UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nuša Zidarič

Implementacija končnih obsegov in eliptičnih krivulj z vezjem FPGA

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Ljubljana, 2011

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Nuša Zidarič

Implementacija končnih obsegov in eliptičnih krivulj z vezjem FPGA

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Aleksandar Jurišić

Somentor: doc. dr. Patricio Bulić

Ljubljana, 2011

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01776/2011
Datum: 05.09.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **NUŠA ZIDARIČ**

Naslov: **IMPLEMENTACIJA KONČNIH OBSEGOV IN ELIPTIČNIH KRIVULJ Z VEZJEM FPGA**

IMPLEMENTATION OF FINITE FIELDS AND ELLIPTIC CURVES ON A FPGA DEVICE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Delo predstavi matematične osnove, potrebne za razumevanje osnovnih računskih operacij v binarnih obsegih, ki so posebno zanimivi za kriptosisteme implementirane v strojni opremi. Osrednja motivacija je učinkovita implementacija seštevanja, kvadriranja, množenja in invertiranja na FPGA (Field Programmable Gate Array), ki se uporabljajo za računanja večkratnika točke na eliptični krivulji in so osnova za protokole zasnovane na kriptografiji javnih ključev, kot npr. digitalni podpisi. Opravi naj se še primerjava različnih algoritmov glede na prostorsko in časovno zahtevnost ter analiza rezultatov, ki omogoča izbiro optimalnih modulov.

Mentor:

prof. dr. Aleksandar Jurišić

Dekan:

prof. dr. Nikolaj Zimic

Somentor:

doc. dr. Patricio Bulić

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisana Nuša Zidarič,

z vpisno številko 63030243,

sem avtorica diplomskega dela z naslovom:

Implementacija končnih obsegov in eliptičnih krivulj z vezjem FPGA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom
  prof. dr. Aleksandra Jurišića
  in somentorstvom
  doc. dr. Patricia Bulića
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 22.12.2011 Podpis avtorice:

Zahvala

Iskreno se zahvaljujem svojemu mentorju, prof. dr. Aleksandru Jurišiću, za potrpežljivost in vso pomoč ter čas in prizadevanje, ki ju je vložil v to diplomsko delo, in prav tako doc. dr. Patriciu Buliću za koristne nasvete in pomoč pri interpretaciji rezultatov. Zahvala tudi Leonu Matohu, za nesebično pomoč in ideje.

Kazalo

Seznam uporabljenih kratic in simbolov

- $\widehat{a}(x)$ - Montgomeryjeva slika polinoma $a(x)$
- ASIC - *Application-Specific Integrated Circuit*
- $\mathcal{C}_h(\mathbb{F})$ - algebraična krivulja nad $\mathbb{F}$
- CLB - *Configurable Logic Block*
- $D(a, b)$ - največji skupni deljitelj števil a in b
- E - eliptična krivulja
- EEA - *Extended Euclidean Algorithm* - razširjen Evklidov algoritem
- $f(x)$ - nerazcepni polinom
- $\mathbb{F}$ - obseg
- FPGA - *Field Programmable Gate Array*
- G - število hkrati sprocesiranih bitov
- GF(q) - Galoisov obseg moči q
- HW - *hardware*
- IOB - *Input/Output Block*
- LSB - *Least Significant Bit*
- LUT - *Look-Up Table*
- m - razsežnost vektorskega prostora nad $\mathbb{Z}_2$
- MAP - mapiranje
- mod - modul
- MSB - *Most Significant Bit*

- $\mathbb{N}$ - množica naravnih števil
- PAR - *Place And Route* - postavitev in povezovanje
- RTL - *Register Transfer Logic*
- S - število obhodov, ki so potrebni za izračun rezultata pri izbranem G
- st($a(x)$) - stopnja polinoma $a(x)$
- VHDL - *Very-High-Speed Integrated Circuit Hardware Description Language*
- XOR - ekskluzivni ali
- $\mathbb{Z}$ - kolobar celih števil
- $\mathbb{Z}_p$ - praštevilski obseg
- $\mathbb{Z}_p[x]$ - kolobar polinomov nad obsegom polinomov $\mathbb{Z}_p$
- α - ničla nerazcepnega polinoma $f(x)$
- $\varphi(n)$ - Eulerjeva funkcija števila n
- $\lfloor a \rfloor$ - (spodnji) celi del števila a
- $\oplus$ - ekskluzivni ali
- $+_m$ - seštevanje po modulu m
- $*_m$ - množenje po modulu m

Povzetek

Kriptosistemi z javnimi ključi, ki temeljijo na eliptičnih krivuljah nad binarnim končnim obsegom GF(2^m), so čedalje bolj razširjeni. Da bi zadostili potrebam po hitrosti in prepustnosti sistemov, v katerih se uporabljajo, smo se obrnili k implementaciji aritmetike binarnih končnih obsegov in operacij nad točkami eliptične krivulje na FPGA. Raziskali smo različne algoritme za posamezne operacije in izbrali tiste z najugodnejšo časovno in prostorsko zahtevnostjo. Seštevanje elementov binarnega končnega obsega izvajamo z operatorjem XOR, kar dodatno olajša strojno implementacijo. Zelo preprosto in poceni je tudi kvadriranje. Zaradi pogostosti pojavitev je najbolj pomembna optimizacija množenja elementov izbranega končnega obsega. Najboljše rezultate dobimo z uporabo hibridnega Montgomeryjevega množilnika, ki za izračun produkta potrebuje le dve urini periodi. Množenje uporabljamo pri potenciranju v končnih obsegih in pri računanju s točkami eliptične krivulje. Zanimivo pa je, da dobimo pri potenciranju boljše rezultate, če uporabimo kombinatorično kombinatorični množilnik. Zaradi velikega števila obhodov se računanje inverza izkaže za časovno zelo zahtevno. Operacije nad točkami eliptične krivulje so pogojene z učinkovitostjo binarne aritmetike. Algoritme za seštevanje in podvojevanje točk izboljšamo s prehodom na projektivne koordinate. Le-to omogoči implementacijo seštevanja in podvojevanja z uporabo kvadriranj in množenj, časovno zahteven inverz pa je potreben le pri prehodu na običajne koordinate. Najboljše rezultate dosežemo z uporabo projektivnih koordinat Lòpez-Dahab, zato jih uporabimo tudi pri implementaciji množenja točke s skalarjem.

Ključne besede: FPGA, strojna implementacija, binarni končni obsegi, množenje Montgomery, razširjen Evklidov algoritem, eliptične krivulje, projektivne koordinate

Abstract

Nowadays, elliptic curve cryptosystems are widely distributed. Its fundamental operation is scalar multiplication kP, where P is a point of the elliptic curve and k an integer. Following the need for fast scalar multiplication, we decided for a hardware implementation on a FPGA device to achieve an adequate speed-up and increase in throughput. Hardware implementation is additionally simplified by the use of XOR gates for polynomial addition performed in GF(2^m). Squaring turned out to be quite simple and low-cost as well. As one of the most common operations in finite field arithmetic, efficiently implemented multiplication can significantly improve performance of the entire design. Best results were achieved by using the hybrid Montgomery multiplier that was able to compute the product in just two clock cycles. Exponentiation was implemented by the square and multiply algorithm, where the use of a combinatiorial multiplication circuit gave the biggest gain in performance. The best method for inversion proved to be the extended Euclidean algorithm. Exponentiation and inversion turned out to be relatively time consuming, since they both require a high number of iterations. Performance of elliptic curve arithmetic is dependent upon efficiency of binary field operations. The usage of projective coordinates significantly improves point addition and point doubling; it allows us to use only multiplications and squarings and to avoid inversion up to the last moment, i.e. the conversion of the projective point back to ordinary coordinates. Best results were achieved using projective coordinates Lopez-Dahab, which also prooved to be the best choice for point multiplication, that was implemented by a series of point additions and point doublings.

Key words: FPGA, hardware implementation, binary finite fields, Montgomery multiplication, extended Euclidean algorithm, elliptic curves, projective coordinates

Poglavje 1

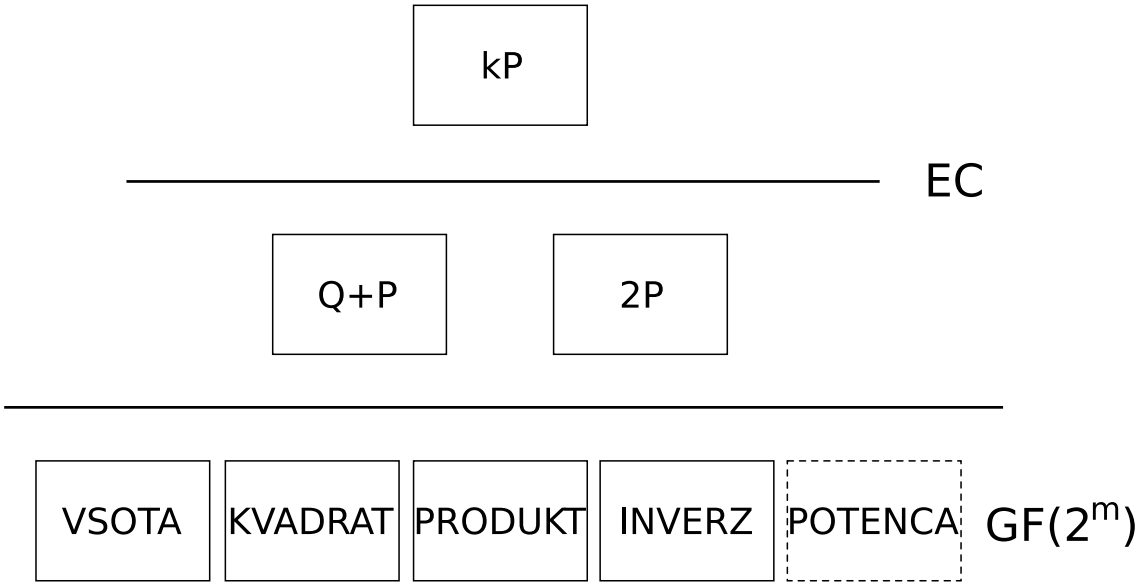
Uvod

Binarni končni obsegi igrajo veliko vlogo v kriptografiji, ki je iz dneva v dan bolj pomembna. Medtem ko se pri bločnih kriptografskih šifrah srečujemo predvsem s končnimi obsegi manjših razsežnosti, imamo v javnih kriptosistemih opravka z $\mathrm{GF}(2^m)$, kjer zadostno varnost zagotovimo z $m > 160$. Od tod tudi izvira potreba po učinkoviti implementaciji binarne aritmetike. Najpogosteje se srečamo z množenjem in kvadriranjem elementov izbranega končnega obsega. Manj pogosto, a računsko zelo zahtevno, je računanje inverza (potrebujemo ga npr. v bločnih šifrah, za računanje javnega in zasebnega ključa pri RSA in v kriptografiji z eliptičnimi krivuljami). Zelo zahtevna operacija je tudi potenciranje, ki jo uporabljata npr. Diffie-Hellmanov protokol za izmenjavo ključa in protokol za digitalno podpisovanje DSS (Digital Signature Standard). Ob vsem tem pa ne smemo pozabiti, da govorimo o sistemih, ki delujejo v realnem času. Potreba po hitrosti in prepustnosti nas vodita k strojni implementaciji osnovnih operacij, na katerih temeljijo izbrani kriptografski protokoli.

Odločili smo se za implementacijo na FPGA (*Field Programmable Gate Array*), ki omogoča enostavno preverjanje funkcionalnosti ter spreminjanje vezja, in s tem iskanje najboljših kompromisov med prostorsko in časovno zahtevnostjo algoritmov ter med njihovo arhitekturo in funkcionalnostjo. FPGA zagotavljajo potrebno hitrost in visoko prepustnost, ter omogočajo visoko stopnjo paralelizma.

Sedaj se lahko s pristopom od zgoraj navzdol vrnemo k eliptičnim krivuljam nad izbranim končnim obsegom. Postopki za generiranje ključev, podpisov in preverjanje le-teh slonijo na množenju točke P izbrane krivulje s skalarjem $k \in \mathbb{N}$. Računanje večkratnika kP je realizirano z metodo "podvoji in seštej". Operaciji podvoji ($2P$) in seštej ($Q + P$) pa sta sestavljeni predvsem iz modularnih množenj in kvadriranj elementov izbranega končnega obsega. Tako dobimo hierarhijo, prikazano na shemi 1.1, ki predstavlja temelj razvoja kriptografskega koprocesorja.

Namen tega diplomskega dela je raziskati in primerjati različne algoritme za posamezne operacije na shemi 1.1. Najučinkovitejše implementacije nižjih nivojev bomo uporabili pri realizaciji op-



Slika 1.1: Hierarhija binarne aritmetike (najnižji nivo) in operacij nad točkami eliptične krivulje

eracij višjih nivojev in na koncu predstavili najboljšo arhitekturo.

Diplomsko delo je razdeljeno na tri sklope. Prvi obsega dve uvodni poglavji. V prvem uvodnem poglavju bomo predstavili zgradbo FPGA in podrobneje razložili kriterije za primerjavo implementiranih algoritmov. Naslednje poglavje predstavi teoretične osnove končnih obsegov in eliptičnih krivulj. Drugi sklop zajema poglavji o aritmetiki in implementaciji. V prvem so predstavljeni različni algoritmi za izvajanje posameznih operacij iz sheme 1.1, v drugem pa so podrobneje opisane njihove realizacije in spremembe, ki so bile potrebne za implementacijo na FPGA. V zadnjem sklopu opišemo testne module, s katerimi smo preverili funkcionalnost implementiranih algoritmov.

Poglavje 2

Xilinx FPGA

V tem poglavju bomo predstavili uporabnost in zgradbo FPGA. Povedali bomo, kako podamo časovno in prostorsko zahtevnost vezja, ki bosta služili kot kriterija za primerjavo učinkovitosti algoritmov.

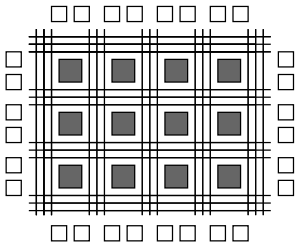
FPGA je programabilno polprevodniško vezje, ki uporabniku omogoča implementacijo poljubnega logičnega vezja, pa naj bodo to preprosta XOR vrata ali pa cel mikroprocesor. Ponašajo se z velikim številom vrat (v milijonih) in z visoko-nivojskimi sistemskimi funkcijami, kot so procesorji, zakasnitvene zanke, upravljalniki ure, spominski moduli, serijski oddajniki, itn., integriranimi na enem samem FPGA-ju [1]. Ko govorimo o majhnih proizvajalnih serijah, je FPGA cenovno veliko ugodnejši od ASIC vezij (*Application-Specific Integrated Circuit*), saj se izognemo visokim stroškom tovarniške izdelave, splošnonamenske mikroprocesorje pa prekaša tako po zmogljivosti (*throughput* in *performance gain*) kot po hitrosti (100-kratna pohitritev [2]). Velika prednost FPGA-jev je njihova fleksibilnost; omogočajo preprosto spreminjanje tako načrtovanega kot tudi že implementiranega vezja, kar v veliki meri izkoriščajo načrtovalci ASIC vezij [3, 4]. Danes najdemo FPGA-je že povsod, vgrajeni so v satelite, letala, Mars Rover-ja, modeme, sisteme za prepoznavanje obraza, pri implementaciji aplikacij, ki potrebujejo hitre števce, veliko število registrov, omogočajo hitre cevovodne realizacije itn. Omogočajo npr. zelo učinkovito implementacijo bločnih šifer, saj gre pri zamikih in permutacijah le za "drugače ožičene" povezave med posameznimi pomnilnimi celicami, substitucijske tabele pa shranimo v registre. Zaradi že omenjene visoke prepustnosti se uporabljajo tudi v sistemih za odkrivanje vdorov (network intrusion detection system), kjer je potrebno preiskati vsak bajt vsakega paketa [2]. Omenimo še, da FPGA-ji omogočajo visoko stopnjo paralelizma ter enostavno spreminjanje parametrov algoritma (v diplomskem delu tako spreminjamo izbrani nerazcepni polinom).

Programiranje FPGA naprave se prične s specifikacijo vezja (Design Entry). Tu imamo dve možnosti: opis funkcionalnosti vezja z uporabo nekega HDL (Hardware Description Language) ali pa z vnosom sheme vezja (v grafičnem vmesniku sestavimo vezje iz elementov, ki so na voljo).

Mi bomo uporabili VHDL (Very-High-Speed Integrated Circuit HDL). Po tem koraku običajno sledijo simulacije, s katerimi preverjamo pravilnost načrta in po potrebi spremenimo vezje. Sledi FPGA sinteza, ki prevede naše vezje v logična vrata glede na izbrano ciljno napravo. Nato sledi mapiranje (MAP), v katerem se sintetizirano vezje preslika na dejanske resurse izbranega FPGA. Naslednji korak je postavitev in povezovanje (Place and Route - PAR), ki izbere kar se da optimalno postavitev logičnih blokov in čimkrajše povezave med njimi. Optimizacije, ki jih prevajalniki naredijo med postopki sinteze, MAP in PAR, so odvisne od izbranega FPGA, razvijalec pa ima tudi možnost postavljanja uporabniških omejitev (user constraints) , ki jih morajo ti postopki upoštevati (npr. ali naj optimizira za hitrejše ali za manjše vezje). Na voljo so nam tudi različna orodja za analizo vezij, npr. za časovno analizo ali pa analizo porabe moči. Zadnji korak je seveda dejansko programiranje FPGA-ja. Pri tem se datoteke, ki jih generirajo prej opisani procesi, prenesejo in zapišejo v SRAM celice znotraj FPGA.

2.1 Osnovna zgradba FPGA vezij

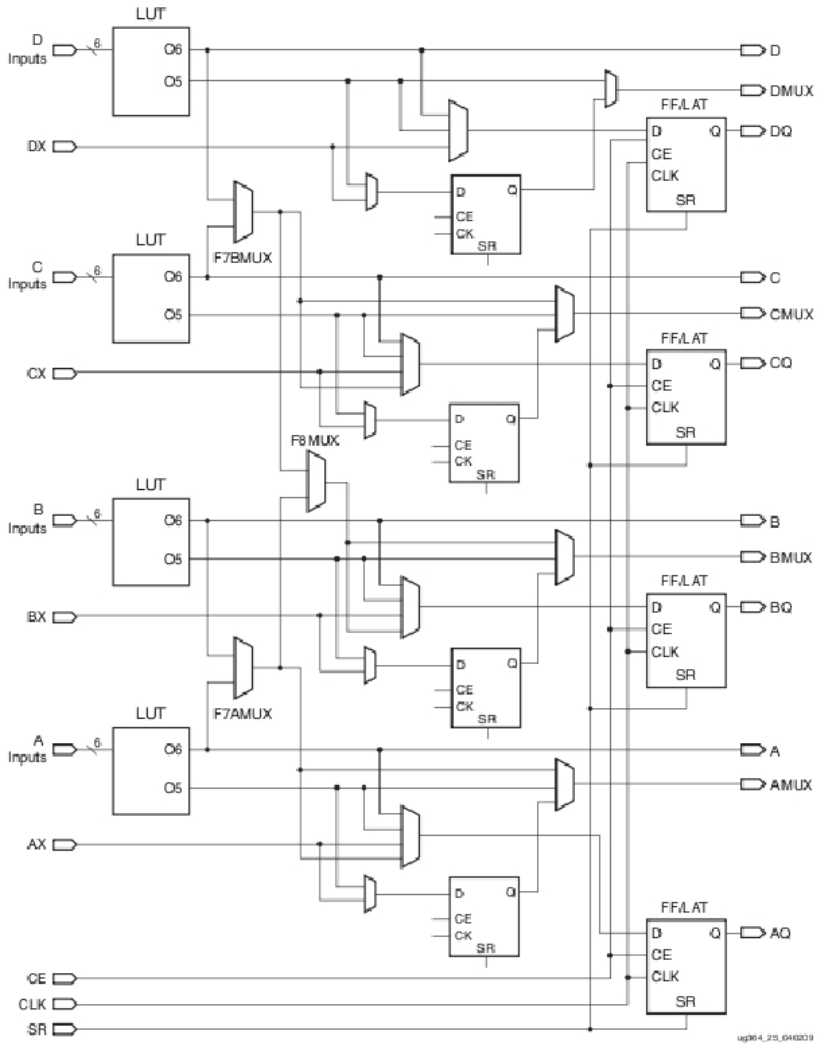
Osnovno zgradbo FPGA prikazuje slika 2.1. Sestavljen je iz velikega števila identičnih logičnih celic (CLB), ki so v bistvu osnovni gradniki vezja. Te celice so postavljene v matriko, obkroženo s posebnimi vhodno/izhodnimi celicami (IOB), povezuje pa jih mreža vodoravnih in navpičnih povezovalnih kanalov [5, 6].



Slika 2.1: Osnovna zgradba FPGA: večji sivi bloki prikazujejo CLB-je, manjši bloki ob robu so IOB-ji, med njimi pa vidimo povezovalne kanale

2.1.1 CLB - Configurable Logic Block

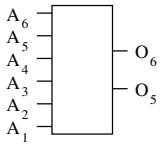
Virtex-6 FPGA-ji imajo CLB razdeljen na dve podobni rezini (*slice*). Posamezna rezina je sestavljena iz štirih LUT-ov (Look-up Table), ki jim pravimo tudi funkcijski generatorji, pomnilnih celic in multiplekserjev [5, 7]. Poenostavljeno shemo rezine prikazuje shema 2.2.



Slika 2.2: Poenostavljena shema rezine v Virtex-6 CLB-ju [8]

LUT

n-vhodni LUT (slika 2.3) je pomnilno polje z 2^n besedami, v katere shranimo logično tabelo želene n-vhodne preklopne funkcije. Vhodi v LUT so naslovni signali, ki izberejo ustrezen bit, njegova vsebina pa se prenese na izhod LUT-a. Vsak LUT ima dva izhoda O_5 in O_6. Kadar želimo realizirati neko 6-vhodno Booleanovo funkcijo, dobimo njeno vrednost na izhodu O_6. Z enim LUT-m pa lahko realiziramo tudi dve 5-vhodni preklopni funkciji (ki imata seveda enake vrednosti vhodov), njuni vrednosti pa dobimo na obeh izhodih, O_5 in O_6 [8]. Izhoda O_5 in O_6 posameznega LUT-a lahko: (i) peljemo direktno na izhod rezine, (ii) uporabimo v dodatni logiki (npr. carry logic), (iii) peljemo na vhod v pomnilno celico ali pa (iv) na vhod enega izmed treh multipleksorjev, ki omogočajo realizacijo 7- in 8-vhodnih preklopnih funkcij. Za realizacijo 7-vhodne funkcije sta na voljo multipleksorja F7AMUX in F7BMUX (glej shemo 2.2), preko katerih povežemo dva LUT-a (LUT A in B ali pa LUT C in D). Za realizacijo 8-vhodne funkcije pa je na voljo še F8MUX (slika 2.2), preko katerega povežemo vse štiri LUT-e v rezini. Funkcije z več kot osem vhodi realiziramo s povezovanjem večjega števila rezin.
Zakasnitev pri prehodu signala skozi LUT ni odvisna od realizirane funkcije (vsebine LUT-a), prav tako tudi ni odvisna od tega, ali z LUT-om realiziramo eno funkcijo ali dve.



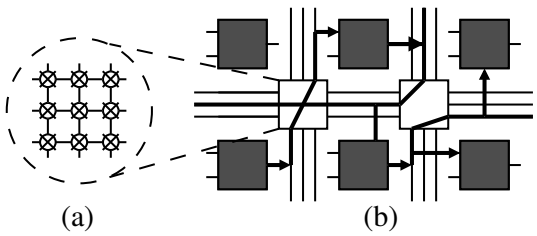
Slika 2.3: Shematski prikaz 6-vhodnega LUT-a

Pomnilne celice

Vsaka rezina vsebuje štiri pomnilne celice, ki jih lahko konfiguriramo kot D-pomnilne celice (flip-flop) ali kot zatiče (latch), in pa štiri dodatne D-pomnilne celice, ki jih lahko uporabimo le, ko so osnovne pomnilne celice konfigurirane kot D-celice [8].
Na vhod osnovne pomnilne celice lahko pripeljemo izhod O_5 ali O_6 pripadajočega LUT-a, pripadajoč "bypass" vhod v rezino (gre mimo LUT-a), ali pa izhod multiplekserjev F7AMUX, F7BMUX ali F8MUX.
Povedali smo že, da rezini nista popolnoma enaki. Razlikujeta se po dodatni logiki, ki omogoča npr. učinkovitejšo realizacijo 32-bitnega pomikalnega registra, porazdeljenega RAM-a , ripple carry seštevalnika, itn.

2.1.2 Povezovalni kanali

FPGA ima vodoravne in navpične povezovalne kanale, sestavljene iz različno dolgih prenosnih segmentov in tranzistorskih stikal (pass tranzistor), ki povezujejo segmente in omogočajo vhode in izhode posameznih CLB-jev [6]. Uporaba različno dolgih segmentov zmanjša latenco posamezne prenosne poti. Za najdaljše prenose uporablja dolge povezave, t.i. "longlines", brez vmesnih stikal, ki bi upočasnila signal. Najkrajši segmenti povezujejo med seboj sosednje CLB-je. Vodoravni in navpični segmenti so med seboj povezani s stikali, oziroma t.i. "switchbox-i", ki vsebujejo stikala, s katerimi vzpostavimo povezavo med dvema segmentoma. Če v SRAM celico zapišemo vrednost 1, se tranzistorsko stikalo obnaša kot zaprto stikalo, in povezava med dvema prenosnima segmentoma je vzpostavljena; če pa shranimo vrednost 0, povezavo prekinemo [9]. Na sliki 2.4 (b) vidimo CLB-je (sive barve) in mrežo povezovalnih kanalov s switchboxi. Odebeljene črte predstavljajo različne povezave [2].



Slika 2.4: Povezovalni kanali: (a) stikala (switchbox); (b) različne povezave med CLB-ji

2.1.3 IOB - Input/Output Block

Matrika CLB-jev je obdana z IOB-ji, kot prikazuje shema 2.1. IOB omogoča konfiguracijo posameznih nožic (pinov) FPGA kot vhode, izhode ali visokoimpedančno stanje. Vsebuje tudi pomnilne celice, ki omogočajo zakasnitev signalov ter serijsko-paralelne in paralelno-serijske pretvornike [10].

2.2 Učinkovitost implementacije

Prednosti strojne implementacije algoritmov so hitrost (čas potreben za en izračun), visoka prepustnost (veliko število sprocesiranih bitov na sekundo) in visoka stopnja paralelizma (če jo seveda algoritem dopušča). Ko govorimo o kompleksnosti implementacije, nas zanimata predvsem časovna in prostorska zahtevnost (pri HW realizacijah so pomembne tudi npr. informacije o porabi energije).

Kadar govorimo o prostorski zahtevnosti FPGA, govorimo o številu logičnih vrat in o uporabljenih pomnilnih celicah. Orodje Xilinx-ISE med implementacijo generira poročila o zasedenosti virov na FPGA-ju, torej o številu porabljenih rezin, LUT-ov, pomnilnih celic, IOB-jev, itn. Podatki o virih, ki so na voljo na Virtex-6 FPGA-ju, so zbrani v tabeli A.1 v dodatku A.
Časovna zahtevnost pa se nanaša na zakasnitev posameznega modula, to je na čas, ki preteče od trenutka, ko so na voljo vhodni podatki (v modul oz. kar FPGA, če se nanaša na najvišji "top" modul), do trenutka, ko se na izhodu pojavi rezultat. Če lahko algoritem realiziramo s kombinatoričnim vezjem (brez pomnilnih elementov), je časovna zahtevnost algoritma enaka kar zakasnitvi signala po kritični poti skozi vezje (pot je zaporedje povezovalnih in logičnih elementov [1]). Tu se pojavi povezava med časovno in prostorsko zahtevnostjo implementacije. Večje število zasedenih CLB-jev pomeni daljše poti signalov skozi vezje, to pa pomeni večjo zakasnitev. Časovno zahtevnost pri sekvenčnih vezjih podamo z urino periodo, ki je odvisna od največje zakasitve v modulu (kritične poti), ter skupnim časom (sk. čas = število ciklov $\times$ urina perioda). V nadaljevanju bomo primerjali različne algoritme za realizacijo osnovnih operacij v GF(2^m) glede na njihovo časovno in prostorsko zahtevnost. V končni dizajn bomo vključili tiste module, ki so dali najboljše rezultate. Videli bomo, da ne moremo upoštevati le njihove časovne ali prostorske zahtevnosti (v teoriji bi lahko imeli le kombinatorična vezja z majhno zakasnitvijo, vendar bi nam kaj kmalu zmanjkalo CLB-jev), temveč iščemo najboljši kompromis med hitrostjo in zasedenim prostorom.

Poglavje 3

Matematično ozadje

Začeli bomo z osnovami teorije števil in pokazali pravilnost Evklidovega algoritma. Nadaljevali bomo s kongruencami ter praštevilskim obsegom $\mathbb{Z}_p$. Definirali bomo polinome ter kolobar polinomov $\mathbb{Z}_p[x]$. Po predstavitvi nerazcepnih polinomov bomo skonstruirali končni obseg $\mathrm{GF}(2^m)$, njegove lastnosti pa bomo pokazali tudi na primeru. Na koncu tega poglavja se bomo seznanili z eliptičnimi krivuljami in predstavili operacije seštevanja, podvojevanja in skalarnega množenja točk izbrane eliptične krivulje nad $\mathrm{GF}(2^m)$.

3.1 Osnove teorije števil

3.1.1 Deljivost

Celo število a deli celo število b, kar označimo z $a \,|\, b$, ko je b (celi) večkratnik števila a. V splošnem imamo za $|b| \geq |a|$ le dve možnosti:

1. število b je večkratnik števila $|a|$ ($b = q|a|$ za nek $q \in \mathbb{Z}$),
2. število b leži med dvema zaporednima večkratnikoma števila $|a|$ na številski premici 3.1, to je med $q|a|$ in $(q+1)|a|$ (glej [11]).

Številu $q := \lfloor \frac{b}{a} \rfloor$ pravimo *kvocient*, številu $r := b - qa$ pa *ostanek*. Na sliki 3.1 vidimo, da je ostanek r razdalja med b in najbližjim levim večkratnikom števila a.

Slika 3.1: Številska premica: $|b| = q|a| + r$, $q = \lfloor \frac{b}{a} \rfloor$ in $r = b \mod a$

S tem smo prišli do osnovnega izreka o deljenju:

Izrek 3.1 [Osnovni izrek o deljenju] *Naj bosta a in b poljubni naravni števili in $b \geq a$. Potem obstajata natanko določeni števili $q \in \mathbb{N}$ in $r \in \mathbb{N}_0$, tako da je $b = aq + r$, $0 \leq r < a$.*

Če neko naravno število d deli a in če d deli tudi b, pravimo, da je d skupni delitelj števil a in b. To pa pomeni tudi, da d deli njuno linearno kombinacijo ($d \,|\, a$ in $d \,|\, b \Rightarrow d \,|\, ax + by$). Ker ima vsako število le končno mnogo deliteljev, obstaja tudi le končno mnogo skupnih deliteljev števil a in b (razen ko $a = b = 0$). Če je vsaj en izmed a in b različen od 0, potem največjemu izmed skupnih deliteljev pravimo *največji skupni delitelj* in ga označimo z $D(a, b)$ (glej [12]).

Trditev 3.2 *Za največji skupni delitelj d števil a in b, tj. $D(a, b) = d$, velja:*

1. *če je d skupni delitelj a in b, potem $d \,|\, a$ in $d \,|\, b$,*
2. *če neko število c deli a in b, potem $c \,|\, d$,*
3. *$D(a, b) = D(a, b \pm a)$.* □

Evklidov algoritem

Osnovni izrek o deljenju nam pove, da za poljubni naravni števili a in b, $b \geq a$, obstajata $q_0 \in \mathbb{N}$ in $r_0 \in \mathbb{N}_0$, tako da je $b = q_0 a + r_0$, kjer je $0 \leq r_0 < a$. Če je $r_0 \neq 0$, postopek ponovimo, tj. osnovni izrek o deljenju nam zagotovi števili $q_1 \in \mathbb{N}$ in $r_1 \in \mathbb{N}_0$ tako, da je $a = q_1 r_0 + r_1$, $0 \leq r_1 < r_0$. Na ta način dobimo strogo padajoče zaporedje ostankov $r_i : r_0 > r_1 > \cdots \geq 0$. Po končnem številu korakov je za nek $n \in \mathbb{N}$ $r_n = 0$:

$$\begin{array}{rcrcll} r_{-2} & = & b & = & q_0 a + r_0 & 0 \leq r_0 < a \\ r_{-1} & = & a & = & q_1 r_0 + r_1 & 0 \leq r_1 < r_0 \\ & & r_0 & = & q_2 r_1 + r_2 & 0 \leq r_2 < r_1 \\ & & & \vdots & & \\ & & r_{n-3} & = & q_{n-1} r_{n-2} + r_{n-1} & 0 \leq r_{n-1} < r_{n-2} \\ & & r_{n-2} & = & q_n r_{n-1} + r_n & 0 = r_n < r_{n-1} \end{array} \tag{3.1}$$

Enačbe (3.1) lahko zapišemo tako, da izrazimo ostanke kot $r_i = r_{i-2} - q_i r_{i-1}$ za $i = 0, 1, \ldots, n$. Iz točke 3 trditve 3.2 potem sledi:

$$D(a, b) = D(r_{-2}, r_{-1}) = D(r_{-1}, r_0) = \cdots = D(r_{n-1}, r_n) = r_{n-1} = d$$

Predzadnji enačaj velja zaradi $r_n = 0$ in $D(r_{n-1}, 0) = r_{n-1}$.
Pokazali smo, da lahko z zaporedjem enačb (3.1) poiščemo največji skupni delitelj števil a in b. Rešiti želimo

$$ax + by = c \tag{3.2}$$

kjer je $a, b, c \in \mathbb{Z}$ in $(x, y) \in \mathbb{Z} \times \mathbb{Z}$. Potreben pogoj zato je $D(a, b) \mid c$. Izkaže se, da je tudi zadosten. Z razširjenim Evklidovim algoritmom poiščimo rešitev $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ enačbe

$$ax + by = d \tag{3.3}$$

Dve skrajnosti, ki ju takoj opazimo, sta

$$\begin{array}{rcl} a \cdot 0 + b \cdot 1 & = & b \\ a \cdot 1 + b \cdot 0 & = & a \end{array} \tag{3.4}$$

Z Evklidovim algoritmom bomo poiskali zaporedji p_i in q_i, za kateri velja (glej [15]): $ap_i + bq_i = r_i$. Iz enačb 3.4 dobimo začetne vrednosti $r_{-2} = b,\ p_{-2} = 0,\ p_{-1} = 1, r_{-1} = a,\ p_{-1} = 1$ in $q_{-1} = 0$. Nato po zgledu (3.1) na vsakem koraku poiščemo števili s_i in r_i, tako da je $r_{i-2} = s_i r_{i-1} + r_i$, kjer je $0 \leq r_i < r_{i-1}$. Pripadajoči stevili p_i in q_i potem izračunamo po

$$\begin{array}{rcl} p_i & = & p_{i-2} - s_i p_{i-1} \\ q_i & = & q_{i-2} - s_i q_{i-1} \end{array} \tag{3.5}$$

Opisani postopek ponavljamo, dokler ne dobimo $r_n = 0$. Rešitev enačbe (3.3) je potem $x = p_{n-1}$, $y = q_{n-1}$, r_n pa je enak d, rešitev enačbe (3.3) pa $x = \frac{c}{d} \cdot p_{n-1}$ in $y = \frac{c}{d} \cdot q_{n-1}$.
Poglejmo si razširjeni Evklidov algoritem 1, ki izračuna $d = D(a, b)$, ter rešitev (x, y) enačbe (3.3); glej [13]. V korakih 5 in 6 algoritma 1 postavimo začetne vredosti parametrov p_i in q_i, tako kot narekujeta enačbi (3.4). V koraku 8 izračunamo kvocient s, v koraku 9 pa ostanek r. V koraku 10 izračunamo nove vrednosti za p in q, kot narekujeta enačbi (3.5). V korakih 11 do 13 pa popravimo vrednosti parametrov za vstop v naslednjo iteracijo.

Algoritem 1 Razširjen Evklidov algoritem

VHOD: nenegativni števili a, b, $b \geq a$
IZHOD: $d = D(a, b)$ in rešitev $(x, y) = (p, q)$ enačbe $ax + by = d$

if a=0 **then**
    $d \leftarrow b, \quad p \leftarrow 1, \quad q \leftarrow 0$
    return (d, p, q)
end if
$p_1 \leftarrow 0$, $q_1 \leftarrow 1$
$p_2 \leftarrow 1$, $q_2 \leftarrow 0$
while $a > 0$ **do**
    $s \leftarrow \lfloor b/a \rfloor$
    $r \leftarrow b - sa$
    $p \leftarrow p_2 - sp_1, \quad q \leftarrow q_2 - sq_1$
    $b \leftarrow a, \quad a \leftarrow r$
    $p_2 \leftarrow p_1, \quad p_1 \leftarrow p$
    $q_2 \leftarrow q_1, \quad q_1 \leftarrow q$
end while
$d \leftarrow b, \quad p \leftarrow p_2, \quad q \leftarrow q_2$
return (d, p, q)

3.1.2 Kongruence

Naj bo m neko naravno število. Pravimo, da sta celi števili a in b kongruentni po modulu m, kar zapišemo $a \equiv b \pmod{m}$, če pri deljenju z m dasta isti ostanek, tj. natanko takrat, ko modul m deli razliko $a - b$.

"Biti kongruenten po modulu m" je ekvivalenčna relacija. Od tod sledi, da množica $\mathbb{Z}$ razpade na m ekvivalenčnih razredov. Poglejmo, kaj to pomeni za poljubno celo število a. Kot smo omenili na začetku 3.1.1, obstajata natanko določeni števili $q, r \in \mathbb{Z}$, da velja $a = mq + r, 0 \leq r < m$, oz. $mq = a - r$. Od tod je razvidno, da m deli razliko $a - r$ in da sta števili a in r pripadnika istega ekvivalenčnega razreda. Označimo ga z $[r]$.

Za poljubno celo število a vedno velja, da je kongruentno po modulu m natanko enemu izmed vseh možnih ostankov pri deljenju z m, torej natanko enemu $r \in \{0, 1, \ldots, m-1\}$. Množici $R = \{0, 1, \ldots, m-1\}$ pravimo tudi *popolni sistem ostankov*. Če vzamemo podmnožico množice R, ki vsebuje le tiste elemente $r_i \in R$, za katere velja $D(r_i, m) = 1$, govorimo o *reduciranem sistemu ostankov po modulu* m.

Omenimo še Mali Fermatov izrek in Eulerjevo posplošitev tega izreka, ki ju bomo potrebovali kasneje. Dokaz Eulerjevega izreka si lahko ogledate v [14].

Izrek 3.3 **[Fermat]** *Za poljubno celo število a in praštevilo p, ki ne deli števila a, velja:*

$$a^{p-1} \equiv 1 \pmod{p}$$

Lema 3.4 *Za naravno število n je Eulerjeva funkcija $\varphi(n)$ število naravnih števil iz intervala $[1, n]$, ki so z n tuja.*

Povejmo, da je $\varphi(1) = 1$, da je za praštevilo p $\varphi(p) = p - 1$ in da je $\varphi(p^k) = p^{k-1}\varphi(p)$.

