NTNU
Det skapende universitet

Feilsimulering og ekvivalens-sjekking med FPGA og innebygd stimuligenerering

Stig Kristian Opstad

Master i elektronikk
Oppgaven levert: Oktober 2007
Hovedveileder: Bjørn B. Larsen, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

For å øke hastigheten på feilsimulering og ekvivalens-sjekking kan man implementere en riktig modell (Golden) og en modell med feil (KUT) i en FPGA. Dette kan også benyttes for å verifisere at en altenativ beskrivelse av en krets er lik den opprinnelige. Med verifisering i FPGA kan man benytte tilsynelatende tilfeldige vektorer (PRV) som testpådrag.

For å effektivisere testingen må det være mulig å injisere og fjerne tidligere injiserte feil i KUT enkeltvis uten å laste hele innholdet på nytt.

Undersøk hva som er skrevet om denne typen testing og verifisering.

Vurder hvor statistisk god en slik verifikasjon av KUT vil være.
Beskriv en egnet algoritme for lokalisering av feillokasjoner som gir høy troverdighet for testen, og en effektiv metode for feilinjeksjon.

Oppgaven gitt: 26. juni 2007
Hovedveileder: Bjørn B. Larsen, IET

Innholdsliste

1 Introduksjon....2
2 Feil og feilmodeller....3
2.1 Logisk nivå feil....3
2.2 Single låst-til feil....4
2.2.1 Feilekvivalens og feildeteksjon....6
2.3 Designverifikasjon....7
2.4 Feildekningsgrad....7
2.5 Statistiske metoder for feilsimulering....8
2.5.1 Generell statistisk teori....8
2.5.2 Bruk av feilprøver....11
3 Feiltest implementasjoner....14
3.1 Design for test....14
3.2 Testmønster generatorer....16
3.3 Pseudotilfeldige generatorer....17
3.3.1 Vektet pseudotilfeldige generatorer....18
3.4 Linear Feedback Shift Register....19
3.5 Cellular Automata....21
3.6 Utgangsresponsanalyse....23
4 Ekvivalentsjekk på FPGA....25
4.1 Design av ekvivalentsjekk....25
4.1.2 LFSR mønstergenerator....25
4.1.2 CA mønstergenerator....26
4.1.3 Komparator tre....27
4.2 Måling av feildekningsgrad....29
4.2.1 Dynamisk konfigurasjon....29
4.2.2 Statisk konfigurasjon....31
4.2.3 Fremgangsmåte for feilinjisering....31
4.3 Testkretser....33
4.4 Implementasjon på Virtex-II pro FPGA....34
5 Resultat og analyse....35
6 Konklusjon....37
Referanseliste....38
Vedlegg A: Primitive polynom for LFSR og CA....39
Vedlegg B: VHDL kode....41
Vedlegg C : Informasjon om FPGA kortet....48

1 Introduksjon

Formal ekvivalentsjekk er en industristandard for verifikasjon av kretsdesign. Verktøy basert på denne metoden har svært god dekningsgrad, men har til felles at de er svært tidkrevende. En fullstendig dekkende test av et design, i form av vektorer påtrykket inngangen, begrenser seg til små design. Hvis kretsen har et stort antall innganger, vil en fullstendig dekkende test være en svært tidkrevende operasjon. En krets med 32 innganger krever $2^{32} = 4.29*10^9$ forskjellige testvektorer for en fullstendig test. I tillegg vil påtrykk på de primære inngangene, kun avdekke et begrenset omfang av feilmodeller, mye på grunn av lange sekvensielle kjeder, og «fan-outs» som gjør noder lite tilgjengelige. Et forslag til en løsning på dette problemet er å benytte en DFT løsning som kan evaluere sin egen testeffektivitet, og optimalisere testen med hensyn på resultatet. Ved å realisere kretsdesignet på en FPGA krets, kan forskjellige DFT løsninger enkelt implementeres, uten å introdusere store kostnader i form av redusert yield eller ekstra pinner. For å evaluere feildekningsgraden, kan feilmodeller injiseres i kretsen uten å skape varige defekter, ettersom en FPGA lagrer kretsdesignet i minneelementer. Enkelte FPGA moduler har også mulighet for partiell rekonfigurasjon, som åpner for evaluasjon av testeffektiviteten uten å gjøre endring på designet.

