 3:
  - EC puntverdubbeling en -optelling circuit (PAD),
  - modulaire multiplicatief invertor (MMI);
- laag 4:
  - Montgomery modulaire vermenigvuldiger (MMM),
  - modulaire opteller/aftrekker (MAS);
- laag 5: opteller/aftrekker (AS).

De interactie tussen deze modules is te zien in figuur 2.2. Deze figuur geeft enkel de controlesignalen weer. Merk op dat de affine naar projectieve omzetting (AtoP) hier weggelaten is. Deze stap is volledig combinatorisch en wordt verwerkt aan de ingangen van de x en y coördinaat in de controle-unit.

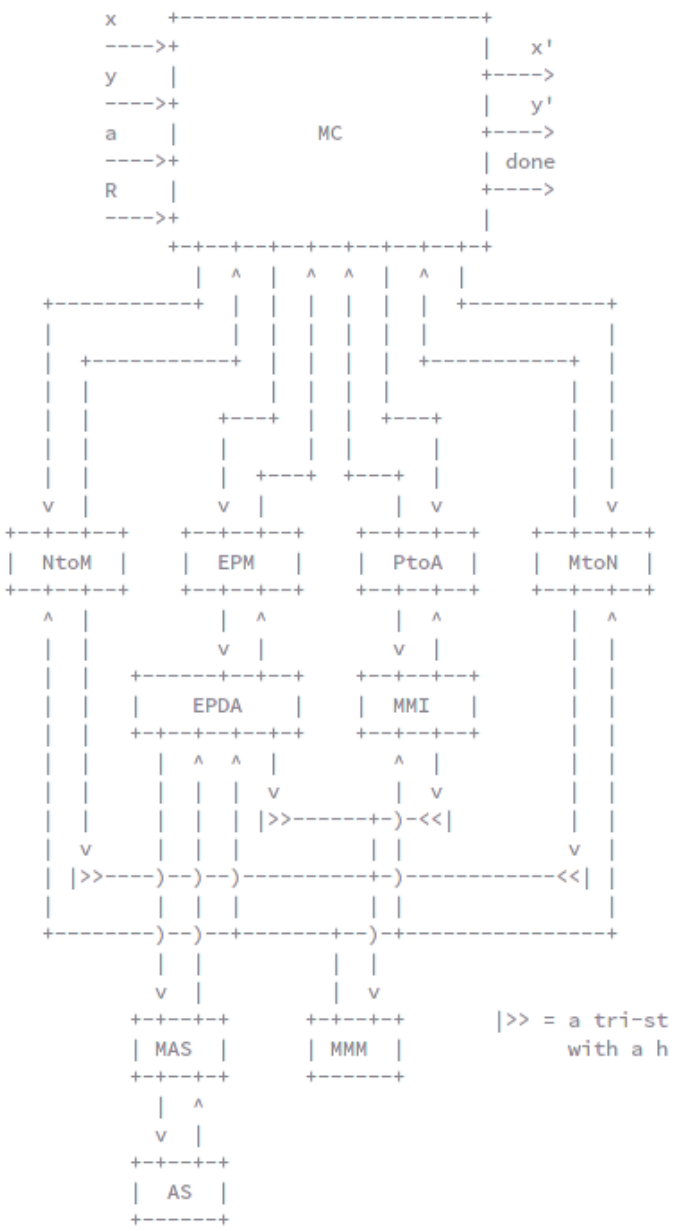
Tijdens dit project werd er gewerkt aan het EC puntverdubbeling en -optelling circuit, de EC puntbewerking en de EC processor controller.

2.3.1 EC puntverdubbeling en -optelling

In [3] werd er reeds gewerkt aan de puntverdubbeling en de puntoptelling. Bij het testen van deze modules bleek echter dat de puntoptelling niet werkte. Hierdoor moest deze module herschreven worden. Deze module is samen met de puntverdubbeling herschreven tot één gecombineerd geheel. Hierdoor kunnen de registers in deze module voor beide stappen gebruikt worden. De FSM die de module aanstuurt heeft 30 states. 14 states voor de puntoptelling, 14 voor de puntverdubbeling, een Idle state en een Done state. De gevraagde bewerking wordt uitgevoerd volgens de algoritmes in tabel 2.2. Voor het geheel zijn 7 registers nodig, waarvan er 3 tijdelijk zijn. Voor de puntoptelling zijn 14 Montgomery bewerkingen en 7 modulaire optellingen nodig. Voor de puntverdubbeling zijn 8 Montgomery bewerkingen en 14 modulaire optellingen nodig.

De state waarin de FSM zich bevindt wordt aangegeven met een 5-bit getal. Met behulp van een statedecoder wordt dit getal omgezet in 7 enable signalen voor de verschillende registers, 7 registerselectors, 4 uitgangsselectors, een enable voor de Montgomery bewerking, een enable voor de modulaire optelling en een add/subtract signaal. De enables zorgen ervoor dat op het juiste moment, de juiste registers overschreven worden. De registerselectors selecteren of de registers de ingang van de Montgomery bewerking of van de modulaire optelling inlezen.

```
reg enable sel = delayEn init
    (doneCalc <&> enable) (muxl sel returnMMM returnMAS)
t1  = reg  enableT1  selT1
t2  = reg  enableT2  selT2
t3  = reg  enableT3  selT3
```



Figuur 2.2: EC processor

```
x3  = reg  enableX3  selX3
y3  = reg  enableY3  selY3
z3  = reg  enableZ3  selZ3
aZ3 = reg enableAZ3 selAZ3
```

Deze code toont meteen een voordeel van Lava. Het is zeer eenvoudig om een functie te definiëren voor zaken die meermaals gebruikt worden.

De uitgangsselectors zijn *one-hot* signalen waarmee bepaald wordt welke registers aan de uitgangen toegankelijk zijn. Twee van deze uitgangen gaan naar de Montgomery bewerking, de andere twee naar de modulaire optelling. Bij de uitwerking werd een poging gedaan om de multiplexers zo klein mogelijk te houden terwijl toch alle nodige bewerkingen mogelijk blijven. Hiervoor wordt een van de multiplexeruitgangen aan een ingang van de andere verbonden. Dit maakt het mogelijk om een signaal aan beide uitgangen toegankelijk te stellen zonder dat het aan de ingang van beide multiplexers aanwezig is. Dit is nodig voor signalen die gekwadrateerd of verdubbeld moeten worden. Sommige registers moeten wel aan beide multiplexers toegankelijk zijn. Dit was onvermijdelijk door de manier waarop de registers in het algoritme verwerkt zijn. Mogelijk is hiervoor een betere manier maar hiernaar werd niet gezocht tijdens dit project. Hieronder wordt de Lava code getoond waarmee de Montgomery bewerking wordt aangestuurd. De uitgangen naar de modulaire optelleraftrekker worden op een gelijkaardige manier aangestuurd. Het organiseren van de multiplexers bij de optelling is echter wat complexer omdat, door de mogelijkheid om af te trekken, ook rekening gehouden moet worden welk register op welke uitgang komt te staan.

```
mmmInput1 = selectG mmmSel1 [x1,y1,z1,t1,t2,t3,y3,z3,aZ3]
mmmInput2 = selectG mmmSel2 [z1,aZ1,x2,y2,a,y3,aZ3,mmmInput1]
mmmSel1 =
    [ s18
    , s15 <|> s19
    , s1
    , orG [s2, s3,  s4,  s10, s16, s17, s27]
    , orG [s5, s7,  s8,  s20]
    , s9  <|> s12
    , s23
    , s11
    , orG [s6, s13, s14]
    ]
mmmSel2 =
    [ orG [s3, s8,  s19]
    , s20
    , s2
    , s4
    , s14
    , orG [s6, s12, s17, s27]
    , s7  <|> s10
    , orG [s1, s5,  s9,  s11, s13, s15, s16, s18, s23]
    ]
```