Izrek 3.5 **[Euler]** *Če je $D(m, a) = 1$, potem velja: $a^{\varphi(m)} \equiv 1 \pmod{m}$.*

3.2 Teorija števil in algebra

3.2.1 Praštevilski obseg $\mathbb{Z}_p$

Popolni sistem ostankov po modulu m, tj. množica $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$, z operacijo $+_m$ tvori Abelovo grupo $(\mathbb{Z}_m, +_m)$, kjer oznaka $+_m$ pomeni seštevanje po modulu m. Pokažimo, da ima $(\mathbb{Z}_m, +_m)$ res vse lastnosti (pogledamo si jih lahko v Dodatku B), ki so potrebne za Abelovo grupo:

- množica $\mathbb{Z}_m$ je zaprta za seštevanje po modulu m;
- operacija $+_m$ je asociativna, saj je $a + (b + c) \equiv (a + b) + c \pmod{m}$;
- enota za seštevanje je 0;
- vsak element svoj inverz: inverz od 0 je 0, vsi ostali elementi $a \in \mathbb{Z}_m$ pa imajo inverz $m - a$;
- operacija $+_m$ je komutativna, saj je $a + b \equiv b + a \pmod{m}$.

Oglejmo si sedaj reduciran sistem ostankov po modulu m, tj. množico

$$\mathbb{Z}_m^* = \{a \in \mathbb{Z}_m \,;\, D(a, m) = 1\}.$$

Le-ta skupaj z operacijo množenja po modulu m tvori Abelovo grupo $(\mathbb{Z}_m^*, *_m)$ (multiplikativna grupa). Enota za množenje po modulu m je 1, vsi elementi iz $\mathbb{Z}_m^*$ so obrnljivi, operacija $*_m$ pa je asociativna in komutativna. Opozorimo na to, da element $0 \in \mathbb{Z}_m$ ni vsebovan v $\mathbb{Z}_m^*$ saj je $D(0, m) = m$. Število elementov množice $\mathbb{Z}_m^*$, oz. moč grupe $(\mathbb{Z}_m^*, *_m)$, je enako $\varphi(m)$. V primeru, da je m praštevilo, velja $\mathbb{Z}_m \backslash \{0\} = \mathbb{Z}_m^*$. Omenimo še, da za operaciji $+_m$ in $*_m$ velja zakon distributivnosti. Tako smo prišli do praštevilskega končnega obsega $(\mathbb{Z}_p, +_p, *_p)$, kjer je p praštevilo.

3.2.2 Polinomi nad končnimi obsegi

Polinomi spremenljivke x nad obsegom $\mathbb{Z}_p$, kjer je p praštevilo, so izraz oblike

$$a(x) = \sum_{i=0}^{m} a_i x^i, \qquad \text{s koeficienti } a_i \in \mathbb{Z}_p \text{ in } m \geq 0$$

Množico polinomov s koeficienti iz $\mathbb{Z}_p$ označimo z $\mathbb{Z}_p[x]$. Največji indeks m, za katerega je $a_m \neq 0$, imenujemo stopnja polinoma $a(x)$, kar zapišemo $\mathrm{st}(a(x)) = m$. Temu koeficientu pravimo vodilni koeficient, koeficientu a_0 pa konstantni koeficient. Polinom, ki ima vse koeficiente razen konstantnega enake 0, je konstanten polinom, njegova stopnja je 0. Ničelni polinom, torej polinom, ki ima vse koeficiente enake 0, pa ima stopnjo $-\infty$. Polinoma $a(x)$ in $b(x)$ sta *enaka*, če so paroma enaki vsi njuni koefiecienti, torej če $a_i = b_i$ za vsak $i \in [0, m]$.
Oglejmo si, kako vpeljemo vsoto in produkt v $\mathbb{Z}_p[x]$:

- **Vsota** polinomov $a(x) = \sum_{i=0}^{n} a_i x^i$ in $b(x) = \sum_{i=0}^{m} b_i x^i$ je definirana kot

  $$a(x) + b(x) = \sum_{i=0}^{\ell} (a_i + b_i) x^i$$

  kjer je $\ell \leq \max\{m, n\}$. Vidimo, da seštevamo po komponentah.

- **Produkt** polinomov $a(x) = \sum_{i=0}^{n} a_i x^i$ in $b(x) = \sum_{i=0}^{m} b_i x^i$ je definiran kot

  $$c(x) = a(x)b(x) = \sum_{k=0}^{n+m} c_k x^k$$

  kjer je $c_k = \sum_{i+j=k} a_i b_j$ (konvolucija). Stopnja produkta je pri tem enaka vsoti stopenj faktorjev, torej $\mathrm{st}(c(x)) = \mathrm{st}(a(x)) + \mathrm{st}(b(x))$.

Sedaj se ni težko prepričati, da je $(\mathbb{Z}_p[x], +)$ Abelova grupa, da je $\mathbb{Z}_p[x]$ zaprta za množenje in da je množenje asociativno, ter da seštevanje in množenje povezuje distributivnost. Govorimo torej o kolobarju polinomov $\mathbb{Z}_p[x]$ nad obsegom $\mathbb{Z}_p$.
Če za nek polinom $f(x)$ in nek $a \in \mathbb{Z}_p$ velja $f(a) = 0$, rečemo, da je a ničla polinoma $f(x)$. Takrat je $f(x)$ deljiv z $(x - a)$. Vse kar smo prej povedali o številih velja tudi za polinome:

Izrek 3.6 *Za vsak polinom $f(x)$ in vsak u iz $\mathbb{Z}_p$ obstaja polinom $g(x)$ tako, da velja $f(x) = (x - u)g(x) + f(u)$, kjer je $\mathrm{st}(g(x)) = \mathrm{st}(f(x)) - 1$.* □

Največji skupni delitelj: Polinom $d(x)$, ki ima vodilni koeficient enak 1, je največji skupni delitelj polinomov $f(x)$ in $g(x)$, če velja:

1. $d(x)|f(x)$ in $d(x)|g(x)$,
2. vsak polinom $c(x)$, ki deli polinoma $f(x)$ in $g(x)$, deli tudi polinom $d(x)$, in je manjše ali enake stopnje kot polinom $d(x)$.

Trditev 3.7 *$d(x)$ lahko zapišemo kot linearno kombinacijo polinomov $f(x)$ in $g(x)$, tj. obstajata polinoma $p(x)$ in $q(x)$ tako, da velja:*

$$d(x) = f(x)p(x) + g(x)q(x) \tag{3.6}$$

Izrek 3.8 [Osnovni izrek o deljenju za polinome] *Za poljubna polinoma $f(x)$ in $g(x)$, kjer $g(x) \neq 0$, obstajata natanko določena polinoma $r(x)$ in $s(x)$, da velja $f(x) = s(x)g(x) + r(x)$, kjer je* $\mathrm{st}(r(x)) < \mathrm{st}(g(x))$. □

Pravimo, da sta polinoma $a(x)$, $b(x) \in \mathbb{Z}_p[x]$ **kongruentna** po modulu polinoma $t(x)$, kar zapišemo $a(x) \equiv b(x) \pmod{t(x)}$, če modul $t(x)$ deli razliko $a(x) - b(x)$. Polinom $t(x)$ razbije $\mathbb{Z}_p[x]$ na ekvivalenčne razrede polinomov iz $\mathbb{Z}_p[x]$, ki imajo stopnjo manjšo od stopnje polinoma $t(x)$. To množico ekvivalenčnih razredov označimo z $\mathbb{Z}_p[x]/t(x)$.

Nerazcepni polinomi

Neničelni polinom $f(x) \in \mathbb{Z}_p[x]$ je *nerazcepen* nad $\mathbb{Z}_p$, če ga ne moremo zapisati kot produkt dveh polinomov $f(x) = g(x)h(x)$, pri čemer sta $g(x), h(x) \in \mathbb{Z}_p[x]$ in imata oba stopnjo večjo od 0. Za vsak $m \in \mathbb{N}$ obstaja nerazcepen polinom stopnje m nad $\mathbb{Z}_p$ [15]. **Primer:** Oglejmo si vse kvadratne polinome iz $\mathbb{Z}_2[x]$:

- $x^2 = x \cdot x$
- $x^2 + 1 = (x + 1)(x + 1)$
- $x^2 + x = x(x + 1)$
- $x^2 + x + 1$

Vidimo, da je edini nerazcepni polinom stopnje 2 nad $\mathbb{Z}_2$ polinom $x^2 + x + 1$.

3.2.3 Končni obseg GF(2^m)

Za končni obseg $\mathbb{Z}_p$ in poljubno naravno število m bomo skonstruirali končni obseg s $q = p^m$ elementi in ga bomo označili GF(q), kjer je GF(q) skrajšano za Galoisov obseg (Galois field). Naj bosta m in n naravni števili, $m > n$, in $f(x) = x^m + x^n + 1$ nerazcepni polinom stopnje m

nad $\mathbb{Z}_p$. Potem je $\mathbb{Z}_p[x]/f(x)$ obseg polinomov nad $\mathbb{Z}_p$, reduciranih po modulu polinoma $f(x)$. Matematiki so pokazali, da lahko poljuben končen obseg s $q = p^m$ elementi prevedemo na to obliko; glej [15].
Galoisov obseg GF(q) s $q = p^m$ elementi lahko obravnavamo kot m-razsežni vektorski prostor nad obsegom $\mathbb{Z}_p$. Potem množici $\{1, x, \ldots, x^{m-1}\}$ pravimo polinomska baza obsega GF(q); glej [15].
Polinom $a(x) = \sum_{i=0}^{m-1} a_i x^i$ stopnje m lahko zapišemo tudi kot m-terico koeficientov

$$(a_m, a_{m-1}, \ldots, a_1, a_0).$$

Povejmo še to, da je vsak element $a \in \mathbb{Z}_p$ hkrati element GF(q), saj lahko smatramo a kot konstanten polinom in ga zapišemo kot $(0, \ldots, 0, a)$ in ima stopnjo 0. Enota za množenje v $\mathbb{Z}_p$ tako postane enota za množenje $(0, \ldots, 0, 1)$ v GF(q). Kolobar polinomov pa iz obsega $\mathbb{Z}_p$ deduje tudi komutativnost množenja. Polinom x pa zapišemo kot $(0, \ldots, 0, 1, 0)$. Tako le-ta skupaj z vsemi polinomi $(0, \ldots, 0, a)$, $a \in \mathbb{Z}_p$ generira GF(q). Omenimo še, da $\mathbb{Z}_p[x]$ nima netrivialnih deliteljev niča, torej da je $a(x)b(x) = 0$ natanko takrat, ko velja $a(x) = 0$ ali $b(x) = 0$. Neničelni elementi iz GF(q) tvorijo multiplikativno grupo GF(q)*, v kateri so vsi elementi obrnljivi. Vidimo, da je GF(q) obseg. Povejmo še, da je GF(q)* ciklična grupa reda $q-1$, zato za vsak $a \in GF(\mathrm{q})^*$ velja $a^{q-1} = 1$.

Primer: Končni obseg GF(2^4) generiran z nerazcepnim polinomom $f(x) = x^4 + x + 1$
Naj bo α ničla polinoma $f(x)$. Elementi obsega GF(2^4) nad $\mathbb{Z}_2$ so oblike $a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$, kjer so koeficienti $a_i \in \mathbb{Z}_2$. GF(2^4) ima $2^4 = 16$ elementov, ki so prikazani v tabeli 3.1: v prvem stolpcu vidimo vektorski zapis oblike (a_3, a_2, a_1, a_0), v drugem stolpcu je pripadajoči polinom, v tretjem stolpcu je le-ta zapisan kot produkt dveh (predhodnih) polinomov, v zadnjem stolpcu pa je isti element zapisan še kot potenca ničle α. Vidimo lahko, da je npr. $\alpha^3+1 = (\alpha+1)(\alpha^2+\alpha+1)$, kar je pisano s potencami enako $\alpha^{10}\alpha^4 = \alpha^{14}$.
Primer seštevanja: $(\alpha^3 + 1) + (\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha$
Primer množenja: $(\alpha^3 + 1)(\alpha^2 + \alpha + 1) = \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1 \equiv \alpha^3 + \alpha \pmod{f(x)}$
Oglejmo si, kako smo prišli do tega rezultata. Delamo po modulu $f(x)$, kar pomeni, da je $f(\alpha) = 0$, torej je $\alpha^4 = \alpha + 1$. To enačbo nato pomnožimo še z α, in dobimo $\alpha^5 = \alpha^2 + \alpha$. Ti dve enačbi vstavimo v zgornji produkt namesto členov α^4 in α^5, in tako dobimo reduciran produkt $\alpha^3 + \alpha$. Več o redukciji bomo povedali v 4.1.1.

Za potrebe diplomske naloge smo izbrali binarne končne obsege GF(2^m), $m \in \mathbb{N}$, generirane z

$a_3a_2a_1a_0$	polinom	faktorizacija	potenca α
0000	0		0
0001	1		$\alpha^{15} = 1$
0010	α		α
0011	$\alpha + 1$	$\alpha^2\alpha^2$	α^4
0100	α^2		α^2
0101	$\alpha^2 + 1$	$(\alpha + 1)(\alpha + 1)$	α^8
0110	$\alpha^2 + \alpha$	$\alpha(\alpha + 1)$	α^5
0111	$\alpha^2 + \alpha + 1$	$(\alpha^2 + \alpha)(\alpha^2 + \alpha)$	α^{10}
1000	α^3		α^3
1001	$\alpha^3 + 1$	$(\alpha + 1)(\alpha^2 + \alpha + 1)$	α^{14}
1010	$\alpha^3 + \alpha$	$\alpha(\alpha + 1)$	α^9
1011	$\alpha^3 + \alpha + 1$	$\alpha^3(\alpha + 1)$	α^7
1100	$\alpha^3 + \alpha^2$	$\alpha^2(\alpha + 1)$	α^6
1101	$\alpha^3 + \alpha^2 + 1$	$(\alpha^2 + 1)(\alpha^2 + \alpha)$	α^{13}
1110	$\alpha^3 + \alpha^2 + \alpha$	$\alpha(\alpha^2 + \alpha + 1)$	α^{11}
1111	$\alpha^3 + \alpha^2 + \alpha + 1$	$\alpha^2(\alpha^2 + \alpha + 1)$	α^{12}

Tabela 3.1: Elementi končnega obsega GF(2^4)

nerazcepnimi polinomi oblike $f(x) = x^m + x^n + 1$, $m, n \in \mathbb{N}, m > n$.
Operaciji seštevanja in odštevanja sta v primeru binarnega končnega obsega enaki in ju izvajamo z XOR operatorjem (ekskluzivni ali) nad posameznimi biti polinomov. Pri računanju produkta in kvadrata se pojavijo polinomi stopnje, manjše od $2m - 1$, ki zahtevajo redukcijo po modulu $f(x)$. Deljenje pa nadomestimo z množenjem z inverznim elementom.

m	$f(x)$
11	$x^{11} + x^2 + 1$
63	$x^{63} + x + 1$
113	$x^{113} + x^{15} + 1$
191	$x^{191} + x^9 + 1$

Tabela 3.2: Razsežnost izbranih obsegov GF(2^m) in ustrezni nerazcepni polinomi

3.2.4 Eliptične krivulje nad GF(2^m)

Naj bo $h(x, y) : \mathbb{F}^2 \to \mathbb{F}$ poljuben polinom nad obsegom $\mathbb{F}$. Množici točk iz $\mathbb{F}^2$, za katere velja $h(x, y) = 0$, pravimo algebraična krivulja (označimo jo $\mathcal{C}_h(\mathbb{F})$). Simbolično zapisano:

$$\mathcal{C}_h(\mathbb{F}) = \{(x, y) \in \mathbb{F}^2; h(x, y) = 0\}$$

Tistim točkam krivulje, ki so večkratne ničle polinoma $h(x, y)$, pravimo singularne. Kadar je polinom $h(x, y)$ stopnje 3 nerazcepen nad $\mathbb{F}$ in njegova krivulja nima singularnih točk, govorimo o *eliptični krivulji*, ki jo označimo z E; za več informacij glej [12]. Zanimajo nas tiste kubične

krivulje, pri katerih lahko z uporabo pravila *sekante in tangente* iz nekaj znanih točk krivulje najdemo nove točke na krivulji. V ta namen definiramo binarno operacijo AB takole: naj bosta A in B točki na krivulji in L premica skozi ti točki. Ta premica seka krivuljo v tretji, enolično določeni točki AB. Ko sta točki A in B različni, govorimo o sekanti, če pa točki sovpadata, torej $A = B$, govorimo o tangenti v tej točki. Če je tangenta vzporedna z y-osjo pravimo, da krivuljo seka v točki v neskončnosti.
Vendar pa operacija AB na eliptični krivulji ne definira grupe, saj ne obstaja taka točka $\mathcal{O}$, da bi veljalo $A\mathcal{O} = A$. S pomočjo operacije AB bomo definirali *seštevanje* $A + B$ *na eliptični krivulji*, oz. v primeru $A = B$ *podvojevanje točke* $2A$, tako da bodo točke krivulje skupaj z operacijo seštevanja tvorile (Abelovo) grupo:

- izberemo poljubno točko $\mathcal{O} \in E$ (lahko tudi točko v neskončnosti);
- s točkama $A, B \in \mathcal{O}$ skonstruiramo točko AB;
- potegnemo premico skozi točki $\mathcal{O}$ in AB, ki seka krivuljo v tretji točki $A + B$. Torej velja $A + B = \mathcal{O}(AB)$.

Tako množica točk eliptične krivulje skupaj z operacijo seštevanja tvori Abelovo grupo $(E, +)$:

- za poljubne točke $A, B, C \in E$ velja $(A + B) + C = A + (B + C)$ (*asociativnost*);
- za vsako točko $A \in E$ velja $A + \mathcal{O} = \mathcal{O} + A = A$ (*enota*);
- za vsako točko $A \in E$ obstaja taka točka $B \in E$, da velja $A + B = B + A = \mathcal{O}$ (*nasprotni element*);
- za poljubni točki $A, B \in E$ velja $A + B = B + A$ (*komutativnost*).

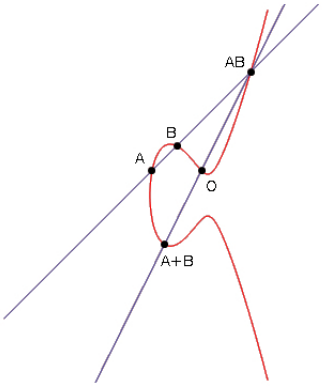
Primer: Oglejmo si seštevanje točk $A = (-3, 2)$ in $B = (-1, 4)$ na krivulji $y^2 = x^3 - 7x + 10$ z izbrano točko $\mathcal{O} = (1, 2)$, prikazano na sliki 3.2.. Tako dobimo točki $AB = (5, 10)$ in $A + B = (-2, -4)$.

Opozorimo še na to, da izbira točke $\mathcal{O}$ vpliva na definicijo seštevanja. Izberimo dve točki $\mathcal{O}, \mathcal{O}' \in E$ in naj bo $A + B$ seštevanje z enoto $\mathcal{O}$, ter $A \oplus B$ seštevanje z enoto $\mathcal{O}'$. Potem dobimo grupi $G = (E, +)$ in $H = (E, \oplus)$, ki pa sta izomorfni, tj. $G \cong H$.

Bralec si lahko več o eliptičnih krivuljah prebere v [16] in [17], mi pa bomo sedaj prešli na eliptično krivuljo nad GF(2^m):

$$y^2 + xy = x^3 + ax^2 + b \tag{3.7}$$

Za enoto $\mathcal{O}$ izberemo točko v neskončnosti ∞. Nasprotni element točke $P = (x_1, y_1)$ je točka $-P = (x_1, x_1 + y_1)$, $P + (-P) = \infty$. Poglejmo še, kako je na dani krivulji definirano seštevanje.



Slika 3.2: Seštevanje točk na krivulji $y^2 = x^3 - 7x + 10$

Vsota točk $P = (x_1, y_x)$, $Q = (x_2, y_2) \in E$, $P \neq \pm Q$ je točka $R(x_3, y_3)$ s koordinatama:

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \end{aligned} \tag{3.8}$$

V primeru, da sta točki enaki, torej $P = Q$, pa namesto o vsoti govorimo o **podvojevanju točke**, s katerim dobimo točko $2P = R = (x_3, y_3)$ s koordinatama:

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a \\ y_3 &= x_1^2 + \lambda x_3 + x_3 \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned} \tag{3.9}$$

Grafični prikaz uporabe pravila sekante in tangente z izbrano točko $\mathcal{O}$ v neskončnosti si lahko ogledate na slikah B.1 in B.2 v dodatku B.

Množenje točke krivulje P s skalarjem k je v bistvu prištevanje točke P same sebi k-krat. Skalarni produkt kP dobimo z zaporedjem seštevanj in podvojevanj.

Poglavje 4

Aritmetika

V prvem delu tega poglavja bomo podrobneje predstavili GF(2^m) aritmetiko. Ogledali si bomo različne pristope k množenju, kvadriranju, potenciranju in računanju inverza. Učinkovitost osnovnih operacij, npr. množenja in kvadriranja, je zelo pomembna za učinkovitost, denimo, potenciranja in pa operacij nad točkami eliptične krivulje, ki si jih bomo ogledali ob koncu tega razdelka.

4.1 Množenje

Ogledali si bomo štiri algoritme za množenje, ki se razlikujejo po pristopu k množenju in po načinu redukcije.

4.1.1 Klasično množenje

Produkt $d(x) = a(x)b(x)$, kjer sta $a(x), b(x) \in \mathrm{GF}(2^m)$, lahko zapišemo v matrični obliki:

$$\begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ \vdots \\ d_{m-1} \\ d_m \\ \vdots \\ \vdots \\ d_{2m-1} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & & \dots & 0 \\ a_1 & a_0 & & \dots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ a_{m-2} & \vdots & & \ddots & 0 \\ a_{m-1} & a_{m-2} & & & a_0 \\ 0 & a_{m-1} & \ddots & & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & & a_{m-1} & a_{m-2} \\ 0 & 0 & \dots & 0 & a_{m-1} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix}$$

ki nam da naslednjo formulo za računanje koeficientov d_k produkta $d(x)$:

$$d_k = \begin{cases} \sum_{i=0}^{k} a_i b_{k-i}, & k = 0, \ldots, m-1; \\ \sum_{i=k}^{2m-2} a_{k-i+(m-1)} b_{i-(m-1)i}, & k = m, \ldots, 2m-2; \end{cases} \tag{4.1}$$

Redukcija

Produkt $d(x)$ je polinom stopnje, manjše od $2m-1$, zato je potrebna redukcija:

$$\begin{aligned} c(x) &= d(x) \bmod f(x) \\ &= (d_{2m-2}x^{2m-2} + \cdots + d_1 x + d_0) \bmod f(x) \\ &= c_{m-1}x^{m-1} + \cdots + c_1 x + c_0 \end{aligned} \tag{4.2}$$

Delamo po modulu $f(x)$, kar pomeni, da je $f(x) = 0$, torej velja $x^m = x^n + 1$. To enačbo nato večkrat množimo z x in dobimo naslednje zveze za redukcijo:

$$\begin{array}{lllll} x^m & = & x^n + 1 & & \\ x^{m+1} & = & x^{n+1} + x & & \\ \ldots & & & & \\ x^{2m-n-1} & = & x^{n+(m-n-1)} + x^{m-n-1} & = & x^{m-1} + x^{m-n-1} \\ x^{2m-n} & = & x^{n+m-n} + x^{m-n} & = & x^{m-n} + x^n + 1 \\ \ldots & & & & \\ x^{2m-2} & = & x^{n+m-2} + x^{m-2} & = & x^{m-2} + x^{2n-2} + x^{n-2} \end{array} \tag{4.3}$$

Prva možnost je, da iz vhodnega vektorja $\boldsymbol{d} = (d_{2m-2}, \ldots, d_m, d_{m-1}, \ldots, d_1, d_0)$ naredimo štiri vektorje $\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y}$ in $\boldsymbol{z}$ dolžine m, reduciran vektor $\boldsymbol{c}$ pa je njihova vsota; glej [18]. Začnemo z vektorjem $\boldsymbol{w} = (d_{m-1}, \ldots, d_1, d_0)$, ki predstavlja spodnjo polovico vhodnega vektorja $\boldsymbol{d}$, in pod bite vektorja $\boldsymbol{w}$ na ustrezna mesta pišemo bite $d_{2m-2}, \ldots, d_m$, kot to narekujejo enačbe (4.3). Vektor $\boldsymbol{x}$ je zgornja polovica vektorja $\boldsymbol{d}$, vektor $\boldsymbol{y}$ je za n mest v levo pomaknjen $\boldsymbol{x}$, vektor $\boldsymbol{z}$ pa vsebuje zgornje n bite, ki pri levem pomiku izpadejo iz vektorja $\boldsymbol{y}$. Tako dobljeni vektorji so prikazani v tabeli 4.1, prazna mesta v tabeli pa predstavljajo bit 0. Opisan postopek je zapisan v algoritmu 4.1.1.

$w\rightarrow$	d_{m-1}	d_{m-2}	$\ldots$	$\ldots$	d_n	$\ldots$	$\ldots$	$\ldots$	d_0	
$y\rightarrow$	d_{2m-n-1}	d_{2m-n-2}	$\ldots$	$\ldots$	d_m					$\downarrow$
$x\rightarrow$		d_{2m-2}	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	d_m	$\oplus$
$z\rightarrow$			d_{2m-1}	$\ldots$	d_{2m-n}		d_{2m-1}	$\ldots$	d_{2m-n}	$\downarrow$
$c\rightarrow$	c_{m-1}	c_{m-2}	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	c_0	

Tabela 4.1: Redukcija s štirimi vektorji

Algoritem 2 Redukcija s štirimi vektorji

VHOD: $d = [d_{2m-2}, \ldots, d_m, d_{m-1}, \ldots, d_1, d_0]$
IZHOD: $c = [c_{m-1}, \ldots, c_1, c_0]$
1: $w \leftarrow [d_{m-1}, \ldots, d_1, d_0]$
2: $x \leftarrow [0, d_{2m-2}, \ldots, d_m]$
3: $y \leftarrow x \cdot x^n = [d_{2m-n-1}, \ldots, d_m, 0, \ldots, 0]$
4: $z \leftarrow [0, \ldots, 0, d_{2m-2}, \ldots, d_{2m-n}, 0, d_{2m-2}, \ldots, d_{2m-n}]$
5: $c \leftarrow w \oplus x \oplus y \oplus z$
6: **return** c

Drugi pristop k redukciji uporablja redukcijsko matriko. Redukcijo zapišemo v matrični obliki:

$$\boldsymbol{c}^T = E\boldsymbol{d}^T \tag{4.4}$$

kjer je E $m \times (2m-1)$ razsežna matrika oblike

$$E = \underbrace{\begin{bmatrix} 1 & 0 & & \ldots & 0 \\ 0 & 1 & & \ldots & 0 \\ \vdots & 0 & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & & & 1 \end{bmatrix}}_{m} \underbrace{\begin{bmatrix} r_{0,0} & \ldots & r_{0,m-2} \\ r_{1,0} & \ldots & r_{1,m-2} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ r_{m-1,0} & \ldots & r_{m-1,m-2} \end{bmatrix}}_{m-1}$$

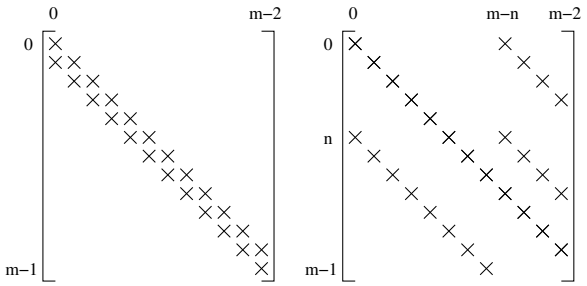
Stolpci $E^0, \ldots, E^{m-1}$ predstavljajo identično matriko I_m, ki le "prekopira" bite $d_0 \ldots d_{m-1}$ vektorja $\boldsymbol{d}$, stolpci $E^m, \ldots, E^{2m-2}$ pa predstavljajo $m \times (m-1)$ matriko R za redukcijo bitov $d_m \ldots d_{2m-2}$.

$$R = \begin{bmatrix} r_{0,0} & \ldots & r_{0,m-2} \\ r_{1,0} & \ldots & r_{1,m-2} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ r_{m-1,0} & \ldots & r_{m-1,m-2} \end{bmatrix} \tag{4.5}$$

Računanje elementov matrike R:

$$r_{j,i} = \begin{cases} f_j, & j = 0, \ldots, m-1, i = 0; \\ r_{j-1,i-1} + r_{m-1,i-1} r_{j,0}, & j = 0, \ldots, m-1, i = 1, \ldots, m-2; \end{cases} \tag{4.6}$$

kjer je f_j j-ti bit nerazcepnega polinoma $f(x)$. Enačba (4.6) velja za poljubne polinome$f(x)$, mi pa smo se omejili na uporabo trinomov. Obliko matrike R za trinome $f(x)$ lahko vidimo na sliki 4.1.



Slika 4.1: Redukcijska matrika za $f(x) = x^m + x^n + 1$, (a) $n = 1$ in (b) $1 < n < \frac{m}{2}$

4.1.2 Množenje s prepletanjem

Množenje s prepletanjem (interleaved multiplication) je realizirano s pomiki in seštevanji, redukcija pa se izvaja sproti (prepleteno); glej [18].
Produkt $c(x)$ lahko zapišemo kot:

$$\begin{aligned} c(x) &= a(x)b(x) \bmod f(x) = a(x) \cdot \left(\sum_{i=0}^{m-1} b_i x^i \right) \bmod f(x) \\ &= \left(\sum_{i=0}^{m-1} b_i a(x) x^i \right) \bmod f(x) \\ &= \left(b_0 a(x) + b_1 a(x) x + \cdots + b_{m-1} a(x) x^{m-1} \right) \bmod f(x) \\ &= (b_0 a(x) + b_1 a(x) x + \cdots + b_{m-1}(\ldots(a(x)x)x)\ldots)x) \bmod f(x) \end{aligned} \tag{4.7}$$

Po splošnem členu $b_i a(x) x^i$ vidimo, da v primeru $b_i = 1$ k produktu prištejemo za i v levo pomaknjen polinom $a(x)$. V zadnji vrstici enačbe (4.7) vidimo, da pomikanje vektorja $\boldsymbol{a}$ poteka postopno, v vsakem koraku le za en bit v levo. Pri pomiku vektorja $\boldsymbol{a}$ dobimo vektor stopnje m, zato tudi v vsakem koraku izvedemo redukcijo, pri čemer uporabimo naslednjo zvezo: $x^m = f_{m-1}x^{m-1} + \cdots + f_1 x + f_0$. Redukcija delnega produkta $d(x)$:

$$\begin{aligned} d(x) &= a(x) \cdot x = a_0 x + a_1 x^2 + \cdots + a_{m-1} x^m \\ &= a_0 x + a_1 x^2 + \cdots + a_{m-1}(f_{m-1}x^{m-1} + \cdots + f_1 x + f_0) \\ &= a_{m-1} f_0 + x(a_0 + a_{m-1} f_1) + \cdots + x^{m-1}(a_{m-2} + a_{m-1} f_{m-1}) \\ &= d_0 + d_1 x + \cdots + d_{m-1} x^{m-1} \end{aligned}$$

kjer je $d_0 = a_{m-1} f_0$ in $d_i = a_{a-1} + a_{m-1} f_i$ za $i = 1, \ldots, m-1$.

4.1.3 Množenje Mastrovito

Množenje Mastrovito je zasnovano na uporabi Mastrovito matrike $Z = Z(a)$, ki združuje množenje in redukcijo v enem koraku; glej [18]. Naj bo $Z = (Z_{ij})$ $m \times m$ razsežna matrika. Njene elemente definiramo na naslednji način:

$$z_{i,j} = \begin{cases} a_i, & i = 0, \ldots, m-1, j = 0; \\ u(i-j)a_{i-j} + \sum_{t=0}^{j-1} p_{j-1-t,i} a_{m-1-t}, & j, i = 0, \ldots, m-1, j \neq 0; \end{cases} \tag{4.8}$$

kjer je $u(\mu)$ stopnica:

$$u(\mu) = \begin{cases} 1, & \mu \geq 0; \\ 0, & \mu < 0; \end{cases}$$

in je $(m-1) \times m$ razsežna matrika $P = (P_{ij})$ funkcija nerazcepnega polinoma $f(x)$:

$$\begin{bmatrix} x^m \\ x^{m+1} \\ \\ \vdots \\ \\ x^{2m-2} \end{bmatrix} = P \begin{bmatrix} 1 \\ x \\ \\ \vdots \\ \\ x^{m-1} \end{bmatrix}$$

Elemente matrike P računamo po enačbi:

$$p_{i,j} = \begin{cases} f_j & i = 0; \\ p_{i-1,m-1} & i = 1, \ldots, m-2, j = 0; \\ p_{i-1,j-1} + p_{i-1,m-1} p_{0,j} & i = 1, \ldots, m-2, j = 1, \ldots, m-1; \end{cases} \tag{4.9}$$

Matrika P je v bistvu transponirana matrika R iz (4.5), tj. $P = R^T$.
Naslednji izrek nam pove, da lahko produkt dveh polinomov po modulu polinoma $f(x)$ izračunamo z množenjem vektorja s pravkar definirano matriko Z.

Izrek 4.1 *Naj bo $f(x)$ polinom stopnje m in $c(x) = a(x)b(x) \bmod f(x)$, kjer sta $a(x)$ in $b(x)$ polinoma stopnje manjše od m, ter $Z = Z(a)$ Mastrovito matrika, ki ustreza polinomu $a(x)$. Potem velja*

$$\boldsymbol{c}^T = Z\boldsymbol{b}^T \tag{4.10}$$

Dokaz. Oglejmo si razcep matrike Z na $Z = L + P^T U = L + RU$, kjer sta L in U matriki

velikosti $m \times m$ oziroma $(m-1) \times m$ naslednje oblike:

$$L = \begin{bmatrix} a_0 & 0 & 0 & \dots & 0 \\ a_1 & a_0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ a_{m-1} & a_{m-2} & \dots & \dots & a_0 \end{bmatrix}, \qquad U = \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \dots & a_1 \\ 0 & 0 & a_{m-1} & \dots & a_2 \\ \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & a_{m-2} \\ 0 & 0 & \dots & 0 & a_{m-1} \end{bmatrix}$$

Produkt $c(x) = a(x)b(x) \bmod f(x)$ lahko zapišemo v obliki:

$$\boldsymbol{c}^T = Z\boldsymbol{b}^T = (L + P^T U)\boldsymbol{b}^T = L\boldsymbol{b}^T + P^T U\boldsymbol{b}^T \tag{4.11}$$

Vpeljimo še dve oznaki: $\boldsymbol{g} = L\boldsymbol{b}^T$ in $\boldsymbol{e} = U\boldsymbol{b}^T$. Zgornjo enačbo sedaj zapišemo v obliki:

$$\boldsymbol{c}^T = \boldsymbol{g} + P^T\boldsymbol{e} = \boldsymbol{g} + R\boldsymbol{e} \tag{4.12}$$

Računanje elementov vektorjev $\boldsymbol{g}$ in $\boldsymbol{e}$:

$$g_i = \sum_{k=0}^{i} a_k b_{i-k}, \quad i = 0, \dots, m-1$$

$$e_i = \sum_{k=i}^{m-2} a_{m-1+(k-i)} b_{k+1}, \quad i = 0, \dots, m-1$$

Iz enačb (4.12) in (4.1) sledi izrek 4.1, saj lahko produkt $\boldsymbol{d}$ zapišemo v obliki

$$\boldsymbol{d} = \begin{bmatrix} \boldsymbol{g} \\ \boldsymbol{e} \end{bmatrix}$$

□

4.1.4 Množenje Montgomery

Namesto $c(x) = a(x)b(x) \bmod f(x)$, računamo Montgomeryjev produkt (glej [18, 19]):

$$c(x) = a(x)b(x)r^{-1}(x) \bmod f(x) \tag{4.13}$$

kjer je $r(x)$ fiksen element iz GF(2^m), za katerega velja D($r(x), f(x)$) = 1. To pomeni, da sta $r(x)$ in $f(x)$ tuja, in torej obstajata polinoma $r^{-1}(x)$ in $f'(x)$, tako da velja:

$$r(x) \cdot r^{-1}(x) + f(x) \cdot f'(x) = 1 \tag{4.14}$$

kjer je $r^{-1}(x)$ inverz polinoma $r(x)$ po modulu $f(x)$. Učinkovitost Montgomeryjevega produkta je odvisna od izbire elementa $r(x)$; [18, 19]. Izbran polinom je $r(x) = x^m$, in ker je $f(x)$

nerazcepen polinom (ni deljiv z x), za $r(x)$ velja: $\mathrm{D}(r(x), f(x)) = 1$.
Iz $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1x + f_0$ sledi
$r(x) = x^m = f_{m-1}x^{m-1} + \cdots + f_1x + f_0$, torej $r(x) = (f_{m-1}, \ldots, f_1, f_0)$, oz.
$r(x) = (0, \ldots, 0, f_n, 0, \ldots 0, f_0)$ za trinom $f(x) = x^m + x^n + 1$.

Algoritem 3 Množenje Montgomery

VHOD: $a(x), b(x), r(x), f'(x)$
IZHOD: $c(x) = a(x)b(x)r^{-1}(x) \bmod f(x)$
1: $t(x) \leftarrow a(x)b(x)$
2: $u(x) \leftarrow t(x) \cdot f('x) \bmod r(x)$
3: $c(x) \leftarrow [t(x) + u(x) \cdot f(x)] / r(x)$

Prvi korak algoritma 3 je navadno množenje polinomov. Drugi korak je množenje po modulu $r(x)$, ki je zaradi izbranega $r(x) = x^m$ zelo hitro, saj preprosto zavržemo člene stopnje $\geq m$. Prav tako hitro in preprosto je deljenje z $r(x)$ v tretjem koraku; izvedeno je s pomikom števca m bitov desno.

