NTNU
Det skapende universitet

RSA krypterings-system for AHEAD

Vidar Eikrem Hervig

Master i elektronikk
Oppgaven levert: Februar 2007
Hovedveileder: Kjetil Svarstad, IET
Biveileder(e): Dag K. Rognlien, SINTEF IKT

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Det skal videreutvikles en demonstrator for AHEAD-konseptet slik at man kan vise både konfigurering og kjøring av RSA-basert krypterings-system med en PDA som mobil klient og en AHEAD-tag som krypterings-server.

Primære mål med oppgaven er:

1. Spesifisere en arkitektur for klient-server-systemet som bestemmer hvilke oppgaver som skal kjøres hvor, og hvordan oppsett og kjørende transaksjoner vil foregå.

2. Utvikle RSA-moduler for HW-realisering på en Spartan Xilinx FPGA, og å tilpasse denne for Suzaku-kortet.

3. Utvikle nødvendig driver-SW for Linux på Suzaku for å gjøre RSA-modulene på FPGA'en tilgjengelig for eksterne klienter.

Hvis tiden tillater det, skal man også forsøke:

- Utvikle et program som skal kjøre på en klient (PDA eller laptop-PC) som kobler seg til AHEAD-tag og bruker RSA-SW og -HW for kryptering av kommunikasjon.

- Utvikle et enkelt program for en PC som kan motta den krypterte data-strømmen fra AHEAD-tag'en, de-kryptere denne, og vise at den korrekte meldingen er mottatt fra AHEAD-tag'en.

Oppgaven gitt: 04. september 2006
Hovedveileder: Kjetil Svarstad, IET

Sammendrag

I denne masteroppgaven er det blitt designet en RSA-modul for forskningsprosjektet AHEAD. Dette er en modul som gjennomfører en kryptografialgoritme. RSA-modulen omformer klartekst til chiffertekst, (kryptering), og tilbake igjen til klartekst (dekryptering). Dette gjøres ved å lage en krets som utfører modulær eksponering.

Kretsrealiseringen av RSA-algoritmen er gjort ved å benytte venstre til høyre binær metoden og Blakleys algoritme. Simulering med 128 bitskryptering gir korrekt oppførsel. Det er blitt testet med to meldinger som er blitt kryptert og dekryptert igjen. Syntese med 128 bitskryptering bruker 13% av slicene på Spartan 3 FPGA'en med en frekvens på 59Mhz. Ved syntese med 256 og 512 bitskryptering øker antall slicer lineært, mens frekvensen synker.

Det prøvd er å tilpasse krypteringsmodulen til utviklingskortet som benyttes innen AHEAD. Arbeidet med utviklingsverktøyene har vært tidkrevende, spesielt har det vært problemer med å laste ned ny bit-fil på kortet. Det har derfor ikke blitt tid til å ferdigstille dette arbeidet. I slutten av denne rapporten er det skissert hvordan dette kan løses, med registertilkoblinger og beskrivelse av programvarens oppgaver på utviklingskortet.

Forord

Oppgaven er blitt utført innen forskningsprosjektet ”Ambient Hardware Embedded Architectures on Demand” (AHEAD) for professor Kjetil Svarstad ved Institutt for elektronikk og telekommunikasjon ved NTNU.

Arbeidet har hatt fokus på raskt å finne en basis for kommunikasjon for en krypteringsmodul laget etter RSA-algoritmen. Denne ble utviklet tidlig i arbeidsperioden og har blitt laget rett frem etter venstre til høyre binær metoden og Blakleys algoritme. Emulering av modulen på det utvalgte utviklingskortet for AHEAD, har vist seg å være komplisert og tiden strakk ikke til for å gjøre dette ferdig.

Ønsker å takke prosjektstudentene for hjelp med utviklingskortet i arbeidsperioden. Spesielt ønsker undertegnede å takke veileder professor Kjetil Svarstad for hjelp med oppgaven og veiledning underveis i arbeidet.

Vidar Eikrem Hervig

Trondheim, februar 2007

Innholdsfortegnelse

Illustrasjonsliste

Tabelliste

Ordliste

AHEAD	Ambient Hardware Embedded Architectures on Demand
CLB	Configurable Logic Block
EDK	Embedded Developer's Kit (Xilinx)
FPGA	Field Programmable Gate Array
KBPS	Kilobits per second
LSB	Least Significant Bit
PDA	Personal Digital Assistent
I/O	Input/Output
ISE	Integrated Software Enviroment (Xilinx)
KLIENT	Betegnelse på en bruker av AHEAD.
OPB	On-chip Peripheral Bus
PDA	Personal Digital Assistant
RSA	Rivest, Shamir og Adleman, oppfinnere av algoritmen for asymmetrisk nøkler.
TAG	Betegnelse på preprosesseringsplattformen i AHEAD. Dette er lette data-maskiner med FPGA ombord.
TJENER	Betegnelse på tjenestetilbyderen i AHEAD
WLAN	Wireless Loacal Area Network
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
XST	Xilinx Synthesis Technologi

1 Introduksjon

Denne oppgaven er en del av forskningsprosjektet ”Ambient Hardware Embedded Architectures on Demand” (AHEAD) startet av Professor Kjetil Svarstad.

1.1 AHEAD

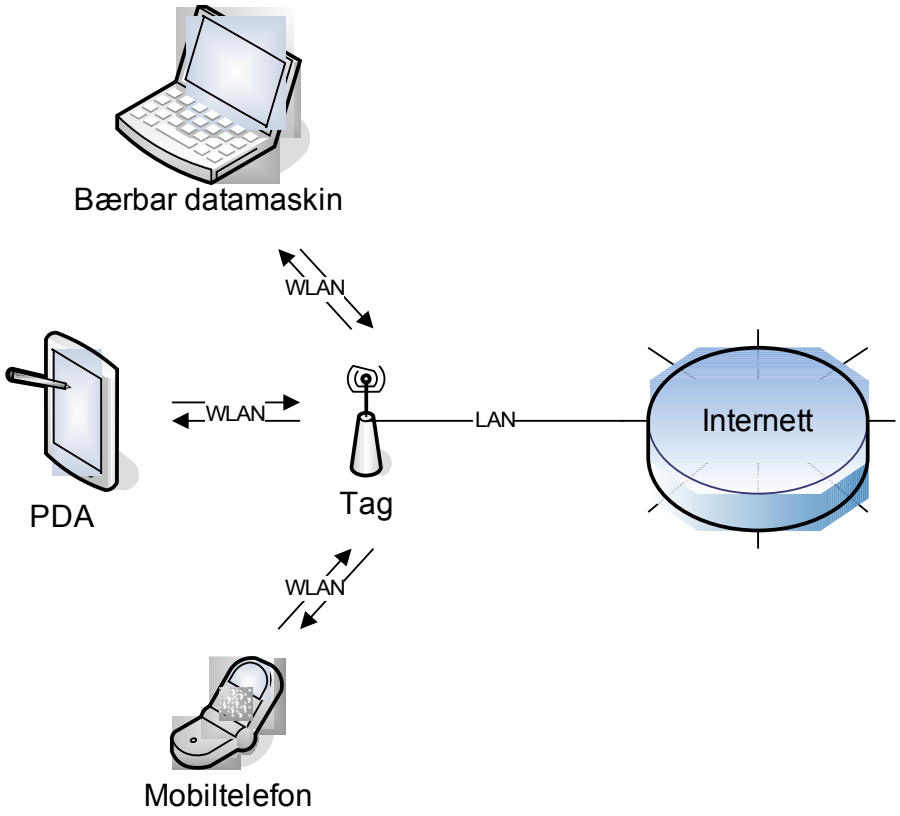
Utgangspunktet for AHEAD er den stadige utviklingen og miniatyriseringen av maskinvare. Det vil gjøre det mulig å integrere maskinvare som en del av våre omgivelser, dette kalles gjerne intelligente omgivelser.

Et typisk eksempel vil være at man har litt tid til overs og har et ønske om å bli underholdt. Ved hjelp av sin PDA, mobiltelefon, eller en bærbardatamaskin(klienter) vil man da kunne koble seg opp mot en såkalt AHEAD-tag. Brukeren vil da, gjerne med trådløs overføring, kunne velge om han vil se en video, spille av musikk, eller spille et spill på sin klient. AHEAD-tagen vil da bli satt opp til å levere de ønskede tjenestene til klienten.

Stadig kraftigere maskinvare gjør også at man kan løse oppgaver på nye måter, som ved delvis rekonfigurering av programmerbare logiske brikker(FPGA). Det vil da være mulig å kjøre enkelte oppgaver på deler av FPGA'en, mens det settes opp ny oppgaver på andre deler. Eksempel på andre oppgaver enn kryptering vil være dekoding av videoformatet mpeg4 til mpeg. Det slik mellomprosessering av data, for at håndholdt enheter skal kunne kjøre tunge oppaver som er hensikten med AHEAD-tagen.

Prosjektet er fortsatt på et eksperimentelt stadium, så hva som er mulig og ikke, er ikke helt sikkert enda. Selve tagen vil likne på utviklingskortet som det i dag benyttes til å teste på, kortet vil bli nærmere beskrevet i kapittel 2.1.

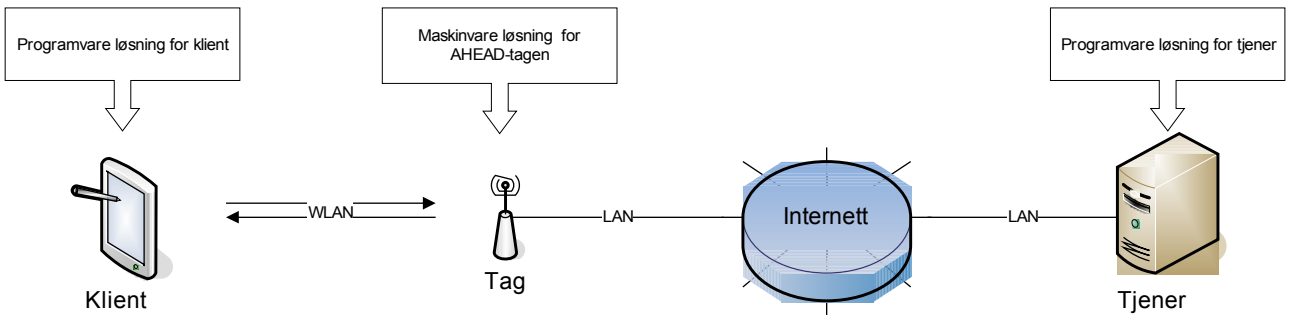
Et oppsett med klient og en tag tilknyttet internett er vist i Illustrasjon 1. Her er den trådløse tilkoblingen forslått å være WLAN, men det kan hende det vil være like hensiktsmessig å benytte blåtann. I illustrasjonen er det foreslått koblet til bærbare enheter som; mobiltelefon, PDA og bærbar datamaskin.



Illustrasjon 1: AHEAD-tagen med sin trådløse sender koblet til internett og trådløse enheter.

1.2 Målet for denne oppgaven

Ved kommunikasjon mellom de enkelte enhetene innen AHEAD-systemet, kan det være ønskelig for både leverandør av tjenester og brukeren av klienten å holde denne skjult for uvedkommende. Det vil derfor være nødvendig å utvikle en metode for å gjøre en melding mellom disse to aktørene uleselig for utenforstående.



Illustrasjon 2: Hvor vil tenkt utvikling av programvare/maskinvare finne sted.

I denne oppgaven vil kryptering ved hjelp av RSA-algoritmen gi en løsning på dette. Det betyr at

det må utvikles programmer både i maskinvare og programvare. For klient og tjener vil det være mest naturlig med en programvare løsning, mens tagen er tiltenkt en maskinvare løsning som vist i Illustrasjon 2.

Hovedmålet med oppgaven er i første omgang å skissere klient-tjener-systemet. Så vil fokuset være rettet mot å utvikle maskinvare-biten for AHEAD-tagen. Her også å tilpasse dette til utviklings-kortet. Hvis tiden strekker til; utvikle programvare for klient og tjener. Da vil man ha en ferdig prototyp for den ønskede kommunikasjonen.

1.3 Tidligere arbeid

Innen forskingsprosjektet AHEAD har det tidligere vært arbeidet med flere prosjekt- og masteroppgaver. Det arbeidet som har vært mest påvirkende for dette arbeidet har vært Stian Arntsens arbeid[ARN06]. Det har blitt valgt et egnet kort for å utvikle på i prosjektoppgaven. I masteroppgaven er det utviklet et enkelt klient-tjener-system som kan rekonfigurere maskinvaren fra en nettverkstjener.

1.4 Organisering av rapporten

Første del prøver å gi en introduksjon til AHEAD-prosjektet og hva denne oppgaven er ment å gjøre. Andre del er en skisse for kommunikasjonen i klient-tjener-systemet. Her er det foreslått en lettere krypteringsform i tillegg til den tiltenkte RSA-krypteringen. Det er også en kort beskrivelse av utviklingskortet. Dette er grunnlaget, som arbeidet bygger på.

I kapittel 3 er en beskrivelse av kryptografi, med bruksområde og tiltenkt bruk innen AHEAD. Deretter vil rapporten gå nærmere inn på RSA kryptering. Herunder vil en ta opp algoritmer for å realisere kryptering i maskinvare. Det er her forståelsen for kapittel 4 bygges opp. Det kapittelet viser hvordan RSA-algoritmen er implementert i det maskinvare beskrivende språket VHDL. Modulen er her nøye beskrevet og det er illustrert med tilstands- og flytdiagram.

Testing og syntese av kretsen er beskrevet i kapittel 6. Dette er oppsummert med tabeller og grafer, men både for kapittel 4 og 5 anbefales det å lese anviste vedlegg for en dypere forståelse.