Tabel 2.2: EC puntoptelling en -verdubbelings algoritmes

EC puntoptelling							
uitgaande van: $P_1 = (X_1, Y_1, Z_1, aZ_1^4); P_2 = (X_2, Y_2, 1, a)$							
bekomen we: $P_1 + P_2 = P_3 = (X_3, Y_3, Z_3, aZ_3^4)$							
volgens:							
1.	T_1	$\leftarrow$	Z_1^2				
2.	Y_3	$\leftarrow$	$X_2 \cdot T_1$				
3.	T_1	$\leftarrow$	$Z_1 \cdot T_1$	T_2	$\leftarrow$	$X_1 - Y_3$	
4.	T_1	$\leftarrow$	$Y_2 \cdot T_1$				
5.	aZ_3^4	$\leftarrow$	T_2^2	T_3	$\leftarrow$	$Y_1 - T_1$	
6.	Y_3	$\leftarrow$	$Y_3 \cdot aZ_3^4$				
7.	aZ_3^4	$\leftarrow$	$T_2 \cdot aZ_3^4$	X_3	$\leftarrow$	$2 \cdot Y_3$	
8.	Z_3	$\leftarrow$	$Z_1 \cdot T_2$	X_3	$\leftarrow$	$X_3 + aZ_3^4$	
9.	T_2	$\leftarrow$	T_3^2				
10.	T_1	$\leftarrow$	$T_1 \cdot aZ_3^4$	X_3	$\leftarrow$	$T_2 - X_3$	
11.	aZ_3^4	$\leftarrow$	Z_3^2	Y_3	$\leftarrow$	$Y_3 - X_3$	
12.	T_2	$\leftarrow$	$Y_3 \cdot T_3$				
13.	aZ_3^4	$\leftarrow$	$(aZ_3^4)^2$	Y_3	$\leftarrow$	$T_2 - T_1$	
14.	aZ_3^4	$\leftarrow$	$a \cdot aZ_3^4$				

EC puntverdubbeling							
uitgaande van: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$							
bekomen we: $2P_1 = P_3 = (X_3, Y_3, Z_3, aZ_3^4)$							
volgens:							
1.	T_1	$\leftarrow$	Y_1^2	Y_3	$\leftarrow$	$2 \cdot X_1$	
2.	T_2	$\leftarrow$	T_1^2	Y_3	$\leftarrow$	$2 \cdot Y_3$	
3.	T_1	$\leftarrow$	$Y_3 \cdot T_1$	T_2	$\leftarrow$	$2 \cdot T_2$	
4.	Y_3	$\leftarrow$	X_1^2	T_2	$\leftarrow$	$2 \cdot T_2$	
5.	aZ_3^4	$\leftarrow$	$Y_1 \cdot Z_1$	T_2	$\leftarrow$	$2 \cdot T_2$	
6.	T_3	$\leftarrow$	$T_2 \cdot aZ_1^4$	X_3	$\leftarrow$	$2 \cdot Y_3$	
7.				Y_3	$\leftarrow$	$Y_3 + X_3$	
8.				Y_3	$\leftarrow$	$Y_3 + aZ_1^4$	
9.	X_3	$\leftarrow$	Y_3^2	Z_3	$\leftarrow$	$2 \cdot aZ_3^4$	
10.				aZ_3^4	$\leftarrow$	$2 \cdot T_1$	
11.				X_3	$\leftarrow$	$X_3 - aZ_3^4$	
12.				T_1	$\leftarrow$	$T_1 - X_3$	
13.	Y_3	$\leftarrow$	$Y_3 \cdot T_1$	aZ_3^4	$\leftarrow$	$2 \cdot T_3$	
14.				Y_3	$\leftarrow$	$Y_3 - T_2$	

2.3.2 EC puntbewerking

Aangezien de EC puntverdubbeling en -optelling zeer grote veranderingen hebben ondergaan moest ook de EC puntbewerking aangepast worden. Hierbij moest nog een kleine fout gecorrigeerd worden die ervoor zorgde dat de vermenigvuldigingsfactor geen *leading zeroes* mocht hebben. Hiernaast moest er ook een uitbreiding voorzien worden waardoor het mogelijk is een optelling te doen van 2 punten. Deze optelling is nodig voor de handtekeningcontrole bij ECDSA.

De module voor de puntbewerking bestaat uit de volgende elementen:

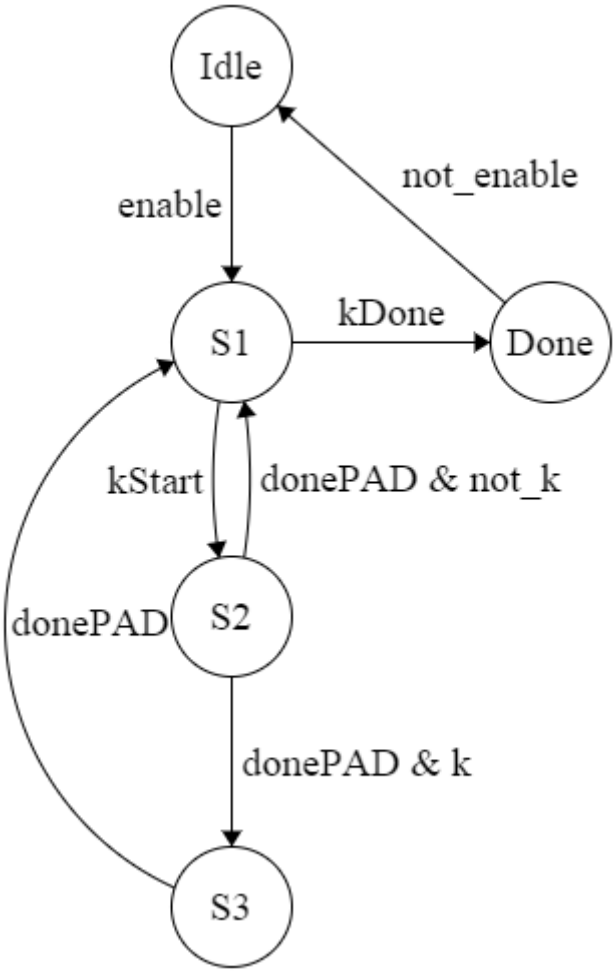
- een puntregister voor de opslag van de tussentijdse resultaten,
- een schuifregister waarin de vermenigvuldigingsfactor k wordt opgeslagen,
- een FSM die puntvermenigvuldiging regelt

De FSM heeft 5 states: Idle, S1, S2, S3, Done. Deze worden voorgesteld in figuur 2.3. Bij de start van de puntvermenigvuldiging bevindt de FSM zich in de Idle state. Als het enableMult signaal hoog gemaakt wordt gaat hij over naar state S1. Deze state is verantwoordelijk voor het doorschuiven van schuifregister en controleert of alle bits van k overlopen zijn. Elke klokpuls, dat de FSM zich in deze state bevindt, wordt één bit uit het schuifregister geschoven. Bij de start wordt een bit kStart op 0 gezet. Deze wordt gebruikt om aan te duiden dat er nog *leading zeroes* uit het register geschoven moeten worden. Als de bit die uit het schuifregister geschoven wordt een 1 is wordt kStart op 1 gezet en vervolgens wordt er overgegaan naar state S2.