Pokažimo, da je izhod algoritma 2 res želen Montgomeryjev produkt (enačba 4.13). Iz $u(x) = t(x) \cdot f'(x) \bmod r(x)$ sledi, da obstaja tak polinom $k(x)$, da velja $u(x) = t(x)f'(x) + k(x)r(x)$, iz enačbe 4.14 pa sledi $f(x)f'(x) = 1 + r(x)r^{-1}(x)$. Tako lahko izračun v koraku 3 zapišemo kot:

$$\begin{aligned}
\frac{t(x) + u(x)f(x)}{r(x)} &= \frac{t(x) + t(x)f'(x)f(x) + k(x)r(x)f(x)}{r(x)} \\
&= \frac{t(x) + t(x)\left(1 + r(x)r^{-1}(x)\right) + k(x)r(x)f(x)}{r(x)} \\
&= \frac{t(x) + t(x) + t(x)r(x)r^{-1}(x) + k(x)r(x)f(x)}{r(x)} \\
&= \frac{t(x)r(x)r^{-1}(x) + k(x)r(x)f(x)}{r(x)} \\
&= \frac{r(x)[t(x)r^{-1}(x) + k(x)f(x)]}{r(x)} \\
&= t(x)r^{-1}(x) + k(x)f(x) \\
&= t(x)r^{-1}(x) \bmod f(x) = a(x)b(x)r^{-1}(x) \bmod f(x)
\end{aligned}$$

Predzadnji enačaj je posledica izbire polinoma $k(x)$, ki nam zagotovi, da je dobljen izraz na levi stopnje kvečjemu $m - 1$. Predhodnemu izračunu polinoma $f'(x)$ v drugem koraku se lahko popolnoma izognemo, če jemljemo bite polinoma $a(x)$ enega za drugim. Najprej zapišemo

Montgomeryjev produkt (4.13) v obliki:

$$c(x) = x^{-m}a(x)b(x) \bmod f(x) = x^{-m}\sum_{i=0}^{m-1} a_i x^i b(x) \bmod f(x)$$

Naj bo $t(x) = \sum_{i=0}^{m-1} a_i x^i b(x) = (a_{m-1}x^{m-1}+\cdots+a_1x+a_0)b(x)$. Produkt $t(x)$ izračunamo tako, da začnemo s tistim bitom vektorja $a(x)$, ki ima največjo težo (*Most Significant Bit*, v nadaljevanju MSB):

```
t(x):=0;
for  j = M − 1 to 0 do
      t(x)=t(x)+a(i)b(x)
      t(x)=xt(x)
end for
```

Ker je $x^{-m}(a_{m-1}x^{m-1}+\cdots+a_1x+a_0) = (a_{m-1}x^{-1}+\cdots+a_{-m+1}x+a_0x^{-m})$, računamo produkt v obratni smeri, torej od bita z najmanjšo težo (*Least Significant Bit*, krajše LSB) k MSB:

```
t(x):=0;
for j = 0 to M-1 do
      t(x)=t(x)+a(i)b(x)
      t(x)=t(x)/x
end for
```

Za seštevanjem vrinemo še en korak, s katerim zagotovimo, da je $t(x)$ vedno deljiv z x: v primeru, da je $t_0 = 1$, vektorju $t(x)$ prištejemo vektor $f(x)$. Po končani zanki vrinemo še dva koraka, tako da nam algoritem vrne produkt $c(x) = a(x)b(x) \bmod f(x)$, ne pa Montgomeryjevega produkta (enačba 4.13). Sedaj lahko zapišemo algoritem za računanje produkta na bitnem nivoju (algoritem 4). Gre za sekvenčno izvedbo Montgomeryjevega množenja, ki v enem ciklu izvede en obhod zanke. Algoritem vrne z x^m pomnožen (korak 7) in reduciran (klic funkcije **reduce** v koraku 8) produkt. Število urinih period lahko občutno zmanjšamo, če v enem obhodu zanke izračunamo več delnih produktov.
Vektor $a(x) = \sum_{i=0}^{m-1} a_i x^i = (a_{m-1}, \ldots, a_1, a_0)$ najprej razdelimo na besede enake dolžine. Izberemo nek G, za katerega velja $M = G \cdot S$, kjer je G dolžina besede v bitih:

$$a(x) = \sum_{i=0}^{S-1} A_i(x) \cdot x^{iG} = A_{S-1}x^{G(S-1)} + \cdots + A_1x^G + A_0 = (A_{S-1}, \ldots, A_1, A_0)$$

kjer je $A_i = (a_{iG+G-1}, \ldots, a_{iG+1}, a_{iG})$. V tretjem koraku algoritma 3 z $A_i \cdot b(x)$ izračunamo G bitov namesto enega, v zadnjem koraku iteracije pa delni produkt množimo z x^{-G} po modulu $f(x)$. Da je delni produkt $c(x) = c(x) + A_ib(x)$ sploh deljiv z x^G, mu moramo prišteti nek večkratnik polinoma $f(x)$.

Algoritem 4 Množenje Montgomery na nivoju bitov

VHOD: $a(x), b(x)$
IZHOD: $c(x) = a(x)b(x) \bmod f(x)$
1: $c(x) \leftarrow 0$
2: **for** $i = 0$ **to** $m - 1$ **do**
3: $\quad c(x) \leftarrow c(x) + a_i \cdot b(x)$
4: $\quad c(x) \leftarrow c(x) + c_0 \cdot f(x)$
5: $\quad c(x) \leftarrow c(x)/x$
6: **end for**
7: $c(x) \leftarrow c(x) \cdot x^m$
8: $c(x) \leftarrow$ **reduce**$(c(x))$
9: **return** $c(x)$

Če velja, da $c(x) \neq 0 \mod x^G$, obstaja tak polinom $M(x)$, stopnje manjše od G, da velja:

$$c(x) + M(x) \cdot f(x) = 0 \pmod{x}^G \tag{4.15}$$

Ker se moramo znebiti le spodnjih G od nič različnih bitov z najmanjšo težo, je dovolj, če delnemu produktu prištejemo le G-bitni polinom $F_0'(x)$ namesto celotnega polinoma $f(x)$. Naj bo $C_0(x)$ beseda z najmanjšo težo polinoma $c(x)$ in $F_0(x)$ beseda z najmanjšo težo polinoma $f(x)$. Iz enačbe (4.15) sledi, da je $M(x) = C_0(x)F_0^{-1}(x) \pmod{x}^G$, kjer je $F_0^{-1}(x) \pmod{x}^G$ enak polinomu $F_0'(x)$, saj po enačbi (4.14) velja

$$x^{SG} \cdot x^{-SG} + f(x) \cdot f'(x) = 1 \pmod{x}^G$$

$$F_0(x) \cdot F_0'(x) = 1 \pmod{x}^G$$

Postopek za Montgomeryjevo množenje na besednem nivoju je podan v algoritmu 5.

Algoritem 5 Množenje Montgomery na nivoju besed

VHOD: $a(x), b(x), f(x), F_0'(x)$
IZHOD: $c(x) = a(x)b(x) \mod f(x)$
1: $c(x) \leftarrow 0$
2: **for** $i = 0$ to $S - 1$ **do**
3: $\quad c(x) \leftarrow c(x) + A_i \cdot b(x)$
4: $\quad M(x) \leftarrow C_0 \cdot F_0'(x) \pmod{x^G}$
5: $\quad c(x) \leftarrow c(x) + M(x) \cdot f(x)$
6: $\quad c(x) \leftarrow c(x)/x^G$
7: **end for**
8: $c(x) \leftarrow c(x) \cdot x^m$
9: $c(x) \leftarrow$ **reduce**$(c(x))$
10: **return** $c(x)$

4.2 Kvadriranje

Najdirektnejši pristop h kvadriranju je množenje elementa samega s sabo. Uporabimo lahko npr. modul za množenje (4.1.1) in ga povežemo tako, da na oba vhoda pripeljemo isti element, izhod pa je njegov kvadrat. Vendar pa obstajajo tudi boljši načini; izkoristimo lahko lastnosti polinomov nad binarnim obsegom, uporabimo metode, ki izvajajo redukcijo sproti, itn.

4.2.1 Klasično kvadriranje

V primeru, ko delamo nad binarmin obsegom, velja:

$$(a+b)^2 = a^2 + b^2$$

Vidimo, da mešanega člena $2ab$ ni, kar pomeni, da lahko kvadrat zapišemo kot:

$$c(x) \equiv a^2(x) \equiv a(x) \cdot a(x) \equiv \left(\sum_{i=0}^{m-1} a_i x^i\right) \cdot \left(\sum_{i=0}^{m-1} a_i x^i\right) \equiv \sum_{i=0}^{m-1} a_i x^{2i} \pmod{f(x)} \tag{4.16}$$

Formula (4.16) narekuje, da koeficiente polinoma $a(x)$ razpršimo, vmesne (lihe) bite pa postavimo na 0:

$$a_{m-1}a_{m-2}\ldots a_1 a_0 \Rightarrow a_{m-1}\ 0\ a_{m-2}\ 0 \ldots\ 0\ a_1\ 0\ a_0$$

Temu koraku nato sledi redukcija po enem izmed načinov, opisanih v poglavju 4.1.1.

4.2.2 Kvadriranje s prepletanjem

Kvadrat lahko izračunamo tudi s prirejenim prepletenim množenjem (glej poglavje 4.1.2). Enačbo (4.16) lahko prepišemo v obliko $c(x) = d(x) \bmod f(x)$, kjer je $d(x) = a(x) \cdot a(x)$, torej nereduciran kvadrat. Zgornja polovica vektorja $d(x)$ predstavlja potence, ki jih moramo reducirati. Pri tem nas zanimajo le sode potence, saj so vsi lihi biti enaki 0. Namesto da bi hranili $m-1$ bitni vektor z vmesnimi ničlami, hranimo kar $\frac{m}{2}$-bitni vektor z s členi $a_{m-1}, a_{m-2}, \ldots a_{\frac{m+1}{2}}$. Spodnje bite vhodnega vektorja d pa že ob inicializaciji priredimo izhodnemu vektorju, tokrat z vmesnimi ničlami: $a_{\frac{m-1}{2}}\ 0\ a_{\frac{m-3}{2}}\ 0 \ldots 0\ a_1\ 0\ a_0$. V vsakem koraku i izračunamo delni rezultat:

$$c(x) \leftarrow c(x) \oplus\ (b(x) \cdot z(i))$$

kjer je $b(x)$ redukcijski vektor za pripadajočo potenco. Začnemo z

$$b(x) = (0 \ldots 0\ 0\ 0\ \overset{n+1}{1}\ 0 \ldots\ 0\ 1\ 0) \text{ za redukcijo elementa } x^{m+1},$$

$$b(x) = (0 \ldots 0\ \overset{n+3}{1}\ 0 \ldots\ 0\ 1\ 0\ 0\ 0) \text{ za redukcijo elementa } x^{m+3},$$

$$\ldots,$$

vmesne potence x^{m+2k} pa preskakujemo, saj je $d_{2k} = 0$. S tem se tudi število korakov, v primerjavi s prepletenim množenjem (m urinih period) , zmanjša za polovico: kvadriranje se izvede v $\frac{m+1}{2}$ urinih periodah.

4.2.3 Kvadriranje Montgomery

Algoritem 3 za Montgomeryevo množenje prilagodimo tako, da izpustimo dejansko množenje v zanki, pred vstopom v zanko pa vhodni vektor razpršimo; glej [18]. Dobimo algoritem 6, ki izračuna $c(x) = a^2(x) \bmod f(x)$.

Algoritem 6 Kvadriranje Montgomery na bitnem nivoju

VHOD: $a(x)$
IZHOD: $c(x) = a^2(x) \mod f(x)$
1: $c(x) \leftarrow \sum_{i=0}^{m-1} a_i x^{2i}$
2: **for** $i = 0$ to $M - 1$ **do**
3: $\quad c(x) \leftarrow c(x) + c_0 \cdot f(x)$
4: $\quad c(x) \leftarrow c(x)/x$
5: **end for**
6: $c(x) \leftarrow c(x) \cdot x^m$
7: $c(x) \leftarrow$ **reduce**$(c(x))$
8: **return** $c(x)$

4.3 Potenciranje

Ogledali si bomo štiri različice potenciranja po metodi "kvadriraj in množi", ki se med seboj razlikujejo le po načinu množenja in načinu kvadriranja. V prvi različici smo uporabili množenje s prepletanjem (4.1.2) in klasično kvadriranje (4.2.1); v drugi pa dva kombinatorična vezja: klasično množenje (4.1.1) in klasično kvadriranje (4.2.1). Preostala dva načina pa oba uporabljata Montgomeryjevo množenje in kvadriranje. Osnovna razlika med njima sta produkt in kvadrat. V prvem primeru računamo Montgomeryjev produkt in kvadrat, v drugem primeru pa le-tega še pomnožimo z $r(x)$ in reduciramo (torej navaden produkt in kvadrat).

Poglejmo, kako deluje metoda "kvadriraj in množi". Naj bo a poljuben element GF(2^m)in e poljubno naravno število. Iščemo $b \in \mathrm{GF}(2^m)$, tako da je $b = a^e$.
Število e lahko zapišemo kot vektor:

$$e = \sum_{i=0}^{k-1} e_i \cdot 2^i = e_{k-1} \cdot 2^{k-1} + \cdots + e_2 \cdot 2^2 + e_1 \cdot 2 + e_0 = (e_{k-1}, \ldots, e_1, e_0), \quad k \in \mathbb{N}, \quad e_i \in \mathbb{Z}_2$$

Sedaj lahko potenciranje zapišemo kot:

$$b = a^e = a^{e_0} \cdot (a^2)^{e_1} \cdot (a^{2^2})^{e_2} \cdot \cdots \cdot (a^{2^{k-1}})^{e_k - 1} = \prod_{i=0}^{k-1} C_i \tag{4.17}$$

kjer je $C_i = (a^{2^i})^{e_i} = \begin{cases} a^{2^i} & , \quad e_i = 1; \\ 1 & , \quad e_i = 0; \end{cases}$

Iz enačbe (4.17) sledi algoritem 7 "kvadriraj in množi".

Algoritem 7 Kvadriraj in množi

VHOD: $a(x) = [a_{m-1}, ..., a_1, a_0], e = (e_{k-1}, ..., e_1, e_0)$
IZHOD: $b(x) = [b_{m-1}, ..., b_1, b_0]$
1: $c(x) \leftarrow a(x)$
2: $b(x) \leftarrow (0 \ldots 01)$
3: **for** $i = 0$ to $k - 1$ **do**
4: **if** $e_i = 1$ **then**
5: $b(x) \leftarrow b(x) \cdot c(x) \mod f(x)$
6: **end if**
7: $c(x) \leftarrow c^2(x) \mod f(x)$
8: **end for**
9: **return** b

Kvadriraj in množi - Montgomery I

Tokrat bomo računali Montgomeryjev produkt $a(x)b(x)r^{-1}(x) \bmod f(x)$ (glej 4.1.4) in Montgomeryjev kvadrat $a^2(x))r^{-1}(x) \bmod f(x)$ (glej 4.2.3). Gre za uporabo prirejenega algoritma 4, brez koraka 7 in 8, ter algoritma 6, brez koraka 6 in 7, torej brez množenja z $r(x)$ in reduciranja. Za razlikovanje Montgomeryjevega množenja od navadnega bomo v prihodnje Montgomeryjev produkt označevali z $\times$ (glej [18]), torej

$$a(x) \times b(x) := a(x)b(x)r^{-1}(x) \bmod f(x)$$

Trditev 4.2 *Naj bo $\widehat{a}(x) := a(x)r(x)$, ti. Montgomeryjeva slika polinoma $a(x)$ pod $r(x)$. Tedaj je $\widehat{a}(x) \times \widehat{b}(x)$ Montgomeryjeva slika produkta $a(x)b(x)$ pod r(x).*

Dokaz.

$$\widehat{a}(x) \times \widehat{b}(x) = \widehat{a}(x)\widehat{b}(x)r^{-1}(x) = (a(x)r(x))(b(x)r(x))r^{-1}(x) = a(x)b(x)r(x)$$

□

Zvezo med polinomi in njihovimi Montgomeryjevimi slikami prikazuje komutativen diagram:

$$\begin{array}{ccc} (a(x), b(b)) & \xrightarrow{\cdot} & a(x) \cdot b(x) \\ {\scriptstyle \cdot r(x)}\downarrow & & \downarrow{\scriptstyle \cdot r(x)} \\ (\widehat{a}(x), \widehat{b}(x)) & \xrightarrow{\times} & \widehat{a}(x) \times \widehat{b}(x) \end{array}$$

Sedaj lahko Algoritem 7 spravimo v naslednjo obliko:

Algoritem 8 Kvadriraj in množi - Montgomery I

VHOD: $a(x) = [a_{m-1}, ..., a_1, a_0], e = (e_{k-1}, ..., e_1, e_0)$
IZHOD: $b(x) = [b_{m-1}, ..., b_1, b_0]$

1: $\widehat{b} \leftarrow 1 \cdot r$
2: $\widehat{a} \leftarrow a \cdot r$
3: **for** $i = 0$ to $k - 1$ **do**
4: **if** $e_i = 1$ **then**
5: $\widehat{b} \leftarrow \widehat{b} \times \widehat{a}$
6: **end if**
7: $\widehat{b} \leftarrow \widehat{b} \times \widehat{b}$
8: **end for**
9: $b \leftarrow \widehat{b} \times 1$
10: **return** b

Spomnimo se, da smo izbrali $r(x) = x^m$, zato v prvem koraku algoritma 8 izračunamo Montgomeryjevo sliko polinoma 1 z $\widehat{b} \leftarrow x^n + 1$, v drugem koraku pa Montgomeryjevo sliko polinoma $a(x)$. Nato v vsakem obhodu zanke izračunamo Montgomeryjev kvadrat $\widehat{b} \times \widehat{b} = \widehat{\widehat{b}\widehat{b}}r^{-1} = b^2 r$ (korak 7) in, če je pripadajoči bit eksponenta enak 1, še Montgomeryjev produkt $\widehat{b} \times \widehat{a} = \widehat{\widehat{b}\widehat{a}}r^{-1} = bar$ (korak 5). Vidimo, da je rezultat po končani zanki Montgomeryjeva slika $\widehat{b} = br$, zato je potreben še korak 9, ki nam vrne iskano potenco b v normalni obliki: $\widehat{b} \times 1 = \widehat{b} \cdot 1 \cdot r^{-1} = brr^{-1} = b$.

Kvadriraj in množi - Montgomery II

Tokrat uporabimo algoritem 7 v osnovni obliki z Montgomeryjevim množenjem (algoritem 4) za računanje produkta $c(x) = a(x)b(x) \bmod f(x)$ (4.1.4) ter Montgomeryjevim kvadriranjem (algoritem 6) za računanje kvadrata $c(x) = a^2(x) \bmod f(x)$ (4.2.3). Videli bomo, da se tako izognemo časovno zahtevnemu začetnemu računanju Montgomeryjeve slike $\widehat{a}(x)$ ter končnemu pretvarjanju Montgomeryjeve slike $\widehat{b}(x)$ v običajno obliko $b(x)$.

4.4 Računanje inverza

V poglavju 5.3, kjer bomo povedali več o implementaciji potenciranja, bomo videli, da lahko inverz izračunamo s potenciranjem, saj je $a^{2^m-2} = a^{-1}$ za vsak $a \in \mathrm{GF}(2^m)$. Postopek si lahko ogledamo na primeru zapisanem v tabeli 5.28. Vendar pa to ni najustreznejši način (glej [15]), zato bomo podali še tri postopke računanja inverza, ki izhajajo iz Evklidovega algoritma.

4.4.1 Razširjen Evklidov algoritem

Za polinom $a(x) \in \mathrm{GF}(2^m)$ iščemo polinoma $p(x), q(x) \in \mathrm{GF}(2^m)$, ki rešita enačbo $a(x)p(x) + f(x)q(x) = 1$. To enačbo lahko prepišemo kot $a(x)p(x) \equiv 1 \pmod{f(x)}$, od koder vidimo, da je

$p(x) \equiv a^{-1}(x) \pmod{f(x)}$ res inverz polinoma $a(x)$.
V nadaljevanju bomo poenostavili zapis in namesto $a(x)$ pisali kar a.
Kot smo že omenili v poglavju 3.1.1, na vsakem koraku Evklidovega algoritma poiščemo polinoma s_i in r_s tako da je $r_{i-2} = s_i r_{i-1} + r_i$, kjer je stopnja polinoma r_{i-1} manjša od stopnje polinoma r_i; glej [15]. Zanima nas le inverz, zato se lahko popolnoma izognemo računanju zaporedja polinomov $\{q_i\}$. Pripadajoče zaporedje polinomov $\{p_i\}$ izračunamo, kot narekuje enačba (3.5), s to razliko, da namesto $-$ sedaj pišemo kar $+$, saj sta v dvojiških obsegih operaciji seštevanja in odštevanja enaki, torej $p_i = p_{i-2} + s_i p_{i-1}$. Veljavni sta tudi enačbi (3.4), v katerih b nadomestimo s f in dobimo začetne vrednosti $r_2 = f,\ p_2 = 0,\ r_1 = a$ in $p_1 = 1$.
Razširjen Evklidov algoritem za a in b, kot je podan v algoritmu 1, sedaj prepišemo za polinome a in f; glej [20, 21].

Algoritem 9 Razširjen Evklidov algoritem za polinome 1

VHOD: polinom a
IZHOD: polinom p (tj. inverz a^{-1})
1: $r_2 \leftarrow f$, $p_2 \leftarrow 0$
2: $r_1 \leftarrow a$, $p_1 \leftarrow 1$
3: **while** $\mathrm{st}(r_1) \neq 0$ **do**
4: $\quad s \leftarrow r_2$ **div** r_1
5: $\quad r \leftarrow r_2 + s r_1$
6: $\quad p \leftarrow p_2 + s p_1$
7: $\quad r_2 \leftarrow r_1, \quad r_1 \leftarrow r$
8: $\quad p_2 \leftarrow p_1, \quad p_1 \leftarrow p$
9: **end while**
10: **return** p_1

Poudarimo še eno (očitno) razliko med algoritmoma 1 in 9. Algoritem 1 se konča, ko dobimo $r = 0$. To pomeni, da smo $D(a, b) \geq 1$ in inverz p izračunali v predzadnjem obhodu zanke in v koraku 15 algoritma 1 vrnemo vrednosti $d \leftarrow b$ in $p \leftarrow p_2$. Drugače je pri polinomih, saj je polinom $f(x)$ nerazcepen, zato za vsak $a(x) \in \mathrm{GF}(2^m)$ velja $D(a(x), f(x)) = 1$. To pa pomeni, da lahko pogoj $r_1 \neq 0$ nadomestimo s pogojem $r_1 \neq 1$ oz. $\mathrm{st}(r_1) \neq 0$ in *while* zanko končamo en obhod prej. V algoritmu 9 izračunamo $D(a, b) = 1 = r_1$ ter inverz p_1 v zadnjem obhodu zanke.

Vendar pa tudi algoritem 9 ni v primerni obliki za implementacijo. Oznaka **div** v koraku 4 pomeni deljenje. Povedali smo že, da računamo inverz prav zato, da deljenje nadomestimo z množenjem z inverzom. Namesto da bi polinom r_2 delili s polinomom r_1, ki je nižje stopnje, bomo z ustrezno potenco elementa x pomnožen r_1 prišteli k r_2. Tako se znebimo najvišje potence oz. bita z največjo težo v števcu. Korake 4, 5 in 6 algoritma 9 nadomestimo z zaporedjem izačunov $d \leftarrow st(r_2) - st(r_1)$, $r_2 \leftarrow r_2 \oplus x^d r_1$ in $p_2 \leftarrow p_2 \oplus x^d p_1$ (koraki 4, 8 in 9 algoritma

10). Množenje polinoma z x^d je enostavno in hitro, saj ga realiziramo z levim pomikom registra r_1 za d mest. Ta postopek ponavljamo, dokler je stopnja deljenca večja ali enaka stopnji delitelja; glej [20]. Ko je $\mathrm{st}(r_2) < \mathrm{st}(r_1)$, zamenjamo vrednosti registrov ($r_2 \leftrightarrow r_1,\ p_2 \leftrightarrow p_1$) ter nadaljujemo z naslednjim obhodom zanke. Tako modificiran algoritem zapišemo na naslednji način:

Algoritem 10 Razširjen Evklidov algoritem za polinome 2

VHOD: polinom a
IZHOD: inverz $p = a^{-1}$
1: $r_2 \leftarrow f$, $p_2 \leftarrow 0$
2: $r_1 \leftarrow a$, $p_1 \leftarrow 1$
3: **while** $st(r_1) \neq 0$ **do**
4: $d \leftarrow st(r_2) - st(r_1)$
5: **if** $d < 0$ **then**
6: $r_2 \leftrightarrow r_1, \quad p_2 \leftrightarrow p_1, \quad d \leftarrow -d$
7: **end if**
8: $r_2 \leftarrow r_2 \oplus x^d r_1$
9: $p_2 \leftarrow p_2 \oplus x^d r_1$
10: **end while**
11: **return** p_1

4.4.2 Berlekampov algoritem

Berlekampov algoritem je realizacija razširjenega Evklidovega algoritma, ki bazira na zvezi med enačbama $r \leftarrow r_2 \oplus sr_1$ ter $p \leftarrow p_2 \oplus sr_1$. Če namreč r_1 množimo z x, moramo enako storiti tudi s p_1, kar pomeni, da zamikamo pare polinomov. Namesto štirih m-bitnih registrov $r_2, \quad r_1, \quad p_2$ in p_1 potrebujemo le dva $(m+2)$-bitna registra, v katera paroma združimo po dva polinoma in ju ločimo z (navidezno) vejico. Povedali smo že, da bite registra r_i uničujemo z leve proti desni, tj. od MSB proti LSB. Ko se rešimo MSB, sledi levi pomik registra, pri čemer postane LSB enak '0'. Ker te ničle niso pomembne za naslednje korake, lahko v ta del registra shranjujemo bite polinoma p_j, oba dela pa ločimo z vejico. Povejmo še, da bite polinoma p_j pišemo v obratnem vrstnem redu, kot lahko vidimo na primeru za $m = 5$: $(r_3 \quad r_2 \quad r_1 \quad r_0, \quad p_0 \quad p_1 \quad p_2)$. Tako vsakič, ko pomaknemo npr. zgornji register za eno mesto v levo, izvedemo operacijo $r_2 \leftarrow xr_2$ na delu registra levo od vejice, ter operacijo $p_1 \leftarrow p_1/x$ na desnem delu registra, saj je polinom p_1 shranjen v obratnem vrstnem redu. Le-to pa ne povzroča nobenih težav, saj sta operaciji $p_1 \leftarrow p_1/x$ in $p_2 \leftarrow xp_2$ enakovredni. Na začetku postavimo v levi del zgornjega registra bite $r_2 = f$, v desni del za vejico pa $p_1 = 1$. Skrajni levi bit spodnjega registra postavimo na 0, nato sledijo biti $r_1 = a$, v desni del oz. bit spodnjega registra pa zapišemo $p_2 = 0$.

Primer: Oglejmo si delovanje Berlekampovega algoritma na primeru $\mathrm{GF}(2^5)$ z nerazcepnim polinomom $r_2 = f(x) = x^5 + x^2 + 1$ in poiščimo inverz elementa $a(x) = x^4 + x + 1$.

$$
\begin{array}{lcl}
\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1, & 1 \\ 0 & 1 & 0 & 0 & 1 & 1, & 0 \end{pmatrix} & = & \begin{pmatrix} r_2 & , & p_1 \\ r_1 & , & p_2 \end{pmatrix} \\
\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1, & 1 \\ 1 & 0 & 0 & 1 & 1, & 0 & 0 \end{pmatrix} & = & \begin{pmatrix} r_2 & , & xp_1 \\ xr_1 & , & p_2 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1, & 1 \\ 1 & 0 & 0 & 1 & 1, & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} r_2 + xr_1 & , & xp_1 \\ xr_1 & , & p_2 + xp_1 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 & 0 & 1 & 1, & 1 & 0 \\ 1 & 0 & 0 & 1 & 1, & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} r_0 & , & p_1 \\ r_1 & , & p_0 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 & 1 & 1, & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1, & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} xr_0 & , & p_1 \\ r_1 & , & xp_0 \end{pmatrix} \\
\begin{pmatrix} 0 & 1 & 1, & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1, & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} x^2r_0 & , & p_1 \\ r_1 & , & x^2p_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1, & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} x^3r_0 & , & p_1 \\ r_1 & , & x^3p_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1, & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} x^3r_0 & , & p_1 + x^3p_0 \\ r_1 + x^3r_0 & , & x^3p_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1, & 0 & 1 & 0 \end{pmatrix} & = & \begin{pmatrix} x^2r_0 & , & p_1 + x^3p_0 \\ r_1 + x^3r_0 & , & x^2p_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1, & 0 & 1 & 0 \end{pmatrix} & = & \begin{pmatrix} xr_0 & , & p_1 + (x^3 + x^2)p_0 \\ r_1 + (x^3 + x^2)r_0 & , & xp_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1, & 0 & 1 & 0 & 0 \end{pmatrix} & = & \begin{pmatrix} xr_0 & , & p_1 + (x^3 + x^2)p_0 \\ r_1 + (x^3 + x^2)r_0 & , & xp_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1, & 0 & 1 & 0 & 0 \end{pmatrix} & = & \begin{pmatrix} xr_0 & , & p_1 + (x^3 + x^2 + x)p_0 \\ r_1 + (x^3 + x^2 + x)r_0 & , & xp_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 1 & 1 & 1 \\ 0 & 1, & 0 & 1 & 0 & 0 & 0 \end{pmatrix} & = & \begin{pmatrix} r_0 & , & p_1 \\ r_1 & , & p_0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1, & 1 & 0 & 1 & 1 & 1 \\ 1, & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} & = & \begin{pmatrix} r_0 & , & xp_1 \\ xr_1 & , & p_0 \end{pmatrix} \\
\begin{pmatrix} 0 & 1, & 1 & 0 & 1 & 1 & 1 \\ 1, & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} & = & \begin{pmatrix} r_0 + xr_1 & , & xp_1 \\ xr_1 & , & p_0 + xp_1 \end{pmatrix} \\
\begin{pmatrix} 1, & 1 & 0 & 1 & 1 & 1 & 0 \\ 1, & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} & = & \begin{pmatrix} r_0 + xr_1 & , & p_1 \\ r_1 & , & p_0 + xp_1 \end{pmatrix} \\
\begin{pmatrix} 0, & 1 & 0 & 1 & 1 & 1 & 0 \\ 1, & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} r_0 + (x+1)r_1 & , & p_1 \\ r_1 & , & p_0 + (x+1)p_1 \end{pmatrix} \\
\begin{pmatrix} ,1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1, & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} xr_2 & , & p_1 \\ r_1 & , & xp_2 \end{pmatrix}
\end{array}
$$

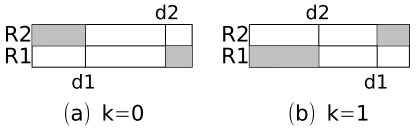
Algoritem se konča, ko se prva vejica premakne "izven"registra, inverz pa je prav v tem registru; glej [22]. Ta register pomaknemo $(m+1)$-krat, drug register pa m-krat in tako je skupno število pomikov $2m+1$. Vsakemu seštevanju sledi pomik, zato je lahko kvečjemu $2m$ seštevanj, in algoritem se zagotovo izteče v $4m+1$ korakih. V našem primeru je bil inverz v zgornjem registru:

$$\begin{pmatrix} ,p_0 & p_1 & p_2 & p_3 & p_4 & 0 & 0 \\ ,1 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Sedaj ga odčitamo: $a^{-1}(x) = x^4 + x^3 + x^2 + 1$.

Zgornji primer lepo prikaže delovanje Berlekampovega algoritma. Vidimo, da vedno spreminjamo levi del tistega registra, ki ima vejico bolj na desni. Naj bo d_2 oz. d_1 indeks, na katerem se nahaja vejica v zgornjem registru r_2 oz. v spodnjem registru r_1. Uvedimo parameter k, ki spremlja kateri register spreminjamo. Če je $d_1 > d_2$, je $k = 0$, in spreminjamo levi del zgornjega

registra R2 ter desni del spodnjega registra R1. To situacijo prikazuje primer (a) na sliki 4.2, spreminjamo pa osenčen del registrov.



Slika 4.2: Spreminjanje vsebine registrov - pišemo v osenčen del registra: (a) pri $d_1 > d_2$ in (b) pri $d_1 < d_2$, črta v registru predstavlja vejico oz. pozicijo, na katero kaže d_1 oz. d_2

Opišimo delovanje Berlekampovega algoritma (glej [22]):

- če je MSB registra r_2 enak 0 ($r_2(m+1) = 0$), pomakni register r_2, vključno z vejico (povečaj tudi indeks d_2), eno mesto v levo;
- če je MSB registra r_1 enak 0 ($r_1(m+1) = 0$), pomakni register r_1, vključno z vejico (povečaj tudi indeks d_1), eno mesto v levo;
- če sta MSB obeh registrov enaka 1 ($r_2(m+1) = r_1(m+1) = 1$) in je $k = 0$, prištej levi del (od MSB do vejice d_1) spodnjega registra r_1 v levi del zgornjega registra r_2 in desni del zgornjega registra r_2 (od vejice d_2 do LSB) v desni del spodnjega registra r_1, kot je prikazano na sliki 4.2 (a);
- če sta MSB obeh registrov enaka 1 ($r_2(m+1) = r_1(m+1) = 1$) in je $k = 1$, prištej levi del (od MSB do vejice d_2) zgornjega registra r_2 v levi del spodnjega registra r_1 in desni del spodnjega registra r_1 (od vejice d_1 do LSB) v desni del zgornjega registra r_2, kot je prikazano na sliki 4.2(b).

4.4.3 Modificiran skoraj inverzni algoritem

Pri postopkih, opisanih v predhodnih dveh poglavjih, računamo inverz "od leve proti desni", tj. od MSB proti LSB. Mogoče pa je tudi računanje v obratni smeri. Namesto da delitelj množimo, lahko deljenec delimo z ustrezno potenco elementa x. Seveda tudi tokrat delimo postopoma. Deljenje z x je prav tako enostavna operacija, ki jo izvedemo z desnim pomikom registra za en bit. Kadarkoli delimo r_1, moramo deliti tudi njemu pripadajoči p_1. Deljivost polinoma r_1 z elementom x preverjamo v pogoju *while* zanke, nimamo pa nobenega zagotovila, da bo z x deljiv tudi p_1. Prva rešitev je zamenjava operacije $p_1 \leftarrow p_1/x$ z operacijo $p_2 \leftarrow xp_2$, vendar pa to pripelje do končnega rezultata $x^{-k}p$, kjer je k skupno število deljenj registrov r_1 in r_2. Da dobimo inverz, moramo ta rezultat še množiti z x^k in reducirati. Takšen algoritem je znan pod imenom "skoraj inverzni algoritem" (*Almost Inverse Algorithm*, AIA). Druga možnost, znana

pod imenom "modificiran skoraj inverzni algoritem" ($MAIA$), glej [21, 18], pa ne nadomesti deljenja z množenjem, ampak v primeru, ko velja $p_1(0) = 1$, in torej p_1 ni deljiv z x, polinomu p_1 prišteje nerazcepni polinom $f(x)$, nato pa deli njuno vsoto (koraki 6 do 10 in 14 do 18 v algoritmu 11). Podobno kot pri Berlekampovem algoritmu, tudi tukaj vedno spreminjamo tisti register r_i, ki vsebuje polinom višje stopnje. Algoritem se konča, ko postane eden izmed registrov $r_1,\ r_2$ enak 1, iskani inverz pa je njemu pripadajoči register p_i (koraki 25 do 29 v algoritmu 11).

Algoritem 11 MAIA - Modificiran skoraj inverzni algoritem 1

VHOD: polinom a
IZHOD: inverz $p = a^{-1}$
1: $r_2 \leftarrow f$, $p_2 \leftarrow 0$
2: $r_1 \leftarrow a$, $p_1 \leftarrow 1$
3: **while** $r_2 \neq 0$ **and** $r_1 \neq 0$ **do**
4: **while** x **deli** r_1 **do**
5: $r_1 \leftarrow r_1/x$
6: **if** x **deli** p_1 **then**
7: $p_1 \leftarrow p_1/x$
8: **else**
9: $p_1 \leftarrow (p_1 + f)/x$
10: **end if**
11: **end while**
12: **while** x **deli** r_2 **do**
13: $r_2 \leftarrow r_2/x$
14: **if** x **deli** p_2 **then**
15: $p_2 \leftarrow p_2/x$
16: **else**
17: $p_2 \leftarrow (p_2 + f)/x$
18: **end if**
19: **end while**
20: **if** $st(r_1) > st(r_2)$ **then**
21: $r_1 \leftarrow r_1 \oplus r_2, \quad p_1 \leftarrow p_1 \oplus p_2$
22: **else**
23: $r_2 \leftarrow r_1 \oplus r_2, \quad p_2 \leftarrow p_1 \oplus p_2$
24: **end if**
25: **if** $r_1 = 1$ **then**
26: $p \leftarrow p_1$
27: **else**
28: $p \leftarrow p_2$
29: **end if**
30: **end while**
31: **return** p

4.5 Eliptične krivulje nad GF(2^m)

Definiciji seštevanja in podvojevanja točk iz $\mathbb{F}^3$ na eliptični krivulji smo spoznali v razdelku 3.2.4. Formule 3.8 in 3.9 za izračun nove točke sicer lahko implementiramo, vendar je ta pristop

zamuden, saj računanje smernega koeficienta λ vključuje deljenje. Algoritmi se zelo poenostavijo s prehodom na projektivni prostor, kjer vsako točko opišemo s tremi koordinatami; glej [21, 23]. Iz $\mathbb{F}^3$ naredimo projektivni prostor na naslednji način: za urejeni trojici (X_1, Y_1, Z_1) in (X_2, Y_2, Z_2) iz $\mathbb{F}^3 \setminus (0,0,0)$ definiramo ekvivalenčno relacijo $\sim$ z:

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \iff \exists \mu \in \mathbb{F}^* \text{ tak, da } \quad X_1 = \mu^c X_2, \quad Y_1 = \mu^d Y_2, \quad Z_1 = \mu Z_2$$

kjer sta c in d naravni števili. Tako dobimo ekvivalenčne razrede, ki predstavljajo točke v projektivnem prostoru:

$$(X : Y : Z) = \{(\mu^c X, \mu^d Y, \mu Z); \mu \in \mathbb{F}^*\},$$

trojica (X, Y, Z) pa je predstavnik posameznega razreda. Če je $Z = 0$, govorimo v ravnimi o točki v neskončnosti, vse trojice, ki imajo $Z = 0$, pa sestavljajo premico v neskončnosti. Ko je $Z \neq 0$, pa lahko projektivno točko $(X : Y : Z)$ predstavimo s trojico $(X/Z^c, Y/Z^d, 1)$. Le-ta je edini predstavnik tega ekvivalenčnega razreda, ki ima koordinato Z enako ena. Tedaj lahko točko $(X : Y : Z)$ zapišemo z dvema koordinatama $(X/Z^c, Y/Z^d)$, kot smo bili navajeni v razdelku 3.2.4. Sedaj lahko v krivuljo 3.7 vstavimo $x = \frac{X}{Z^c}$ in $y = \frac{Y}{Z^d}$. Ko odpravimo ulomke, enačba krivulje preide v naslednjo obliko:

$$Y^2 Z^{3c} + XY Z^{2c+d} = X^3 Z^{2d} + aX^2 Z^{c+2d} + bZ^{3c+2d} \tag{4.18}$$

Projektivne koordinate nam omogočijo, da izvajamo operacije brez vmesnega deljenja. Ko so vse operacije končane, naredimo prehod v običajne koordinate, tj. na $\mathbb{F}^2$, ki zahteva le dve deljenji, in s tem le dva računsko zahtevna inverza. Razume se, in ne bomo dokazovali, da v primeru, kadar najprej izvedemo seštevanje ali podvojevanje v običajnih koordinatah, in nato preidemo v projektivni prostor, dobimo enako projektivno točko, kot če najprej preidemo iz običajnih v projektivne koordinate ter šele nato izvedemo seštevanje ali podvojevanje. V nadaljevanju bomo podali eliptično krivuljo 3.7 ter formule za seštevanje in podvojevanje v standardnih projektivnih koordinatah in v projektivnih koordinatah Lòpez-Dahab.