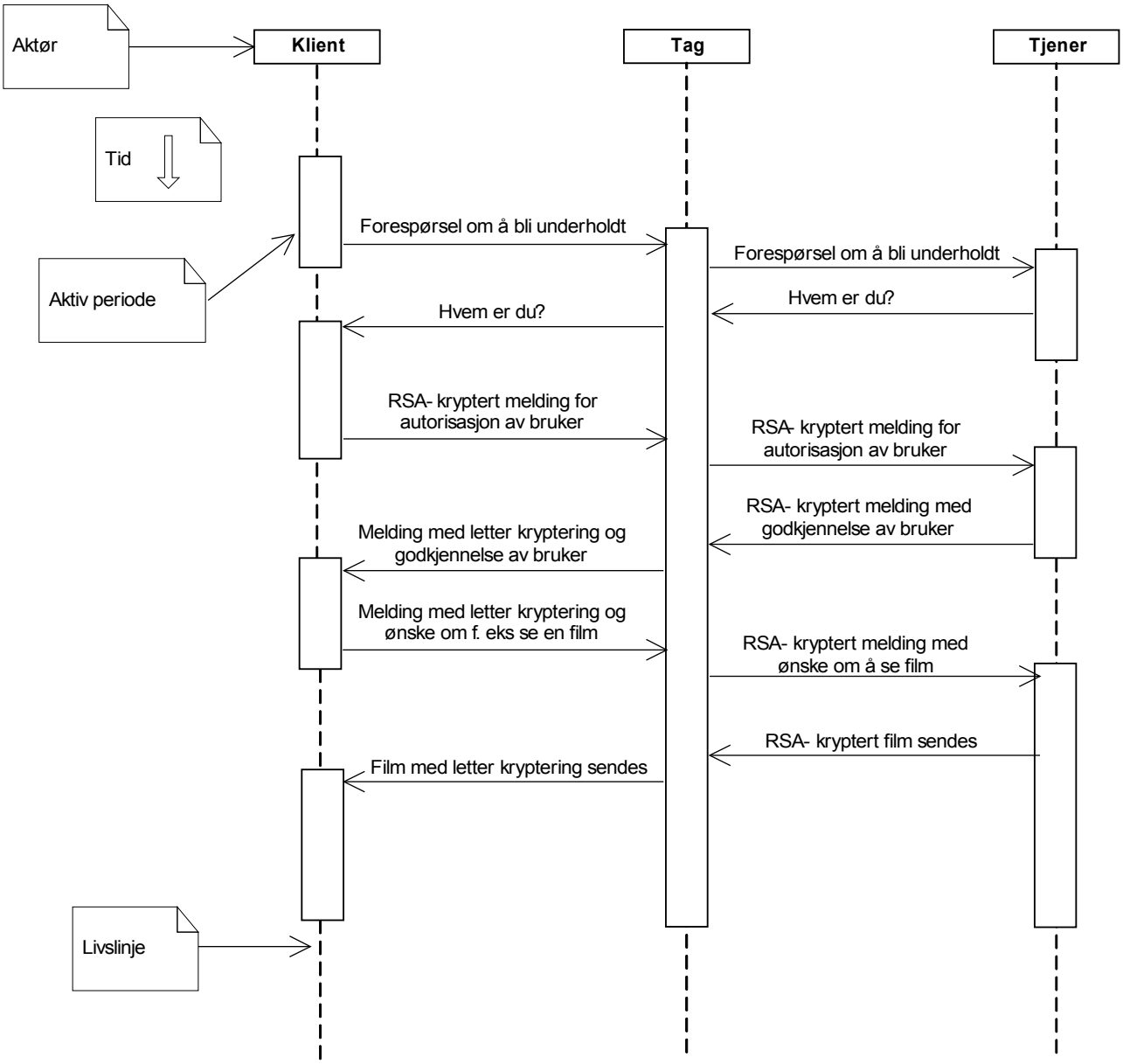
Så i kapittel 6 er arbeidet med Suzaku kortet og dets nødvendige utviklingsverktøy. Her er det gitt noen tips til hva som kan lønne seg å gjøre ved arbeid med Suzaku kortet. Det er så gitt forslag om hva som må gjøres videre for å få en ferdig prototyp. Til slutt vil resultater oppsummeres og arbeidet diskuteres før det gies en konklusjon.

2 Klient-tjener-systemet

Det ble først prøvd å lage en oversikt over kommunikasjonene for å definere systemet. I systemet må en da vite hvilken aktør som initierte de ulike handlingene og kommunikasjonene. Det må vises hvilken respons dette måtte ha. Dette er vist i Illustrasjon 3, som en ser blir dette en salgs "handshake" tankegang. Det vil si et system der tjeneren autentiserer klienten ved å sende en nøkkel for å kryptere brukernavn og passord. Dette gjør det mulig å sende brukernavn og passord i kryptert form. Så fra illustrasjonen ser man at tagen i starten bare videreformidler informasjonen med utlevering av nøkler og oversending av første krypterte melding med brukernavn og passord. Når forbindelsen er satt, vil tagen starte sin preprosessering av data. Her kommer dens omgjøring av kryptering fra en tung RSA-krypteringen til en kryptering der ikke krever så mye ressurser.

Problemet vil være hvordan klienten skal dekryptere meldingen fra tagen. En løsning med ekstra maskinvare er lite sannsynlig, da utvikling av enheter hvor alt er inkludert i en enhet ser ut til å selge best. Derfor er det en programvare løsning som her er foreslått, men dette vil kreve utrolig mye ressurser fra enheten. Derfor vil en foreslå at en tung kryptering som RSA, kun vil bli benyttet autorisasjon og en lettere kryptering vil ta seg av resten av kommunikasjonen. Autorisasjon vil si å bekrefte identiteten til brukeren. Den lettere krypteringen mellom tagen og klient vil foregå bak en brannmur og på lukket område, så det vil være mindre risiko ved denne kommunikasjonen enn den mellom tag og tjener.

Deretter prøvde man å se på hva man måtte utvikle av programvare og maskinvare. På tagen vil det være mest hensiktmessig med en maskinvare løsning. Det vil kreve for mye ressurser av den svake prosessoren å prøve og lage en programvare løsning for tagen. Tjeneren vil kunne benytte både en maskinvare og programvare løsning, men ettersom datamaskiner i dag er såpass kraftige vil en programvare løsning være billigst og raskt nok.



Illustrasjon 3: Tenkt scenario ved oppstart av kommunikasjon.

2.1 AHEAD-tagen

AHEAD-tagen som er valgt er et utviklingskort som heter Suzaku-S SZ030-U00. Det har en FPGA fra Xilinx med Spartan 3 løsning. FPGA'en brukt i denne oppgaven heter XC3S1000 og ut fra de tekniske spesifikasjonene i Tabell 1, kan en lese at det har i en million system gates og et totalt antall configurable logic blocks(CLBs) på 1920.

Tabell 1: Tekniske spesifikasjonene for Spartan 3[XIL06].

Spartan 3 XC3S100	
System gates	1M
Eqvivalent Logic Cells	17 280
Total CLBs	1 920
Distributed Ram Bits	129K
Block RAM Bits	432K
Dedicated Multipliers	24
DCMs	4
Maximum User I/O	391
Maximum Differential I/OPairs	175

På Suzaku kortet sitter også en 32 bits MicroBlaze softprosessor. Denne er satt opp med µLinux operativsystem med flere ferdig programmer installert, for nærmere beskrivelse se Software-maualen[ATM05].

3 Kryptering

3.1 Hva er kryptografi

Kryptografi går ut på å omforme klartekst til chiffertekst, og tilbake igjen til klartekst. Disse prosessene blir kalt kryptering og dekryptering. Målet er å gjøre meldinger uleselige for alle andre enn mottaker og sender. De trenger heller ikke å være leselig for sender.

Ut fra dette følger det at omformingen har som krav at en gitt chiffertekst bare kan representere en spesifikk melding. En gitt melding kan derimot resultere i flere utfall, men da må chifferteksten være større en den opprinnelige meldingen og alle utfallene må representere den samme meldingen. For å få god nok sikkerhet bør meldingen deles opp i tilstrekkelig store biter slik at det blir vanskelig å lage en oppslagstabell over klartekst-chiffertekst-par for uvedkommende. Den vanligste måten å løse kryptering på er at man benytter er matematisk formel for omforme klarteksten og en for omforme chifferteksten. Det er en fordel om den samme formelen kan benyttes. For å kunne benytte seg av samme formelen mellom flere sendere og mottakere som ikke skal kunne dekode hverandres meldinger så må formelen være parametriserbar, dette vil si at formelen må ta inn parametere som bare mottaker og sender vet om og som muliggjør kryptering/dekryptering. Disse parametrene kalles nøkler.

Eksempel 1: Man gir hver bokstav i alfabetet et nummer, forløpende. Slik at A= 1, B=2, C= 3 og så videre.

Krypteringen består av følgende formel:

$C = M + n$

Hvis en sier at n=3 så blir A=D, B=E osv.

Dekrypteringen blir da:

$M = C – n$

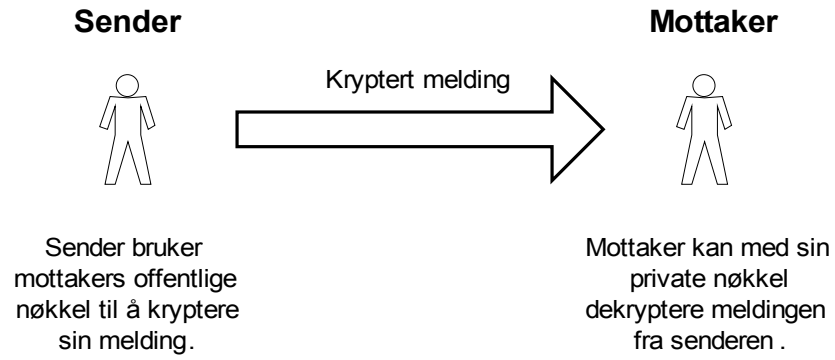
I Eksempel 1 beskrives den enkleste formen for kryptering, der en bokstav byttes ut med et tall. Hvis en istedenfor snur nummereringen på alfabetet i eksemplet, så vil man slippe unna med samme

formel. Dette gir en veldig enkel kryptering da meldingen deles opp i veldig små biter, en bokstav om gangen, og formelen er svært enkel. Hvis uvedkommende finner bare to klartekst-chiffertekst-par så vil man lett se sammenhengen og da kunne dekode meldingen. To nesten like klartekstbiter vil da måtte, med god kryptering, resultere i svært forskjellige chiffertekstbiter.

Videre følger det et problem med utveksling av nøklene for kryptering og dekryptering. Partene er nødt til å utveksle nøkler på en sikker måte før kryptert kommunikasjon kan foregå. Tradisjonelt har det vært brukt samme nøkkel for kryptering og dekryptering; symmetrisk nøkkel.

3.2 RSA-kryptering

I 1976 ble systemet med asymetriske nøkler funnet opp av Whitfield Diffie. Hans ide gikk ut på å ha et system med en nøkkel til kryptering og en annen nøkkel til dekryptering, disse kalles henholdsvis offentlig nøkkel og privat nøkkel. Den offentlig nøkkelen skal kunne gjøres kjent til allmennheten og den private skulle holdes hemmelig. Meldinger som krypteres med den offentlige nøkkelen kan bare dekrypteres med den private nøkkelen; se Illustrasjon 4.



Illustrasjon 4: Slik er systemet med asymetriske nøkler tenkt å fungere.

I april 1977 klarte tre forsker ved MIT, Ronald Rivest, Adi Shamir og Leonard Adleman å gjøre om Diffies ide til et matematisk system, som kunne benyttes til kryptering. Det er dette systemet som i dag kalles RSA etter oppfinnerne og som denne oppgaven skal lage en versjon av.

3.2.1 Den matematiske formulering av RSA-kryptering

RSA kryptografi er baserer seg på modulær eksponering. Formlene som de tre tidligere nevnte kom frem til er gjengitt under i formel 1 og 2[RSA02].

$C = M^e \bmod n$ formel 1

$M = C^d \bmod n$ formel 2

I formlene har en følgende; *C* er kryptert melding, *M* er melding som skal sendes, *e og n* er den offentlige nøkkelen, *d* er den private nøkkelen. En ser at *n* er en del av både kryptering og dekryptering, dette er ikke et problem så lenge *d* er privat og holdt skjult for andre aktører. Modulo operatoren mod er en operasjon som finner resten ved divisjonen av to tall.

3.2.2 Nøkkelgenerering for RSA-kryptering

Nøklene i RSA krypteringen fra formel 1og 2, bestemmes på følgende måte:

1. Først genereres en offentlig nøkkel. Det velges to store primtall *p* og *q*.
2. Produktet av disse kalles *n*. Altså $n = p * q$.
3. Så velges et tall *e* som ikke har felles faktor med $(p-1)*(q-1)$.
4. Tallparet *n* og *e* utgjør den offentlige nøkkelen.
5. Den private nøkkelen *d* regnes ut ved å løse $e = d^{-1} \bmod (p-1)*(q-1)$.

Eksempel 2: Utregning med RSA-formelen.

p=13 og q=11=>n=143, e=7, d=103, velger M=89

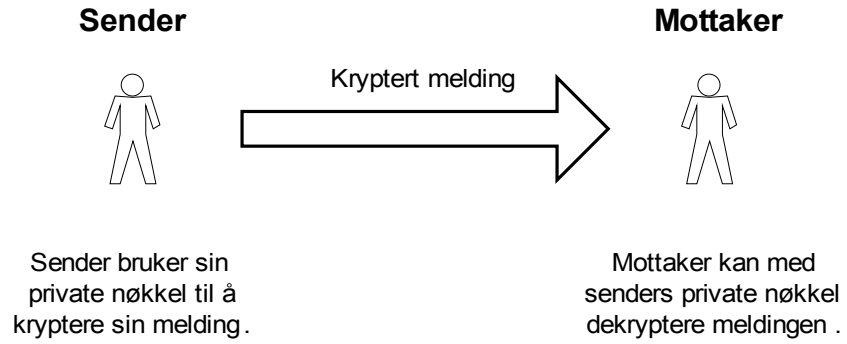
Kryptering; $C=89^7 \bmod 143=67$

Dekryptering; $M=67^{103} \bmod 143=89$

Nøkkelgenerering regnes som en tung operasjon, men det er ikke nødvendig å utføre denne mer enn en gang. De samme nøklene kan brukes mot flere mottakere/sendere siden man aldri utleverer den private nøkkelen. I Eksempel 2 er det vist nøkkelgenerering og utregning med de to RSA formlene.

3.2.3 Autorisasjon med RSA-kryptering

RSA-algoritmen kan også bli benyttet for autorisasjon av klienten. Et eksempel på dette er vist i Illustrasjon 5. Her ønsker senderen å forsikre mottakeren at han er den personen som han gir seg ut for å være. For å overbevise mottakeren krypterer han en melding med sin private nøkkel slik at han får en chiffertekst melding. Mottakeren bruker da senderen offentlige nøkkel til å dekryptere meldingen. Senderen må da ha laget denne meldingen, siden han er den eneste personen som vet om den private nøkkelen.



Illustrasjon 5: Autorisasjon med RSA.

3.3 Metoder og algoritmer

Ved store tall blir RSA kryptografi en tung operasjon å utføre og det trengs gode algoritmer for å bryte den ned til operasjoner som kan utføres enkelt og raskt. Spesielt for implementering i maskinvare. Algoritmene er fra Kocs "*RSA Hardware Implementation*"[KOC95]

3.3.1 Venstre til høyre binær metode

I denne algoritmen brytes potens ned og utføres som en multiplikasjon om gangen., i tillegg utføres en modulo operasjon samtidig. Dette gjør at det ikke er nødvendig med store multiplikatorer eller registre. For å utføre modulær multiplikasjon, finnes flere det algoritmer, blant annet Montgomery og Blakley. De er beskrevet i de to neste kapitlene.

```
Inn: M, e, n
Ut: C: = M^e mod n
1.      if e_{h-1} =1 then C: = M else C: = 1
2.      for i = h-1 downto 0
2a.     C: = (C*C) mod n
2b.     if e_i =1 then C: = (C*M) mod m
3.      return C
```

3.3.2 Blakley-algoritmen

For hver iterasjon i denne algoritmen skal det utføres en dobling av P, eventuelt en addisjon av A og modulo n av P. Dobling av P vil si i kretssammenheng å skifte P et hakk til venstre, altså en svært enkel operasjon. Å legge på A er litt mer krevende, det vil enten ta lang tid eller mye kombinatorikk alt etter hvilken type adderer som benyttes. P mod n, vil si å trekke fra n, helt til n er større enn P.