In S2 wordt de EC puntverdubbeling gestart. Als deze voltooid is kijkt de FSM naar de eerste bit van het schuifregister. Is dit een 0, dan gaat de FSM terug naar state S1. S1 schuift het register dan 1 plaats door en controleert opnieuw of de vermenigvuldiging klaar is. Als dit niet het geval is gaat de FSM opnieuw naar state S2 voor de volgende verdubbeling. Indien, bij het beëindigen van de verdubbeling, de eerste bit van het schuifregister een 1 is wordt er verder gegaan met de puntoptelling in state S3.

Na deze optelling gaat de FSM terug naar state S1. Wanneer de vermenigvuldiging voltooid is gaat de FSM van S1 naar de Done state. Deze zet het done signaal hoog. De FSM blijft in deze state tot het enable signaal laag wordt gemaakt, waarna overgegaan wordt naar de Idle state.

De FSM kan omzeild worden als er enkel een optelling moet gebeuren. Dit gebeurt met behulp van de store en enableAdd ingangen. De store ingang wordt hoog gemaakt als het eerste punt ingelezen moet worden. Hierdoor wordt dit punt opgeslagen in het puntregister. Vervolgens wordt de enableAdd ingang hoog gemaakt. Dit signaal wordt doorgegeven naar de startPAD uitgang. Als de optelling voltooid is wordt het donePAD signaal doorgegeven aan de donePtOp uitgang. Dit geeft aan dat de gewenste puntbewerking voltooid is.



Figuur 2.3: De FSM van de puntbewerking

2.3.3 EC processor controller

De controller bestaat uit een FSM en een puntregister. Dit puntregister bestaat uit 4 n-bit registers waarin de X, Y, Z en aZ^4 coördinaten worden opgeslagen. De inhoud van het register wordt bij het voltooien van elke tussenstap overschreven met de nieuwe waarde.

De controller heeft 2 functies: het verwerken van een puntvermenigvuldiging en het verwerken van een puntoptelling. In figuur 2.4 is een schematische voorstelling te zien van de state overgangen zoals hieronder beschreven.

Puntvermenigvuldiging Bij de start van een bewerking is deze in de Idle state. Als het enable signaal hoog wordt, gaat de FSM over naar de state NtoM1. Hier wordt het startNtoM signaal hoog gemaakt waardoor de NtoM module het ingegeven punt omzet naar Montgomery voorstelling. Als deze stap voltooid is zet de NtoM module het doneNtoM signaal hoog. De controller zal nu over gaan naar de PM state als het add signaal aan de ingang laag is.

De PM state stuurt de EPOp module aan om de puntvermenigvuldiging te starten. Als deze voltooid is maakt de EPOp module de donePtOp ingang hoog.

Hierna gaat de FSM over naar de PtoA state. Deze state is verantwoordelijk voor de omzetting van het opgeslagen punt van projectieve naar affine coördinaten. Bij het voltooien wordt de donePtoA ingang hoog gezet door de PtoA module.

Hierna gaat de controller over naar de NtoM state waarin de coördinaten omgezet worden van de Montgomery voorstelling naar de normale voorstelling.

Uiteindelijk komt de controller in de Done state waarin het done uitgangssignaal hoog gemaakt wordt. De FSM blijft in deze state tot de enable input laag gemaakt wordt waardoor de FSM weet dat de uitgangen uitgelezen zijn.

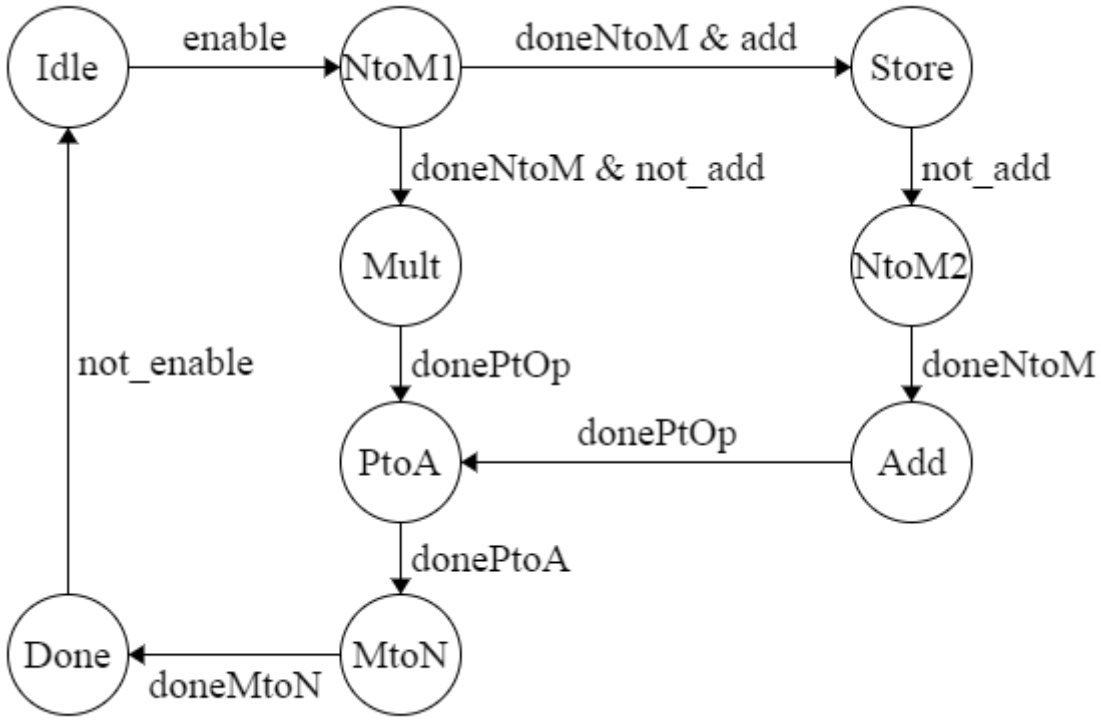
Vervolgens gaat de FSM over naar de Idle state. Van hieruit kan de volgende puntbewerking gestart worden.

Puntoptelling De Idle state en de NtoM1 state zijn bij de puntoptelling hetzelfde als bij de puntvermenigvuldiging. Bij de overgang van NtoM1 naar de volgende state wordt gekeken naar de add input. Als het add signaal aan de ingang hoog is, gaat de FSM over naar de Store state.

De Store state zorgt ervoor dat de inhoud puntregister gekopieerd wordt naar het puntregister in de EPOp module. Dan zet de controller de store output hoog om aan te geven dat het eerste punt opgeslagen is. Door de add input laag te maken geeft weet de controller dat het volgende punt klaar staat om ingelezen te worden aan de ingang.

Na de Store state gaat de FSM over naar state NtoM2 waarin het 2de punt ingelezen en omgezet wordt. De NtoM module zet, als hij klaar is, opnieuw de doneNtoM ingang hoog.

De volgende state is de Add state. Hier stuurt de FSM de puntbewerking module aan. Deze bewerking telt de punten in het puntregister van de controller en in het puntregister van de EPOp module bij elkaar op. Als de optelling voltooid is zet de EPOp module de donePtOp hoog en vervolgens gaat de controller over naar de PtoA state.