4.5.1 Standardne projektivne koordinate

Krivulja 4.18 v standardnih projektivnih koordinatah, ko je $c = d = 1$, zavzame obliko:

$$Y^2 Z + XYZ = X^3 + aX^2 Z + bZ^3 \tag{4.19}$$

kjer točka $(X : Y : Z)$ predstavlja točko $(X/Z, Y/Z)$, pri čemer mora veljati $Z \neq 0$. To pomeni tudi, da bomo pri prehodu na običajne koordinate potrebovali dve deljenji, ki ju lahko izvedemo z enim računanjem inverza in dvema množenjema. Če je koordinata $Z = 0$, govorimo

o točki v neskončnosti (∞, ∞), ki jo v standardnih projektivnih koordinatah predstavlja točka $P_\infty = (0 : 1 : 0)$. Nasprotna točka točke $(X : Y : Z)$ je $(X : X + Y : Z)$.
Vsoto $R = (X_3 : Y_3 : Z_3) = P + Q$ dveh različnih točk $P = (X_1 : Y_1 : Z_1)$ in $Q = (X_2 : Y_2 : Z_2)$, pri čemer velja $P \neq \pm Q$, izračunamo takole; glej [21, 23]:

$$\begin{gathered} A = Y_1 Z_2 + Z_1 Y_2 \qquad B = X_1 Z_2 + Z_1 X_2 \\ C = B^2 \qquad D = Z_1 Z_2 \\ E = (A^2 + AB + aC)D + BC \\ X_3 = BE \qquad Z_3 = B^3 D \\ Y_3 = C(AX_1 + Y_1 B)Z_2 + (A + B)E \end{gathered} \tag{4.20}$$

Kadar je $P = -Q$, je vsota točka v neskončnosti, kadar je $P = Q$, pa uporabimo formule za podvojevanje točke. Koordinate podvojene točke $R = 2P = (X_3 : Y_3 : Z_3)$, kjer je $P = (X_1 : Y_1 : Z_1)$, dobimo takole:

$$\begin{gathered} A = X_1^2 \qquad B = A + Y_1 Z_1 \qquad C = X_1 Z_1 \qquad D = C^2 \\ E = B^2 + BC + aD \\ X_3 = CE \qquad Y_3 = (B + C)E + A^2 C \qquad Z_3 = CD \end{gathered} \tag{4.21}$$

4.5.2 Projektivne koordinate Lòpez-Dahab

Parametra c in d v enačbi 4.18 tokrat zavzameta vrednosti $c = 1$ in $d = 2$, urejena enačba eliptične krivulje v Lòpez-Dahab koordinatah pa izgleda takole; glej [24]:

$$Y^2 + XYZ = X^3 Z + aX^2 Z^2 + bZ^4 \tag{4.22}$$

Če je $Z \neq 0$, projektivna točka $(X : Y : Z)$ predstavlja točko $(X/Z, Y/Z^2)$ v $\mathbb{F}^2$. Za prehod na običajne koordinate sta potrebna dva inverza ter računanje Z^2. Če je $Z = 0$, pa točka $(X : Y : Z)$ predstavlja točko v neskončnosti, ki jo v projektivnih koordinatah zapišemo $P_\infty = (1 : 0 : 0)$. Nasprotna točka točke $(X : Y : Z)$ je $(X : XZ + Y : Z)$.
Vsota $R = (X_3 : Y_3 : Z_3) = P + Q$ točk $P = (X_1 : Y_1 : Z_1)$ in $Q = (X_2 : Y_2 : Z_2)$, $P \neq \pm Q$:

$$\begin{gathered} A_1 = Y_2 Z_1^2 \quad A_2 = Y_1 Z_2^2 \quad B_1 = X_2 Z_1 \quad B_2 = X_1 Z_2 \\ E = Z_1 Z_2 \qquad F = (B_1 + B_2)E \qquad Z_3 = F^2 \\ G = (A_1 + A_2)F \qquad H = (B_1 + B_2)^2 (F + aE) \\ X_3 = (A_1 + A_2)^2 + G + H \\ I = (B_1 + B_2)^2 B_1 E + X_3 \quad J = (B_1 + B_2)^2 A_1 + X_3 \\ Y_3 = IG + Z_3 J \end{gathered} \tag{4.23}$$

V primeru $P = -Q$ je vsota točka v neskončnosti, v primeru $P = Q$ pa moramo uporabiti formule za podvojevanje točke.

Poglejmo še, kako dobimo koordinate podvojene točke $R = 2P = (X_3 : Y_3 : Z_3)$, kjer je $P = (X_1 : Y_1 : Z_1)$:

$$\begin{array}{cccc} A = Z_1^2 & B = bA^2 & C = X_1^2 & Z_3 = AC \\ X_3 = C^2 + B & & Y_3 = (Y_1^2 + aZ_3 + B)X_3 + Z_3B & \end{array} \tag{4.24}$$

4.5.3 Množenje točk na eliptični krivulji

Poglejmo sedaj, kako z uporabo seštevanj in podvojevanj realiziramo množenje točke s skalarjem; glej [21]. Skalar $k \in \mathbb{N}$ predstavimo kot vektor $k = (k_{m-1}, \ldots, k_1, k_0)$.

Algoritem 12 Množenje točke s skalarjem

VHOD: skalar $k = (k_{m-1}, \ldots, k_1, k_0)$, točka $P = (x, z, y) \in E$
IZHOD: $Q = kP$
1: $Q \leftarrow P$
2: **for** $i = m - 1$ **to** 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** Q

V vsakem obhodu algoritma 12 izvedemo eno podvajanje točke, če je trenutni bit skalarja enak ena, pa še seštevanje.

Poglavje 5

Implementacija in rezultati

V tem razdelku si bomo ogledali implementacijo algoritmov, opisanih v prejšnjem poglavju. Opozorili bomo na modifikacije, ki so potrebne za implementacijo teh algoritmov na FPGA, ter na nekatere optimizacije, ki jih naredi `Xilinx-ISE` sam. Posamezne algoritme bomo primerjali na podlagi njihove časovne in prostorske zahtevnosti. Pri tem moramo poudariti razliko med kombinatoričnimi in sekvenčnimi vezji. Kot smo že omenili v drugem poglavju, opazujemo v primeru kombinatoričnih vezij število porabljenih LUT-ov in rezin, ter zakasnitev signala skozi modul. Pri sekvenčnih vezjih pa nas zanima tudi število porabljenih pomnilnih celic (flip-flop, v tabelah navedeno v stolpcu FF). V tem primeru podamo časovno zahtevnost z minimalno urino periodo ter skupnim časom, ki ga izračunamo kot produkt urine periode in števila ciklov. Na podlagi teh rezultatov bomo izbrali najbolj ugodne module za posamezno operacijo, ki jih bomo kasneje povezovali v večje module.
Rezultati za posamezne module so zbrani v tabelah za različne vrednosti parametra m (pripadajoči nerazcepni polinomi so zbrani v tabeli 3.2). Vsi časi (zakasnitev, dolžina cikla oz. urine periode, v tabelah navedena v stolpcu u.p., in skupen čas) so v tabelah navedeni v nanosekundah $[ns]$ ali mikrosekundah $[\mu s]$ (pri implementaciji eliptičnih krivulj).

5.1 Množenje

5.1.1 Redukcija

Redukcija s štirimi vektorji

Za boljšo predstavo o tem, kaj se dejansko zgodi znotraj FPGA, si podrobneje oglejmo redukcijo za nerazcepni polinom $f(x) = x^{11} + x^2 + 1$. Pri množenju dveh 11-bitnih vektorjev dobimo 21-bitni produkt $d(x)$. Tabela 5.5 prikazuje tabelo 4.1, prepisano za 21-bitni $d(x)$.
Za 11-bitni izhodni vektor $c(x)$ potrebujemo 11 LUT-ov, to je en LUT za vsak bit izhodnega

$w\rightarrow$	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
$y\rightarrow$	d_{19}	d_{18}	d_{17}	d_{16}	d_{15}	d_{14}	d_{13}	d_{12}	d_{11}	d_{12}	d_{11}
$x\rightarrow$	0	d_{20}	d_{19}	d_{18}	d_{17}	d_{16}	d_{15}	d_{14}	d_{13}	0	0
$z\rightarrow$	0	0	0	0	0	0	0	0	d_{20}	0	d_{20}
$c\rightarrow$	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	d_0

Tabela 5.1: Redukcija s štirimi vektorji za $m = 11$

vektorja. Shemo modula *reducer* za redukcijo s štirimi vektorji vidimo na sliki 5.1. Izhodni bit c_2 potrebuje 4-vhodni LUT, bita c_1 in c_{10} 2-vhodni LUT, ostali biti pa 3-vhodne LUT-e. Tabele 5.2, 5.3 in 5.4 prikazujejo logične funkcije, ki so realizirane s posameznimi 2-, 3- in 4-vhodnimi LUT-i.

$\mathbf{I_1}$	$\mathbf{I_0}$	**O**
0	0	0
0	1	1
1	0	1
1	1	0

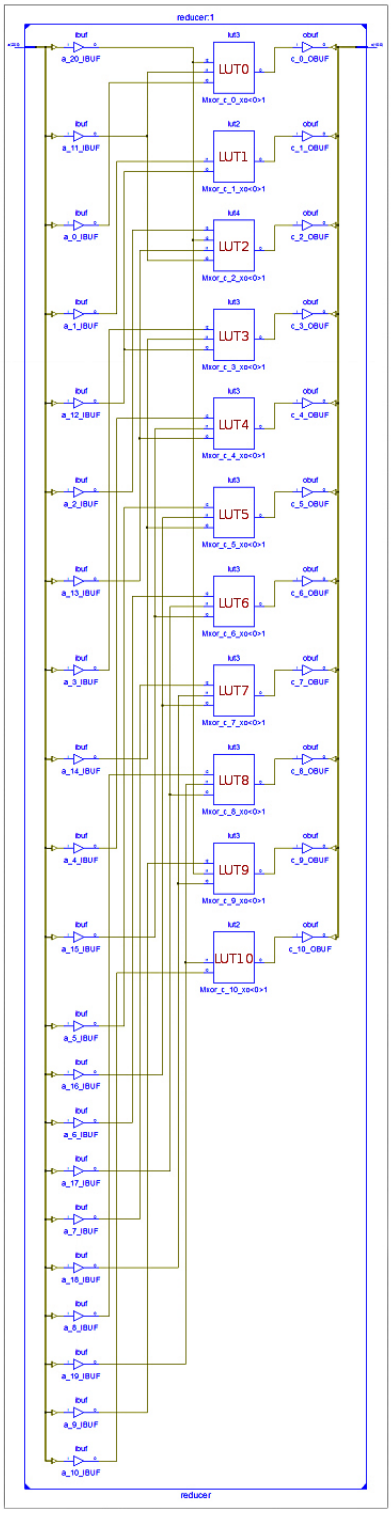
Tabela 5.2: Pravilnostna tabela za preklopno funkcijo, realizirano z 2-vhodnim LUT-om

$\mathbf{I_2}$	$\mathbf{I_1}$	$\mathbf{I_0}$	**O**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabela 5.3: Pravilnostna tabela za preklopno funkcijo, realizirano s 3-vhodnim LUT-om

$\mathbf{I_3}$	$\mathbf{I_2}$	$\mathbf{I_1}$	$\mathbf{I_0}$	**O**
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Tabela 5.4: Pravilnostna tabela za preklopno funkcijo realizirano s 4-vhodnim LUT-om



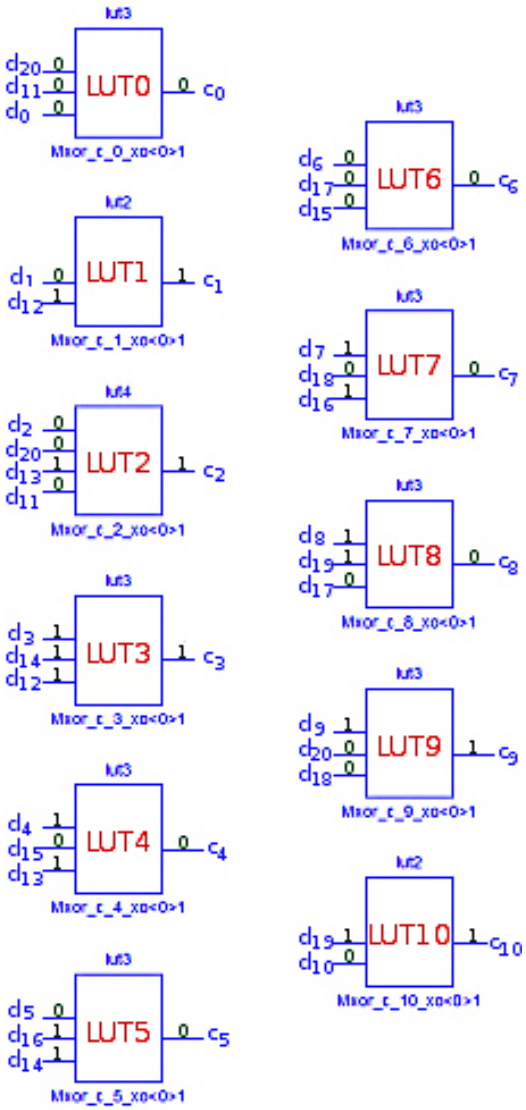
Slika 5.1: Shema vezja za redukcijo s štirimi vektorji za 11-bitno verzijo

Primer: Oglejmo si, kaj se zgodi pri redukciji polinoma $d(x) = x^{19} + x^{16} + x^{14} + x^{13} + x^{12} + x^9 + x^8 + x^7 + x^4 + x^3$. Stanje na vhodih in izhodih LUT-ov vidimo na shemi 5.2. Dobljen reduciran vektor je $x^{10} + x^9 + x^3 + x^2 + x$. Ta rezultat preverimo s tabelo 5.5 tako, da vpišemo vrednosti bitov d_i, zadnjo vrstico tabele pa dobimo z XOR-om prvih štirih:

$w\rightarrow$	0	1	1	1	0	0	1	1	0	0	0
$y\rightarrow$	1	0	0	1	0	1	1	1	0	1	0
$x\rightarrow$	0	0	1	0	0	1	0	1	1	0	0
$z\rightarrow$	0	0	0	0	0	0	0	0	0	0	0
$c\rightarrow$	1	1	0	0	0	0	0	1	1	1	0

Tabela 5.5: Redukcija s štirimi vektorji za 11-bitni vektor $d(x)$

Podrobneje si oglejmo 3-vhodni LUT9 na shemi 5.2. Na vhodne pripeljemo signale d_9, d_{20} in d_{18}, kar sovpada z izhodu c_9 pripadajočim stolpcem tabele 5.5. Vhodna kombinacija (1,0,0) da na izhod LUT-a vrednost 1 (peta vrstica v tabeli 5.3).
Podobno dobimo na 4-vhodnem LUT2 pri vhodni kombinaciji (0,0,1,0) izhod 1 (tretja vrstica v tabeli 5.4).



Slika 5.2: Shema vezja za redukcijo 11-bitnega polinoma $d(x)$ z vhodnimi in izhodnimi signali LUT-ov

Redukcija z redukcijsko matriko

S prirejenim algoritmom (5.1) za računanje elementov redukcijske matrike R lahko zmanjšamo število XOR ter AND vrat, ki bi jih potrebovali pri implementaciji direktno po formuli 4.6. Tako iz zapisa 5.2 razberemo, da je za redukcijo polinoma potrebnih $m(m-1)$ XOR vrat in $m(m-1)$ AND vrat. Za redukcijo s štirimi vektorji (Algoritem 4.1.1) pa potrebujemo $3(m-1)$ XOR vrat. Pričakovali bi, da bo redukcija s štirimi vektorji ugodnejša, vendar pa rezultati (tabeli 5.6 in 5.7) kažejo, da sta redukcijska modula enakovredna, saj je njuna prostorska in časovna zahtevnost skoraj enaka.

To se zgodi, ker je redukcijska matrika R (4.5) ob izbranem končnem obsegu in pripadajočem nerazcepnem vektorju $f(x)$ fiksna - `XilinxISE` to zazna, in upošteva pri implementaciji. Kako, si oglejmo na primeru z nerazcepnim polinomom $f(x) = x^{11} + x^2 + 1$.

VHDL koda 5.1: Računanje elementov redukcijske matrike R za trinome

```
1   for j in 0 to M-1 loop
2      for i in 0 to M-2 loop
3         R(j)(i) := '0';
4      end loop;
5   end loop;
6   for j in 0 to M-2 loop
7     R(j)(j):='1';
8       if( j < M - N) then
9         R(j + N)(j):='1';
10      else
11        R(j - M + N)(j):='1';
12        R(j - M + 2*N)(j):='1';
13      end if;
14    end loop;
```

VHDL koda 5.2: Redukcija vektorja d dolžine $2m-2$ z redukcijsko matriko MR in rezultatom r

```
1  for j in 0 to M-1 generate
2      process(d)
3         variable aux: std_logic :='0';
4         begin
5            aux:=d(j);
6            for i in 0 to M-2 loop
7                 aux := aux xor (d(M+i) and MR(j)(i));
8            end loop;
9            r(j) <= aux;
10     end process;
11  end generate;
```

Enačbo $\boldsymbol{c}^T = E\boldsymbol{d}^T$ prepišimo za $f(x) = x^{11} + x^2 + 1$ in 21-bitni vektor $d(x)$.

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\begin{pmatrix} d_0 \\ d_1 \\ \\ \vdots \\ \\ d_{19} \\ d_{20} \end{pmatrix} =$$

$$\begin{bmatrix}
d_0 + d_{11} + d_{20} \\
d_1 + d_{12} \\
d_2 + d_{11} + d_{13} + d_{20} \\
d_3 + d_{12} + d_{14} \\
d_4 + d_{13} + d_{15} \\
d_5 + d_{14} + d_{16} \\
d_6 + d_{15} + d_{17} \\
d_7 + d_{16} + d_{18} \\
d_8 + d_{17} + d_{19} \\
d_9 + d_{18} + d_{20} \\
d_{10} + d_{19}
\end{bmatrix}$$

Orodje `Xilinx-ISE` implementira vezje za izračun elementov izhodnega vektorja $\boldsymbol{c}^T$, kot vidimo v rezultatu zgornjega izračuna (le-ta pa sovpada s stolpci tabele 5.5). Shema vezja, s katerim je realizirana redukcija z redukcijsko matriko, je tako skoraj identična shemi vezja za redukcijo s štirimi vektorji (slika 5.1). Ponovno se srečamo z 11 LUT-i, od tega sta dva 2-vhodna, en 4-vhoden in preostali 3-vhodni, od LUT-ov v shemi 5.1 pa se razlikujejo le po vrstnem redu vhodov v posamezni LUT.

Rezultati implementacije

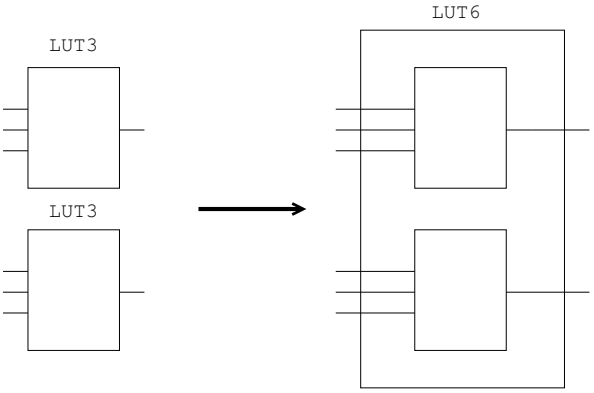
m	LUT-i	Rezine	Zakasnitev[ns]
11	6	5	5.407
63	32	31	9.320
113	60	50	11.029
191	103	87	11.497

Tabela 5.6: Redukcija s štirimi vektorji - rezultati impl ementacije

m	LUT-i	Rezine	Zakasnitev[ns]
11	6	5	5.386
63	32	32	9.008
113	59	48	11.781
191	103	93	11.181

Tabela 5.7: Redukcija z redukcijsko matriko - rezultati implementacije

Na shemi 5.1 vidimo 11 LUT-ov, rezultati za $m = 11$ (tabeli 5.6 in 5.7) pa kažeta 6 LUT-ov. Kot smo že povedali v poglavju 2, Virtex-6 vsebuje 6-vhodne LUT-e z dvema izhodoma. To pomeni, da lahko npr. dva 3-vhodna LUT-a "zložimo" v en 6-vhodni LUT (shema 5.3). V



Slika 5.3: Dva 3-vhodna LUT-a, realizirana s 6-vhodnim LUT-om

primeru modulov za redukcijo je teh 11 LUT-ov skritih v petih LUT-ih, ki uporabljajo oba izhoda $\mathbf{O_6}$ in $\mathbf{O_5}$, in v enem LUT-u, ki uporablja samo izhod $\mathbf{O_6}$. Seveda se s tem spremenijo tudi posamezne pravilnostne tabele.

Povedali smo že, da je generirano vezje v obeh primerih skoraj popolnoma enako. Posledično so tudi rezultati implementacije obeh modulov zelo podobni (tabeli 5.6 in 5.7). Redukcija z redukcijsko matriko se izkaže kot ugodnejša z vidika časovne zahtevnosti, zato jo bomo uporabljali tudi v prihodnje.

5.1.2 Klasično množenje

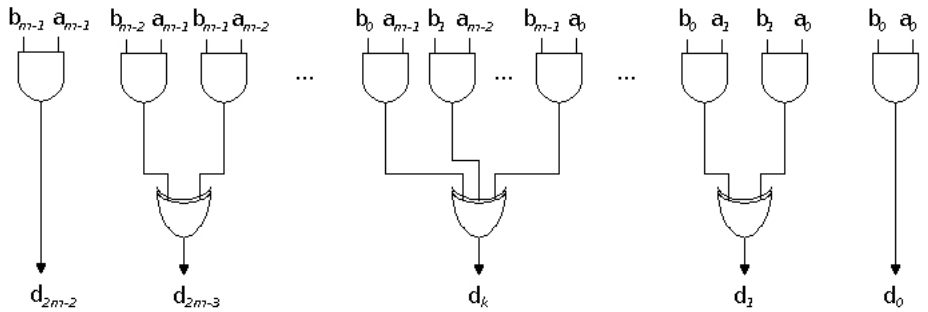
Kot smo že povedali, je množenje sestevljeno iz dveh korakov, dejanskega množenja in pa redukcije. Klasično množenje po enačbi (4.1), katere implementacijo vidimo v 5.3, izvaja modul *ClassicMul*. VHDL opis računanja koefiecientov produkta lahko vidimo v 5.3. Shematski prikaz računanja koeficientov produkta vidimo na sliki 5.4.

Modul *Cmultiplier* torej povezuje modul *ClassicMul* in modul *reduceB* (slika 5.5), ki izvaja redukcijo z redukcijsko matriko, oz. modul *ClassicMul* in modul *reducer*, ki izvaja redukcijo s štirimi

vektorji. V tabeli 5.8 so zbrani rezultati za prvo možnost. Kot smo opazili že pri redukciji, z večanjem parametra m narašča ne le število elementov, temveč tudi zakasnitev.

VHDL koda 5.3: Računanje koeficientov produkta $d(x)$

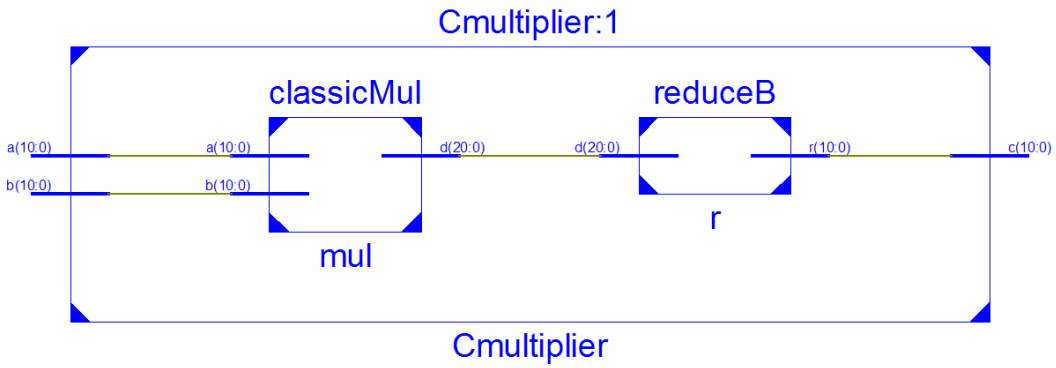
```
2   gen_xors1: for k in 0 to M-1 generate
3     l1: process(a,b)
4       variable aux: std_logic:='0';
5         begin
6           aux:='0';
7           for i in 0 to k loop
8             aux:= aux xor (a(i) and b(k-i));
9           end loop;
10          d(k) <= aux;
11    end process;
12  end generate;

14  gen_xors2: for k in M to 2*M-2 generate
15    l2: process(a,b)
16      variable aux: std_logic:='0';
17        begin
18          aux:='0';
19          for i in k to 2*M-2 loop
20            aux:= aux xor (a(k-i+(M-1)) and b(i-(M-1)));
21          end loop;
22          d(k) <= aux;
23    end process;
24  end generate;
```



Slika 5.4: Računanje koeficientov produkta $d(x)$

m	LUT-i	Rezine	Zakasnitev[ns]
11	58	34	9.741
63	1671	716	15.143
113	5401	1830	19.245
191	15359	5034	25.760

Tabela 5.8: Klasično množenje - redukcija z redukcijsko matriko



Slika 5.5: Modul *Cmultiplier* povezuje modul *ClassicMul* in modul *reduceB* - RTL shema

5.1.3 Množenje s prepletanjem

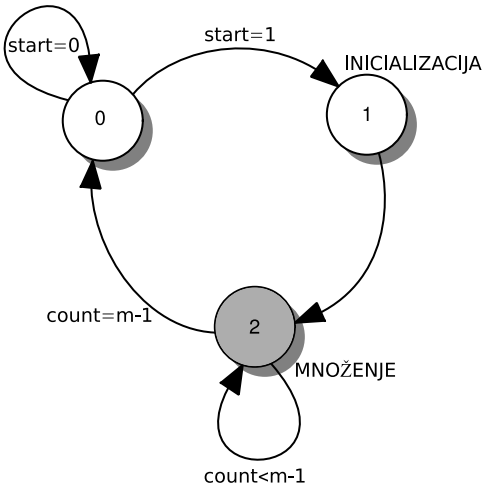
Množilnik je implementiran kot končni avtomat (v nadaljevanju KA) s tremi stanji in tremi registri A, B in C. Stanje 0 je začetno in končno stanje, v katerem množilnik miruje oz. čaka na aktiviranje signala `start` (aktiven v visokem stanju). Aktiven `start` sproži prehod v stanje 1, v katerem postavi signal `inic` za inicializacijo registrov (tako v naslednji urini periodi, torej v stanju 2, v registra A in B vpiše vrednosti vhodnih polinomov $a(x)$, $b(x)$, register C pa postavi na 0). Sledi prehod v stanje 2, v katerem izvaja množenje. Za množenje (stanje 2) je potrebnih m korakov oz. urinih period (potrebujemo števec `count`), nato pa se vrne v stanje 0 in aktivira signal `done`, ki pomeni, da je produkt izračunan. Prehajanje stanj vidimo na sliki 5.1.3. V stanju 2 se spreminjajo vsebine registrov A, B in C. Register A hrani trenutni produkt $d(x)$, kjer $d(x)$ izračunamo po

$$d_0 = a_{m-1}f_0 \quad \text{in} \quad d_i = a_{i-1} + a_{m-1}f_i,\ i = 1, \ldots, m-1.$$

Register B se v vsaki urini periodi pomakne en bit desno (tako na koncu registra dobi želen bit b_i). V isti urini periodi se izvede tudi korak

$$c(x) \leftarrow c(x) \oplus b_i d(x),$$

trenutni rezultat množenja $c(x)$ pa se vpiše v register C.
Za množenje dveh m-bitnih polinomov ostane avtomat m ciklov v stanju 2, množenje v celoti pa traja $(m+3)$ urine periode. Za praktične namene je tak pristop neuporaben, saj je npr. že pri klasični metodi zakasnitev vezja neprimerno manjša od skupnega časa (produkt urine periode in števila urinih period), ki ga za rezultat potrebuje množenje s prepletanjem. Modul je narejen po zgledu iz [18].



Slika 5.6: Množenje s prepletanjem - prehajanje stanj v KA

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
11	41	24	10	1.861	14	26.054
63	199	106	44	1.891	66	124.806
113	351	182	77	2.074	116	240.584
191	586	302	120	3.126	194	606.444

Tabela 5.9: Množenje s prepletanjem - rezultati implementacije (množenje se izvede v $m + 3$ urinih periodah)

5.1.4 Množenje Mastrovito

Enačbo (4.12) in enačbi za računanje elementov vektorjev $\boldsymbol{g}$ in $\boldsymbol{e}$ uporabimo za implementacijo množenja. Vezje je kombinatorično in je glede prostorske zahtevnosti je (tabela 5.10) primerljivo s klasičnim množenjem (tabela 5.8): v primerjavi z množenjem s prepletanjem (tabela 5.9) porabita oba dokaj veliko število LUT-ov, njuna prednost pa je v hitrosti. Opazimo, da iščemo kompromis med prostorsko in časovno zahtevnostjo množilnika.

m	LUT-i	Rezine	Zakasnitev[ns]
11	54	33	8.543
63	1663	708	14.163
113	5405	1906	16.163
191	15362	2084	20.397

Tabela 5.10: Množenje Mastrovito - rezultati implementacije

5.1.5 Množenje Montgomery

Algoritem 4 računa produkt postopno, bit za bitom, vmesne rezultate pa hrani v registre, torej govorimo o sekvenčnem vezju. Algoritem 5 računa delne produkte za G bitov polinoma $a(x)$ naenkrat. V skrajnem primeru lahko izberemo $G = m$, torej $S = 1$, kar pomeni le en obhod zanke. Takrat vezje ne vsebuje pomnilnih elementov in je kombinatorično. Pričakujemo večjo časovno zahtevnost pri sekvenčni izvedbi (algoritem 4) in večjo prostorsko zahtevnost pri kombinatorični izvedbi (algoritem 5 z $G = m$). Z drugače izbranim G pa dobimo hibridno vezje , ki ima večjo stopnjo paralelnosti (stavki v *for* zanki). Vmesne rezultate hrani v registre, hkrati pa ima manjše število obhodov zanke. Najugodnejši kompromis med prostorsko in časovno zahtevnostjo iščemo s spreminjanjem parametra G (dolžine besede).

Sekvenčna različica

Koraki 3, 4 in 5 algoritma 4 so združeni v modulu *montg_cell* (njegov VHDL opis lahko vidimo v zapisu 5.4), ki ima vhode $(c_{m-1}, \ldots, c_0)$, $(b_{m-1}, \ldots, b_0)$ in a_i, ter izhod $(new\_c_{m-1}, \ldots, new\_c_0)$

VHDL koda 5.4: En obhod for zanke (modul *montg_cell*)

```
1 prev_c0 <= c(0) xor (a_i and b(0));
2 datapath: for i in 1 to M-1 generate
3   new_c(i-1) <= c(i) xor (a_i and b(i)) xor (F(i) and prev_c0);
4 end generate;
5 new_c(M-1) <= prev_c0;
```

Modul *montg_cell* je povezan v modul *montgomery_mult*, v katerem izračunamo produkt $c(x) = a(x)b(x) \mod f(x)$. Modul *montgomery_mult* je realiziran kot KA s tremi stanji in tremi registri A, B in C (shema prehajanja stanj avtomata je enaka shemi prehajanja stanj pri množenju s prepletanjem 5.1.3). Stanje 0 je začetno in končno stanje (stanje mirovanja), iz katerega ob aktivnem signalu `start` preide v stanje 1, kjer začne inicializacijo registrov. Nato preide v stanje 2, v katerem izvaja množenje. V vsaki urini periodi izvede en obhod zanke: v modul *montg_cell* pošlje vrednosti registrov B in C ter trenutni bit a_i (v vsaki urini periodi pomakne vsebino registra A en bit v desno in trenutni bit dobi z $a_i = A(0)$), delni produkt pa v naslednji urini periodi vpiše v register C. Avtomat ostane v tem stanju m urinih period. Po tem času je na voljo Montgomeryev produkt $c(x) = a(x)b(x)x^{-m} \mod f(x)$, ki ga pomnoženega z x^m reduciramo (koraka 7 in 8 algoritma 4), in s tem dobimo končni rezultat $c(x) = a(x)b(x) \mod f(x)$, ki je na razpolago po $m + 3$ urinih periodah.

Kombinatorična različica

Kombinatorično vezje pa enostavno povezuje m modulov *montg_cell* tako, da je izhod prejšnjega modula vhod $c(x)$ v naslednjega, z x^m pomnožen in reduciran izhod zadnjega modula pa je iskani

produkt. Produkt $c(x) = a(x)b(x) \bmod f(x)$ je na voljo v času, ki je enak zakasnitvi vezja. Če več zaporedno povezanih osnovnih modulov *montg_cell* smatramo kot "več bitov naenkrat", lahko gledamo na kombinatorično izvedbo kot na algoritem 5 z $G = m$.

Sekvenčna in kombinatorična različica predstavljata dve skrajnosti. Končni avtomat, s katerim je implementirana sekvenčna različica, ima relativno majhno prostorsko zahtevnost, vendar za izračun produkta potrebuje $m + 3$ urine periode. Kombinatorična izvedba ima majhno časovno zahtevnost, vendar toliko večjo prostorsko zahtevnost in daljše podatkovne poti, ki tudi povečajo zakasnitev med vhodom in izhodom vezja. Poiskati je treba ustrezen kompromis med obema možnostima.

Hibrid

Rešitev ponuja končni avtomat, podoben tistemu v sekvenčni realizaciji, vendar pa namesto enega zaporedno povezuje G modulov *montg_cell*, in tako v enem ciklu izračuna G delnih produktov. To zahteva, da register A pomaknemo G bitov desno.

VHDL koda 5.5: En obhod for zanke (G modulov *montg_cell*)

```
1  ccc(0) <= cc;
2  Gcells: for i in 0 to G-1 generate
3   cell: montg_cell port map ( ccc(i), bb,  aa(i),  ccc(i+1));
4  end generate;
```

Dejansko izvajamo prirejeno različico algoritma 5, kjer vnaprejšnje računanje polinoma F_0' ni potrebno, ker z vezavo G modulov *montg_cell* delni produkt $c(x)$ delimo G-krat z x namesto z x^G. VHDL opis za vezavo G modulov *montg_cell* je prikazan v 5.5.

Primer: Oglejmo si razliko med čistim kombinatoričnim in hibridnim vezjem (za izbrane dolžine besede $G = 2, 3, 4$) na primeru dveh polimonov $a(x), b(x)$ iz GF(2^{11}):

$$a(x) = x^9 + x^6 + x^4 + 1, \qquad b(x) = x^{10} + x^4 + x^3$$

Če uporabimo za modulski polinom $f(x) = x^{11} + x^2 + 1$, dobimo:

$$a(x) \cdot b(x) = x^{10} + x^9 + x^3 + x^2 + x$$

V tabeli 5.11 so zbrani delni produkti, izračunani v posamezni urini periodi (*števec* šteje urine periode, ko je končni avtomat v stanju 2) pri kombinatorični različici.

V $11-$ti urini periodi dobljen polinom množimo in reduciramo:

$$(x^9 + x^8 + x^6 + x^4 + x^2 + x + 1) \cdot x^{11} = x^{10} + x^9 + x^3 + x^2 + x$$

števec	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
1	0	1	0	0	0	0	0	1	1	0	0
2	0	0	1	0	0	0	0	0	1	1	0
3	0	0	0	1	0	0	0	0	0	1	1
4	1	0	0	0	1	0	0	0	0	1	1
5	1	0	0	0	0	1	0	1	1	1	1
6	1	1	0	0	0	0	1	0	1	0	1
7	1	0	1	0	0	0	0	0	1	0	0
8	0	1	0	1	0	0	0	0	0	1	0
9	0	0	1	0	1	0	0	0	0	0	1
10	1	1	0	1	0	1	0	1	1	1	0
11	0	1	1	0	1	0	1	0	1	1	1

Tabela 5.11: Kombinatorično vezje (11 urinih period)

Ker ne obstajata G in S tako, da velja $G \cdot S = 11$, v zadnjem obhodu ne upoštevamo zadnjega delnega produkta, temveč tistega, ki ga potrebujemo. Tako na primer za $G = 2$ vzamemo $11 = 6 \cdot 2 - 1$, kar pomeni 6 urinih period v stanju 2 ($S = 6$ =število obhodov zanke), pri čemer v zadnjem ciklu upoštevamo prvi izračunani delni produkt (tabela 5.12). Stolpec *komb* pomeni vrstico oz. urino periodo, v kateri dobimo ta rezultat pri kombinatoričnem vezju, stolpec *števec* pa urino periodo, v kateri dobimo enak delni produkt pri hibridni različici z izbranim G.
V dodatku D je prikazana in opisana RTL shema modula za hibridno Montgomeryjevo množenje pri $m = 11$ in $G = 8$.
Tabele 5.12, 5.13 in 5.14 prikazujejo rezultate, dobljene s hibridnim vezjem pri $G = 2, 3, 4$. Lepo je tudi razvidno, kako se zmanjšuje število potrebnih ciklov. Naj še enkrat poudarimo, da tukaj štejemo le število ciklov pri dejanskem množenju v stanju 2; prehajanje med stanji pomeni še tri dodatne cikle (tabele 5.17, 5.18, 5.19 in 5.20 beležijo celotno število urinih period, ki znaša $S + 3$).

števec	*komb.*	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
1	2	0	1	0	0	0	0	0	1	1	0	0
2	4	1	0	0	0	1	0	0	0	0	1	1
3	6	1	1	0	0	0	0	1	0	1	0	1
4	8	0	1	0	1	0	0	0	0	0	1	0
5	10	1	1	0	1	0	1	0	1	1	1	0
6	11	0	1	1	0	1	0	1	0	1	1	1

Tabela 5.12: Hibridno vezje z $G = 2$ (6 urinih period)

števec	*komb.*	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
1	3	0	0	0	1	0	0	0	0	0	1	1
2	6	1	1	0	0	0	0	1	0	1	0	1
3	9	0	0	1	0	1	0	0	0	0	0	1
4	11	0	1	1	0	1	0	1	0	1	1	1

Tabela 5.13: Hibridno vezje z $G = 3$ (4 urine periode)

števec	*komb.*	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
1	4	1	0	0	0	1	0	0	0	0	1	1
2	8	0	1	0	1	0	0	0	0	0	1	0
3	11	0	1	1	0	1	0	1	0	1	1	1

Tabela 5.14: hibridno vezje z $G = 4$ (3 urine periode)

V tabelah 5.24 in 5.25 vidimo primerjavo sekvenčnega in kombinatoričnega vezja. Kombinatorični Montgomeryjev množilnik vrne rezultat po znatno krajšem času, vendar je njegova prostorska zahtevnost toliko večja (npr. pri $m = 191$ je zasedenih 15% rezin).
Tabele 5.17, 5.18, 5.19 in 5.20 prikazujejo primerjavo sekvenčnega, kombinatoričnega in pa hibridnega vezja pri različnih dolžinah besede G. Stolpec *lastG* v tabelah pove, kateri izmed modulov *montg_cell* bo v obhodu S vrnil pravilni produkt. Za npr. $m = 63$ in $G = 10$ dobimo v sedmem obhodu pravilni rezultat na izhodu tretjega modula *montg_cell*. Če prvi obhod označimo z `count=0`, je pri zadnjem obhodu `count=6` in *lastG* lahko izračunamo takole: $lastG = m - (S-1)G$, torej $lastG = 63 - 6 \cdot 10$.