Ved en nærmere kikk på algoritmen viser det seg at P maksimalt kan bli 3n-3, det betyr at det aldri skal være nødvendig å trekke fra n mer enn to ganger. Det skal maksimalt utføres en addisjon, en skifting og to subtraksjoner.

```
Inn: A, B, n  n >= A
Ut: P:= A*B mod n
1.      P := 0
2.      for i = 0 to k-1
2a.     P := 2P + A*B_{k-1-i}
2b.     P := P mod n
3.      return P
```

3.3.3 Montgomery-algoritmen

Det er Montgomery algoritmen, som oftest blir benyttet for å lage RSA-moduler. Disse er gjerne modifiserte utgaver av Montgomery algoritmen, slik som hos Yingli[YIN01] og Fournaris[FOU05]. Her er en kort presentasjon:

```
1. P := 0
2. for i := 0 to k - 1
2a. Q := (P(0) + x(i)*y(0) ) mod 2
2b. P := (P + x(i)*y + Q*n) div 2
2c. P >= n then P := P - n
3. return P
```

Modifisert venstre til høyre binær metode ved å bruke Montgomerys metode:

```
1. C := R
2. M' := MontMult(M,R^2)
3. for i := h - 1 downto 0
3a. C := MontMult(C,C)
3b. if e(i) = 1 then C := MontMult(C,M')
4. C := MontMult(C,1)
5 return C
```

Montgomery sin algoritme benytter seg av en modifisert versjon av venstre til høyre binær metoden. I MontMult skal det utføres en mod 2 operasjon, en div 2 operasjon, en addisjon og en

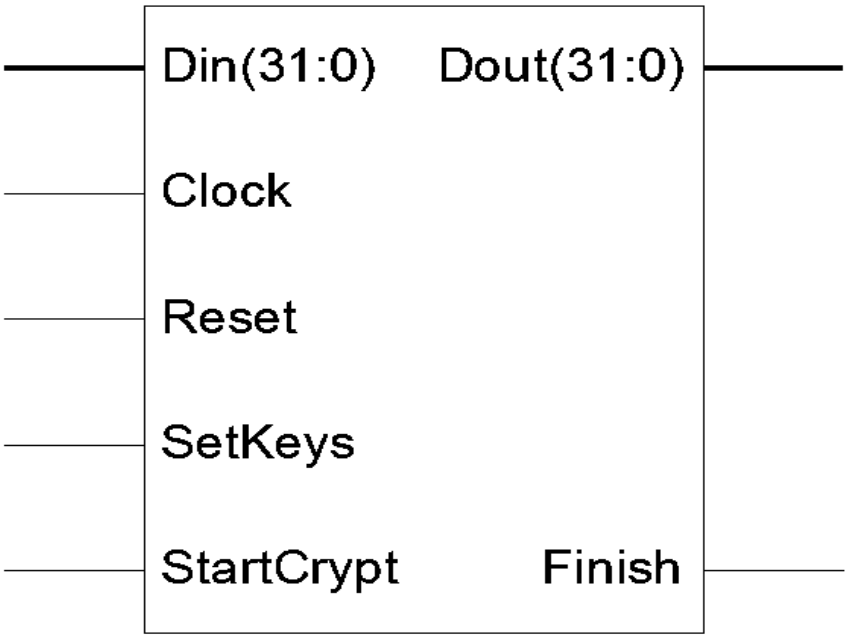
subtraksjon. Mod 2 er svært enkel da svaret er LSB i faktoren. Div 2 vil bli å skifte faktoren en gang til høyre.

3.4 Valg av algoritme

Det er valgt å bruke Blakley algoritmen i denne oppgaven. Dette skyldes at denne så ut til å være raskest å implementere. Med tanke på at dette bare er en del av oppgaven, vil det være hensiktsmessig å rask kunne utvikle en modul.

4 Kretsrealisering av RSA-kryptering

Kretsen er laget på grunnlag av venstre til høyre binær metode og Blakleys algoritme for å utføre modulær multiplikasjon. Modulærmultiplikasjonen er skilt ut som en egen modul; modmult og den er laget slik at den bruker tre klokkeperioder for hver iterasjon av algoritmen.



Illustrasjon 6: ”Blackbox” med inn- og utgangssignaler for systemet.

Illustrasjon 6 viser en såkalt “blackbox” av inn- og utgangene til RSAcore modulen. I Tabell 2 gies det en forklaring på hvordan de forskjellige inn- og utgangsignaler er ment å opptre.

Tabell 2: Spesifikasjonen av RSAcore-modulens inn- og utganger

Signal	Beskrivelse
Clock	Vippene i kretsen skal være klokket på positiv flanke av Clock.
Reset	Vippene i kretsen resettes asynkront når Reset settes lavt
SetKeys	En pulse på SetKeys indikerer at en pakke med konfigurasjon data skiftes inn på Din. Ved de påfølgende klokkeflankene vil resten av pakken skiftes inn.
StartCrypt	En pulse på StartCrypt indikerer at en meldingsblokk skiftes inn på Din og at RSA modulen skal kryptere/dekryptere denne meldingsblokken.
Din	Buss som brukes for å sende inn pakker med krypterte/dekrypterte meldinger ut av kretsen.
Finished	Utgangssignal som er aktivert så lenge RSAcore er klar til å motta nye jobber. Må settes lavt klokkeperioden etter at en puls er sent ut på SetKeys eller StartCrypt.

RSA modulen har en 32bits inngang, hvilket betyr at den må motta data og nøkler i 32bits serielle datapakker. RSA modulen skal kommunisere med omverdenen på følgende måte; først sendes initialiseringsdata, det vil si nøkler, inn til kretsen. Så vil meldingen som skal krypteres/dekrypteres komme. Når dette er gjort sendes data ut av kretsen igjen. I tillegg til å fortelle at kjernen er ledig så forteller *Finished* signalet neste gang det går høyt etter en puls på *StartCrypt* at det kommer behandlete data ut.

For adderer ble det bare skrevet ”+” i koden. Ved syntetisering er det da opp til synteseprogrammet å velge hvilken adderer som skal benyttes. Dette er også noe som kan velges enten ved import av bibliotek i VHDL koden eller i synteseverktøyet. Et par eksempler på adderere vil være Rippel Carry Adderer og Carry Look Ahead, den første vil ta lite areal mens den andre vil være raskere.

4.1 Kontrollenheten

RSAcore-modulen består av en tilstandsmaskin med fire registre. To registre for nøklene, et dataregister for data som skal krypteres/dekrypteres og et dataregister til å holde på svaret. I tillegg er det et utgangsregister for å sende svaret ut av kretsen. Kort fortalt så henter kretsen inn nøklene ved *SetKeys* og legger disse i registrene, *keyE* og *keyN*, ved *StartCrypt* legger den meldingen i registeret, *dataM*, og benytter seg av modmult-modulen for å kryptere denne i henhold til venstre til høyre binær metoden.

For dypere innblikk i kretsens virkemåte anbefales det å lese koden med kommentarer i Vedlegg B VHDL kode, her følger en kort beskrivelse av tilstandene:

IDLE:

I tilstanden IDLE så venter den på et av kontrollsignalene, *StartCrypt* og *SetKeys*. Ved en puls på inn *SetKeys* så skifter den inn første del av nøklene via datainngangen *Din*. Datainngangen er på 32bit og minimum bitbreddde er på 64bit. Ved en puls på *StartCrypt* så skifter den inn første del av datapakken. Telleren, *index*, settes til riktig verdi for å kontrollere at tilstandsmaskinen blir stående i de neste tilstandene, *INIT* og *REC_DATA*, lenge nok til å motta nøklene eller datapakken.

INIT:

Her skiftes nøklene inn i registrene, keyE og keyN. Når telleren, index, når null går den tilbake til IDLE.

REC_DATA:

Her skriftes meldingen inn i registret, *dataM*. Den er ferdig når *index* er null. Da skifter tilstanden til arbeidstilstanden *S1* og *index* resettes til bitbredden. Startverdien for første gjennomgang av algoritmen settes også.

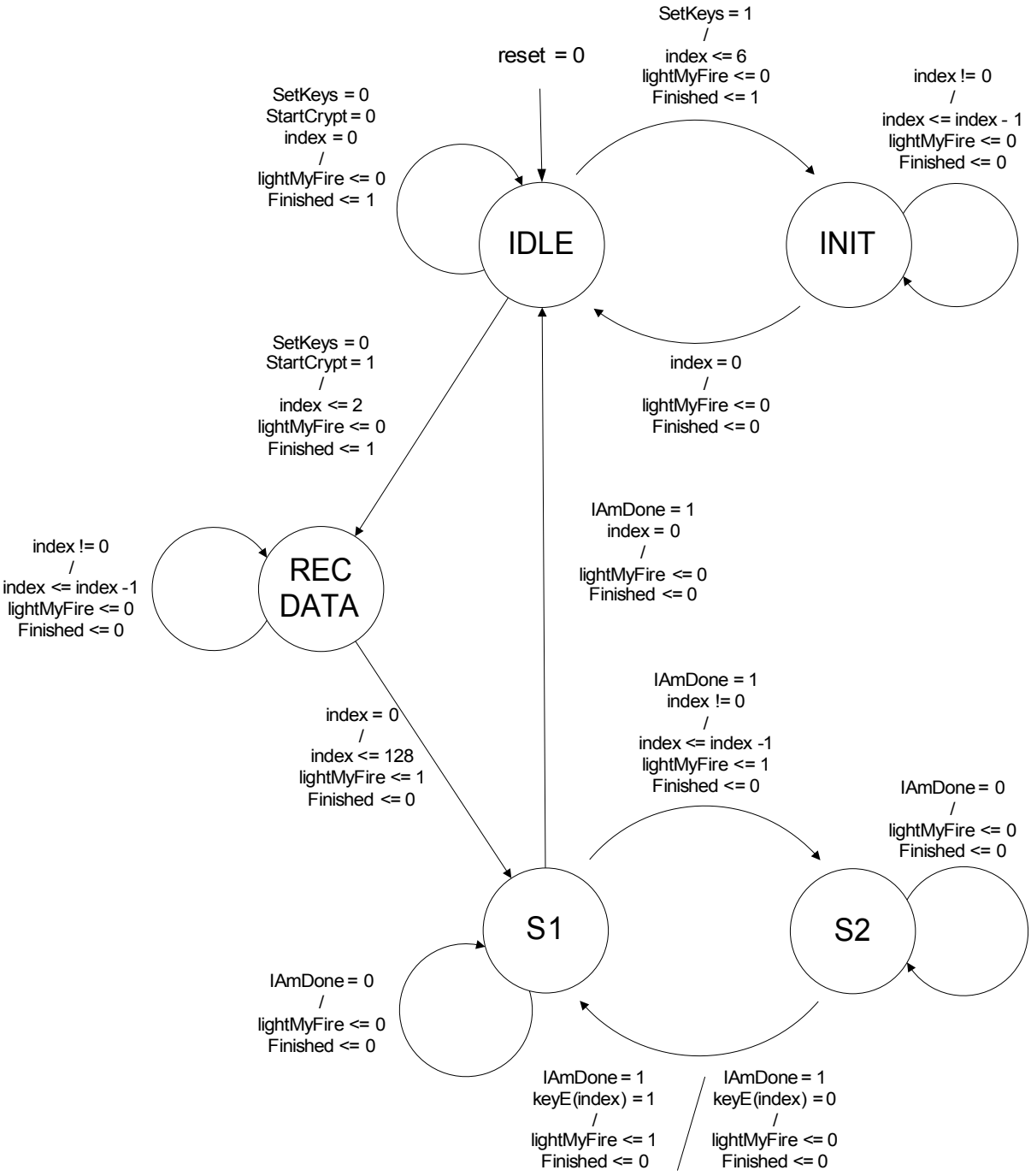
S1:

Første arbeidstilstand, tilsvarer linje 2a i algoritmen, det vil si at den via modulen modmult, utfører *dataC * dataC mod (keyN)*. Går så direkte videre til arbeidstilstanden *S2*.

S2:

Andre arbeidstilstand, tilsvarer linje 2b i algoritmen. Venter her på at modmult er ferdig med operasjonen som ble initiert i *S1*. Skifter *keyE* en gang. Hvis nåverdien av *keyE (index)* er en så utfører den *dataC * dataC mod (keyN)*. Går uansett rett til arbeidstilstand *S1* såfremt *index* ikke er null. Hvis *index* er null legges *dataC* inn i utgangsregisteret og tilstandsmaskinen går tilbake til *IDLE.*

Kontrollsignalet *Finished* er lav for alle andre tilstander enn IDLE. *Dout* er koblet til de 32 minst signifikante bittene i utgangsregisteret. Utgangsregisteret skifter 32bit for hver klokkeperiode i *IDLE*, *INIT* og *REC_DATA*. Det vil si at kjernen alltid er klar til å ta imot nye data eller nøkler når *Finished* er høy. Illustrasjon 7 viser overgangene mellom de ulike tilstandene.



Illustrasjon 7: Tilstandsdiagram for RSAcore.

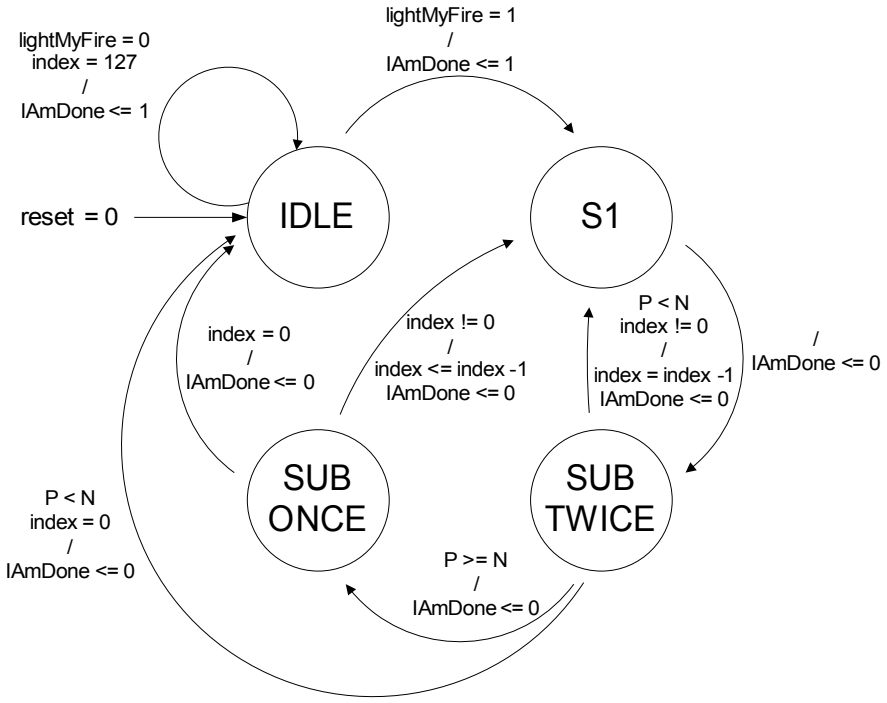
I tilstandene *S1* og *S2*, inkluderes modmult modulen. Det vil derfor være forklarende å se Illustrasjon 8 for kontrollsignalene i disse to tilstandene.