Figuur 2.4: De FSM van de EC processor

Dit is dezelfde PtoA state als beschreven bij de puntvermenigvuldiging. Van hieruit gebeurt de overgang naar de Idle state op een gelijkaardige manier.

2.3.4 ECDH/ECDSA module

In de ECDH/ECDSA module worden de cryptografische algoritmes uitgerekend zoals overlopen in hoofdstuk 2.1.2. Hiervoor is een FSM gemaakt met 3 verschillende takken die de Idle en Done states gemeenschappelijk hebben.

De eerste tak genereert de publieke sleutel voor gebruik bij ECDSA en voert de ECDH bewerkingen uit. Bij het genereren van de publieke sleutel wordt meteen de geheime sleutel opgeslagen in een register. Deze tak bestaat uit 6 states. In de eerste state wordt gewacht tot de geheime sleutel ingevoerd wordt. De tweede state stuurt de EC processor aan om een puntvermenigvuldiging uit te voeren. Vervolgens gaat de FSM over naar de 3de state. Vanuit deze state wordt, bij de sleutel generatie, overgegaan naar state 4 waarin de geheime sleutel omgezet wordt naar zijn Montgomery voorstelling. Deze waarde wordt dan in het register opgeslagen, waarna overgegaan wordt naar de Done state. Indien de FSM aangestuurd werd om ECDH uit te voeren gaat deze over naar state 5. Hier wordt gewacht tot het punt, berekend in state 2, uitgelezen is en het ontvangen punt aan de ingang toegankelijk is. Met dit nieuwe punt wordt in state 6 de ECDH sleutel berekend, waarmee de ECDH bewerking voltooid is en de FSM overgaat naar de Done state.

De tweede tak van de FSM berekend de handtekening van een gevraagde boodschap. Hiervoor wordt in state 7 gewacht tot een tijdelijke sleutel gegenereerd is. Deze wordt dan gebruikt in state 8 bij de puntvermenigvuldiging. Vervolgens worden in states 9 en 10 de x-coördinaat van dit punt en de tijdelijke sleutel omgezet naar hun Montgomery voorstelling. De Montgomery waarde van tijdelijke sleutel wordt in state 11 geïnverteerd. In state 12 wordt het Montgomery product van x en de geheime sleutel berekend. Hierna moet er gewacht worden tot de hash waarde van de boodschap berekend is. Deze wordt dan in state 14 omgezet naar Montgomery voorstelling. Vervolgens wordt, in state 15, de som van de hash en de waarde berekend in state 12 berekend. Van deze som en van het inverse van de tijdelijke sleutel wordt het Montgomery product berekend. Uiteindelijk moet deze waarde in de laatste state (state 17) omgezet worden naar normale voorstelling, waarna overgegaan wordt naar de Done state.

In de derde tak wordt gecontroleerd of de ingegeven handtekening geldig is. Hiervoor wordt eerst de ingegeven handtekening omgezet naar zijn Montgomery voorstelling. Deze waarde wordt vervolgens geïnverteerd. Hierna wordt gewacht tot de r waarde van de handtekening ingegeven is. Ook deze wordt omgezet naar Montgomery voorstelling. Vervolgens wordt het Montgomery product van deze 2 waarden berekend. Dit product wordt dan omgezet naar normale voorstelling, waarna het gebruikt wordt in de eerste puntvermenigvuldiging[2]. Vervolgens moet er gewacht worden tot de hashwaarde van de boodschap berekend is. Deze wordt dan omgezet naar Montgomery voorstelling, waarna het product van de hash en het inverse van de handtekening berekend wordt. Dit product wordt omgezet naar normale voorstelling. Als dit gebeurd is, wacht de FSM tot het tweede punt klaar staat aan de ingangen. Dit tweede punt wordt dan vermenigvuldigd met de berekende waarde. De twee bekomen punten worden vervolgens bij elkaar opgeteld en de x-coördinaat van het bekomen punt wordt in het uitgangsregister opgeslagen. De handtekening is dan geldig als deze waarde hetzelfde is als r uit de handtekening.

2.4 Testresultaten

Om de ECC modules te testen wordt eerst een Haskell implementatie aangemaakt. Deze functies zullen de verschillende stappen van de EC processor en de ECDH/ECDSA module nabootsen. Beide resultaten kunnen dan met elkaar vergeleken worden. Het testen gebeurt aan de hand van een eenvoudige elliptische kromme, namelijk:

$$y^2 = x^3 + 2x + 2 \mod 17 \tag{2.7}$$

met het punt $P = (5, 1)$ als primitief element. Van deze kromme kunnen makkelijk alle punten berekend worden. Deze zijn de volgende:

[2] De eerste puntvermenigvuldiging die uitgevoerd wordt in deze tak is u_2B, hierdoor heeft de hashmodule meer tijd om de boodschap te verwerken.

$2P = (6, 3)$ $8P = (13, 7)$ $14P = (9, 1)$
$3P = (10, 6)$ $9P = (7, 6)$ $15P = (3, 16)$
$4P = (3, 1)$ $10P = (7, 11)$ $16P = (10, 11)$
$5P = (9, 16)$ $11P = (13, 10)$ $17P = (6, 14)$
$6P = (16, 13)$ $12P = (0, 11)$ $18P = (5, 16)$
$7P = (0, 6)$ $13P = (16, 4)$ $19P = \mathcal{O}$

2.4.1 EC processor

Vermenigvuldiging In figuur 2.5 worden de testresultaten getoond van een functie die alle mogelijke punten op de gegeven kromme berekend[3]. De parameters van de functie zijn de veldbreedte, de grote van het datapad en de EC parameters: x_P, y_P, a, p, p', R^2 en k. k wordt hier overlopen van 1 tot 18. De testresultaten zijn als volgt geformatteerd: $(c, [x_R, y_R])$ met c het aantal klokpulsen nodig om de bewerking uit te voeren. Alle punten worden juist berekend. De variatie van de nodige klokpulsen toont dat de EC processor nog niet beveiligd is tegen *side channel attacks*. Zo zou het mogelijk zijn om uit de waarde van c de gebruikt sleutel te ontcijferen.

```
*Main> listOutput $ map (\x ->
  ecMult 5 1 [5,1,2,17,15,4,x]) [1..18]
(1370,[5,1])    (4619,[7,11])
(2065,[6,3])    (5783,[13,10])
(3229,[10,6])   (4619,[0,11])
(2760,[3,1])    (5783,[16,4])
(3924,[9,16])   (5783,[9,1])
(3924,[16,13])  (6947,[3,16])
(5088,[0,6])    (4150,[10,11])
(3455,[13,7])   (5314,[6,14])
(4619,[7,6])    (5314,[5,16])
```

Figuur 2.5: EC processor testresultaten vermenigvuldiging

De functie ecMult die bij deze test gebruikt wordt is een wrapperfunctie die een aantal parameters voor de EC processor invult en alle resultaten uitfiltert waarbij de done uitgang laag is. Hierdoor wordt enkel het eindresultaat getoond.

***Timing attack* op de EC processor** Om een *timing attack* uit te voeren moeten een aantal zaken over de EC processor submodules gekend zijn, namelijk:

- Het aantal klokpulsen nodig voor de Montgomery bewerking $M = 2l^2 + 5l + 7$ met l de bit lengte van de EC processor.