Rezultati implementacije

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
11	40	23	14	1.455	14	20.370
63	199	105	65	1.807	66	119.262
113	350	201	121	2.119	116	245.804
191	586	309	182	2.982	194	578.508

Tabela 5.15: Množenje Montgomery - sekvenčna verzija - rezultati implementacije (množenje se izvede v $m + 3$ urinih periodah)

V naslednjih tabelah so zbrani rezultati hibridnih vezij za različne m in izbrane G. Vključeni so tudi rezultati sekvenčnega $G = S$ in kombinatoričnega $G = K$ vezja iz tabel 5.25 in 5.24 za lažjo primerjavo.

m	LUT-i	Rezine	Zakasnitev[ns]
11	62	20	10.053
63	2048	905	28.683
113	6498	2255	43.598
191	18431	5719	71.846

Tabela 5.16: Množenje Montgomery - kombinatorična verzija - rezultati implementacije

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
11	S	/	/	40	23	14	1.455	14	20.370
11	2	6	1	39	38	20	1.701	9	15.309
11	3	4	2	38	41	28	1.733	7	12.131
11	4	3	3	37	54	29	1.820	6	10.920
11	6	2	5	37	65	36	1.985	5	9.925
11	8	2	3	37	84	30	2.136	5	10.680
11	10	2	1	37	96	39	2.304	5	11.520
11	K	/	/	/	62	20	/	/	10.053

Tabela 5.17: Množenje Montgomery - hibridna verzija z $m = 11$ - rezultati implementacije

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
63	S	/	/	199	105	65	1.807	66	119.262
63	2	32	1	198	200	132	2.049	35	71.715
63	6	11	3	200	332	178	2.278	14	31.892
63	10	7	3	202	383	215	2.563	10	25.630
63	14	5	7	196	506	225	2.840	8	22.720
63	16	4	15	203	557	271	2.954	7	20.678
63	29	3	5	238	931	339	3.472	6	20.832
63	32	2	31	275	1036	444	3.987	5	19.935
63	38	2	25	285	1225	445	4.032	5	20.160
63	42	2	21	267	1361	458	4.364	5	21.820
63	44	2	19	276	1467	479	4.198	5	20.990
63	K	/	/	/	2048	905	/	/	28.683

Tabela 5.18: Množenje Montgomery - hibridna verzija z $m = 63$ - rezultati implementacije

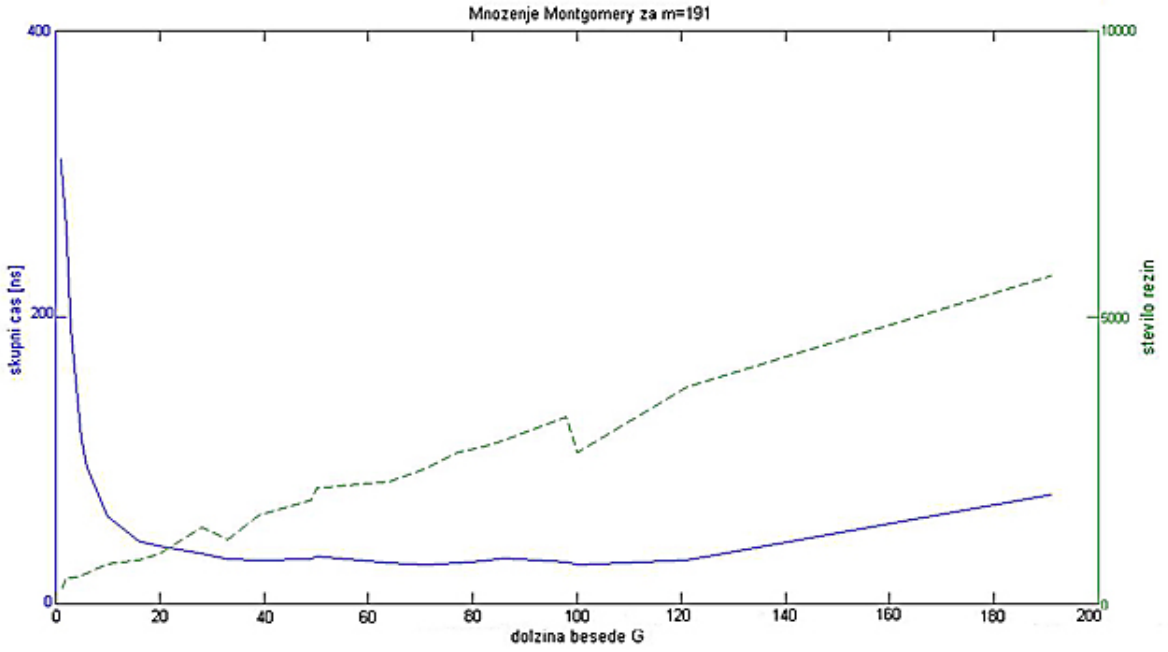
Rezultati potrjujejo, da z večanjem števila bitov G narašča prostorska zahtevnost, hkrati pa se veča tudi minimalna dolžina cikla, saj večji G pomeni več zaporedno vezanih modulov, in s tem večjo zakasnitev. Pri nerazcepnem polinomu z $m = 11$ je kombinatorično vezje skoraj enakovredno hibridnemu pri $G = 6$, ki da najkrajši skupen čas. Prostorska zahtevnost z večanjem parametra G narašča, vendar pa je 39 rezin v primerjavi z vsemi 37680 rezinami, ki so na razpolago, tako zanemarljivo, da morda o prostorski zahtevnosti pri $m = 11$ morda sploh ni smiselno govoriti. Pri ostalih polinomih (npr. $m = 113$ ali $m = 191$) pa opazimo, da so

m	G	S	$lastG$	FF	LUTi	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
113	S	/	/	350	196	129	2.119	116	245.804
113	2	57	1	348	353	217	2.183	60	130.980
113	3	38	2	349	358	257	2.550	41	104.550
113	4	29	1	349	468	241	2.128	32	68.096
113	7	17	1	347	541	319	2.531	20	50.620
113	10	12	3	347	813	382	2.470	15	37.050
113	15	8	8	350	966	438	2.786	11	30.646
113	25	5	13	409	1463	526	3.096	8	24.768
113	36	4	5	435	2064	723	3.166	7	22.162
113	39	3	35	424	2345	739	3.476	6	20.856
113	43	3	27	430	2496	868	3.520	6	21.120
113	48	3	17	422	2818	1093	3.751	6	22.506
113	59	2	54	399	3181	1136	4.176	5	20.880
113	64	2	49	486	3656	1337	4.146	5	20.730
113	71	2	42	503	4025	1419	4.305	5	21.525
113	80	2	33	427	4469	1883	4.805	5	24.025
113	K	/	/	/	6498	2255	/	/	43.598

Tabela 5.19: Množenje Montgomery - hibridna verzija z $m = 113$ - rezultati implementacije

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
191	S	/	/	586	324	210	2.982	194	578.508
191	2	96	1	583	585	445	2.665	99	263.835
191	3	64	2	583	593	446	2.860	67	191.62
191	5	39	1	584	786	475	2.759	42	115.878
191	6	32	5	583	970	528	2.722	35	95.270
191	10	20	1	585	1348	701	2.659	23	61.157
191	16	12	15	591	1757	755	2.900	15	43.500
191	20	10	11	586	2135	865	3.101	13	40.313
191	28	7	23	669	3041	1325	3.468	10	34.680
191	33	6	26	635	3095	1110	3.460	9	31.140
191	39	5	35	676	3967	1538	3.803	8	30.424
191	49	4	44	695	4650	1802	4.388	7	30.716
191	50	4	41	705	4944	2009	4.675	7	32.725
191	64	3	63	665	5729	2132	4.781	6	28.686
191	71	3	49	666	6283	2352	4.543	6	27.258
191	77	3	37	724	7103	2642	4.690	6	28.140
191	82	3	27	796	7546	2737	4.962	6	29.772
191	86	3	19	660	7793	2843	5.262	6	31.572
191	98	2	93	811	8880	3261	5.825	5	29.125
191	100	2	91	759	8879	2634	5.438	5	27.190
191	121	2	70	792	10764	3772	6.112	5	30.560
191	K	/	/	/	18431	5719	/	/	71.846

Tabela 5.20: Množenje Montgomery - hibridna verzija z $m = 191$ - rezultati implementacije



Slika 5.7: Graf prikazuje skupen čas (modre barve) in prostorsko zahtevnost (zelena črtkana črta) hibridnega Montgomeryjevega množilnika v odvisnosti od dolžine besede G pri $m = 191$.

hibridna vezja, ki potrebujejo za izračun produkta več urinih period, ne le veliko hitrejša od kombinatoričnega vezja, temveč tudi prostorsko ugodnejša. Npr. pri $m = 191$ se pri izbranem $G = 100$ poraba rezin v primerjavi s kombinatoričnim vezjem zmanjša skoraj na polovico, skupen čas pa pade na približno 40 odstotkov. Za boljšo preglednost smo podatke o skupnem času in porabljenih rezinah zbrali v grafu 5.7. V vseh primerih je preskok med sekvenčno izvedbo in ostalimi s stališča časovne zahtevnosti ogromen, seveda pa gre na račun povečevanja števila porabljenih elementov.

Primerjava rezultatov vseh implementiranih množilnikov

Ogledali smo si tri kombinatorične množilnike, dva sekvenčna in en hibridni množilnik. Kot pričakovano, so se kombinatorični množilniki (klasično množenje 5.1.2, množenje Mastrovito 5.1.4 in kombinatorični Montgomeryjev množilnik 5.1.5) izkazali za prostorsko najzahtevnejše; samo množenje zasede 13% razpoložljivih rezin. Za časovno najugodnejšega se je izkazal Mastrovito kombinatorični množilnik. Sekvenčni množilnik, osnovan na množenju s prepletanjem 5.1.3, je najpočasnejši, malenskost boljše pa se je odrezala sekvenčna različica Montgomeryjevega množenja. Vendar pa imata sekvenčna množilnika najmanjšo prostorsko zahtevnost. Poglejmo

še hibridne Montgomeryjeve množilnike. V tabelah 5.17, 5.18, 5.19 in 5.20 zbrani rezultati kažejo, da lahko s spreminjanjem parametra G dosežemo najboljši kompromis med prostorsko in časovno zahtevnostjo. Najučinkovitejši hibridni Montgomeryjevi množilniki za dane nerazcepne polinome so zbrani v tabeli 5.21.

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas[ns]
11	6	2	5	37	65	36	1.985	5	9.925
63	32	2	31	275	1036	444	3.987	5	19.935
113	64	2	49	486	3656	1337	4.146	5	20.730
191	100	2	91	759	8879	2634	5.438	5	27.190

Tabela 5.21: Najučinkovitejši hibridni Montgomeryjevi množilniki

5.2 Kvadriranje

5.2.1 Klasično kvadriranje

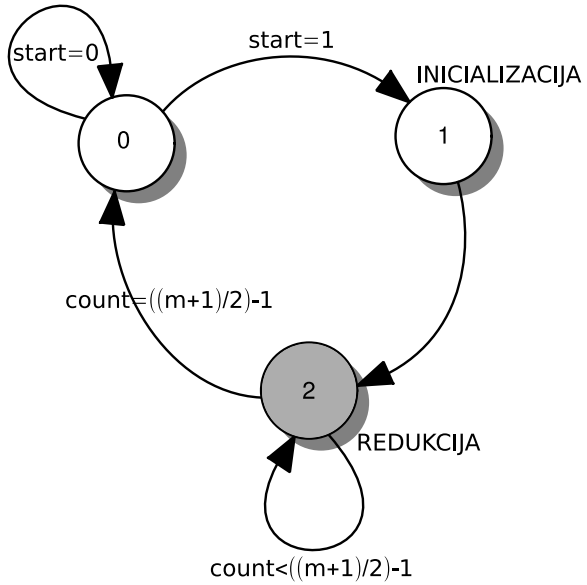
Vhodni vektor $a(x)$ razpršimo, kot narekuje formula (4.16), nato pa reduciramo. Uporabili smo redukcijo z redukcijsko matriko.

m	LUT-i	Rezine	Zakasnitev [ns]
11	3	2	5.054
63	16	16	8.627
113	28	27	10.231
191	48	42	10.196

Tabela 5.22: Klasično kvadriranje - rezultati implementacije

5.2.2 Kvadriranje s prepletanjem

Tudi ta algoritem je realiziran kot končni avtomat s tremi stanji in tremi registri Z, B, W (slika 5.8). Ob inicializaciji v $\frac{m}{2}$-bitni register Z shrani zgornjo polovico vhodnega polinoma $a(x)$, torej vektor $\mathbf{z}$, v m-bitni register W pa spodnjo polovico razpršenega polinoma $a(x)$, torej vključno z vmesnimi ničlami. V register B ob inicializaciji vpiše redukcijski vektor. V stanju 2 izvaja redukcijo; v vsaki urini periodi izračuna vmesni rezultat $c(x) \leftarrow c(x) \oplus (b(x) \cdot z(i))$ ter nov redukcijski vektor $\mathbf{b}$ (za redukcijo v naslednjem obhodu), vsebino registra Z pa pomakne eno mesto v desno. Tako v vsakem obhodu zanke reducira en bit vektorja $\boldsymbol{z}$. V stanju 2 ostane $\frac{m+1}{2} - 1$ urinih period, nato se vrne v stanje 0. Takrat aktivira signal `done`, kar pomeni da, je na izhodu modula iskani kvadrat.



Slika 5.8: Kvadriranje s prepletanjem - prehajanje stanj v KA

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	33	23	12	1.278	9	11.502
63	166	88	26	1.872	35	65.52
113	291	151	48	2.328	60	139.68
191	487	251	121	2.674	99	264.726

Tabela 5.23: Kvadriranje s prepletanjem - rezultati implementacije

5.2.3 Kvadriranje Montgomery

Podobno kot pri Montgomeryjevem množenju imamo tudi tukaj tri možne izvedbe, sekvenčno, kombinatorično in hibridno. Prva in najbolj očitna razlika je v spremenjeni osnovni celici *montg_sq_cell*, ki izvede eno iteracijo zanke algoritma 6.

Sekvenčna različica je realizirana kot KA s tremi stanji, podobno kot pri Montgomeryjevem množenju in množenju s prepletanjem 5.1.3. Namesto treh registrov A, B in C potrebujemo le register C, ki hrani vmesne rezultate, ter števec obhodov zanke `count`. Vhodni polinom $a(x)$ le razpršimo.

Kombinatorično vezje povezuje m modulov *montg_sq_cell* na enak način kot kombinatorično vezje za Montgomeryjev produkt.

Podobno je tudi hibridna različica le prirejeno sekvenčno vezje, ki namesto ene povezuje G osnovnih celic.

Rezultati implementacije

m	FF	LUT	Rezine	u.p.[ns]	Cikli	Skupen čas [ns]
11	29	31	16	1.492	14	20.888
63	134	103	47	2.336	66	154.176
113	235	180	94	2.247	116	260.652
191	393	299	176	3.132	194	607.608

Tabela 5.24: Kvadriranje Montgomery - sekvenčna verzija - rezultati implementacije (kvadriranje se izvede v $m + 3$ urinih periodah)

m	LUT-i	Rezine	Zakasnitev [ns]
11	3	2	5.733
63	16	15	9.347
113	28	25	10.819
191	80	56	12.793

Tabela 5.25: Kvadriranje Montgomery - kombinatorično vezje - rezultati implementacije

Z večanjem dolžine besede G se časovna in prostorska zahtevnost hibridnih modulov za kvadriranje Montgomery spreminjata zelo podobno kot pri hibridnih Montgomeryjevih množilnikih. Zato bomo v tabeli 5.26 za vsak m navedli le najbolj optimalne vrednosti parametra G, tabele z rezultati pa si bralec lahko ogleda v dodatku C.

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	6	2	5	21	31	15	1.535	5	7.675
63	32	2	31	99	166	72	2.487	5	12.435
113	59	2	54	201	283	106	2.099	5	10.495
191	100	2	9	332	532	224	2.469	5	12.345

Tabela 5.26: Kvadriranje Montgomery - hibridna verzija - najboljši rezultati

Primerjava rezultatov modulov za kvadriranje

Ugotovili smo, da je tako časovno kot tudi prostorsko najugodnejše kar klasično kvadriranje 5.2.1. To nas ne preseneča, saj je razpršitev vhodnega vektorja na FPGA trivialna, poraba elementov in zakasnitev skozi vezje pa sta v bistvu odvisna le od redukcije, in še ta je optimizirana, saj orodje `Xilinx-ISE` ugotovi, kateri vhodi so vedno enaki 0, in jih odstrani. Drugo kombinatorično vezje, s katerim se srečamo, je modul za kombinatorično Montgomeryjevo kvadriranje, ki daje podobne rezultate kot klasično kvadriranje.

Kvadriranje s prepletanjem 5.2.2 je sekvenčno vezje, kjer bite, ki predstavljajo potence večje od m, reduciramo postopno, po en bit na obhod. Kljub temu, da lahko smatramo $\frac{m+1}{2} - 1$

kot nizko število obhodov, je skupni čas v primerjavi s kombinatoričnimi vezji zelo visok. Kar občutno večja pa je tudi prostorska zahtevnost. Najslabše rezultate dobimo pri sekvenčnem Montgomeryjevem kvadriranju. Z uporabo hibridnih vezij se sicer zelo približamo nizki časovni zahtevnosti kombinatoričnih vezij, vendar pa ostaja njihova prostorska zahtevnost neprimerljivo večja.

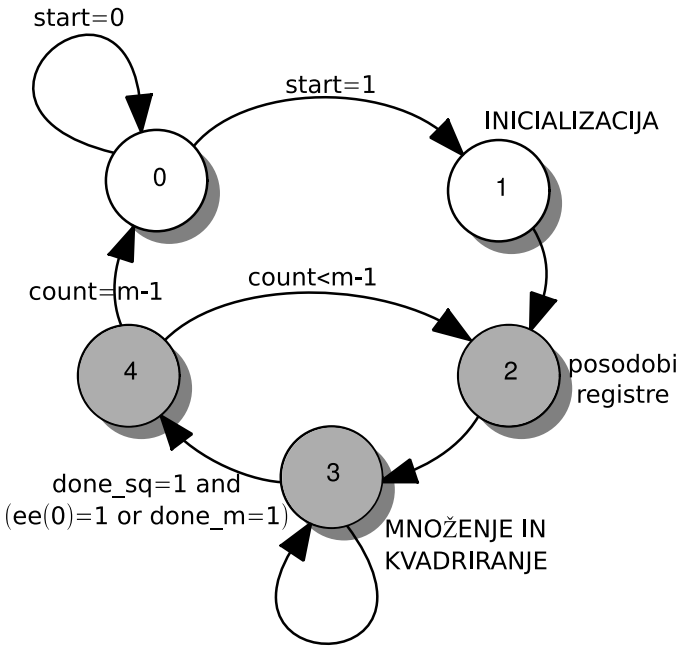
5.3 Potenciranje

5.3.1 Kvadriraj in množi I

Algoritem "kvadriraj in množi" realiziramo s povezavo modula za klasično kvadriranje (poglavje 4.2.1) in modula za množenje s prepletanjem (poglavje 4.1.2). Modul "kvadriraj in množi"je realiziran kot KA s petimi stanji in tremi registri E, C in B ter števcem `count`, ki šteje iteracije zanke (korak 3 v algoritmu 7). Slika 5.9 prikazuje prehajanje stanj avtomata.
Stanje 0 je začetno stanje avtomata, v stanju 1 pa začne inicializacijo registrov (E $\leftarrow$ vhodna potenca, C $\leftarrow$ vhodni vektor $a(x)$, B $\leftarrow$ 1). V stanju 2 postavi kontrolni signal `start_m`, ki je odvisen od trenutnega bita potence, nato pa preide v stanje 3, v katerem izvaja kvadriranje in množenje. V stanju 4 postavi kontrolni bit za shranjevanje (vpis novih vrednosti se izvede v naslednji urini periodi), ter preveri vrednost števca `count`. Ker je potenca e zapisana s k-bitnim vektorjem, potrebujemo k obhodov zanke (korak 3 algoritma 7). Če je števec `count` $< k-1$, se avtomat vrne iz stanja 4 v stanje 2, sicer pa preide v stanje 0 in aktivira signal `done`.

Množenje s prepletanjem potrebuje $m+3$ urine periode za izračun produkta, kvadrat pa je izračunan v eni urini periodi. Tako je tudi število ciklov, ko je avtomat v stanju 3 odvisno od parametra m in od tega, ali v trenutnem obhodu množimo ali ne. Če množenje ni potrebno (pripadajoči bit potence e je enak 0), bo avtomat prešel v stanje 4 po eni urini periodi, če množenje izvaja, pa po $m+3$ urinih periodah.

Opazimo, da je trajanje enega obhoda (in s tem potenciranja) odvisno od tega ali, je potrebno množenje ali ne. V najslabšem primeru, ko je $e = (1 \ldots 1)$, dobimo rezultat po $(3 + k \cdot (m+5))$ urinih periodah, v najboljšem primeru, ko je e potenca števila 2, pa po $3 + 3k$ urinih periodah.
Primer: Tabela 5.27 prikazuje računanje potence a^{27} . Moduli delajo z m-bitnim eksponentom ($k = m$), mi pa si oglejmo primer računanja potence pri $k = 8$. Stolpec *count* pove, v katerem obhodu zanke smo, $e(count)$ pa je pripadajoči bit potence $e = (00011011)$. Stolpca *regC* in *regB* prikazujeta kvadrat in produkt, izračunana v obhodu *count*, zadnji stolpec pa prikazuje število urinih period, potrebnih za posamezen obhod.



Slika 5.9: Kvadriraj in množi potenciranje - prehajanje stanj v KA pri $k = m$

count	$e(count)$	regC	regB	Št. urinih period
0	1	a^2	$a \cdot 1 = a$	$m+5$
1	1	a^4	$a^2 \cdot a = a^3$	$m+5$
2	0	a^8	a^3	3
3	1	a^{16}	$a^8 \cdot a^3 = a^{11}$	$m+5$
4	1	a^{32}	$a^{16} \cdot a^{11} = a^{27}$	$m+5$
5	0	a^{64}	a^{27}	3
6	0	a^{128}	a^{27}	3
7	0	a^{256}	a^{27}	3

Tabela 5.27: Kvadriraj in množi - primer: računanje a^{27}

Iz tabele 5.27 lahko razberemo, da se pri $e = 27$ celotna *for* zanka izvede v $(4m + 32)$ urinih periodah, potenciranje pa traja $(4m + 35)$ urinih period.

Primer: Oglejmo si še primer računanja inverza pri $k = m = 11$, torej potence z eksponentom $2^m - 2$, to je $e = (11111111110)$. Videli bomo, da za računanje inverza potrebujemo $3 + (m-1)(m+5)$ urinih period.

count	*e*(*count*)	regC	regB	Št. urinih period
0	0	a^2	1	3
1	1	a^4	$a^2 \cdot 1 = a^2$	$m+5$
2	1	a^8	$a^2 \cdot a^4 = a^6$	$m+5$
3	1	a^{16}	$a^6 \cdot a^8 = a^{14}$	$m+5$
4	1	a^{32}	$a^{14} \cdot a^{16} = a^{30}$	$m+5$
5	1	a^{64}	$a^{30} \cdot a^{32} = a^{62}$	$m+5$
6	1	a^{128}	$a^{62} \cdot a^{64} = a^{126}$	$m+5$
7	1	a^{256}	$a^{126} \cdot a^{128} = a^{254}$	$m+5$
8	1	a^{512}	$a^{254} \cdot a^{256} = a^{510}$	$m+5$
9	1	a^{1024}	$a^{510} \cdot a^{512} = a^{1022}$	$m+5$
10	1	a^{2048}	$a^{1022} \cdot a^{1024} = a^{2046}$	$m+5$

Tabela 5.28: Kvadriraj in množi - primer: računanje a^{2046}

m	FF	LUTi	Rezine	u.p. [ns]	št ciklov	Skupen čas [ns]
11	82	59	23	1.849	179	330.971
63	401	276	97	2.276	4287	9757.212
113	703	536	243	2.666	13337	35556.44
191	1175	890	385	2.811	37439	105241.03

Tabela 5.29: Kvadriraj in množi I: klasično kvadriranje in množenje s prepletanjem - rezultati implementacije

Rezultati implementacije

Pri potenciranju kvadriraj in množi je število potrebnih urinih period odvisno od potence. V tabeli 5.29 je upoštevan časovno najzahtevnejši primer $e = (1 \ldots 1)$, ki za potenciranje porabi $(3 + m \cdot (m + 5))$ ciklov.

5.3.2 Kvadriraj in množi II

Oglejmo si, kaj se zgodi, če uporabimo klasično množenje namesto množenja s prepletanjem. Ker je klasično množenje realizirano s kombinatoričnim vezjem, lahko spremenimo KA tako, da ima le tri stanja, podobno kot KA za množenje s prepletanjem na sliki 5.1.3, le da sedaj v stanju 2 množimo in kvadriramo. Obe operaciji se izvedeta v eni urini periodi. Tako potrebujemo za potenciranje skupaj le $m + 3$ urine periode. Rezultati modula, ki za potenciranje uporablja klasično množenje 5.1.2 in klasično kvadriranje 5.2.1, so zbrani v tabeli 5.30. Po pričakovanju dobimo veliko večjo prostorsko zahtevnost in daljšo urino periodo, skupen čas pa je občutno manjši.

Kljub lepim rezultatom pa pri implementaciji nastopijo problemi. Opozorila, ki jih izpisuje

m	FF	LUT-i	Rezine	u.p. [ns]	št ciklov	Skupen čas [ns]
11	39	77	35	2.565	14	35.910
63	331	1777	676	7.669	66	506.1054
113	593	5592	2074	8.461	116	981.476
191	951	15694	6315	10.266	194	1991.604

Tabela 5.30: Kvadriraj in množi II: klasično kvadriranje in klasično množenje - rezultati implementacije

orodje `Xilinx-ISE` med postopkom mapiranja (MAP), kažejo, da bo treba za dejansko implementacijo napisati prostorske uporabniške omejitve (user constraints). Modul *expKK* je dovolj enostaven (preprost KA s tremi stanji in majhnim številom kontrolnih signalov in registrov, na nižjem nivoju povezuje dva modula), da si lahko podrobneje oogledamo njegovo RTL shemo, ki jo vidimo na sliki 5.10. Nekateri elementi (npr. register za števec `count` in logika za povečevanje njegove vrednosti) so zaradi večje preglednosti odstranjeni iz sheme. Podatkovni del vezja je razdeljen na tri sklope različnih barv:

- računanje produkta (modro):
  - modul *Cmultiplier* za klasično množenje 5.1.2;
  - multipleksor za izbiro vrednosti registra BB;
  - register BB, ki hrani trenutni produkt;
- računanje kvadrata (rdeče):
  - modul *square* za klasično kvadriranje 5.2.1;
  - multipleksor za izbiro vrednosti registra CC;
  - register CC, ki hrani trenutni kvadrat;
- eksponent (rumeno):
  - multipleksor za izbiro vrednosti registra EE;
  - pomikalni register EE.

V desnem spodnjem kotu sheme vidmo kontrolni del vezja, to je končni avtomat z imenom `current_state` (zelene barve). `current_state` hrani trenutno notranje stanje avtomata, na spremembe vhodnih signalov pa reagira tako, da zamenja notranje stanje in posledično postavi nove vrednosti izhodov. KA ima štiri vhode: `clk, reset, start` in vhod za števec `count`. Ob aktivnem signalu `start` KA preide v stanje 1 in postavi kontrolni signal `inic`. Le-ta je povezan kot izbirni vhod (`select`) na multipleksorje, ki določijo vrednost registrov. Registra `EE` in `CC` sta označena na shemi, register `BB` pa predstavlja "stolpec" D-pomnilnih celic (modre barve, ob desnem robu sheme). Ob aktivnem signalu `inic` multipleksorji izberejo vrednost 0 za eksponent, vhodni vektor $a(x)$ za kvadrat ter vrednost 1 za produkt. Manjkajoči register `count`, ki hrani število obhodov, pa postavi na vrednost 0. Nato KA preide v stanje 2 in aktivira signal

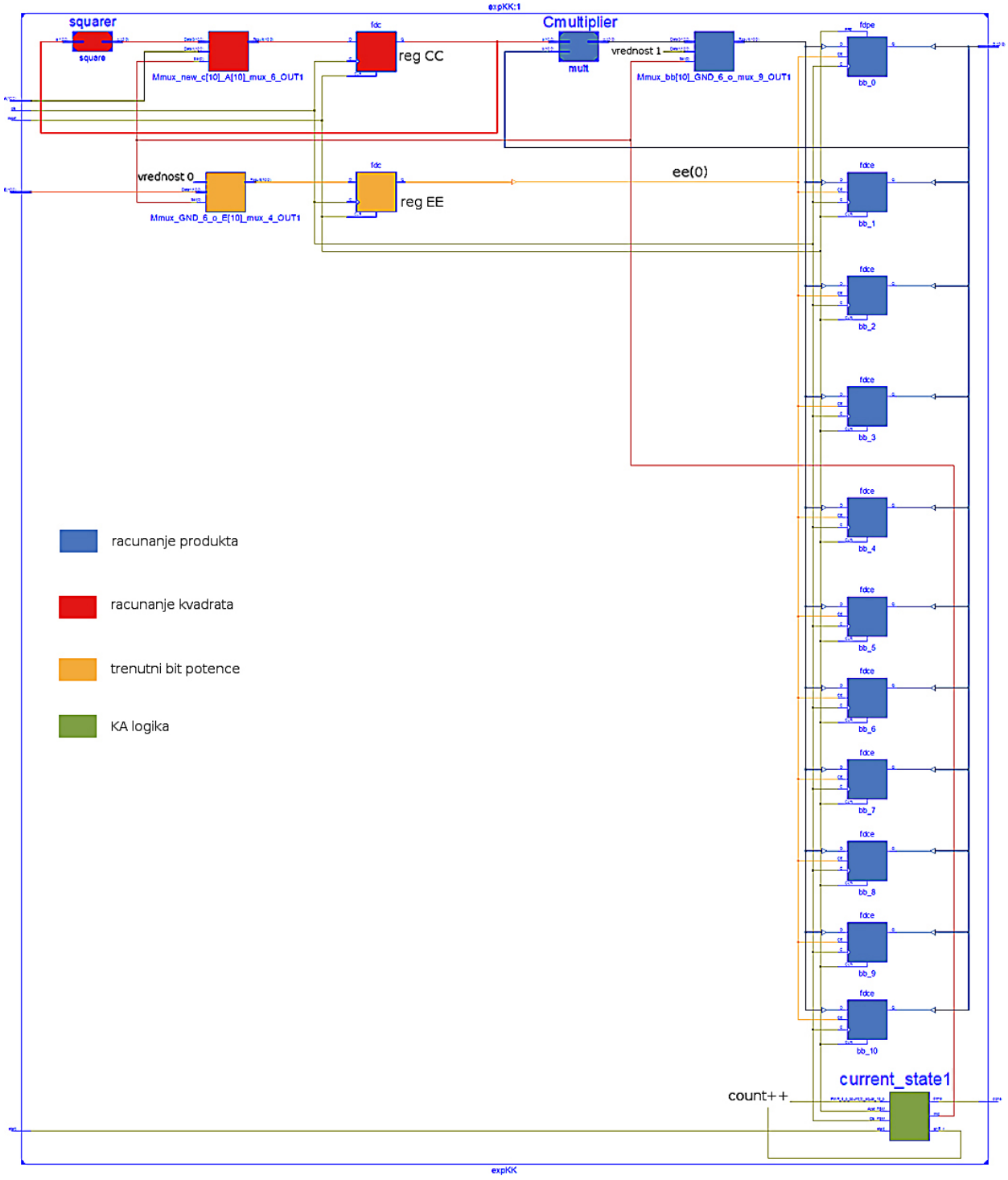
`shift_r`, ki ga potrebuje logika za povečevanje števca.

Modula *square* in *Cmultiplier* v vsaki urini periodi izračunata kvadrat oz. produkt. Ker je signal `inic` v stanju 2 enak 0, bosta multipleksorja spustila izhode teh dveh modulov v registre. Register `EE` se vsako urino periodo pomakne en bit v desno. Trenutni bit $ee(0)$ služi kot `CE` (clock enable) vhod v pomnilne celice registra `BB`. Če je $ee(0) = 1$, se produkt shrani v posamezne pomnilne celice, če je $ee(0) = 0$, pa se stanje celic ne spremeni. Po m urinih periodah bodo v registru `EE` same ničle. Od tega trenutka naprej se vrednost registra `BB` ne spreminja več. KA takrat preide v stanje 0 in postavi izhodni bit `done`, ki pove, da je na izhodu modula *expKK* pravilen rezultat.

5.3.3 Kvadriraj in množi - Montgomery I

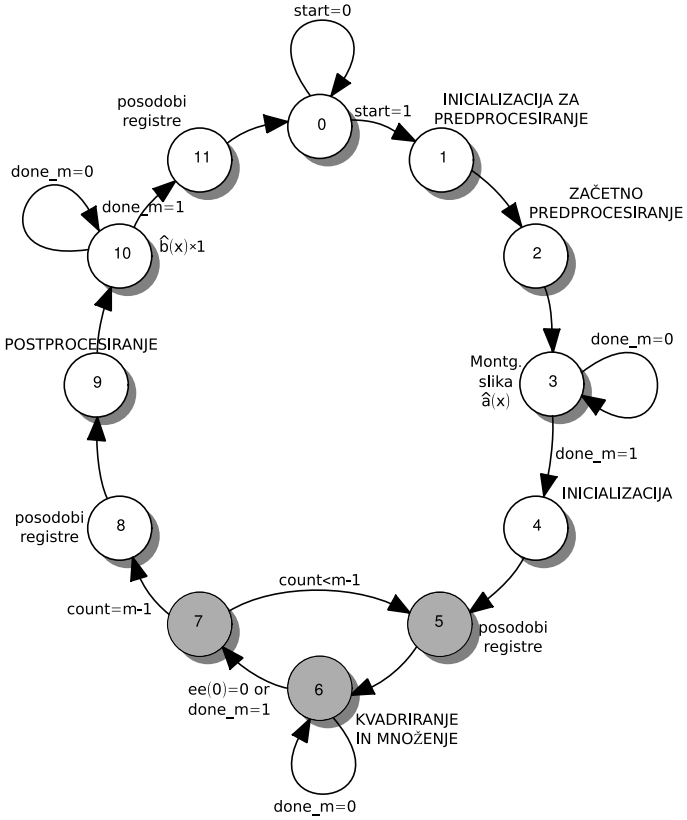
Zaradi prehoda na Montgomeryjevo sliko imamo kar nekaj pred- in post-procesiranja, in zato tudi končni avtomat s kar dvanajstimi stanji, ki ga lahko vidimo na sliki 5.11. Še vedno imamo tri registre: pomikalni register E, ki hrani eksponent, register C, ki hrani trenutni kvadrat, in pa register B, ki hrani trenutno potenco. Prav tako ostane števec `count`, ki šteje število obhodov zanke.

Iz začetnega stanja 0 ob aktivnem signalu `start` KA preide v stanje 1. V stanju 1 postavi kontrolni signal `first`, s katerim doseže, da se v stanju 2 v register C vpiše polinom $r(x)$, v register B pa vhodni polinom $a(x)$. KA v stanju 2 postavi tudi signal `start_m`, in tako se v stanju 3 začne računanje Montgomeryjeve slike $\widehat{a}(x)$ (korak 2 algoritma 8). Število urinih period, ki so potrebne za izračun $\widehat{a}(x)$, je odvisno od uporabljene različice Montgomeryjevega množilnika. Po rezultatih implementacije Montgomeryjevega množenja (5.1.5) vidimo, da je smiselna uporaba hibridnega množilnika. Tako množenje pri izbrani dolžini besede G traja $S + 3$ urine periode. Po končanem množenju avtomat preide v stanje 4 in aktivira signal `inic`. V stanju 5 shrani začetne vrednosti v registre (E $\leftarrow \mathbf{e}$, B $\leftarrow \mathbf{r}$ in C $\leftarrow \widehat{\mathbf{a}}$). Nato začne z dejanskim potenciranjem. V stanju 5 postavi kontrolni bit `start_m`, ki je odvisen od trenutnega bita eksponenta. V stanju 6 izračuna kvadrat (po rezultatih implementacije Montgomeryjevega kvadriranja (5.2.3) izberemo kombinatorično različico), in če to zahteva signal `start_m`, tudi produkt. Če je množenje potrebno, se KA v stanju 6 nahaja $countG + 3$ urine periode, sicer pa v stanje 7 preide že po eni urini periodi. V stanju 7 preveri vrednost števca `count` in postavi kontrolni signal `shift_r`, ki pomeni, da bo v naslednji urini periodi (stanje 5 ali stanje 8) v registre vpisal nove vrednosti. Stanja 5, 6 in 7 predstavljajo en obhod *for* zanke iz algoritma 8. Po $m - 1$ obhodih zanke je potenciranje končano.



Slika 5.10: Kvadriraj in množi potenciranje - prehajanje stanj v KA pri $k = m$

V stanju 8 KA še zadnjič posodobi vse registre in postavi kontrolni signal `last`, ki pove, da sledi še zadnje Montgomeryjevo množenje: v stanju 9 pripravi ustrezne vrednosti registrov (B ostane nespremenjen, to je potenca $\widehat{b}$, C $\leftarrow 1$), ter postavi bit `start_m` za začetek množenja. V stanju 10 izračuna produkt $\widehat{b} \times 1$ (v tem stanju KA ostane $S+3$ urine periode), ki se v stanju 11 zapiše v register B.



Slika 5.11: Potenciranje - Montgomery I - prehajanje stanj v KA

Pri potenciranju, ki uporablja hibriden Montgomeryjev množilnik, je število potrebnih urinih period odvisno od potence in izbrane dolžine besede G. Tako imamo $S+7$ urinih period pred vstopom v zanko, $m(S+5)$ urinih period za zanko in $S+7$ urinih period po zanki, skupaj torej $((m+2) \cdot S + 5m + 14)$ ciklov za izračun potence.

Rezultati implementacije

V tabeli 5.31 je upoštevan časovno najzahtevnejši primer $e = (1 \ldots 1)$, ki za potenciranje porabi $((m+2) \cdot S + 5m + 14)$ ciklov.

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	6	2	5	78	106	40	2.388	95	226.860
63	32	2	31	478	1222	447	5.034	459	2310.606
113	64	2	49	740	3689	1230	4.870	809	3939.830
191	100	2	91	1401	9487	3192	6.103	1355	8269.565

Tabela 5.31: Kvadriraj in množi - Montgomery I

5.3.4 Kvadriraj in množi - Montgomery II

Končni avtomat je v tem primeru enak kot KA na sliki 5.1.3, le da se v koraku 3 izvajata Montgomeryjevo množenje (hibridna različica) in kvadriranje (kombinatorična različica), ki vrneta že z $r(x)$ pomnožen in reduciran rezultat. Potrebno število urinih period se tako zmanjša iz $3 + m(m + 5)$ na $3 + m(S + 5)$.

Rezultati implementacije

V tabeli 5.32 je upoštevan časovno najzahtevnejši primer $e = (1 \dots 1)$, ki za potenciranje porabi $(m(S + 5) + 3)$ ciklov.

m	G	S	$lastG$	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	6	2	5	78	111	52	2.731	80	218.480
63	32	2	31	478	1255	493	4.503	444	1999.332
113	64	2	49	740	3751	1285	5.050	794	4009.700
191	100	2	91	1401	9494	3419	6.020	1340	8066.800

Tabela 5.32: Kvadriraj in množi - Montgomery II

Primerjava rezultatov implementiranih modulov za potenciranje

Ogledali smo si štiri različice potenciranja po metodi "kvadriraj in množi". Najboljše rezultate dobimo za modul *expKK*, ki uporablja klasično kvadriranje in klasično množenje. Modula, ki uporabljata Montgomeryjevo množenje, dajeta dosti boljše rezultate modula, ki uporablja množenje s prepletanjem. Slednji ima sicer najmanjšo prostorsko zahtevnost, vednar pa je s stališča skupnega časa praktično neuporaben. V primerjavi s potenciranjem Montgomery II, porabi modul *expKK* dvakrat več logičnih elementov, vendar pa rezultat vrne v štirikrat krajšem času. Tako ugotovimo, da je najbolj optimalno potenciranje, ki je opisano pod "kvadriraj in množi II", tj. modul *expKK*.

5.4 Računanje inverza

Povedali smo že, da lahko inverz izračunamo s potenciranjem, ki pa je, kot smo videli v prejšnjem razdelku, drago in zamudno. Zato si bomo ogledali še nekaj postopkov za izračun inverza, ki pa temeljijo na Evklidovem algoritmu.