4.2 Modulær multiplikasjon

Modulen modmult utfører modulær multiplikasjon ut fra Blakleys algoritme. Den bruker tre klokkeperioder per iterasjon av algoritmen. Løsningen er i utgangpunktet tenkt å bruke lite areal. I Tabell 3 er inn- og utgangssignalene til modulen beskrevet

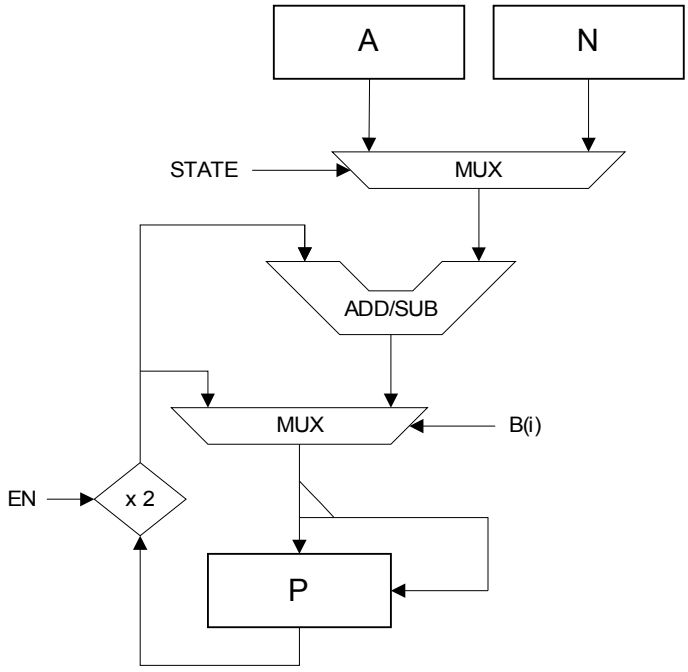
Tabell 3: Forklaring på inn- og utgangene til modmult-modulen

Signal	Beskrivelse
Clock, reset	Samme som for RSAcore
factorA, factorBin	Datainngangene for faktor A og faktor B. Faktor A må våre påtrykt under hele operasjonen.
moduloOperator	Modulo operanden. Må være påtrykt under hele operasjonen.
lightMyFire	Startsignal for kretsen.
IamDone	Stoppsignal



Illustrasjon 8: Tilstandsdiagram for modmult.

Arkitekturen vist tilstandsdiagrammet i Illustrasjon 8 og flytdiagrammet i Illustrasjon 9. utfører Blakleys algoritme sekvensielt. Først legger den til faktor *A* på *P* hvis *B(index)* er lik en, deretter trekker den fra *N* hvis *P* > *N*, opp til to ganger. I *S1* legges *A* til *2P*. Neste tilstand, *SUB TWICE*, trekker *N* fra *P*, hvis svaret er positivt legges det inn i *P* og neste tilstand blir *SUB ONCE*, hvis svaret er negativt blir neste tilstand *S1* og *P* forblir det samme. I *SUB ONCE* trekkes *N* fra *P* en gang til og legges inn i *P* hvis svaret er positivt. Neste tilstand blir *S1*. Hvis *index* er null når tilstandsmaskinen skal returnere til *S1* så går den til *IDLE* i stedet, da er den ferdig. Kommentert kildekode ligger i Vedlegg B VHDL kode.



Illustrasjon 9: Flytdiagram for modmult

5 Testing av RSA-modul

Skriving, kompilering og simulering av kode ble utført i ModelSim SE 5.6a. Syntetisering ble utført i Xilinx ISE 8.2i sitt synteseverktøy Xilinx Synthesis Technologi (XST).

5.1 Testbenker og simuleringsresultater

Det ble laget to testbenker, en for hele RSAcore og en for modMult modulen. Disse ligger vedlagt i Vedlegg E Testbenker.

5.1.1 Simulering av modmult

I testbenken for modmult er modulen blitt testet tre ganger, dette er gjengitt i Tabell 4 og Tabell 5. De tre inngangene *Faktor A, Faktor B* og *Modulo operator* er først blitt påtrykk små verdier henholdsvis A, A og D, dette er heksadesimale verdier. Til utgangene er det beregnet et forventet produkt i Matlab og Simulert produkt er resultatet av beregningen til kretsen; *Simulert produkt = Faktor A * Faktor B mod(Modulo operator)*. I test nummer to er tre 128 bits tall påtrykk inngangene. Test nummer tre påtrykker også tre 128 bits tall på inngangene, men her er Modulo operatoren større enn faktor A. Dette bryter betingelsen for Blakleys algoritme, se kapittel 3.3.2.

Tabell 4: Påtrykte verdier for modmult

nr	Faktor A	Faktor B	Modulo operator
1	A	A	D
2	001E11180B2584C0D0B240E1662FBB3D	8003350C2E23DD711667F27D48ADD81D	8000424B7EC569CEDB8B4FA67785F97E
3	8785EC0D6D5FA61DDA41D2E0C4CD25C0	87247D31A39B21DFCD8826FB6B95FBC6	0000000000000DCC688517346C05F97E

Tabell 5: Resultat for modmult

Nr	Forventet produkt	Simulert produkt
1	1	1
2	73BA30B9116929D997E5CB84B5E6AFD7	73BA30B9116929D997E5CB84B5E6AFD7
3	3A70505A97409978D6	35CEED7F3DD358B9D4392F621ACED0A98

5.1.2 Simulering av RSAcore

Testbenkene for hele systemet med RSAcore og modmult. Det legges først inn nøklene *KeyE* og *KeyN*, for første melding *M* sendes til modulen, som en kan se fra Tabell 6. I simulering nummer to beholdes nøklene, mens det legges inn en ny melding. Resultatene fra simulering ses i Tabell 7, disse er blitt benyttet i simulering nummer tre og fire. Før simulering nummer tre blir KeyE byttet ut. I simulering tre og fire sendes resultatene fra simulering en og to inn og man får ut de opprinnelige meldingene.. Her kan da KeyE ses på som den offentlige nøkkelen i simulering en og to, mens den er den private nøkkelen i tilfelle tre og fire.

Tabell 6: Påtrykte verdier for RSAcore

Nr	KeyE	KeyN	M
1	00000000000000000000000000010001	819DC6B2574E12C3C8BC49CDD79555FD	0AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2	00000000000000000000000000010001	819DC6B2574E12C3C8BC49CDD79555FD	0C0FFEE0C0FFEE0C0FFEE0C0FFEE0000
3	819D451512390089590AF8BB46DB0001	819DC6B2574E12C3C8BC49CDD79555FD	7637EA28188632D8F2D92845DB649D14
4	819D451512390089590AF8BB46DB0001	819DC6B2574E12C3C8BC49CDD79555FD	30070EBE021626FA3380A7192C956B6D

Tabell 7: Resultater for RSAcore

Nr	Simulert produkt
1	7637EA28188632D8F2D92845DB649D14
2	30070EBE021626FA3380A7192C956B6D
3	0AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
4	0C0FFEE0C0FFEE0C0FFEE0C0FFEE0000

I Vedlegg F Tidsdiagram finnes deler av bølgediagrammene, med kommentarer, for både modMult og RSAcore. Det er vist overgangen mellom to pakker for RSA modulen og for modmult er det følgende fire; innlegging av nøkler, resultat av beregning, hvordan index virker og hvordan delproduktene forandrer seg for hver av tilstandene.

5.2 Syntese

I Xilinx ble parameterne satt til å passe FPGA'en på utviklingskortet, se Illustrasjon 10. FPGA'en ble satt til å være Spartan 3 med enhetsnummer XC3S1000, gitt i databladet[XIL06]. Pakke ble satt til FT256, mens hastighet er satt til -4. Dette er også spesifisert i databladet til Spartan 3 familien.

Property Name	Value
Product Category	All
Family	Spartan3
Device	XC3S1000
Package	FT256
Speed	-4
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim-SE VHDL
Enable Enhanced Design Summary	☑
Enable Message Filtering	☐
Display Incremental Messages	☐

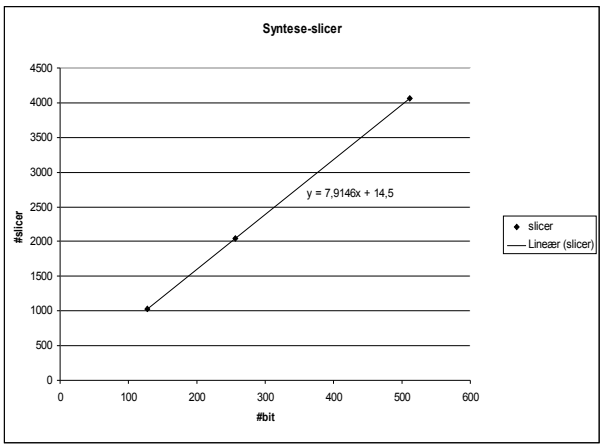
Illustrasjon 10: Parametere valgt for syntetisering i Xilinx XST.

Under i Tabell 8 er de viktigste resultatene fra syntetiseringen gjengitt. Her er hele systemet syntetisert med modmult arkitekturen inkludert i RSAcore arkitekturen. Minimums frekvens for hele systemet er beregnet til å være 59,157MHz. Antall klokkesykluser kan variere fra 32768-98304, dette grunnet utregningen av den modulære eksponeringen. Den kan være ferdig etter en utregning eller måtte gå gjennom utregning tre ganger.

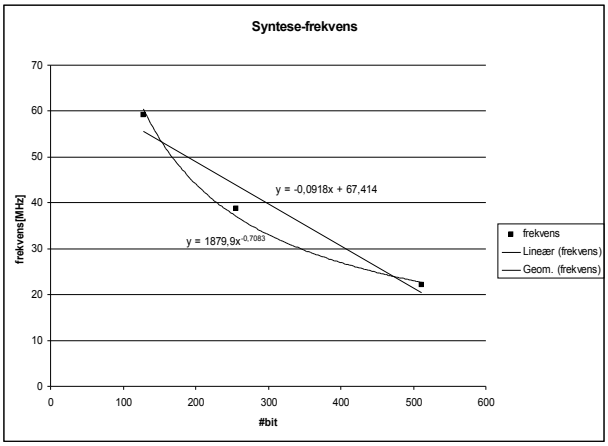
Tabell 8: Utdrag fra synteserapport

Struktur	RSAcore
Frekvens	59.157MHz
#klokkesykler(min)	32768
#klokkesykler(maks)	98304
#Slice	1026 (13%)
#LUT	1955 (12%)
#Vipper	918 (5%)
Throughput	77-231kbps

Arealbruk er beregnet til å være 1026 slicer, noe som er 13% av tilgjenglige logiske blokker. Look-up Table(LUT) og vipper er en del slicene, av disse er det benyttet 12% og 5%. For mer detaljer fra syntesen, se forøvrig synteserapport fra Xilinx XST lagt ved i Vedlegg A Synteserapport.



Illustrasjon 11: Arealøkning med 128-, 256- og 512 bitskryptering.



Illustrasjon 12: Frekvensreduksjon med 128-, 256- og 512 bitskryptering

I Illustrasjon 11 og Illustrasjon 12er kretsen syntetisert med 128-, 256- og 512 bitskryptering. Fra Illustrasjon 11 sees at antall slicer øker tilnærmet lineært. I Illustrasjon 12 er frekvensen plottet mot størrelsen på krypteringen. Her synker frekvesen etter en geomtrisk funksjon. Det er gjort forsøk med 1024 bitskryptering, men kretsen blir da større en tilgjengelige antall slicer som blir oppgitt til å være 9124.

6 Emulering på utviklingskortet

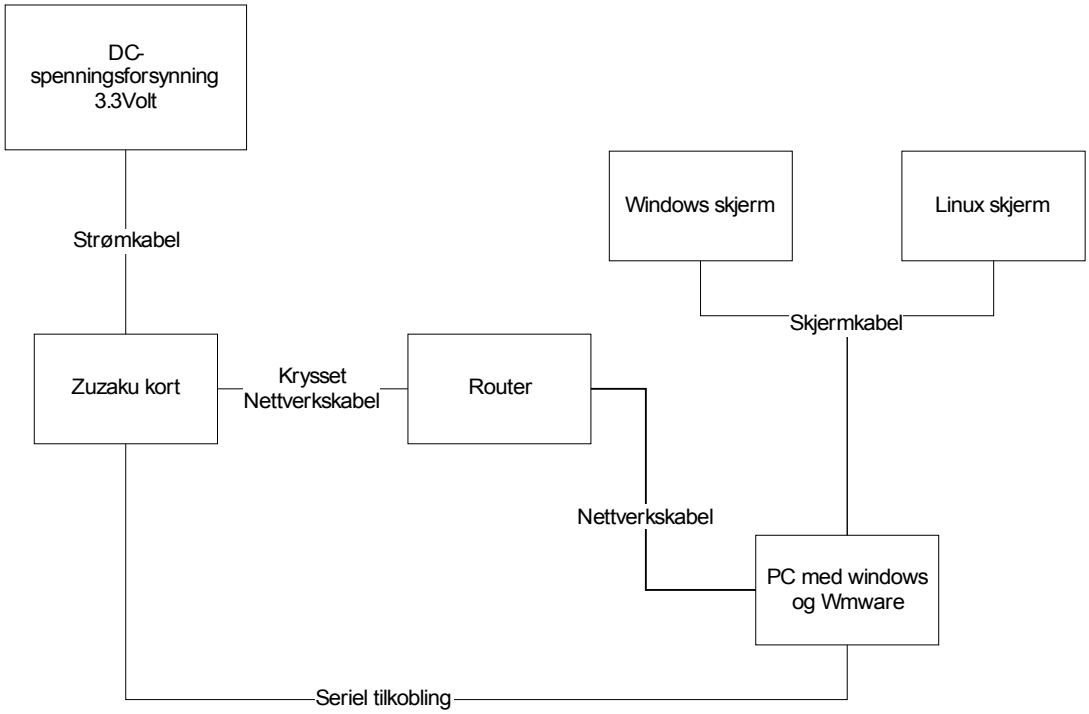
Her i denne delen vil det bli beskrevet et arbeidsoppsett for å emulere den ferdig simulerte og syntetiserte RSA modulen på AHEADs utviklingskort. Arbeidet med dette har bydd på store utfordringer og er ikke blitt ferdigstilt.

6.1 Beskrivelse av utviklingsprogrammer

For utvikling til Suzaku kortet er det nødvendig med to operativ systemer. Windows trengs til å kjøre programmene Xilinx ISE/EDK, ModelSim og Lbplay2. For å lage image- og bit-fil er det kun utviklingsverktøy i Linux som er tilgjengelig. Det er også slik at disse verktøyene bare funger med enkelte versjoner av Linux. Dette skyldes at de er utviklet for den gamle Linux kjernen 2.4.x, så bare eldre versjoner som RedHat9 vil kunne kjøre programmene. Beste løsningen vil nok trolig være å benytte en Windows maskin med mye minne(2Gb) og kjøre VMware. På hjemmesidene til Atmark Techno Development Enviroment[ATD07] kan det lastes ned et VMware image, *atde-20061227.zip,* hvor et utviklingsmiljø er ferdig. Denne pakken kjører Debian som operativsystem.