Ordforklaringer	
GOLD	Ved simulering av flere kretser, representerer GOLD fasiten.
KUT	Krets Under Test
DFT	Design For Test
Feildekningsgrad	Forholdet mellom antall feil detektert i en test, og det totale antallet av mulige feilmodeller i en krets.
Testmønster	Beskriver et sett av testvektorer.

2 Feil og feilmodeller

Teksten i dette kapittelet referer til [9].

Definisjon av **defekt**: En defekt i et elektronisk system er en forskjell mellom en realisert krets og et kretsdesign, som ikke er planlagt. Defekter oppstår enten under produksjon av kretsen eller ved bruk av modulen. Gjentakende dannelse av samme defekt indikerer behov for forbedring av produksjonsprosessen eller designet av modulen.

Definisjon av **avvik**(fra eng. «error»): Feil i utgangssignalet produsert av et defekt system er kalt et avvik, og er et resultat av en defekt.

Definisjon av **feil/feilmodell**: En representasjon av en defekt på et abstrakt funksjonsnivå er kalt en feilmodell. Forskjellen mellom defekt og feil er ikke helt entydig, men de er henholdsvis ufullstendighet i hardware og funksjon.

Å modellere feil samsvarer i stor grad med modellering av selve kretsen. Ordet «nivå» refererer til grad av abstraksjon i et design hierarki, hvor oppførselsnivået(høyt nivå) har færre implementeringsdetaljer. Feilmodeller på dette nivået har ingen klare sammenhenger med produksjonsfeil, og har størst betydning i designverifikasjoner basert på simulering.[9] RTL(Register-Transfer Level) eller logisk nivå inneholder en nettliste av porter. Låst-til(fra eng. «stuck-at», refereres som s-a-0 og s-a-1, eller sa-0 og sa-1) feil er de mest populære feilmodellene på dette nivået i digital testing. Andre feilmodeller er «bridging faults» og «delay faults». Transistornivå feil inkluderer blant annet «stuck-open» feil, som også refereres til som teknologiavhengige feil. Transistornivå feilmodeller brukes i hovedsak i testing av analoge kretser.

2.1 Logisk nivå feil

Definisjoner fra [9]:

En **buss feilmodell** setter hver linje i en databuss som sa-0, sa-1 eller feilfri. For en n-bit buss er det `3n-1` buss feilmodeller. En total buss feilmodell antar alle linjene i bussen til å være enten sa-0 eller sa-1.

Forsinkelse feil(fra eng. «delay) forårsaker den kombinatoriske forsinkelsen i en krets til å overskride klokkeperioden.

Logiske feil omfatter feilmodeller som påvirker tilstanden til logiske signaler. Normalt modelleres de som 0, 1, X(ukjent) og Z(høy impedans). En slik feil kan transformere en korrekt verdi til

hvilken som helst av de nevnte verdiene.

Potensielle detekterbare feil har følgende karakteristikk. Når en test er påtrykket en sekvensiell krets, vil noen feilmodeller produsere en ukjent tilstand på utgangen når deterministisk utgangsverdi er forventet i den feilfrie kretsen. Slike forhold defineres som potensiell deteksjon. Deterministisk deteksjon krever at både den feilaktige og feilfrie utgangsverdien skal være forskjellig og entydig(logisk 0 eller 1). Generelt omtales låst-til feil som kun kan detekteres potensielt, som «potensielt detekterbare feil».