5.4.1 Razširjen Evklidov algoritem

V poglavju 4.4.1 zapisan algoritem 10 ima le še eno pomankljivost, in sicer ugotavljanje stopnje polinoma. Pogoj v *while* zanki sicer lahko nadomestimo z $r_1 \neq 1$, vendar pa sama *while* zanka za implementacijo na FPGA ne pride v poštev; poznati moramo točno število obhodov. Računanju razlike v stopnjah polinomov se izognemo tako, da namesto enkratnega zamika za d mest register zamikamo postopno, za eno mesto naenkrat, tj. množenje $x^d r_1$ nadomestimo z d-kratnim množenjem xr_1. Pri vsakem pomiku pa povečamo vrednost registra d, ki hrani iskano razliko v stopnjah polinomov r_1 in r_2. Deljitelj r_2 torej delimo z $x^d r_1$. Tako se znebimo najvišje potence deljitelja. Če je stopnja števca po tem še vedno večja od stopnje originalnega (nezamaknjenega) imenovalca, tj. če $d > 0$, nadaljujemo z naslednjim bitom števca. Če je le-ta enak 0, moramo števec pomikati v levo (množiti z x) toliko časa, da je $r_2 = 1$. Pri tem seveda ustrezno zmanjšujemo d. Ko stopnja števca pade pod stopnjo imenovalca, tj. $d = 0$, zamenjamo vrednosti registrov ($r_2 \leftrightarrow r_1$). Takšno delno deljenje potrebuje $2m$ obhodov zanke; glej [13]. Povejmo še, da moramo vselej, kadar množimo ali delimo register r_i, enako narediti tudi s pripadajočim p_i, npr. $r_2 \leftarrow r_2 \oplus x^d r_1$ in $p_2 \leftarrow p_2 \oplus x^d p_1$.
Oglejmo si sedaj algoritem 13 (glej [20, 18]), ki polinome deli postopoma, razliko v stopnjah pa hrani v registu d, ki v bistvu šteje leve ($d \leftarrow d + 1$) in desne ($d \leftarrow d - 1$) pomike registrov.

Implementacija

Algoritem 13 realiziramo kot končni avtomat s tremi stanji in petimi registri (`R2, R1, P2, P1, d`) ter števcem `count`, ki hrani število obhodov *for* zanke. KA je podoben tistemu na sliki 5.1.3, s to razliko, da je pogoj za prehod iz stanja 2 v stanje 0 `count`$= 2m$ in da v stanju 2 izvaja delno deljenje (en obhod *for* zanke). Notranjost *for* zanke je realizirana v modulu *eea_data_path* z nekaterimi spremembami. Algoritem 13 je zapisan tako, da se stavki izvajajo en za drugim. To pomeni, da mora na primer vrednost r_2 v koraku 15 upoštevati že spremenjeno vrednost iz stavka 10. Če dva *if* bloka zapišemo v VHDL jeziku tako, kot sta zapisana v algoritmu 13, se njuni stavki izvedejo sočasno. Tudi stavek v koraku 13 se izvede hkrati s stavki v obeh *if* blokih. Ta problem rešimo tako, da enega izmed obeh *if* blokov repliciramo in ga postavimo v drugi *if* blok, kot vidimo v 5.6, in tako, da posamezne korake združimo, npr. korak 10 in korak 13 iz algoritma 13 združimo v $r_2 \leftarrow (r_2 \oplus r_1) \cdot x$; glej [18]. Implementacijo z omenjenimi spremembami

Algoritem 13 Razširjen Evklidov algoritem za polinome 3

VHOD: polinom a
IZHOD: inverz $p = a^1$

```
 1: r_2 ← f, p_2 ← 0
 2: r_1 ← a, p_1 ← 1
 3: for i = 0 to 2m − 1 do
 4:       if r_1(m) = 0 then
 5:             r_1 ← x r_1
 6:             p_1 ← x p_1
 7:             d ← d + 1
 8:       else
 9:             if r_2(m) = 1 then
10:                   r_2 ← r_2 ⊕ r_1
11:                   p_2 ← p_2 ⊕ p_1
12:             end if
13:             r_2 ← x r_2
14:             if d = 0 then
15:                   {r_2 ↔ r_1}
16:                   {p_1 ← x p_2,   p_2 ← p_1}
17:                   d ← 1
18:             else
19:                   p_1 ← p_1/x
20:                   d ← d − 1
21:             end if
22:       end if
23:       i ← i + 1
24: end for
25: return p_1
```

si lahko ogledate v VHDL zapisu 5.6.

VHDL koda 5.6: Modul *eea_data_path*

```
process(r1,r2,p1,p2,d)
begin
  if r1(m) = '0' then
    new_r1 <= r1(M-1 downto 0) & '0';
    new_p1 <= p1(M-1 downto 0) & '0';
    new_r2 <= r2;
    new_p2 <= p2;
    new_d <= d + 1;
  else
   if d = ZERO then
     if r2(m) = '1' then
       new_r1 <= (r2(M-1 downto 0) xor r1(M-1 downto 0)) & '0';
       new_p1 <= (p2(M-1 downto 0) xor p1(M-1 downto 0)) & '0';
     else
       new_r1 <= r2(M-1 downto 0) & '0';
       new_p1 <= p2(M-1 downto 0) & '0';
     end if;
     new_r2 <= r1;
     new_p2 <= p1;
     new_d <= (0=> '1', others => '0');
   else
     new_r1 <= r1;
     new_p1 <= '0' & p1(M downto 1);
     if r2(m) = '1' then
       new_r2 <= (r2(M-1 downto 0) xor r1(M-1 downto 0)) & '0';
       new_p2 <= (p2 xor p1);
     else
       new_r2 <= r2(M-1 downto 0) & '0';
       new_p2 <= p2;
     end if;
     new_d <= d - 1;
   end if;
  end if;
end process;
```

m	FF	LUT-i	Rezine	u.p. [ns]	Št. ciklov	Skupen čas [ns]
11	59	92	35	1.571	25	39.275
63	270	441	209	2.521	129	325.209
113	472	663	370	3.473	229	795.317
191	785	1092	612	3.882	385	1494.57

Tabela 5.33: Razširjen Evklidov algoritem - rezultati implementacije

5.4.2 Berlekampov algoritem

Posamezen korak Berlekampovega algoritma realiziramo v modulu *eea_bMask*. Kot smo že povedali v poglavju 4.4.2, imamo v tem koraku le 4 možnosti: zamikanje enega ali drugega registra in pa operacije nad enim in drugim registrom. Podrobneje si oglejmo primer, ko sta MSB obeh registrov enaka 1 ($r_2(m+1) = r_1(m+1) = 1$) in je $k = 0$, pri čemer prištejemo levi del (od MSB do vejice d_1) spodnjega registra r_1 v levi del zgornjega registra r_2 in desni del

zgornjega registra r_2 (od vejice d_2 do LSB) v desni del spodnjega registra r_1. Realizacijo teh operacij vidimo v zapisu 5.7.

VHDL koda 5.7: korak Berlekampovega algoritma za $k = 0$

```
1 --spremenimo levi del zgornjega registra r2
2         new_r2(M+1 downto d1)<=r2(M+1 downto d1) xor r1(M+1 downto d1);
3         new_r2(d1-1 downto 0)<=r2(d1-1 downto 0);
4 --spremenimo desni del spodnjega registra r1
5         new_r1(M+1 downto d2)<=r1(M+1 downto d2);
6         new_r1(d2-1 downto 0)<=r2(d2-1 downto 0) xor r1(d2-1 downto 0);
```

Uporaba dveh povečujočih se indeksov je nerodna, zato namesto števil d_1 in d_2 raje uporabimo dva $(m + 2)$-bitna registra masko. Spodaj vidimo povezavo med registrom in njegovo masko: tabela 5.34 prikazuje začetno stanje registra `R2` in njegovo masko, tabela 5.35 pa vmesno stanje registra `R1` in njegovo masko. V spodnjih tabelah smo navidezno vejico nadomestili s črto med posameznimi biti.

R2	r_3	r_2	r_1	r_0	p_0
D2	1	1	1	1	0

Tabela 5.34: Začetno stanje registra `R2` in njegova maska

R1	r_1	r_0	p_0	p_1	p_2
D1	1	1	0	0	0

Tabela 5.35: Vmesno stanje registra `R1` in njegova maska

V poglavju 4.4.2 smo povedali, da je $k = 0$, ko je $d_1 > d_2$. Z uporabo mask se ta pogoj obrne. Če je vejica v registru r_1 bolj levo od vejice v registru r_2, je sicer indeks, na katerem se nahaja "bolj leva" vejica, res večji od indeksa druge vejice, ko pa uporabimo maske, pa ima maska za "bolj levo" vejico manj enic kot maska za drugo vejico. Tako je tudi vrednost maske d_1 v resnici manjša od vrednosti maske d_2.

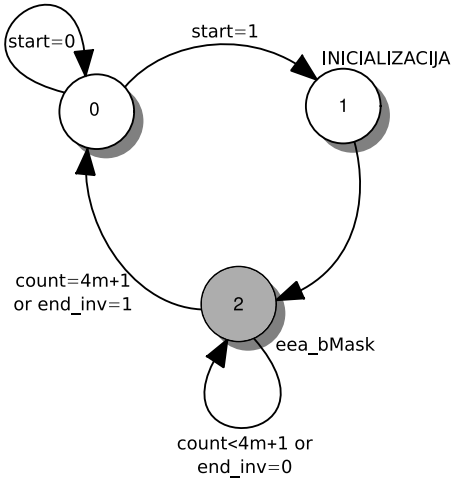
Z uporabo mask se zapis 5.7 spremeni takole:

VHDL koda 5.8: korak Berlekampovega algoritma z uporabo mask

```
1 --spremenimo levi del zgornjega registra r2
2         new_r2<=(r1 and d1) xor r2;
3 --spremenimo desni del spodnjega registra r1
4         new_r1<=r1 xor (r2 and (not d2));
```

Berlekampov algoritem realiziramo kot KA (slika 5.12) s tremi stanji in 4 registri (`r1, d1, r2, d2`), števcem `count` ter bitom `k`. Vrednosti, ki jih izračuna modul `eea_bMask`, posodobimo v stanju 2. Po $4m + 1$ korakih je ena maska postala enaka 0. $4m + 1$ je zgornja meja števila obhodov, vendar običajno maska postane 0 že prej. Zato avtomatu dodamo še kontrolni signal `end_inv`. Ko je `count`$= 4m + 1$ ali ko je postavljen signal `end_inv`, se KA vrne v stanje 0, na izhod pa pošlje "obrnjen" inverz iz maski pripadajočega registra.

V tabeli 5.36 imamo dva stolpca z imenom "Število ciklov", in tako tudi dva stolpca s skupnim časom. V prvem stolpcu je upoštevana maksimalna zgornja meja $4m + 4$, vrednosti v stolpcu



Slika 5.12: Berlekampov algoritem za računanje inverza - prehajanje stanj v KA

m	FF	LUT-i	Rezine	u.p. [ns]	Število ciklov	Skupen čas [ns]	Število ciklov 2	Skupen čas 2 [ns]
11	70	146	48	2.690	48	129.12	38	102.22
63	332	598	194	3.672	256	940.032	193	708.696
113	583	1052	310	4.098	456	1868.688	344	1409.712
191	783	1405	410	6.932	768	5323.776	578	4006.696

Tabela 5.36: Berlekampov algoritem - rezultati implementacije

"Število ciklov 2" pa so dobljene v simulacijah kot povprečno število ciklov za izračun inverza na 5000 vhodnih vektorjih.

5.4.3 Modificiran skoraj inverzni algoritem

Korake 20 do 29 algoritma 11, ki novo vrednost (vsoto $r_1 \oplus r_2$) priredijo tistemu registru, ki vsebuje polinom višje stopnje, lahko nadomestimo z zaporedjem

if $\mathrm{st}(r_1) < \mathrm{st}(r_2)$ **then**

$r_1 \leftrightarrow r_2, \quad p_1 \leftrightarrow p_2$

end if

$r_1 \leftarrow r_1 \oplus r_2, \quad p_1 \leftarrow p_1 \oplus p_2$

S takšno zamenjavo polinomov se lahko znebimo druge notranje *while* zanke (koraki 12 do 19 v algoritmu 11) in pa preverjanja na koncu, saj bo inverz zagotovo p_1. Korake 25 do 29 nadomestimo s stavkom `if` r_1 `then return` p_1, in tako končamo zunanjo while zanko. Glede preverjanja stopnje polinomov imamo dve možnosti. Prva in prostorsko zahtevnejša izračuna stopnjo polinomov (izračun vidimo v zapisu 5.9), druga pa le primerja njuni vrednosti obeh

polinomov. Tako bo imel npr. polinom $r_1 = x^3 + x$ binarno vrednost 10, polinom $r_2 = x^2$ pa vrednost 4. Stopnja prvega je 3, stopnja drugega pa 2. V tem primeru je vrednost izjave $r_1 < r_2$ enaka vrednosti izjave $st(r_1) < st(r_2)$. Pričakovali bi, da s primerjanjem vrednosti pride do problemov pri polinomih z enako stopnjo. Če primerjamo npr. polinom $r_1 = x^3 + x$ vrednosti 10, s polinomom $r_2 = x^3 + x + 1$ vrednosti 11, bo prišlo do zamenjave registrov, kljub temu, da je njuna stopnja enaka. Vendar pa to ne predstavlja nobenega problema, saj manipuliramo par polinomov (r_1, p_1), zaradi česar se bo postopek nadaljeval pravilno, vendar šele v naslednji iteraciji zunanje *while* zanke.

VHDL koda 5.9: Računanje stopnje polinoma a

```
1  degr_a: process(clk, reset)
2     variable d: natural;
3   begin
4     d := 0;
5     for i in 0 to m-1 loop
6       if (a(i)) = '1' then d := i; end if;
7     end loop;
8     degree_a <= conv_std_logic_vector(d,logM+1);
9   end process;
```

Algoritem 14 MAIA - Modificiran skoraj inverzni algoritem 2

VHOD: polinom a
IZHOD: inverz $p = a^1$

1: $r_2 \leftarrow f$, $p_2 \leftarrow 0$
2: $r_1 \leftarrow a$, $p_1 \leftarrow 1$
3: **while** $r_1 \neq 1$ **do**
4: **while** $r_1(0) = 0$ **do**
5: $r_1 \leftarrow r_1/x$
6: **if** $p_1(0) = 0$ **then**
7: $p_1 \leftarrow p_1/x$
8: **else**
9: $p_1 \leftarrow (p_1 + f)/x$
10: **end if**
11: **end while**
12: **if** $st(r_1) < st(r_2)$ **then**
13: $r_1 \leftrightarrow r_2, \quad p_1 \leftrightarrow p_2$
14: **end if**
15: $r_1 \leftarrow r_1 \oplus r_2, \quad p_1 \leftarrow p_1 \oplus p_2$
16: **end while**
17: **return** p_1

Algoritem 14 je realiziran kot KA s tremi stanji in štirimi registri `r1, p1, r2` in `p2`, pri čemer v primeru računanja stopnje polinomov potrebujemo še dva dodatna registra za stopnji polinomov

r_1 in r_2. KA v stanju 2 računa inverz (notranjost *while* zanke v koraku 3 algoritma 14), v stanje 0 pa se vrne, ko postane vrednost registra `r1` enaka 1 (takrat postavi kontrolni signal `end_inv`). Tako število urinih period, potrebnih za izračun inverza, ni znano in se spreminja z vhodnimi vektorji (polinomi). Vemo le, da bomo inverz dobili po največ $3m$ obhodih zunanje *while* zanke, skupaj torej po največ $3m + 3$ urinih periodah. KA je podoben tistemu na sliki 5.12, le da je pogoj `count` sedaj enak $3m$. V tabeli 5.37 so zbrani rezultati modula, ki računa dejansko stopnjo polinomov, v tabeli 5.38 pa rezultati modula, ki direktno primerja kar vrednosti polinomov in se izogne računanju stopnje. Vidimo, da druga možnost ni le prostorsko, temveč tudi časovno ugodnejša. V tabelah imamo dva stolpca z imenom "Število ciklov", in tako tudi dva stolpca s skupnim časom. V prvem stolpcu je upoštevana maksimalna zgornja meja $3m + 3$. Vrednosti v stolpcu "Število ciklov 2" pa so dobljene v simulacijah kot povprečno število ciklov za izračun inverza na 5000 vhodnih vektorjih. Če opazujemo povprečno število obhodov *while* zanke, sta modula enakovredna.

m	FF	LUT-i	Rezine	u.p. [ns]	Število ciklov	Skupen čas [ns]	Število ciklov 2	Skupen čas 2 [ns]
11	57	85	45	2.654	36	95.544	25	66.35
63	269	497	171	4.129	192	792.768	155	639.995
113	471	889	397	4.986	342	1705.212	279	1391.094
191	784	1500	593	7.168	576	4128.768	475	3404.8

Tabela 5.37: MAIA z računanjem stopnje - rezultati implementacije

m	FF	LUT-i	Rezine	u.p. [ns]	Število ciklov	Skupen čas [ns]	Število ciklov 2	Skupen čas 2 [ns]
11	49	56	17	2.564	36	92.304	24	61.536
63	257	302	111	4.144	192	795.648	155	642.32
113	457	538	191	4.943	342	1690.506	279	1379.097
191	769	905	280	5.891	576	3393.216	475	2798.225

Tabela 5.38: MAIA s primerjanjem vrednosti polinomov - rezultati implementacije

Primerjava rezultatov implementiranih modulov za invertiranje

Vsa opisana vezja so sekvenčna, zato imajo tudi pričakovano majhno prostorsko zahtevnost. Razlike med moduli glede porabe elementov so občutno manjše kot razlike v skupnem času, potrebnem za izračun inverza. Kot najugodnejši se izkaže modul 5.4.1, tj. razširjen Evklidov algoritem.

5.5 Eliptične krivulje nad GF(2^m)

Pri implementaciji modulov za seštevanje, podvojevanje in množenje točk eliptičnih krivulj s skalarjem smo naleteli na omejitve. Eliptična krivulja $E : y^2 + xy = x^3 + ax^2 + b$ je podana s parametroma a in b iz GF(2^m), ki sta m-bitna vektorja. Tako bi pri seštevanju točk $R = P + Q$, pri čemer so točke podane z dvema koordinatama, npr. $P = (x_1, y_1)$, potrebovali še 6 m-bitnih vhodno/izhodnih vektorjev. Ker ima izbran FPGA premalo IOB-jev, bomo ta problem morali reštevati tako, da bomo podatke sprejemali zaporedno; najprej parametra a in b, nato koordinati prve točke in nato koordinati druge točke, na koncu pa še kontrolni signal, da lahko FPGA začne z delom. To razreši problem pomanjkanja IOB do razsežnosti končnega obsega $m = 113$, vendar pa pri $m = 191$ težave ostanejo. Zato tudi koordinati izhodne točke vračamo zaporedno.

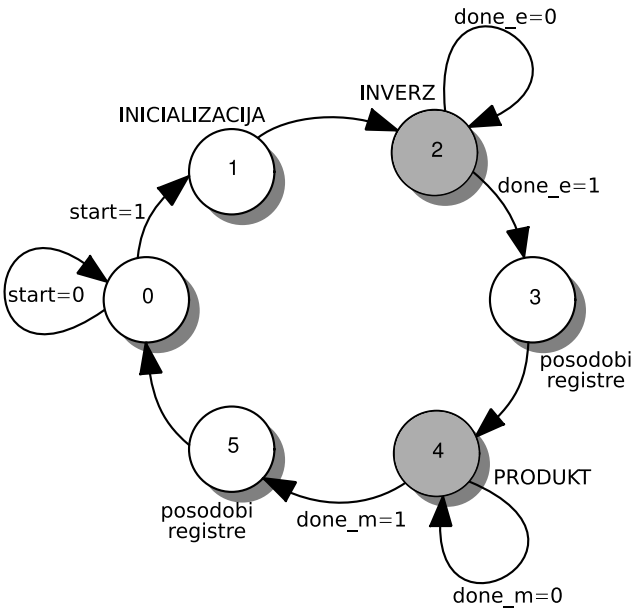
5.5.1 Seštevanje in podvojevanje točk

Najprej si bomo ogledali implementaciji seštevanja in podvojevanja točk z izračunom smernega koeficienta λ sekante (za seštevanje točk po formuli (3.8)) oz. tangente (za podvojevanje točke po formuli (3.9)). Za izračun λ potrebujemo deljenje, ki ga realiziramo v modulu *deli*. Kot smo že povedali, deljenje nadomestimo z množenjem inverznega elementa, torej računamo $y \cdot x^{-1}$ namesto y/x. Modul *deli* zato povezuje modul *eea* (razširjen Evklidov algoritem 5.4.1) za računanje inverza, ter modul *mont_hybrid* (hibridni Montgomeryjev množilnik 5.1.5) za računanje produkta. KA, ki nadzira delovanje modula *deli*, je prikazan na sliki 5.13. Za izračun kvocienta potrebuje $(10 + 2m + S)$ urinih period. Modul *deli* izvaja računanje inverza x^{-1} v stanju 2, računanje produkta $y \cdot x^{-1}$ pa v stanju 4.

Seštevanje točk

Vsoto točk $P = (x_1, y_x)$ in $Q = (x_2, y_2) \in E$ izračuna modul *addP*, ki kot vhodne parametre prejme koordinate obeh točk, tj. štiri vektorje dolžine m, in kontrolne signale (`clk`, `reset` in `start`), na izhod pa da koordinati vsote $R(x_3, y_3)$, izračunane po formuli (3.8). Izhodni signal `done` pomeni, da je na izhodu veljavna vsota.

Formule za izračun vsote (3.8) narekujejo pot računanja "vmesnih rezultatov", ki je prikazana na shemi 5.14. Le-ta služi kot osnova za implementacijo modula *addP*. Vidimo, da bomo potrebovali modul *deli* (na shemi označen **D**) za izračun λ, modul *squarer* (klasično kvadriranje 5.2.1, na shemi označeno **SQ**) za izračun λ^2 in modul *mont_hybrid* (Hibridni Montgomeryjev množilnik 5.1.5, na shemi označen z **M**) za izračun produkta $\lambda(x_1 + x_3)$. Črtkane črte na shemi nakazujejo, kako lahko posamezne korake računanja razbijemo na stanja KA, ki je prikazan na sliki 5.15.

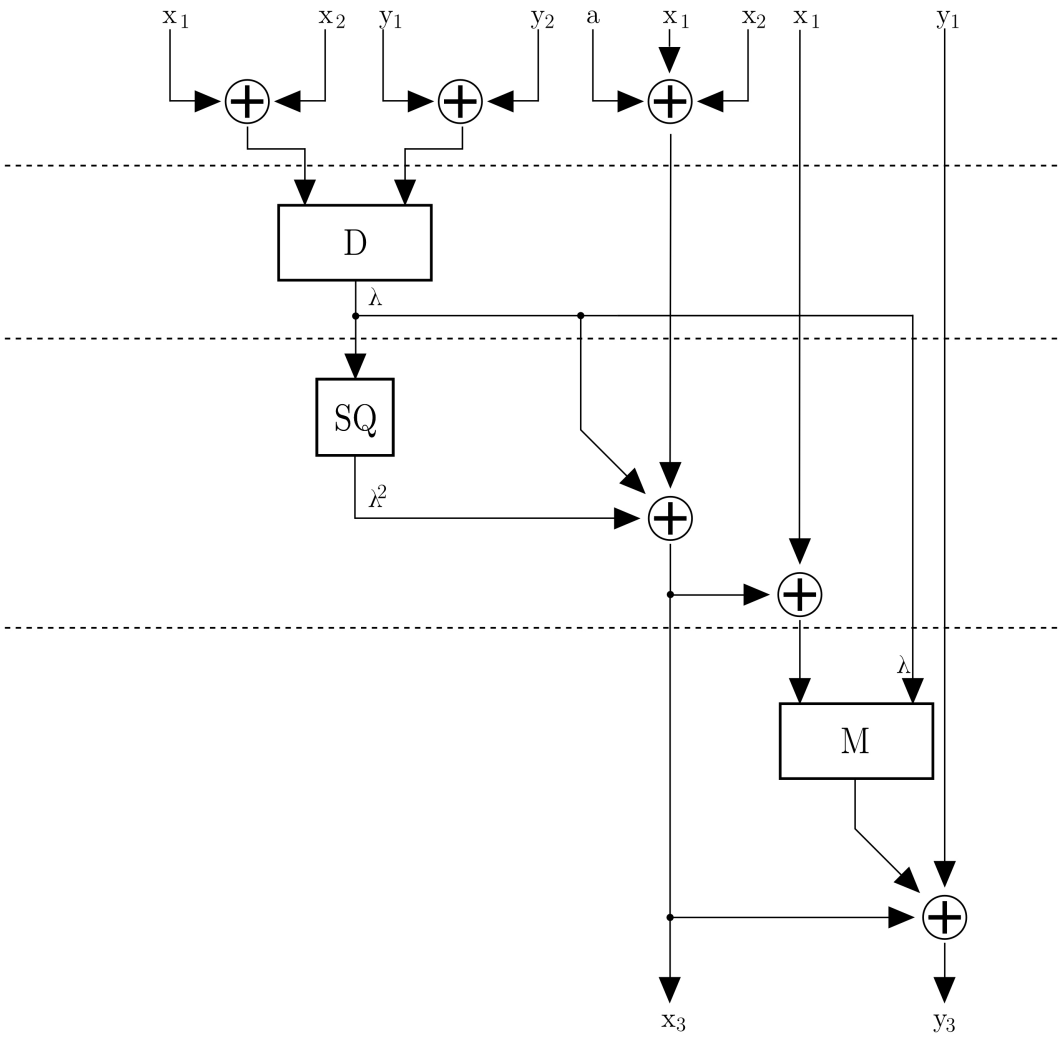


Slika 5.13: Modul *deli* - prehajanje stanj KA

Računanje koeficienta λ izvaja v stanju 3, kvadriranje in izračun koordinate x_3 vsote pa v stanju 4, kjer KA tudi pripravi vhodne vektorje za množenje. Le-to se izvaja v stanju 5. Ko je produkt izračunan, izvede še zadnje seštevanje v stanju 6.
Potrebujemo še nek mehanizem, ki preveri, ali seštevamo enaki ali nasprotni točki; v obeh primerih je njuna vsota točka v neskončnosti. Če KA ugotovi, da sta x-koordinati obeh točk enaki, preide v stanje 7, kjer postavi izhodne vektorje na vrednost 0, nato pa se vrne v stanje 0. Vsota je na voljo po $(19 + 2m + 2S)$ urinih periodah. V poglavju 5.1.5 smo pri vseh vrednostih parametra m ugotovili, da so najučinkovitejši tisti Montgomeryjevi hibridni množilniki, pri katerih je parameter S enak 2 (glej tabelo 5.21), torej potrebujemo za izračun vsote skupaj $(23 + 2m)$ ciklov.

Podvojevanje točke

Modul *doubleP* je sestavljen zelo podobno kot modul *addP*. Najprej izračuna smerni koeficient λ, nato pa sledi računanje x-koordinate podvojene točke. Pri tem potrebujemo modul za kvadriranje. V formulah (3.9) vidimo, da bomo kvadrat računali dvakrat. Če kvadrat x_1^2 računamo istočasno kot produkt λx_3, potrebujemo le en modul za kvadriranje *squarer*, ki ga povežemo tako, da preko multipleksorja spreminjamo njegov vhod. Tako bomo na vhod modula *squarer* najprej pripeljali koeficient λ, nato pa dve urini periodi kasneje še koordinato x_1. Modul *doubleP*

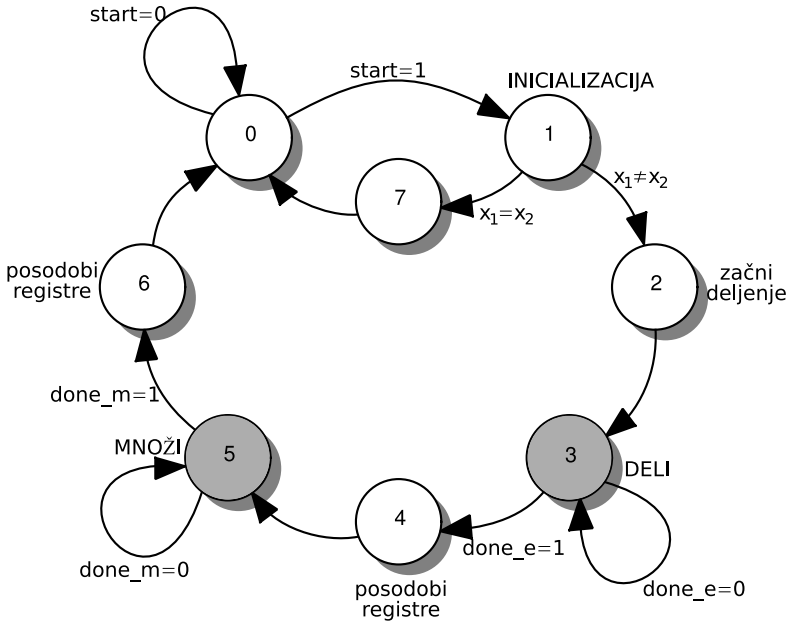


Slika 5.14: Pot računanja vsote točk (x_1, y_x) in (x_2, y_2)

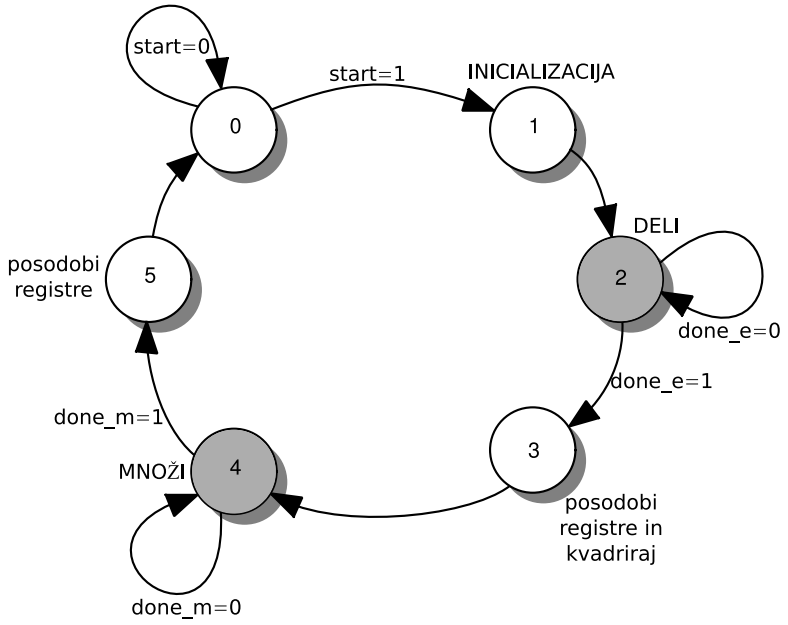
je torej sestavljen iz modula *deli*, modula *squarer* in množilnika *mont_hybrid*, vmesne rezultate pa hrani v registrih. Delovanje posameznih modulov in vpis v registre vodi KA s šestimi stanji, ki ga vidimo na sliki 5.16.

Podvojena točka je na voljo po $(18 + 2m + 2S)$ urinih periodah.

Povejmo še, da bi z nekoliko drugačno realizacijo lahko prihranili 3 urine periode: namesto da uporabimo modul *deli*, povežemo modula *eea* in *mont_hybrid* direktno v modul *doubleP*. Vendar pa so 3 urine periode zanemarljive v primerjavi z visokim številom urinih period, ki jih potrebuje modul *eea* za izračun inverza. V poglavju 4.5 smo povedali, da se lahko z uporabo projektivnih koordinat izognemu računanju inverza; le-ta je potreben le na koncu, pri prehodu s projektivnih na običajne koordinate. Poglejmo, kako se spremenijo moduli za seštevanje in podvojevanje točk



Slika 5.15: Seštevanje točk eliptične krivulje - prehajanje stanj KA



Slika 5.16: Podvojevanje točke eliptične krivulje - prehajanje stanj KA

pri uporabi projektivnih koordinat.

Rezultati implementacije

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	298	338	147	2.573	45	115.785
63	1607	3112	1119	4.807	149	716.243
113	2700	8616	2985	5.213	249	1298.037
191	4809	20919	9489	8.016	405	3246.480

Tabela 5.39: Seštevanje točk eliptične krivulje - modul *addP* - rezultati implementacije (vsota je na voljo po $23 + 2m$ urinih periodah)

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	293	346	157	3.391	44	149.204
63	1603	2757	1006	4.553	148	901.494
113	2614	12113	4285	6.423	248	1592.904
191	3956	19517	7050	7.213	404	2914.052

Tabela 5.40: Podvojevanje točke eliptične krivulje - modul *doubleP* - rezultati implementacije (podvojena točka je na voljo po $22 + 2m$ urinih periodah)

5.5.2 Standardne projektivne koordinate

V poglavju 4.5.1 smo povedali kako se spremenijo krivulja ter formule za izračun vsote in podvojene točke pri prehodu na standardne projektivne koordinate. Če sedaj formule za izračun vsote (4.20) in za podvojevanje točke (4.21) zapišemo nekoliko drugače, dobimo "recept" za implementacijo obeh operacij.

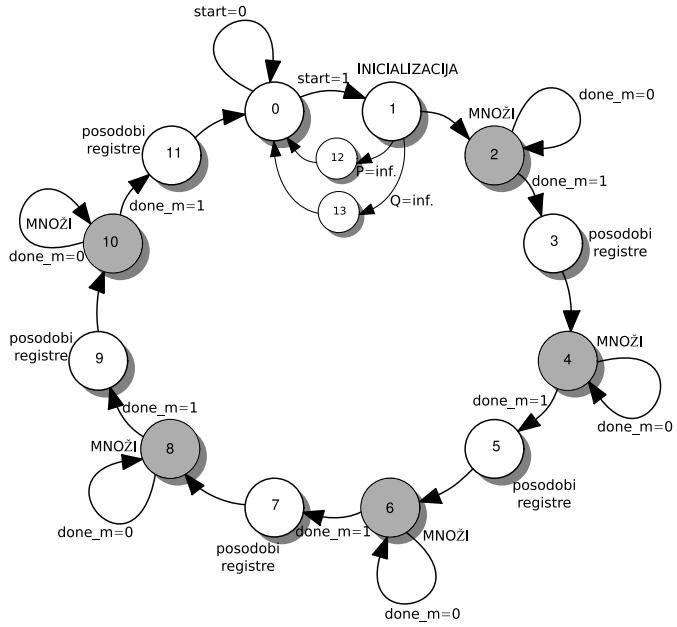
Seštevanje točk

Formule (4.20) za seštevanje točk $P = (X_1 : Y_1 : Z_1)$ in $Q = (X_2 : Y_2 : Z_2)$ bomo prepisali takole:

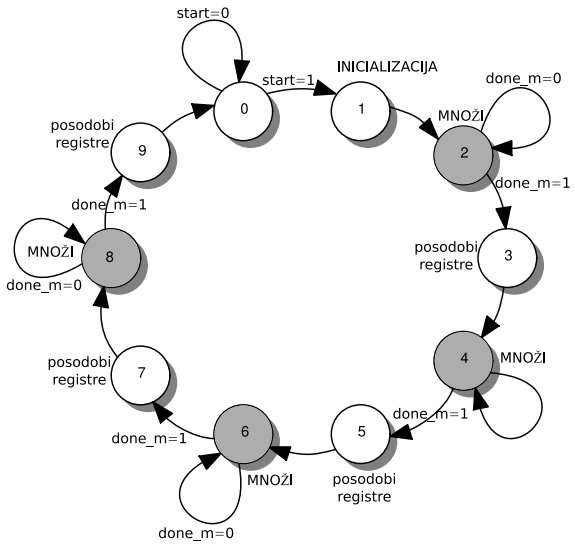
$$\begin{gathered} A = Y_1Z_2 + Z_1Y_2 \qquad B = X_1Z_2 + Z_1X_2 \\ C = Z_1Z_2 \quad D = AX_1 + BY_1 \qquad Z_3 = B^3C \\ E_1 = A^2 + AB + aB^2 \qquad E = E_1C + B^3 \\ X_3 = BE = BCE_1 + B^4 \\ Y_3 = B^2Z_2D + (A+B)CE_1 + (A+B)B^3 \end{gathered} \tag{5.1}$$

Če sledimo enačbam, kot so zapisane v formuli (5.1), dobimo shemo za realizacijo, ki je prikazana na sliki 5.19. Črtkane črte na sliki prikazujejo delitev posameznih izračunov na stopnje, ki se izvajajo v zaporednih stanjih KA. Pri tem moramo opozoriti še na to, da je med dvema stopnjama na shemi 5.19 vedno vmes še eno stanje KA, v katerem se delni rezultati vpišejo v

registre, pripravijo pa se tudi novi vhodi za kvadrat in množenja, ki se bodo izračunala v naslednji stopnji sheme oz. naslednjem stanju KA. Vidimo, da lahko vse operacije, ki so potrebne za vsoto po formulah (5.1), izvedemo s štirimi vzporedno vezanimi množilniki in enim modulom za kvadriranje. Prvo množenje, oz. prva štiri množenja, izvedemo v stanju 2 KA, ki ga vidimo na sliki 5.17. Dobljene produkte seštejemo in vsoti shranimo v registra `A` in `B`. V stanju 4 ponovno uporabimo vse štiri množilnike, izračunamo pa tudi že prvi kvadrat, tj. vrednost B^2. S tem, ko smo zamaknili računanje produkta $C = Z_1 Z_2$ eno stopnjo nižje na shemi 5.19, smo se izognili uporabi petih množilnikov. V stanju 6 sledi računanje kvadrata in produktov, ki jih potrebujemo za izračun vrednosti E_1, ter produktov BC, $(A + B)C$ in $B^3 = BB^2$. Postopka ne bomo opisali do konca, saj je vrstni red lepo razviden iz same sheme. Povejmo le še to, da so nekateri vmesni izračuni, čeprav imamo na voljo že vse potrebne vhodne vektorje, zamaknjeni v kasnejše stopnje sheme zato, ker jih še ne potrebujemo za noben drug izračun, in ker tako zmanjšamo porabo modulov gradnikov in s tem seveda tudi število porabljenih logičnih elementov.
Vsota je izračunana v $(23 + 5S)$ urinih periodah. Na sliki 5.17 vidimo še dodatni stanji 12 in 13. V primeru, da je točka $P = (0 : 1 : 0)$, tj. točka v neskončnosti, KA preide v stanje 12 in postavi na izhodne signale koordinate točke Q. Prav tako v primeru, ko je $Q = (0 : 1 : 0)$, KA preide v stanje 13, za vsoto pa vzame točko P.



Slika 5.17: Seštevanje točk, podanih v standardnih projektivnih koordinatah - prehajanje stanj KA



Slika 5.18: Podvojevanje točke v standardnih projektivnih koordinatah - shema prehajanja stanj KA

Podvojevanje točke

Če sledimo enačbam (4.21), dobimo pot računanja, ki je prikazana na sliki 5.20.

Na shemi vidimo, da potrebujemo za podvojevanje točke v standarnih projektivnih koordinatah 8 množenj, ki jih zaporedno izvajamo na treh množilnikih (v modul *SPdouble* vzporedno povežemo tri module *mont_hybrid*) in štiri kvadriranja, ki jih izvajamo na dveh modulih *squarer*. Delovanje modulov in vpis njihovih rezultatov v registre vodi KA z 10 stanji, ki ga lahko vidimo na sliki 5.18. Podvojena točka je tako izračunana v $(19 + 4S)$ urinih periodah.