Fra Illustrasjon 13 ser en at i tillegg til programvare vil man trenge

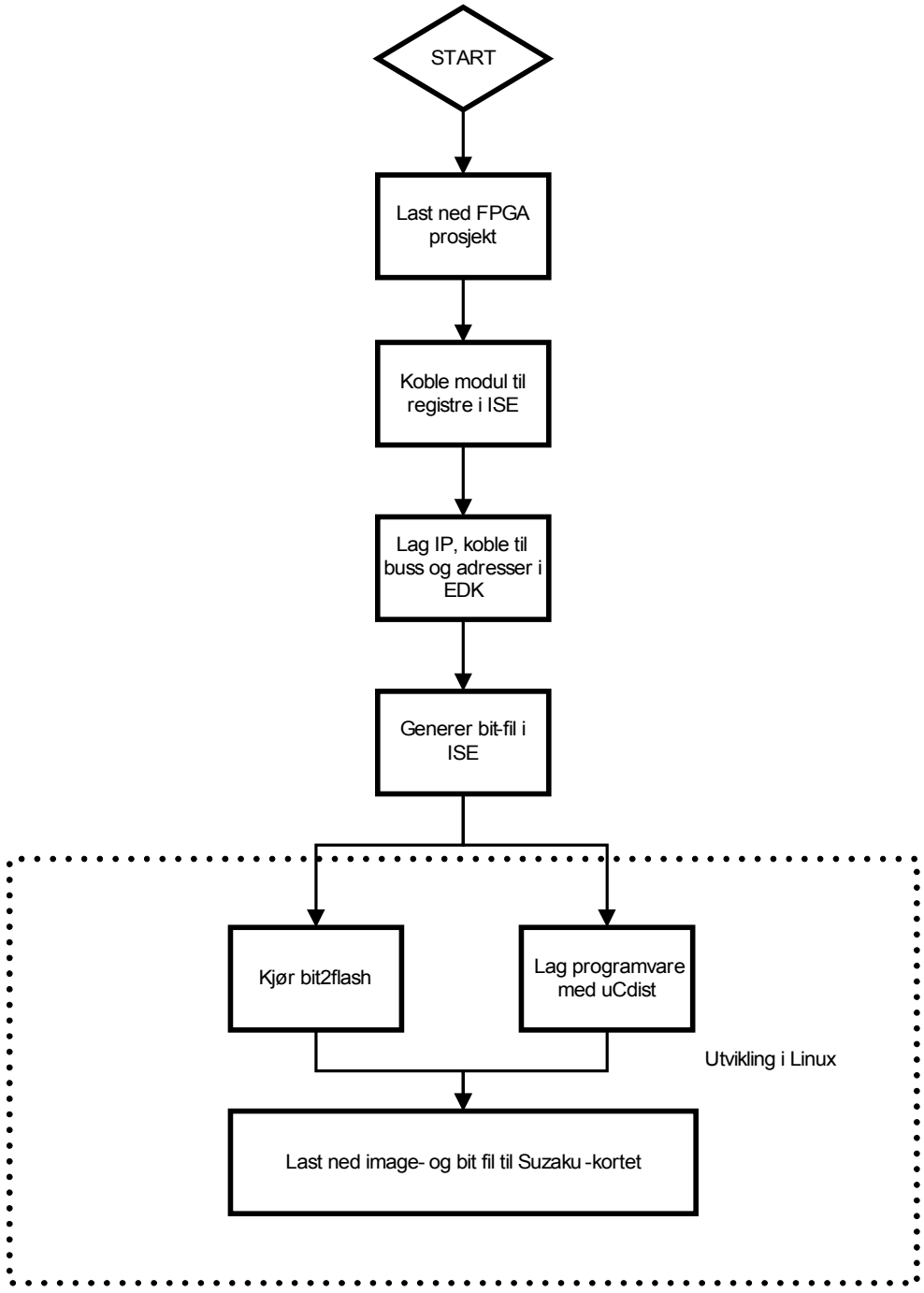
- 3.3Volts spenningsforsynning
- seriell tilkoblingskabel
- nettverkskabel
- krysset nettverkskabel
- router
- JTAG kabel



Illustrasjon 13: Tilkobling av Suzaku-kortet (JTAG kabelen er ikke tegnet inn).

6.2 Designflyt for arbeidet med Suzaku-kortet

For å lage bit- og image-fil ble adderer-eksemplet i Arntsen sin masteroppgave[ARN06] fulgt, i tillegg ble Softwaremanualen benyttet[ATM05]. Fra hjemmesiden til kortet ble FPGA prosjektet til Xilinx 8.2i lastet ned slik en kan se fra Illustrasjon 14.



Illustrasjon 14: Designflyt for utvikling på Suzaku-kortet.

En kobler så til modulen som er utviklet til FPGA prosjektet. Går inn i EDK for å lage en egen Intelectual Property(IP) og kobler til buss og adresser. Tilbake i EDK genereres en bit-fil, som overføres til utviklingsmiljøet i Linux. Her brukes bit2flash for at bit-filen skal passe kortet.

For å lage en image-fil måtte en installeres to utviklingspakker i ATDE:

microblaze-elf-tools-20040315.tar.gz

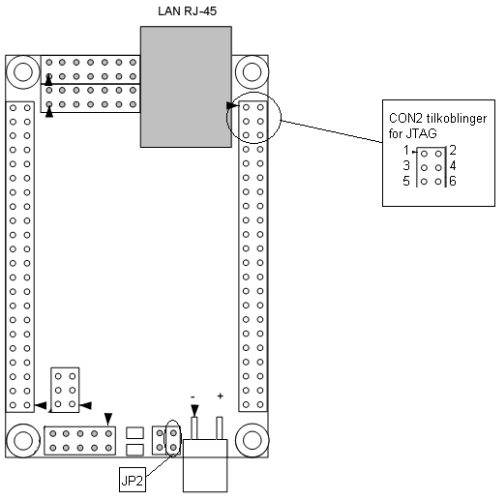
uClinux-dist-20040408-suzaku4.tar.bz2

Før å overføre et nytt image til Suzaku kortet ble det satt opp en Apache html-tjener. Hvordan dette gjøres, er beskrevet i Arntsens masteroppgave. For tilkobling med en SSH-klient er det en fordel med statisk IP-adresse for eksempel 192.168.1.103, slik en kan finne kortet. Med anbefalte minicom kommer bootloaden automatisk opp og man kan koble seg til.

For å laste ned bit- og image-fil ble netflash benyttet. Ved å se på klokkeslett og dato for image, når kortet starter opp, ser man at et nytt image er lastet opp. Ved å kjøring av adderen fikk man kun svaret 0. Man ser ikke umiddelbart om bit-filen er korrekt nedlastet. Det skal være mulig å koble JTAG'en til kortet og lastet ned innholdet på FPGA'en, slik at en kan se om nytt innhold er blitt lastet opp. En vet da om det er nedlastingen eller bit-filen det er noe galt med. Det har ikke blitt tid til å prøve ut dette.

6.2.1 Gjenoppretting av flashminnet

Når man flasher minnet med ny bit- eller imagefil, hender det noen ganger at bootloaderen ikke vil starte opp igjen. Ved å benytte .mcs fila og Lbplay2 fra FPGA prosjektet som kan lastes ned fra hjemmesiden til kortet[ATM07], kan man gjenopprette bootloaderen. Dette er beskrevet i kap10.2 i Hardwareguiden[ATM04]. Til koblingen av JTAG til kontaktpunkt (CON2) er vist i Illustrasjon 15 som er beskrevet i kapittel 7.6. Det går ikke klart frem hva som er pinne 1-6 slik denne illustrasjonen viser.

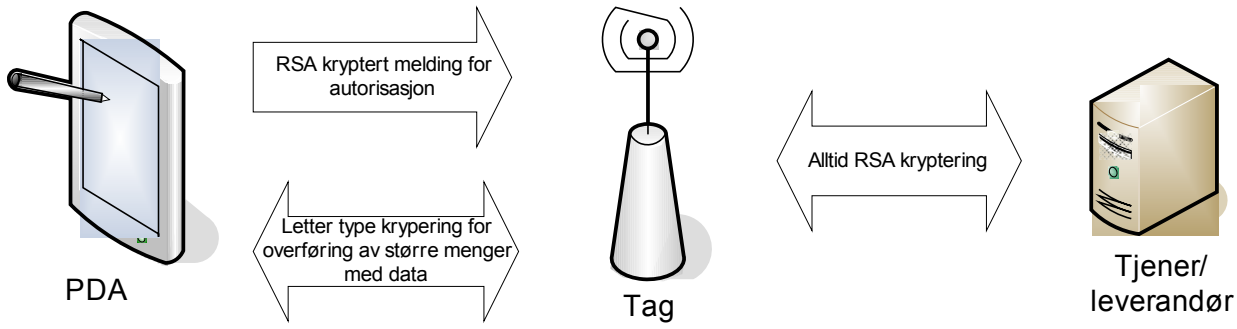


Illustrasjon 15: Tilkobling av JTAG til Suzaku-kortet.

7 Fremtidig arbeid

For å få en ferdig prototyp for kryptering til AHEAD prosjektet gjenstår det enda mye arbeid. Med kunnskap opparbeidet om kortet av prosjektstudentene høsten 2006 bør det være mulig å få testet en bit-fil med den utviklete RSA modulen. Det bør i tillegg utvikles en løsning med en mindre ressurskrevende kryptering for klienten. Ved overføring av informasjon mellom bruker vil man kunne autorisere brukeren med RSA kryptering for så å bruke den lettere kryperingen til selve overføringen.

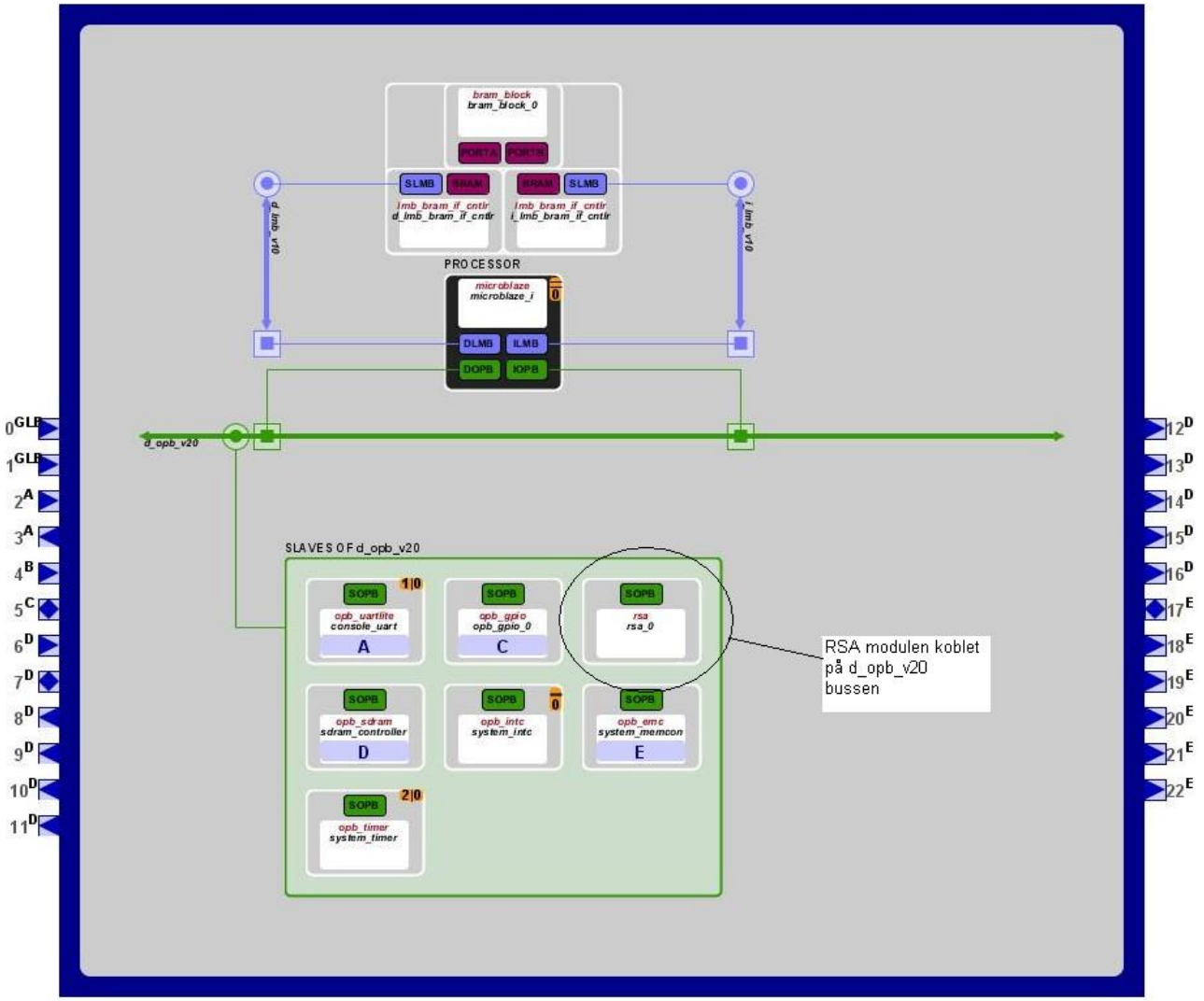
For hele tiden å ha en sikker kommunikasjon over internett mellom tjeneren og tagen vil disse hele tiden kunne benytte RSA-kryptering. Tagen vil da benyttes til å gjøre om RSA kryptert melding til den lette krypteringen. Dette scenarioet er forsøkt vist i Illustrasjon 16. Denne vil gi mindre belastning på den håndholdte enheten.



Illustrasjon 16: Et tenkt løsning for kryptering.

7.1 Register tilkobling av RSA-modulen

I teorien bør man kunne bygge ut adder eksemplet med en registerbank og koble til denne. En vil kunne lage en Intelectual Property og benytte seg av OPB -bussen. Da vil reistrene til RSA modulen kobles til adressene på bussen. De ledige adressene ligger fra 0x81000000 og utover. I Illustrasjon 17 er det vist en RSA-IP koblet på v_opb_20 bussen.



Illustrasjon 17: RSA-modulen koblet til d_opb_v20 bussen.

7.2 Programvaren på microBlaze

Denne må kunne ta imot meldinger og dele disse opp i mindre blokker, gjerne på 128 som her er blitt benyttet. Programvaren må også styre kontrollsignalene til hardwaren(Spartan 3 FPGA'en). Ettersom kommunikasjonen er på 32 bit vil blokkene deles opp i fire igjen. Programvaren vil først være nødt til å sende inn pakker med to nøkler før selve meldingen kan sendes.

Det må i programvaren også mottas dataen som blir lagt på utgangen fra FPGA'en og sende den til riktig mottaker.

8 Diskusjon

Denne oppgaven startet med å tenke litt rundt hvordan man skulle utvikle det tenkte klient-tjener-systemet. Uten mye erfaring med å sette opp et slikt system ble det satt opp en løs skisse til hvordan kommunikasjonen skulle foregå, slik det er vist i Illustrasjon 3. Det ble bestemt å starte på delen med RSA kryptering i maskinvare. Ut i fra de algoritmer som er blitt presentert, har det blitt utviklet en modul basert på venstre til høyre binær metoden og Blakleys algoritme. Dette er den som ble antatt å være enklest å realisere, og valgt grunnet tidspresset.

Ut i fra de testbenkene som er laget for 128 bitskryptering, ser kretsen ut til fungere ved simulering i ModelSim og den lar seg syntetisere ved hjelp Xilinxs XST. Riktignok blir siste svaret galt i simuleringen av modmult, dette stemmer med kravene som sier at modulo operanden må være større enn faktor A. Dette vil ikke ha noe å si, når kretsen benyttes sammen med RSAcore. Maksimal klokkefrekvens vil nok være veldig avhengig av hvilken teknologi det syntetiseres mot, men med utviklingskortet fra Suzaku blir minimum klokkefrekvens 59.157MHz.