[3] Deze test werd uitgevoerd met GHC v. 7.8.4. Nieuwere versies kunnen mogelijke geen resultaten geven door compatibiliteitsproblemen met de Lava bibliotheek.

- Het aantal klokpulsen nodig voor de modulaire optelling $S = 4$
- Het aantal klokpulsen nodig voor de puntoptelling $P_{opt} = 14M + 15$
- Het aantal klokpulsen nodig voor de puntverdubbeling $P_{vd} = 8M + 6S + 15$.
- Er zijn 10 Montgomery bewerkingen nodig bij de voor- en nabewerkingen (zonder modulair inverse).
- Het aantal klokpulsen nodig voor de modulaire inverse is $I = (k-1)M + (n-1)M + l + 3$ met k het aantal significante bits in de exponent en n het aantal 1'en.

Voor de gebruikte kromme is de exponent $p - 2 = 15$ hieruit bekomen we dan:

$$\begin{aligned} M &= 82 \\ S &= 4 \\ P_{opt} &= 695 \\ P_{vd} &= 1163 \\ I &= 500 \end{aligned}$$

Als we bijvoorbeeld kijken naar de uitkomst $(4619, [0, 11])$, weten we dat er ongeveer 1320 klokpulsen nodig zijn voor de voor- en nabewerkingen. De puntvermenigvuldiging heeft dus ongeveer 3300 klokpulsen nodig. Hieruit kunnen we afleiden dat er waarschijnlijk 3 verdubbelingen nodig waren en 1 optelling, want $3 \cdot 695 + 1 \cdot 1163 = 3248$. Hieruit weten we dat het punt $(0, 11)$ gelijk is aan 9, 10 of 12 maal P. Uit de lijst van punten is te zien dat het inderdaad $12P$ is. Deze bewerking verminderde de waardes die gecontroleerd moeten worden van 19 naar 3. Dit is een zeer eenvoudig voorbeeld maar hieruit is al duidelijk te zien dat de huidige EC processor niet veilig is voor cryptografisch gebruik.

Optelling De EC processor kan ook 2 punten bij elkaar optellen. Resultaten van de optellingstest zijn te zien in figuur 2.6. Hierbij werden een aantal willekeurige punten bij elkaar opgeteld. Deze bewerking geeft het juiste resultaat als de som niet gelijk is aan $19P$ zoals te zien is in de afbeelding. Dit komt omdat de EC processor niet overweg kan met het punt op oneindig. Deze beperking is echter geen probleem in een praktische implementatie omdat de random bit generator gelimiteerd is tot $p - 1$.

2.4.2 ECDH/ECDSA module

Om de ECDH/ECDSA module te testen werd het voorbeeld uit [8] p. 285 nagebootst. Hier stuurt een partij B een boodschap naar A. Alvorens dit kan gebeuren moet B eerst een asymmetrisch sleutelpaar genereren. Hiervoor werd een geheime sleutel gelijk aan 7 gekozen. De publieke sleutel is dan $(p, a, b, q, P, Q) = (17, 2, 2, 19, (5, 1), (0, 6))$. Het resultaat van de ecdsaKeyGen wrapper functie geeft het punt Q correct (zie figuur 2.7). De boodschap die B wil versturen heeft een hashwaarde van 26 en de tijdelijke sleutel is 10.

```
*Main> listOutput $ map (\(x1,y1,x2,y2) -> ecAdd 5 1 [x1,y1,x2,y2,2,17,15,4,1])
[(5,1,6,3),(0,6,13,10),(16,4,5,1),(6,14,16,4),(9,16,9,1)]
(2867,[10,6])
(2867,[5,16])
(2867,[9,1])
(2867,[13,10])
(2867,[0,0])
```

Figuur 2.6: EC processor testresultaten optelling

De ecdsaSigGen wrapper berekent hiervoor de juiste handtekening[4].

Voor de test van de handtekening controle werd geen wrapper functie geschreven omdat, door een gebrek aan RAM geheugen op de gebruikte computer, geen volledige test uitgevoerd kon worden. Om deze functie te testen werden de interne registers handmatig uitgelezen wanneer de FSM in state 30 is. De registers kunnen dan geïnitialiseerd worden op deze waarden en vervolgens kan de FSM geherprogrammeerd worden om de overige states over te slaan. Hierdoor kan gecontroleerd worden dat de handtekeningcontrole het juiste punt bekomt.

```
*Main> ecdsaKeyGen 5 1 [5,1,2,17,15,4,19,5,17,7]
(5174,[0,6])
*Main> ecdsaSigGen 5 1 [5,1,10,26]
(5547,[7,17])
```

Figuur 2.7: ECDH/ECDSA module testresultaten

2.5 Beperkingen en mogelijke verbeteringen

Er zijn nog een aantal beperkingen aan de huidige ECDH/ECDSA implementatie. Allereerst kan er op dit moment geen gebruik gemaakt worden van 2 verschillende elliptische krommen met dezelfde hardware. Dit is te wijten aan een fout in de Montgomery bewerking waardoor deze niet overweg kan met *leading zeroes* in de modulus. Dit moet verbeterd worden waarna er gecontroleerd moet worden of de andere modules geen gelijkaardige fouten in het systeem introduceren.

Een tweede verbetering die nog kan gebeuren is het opsplitsen van de modules en hun submodules. In figuur 2.2 wordt het schema weergegeven van hoe de modules verbonden moeten zijn. In realiteit zijn de NtoM, de MMI en de MtoN elk met een eigen Montgomery bewerking verbonden. Ook heeft de ECDH/ECDSA een eigen Montgomery bewerking, modulaire optelling en modulair inverse. Dit werd zo gedaan bij voorgaande projecten om het testen van de modules te vereenvoudigen. Dit kan ertoe bijdragen dat de simulaties veel trager verlopen omdat deze modules elk apart gesimuleerd worden. De opsplitsing van de modules is reeds gebeurd bij de EPDA en EPOp modules.

[4] Deze wrapper heeft minder inputs omdat de simulateSeq functie die Lava gebruikt niet werkt met een te groot aantal inputs

Een derde probleem is de opbouwende complexiteit van de schakeling. Dit zorgt ervoor dat het simuleren van 5-bit elliptische kromme al zeer traag verloopt en soms zelfs een crash van GHC veroorzaakt. Dit komt omdat York Lava elke schakeling volledig als hardware simuleert. Zo zal bijvoorbeeld een optelling die gebruik maakt van een *ripple carry adder* elke XOR en elke AND poort simuleren. Voor grote schakelingen zoals deze ECDH/ECDSA module die, zelf voor een 5-bit elliptische kromme, al enkele duizenden klokcycli nodig heeft, loopt dit al snel op tot een zeer groot geheugengebruik.

Op dit moment zijn de EC processor en de ECDH/ECSDA module nog vatbaar voor *timing attacks*. Dit komt omdat de puntoptelling steeds overgeslagen wordt als deze niet nodig is. In de module die het modulair inverse berekent is ook een soortgelijke kwetsbaarheid. Hierdoor is het mogelijk om vanuit de handtekeninggeneratie, de tijdelijke sleutel te bekomen en hiermee kan dan de geheime sleutel berekend worden.