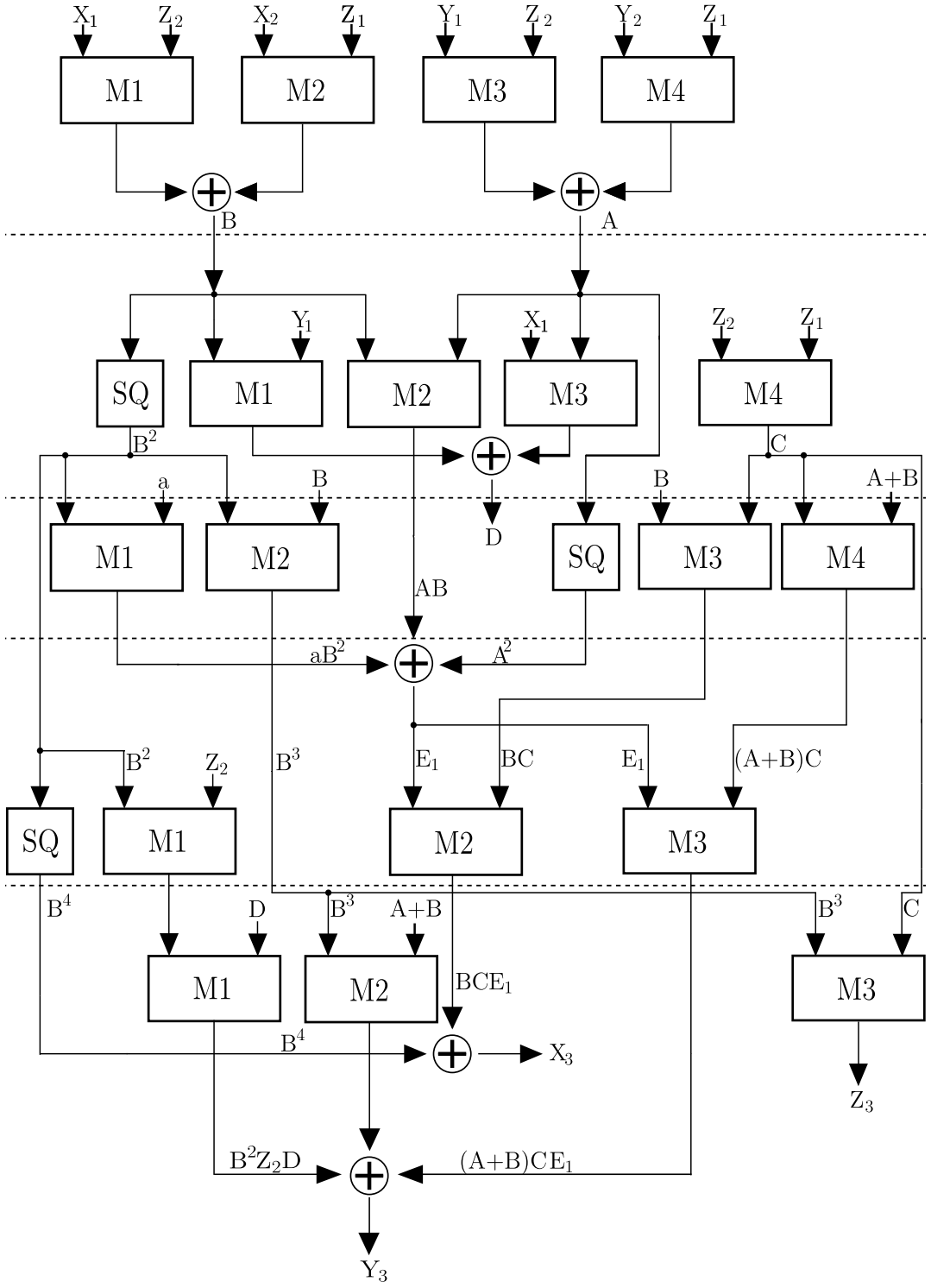
Po končanem računanju je potreben prehod na običajne koordinate. V ta namen smo napisali modul *SPtoA*, ki projektivno točko $(X : Y : Z)$ pretvori v točko $(X/Z, Y/Z)$. Modul *SPtoA* sestavljajo modul *eea* za računanje inverza in dva modula *mont_hybrid*, ki sta vezana vzporedno. Preverjanje, ali je $Z = 0$, ni potrebno, saj bo modul *eea* v tem primeru vrnil "inverz" $0 \ldots 0$, in tako bomo po množenju dobili točko $(0 \ldots 0, 0 \ldots 0)$, ki jo razumemo kot točko v neskončnosti. Prehod na običajne koordinate traja $10 + 2m + S$ ciklov.

Rezultate implementacije modulov *SPaddP* in *SPdoubleP* brez modula *SPtoA*, torej brez prehoda na običajne koordinate, lahko vidimo v tabelah 5.41 in 5.42.

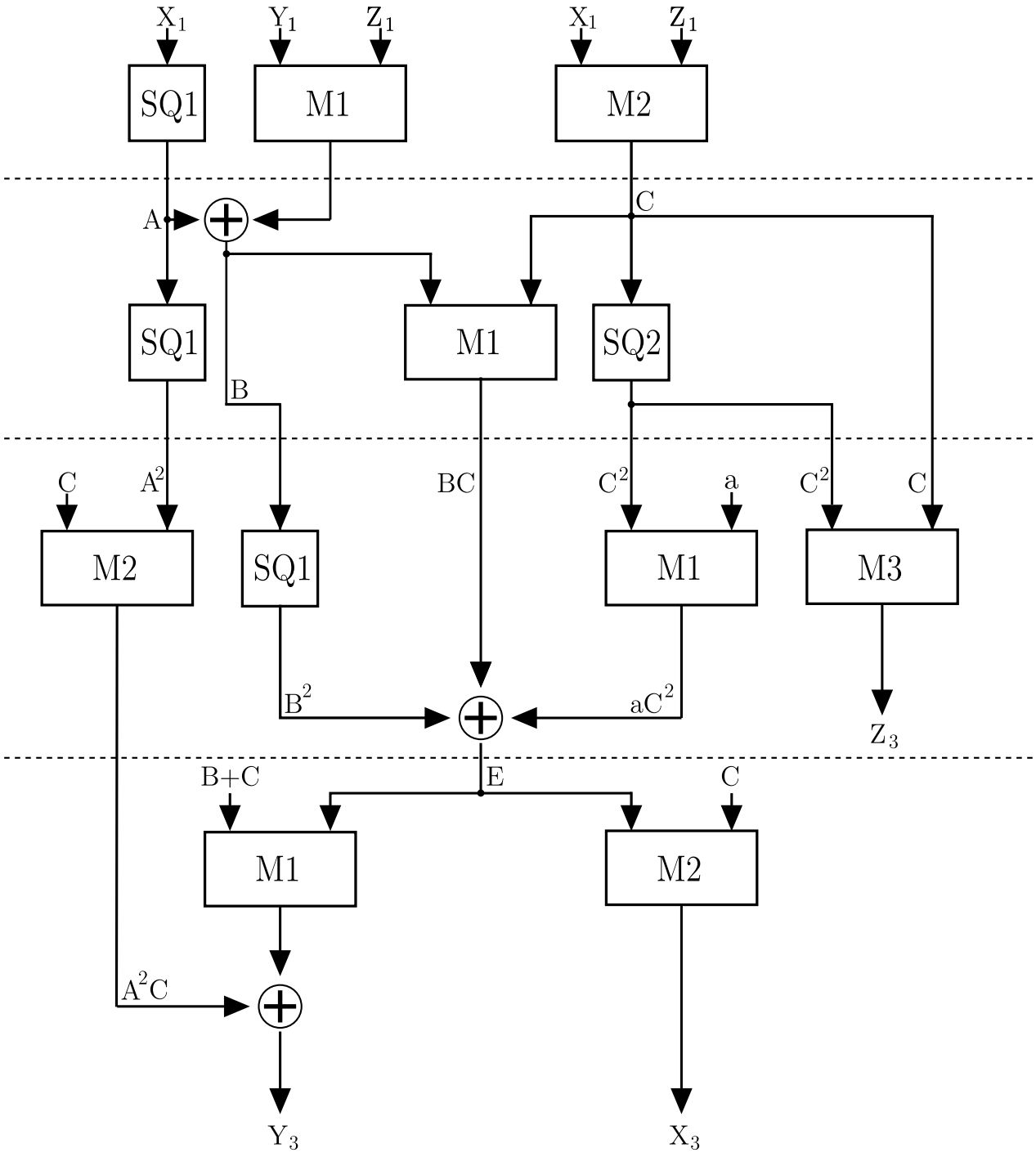
5.5.3 Projektivne koordinate Lòpez-Dahab

Seštevanje točk

Pot računanja vsote, ki jo narekujejo enačbe 4.23, je prikazana na shemi 5.22. Modul *LDaddPZ* je sestavljen iz treh vzporedno vezanih množilnikov in dveh modulov za kvadriranje. Shema



Slika 5.19: Seštevanje točk, podanih v standardnih projektivnih koordinatah



Slika 5.20: Podvojevanje točke v standardnih projektivnih koordinatah

prehajanja stanj KA, ki vodi delovanje modula *LDaddPZ*, je enaka shemi prehajanja stanj KA modula *SPaddP* za seštevanje v standardnih projektivnih koordinatah (glej sliko 5.18). S tem

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	408	638	245	2.519	33	83.127
63	3013	5943	2095	5.538	33	182.754
113	5006	17880	6625	6.704	33	221.232
191	8037	41155	19297	8.354	33	275.682

Tabela 5.41: Seštevanje točk eliptične krivulje v standardnih projektivnih koordinatah - modul *SPaddP* - rezultati implementacije (vsota je na voljo po 33 urinih periodah)

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	458	388	236	3.871	27	104.517
63	2031	4039	1452	4.662	27	125.874
113	3102	12732	4884	6.821	27	184.167
191	5153	29908	11960	8.334	27	255.018

Tabela 5.42: Podvojevanje točke eliptične krivulje v standardnih projektivnih koordinatah - modul *SPdoubleP* - rezultati implementacije (rezultat je na voljo po 27 urinih periodah)

je tudi število urinih period, ki jih za izračun vsote potrebuje modul *LDaddPZ*, enako kot pri modulu *SPaddP*, tj. $18 + 5S$.

Podvojevanje točke

Formule (4.24) za izračun točke $2P = R = (X_3 : Y_3 : Z_3)$, kjer je $P = (X_1 : Y_1 : Z_1)$, preuredimo takole:

$$\begin{array}{cc} A = Z_1^2 & B = bZ_1^4 \qquad C = X_1^2 \\ Z_3 = X_1 2Z_1^2 & X_3 = X_1^4 + bZ_1^4 \\ E_1 = Y_1^2 + aZ_3 + bZ_1^4 & E_2 = Z_3 bZ_1^4 \\ \multicolumn{2}{c}{Y_3 = E_1 X_3 + E_2} \end{array} \tag{5.2}$$

Po preurejenih formulah sedaj zapišemo shemo za izračun podvojene točke (slika 5.23). Vidimo, da bomo za realizacijo modula *LDdoubleP* potrebovali dva množilnika in dva modula za kvadriranje, kar je en množilnik manj kot pri modulu *SPdoubleP* za podvojevanje točke v standardnih projektivnih koordinatah. Prav tako je iz sheme razvidno, da bo modul *LDdoubleP* potreboval manj ciklov kot modul *SPdoubleP*, saj prva stopnja računanja ne vključuje množenja, temveč le dva kvadrata. Na sliki 5.21, ki prikazuje prehajanje stanj KA, ki vodi delovanje modula *LDdoubleP*, vidimo, da množimo le v stanjih 3, 5 in 7. Podvojena točka je torej izračunana že po $16 + 3S$ urinih periodah.

Ponovno potrebujemo tudi modul za pretvorbo projektivnih koordinat v običajne koordinate $(X/Z, Y/Z^2)$, za kar potrebujemo en inverz, eno kvadriranje in dve množenji.

Rezultati implementacije

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	660	499	369	3.165	33	104.445
63	2602	4497	1679	5.612	33	185.196
113	3950	13670	4793	6.872	33	226.776
191	6590	30968	13049	8.192	33	270.336

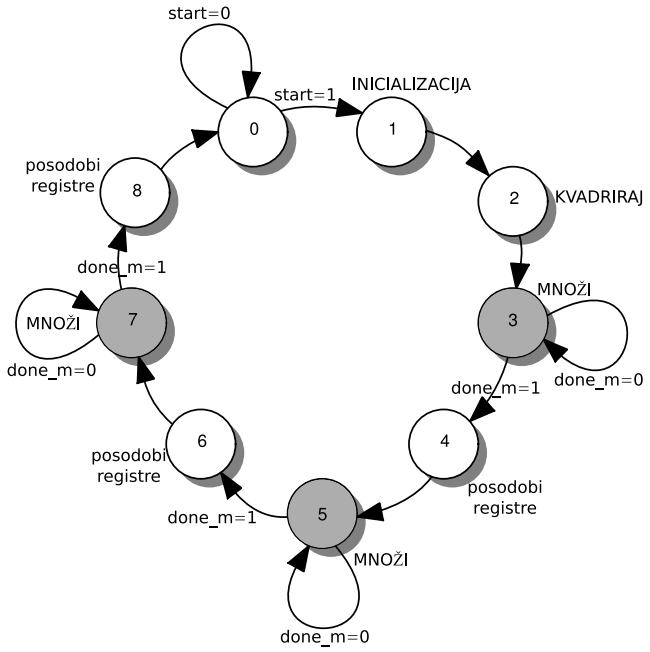
Tabela 5.43: Seštevanje točk eliptične krivulje v projektivnih koordinatah Lòpez-Dahab - modul *LDaddPZ* - rezultati implementacije (vsota je na voljo po 33 urinih periodah)

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [ns]
11	212	268	115	2.423	22	53.306
63	1230	2676	1010	4.118	22	90.596
113	2462	8403	3233	5.747	22	126.434
191	3768	19435	6863	6.539	22	143.858

Tabela 5.44: Podvojevanje točke eliptične krivulje v projektivnih koordinatah Lòpez-Dahab - modul *LDdoubleP* - rezultati implementacije (rezultat je na voljo po 22 urinih periodah)

5.5.4 Množenje točk na eliptični krivulji

Algoritem 12 za množenje točke s skalarjem je zelo podoben algoritmu "kvadriraj in množi" za potenciranje (algoritem 7 v poglavju 4.3), le da sedaj množenje nadomestimo s seštevanjem in kvadriranje s podvojevanjem točke. Modul *mult_kP*, ki izvaja množenje točke s skalarjem (pri tem uporabi modula *addP* in *doubleP*, ki zahtevata računanje smernega koeficienta λ), je zato zelo podoben modulu za potenciranje "kvadriraj in množi I" (glej shemo prehajanja stanj KA 5.9 v razdelku 5.3.1). Realiziran je kot KA s petimi stanji in petimi registri, ki hranijo vrednost skalarja k (pomični register), koordinati vsote $Q + P$, ter koordinati podvojene točke $2P$. V vsakem obhodu zanke v algoritmu 12 se izvede podvojevanje točke in če je pripadajoči bit skalarja k enak 1, še seštevanje. To pa hkrati pomeni, da je število ciklov, potrebnih za izračun skalarnega produkta, odvisno od števila seštevanj, ki se izvedejo. Naj bo $\ell < m$ število bitov skalarja k, ki so enaki ena. Tedaj je število ciklov, ki jih potrebuje modul *mult_kP* za izračun prdukta kP, enako: $3 + \ell(19 + 2m + 2S) + (m - \ell)(18 + 2m + 2S)$, kar pri vrednosti $S = 2$ znaša $3 + \ell + m(22 + 2m)$ urinih period.



Slika 5.21: Podvojevanje točke v projektivnih koordinatah Lòpez-Dahab - shema prehajanja stanj KA

Rezultati implementacije

V naslednjih tabelah je dolžina cikla navedena v ns, skupen čas pa v μs.

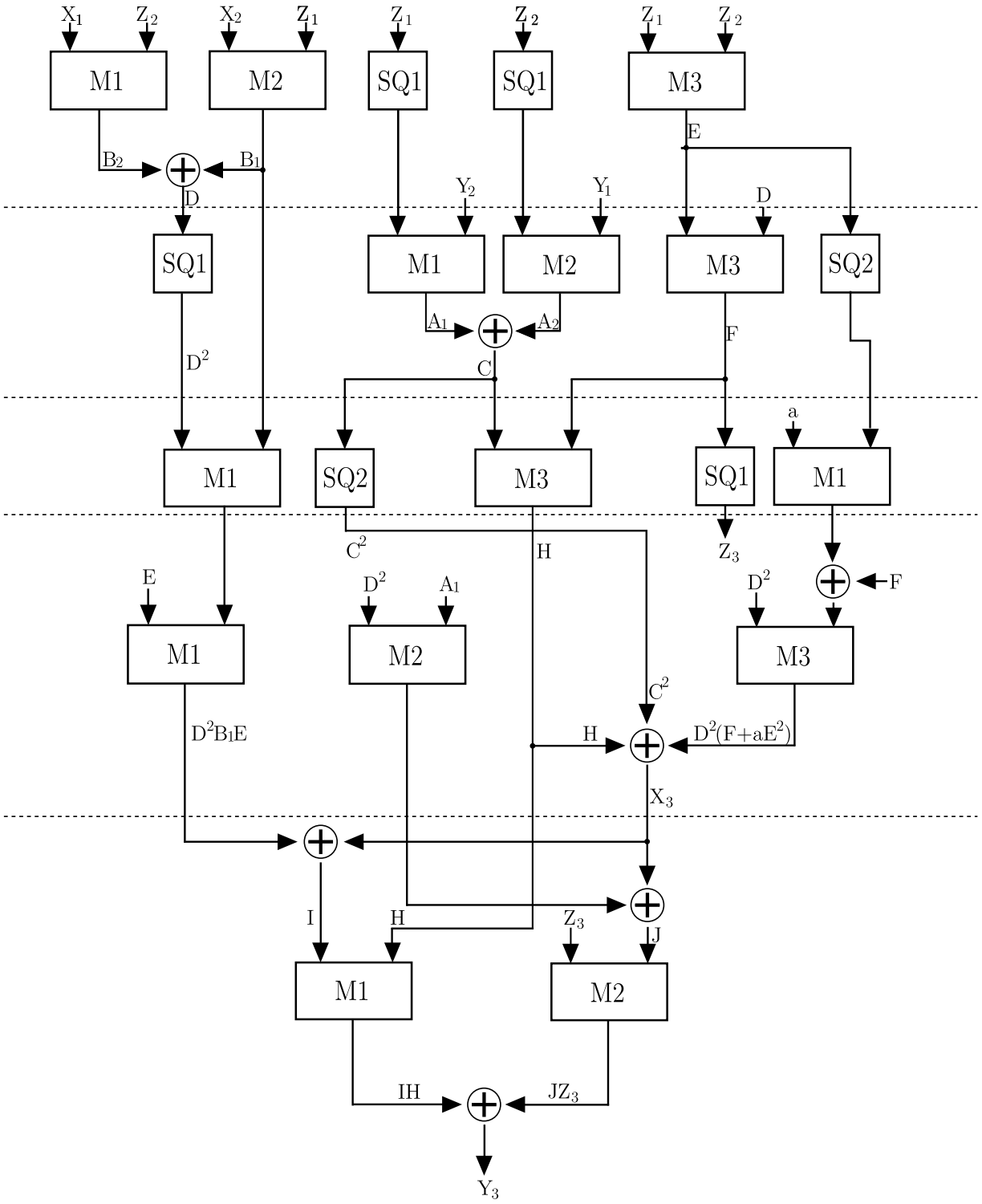
m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [μs]
11	6706	796	384	4.334	498	2.158
63	3346	5960	2199	4.717	9390	44.293
113	5645	17860	6222	6.275	28140	176.579
191	8835	40966	15795	8.263	77358	639.209

Tabela 5.45: Množenje točke s skalarjem - modul *mult_kP* - rezultati implementacije (produkt kP je na voljo po $3 + \ell + m(22 + 2m)$ urinih periodah)

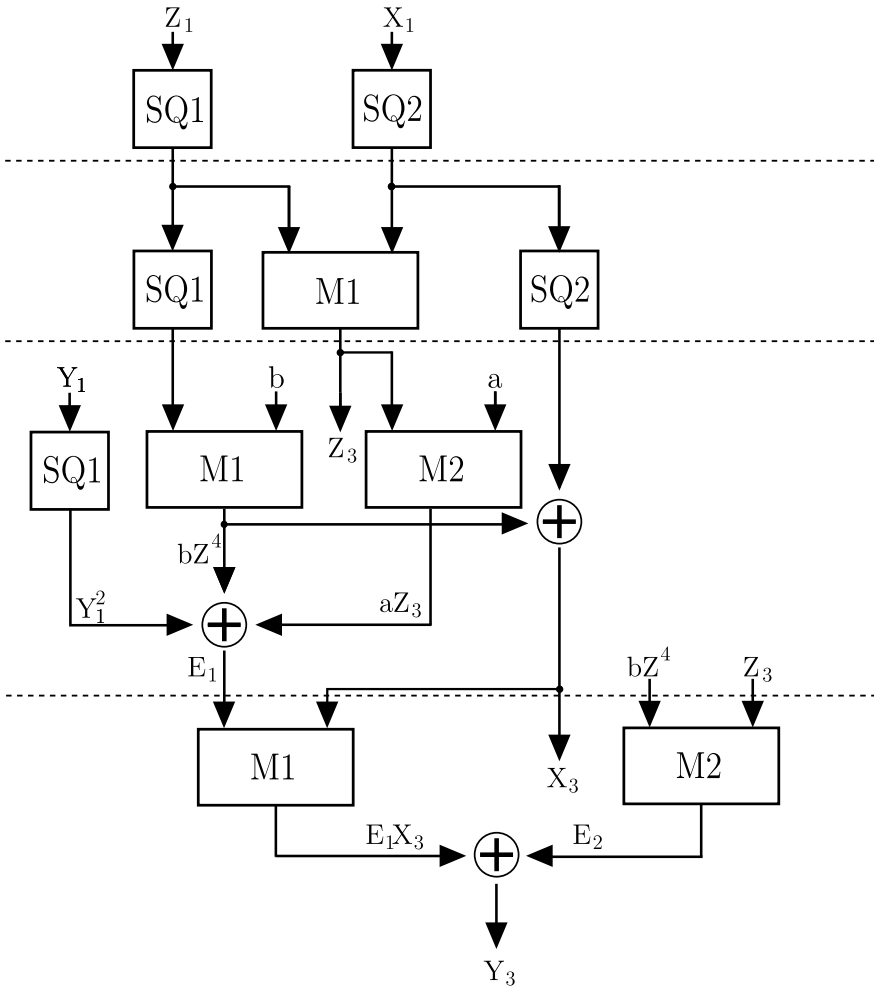
Kot smo pričakovali, je časovna zahtevnost modula *mult_kP* zelo visoka. Oglejmo si sedaj, kako se rezultati izboljšajo pri uporabi projektivnih koordinat.

Množenje točke s skalarjem v standardnih projektivnih koordinatah

Omenili smo, da je pri uporabi projektivnih koordinat na koncu potreben še prehod na običajne koordinate. Le-ta zahteva, da v modul za izračun produkta kP za vzporedno povezanima moduloma *SPaddP* in *SPdoubleP* povežemo še modul *SPtoA* za pretvorbo, in da v KA dodamo



Slika 5.22: Seštevanje točk, podanih v projektivnih koordinatah Lòpez-Dahab



Slika 5.23: Podvojevanje točke, podane v projektivnih koordinatah Lòpez-Dahab

še stanji 5 in 6. Modul *SPmult_kP* potrebuje tudi dva dodatna registra za z-koordinati vsote in podvojene točke. Večkratnik kP je na voljo po $16 + 6\ell + 29m$ urinih periodah (modul za pretvorbo *SPtoA* potrebuje $10 + S + 2m$ urinih period).

Rezultati implementacije

Rezultati so podani za časovno najzahtevnejši primer, ko je $\ell = m$, in izračun produkta traja $16 + 35m$ ciklov.

m	FF	LUT-i	Rezine	u.p.[ns]	Št. ciklov	Skupen čas [μs]
11	1465	1409	775	3.401	401	1.364
63	7012	12782	5128	7.441	2221	16.527
113	11256	39596	16292	7.797	3971	30.962
191	16784	89337	30546	8.515	6701	57.059

Tabela 5.46: Množenje točke s skalarjem v standardnih projektivnih koordinatah - modul *SPmult_kP* - rezultati implementacije (večkratnik kP je pri $\ell = m$ na voljo po $16 + 35m$ urinih periodah)

Kot smo pričakovali, je časovna zahtevnost modula *mult_kP* zelo visoka. Oglejmo si sedaj, kako se rezultati izboljšajo pri uporabi projektivnih koordinat.

Množenje točke s skalarjem v projektivnih koordinatah Lòpez-Dahab

Modul *LDmult_kP*, ki za izračun produkta kP uporablja projektivne koordinate Lòpez-Dahab, je zgrajen enako kot modul *SPmult_kP*, le da sedaj povezuje module *LDaddPZ*, *LDdoubleP* in *LDtoA*. Večkratnik kP je na voljo po $17 + 11\ell + 24m$ urinih periodah. Primerjava rezultatov implementacije modulov za seštevanje in podvojevanje (tabele 5.41, 5.42, 5.47 in 5.44) pove, da je uporaba projektivnih koordinat Lòpez-Dahab tako časovno (podvojevanje) kot tudi prostorsko (seštevanje) ugodnejša, zato pričakujemo boljše rezultate tudi pri množenju s skalarjem.

Rezultati implementacije

Rezultati so podani za časovno najzahtevnejši primer, ko je $\ell = m$, in izračun produkta traja $17 + 35m$ ciklov.

m	FF	LUT-i	Rezine	u.p.[μs]	Št. ciklov	Sk. čas [μs]
11	1239	1285	687	3.682	368	1.355
63	5982	10366	4366	5.673	2222	12.605
113	9696	31506	13196	7.550	3972	29.989
191	14350	70410	25912	8.476	6702	56.806

Tabela 5.47: Množenje točke s skalarjem v projektivnih koordinatah Lòpez-Dahab - modul *LDmult_kP* - rezultati implementacije (večkratnik kP je pri $\ell = m$ na voljo po $17 + 35m$ urinih periodah)

Rezultati kažejo, da sta oba modula, ki uporabljata projektivne koordinate, neprimerljivo hitrejša od modula *mult_kP*, čeprav porabita skoraj dvakrat toliko logičnih elementov. V najslabšem

primeru, ko je $\ell = m$, se modula *SPmult_kP* in *LDmult_kP* obnašata zelo podobno. Kljub temu vidimo, da je modul *SPmult_kP*, tako prostorsko kot tudi časovno, zahtevnejši od modula *LDmult_kP*. Povprečno število potrebnih seštevanj je $\ell = \frac{m-1}{2}$; glej [4]. V tabeli 5.48 lahko vidimo, koliko modul *LDmult_kP* pridobi zaradi enostavnejšega podvojevanja.

		Stdandardne projektivne koord.			Projektivne koord. Lòpez-Dahab		
m	ℓ	u.p.[ns]	Št. ciklov	Sk. čas [μs]	u.p.[ns]	Št. ciklov	Sk. čas [μs]
11	5	365	3.401	1.241	336	3.682	1.237
63	31	2029	7.441	15.098	1870	5.637	10.609
113	56	3629	7.797	28.295	3345	7.550	25.255
191	95	6125	8.515	52.154	5646	8.476	47.856

Tabela 5.48: Primerjava modulov za množenje točke s skalarjem v standardnih projektivnih koordinatah in projektivnih koordinatah Lòpez-Dahab pri povprečni vrednosti $\ell = \frac{m-1}{2}$

Poglavje 6

Testiranje pravilnosti

Pravilnost delovanja modulov smo preverili s simulacijami v orodju ModelSim. V ta namen smo spisali testne module, ki so preverjali pravilnost oz. enakost rezultatov na velikem številu naključno generiranih vhodnih vektorjev.

6.1 Množenje

6.1.1 Testiranje redukcije

Prvi tak testni modul je modul je modul *testRed*, ki preverja pravilnost redukcije. Redukcijska modula *reducer* (redukcija s štirimi vektorj 5.1.1) in pa *reduceB* (redukcija z matriko 5.1.1), sta povezana tako, da na njuna vhoda pripeljemo enak $(2m-1)$-bitni vhodni vektor, nato pa preverimo, ali sta izhoda obeh modulov enaka.

6.1.2 Testiranje množenja

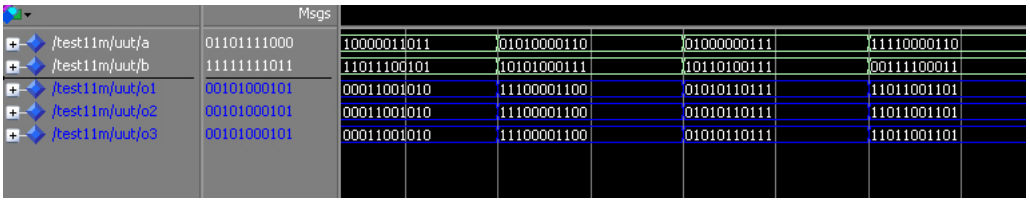
Testiranje kombinatoričnih množilnikov

Testiranje množenja smo razbili na tri dele oz. module. Prvi testira kombinatorične množilnike, drugi pa sekvenčne. Oglejmo si najprej testiranje kombinatoričnih množilnikov *Cmultiplier* (modul za klasicno množenje 5.1.2), *masMul* (modul za množenje Mastrovito 5.1.4) in pa *montgomery_comb_mul* (modul za Montgomeryevo množenje - kombinatorična različica 5.1.5). Te tri module povežemo v modulu *testMnozi*, ki ga lahko vidimo na sliki D.2 v dodatku D.2, tako, da vhoda `a` in `b` testnega modula peljemo na vhode vseh treh množilnikov, njihove izhode pa potem primerjamo med seboj. Če so vsi trije produkti enaki, postavimo izhod `x` modula *testMnozi* na 1 (glej VHDL opis 6.1).

VHDL koda 6.1: modul *testMnozi*: primerjanje produktov kombinatoričnih množilnikov

```
2  classic: Cmultiplier port map (a,b,c1);
3  mastrovito:  masMul  port map (a,b,c2);
4  montgomery: montgomery_comb_mult port map (a,b,c3);

6  test1<= c1 xor c2;
7  test2<= c1 xor c3;
8  test3<= c2 xor c3;

10 x <= '1' when (test2=zero and test1=zero and test3=zero)
11          else '0';
12 o1<=c1;
13 o2<=c2;
14 o3<=c3;
```

Teste smo izvedli na 1000 vhodnih parih a in b za posamezen $m = 11, 63, 113$ in $m = 191$. Slika 6.1 prikazuje simulacijo za $m = 11$.



Slika 6.1: Simulacija modula *testMnozi* za $m = 11$

Testiranje sekvenčnih množilnikov

Modul *testMsekv* preverja enakost delovanja sekvenčnih množilnikov, tj. modulov *interleaved_mult* (množenje s prepletanjem 5.1.3), *montgomery_mult* (sekvenčna različica Montgomeryjevega množilnika 5.1.5) ter *mont_hybrid* (hibridna različica Montgomeryjevega množilnika 5.1.5). Za primerjavo s prejšnjim testom smo dodali še kombinatorično različico Montgomeryjevega množilnika *montgomery_mult_comb*. Tokrat testov nismo izvedli le za vse štiri vrednosti parametra m, temveč še za veliko število različnih parametrov G pri istem m, in tako preverili tudi delovanje hibridnega Montgomeryjevega množilnika pri različnih parametrih.

Testiranje hibridnega Montgomeryjevega množilnika

Ta test je bil namenjen le preverjanju delovanja hibridnega Montgomeryjevega množilnika. V ta namen smo spremenili deklaracijo modula *mont_hybrid* tako, da namesto fiksnih vrednosti G (število bitov, ki jih sprocesira v enem obhodu), S (število potrebnih obhodov) in *lastG* (produkt, izračunan v zadnjem obhodu) omogoča povezavo večih

modulov *mont_hybrid* z različnimi parametri. Primer povezave modulov za $m = 11$ prikazuje koda 6.2:

VHDL koda 6.2: modul *testG*: primerjanje produktov hibridnih Montgomeryjevih množilnikov

```
1  kombM: montgomery_comb_mult port map (a,b,m1);
2  h2: mont_hybrid generic map (2,6,1) port map
3                               (a,b,clk,reset,start,m2,d2);
4  h3: mont_hybrid generic map (3,4,2) port map
5                               (a,b,clk,reset,start,m3,d3);
6  h4: mont_hybrid generic map (4,3,3) port map
7                               (a,b,clk,reset,start,m4,d4);
8  h5: mont_hybrid generic map (5,3,1) port map
9                               (a,b,clk,reset,start,m5,d5);
10 h6: mont_hybrid generic map (6,2,5) port map
11                              (a,b,clk,reset,start,m6,d6);
12 h7: mont_hybrid generic map (7,2,4) port map
13                              (a,b,clk,reset,start,m7,d7);
14 h8: mont_hybrid generic map (8,2,3) port map
15                              (a,b,clk,reset,start,m8,d8);
16 h9: mont_hybrid generic map (9,2,2) port map
17                              (a,b,clk,reset,start,m9,d9);
18 h10: mont_hybrid generic map (10,2,1) port map
19                             (a,b,clk,reset,start,m10,d10);
```

Testni modul *montG* je ponovno zasnovan tako, da preveri enakost vseh produktov in postavi izhodni signal `x=1`. Na sliki D.5 v dodatku D.2 vidimo, kako se krajša čas oz. število ciklov, potrebno za izračun produkta, z večanjem parametra G. Signali `m2` do `m10` so izračunani produkti hibridnih množilnikov, signali `d2` do `d10` so njim pripadajoči kontrolni signali `done`, produkt `m1` pa je izhod kombinatoričnega množilnika in služi kot referenca.

6.1.3 Kvadriranje

Testni modul *testSQ* povezuje vseh pet modulov za kvadriranje: *squarer* za klasično kvadriranje 5.2.1, *Isquarer* za kvadriranje s prepletanjem 5.2.2, ter module za Montgomeryjevo kvadriranje 5.2.3: sekvenčni *montg_square*, hibridni *montg_square_hybrid* in kombinatorični modul *montg_comb_square*. Posamezni moduli so ponovno povezani tako, da imajo enake vhode, njihove izhode pa primerjamo med seboj. Shematski prikaz modula *testSQ* si lahko ogledate na sliki D.3 v dodatku D.2.
Teste smo izvedli na 1000 vhodnih parih a in b za $m = 11, 63, 113$ ter 5000 parih za $m = 191$.

6.1.4 Potenciranje

Podobno kot pri prejšnjih testih smo tudi tukaj napisali testni modul *test3exp*, ki preverja enakost delovanja modulov za potenciranje po metodi "kvadriraj in množi": modul, ki uporablja klasično kvadriranje in množenje s prepletanjem 5.3.1, ter oba modula za Montgomery potenciranje (5.3.3 in 5.3.1). Vsi trije moduli dobijo na vhod enako osnovo a in exponent e, modul

test3exp pa preveri enakost njihovih izhodov. Teste smo izvedli na 1000 vhodnih parih.
Posebej zanimiv je primer, ko je eksponent enak $2^m - 2$, tj. računanje inverza. Vemo, da je $a^{-1}a = 1$, in to uporabimo kot izhodišče še enega testiranja pravilnosti potenciranja. Tokrat ne primerjamo različnih načinov potenciranja, ampak stestiramo delovanje vsakega modula za potenciranje posebej. V testnem modulu povežemo izhod modula za potenciranje na vhod modula za množenje. Drug vhod v množilnik je osnova a, pričakovan produkt potence in osnove pa mora biti enak `'0...01'`.
Delovanje četrtega modula *expKK* smo preverili le s pravkar opisanim testom, tj. le pravilno izračunan inverza za 1000 naključnih vhodov.

6.1.5 Inverz

Testni moduli za preverjanje pravilnosti delovanja modulov *eea* (razširjen Evklidov algoritem 5.4.1), *eeaBM* (Berlekampov algoritem 5.4.2), ter modula $MAIA$ (primerja stopnje polinomov) in $MAIA2$ (primerja vrednosti polinomov), ki implementirata modificiran skoraj inverzni algoritem 14, so podobni testnim modulom za iskanje inverza s potenciranjem. Rezultate zgoraj navedenih modulov za računanje inverza povežemo na vhod v množilnik, produkt inverza in vhodnega vektorja pa mora biti enak `'0...01'`. RTL shemo testnega modula za Be rlekampov algoritem si lahko ogledate na sliki D.4 v dodatku D.2. Za vsakega izmed zgoraj naštetih modulov smo naredili svoj testni modul, teste pa smo izvedli na 5000 vhodnih vektorjih za $m = 11, 63, 113$ in $m = 191$.

6.1.6 Aritmetika nad eliptičnimi krivuljami

Testiranje modulov smo izvedli nad GF(2^4). Parametra a in b ter točke krivulje smo dobili na spletni strani http://lkrv.fri.uni-lj.si/~ajurisic/tec3/gradiva/ECClassroom/Files/EC_4_3.htm, kjer smo lahko preverili tudi pravilnost izračunanih vsot in dvojnih točk. Izbrali smo tri eliptične krivulje:

- $a = \alpha^{11}, \quad b = \alpha^3$;
- $a = \alpha^{10}, \quad b = \alpha^3$;
- $a = \alpha^3, \quad b = \alpha^5$.

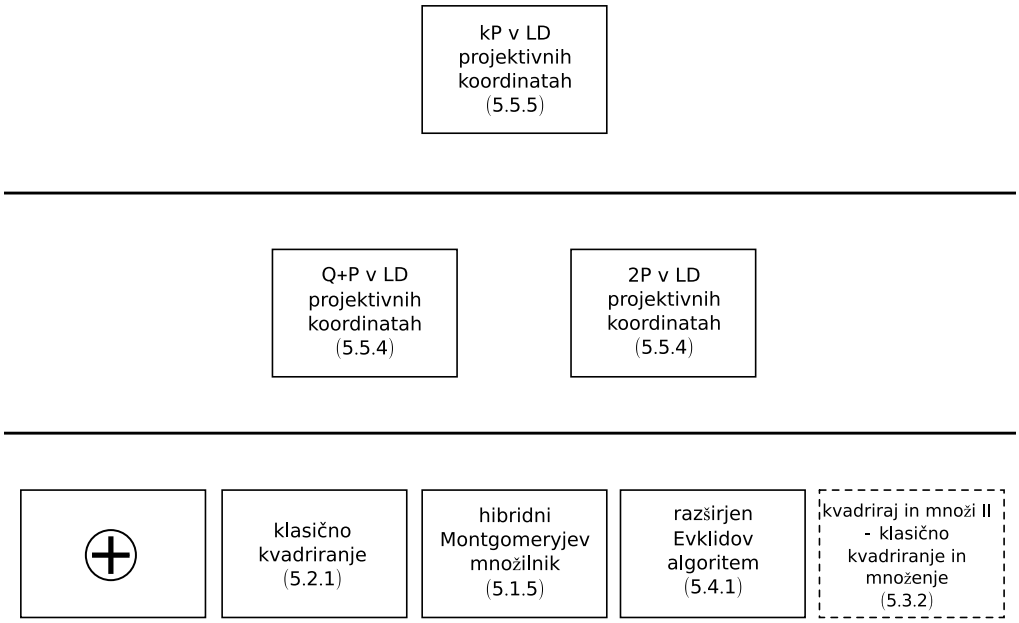
Testni moduli so zgrajeni podobno kot v prejšnjih primerih: npr. modul za preverjanje vsote povezuje vse tri module za seštevanje (*addP*, *SPaddP*, *LDaddPZ*) na tak način, da jim pošlje enake vhodne vektorje, nato pa preverja enakost njihovih rezultatov. Modula za seštevanje v projektivnih koordinatah seveda potrebujeta še modula za prehod na običajne koordinate. Podobno

smo testirali tudi množenje točke s skalarjem: testni modul povezuje vse tri "množilnike" ter primerja njihove rezultate.

Poglavje 7

Zaključek

Ogledali smo si različne algoritme za implementacijo osnovnih operacij nad elementi končnih obsegov in točkami eliptične krivulje. Te algoritme smo primerjali glede na časovno in prostorsko zahtevnost vezja, s katerim so realizirana, in izbrali tiste, ki so najprimernejši za implementacijo na FPGA. Shemo 1.1 iz uvodnega poglavja dopolnimo tako, da prikazuje izbrani dizajn (slika 7.1).