Testbenkene kunne vært laget langt mer avanserte. Det ville vært naturlig å sende en fil med melding i, kryptert denne til chiffertekst og dekryptert den til klartekst igjen.

Det er benyttet plusstegn i koden og det er dermed opp til syntheseverktøyet og velge hvilke adderer som skal benyttes. Det er også grunn til å tro at dette spesifikke kortet hadde innebygde funksjoner som muliggjorde rask addisjon. Ved å benytte andre bibliotek, enten ved syntetisering eller i koden, kan det benyttes akkurat de addisjons- og subtraksjonsløsningene spesifikasjonene for areal og hastighet krever.

Ressursbruken på kortet er akseptabelt da kretsen bare benytter 13% av de tilgjengelige ressursene. Ettersom softwaren sannsynligvis ikke kan levere nye datapakker, rett etter at kretsen er ferdig med å behandle en annen, kan en mest sannsynlig fjerne utgangsregisteret. Utgangsregisteret kunne uansett kanskje ha blitt kombinert med mellomsvar registeret dataC.

Selv om andre algoritmer ikke ble forsøkt realisert, spesielt Montgomery ville vært interessant, så er arkitekturen fullgod løsning til å arbeide videre med. En throughput på 77 kbps vil i de fleste tilfeller være nok, unntaket kan være direkteoverføring av video med høy bildekvalitet. Det vil i Montgomerys metode være nødvendig å utføre MontMult like mange ganger som ModMult i Blakleys, så hvis frekvens er satt så vil ikke den bli noe raskere

Veldig mye av arealet i kretsen går med til vipper, som måtte vært med uansett, så det vil heller ikke være veldig mye å spare der. Frekvens derimot kunne blitt høyere ved Montgomerys metode. Ut over dette er det vanskelig å anta noe mer da det ikke er realisert noen versjon av Montgomerys

metode. Kretsen kan skaleres ved hjelp av den generiske verdien, bitbredde, i koden til 32*n bit hvor n er et heltall større enn en. De største elementene i kretsen er registre og adderere, addererne er også de tregeste.

Tabell 9: Sammenlikningstabell for 1024-bitskryptering.

Design	CLBs	Clock frequecy[MHz]	Troughput[bit/sec]
Fournaris	7873	129	7,17M
McIvor	22076	93,34	4,66M
Daly	10369	49,63	45,8K
Mazzeo	2902	32,36	342,4K
Hervig	**2283?**	**-**	**-**

Det har blitt gjort forsøk på å skalere kretsen opp til 1024 bitskryptering. Hvis en ser i Tabell 9, som er hentet fra Fournaris artikkel[FOU05], ser en at resultatene er meget sprikende. De fire realiseringene hentet fra artikkelen er alle realisert ut i fra modifiserte utgaver av Montgomery-algoritmen og er derfor ikke direkte sammenlignbare med kretsen som her er realisert. I databladet [XIL06] står det at 1 CLB = 4 slicer, noe som tilvarer 2282,75 slicer for denne kretsen. Det er mange mulig feilfaktorer her, så det vil ikke bli lagt vekt på denne sammenlikningen i konklusjonen.

I forsøket på å emulere kretsen på utviklingskortet, har det vært en del problemer med utviklingsprogrammene. Dokumentasjonen fra leverandøren Atmark er i utgangspunktet på japansk, og bare Hardwaremanualen og Softwaremaunalen er oversatt til engelsk, noe som har gjort feilsøking vanskelig. Kort oppsummert har man kommet frem til at ATDE pakken med VMware er det enkleste oppsettet for arbeidet med utvikling på kortet.

Det kunne vært spart en del på arbeidet med disse utviklingsprogrammene ved et tettere samarbeid med prosjektstudentene. Her kunne blant annet tipset om Tsois *SoC Degsign* stensiler[TSO04], spart mye tid hvis det hadde komme litt før. Generelt vil det være en fordel for videre utvikling av AHEAD prosjektet, at det taes vare på oppbygd kompetanse rundt utviklingskortet.

Det er ikke gjort forsøk på å forbedre hastigheten på RSA-modulen. Dette kunne eventuelt vært gjort etter at den prototyp hadde vært ferdigstilt.

9 Konklusjon

RSA krypteringsalgoritmen er blitt implementert i maskinvare. Dette er blitt gjort ved hjelp av venstre til høyre binær metoden og Blakleys algoritme. Ut ifra simulering og syntese er det ikke påvist noen feil ved kretsene. Ved 128bits kryptering/dekryptering har kretsen en throughput på 59kbps, mens den har et areal som utgjør 13% av tilgjenglige slicer.

Sammenlikning med andre implementasjoner viser at frekvensen og throughputen er tilfredsstillende. Arealet ser ut til å øke lineært ved en oppskalering, men det vil ikke være plass til en 1024 bitskryptering med denne løsning på utviklingskortet.

Det kan anbefales å benytte ATDE-pakken med VMware som arbeidsoppsett, for arbeid med utviklingskortet. En har da fordelen av å ha alle filer på samme datamaskin. For videre arbeid med et RSA krypterings-system må det gjøres en mer inngående analyse av hvilken type kryptering som skal benyttes av klienten.

Referanseliste

[ARN06] Arntsen, S. R, FPGA-plattform for AHEAD

[ATD07] Hjemmeside til Atmark Techno Development Environment, http://download.atmark-techno.com/atde/index.html.en

[ATM04] Atmark Techno, Inc., “SUZAKU Hardware Manual”, June 15, 2004

[ATM05] Atmark Techno, Inc., “SUZAKU Software Manual”, January 31, 2005

[ATM07] Atmark Techno sin hjemmeside for utviklingskortet sine filer http://suzaku-en.atmark-techno.com/downloads/all

[FOU05] Fournaris, A. P og Koufopavlou, A Nev RSA Encryption Architectuere and Hardware Implementation based on Optimized Montgomery Multiplication.

[KOC05] Koc, C. K, RSA Hardware Implementation, RSA Laboratories, August 1995

[XIL06] Xilinx, Spartan-3 FPGA Family: Complete Data Sheet -Product Specification, April 26, 2006

[RSA02] RSA Laboratories, PKCS #1 v2.1: RSA Cryptography Standard, June 14, 2002

[TSO04] Tsoi, K. H, SoC Design

[YIN01] Yingli F, A New RSA Cryptosystem Hardware Implementation Based on High-Radix Montgomery's Algorthm.

Vedlegg

Vedlegg A Synteserapport

```
=========================================================================
*                        Advanced HDL Synthesis                         *
=========================================================================

Macro Statistics
# FSMs                                                 : 2
# Adders/Subtractors                                   : 4
 130-bit adder                                         : 1
 130-bit subtractor                                    : 1
 7-bit subtractor                                      : 1
 8-bit subtractor                                      : 1
# Registers                                            : 918
 Flip-Flops                                            : 918

Final Register Report

Macro Statistics
# Registers                                            : 918
 Flip-Flops                                            : 918

Device utilization summary:

 Number of Slices:                     1026  out of   7680    13%
 Number of Slice Flip Flops:            918  out of  15360     5%
 Number of 4 input LUTs:               1955  out of  15360    12%
 Number of IOs:                          69
 Number of bonded IOBs:                  69  out of    173    39%
 Number of GCLKs:                         1  out of      8    12%

   Minimum period: 16.904ns (Maximum Frequency: 59.157MHz)
   Minimum input arrival time before clock: 8.116ns
   Maximum output required time after clock: 9.874ns
   Maximum combinational path delay: No path found
```

Vedlegg B VHDL kode

RSACore

```
--Del av masteroppgaven RSA krypterings-system for AHEAD
--Institutt for elektronikk og telekommunikasjon
--ved Norges Teknisk-Naturvitenskapelige Universitet, NTNU
--Utført av: Vidar Eikrem Hervig, høst 2006
--VHDL 1993 syntax
--*********************************************************************************
--*RSA-kjerne. Utfører krypteringsalogritmen C = M^E mod N.                       *
--*Tar inn 32-bits blokker med data eller parametre. Sender ut i 32-bits blokker også*
--*Den generiske verdien bitbredde bestemmer størrelsen på krypteringen           *
--*Benytter seg av ekstern modul for modulærmultiplikasjon.                       *
--*Henter inn E og N under initsiering. Under utførelse av algoritmen hentes M inn og*
--*C sendes ut.                                                                   *
--*Alle vipper trigger på positiv klokkeflanke og har aktiv lav asynkron reset.   *
--*********************************************************************************
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity RSAcore is
generic( bitbredde : natural := 128);
port(
Clock: in std_logic; --klokkesignal
Reset: in std_logic; --asynkron aktiv lav reset
SetKeys: in std_logic; --initsieringssignal, kommer samtidig med initsieringsdata.
StartCrypt: in std_logic; --Dinitsieringssignal, kommer samtidig med data.
Din : in std_logic_vector(31 downto 0); --32bits seriell Dinngang.
Dout : out std_logic_vector(31 downto 0); --32bits seriell datautgang.
Finish : out std_logic); --Kjernen er ferdig med tildelt oppgave og klar for en ny.
end;
--end entity RSACore;
architecture behaviour of RSAcore is
signal keyE, keyN: std_logic_vector((bitbredde -1) downto 0); --nøkler. Kalt E og N
signal dataM, dataC: std_logic_vector((bitbredde -1) downto 0);
    --databuffer. dataM er data som skal krypteres/dekrypteres og dataC er resultatet.
signal register_c: std_logic_vector((bitbredde -1) downto 0);
  --utgangsbuffer. Tar inn siste iterasjon av dataC og skifter dette ut på Dout.
signal factorB: std_logic_vector((bitbredde -1) downto 0); --ene faktoren til modmult.
signal product: std_logic_vector((bitbredde + 1) downto 0); --resultatet fra modmult
signal index: natural range 0 to bitbredde;
 --teller for å ha kontroll på antall iterasjoner av forskjellige deler av algoritmen.
signal IAmDone, lightMyFire: std_logic; --Kontrollsignal til og fra modmult.
type state_type is (IDLE,INIT,REC_DATA,S1,S2); --tilstander i tilstandsmaskinen.
signal state : state_type;
begin
--*********************************************************************************
--*Oppkobling til modmult modulen.                                                 *
--*DataC alltid vil være den ene faktoren til modmult operasjonen og er hardkoblet  *
--*til faktor A, det samme gjelder keyN som er moduluo operatoren. Faktor B varierer *
--*og kobles da til databussen factorB.                                           *
--*Klokken og resetsignalet er det samme som for kjernen. Product er en utgang,   *
--*IAmDone og lightMyFire er stop og startsignal til og fra modmult.              *
--*********************************************************************************
modulo: entity work.modmult(three_Clock)
port map( dataC, factorB, keyN, product, lightMyFire, IAmDone, Clock, Reset);
Dout <= register_c(31 downto 0); --Dout er hardkoblet mot første del av register_c,
--register_C skifter for å skfite data ut.
--*************************************************************************
--*Kombinatoriske delen av tilstandsmaskinen, styrer alle kontrollsignalene samt
```

```
--*databussen factorB. Nærmere forklaring av tilstanden er forklart i den ikke-
--*kombinatoriske delen.
--*****************************************************************************

process(state, IAmDone, index, keyE((bitbredde -1)), StartCrypt, SetKeys, product, datam)
is
begin
case state is
when IDLE =>
 lightMyFire <= '0'; --kontrollsignal til modmult, 1 for utføre modmult.
 factorB <= (others => '0');
              --Faktor B i modmult, don't care untatt når modmult skal benyttes.
 Finish <= '1';
 --Så lenge kretsen er IDLE skal Finish stå høyt. Ellers skal den alltid vær
when INIT =>
 lightMyFire <= '0';
 factorB <= (others => '0'); --don't care
 Finish <= '0';
when REC_DATA =>
 Finish <= '0';
 factorB(0) <= '1'; --initsieringsverdi, ved første gjennomgang skal begge faktore
                    --C og C har en startverdi på en.
 factorB((bitbredde -1) downto 1) <= (others => '0');
 if index = 0 then
     --når index = 0 er dataene hentet inn og første modulærmultiplikasjon skal ut
  lightMyFire <= '1';
 else
  lightMyFire <= '0';
 end if;
when S1 =>
 Finish <= '0';
 factorB <= product((bitbredde -1) downto 0); --dataC og product er det samme.
 if IAmDone = '1' then
  if index = 0 then
   lightMyFire <= '0';
  else
   lightMyFire <= '1';
  end if;
 else
  lightMyFire <= '0';
 end if;
when S2 =>
 Finish <= '0';
 factorB <= dataM; --factorB settes til dataM
 if IAmDone = '1' then
  if keyE((bitbredde -1)) = '1' then
   lightMyFire <= '1';
  else
   lightMyFire <= '0';
  end if;
 else
  lightMyFire <= '0';
 end if;
end case;
end process;
--end process;
process(Reset, Clock) is
begin
--asynkron aktiv lav reset. Siden alle vipper vil bli gitt riktige verdier under
--initsiering
--vil det bli "don't care" hva de settes til. Tilstanden må settes til IDLE.
if Reset = '0' then
 state <= IDLE;
 index <= 0; --don't care
 keyN <= (others => '0'); --don't care
 keyE <= (others => '0'); --don't care
 dataM <= (others => '0'); --don't care
 dataC <= (others => '0'); --don't care
```

```
 register_c <= (others => '0'); --don't care
--Alle vipper trigger ved positiv klokkeflanke.
elsif Clock'event and Clock = '1' then
--*****************************************************************************
--*Tilstandmaskinen, fem tilstander. IDLE, INIT, REC_DATA, S1, S2.              *
--*I tillegg til å utføre operasjoner i de forskjellige tilstandene vil de fleste    *
--*transaksjonene til disse tilstandene utføre de samme.                        *
--*****************************************************************************
 case state is
--*****************************************************************************
--*Ved IDLE så venter den på SetKeys eller StartCrypt for å enten gå til initsierings*
--*eller motta data- tilstanden. Innhenting av data fra inngangen vil skje allerede i*
--*transaksjonene til disse tilstandene.                                        *
--*****************************************************************************
 when IDLE =>
  register_C((bitbredde - 33) downto 0) <= register_C((bitbredde -1) downto 32);
  if SetKeys = '1' then
   index <= ((bitbredde / 16) - 2);
 --bitbredde/32 * 2 - 2 klokkeperioder for å hente inn to nøkler.
   state <= INIT;
   keyN <= Din & keyN((bitbredde - 1) downto 32); --skifte Din på enden av keyN
   keyE <= keyN(31 downto 0) & keyE((bitbredde -1) downto 32);
 --og slutten av keyN inn på starten av keyE
  elsif StartCrypt = '1' then
   index <= ((bitbredde/32) - 2); --Fire sykluser - denne og '0'
   state <= REC_DATA;
   dataM <= Din & dataM((bitbredde - 33) downto 0);
  else
   index <= 0;
   state <= IDLE;
  end if;
--*****************************************************************************
--*INIT er initsieringstilstanden hvor nøklene hentes inn fra Dinngangen        *
--*Returnerer til IDLE når den er ferdig.                                       *
--*****************************************************************************
 when INIT =>
  register_C((bitbredde - 33) downto 0) <= register_C((bitbredde -1) downto 32);
  keyN <= Din & keyN((bitbredde -1) downto 32);
  keyE <= keyN(31 downto 0) & keyE((bitbredde -1) downto 32);
  if index = 0 then --ved index null går den tilbake til idle
   state <= IDLE;
  else --ellers så vil den trekke fra en på index og bli være ende i denne tilstanden
   index <= index - 1;
   state <= INIT;
  end if;

--*****************************************************************************
--*REC_DATA er også en initsieringstilstand. Data som skal behandles hentes fra      *
--*Dinngangen og legges i dataM.                                                *
--*Blir stående her frem til hele meldingsblokken er lastet inn og går deretter rett *
--*til arbeidstilstanden S1.                                                    *
--*****************************************************************************
 when REC_DATA =>
  register_C((bitbredde - 33) downto 0) <= register_C((bitbredde -1) downto 32);
  dataM <= Din & dataM((bitbredde -1) downto 32);
   --Din skiftes inn på enden av dataM.
  if index = 0 then --ved index lik 0 så er den ferdig og går videre til S1.
   index <= bitbredde;
   state <= S1;
   dataC(0) <= '1';--initsieringsverdi av dataC, må sette til 1.
   dataC((bitbredde -1) downto 1) <= (others => '0');
  else
   state <= REC_DATA;
   index <= index - 1;
  end if;
--*****************************************************************************
--*Arbeidstilstandene S1 og S2 er nesten helt like. S1 initsierer C*C mod N og S2    *
--*initsierer C*M mod N hvis E(index)= '1'. Før initsiering av modmult så venter hver*
```

```
--*tilstand på at modmult er ferdig med initsiering som eventuellt ble utført i      *
--*forrige tilstand. Når index når null er den ferdig og nestetilstand er IDLE       *
--*************************************************************************************
 when S1 =>
  if IAmDone = '1' then --venter på at modmult er ferdig.
   if index = 0 then --sjekker om den er ferdig med alle iterasjonene av algoritmen.
    state <= IDLE; --algoritmen er utført og kretsen går tilbake til IDLE.
    register_C <= product((bitbredde -1) downto 0);
    --det siste produktet fra modmult legges i utgangsregistret
 else
    state <= S2; --Hvis den ikke er ferdig så blir den stående å vente.
    index <= index - 1; --trekker fra en på index.
    dataC <= product((bitbredde -1) downto 0);--Legger siste produkt inn i C
   end if;
  else
   state <= S1;
  --Hvis modmult ikke er ferdig vil kretsen fortsette å vente i denne tilstanden.
  end if;
 when S2 =>
  if IAmDone = '1' then --venter på at modmult er ferdig.
   keyE <= keyE((bitbredde -2) downto 0) & keyE((bitbredde -1));
        --skifter keyE istedet for velge ut fra index.
   state <= S1;
   dataC <= product((bitbredde -1) downto 0);--Legger siste produkt inn i C
  else
   state <= S2;
  end if;
 end case;
end if;
end process;
end;--end architecture behaviour;
```