Hoofdstuk 3

Hash functie

Hash functies worden gebruikt om ingevoerde data, die een onbepaalde lengte heeft, om te zetten in data met een vastgelegde lengte. Hierbij worden een aantal eisen opgelegd voor de hash functie, namelijk:

- Het moet makkelijk zijn om de hashwaarde van een bepaalde input te berekenen;
- Het moet praktisch onmogelijk zijn om van een hashwaarde de originele boodschap te genereren;
- Het moet praktisch onmogelijk zijn om een boodschap aan te passen zonder dat de hashwaarde verandert;
- Het moet praktisch onmogelijk zijn om twee verschillende boodschappen te vinden met dezelfde hashwaarde.

Hashfuncties worden gebruikt bij het zetten van digitale handtekeningen omdat ze het nodige rekenwerk bij grote boodschappen sterk verminderen. Het ondertekenen van de hash van een boodschap levert namelijk dezelfde integriteitsgarantie als het ondertekenen van de boodschap zelf.

3.1 SHA-2

SHA, kort voor *Secure Hash Algorithm*, is een groep van cryptografische hashfuncties ontwikkeld door de National Security Agency van de Verenigde Staten. SHA-2 bestaat uit zes hashfuncties: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. Tijdens het project werd een implementatie gemaakt voor SHA-224 en SHA-256.

Het SHA-256 algoritme bestaat uit 3 stappen:

- de voorbewerking,
- de boodschapverwerking,
- de compressie.

Tijdens de voorbewerking wordt de boodschap aangevuld met padding en de lengte van de boodschap. Hierdoor wordt een geheel bekomen met een lengte dat een veelvoud is van 512. Dit geheel wordt in blokken van 512 bits opgesplitst voor de boodschapverwerking. Hier wordt elke blok opgesplitst in 16 32-bit woorden. Uit deze 16 woorden worden nog 48 andere woorden gegenereerd. Deze extra woorden zorgen ervoor dat door een kleine verandering in de input de uiteindelijke hashwaarde compleet verandert. Dit wordt het lawine-effect genoemd. Tijdens de compressiestap worden de 64 woorden verwerkt tot een tussentijdse hashwaarde. De boodschapverwerking en de compressiestap worden herhaald totdat alle 512-bit blokken verwerkt zijn.

Het SHA-224 algoritme werkt op dezelfde manier. Hierbij wordt echter gebruik gemaakt van een andere initialisatiewaarde in de compressiestap en de laatste 32 bits van de output worden weggelaten.

3.2 Implementatie

De SHA-256 module heeft 5 ingangen: input, shaVersion, last, bytes en reset. De input is een 32-bit kanaal waar de te verwerken boodschap stap voor stap wordt ingevoerd. shaVersion is een selectiebit voor SHA-224 of SHA-256 werking. De last ingang wordt gebruikt om aan de duiden dat de huidige input de laatste geldige input is. De bytes input wordt gebruikt samen met de last bit. Als bytes gelijk is aan '00' betekend dit dat enkel de eerste byte van de huidige input geldig is, '01' betekend de eerste 2 bytes, enzoverder. Reset wordt gebruikt om de interne registers leeg te maken.

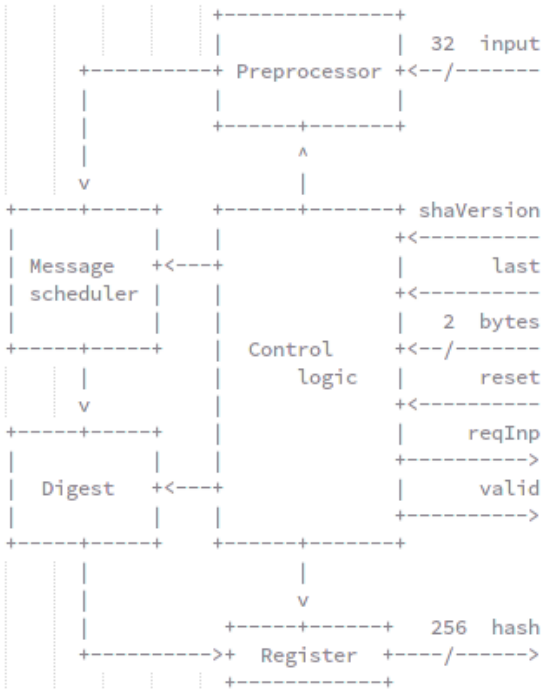
De module heeft 3 uitgangen: hash, valid en reqInp. Hash is de 256-bit hash output. Valid duidt erop dat de hash uitgang de correcte hashwaarde toont. De valid uitgang blijft dan hoog totdat de module gereset wordt. ReqInp is hoog als op de volgende klokcyclus de volgende 32-bit input verwacht wordt. In figuur 3.1 zijn de in en uitgangen van de SHA-256 module te zien.

De interne werking van de hashfunctie is opgedeeld in 4 delen:

- de controle logica,
- de voorbewerking,
- de boodschapverwerking,
- de compressie.

3.2.1 De controle logica

Vanuit de controle logica worden de voorbewerking, de boodschapverwerking en de compressie van de hashfunctie aangestuurd. De controle logica bestaat uit een FSM, een 6-bit counter en het uitgangsregister. De waarde van de counter telt op bij elke klokcyclus. Deze waarde bepaalt de toestandsovergangen van de FSM en de werking van de boodschapverwerking en compressiestap. De FSM regelt de werking van de voorbewerking.

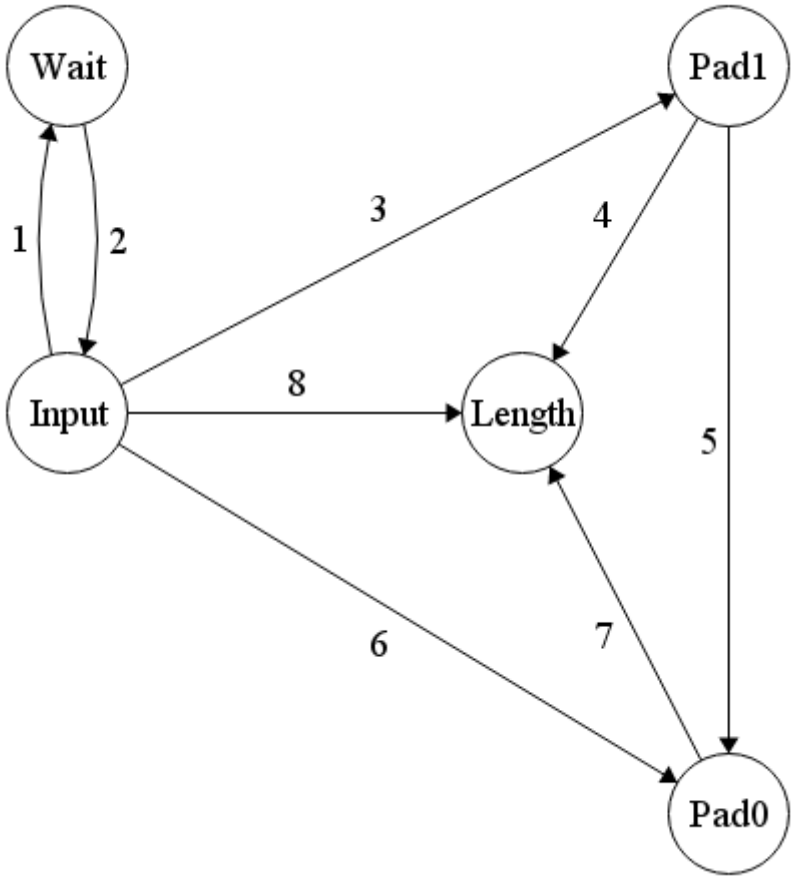


Figuur 3.1: SHA-256 module

3.2.2 De voorbewerking

De voorbewerking is een circuit dat de ingang omvormt tot de aangevulde input. Dit gebeurt door middel van een combinatorisch circuit en een blokcounter. De last en bytes ingangen bepalen welke bytes van de input vervangen moeten worden door padding bits. De toestand van de FSM (te zien in figuur 3.2) bepaalt dan of de aangepaste input, de lengte of padding wordt doorgegeven naar de boodschapverwerking. De overgangssignalen uit deze figuur zijn afhankelijk van het reset, de last, de bytes en de counter signalen. Als het reset signaal hoog is, gaat de FSM altijd over naar de Input state. Anders zijn de voorwaarden voor een overgang de volgende:

- Overgang 1: de interne counter is gelijk aan 15.
- Overgang 2: de counter is gelijk aan 63.
- Overgang 3: het last input signaal en beide byte bits zijn hoog.
- Overgang 4: de counter is gelijk aan 13.
- Overgang 5: de counter is kleiner dan 16 maar niet gelijk aan 13.
- Overgang 6: de last bit is hoog, de counter is niet gelijk aan 13 en er wordt niet voldaan aan Overgang 3.
- Overgang 7: de counter is gelijk aan 13.
- Overgang 8: de last bit is hoog, de counter is gelijk aan 13 en er wordt niet voldaan aan Overgang 3.



Figuur 3.2: SHA Voorbewerking FSM

Tijdens de toestand Input wordt input doorgegeven naar de boodschapverwerking. De reqInp uitgang wordt in deze toestand hoog gehouden. De toestanden Pad0 en Pad1 voegen de padding toe aan de ingevoerde boodschap en de Length toestand voegt de lengte toe.

De lengte is opgebouwd uit de inhoud van een blokcounter, die elke 512-bit blok telt, en de inhoud van de 6-bit counter uit de controle logica. Deze wordt vastgezet in een register tijdens het inlezen van de laatste input.

3.2.3 De boodschapverwerking

In de boodschapverwerking wordt de output van de voorbewerking opgeslagen in 16 32-bit registers. Deze worden tijdens 16 klokcycli opgevuld met de output van de voorbewerking. De volgende 48 klokcycli dienen dan om de overige 48 woorden te genereren die nodig zijn de compressiestap. Het eerste register geeft de uitgang van de boodschapverwerking. Hierdoor is de compressiestap met één klokcyclus vertraagd. Dit heeft als voordeel dat het kritische pad van de hashfunctie verkort is.

Deze module heeft 2 inputs: de 32-bit boodschap en een selectiebit, en 1 output: de 32-bit verwerkte boodschap. De selectie bit wordt hoog gehouden tijdens de eerste 16 klokcycli om de registers op te vullen.

In figuur 3.3 is een schematische voorstelling van de boodschapverwerking te zien.

3.2.4 De compressie

Deze stap vormt de 32-bit woorden van de boodschapverwerking om tot de 256-bit hash output. De stap is schematisch voorgesteld in figuur 3.4

De compressiemodule bestaat uit 2 256-bit registers, een ROM, het interne compressiealgoritme en acht 32-bit adders. Register0 slaat de tussentijdse hashwaarde op tijdens elke 64ste klokcyclus. Bij een reset wordt hier de initialisatiewaarde ingeladen. Register1 dient voor de opslag van de interne compressiewaarde tijdens elke klokcyclus. De output van het ROM wordt gekozen a.d.h.v. de 6-bit counter van de controle logica.

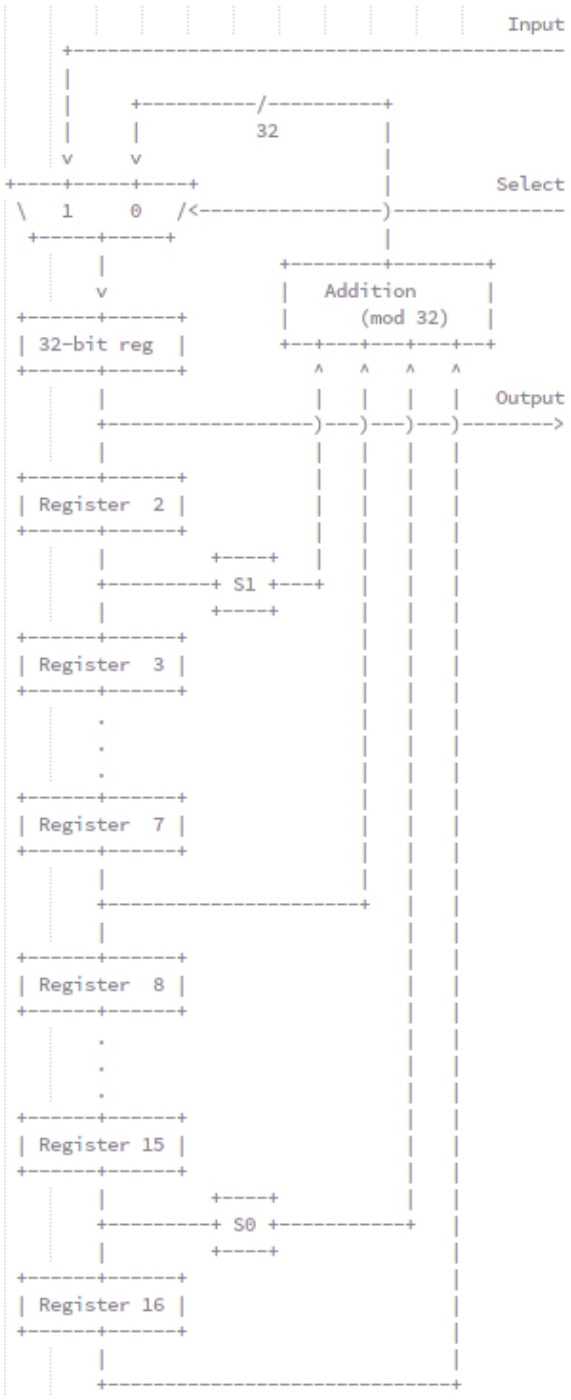
Het interne compressiealgoritme bepaalt de interne compressiewaarde a.d.h.v. de output van het ROM, de verwerkte boodschap en de output van Register0 of Register1. Tijdens elke 1ste klokcyclus wordt de inhoud van Register0 gebruikt, anders wordt er gekozen voor de inhoud van Register1.

De 32-bit adders worden gebruikt om de interne compressiewaarde samen te voegen met de tussentijdse hashwaarde. Om te voorkomen dat voor deze optelling een extra klokcyclus nodig is wordt de compressiewaarde elke cyclus bij een nulvector opgeteld. Enkel tijdens de 64ste cyclus wordt de output van Register0 gebruikt. Een schematische voorstelling hiervan is te vinden in figuur 3.5.