Slika 7.1: Izbrani algoritmi za implementacijo binarne aritmetike in operacij nad točkami eliptične krivulje na FPGA

Seštevanje elemetov binarnega končnega obsega izvajamo kar z operatorjem XOR, kar dodatno

olajša strojno implementacijo. Vezja za realizacijo množenja lahko v grobem razdelimo na dve skupini. V prvo spadajo množilniki, ki izračunajo en bit produkta na cikel, in zato za končni rezultat potrebujejo veliko število urinih period. Realizirajo jih sekvenčna vezja z nizko prostorsko zahtevnostjo. Vendar pa pri implementaciji na FPGA majhna poraba logičnih elemetov ni opredeljen a kot prednost, saj skušamo v kar največji meri izkoristiti paralelizem. V primeru uporabe FPGA za izdelavo prototipa ASIC vezja je majhna poraba logičnih elementov in prostora zaželena, če pa je FPGA naša ciljna naprava, tedaj razumemo neizkoriščen prostor kot slabost. Zato smo raziskali tudi množilnike, ki izračunajo vse bite produkta hkrati. Realizirani so s kombinatoričnimi vezji, ki so zelo hitra, a prostorsko zelo zahtevna. Kompromis med prostorsko in časovno zahtevnostjo ponuja hibridni množilnik, ki v eni urini periodi izračuna več bitov produkta. S spreminjanjem števila vzporedno izračunanih bitov smo poiskali najoptimalnejše množilnike pri različnih razsežnostih izbranih obsegov GF(2^m). Izkaže se, da lahko v vseh primerih uporabimo množilnik, ki produkt izračuna v dveh urinih periodah (pri tem ne štejemo dodatnih ciklov, potrebnih za sprejem operandov, in vpis rezultata v registre). Optimalnost množenja je zelo pomembna, saj je množenje osnovni gradnik mnogih drugih operacij, in zato pogojuje učinkovitost njihove implementacije. Pri implementaciji kvadriranja se je najbolje izkazalo kar kombinatorično vezje, ki ima v primerjavi s kombinatoričnimi množilniki tudi majhno prostorsko zahtevnost. Pri implementaciji potenciranja pa dobimo najboljše rezultate z uporabo kombinatoričnega množilnika; štirikratna pohitritev v primerjavi z modulom, ki za množenje uporabi hibridni Montgomeryjev množilnik, namreč več kot le odtehta večjo porabo virov na FPGA.

Oglejmo si še zgornja nivoja sheme 7.1. Učinkovitost seštevanja in podvojevanja točk eliptične krivulje je pogojeno z učinkovitostjo operacij v končnih obsegih. Časovno zahtevno invertiranje se izkaže kot velika omejitev, zato je bilo predlaganih mnogo algoritmov osnovanih tako, da potrebujejo čim manj inverzov. S prehodom na projektivne koordinate dobimo implementacije seštevanja in podvojevanja brez računanja inverza, ki je porebno šele ob prehodu na običajne koordinate. Pri realizaciji teh modulov se pokaže lepota implementacije na FPGA; izvajamo več množenj hkrati, rezultati pa so na voljo istočasno. Najboljše rezultate smo dosegli z uporabo projektivnih koordinat Lòpez-Dahab, ki omogočijo kar 11-kratno pohitritev množenja točke s skalarjem. Le-to smo implementirali v najbolj osnovni obliki, to je z uporabo seštevanj in podvojevanj. Obstaja še vrsta algoritmov, ki z uporabo vnaprejšnjega računanja in drugačnih načinov zapisa skalarja optimizirajo to operacijo. Nadaljni razvoj projekta bi zahteval njihovo implementacijo in primerjavo učinkovitosti, ki bi omogočila izbiro najustreznejšega algoritma za množenje kP.

Opisana arhitektura zagotavlja strojno podporo kriptografskim protokolom. Izbrani dizajn zaseda 70 odstotkov resursov FPGA. Tako ostane dovolj prostora za implementacijo protokolov, ki bi zagotovili učinkovito komunikacijo med programsko opremo in koprocesorjem.

Dodatek A

Virtex-6 FPGA družina

Izbran je bil Virtex-6 FPGA XC6VLX240T v FF1156 ohišju. Njegove specifikacije so prikazane v tabeli A.1. Na voljo imamo 37680 rezin, ki imajo skupaj 241152 LUT-ov in pa 301440 D-pomnilnih celic (flipflops). V FF1156 ohišju imamo na voljo 600 vhodno/izhodnih nožic.

VIRTEX®-6 FPGAS

XILINX

		Virtex-6 LXT FPGAs Optimized for High-Performance Logic and DSP with Low-Power Serial Connectivity (1.0V, 0.9V)						Virtex-6 SXT FPGAs Optimized for Ultra High-Performance DSP with Low-Power Serial Connectivity (1.0V, 0.9V)		Virtex-6 HXT FPGAs Optimized for Communications Systems that Require Highest-Bandwidth Serial Connectivity (1.0V)			
	Part Number	XC6VLX75T	XC6VLX130T	XC6VLX195T	XC6VLX240T	XC6VLX365T	XC6VLX550T	XC6VLX760	XC6VSX315T	XC6VSX475T	XC6VHX250T	XC6VHX255T	XC6VHX380T	XC6VHX565T
	EasyPath™ FPGA Cost Reduction Solutions[1]	XCE6VLX75T	XCE6VLX130T	XCE6VLX195T	XCE6VLX240T	XCE6VLX365T	XCE6VLX550T	XCE6VLX760	XCE6VSX315T	XCE6VSX475T	XCE6VHX250T	XCE6VHX255T	XCE6VHX380T	XCE6VHX565T
Logic Resources	Slices[2]	11,640	20,000	31,200	37,680	56,880	85,920	118,560	49,200	74,400	39,360	39,600	59,760	88,560
	Logic Cells[3]	74,496	128,000	199,680	241,152	364,032	549,888	758,784	314,880	476,160	251,904	253,440	382,464	566,784
	CLB Flip-Flops	93,120	160,000	249,600	301,440	455,040	687,360	948,480	393,600	595,200	314,880	316,800	478,080	708,480
	Maximum Distributed RAM (Kb)	1,045	1,740	3,040	3,650	4,130	6,200	8,280	5,090	7,640	3,040	3,050	4,370	6,370
Memory Resources	Block RAM/FIFO w/ECC (36 Kb each)	156	264	344	416	416	632	720	704	1,064	504	516	768	912
	Total Block RAM (Kb)	5,616	9,504	12,384	14,976	14,976	22,752	25,920	25,344	38,304	18,144	18,567	27,648	32,832
Clock Resources	Mixed-Mode Clock Managers (MMCM)	6	10	10	12	18	18	18	12	18	12	12	18	18
	Maximum Single-Ended I/O	360	600	600	720	720	1,200	1,200	720	840	320	480	720	720
I/O Resources[4,5]	Maximum Differential I/O Pairs	180	300	300	360	360	600	600	360	420	160	240	360	360
	DSP48E1 Slices	288	480	640	768	576	864	864	1,344	2,016	576	576	864	864
	PCI Express® Interface Blocks	1	2	2	2	2	2	–	2	2	4	2	4	4
Embedded Hard IP Resources[6]	10/100/1000 Ethernet MAC Blocks	4	4	4	4	4	4	–	4	4	4	2	4	4
	GTX Low-Power Transceivers	12	20	20	24	24	36	–	24	36	48	24	48	48
	GTH High-Speed Transceivers	–	–	–	–	–	–	–	–	–	–	–	–	24
Speed Grades	Commercial	-1,-1,-2,-3	-1,-1,-2,-3	-1,-1,-2,-3	-1,-1,-2,-3	-1,-1,-2,-3	-1,-1,-2	-1,-1,-2	-1,-1,-2,-3	-1,-1,-2	-1,-2,-3	-1,-2,-3	-1,-2,-3	-1,-2
	Extended	–	–	–	–	–	-2	-2	–	-2	–	–	-2	–
	Industrial	-1,-1,-2	-1,-1,-2	-1,-1,-2	-1,-1,-2	-1,-1,-2	-1,-1	-1,-1	-1,-1,-2	-1,-1,-1	-1,-2	-1,-2	-1,-2	-1
Configuration	Configuration Memory (Mb)	28.3	43.8	61.6	73.9	96.1	144.1	184.9	104.5	156.7	79.9	79.9	119.8	160.7
	Package[7]	Area	Available User I/O: SelectIO™ Interface Pins[4,5] (GTX Low-Power Transceivers, GTH High-Speed Transceivers)											
FFA Packages (FF): Flip-Chip, fine-pitch BGA (1.0 mm ball spacing)														
	FFE484	23 x 23 mm	240 (8, 0)	240 (8, 0)										
	FFE784	29 x 29 mm	360 (12, 0)	400 (12, 0)	400 (12, 0)									
	FFE1156	35 x 35 mm		600 (20, 0)	600 (20, 0)	600 (20, 0)			600 (20, 0)	600 (20, 0)				
	FFE1759	42.5 x 42.5 mm				720 (24, 0)	840 (36, 0)		720 (24, 0)	840 (36, 0)				
	FFE1760	42.5 x 42.5 mm					1,200 (0, 0)	1,200 (0, 0)						
	FFE1154	35 x 35 mm									360 (48, 0)		320 (48, 0)	
	FFE1155	35 x 35 mm										440 (24, 12)	440 (24, 12)	
	FFE923	45 x 45 mm										480 (24, 24)	720 (40, 24)	720 (40, 24)
	FFE924	45 x 45 mm											640 (48, 24)	640 (48, 24)

XMP068 (v1.1)

Notes: 1. EasyPath FPGAs provide a conversion-free, low-risk path to volume production.
2. A single Virtex-6 FPGA CLB comprises two slices, each containing four 6-input LUTs and eight flip-flops (twice the number found in a Virtex-4 FPGA slice), for a total of eight 6-input LUTs and 16 flip-flops per CLB.
3. Virtex-6 FPGA logic cell ratings reflect the increased logic capacity offered by the 6-input LUT architecture.
4. Digitally Controlled Impedance (DCI) is available on I/Os of all devices.
5. Supported I/O standards include: HT, LVCMOS (1.2V, 1.5V, 1.8V, 2.5V); HSTL I (1.2V,1.8V), HSTL II (1.5V,1.8V), HSTL III (1.5V,1.8V), LVDS, Extended LVDS, RSDS, Bus LVDS, LVPECL, SSTL I (1.8V, 2.5V), SSTL II (1.8V, 2.5V), and SSTL (1.5V).
6. One System Monitor block is included in all devices.
7. All products are available Pb-free and RoHS-Compliant (FFG).
5. Supported I/O standards include: HT, LVDS, LVDSEXT, RSDS, BLVDS, ULVDS, LVPECL, LVCMOS33, LVCMOS25, LVCMOS18, LVCMOS15, LVTTL, PCI33, PCI66, PCI-X, GTL, GTL+, HSTL I (1.2V, 1.5V, 1.8V), HSTL II (1.5V, 1.8V), HSTL III (1.5V, 1.8V), HSTL IV (1.5V, 1.8V), SSTL2 I, SSTL II, SSTL18 I, and SSTL18 II.

Important: Verify all data in this document with the device data sheets found at www.xilinx.com

Slika A.1: Virtex-6 FPGA družina

Dodatek B

Algebra

Definicija B.1 *Neprazna množica $\mathcal{G}$, na kateri je definirana binarna operacija $\circ : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$, tvori grupo $(\mathcal{G}, \circ)$, če velja:*

1. *za vsaka $a, b \in (G)$ je $a \circ b \in \mathcal{G}$ (zaprtost za $\circ$),*
2. *obstaja tak element $e \in (G)$, da za vsak $g \in \mathcal{G}$ velja $e \circ g = g \circ e = g$ (enota),*
3. *za vsak $g \in \mathcal{G}$ obstaja $f \in \mathcal{F}$, tako da velja $g \circ f = f \circ g = e$ (inverz),*
4. *za vse $a, b, c \in \mathcal{G}$ velja $(a \circ b) \circ c = a \circ (b \circ c)$ (asociativnost).*

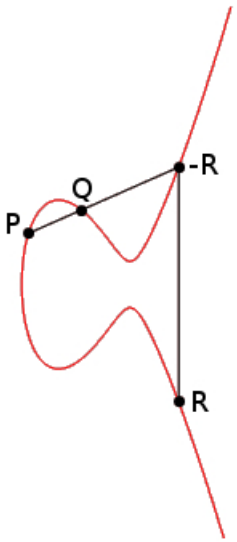
Če je operacija $\circ$ tudi komutativna, tako da za vsak $a, b \in \mathcal{G}$ velja $a \circ b = b \circ a$, govorimo o *komutativni* oz. *Abelovi grupi*.

Definicija B.2 *Neprazna množica $\mathcal{K}$ z binarnima operacijama $+, * : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$ tvori kolobar $(\mathcal{K}, +, *)$, če velja:*

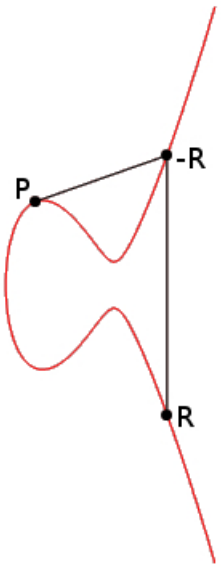
1. *$((K), +)$ je grupa z enoto za seštevanje 0,*
2. *asociativnost $*$: za vse $a, b, c \in \mathcal{K}$ je $a * (b * c) = (a * b) * c$,*
3. *distributivnost: za vse $a, b, c \in \mathcal{K}$ velja $a * (b + c) = a * b + a * c$ in $(b + c) * a = a * a + c * a$.*

Definicija B.3 *Neprazna množica $\mathcal{O}$ z binarnima operacijama $+, * : \mathcal{O} \times \mathcal{O} \to \mathcal{O}$ tvori obseg $(\mathcal{O}, +, *)$, če velja:*

1. *$((O), +)$ je grupa z enoto za seštevanje 0,*
2. *$((O) \backslash \{0\}, *)$ je grupa z enoto za množenje 1,*
3. *distributivnost: za vse $a, b, c \in \mathcal{O}$ velja $a * (b + c) = a * b + a * c$ in $(b + c) * a = a * a + c * a$.*



Slika B.1: Pravilo sekante za seštevanje točk na eliptični krivulji: $P + Q = R$



Slika B.2: Pravilo tangente za podvojevanje točke na eliptični krivulji: $2P = R$

Dodatek C

Rezultati hibridnih modulov za kvadriranje Montgomery

m	G	S	$lastG$	FFs	LUTi	Rezine	u.p.	Št. ciklov	Skupen čas
11	S	/	/	29	31	16	1.492	14	20.888
11	2	6	1	26	31	16	1.545	9	13.905
11	3	4	2	26	33	13	1.437	7	10.059
11	4	3	3	23	31	15	1.515	6	9.090
11	6	2	5	21	31	15	1.535	5	7.675
11	8	2	3	21	33	14	1.614	5	8.070
11	10	2	1	22	33	17	1.834	5	9.170
11	K	/	/	/	3	2	/	/	5.733

Tabela C.1: Kvadriranje Montgomery - hibridna verzija z $m = 11$ - rezultati implmentacije

m	G	S	$lastG$	FFs	LUTi	Rezine	u.p.	Št. ciklov	Skupen čas
63	S	/	/	134	103	47	2.336	66	154.176
63	2	32	1	133	131	47	2.365	35	82.775
63	3	21	3	132	102	46	2.488	24	59.712
63	6	11	3	128	161	69	2.727	14	38.178
63	10	7	3	124	160	65	2.270	10	22.700
63	12	6	3	123	161	68	2.379	9	21.411
63	16	4	15	115	152	56	2.212	7	15.484
63	20	4	3	119	176	75	2.465	7	17.255
63	29	3	5	115	198	87	2.393	6	14.358
63	32	2	31	99	166	72	2.487	5	12.435
63	38	2	25	99	226	95	2.884	5	14.420
63	42	2	21	98	212	97	2.944	5	14.720
63	44	2	19	98	194	88	2.642	5	13.210
63	K	/	/	/	16	15	/	/	9.347

Tabela C.2: Montgomery kvadriranje - hibridna verzija z $m = 63$ - rezultati sinteze

m	G	S	$lastG$	FFs	LUTi	Rezine	u.p.	Št. ciklov	Skupen čas
113	S	/	/	235	180	94	2.247	116	260.652
113	2	57	1	234	277	102	2.963	60	177.78
113	3	38	1	234	278	139	2.640	41	108.24
113	4	29	3	232	277	112	2.815	32	90.080
113	7	17	6	230	274	99	2.508	20	50.160
113	10	12	7	227	276	109	2.339	15	35.085
113	15	8	7	228	278	121	2.204	11	24.244
113	25	5	12	220	291	150	1.961	8	15.688
113	30	4	7	215	286	146	1.943	7	13.601
113	39	3	4	211	288	110	1.914	6	11.484
113	43	3	16	211	287	117	1.878	6	11.268
113	52	3	43	201	283	106	2.026	6	12.156
113	59	2	54	201	283	106	2.099	5	10.495
113	64	2	49	174	239	103	2.139	5	10.695
113	71	2	42	197	267	118	2.261	5	11.305
113	80	2	33	174	274	130	2.145	5	10.725
113	K	/	/	/	28	25	/	/	10.819

Tabela C.3: Kvadriranje Montgomery - hibridna verzija z $m = 113$ - rezultati sinteze

m	G	S	$lastG$	FFs	LUTi	Rezine	u.p.	Št. ciklov	Skupen čas
191	S	/	/	393	299	176	3.132	194	607.608
191	2	96	1	390	479	207	3.595	99	355.905
191	3	64	2	390	480	199	2.635	67	176.545
191	4	48	1	389	475	194	2.537	51	129.387
191	6	32	1	389	475	196	2.961	35	103.635
191	10	20	9	383	471	192	2.317	23	53.291
191	17	12	13	381	488	315	2.465	15	36.975
191	20	10	9	375	482	258	2.185	13	28.405
191	28	7	5	372	489	225	2.323	10	23.230
191	33	6	7	372	500	266	2.381	9	21.429
191	38	6	37	352	489	290	2.452	9	22.068
191	49	4	5	364	507	258	2.144	7	15.008
191	50	4	9	359	506	272	2.126	7	14.882
191	64	3	1	356	478	196	2.320	6	13.920
191	71	3	22	351	504	210	2.232	6	13.392
191	77	3	40	348	488	198	2.317	6	13.902
191	82	3	55	319	467	202	2.440	6	14.640
191	86	3	67	309	444	181	2.572	6	15.432
191	98	2	5	335	545	221	2.485	5	12.425
191	100	2	9	332	532	224	2.469	5	12.345
191	106	2	21	322	489	207	2.638	5	13.190
191	115	2	39	329	490	198	2.579	5	12.895
191	121	2	51	325	516	206	2.758	5	13.790
191	K	/	/	/	80	56	/	/	12.793

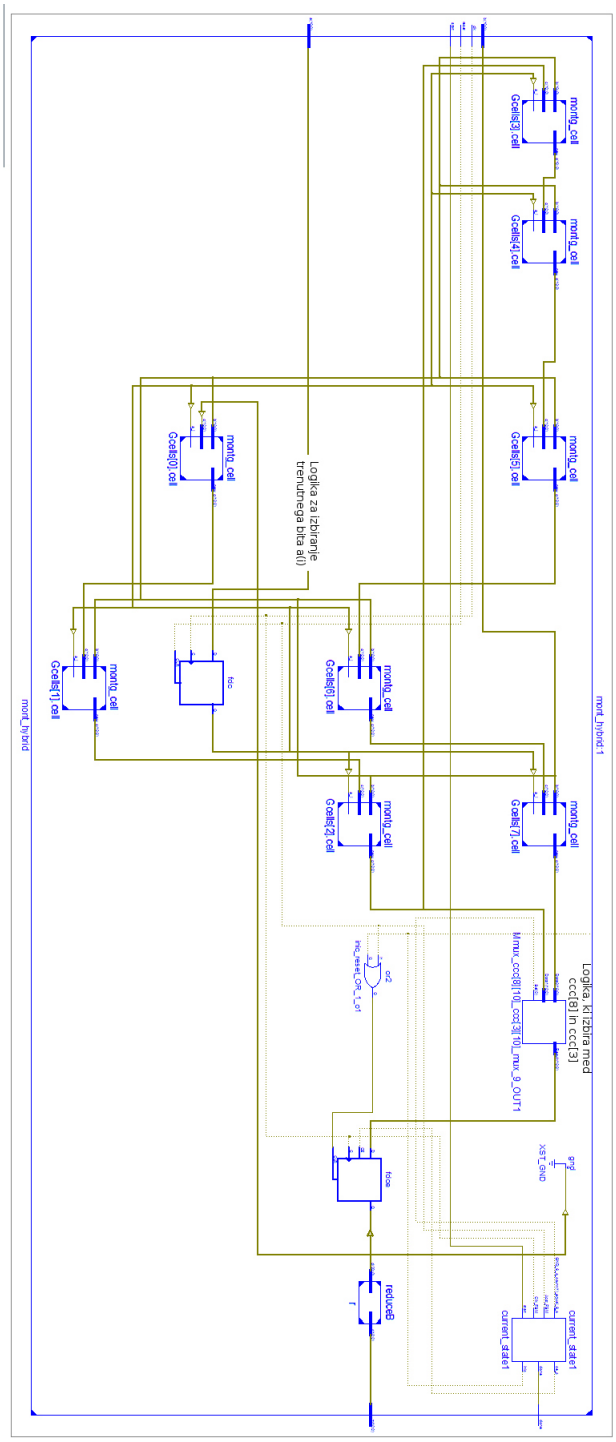
Tabela C.4: Kvadriranje Montgomery - hibridna verzija z $m = 191$ - rezultati sinteze

Dodatek D

RTL sheme nekaterih modulov

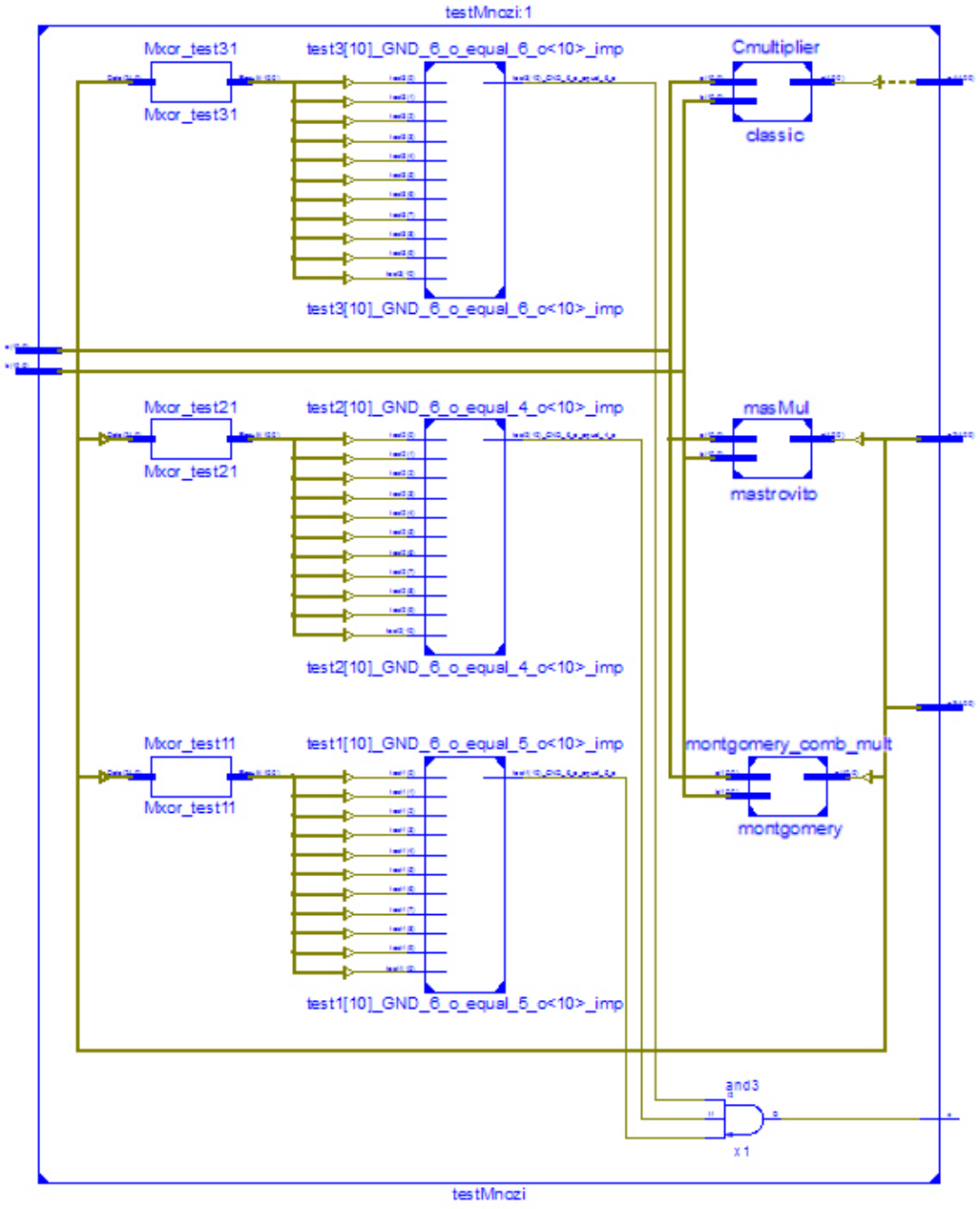
D.1 Hibridni Montgomeryjev množilnik

Pri izbranem $G = 8$ izračunamo 8 delnih produktov v eni urini periodi. To pomeni vezavo osmih modulov *montg_cell*. Na sliki D.1 vidimo, da je izhod prvega modula *montg_cell* (z oznako `Gcells[0].cell`) povezan na vhod drugega *montg_cell* modula, izhod le-tega na vhod tretjega modula, itn. Ker velja $11 = 1 \cdot 8 + 3$, je produkt na voljo po dveh urinih periodah. Multipleksor (na sliki je označen z "logika, ki izbira med ccc[8] in ccc[3]"), ki ima izbirni vhod vezan na vrednost števca `count`, izbira med izhodoma dveh modulov *montg_cell*. V prvem obhodu izbere rezultat `ccc[8]` modula z oznako `Gcells[7].cell`, v drugem obhodu pa `ccc[3]`, ki ga je izračunal modul `Gcells[2].cell`. Izbran (delni) produkt shrani v register. Izhod tega registra je povezan kot vhod prve Montgomeryjeve celice in kot vhod v modul za redukcijo *reduceB*. Ko KA preide v stanje 2 in aktivira signal `done`, je na izhodu modula *mont_hybrid* veljavni produkt.

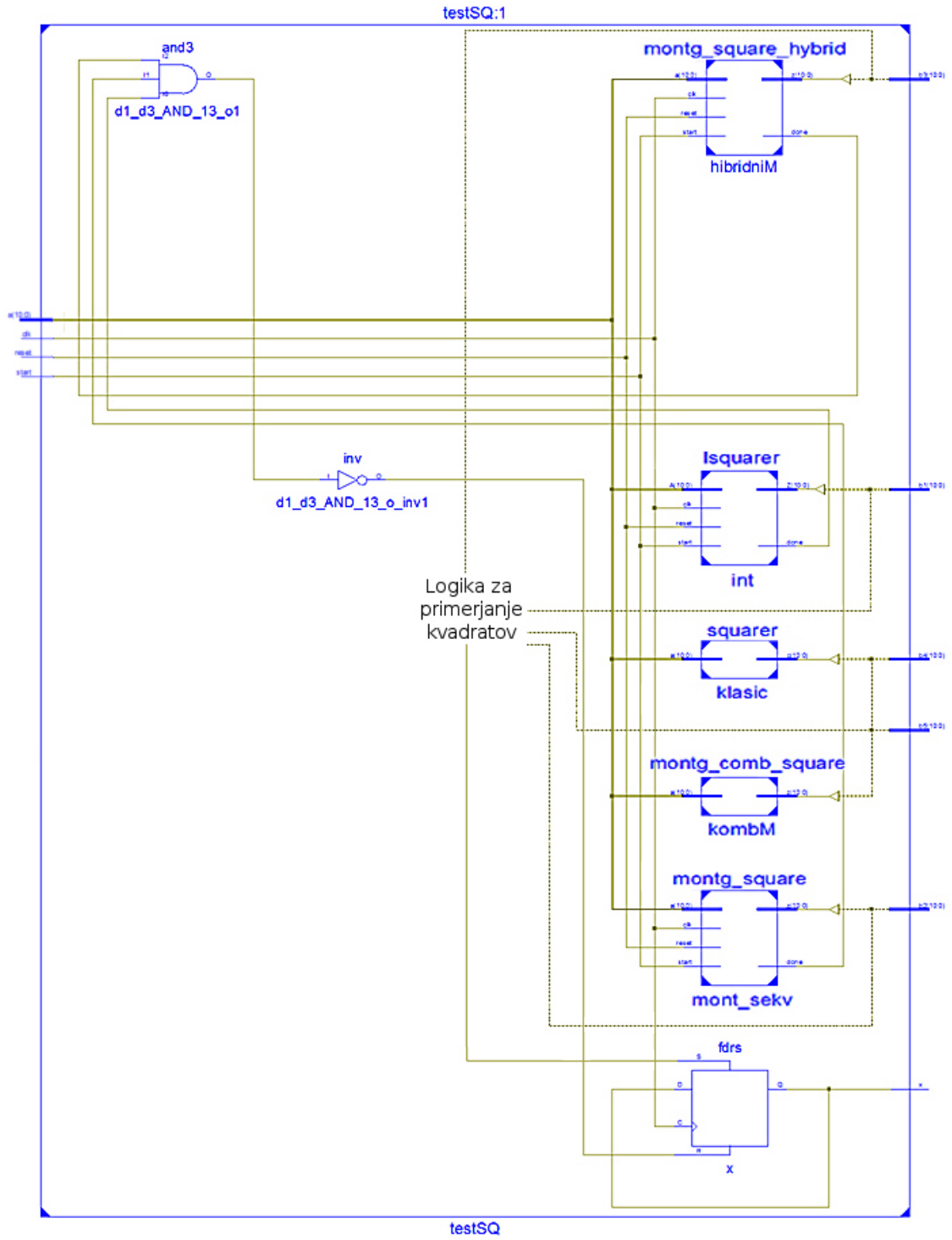


Slika D.1: Modul *mont_hybrid* za $m = 11$ in $G = 8$ - RTL shema

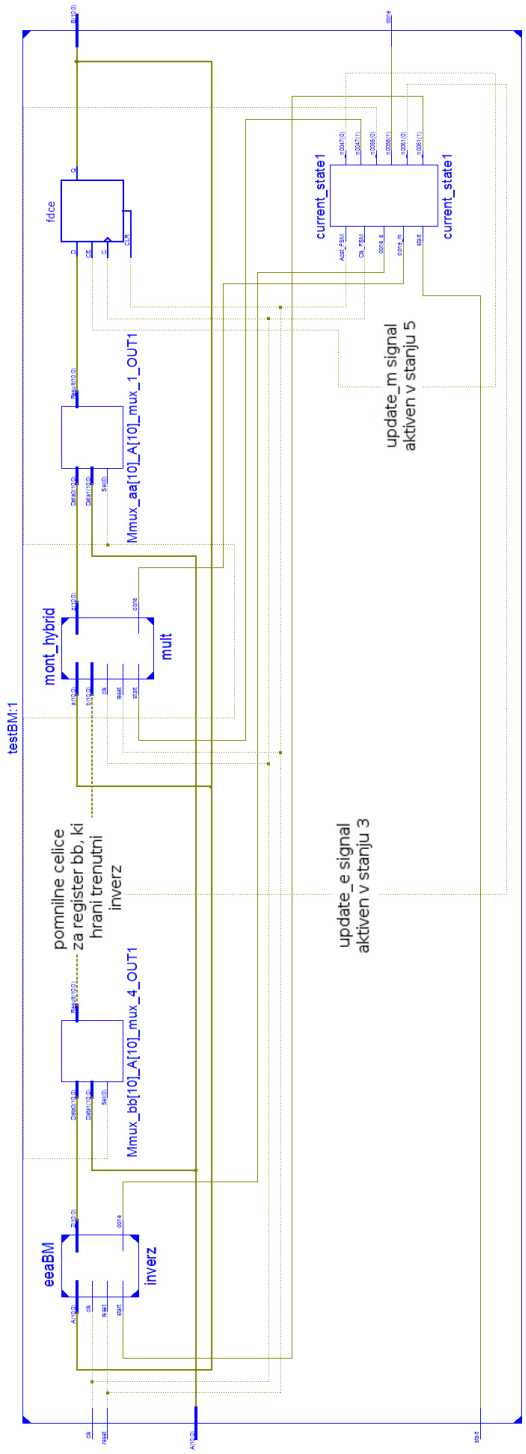
D.2 RTL sheme testnih modulov



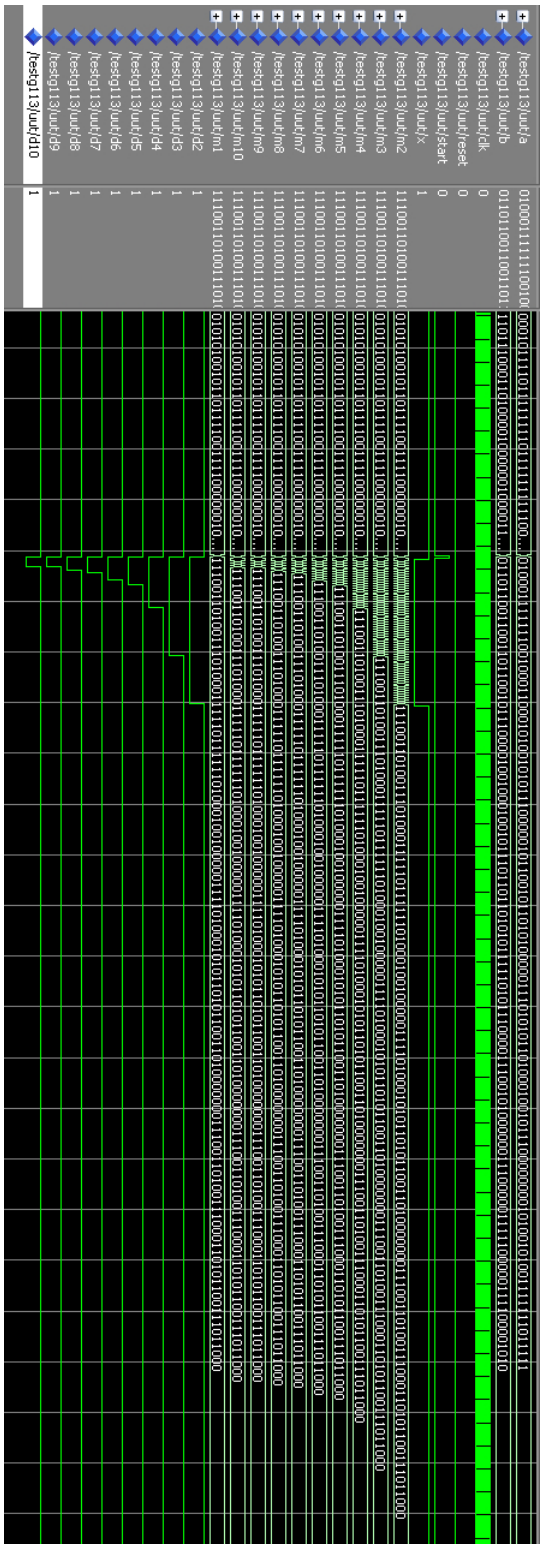
Slika D.2: Testni modul $testMnozi$ za $m = 11$ - RTL shema



Slika D.3: Testni modul *testSQ* za $m = 11$ - RTL shema



Slika D.4: Testni modul $testEEABM$ za računanje inverza po Berlekampovem algoritmu $m = 11$ - RTL shema



Slika D.5: Simulacija modula *testG* za $m = 113$

Slike

Tabele

Seznam algoritmov

Literatura

[1] (2011) Xilinx - Glossary (UG659)
Dostopno na :
http://www.xilinx.com/support/documentation/sw_manuals/glossary.pdf

[2] T. Huffmire, C. Irvine, Thudy D. Nguyen, T. Levin, R. Kastner, T. Sherwood, "Handbook of FPGA Design Security", Dordrecht, Heidelberg, London, New York: Springer , 2010, pogl. 1.

[3] (2010) Xilinx - Corporate Backgrounder
Dostopno na:
http://www.xilinx.com/publications/prod_mktg/corporate-backgrounder.pdf

[4] Francisco Rodriguez-Henriquey, N.A. Saqib, A. Diaz Perez, Cetin Kaya Koc, "Cryptographic Algorithms on Reconfigurable Hardware", New York: Springer , 2006, pogl. 3, 4, 10.

[5] (2008) Xilinx - Programmable Logic Design - Quick Start Guide (UG500(v1.0))
Dostopno na :
http://www.xilinx.com/support/documentation/boards_and_kits/ug500.pdf

[6] Stephen Brown and Jonathan Rose, "Architecture of FPGAs and CPLDs: A Tutorial", *IEEE Design and Test of Computers*, Vol. 13, No. 2, pp. 42-57, 1996. Dostopno na:
http://www.eecg.toronto.edu/˙ jayar/pubs/brown/survey.pdf

[7] (2011) Xilinx - Virtex-6 Family Overview (DS150(v2.3))
Dostopno na :
http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[8] (2009) Xilinx - Virtex-6 FPGA Configurable Logic Block - User Guide (UG364(v1.1))
Dostopno na :
http://www.xilinx.com/support/documentation/user_guides/ug364.pdf

[9] J. Rose, A.El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", *Proceedings of the IEEE*, Vol.81, No. 7, July 1993
Dostopno na:
http://www.eecg.toronto.edu/ jayar/pubs/rose/PIEEE93a.pdf

[10] (2010) Xilinx - Virtex-6 FPGA SelectIO Resources - User Guide (UG361(v1.3))
Dostopno na :
http://www.xilinx.com/support/documentation/user_guides/ug361.pdf

[11] Irena Majcen, "Smelo na Olimp: 303 rešene naloge iz teorije števil", Ljubljana: Društvo matematikov, fizikov in astronomov Slovenije, 2011, pogl. 1.

[12] Ivan Niven, Herbert S. Zuckerman, Hugh L. Montgomery, "An Introduction to the Theory of Numbers, 5th. ed.", New York: John Wiley & Sons , 1991, pogl. 1, 2, 5.

[13] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, "Handbook of applied cryptography", New York: CRC Press , 1996, pogl. 2, 3.

[14] Dušan Pagon, "Kongruence in Eulerjev izrek", Presek, 15(1987/1988), 194-196

[15] Jernej Tonejc, "Aritmetika dvojiških končnih obsegov", Obzornik za matematiko in fizko (57), 157-175

[16] Jernej Barbič, Schoof Algorithm (Schoofov algoritem), August 11, 2000.

[17] Dale Husemŏller, "Elliptic curves, 2nd. ed.", New York: Springer Verlag, 2000, pogl. 1, 2.

[18] J. Deschamps, J.L. Imana in G. D. Sutter , "Hardware Implementation of Finite-Field Arithmetic", 2009, pogl.6, 7 .

[19] Cetin K. Koc, T. Acar "Montgomery multiplication in GF(2^k)", Designs, Codes and Cryptography, 14(1), 57-69, Kluwer Academic Publishers, Boston, 1998

[20] K. Kobayashi, N. Takagi, K. Takagi, "An Algorithm for Inversion in GF(2^m) Suitable for Implementation Using a Polynomial Multyply Instruction on GF(2)" Dostopno na:
http://www2.lirmm.fr/arith18/papers/kobayashi-AlgorithmInversionUsingPolynomialMultiplyInstruction.pdf

[21] Darrel Hankerson, Alfred J. Menezes, Scott A. Vanstone, "Guide to Elliptic Curve Cryptography", New York: Springer-Verlag , 2004, pogl. 2, 3.

[22] Elwyn R. Berlekamp, "Algebraic Coding Theory", Laguna Hills, Aegean Park Press, 1984, pogl. 2

[23] H.Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography", Boca Raton: Chapman & Hall//CRC, 2006, pogl. 13.

[24] Julio Lòpez, Ricardo Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in GF2^n",SAC'98, LNCS Springer Verlag, 1998