modmult

```
--Del av masteroppgaven RSA krypterings-system for AHEAD
--Institutt for elektronikk og telekommunikasjon
--ved Norges Teknisk-Naturvitenskapelige Universitet, NTNU
--Utført av: Vidar Eikrem Hervig, høst 2006
--VHDL 1993 syntax
--*************************************************************************************
--*modmult, utfører modulær multiplikasjon. Det vil si A*B mod M                      *
--*Benytter seg av Blakleys algoritme. Laget for å passe inn i en RSA-kjerne.       *
--*M må være større enn A og B                                                       *
--*Den generiske verdien bitbredde bestemmer størrelsen på både A, B og M           *
--*Faktor A må påtrykt inngangen under hele beregning, mens faktor B legges inn i et *
--*eget register ved initsieringen.                                                  *
--*Tre forskjellige arkitekturer kalt one_Clock, two_Clock og three_Clock. Disse er  *
--*ment til å bruke henholdsvis en, to og tre klokkeperioder for hver iterasjon.     *
--*Størrelsen og minimum klokkeperiode ved syntesering forventes å bli mindre for    *
--*two_Clock enn for one_Clock, og for three_Clock enn for two_Clock.                *
--*************************************************************************************
library ieee; --import av biblotek.
use ieee.std_logic_1164.all; --biblotek for vanlig std_logic operasjoner.
use ieee.std_logic_arith.all; --artimetiske operasjoner.
use ieee.std_logic_unsigned.all; --gjør at alle std_logic_vectorer tolkes som unsigned.
entity modmult is
generic( bitbredde : natural := 128);
port(
 factorA, factorBin : in std_logic_vector((bitbredde - 1) downto 0);
                     --multiplikand og multiplikator
 modulo_operator : in std_logic_vector((bitbredde - 1) downto 0); --modulo operator.
 product : inout std_logic_vector((bitbredde + 1) downto 0); --modmult produkt.
 lightMyFire : in std_logic; --Startsignal
 IAmDone : out std_logic; --stoppsignal
 Clock, reset : in std_logic ); --Klokke og aktiv lav asynkron reset
end entity modmult;
--*************************************************************************************
--*three_Clock arkitektur, utfører hver iterasjon på opp til tre klokkesykeler,      *
--*benytter seg av lite kombinatorikk.                                               *
--*************************************************************************************
architecture three_Clock of modmult is
type state_type is (idle, s1, subonce, subtwice);
signal state: state_type;
signal index: natural range 0 to (bitbredde - 1);
signal factorB : std_logic_vector((bitbredde - 1) downto 0);
begin
--*************************************************************************************
--*Kombinatorisk del. IAmDone er det eneste kontrollsignalet i denne kretsen.        *
--*IAmDone benyttes til å fortelle at kretsen er ferdig og klar til å motta nye data.*
--*************************************************************************************
process(state) is
begin
 case state is
 when IDLE =>
  IAmDone <= '1';
 when others =>
  IAmDone <= '0';
 end case;
end process;
process(Clock, reset) is
variable temp_product : std_logic_vector((bitbredde + 1) downto 0);
begin
--asynktron aktiv lav reset.
if reset = '0' then
 state <= idle;
 product <= (others => '0'); --don't care
 factorB <= (others => '0'); --don't care
 Index <= (bitbredde - 1); --don't care
```

```
--alle vipper trigger på positiv klokkeflanke
elsif Clock'event and Clock = '1' then
--*****************************************************************************
--*Tilstandsmaskin, fire tilstander IDLE(tomgangs tilstand) S1(utfører P <= 2P+AB(i))*
--*SUBONCE og SUBTWICE(Begge utfører P := P - N hvis P >= N).                    *
--*Det vil ta mellom bitbredde*2 og bitbredde*3 klokkeperioder for tilstandsmaskinen *
--*å vende tilbake til IDLE etter den har tatt imot lightMyFire signalet.       *
--*****************************************************************************
 case state is
 when idle =>
  factorB <= factorBin;
  index <= (bitbredde - 1);
  if lightMyFire = '1' then
   product <= (others => '0');
   state <= S1;
  else
   state <= idle;
  end if;
 when S1 =>
  state <= subtwice;
  factorB((bitbredde - 1) downto 1) <= factorB((bitbredde - 2) downto 0);
  if factorB((bitbredde - 1)) = '1' then
            --sjekk på forrige verdi av factorB((bitbredde - 1))
   product((bitbredde + 1) downto 0) <= ('0' & product((bitbredde - 1) downto 0) & '0') +
("00" & factorA);
  else
   product((bitbredde + 1) downto 0) <= ('0' & product((bitbredde - 1) downto 0) & '0');
  end if;
--*****************************************************************************
--*SUBTWICE trekker fra N en gang til sjekker om svaret blir positivt og legge det da*
--*inn i product-registret.  Nestetilstand blir i såfall SUBONCE.                *
--*Ellers returneres den til S1                                                 *
--*****************************************************************************
 when subTwice =>
  temp_product := product - ("00" & modulo_operator);
  if temp_product((bitbredde + 1)) = '0' then
         --hvis temp_product er positvt legges den inn i product
   state <= subOnce;
   product <= temp_product;
  elsif index = 0 then
    state <= idle;
  else
    state <= S1;
    index <= index - 1;
  end if;
--*****************************************************************************
--*SUBONCE trekker fra N en gang til sjekker om svaret blir positivt og legge det da *
--*inn i product-registret.                                                      *
--*Returnerer til S1.                                                           *
--*****************************************************************************
 when subOnce =>
  temp_product := product - ("00" & modulo_operator);
  if temp_product((bitbredde + 1)) = '0' then --hvis temp_product er positvt legges den
--inn i product
   product <= temp_product;
  end if;
  if index = 0 then
   state <= idle; --nestetilstanden er IDLE hvis den er ferdig med alle iterasjonene.
  else
   state <= S1;
   index <= index - 1;
  end if;
end case;
end if;
end process;
end architecture three_Clock;
```

Vedlegg C Testbenker

Testbenk for RSACore

```
--Del av masteroppgaven RSA krypterings-system for AHEAD
--Institutt for elektronikk og telekommunikasjon
--ved Norges Teknisk-Naturvitenskapelige Universitet, NTNU
--Utført av: Vidar Eikrem Hervig, høst 2006
--VHDL 1993 syntax
--*****************************************************************************
--*Testbenk for kretsene laget til semesteroppgaven.                            *
--*****************************************************************************
library ieee;
use ieee.std_logic_1164.all;
entity testbenk is
end entity; --testbenk;
--*****************************************************************************
--*****************************************************************************
--*Testbenk for RSACore
--*****************************************************************************
--*****************************************************************************
architecture RSAcore_test of testbenk is
signal Din, Dout : std_logic_vector(31 downto 0);
signal StartCrypt,SetKeys,Finish,Reset : std_logic;
signal Clock : std_logic := '0';
signal keyE,keyN,keyD,dataC,dataM : std_logic_vector(127 downto 0);
begin
DUT: entity work.RSAcore(behaviour)
port map(
Din=>Din,
Dout=>Dout,
StartCrypt=>StartCrypt,
SetKeys=>SetKeys,
Finish=>Finish,
Reset=>Reset,
Clock=>Clock
);
klokke: process is
begin
Clock <= not(Clock);
wait for 5 ns;
end process;
process is
begin
Reset <= '0';
--************************************************
--*Privat og offentlig nøkkel samt først datapakke*
--************************************************
keyE <= X"00000000000000000000000010001";
keyN <= X"819DC6B2574E12C3C8BC49CDD79555FD";
keyD <= X"819D451512390089590AF8BB46DB0001";
dataM <= X"0AAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
--************************************************
--svar:
--c=m^e mod n=7637EA28188632D8F2D92845DB649D14
--************************************************
SetKeys <= '0';
StartCrypt <= '0';
Din <= (others => '0');
dataC <= (others => '0');
wait for 10 ns;
Reset <= '1';
--************************************************
```

```
--*Innlegging av nøkler i kjernen.                *
--************************************************
SetKeys <= '1';
Din <= keyE(31 downto 0);
wait for 10 ns;
SetKeys <= '0';
Din <= keyE(63 downto 32);
wait for 10 ns;
Din <= keyE(95 downto 64);
wait for 10 ns;
Din <= keyE(127 downto 96);
wait for 10 ns;
Din <= keyN(31 downto 0);
wait for 10 ns;
Din <= keyN(63 downto 32);
wait for 10 ns;
Din <= keyN(95 downto 64);
wait for 10 ns;
Din <= keyN(127 downto 96);
--************************************************
--*Innlegging av datapakke                         *
--************************************************
wait until Finish = '1';
wait for 10 ns;
StartCrypt <= '1';
Din <= dataM(31 downto 0);
wait for 10 ns;
StartCrypt <= '0';
Din <= dataM(63 downto 32);
wait for 10 ns;
Din <= dataM(95 downto 64);
wait for 10 ns;
Din <= dataM(127 downto 96);
--************************************************
--*Venter på svar fra kjernen og legger svaret inn*
--*i dataC. Sender samtidig inn neste datapakke.  *
--************************************************
wait until Finish = '1' ;
wait for 10 ns;
dataC <= Dout & dataC(127 downto 32);
dataM <= X"0C0FFEE0C0FFEE0C0FFEE0C0FFEE0000";
--************************************************
--svar:
--c=m^e mod n=30070EBE021626FA3380A7192C956B6D
--************************************************
wait for 10 ns;
StartCrypt <= '1';
Din <= dataM(31 downto 0);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
StartCrypt <= '0';
Din <= dataM(63 downto 32);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= dataM(95 downto 64);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= dataM(127 downto 96);
wait for 10 ns;
--************************************************
--*venter på at kjernen er ferdig og gjør klar     *
--*dekrypteringen ved å legge siste svar inn i dataM
--************************************************
dataM <= dataC;
wait until Finish = '1';
wait for 10 ns;
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
```

```
SetKeys <= '1';
Din <= keyD(31 downto 0);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= keyD(63 downto 32);
SetKeys <= '0';
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= keyD(95 downto 64);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= keyD(127 downto 96);
wait for 10 ns;
Din <= keyN(31 downto 0);
wait for 10 ns;
Din <= keyN(63 downto 32);
wait for 10 ns;
Din <= keyN(95 downto 64);
wait for 10 ns;
Din <= keyN(127 downto 96);
--***********************************************
--*Legger inn datapakke for dekryptering          *
--***********************************************
wait until Finish = '1' ;
wait for 10 ns;
StartCrypt <= '1';
Din <= dataM(31 downto 0);
wait for 10 ns;
StartCrypt <= '0';
Din <= dataM(63 downto 32);
wait for 10 ns;
Din <= dataM(95 downto 64);
wait for 10 ns;
Din <= dataM(127 downto 96);
wait for 10 ns;
--legger inn siste krypterte pakke inn i dataM for dekryptering
dataM <= dataC;
wait until Finish = '1' ;
wait for 10 ns;
StartCrypt <= '1';
Din <= dataM(31 downto 0);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
StartCrypt <= '0';
Din <= dataM(63 downto 32);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= dataM(95 downto 64);
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
Din <= dataM(127 downto 96);
dataC <= Dout & dataC(127 downto 32);
wait until Finish = '1';
wait for 10 ns;
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
dataC <= Dout & dataC(127 downto 32);
wait for 10 ns;
report "Finished"
severity Failure;
end process;
end architecture;
```