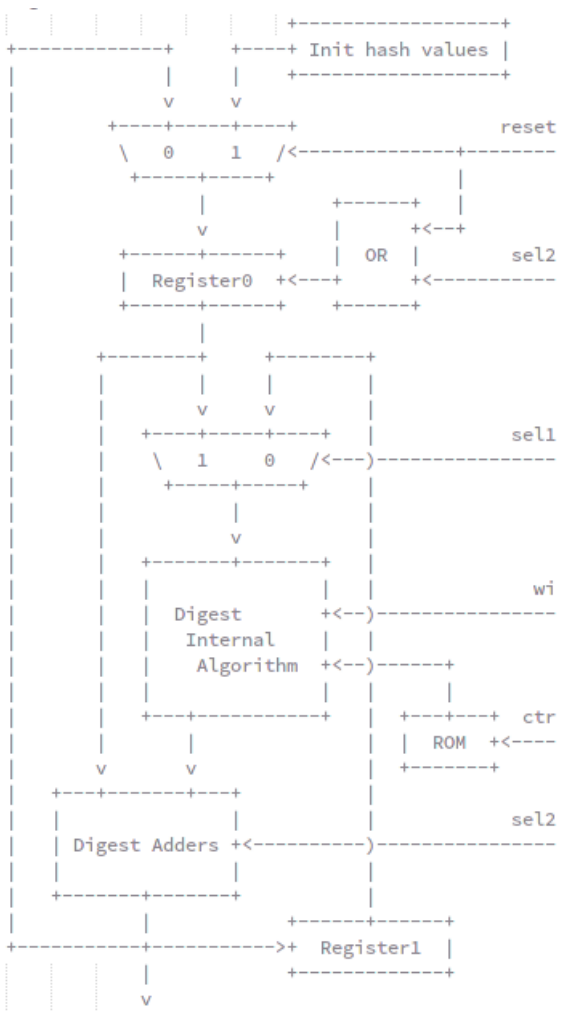
3.2.5 Beperkingen

Er zijn een aantal beperkingen aan deze implementatie. Allereerst is de lengte input altijd een veelvoud van 8. Dit komt door de beperkingen van de bytes input. In de praktijk zal deze beperking zelden tot problemen leiden gezien de input die gehashed moet worden bijna altijd bestanden of tekst zijn.

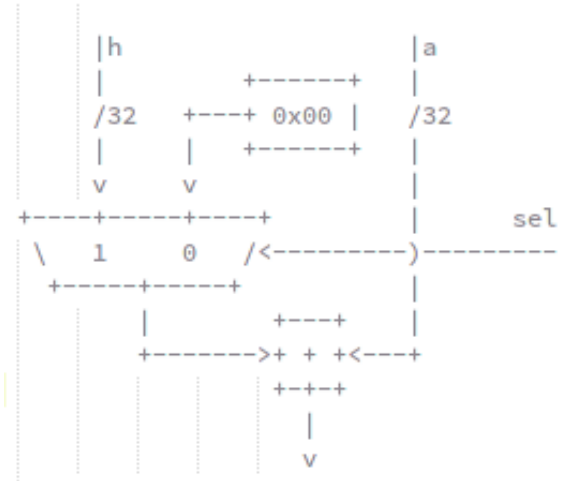
Een tweede beperking is dat er geen mogelijkheid is om de input tijdelijk te onderbreken. Dit kan problemen geven als de data die gehashed moet worden eerst door een andere module bewerkt of gegenereerd moet worden. Het is mogelijk de SHA-256 module in de toekomst uit te breiden om hiervoor ondersteuning te bieden.



Figuur 3.3: SHA boodschapverwerking



Figuur 3.4: SHA compressie



Figuur 3.5: SHA compressie optelling

Hoofdstuk 4

Besluit

De ECDH/ECDSA module is bijna volledig functioneel in Lava. Om deze te voltooien moet er enkel nog een random nummer generator toegevoegd worden. Dit is echter niet voldoende om de bibliotheek te gebruiken in het ontwerp van hardware. Uit de Lava code kan immers nog geen VHDL code gegenereerd worden.

Voordat hieraan gewerkt wordt, moet de bruikbaarheid van York Lava binnen het project overwogen worden. Deze versie van Lava wordt immers niet meer ondersteund en is niet compatibel met de huidige versie van de *Glasgow Haskell Compiler*. Hiernaast is deze versie van Lava niet geschikt om grote hardwaremodules te testen door de manier waarop simulaties uitgevoerd worden. ES&S heeft al plannen om over te stappen naar Kansas Lava. Dit kan een goed alternatief zijn maar hierover kan op dit moment nog niets met zekerheid gezegd worden.

Bibliografie

[1] Henri Cohen, Atsuko Miyaji en Takatoshi Ono. "Effcient Elliptic Curve Exponentiation Using Mixed Coordinates". In: *Lecture Notes in Computer Science* 1514 (1998), p. 51–65.

[2] Whitfield Diffie en Martin Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (sep 2006), p. 644–654.

[3] Nicky Hannosset. "Cryptografische blokken genereren met Lava". Masterscriptie. UHasselt en KULeuven, 2014.

[4] Don Johnson, Alfred Menezes en Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International Journal of Information Security* 1.1 (2001), p. 36–63.

[5] Laurie Law e.a. "An Efficient Protocol for Authenticated Key Agreement". In: *Designs, Codes and Cryptography* 28.2 (1998), p. 119–134.

[6] Peter Montgomery. "Modular Multiplication Without Trial Division". In: *Mathematics of Computation* 44 (1985), p. 519–521.

[7] Siddika Berna Örs, Lejla Batina en Bart Preneel. "Hardware Implementation of Elliptic Curve Processor over GF(p)". In: (2003).

[8] Christof Paar en Jan Pelzl. *Understanding Cryptography*. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-44649-8.

Bijlage A

ECDSA sleutel hergebruik

Uitgaande van de publieke sleutel en 2 boodschappen:

$$\begin{aligned} k_{Pub} &= (p, a, b, q, A, B) \\ m_1 &= (x_1, (r_1, s_1)) \\ m_2 &= (x_2, (r_2, s_2)) \end{aligned}$$

Als de twee handtekeningen gegenereerd werden met dezelfde tijdelijke sleutel zijn de waarden $r_1 = r_2 = r$. Hieruit kunnen dan de volgende vergelijkingen afgeleid worden:

$$\begin{cases} s_1 = (x_1 + d \cdot r) \cdot k_E^{-1} \\ s_2 = (x_2 + d \cdot r) \cdot k_E^{-1} \end{cases} \mod q$$

$$\Rightarrow \begin{cases} d = (s_1 \cdot k_E - x_1) \cdot r^{-1} \\ d = (s_2 \cdot k_E - x_1) \cdot r^{-1} \end{cases} \mod q$$

$$\Rightarrow \begin{cases} d = (s_1 \cdot k_E - x_1) \cdot r^{-1} \\ (s_1 \cdot k_E - x_1) \cdot r^{-1} = (s_1 \cdot k_E - x_1) \cdot r^{-1} \end{cases} \mod q$$

$$\Rightarrow \begin{cases} d = (s_1 \cdot k_E - x_1) \cdot r^{-1} \\ s_1 \cdot k_E - x_1 = s_1 \cdot k_E - x_1 \end{cases} \mod q$$

$$\Rightarrow \begin{cases} d = (s_1 \cdot k_E - x_1) \cdot r^{-1} \\ k_E = (x_1 - x_2) \cdot (s_1 - s_2)^{-1} \end{cases} \mod q$$

De geheime sleutel kan dus eenvoudig bekomen worden met dit laatste stelsel van vergelijkingen.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Een bibliotheek van cryptografische operaties met Lava

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2015**

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

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

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

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

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

Voor akkoord,

Sielski, Bartel

Datum: **21/08/2015**