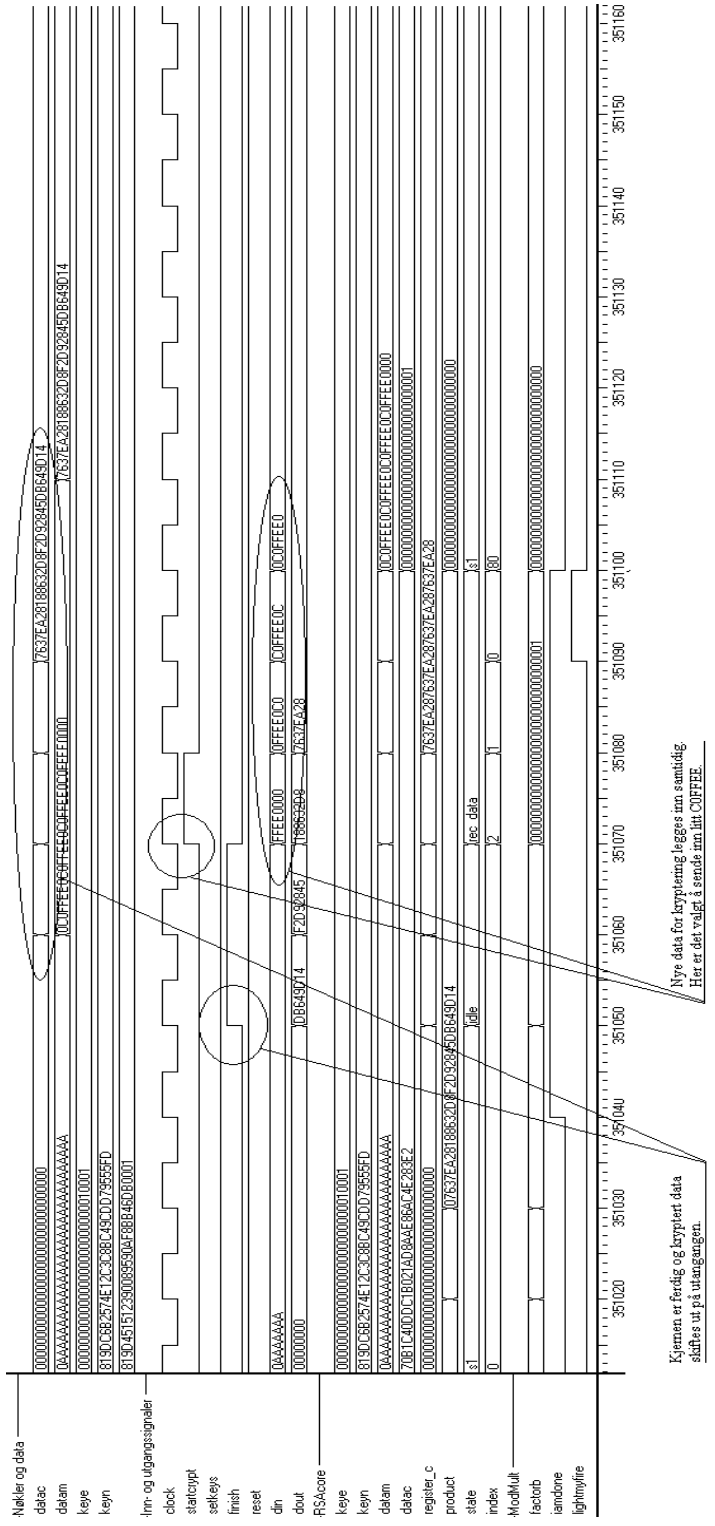
Testbenk for modmult

```
--Del av masteroppgaven RSA krypterings-system for AHEAD
--Institutt for elektronikk og telekommunikasjon
--ved Norges Teknisk-Naturvitenskapelige Universitet, NTNU
--Utført av: Vidar Eikrem Hervig, høst 2006
--VHDL 1993 syntax
library ieee;
use ieee.std_logic_1164.all;
entity testbenk is
end; --testbenk;

architecture test_modmult of testbenk is
signal factorA, factorBin, moduloOperator : std_logic_vector(127 downto 0);
signal lightMyFire, reset : std_logic;
signal Clock : std_logic := '0';
signal product : std_logic_vector(129 downto 0);
signal IAmDone : std_logic;
begin
test: entity work.modmult(three_Clock)
port map( factorA, factorBin, moduloOperator, product, lightMyFire, IAmDone, Clock,
reset);
--*********************************************************
--*Simulering av klokke, 10ns klokkeperiode gir 100MHz    *
--*Frekvensen spiller ingen rolle for logikken i          *
--*simuleringen siden tidsforsinkelser ikke er med.       *
--*********************************************************
klokke: process is
begin
Clock <= not(Clock);
wait for 5 ns;
end process;

stimulans: process is
begin
factorA <= x"00000000000000000000000000000000";
factorBin <= x"00000000000000000000000000000000";
moduloOperator <= x"00000000000000000000000000000000";
lightMyFire <= '0';
reset <= '1';
wait for 10 ns;
reset <= '0'; --aktiv lav reset.
wait for 10 ns;
reset <= '1';
wait for 20 ns;
--*********************************************************
--*Sjekker for to små faktorer og en liten modulo operator *
--* (12*12) mod 13 = 1                                    *
--*********************************************************
lightMyFire <= '1';
factorA <= x"0000000000000000000000000000000C";
factorBin <= x"0000000000000000000000000000000C";
moduloOperator <= x"0000000000000000000000000000000D";
wait for 10 ns;
lightMyFire <= '0';
wait until IAmDone = '1';
wait for 10 ns;
--*********************************************************
--*Sjekker for tilfeldige store faktorer og modulo operator *
--*Fasiten er funnet med hjelp av matlab.                  *
--*Svaret skal bli 73BA30B9116929D997E5CB84B5E6AFD7        *
--*********************************************************
lightMyFire <= '1';
factorA <= x"001E11180B2584C0D0B240E1662FBB3D";
factorBin <= x"8003350C2E23DD711667F27D48ADD81D";
moduloOperator <= x"8000424B7EC569CEDB8B4FA67785F97E";
```

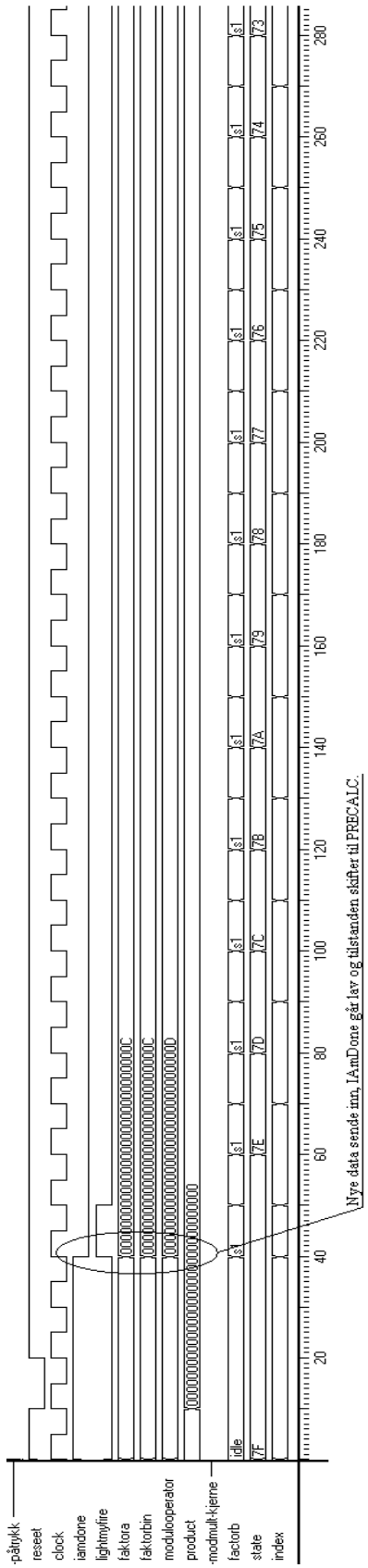
```
wait for 10 ns;
lightMyFire <= '0';
wait until IAmDone = '1';
wait for 10 ns;
--*********************************************************
--*Sjekker for tilfeldige store faktorer og modulo operator *
--*Modulo operatoren er med vilje valgt mindre enn factor A *
--*Algoritmen tar ikke høyde for dette og vil gi galt svar  *
--*Svaret skal bli 3A70505A97409978D6                       *
--*********************************************************
lightMyFire <= '1';
factorA <= x"8785EC0D6D5FA61DDA41D2E0C4CD25C0";
factorBin <= x"87247D31A39B21DFCD8826FB6B95FBC6";
moduloOperator <= x"0000000000000DCC688517346C05F97E";
wait for 10 ns;
lightMyFire <= '0';
wait;
end process;
end architecture;
```

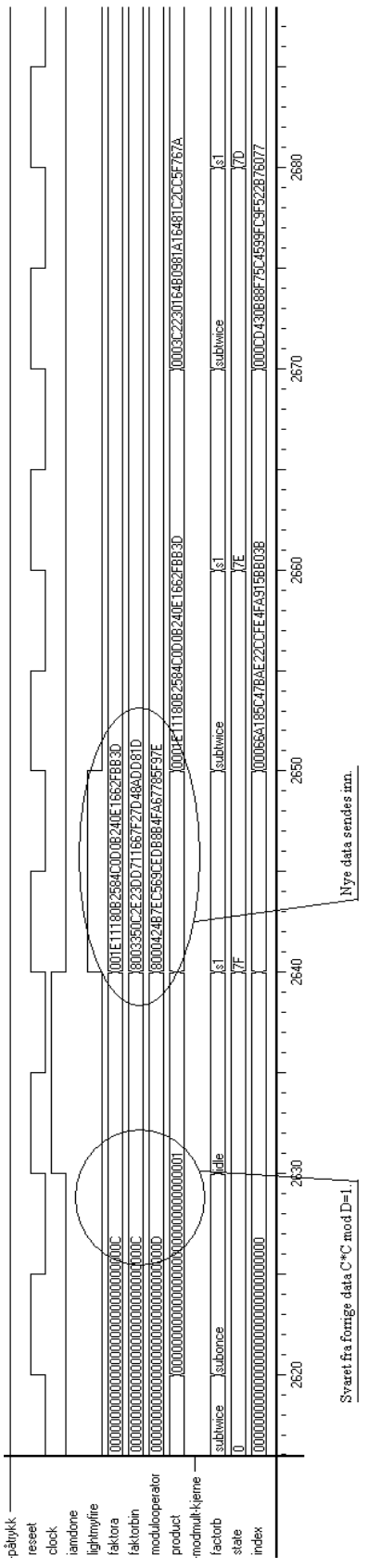
Vedlegg D Tidsdiagram

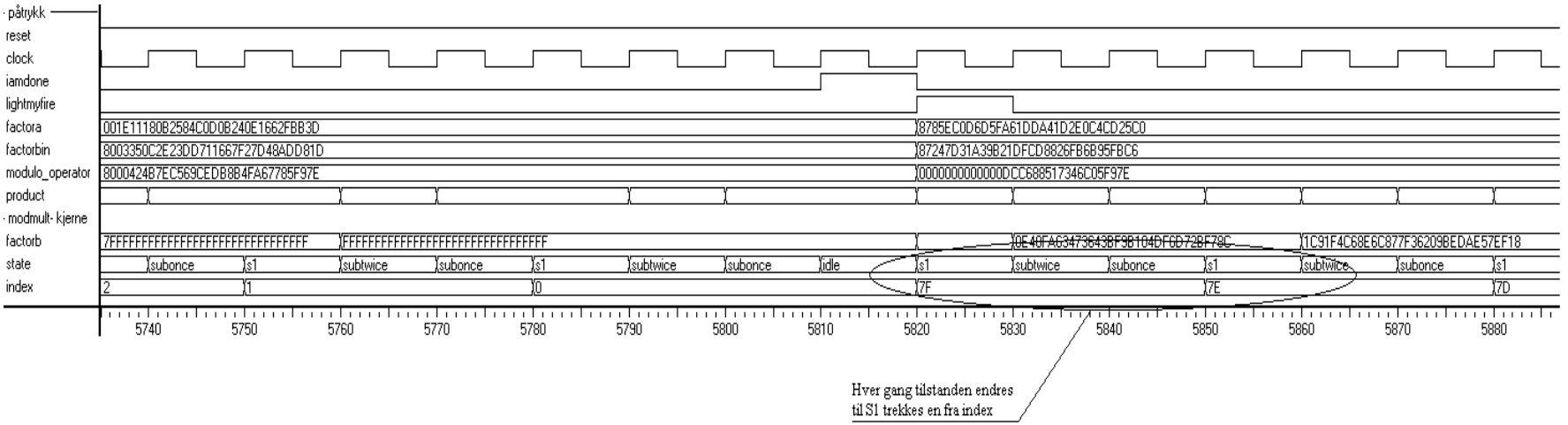
Tidsdiagram for RSAcore

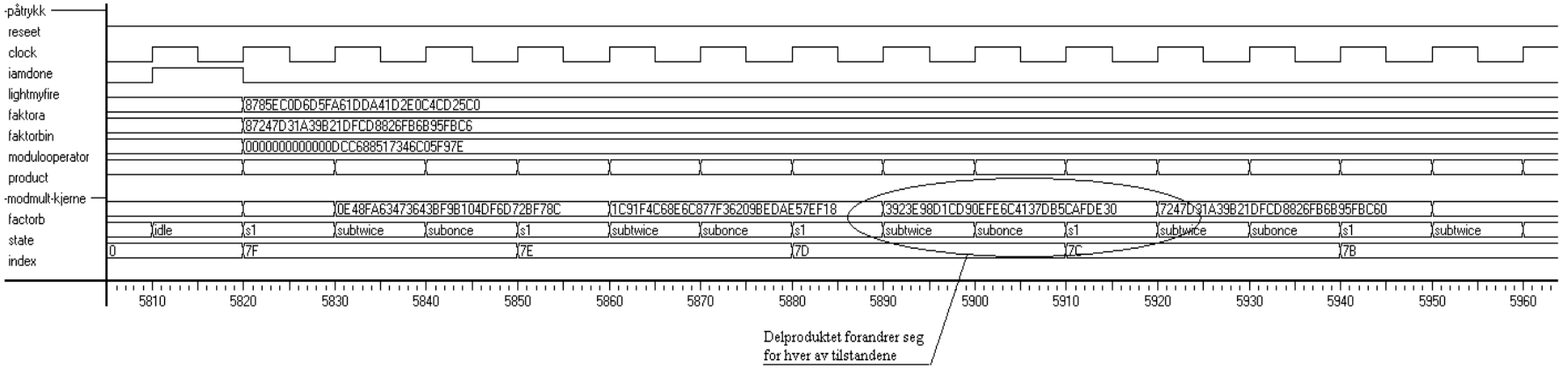


Tidsdiagram modmult






påtrykk
reset
clock
iamdone
lightmyfire
factora
factorbin
modulo_operator
product
modmult- kjerne
factorb
state
index
001E11180B2584C0D0B240E1662FBB3D
8003350C2E23DD711667F27D48ADD81D
8000424B7EC569CEDB8B4FA67785F97E
8785EC0D6D5FA61DDA41D2E0C4CD25C0
87247D31A39B21DFCD8826FB6B95FBC6
0000000000000DCC688517346C05F97E
7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0E48FA63473643BF9B104DF6D72BF78C
1C91F4C68E6C877F36209BEDAE57EF18
subonce
s1
subtwice
idle
2
1
0
7F
7E
7D
5740
5750
5760
5770
5780
5790
5800
5810
5820
5830
5840
5850
5860
5870
5880
Hver gang tilstanden endres
til S1 trekkes en fra index


påtrykk
reseet
clock
iamdone
lightmyfire
faktora
faktorbin
modulooperator
product
modmult-kjerne
factorb
state
index
8785EC0D6D5FA61DDA41D2E0C4CD25C0
87247D31A39B21DFCD8826FB6B95FBC6
0000000000000DCC688517346C05F97E
0E48FA63473643BF9B104DF6D72BF78C
1C91F4C68E6C877F36209BEDAE57EF18
3923E98D1CD90EFE6C4137DB5CAFDE30
7247D31A39B21DFCD8826FB6B95FBC60
idle
s1
subtwice
subonce
0
7F
7E
7D
7C
7B
5810
5820
5830
5840
5850
5860
5870
5880
5890
5900
5910
5920
5930
5940
5950
5960
Delproduktet forandrer seg
for hver av tilstandene