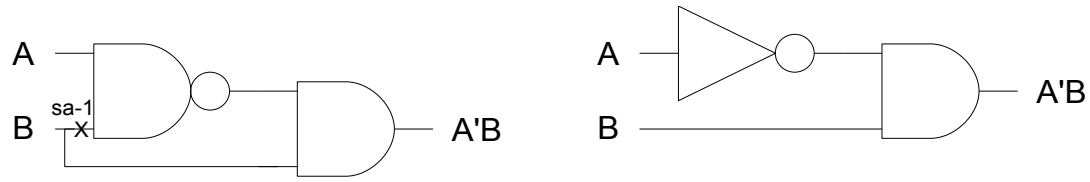
Udetekterbar feil omfatter alle feil hvor det ikke finnes en test, det vil si at det ikke finnes en test som kan detektere de. Det er to typer ikke-testbare feil, redundante feil og initialiseringfeil.

Initialiseringfeil finnes i kretser med minneelementer som er laget slik at de skal initialiseres av et inngangssignal. Feil som forstyrrer denne prosessen er kalt initialiseringfeil. Et typisk eksempel er klokkeinngangen på en vippe som er låst i en passiv tilstand.

Redundante feil er feil som ikke endrer inn-ut funksjonen av en kombinatorisk krets. En redundant feil kan ikke bli detektert av noen tester. Slike feil kan fjernes fra kretsen uten å ha påvirkning på kretsfunksjonen, og en slik operasjon blir ofte kalt optimalisering. Slike feil i sekvensielle kretser omtales som ikke-testbare feil.

I *figur 1* deler signalet B seg og møtes i AND-porten, mens til høyre er det ingen linjer som deler seg. Et kretsdesign hvor ingen linjer deler seg kalles et tre, og i en slik struktur kan det ikke eksistere single redundante låst-til feil.[9]

Låst-til feil(fra eng. «stuck-at) tildeler en signallinje en bestemt verdi, logisk 0 eller 1. En signallinje er inngangen eller utgangen av en logisk port eller vippe. Den mest populære formen er single låst-til feilmodeller som har to feil per linje, sa-0 eller sa-1.



Figur 1: Til venstre en redundant feil, og optimalisering av samme logiske funksjon til høyre.

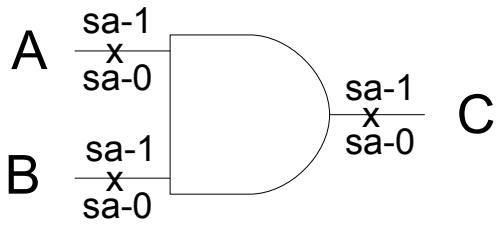
2.2 Single låst-til feil

Single låst-til feil antas å kun påvirke koblingen mellom porter. Hver koblingslinje kan ha to typer feil, sa-0 og sa-1. En linje med sa-1 vil alltid ha den logiske verdien 1 uavhengig av den logiske

funksjonen fra porten som driver den. Ved tilstedeværelse av flere låst-til feil samtidig på `n` linjer, kan de ha `3n-1` forskjellige kombinasjoner. Det kommer av at hver linje kan ha tre forskjellige tilstander sa-0, sa-1 og feilfri. Til og med et fåtall av linjer vil gi et betydelig antall låst-til kombinasjoner. Derfor er det vanlig å kun modulere single låst-til feil.

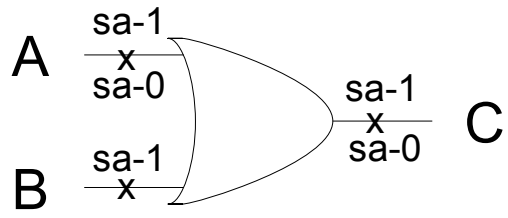
Figur 2, 3 og 4 viser singel låst-til oppførsel på henholdsvis en AND-, OR og NOT-port. De to kolonnene til venstre i tabellene representerer sannhetstabellen til den tilhørende porten. De tre brede kolonnene til høyre (kun to i *figur 4*) representerer utgangsresponsen på C som resultat av de forskjellige kombinasjonene av single sa-0 og sa-1 på porten. De resultatene som har avvik i forhold til den feilfrie utgangsresponsen er farget grå.

AND		A		B		C	
AB	C	sa0	sa1	sa0	sa1	sa0	sa1
00	0	0	0	0	0	0	1
01	0	0	1	0	0	0	1
10	0	0	0	0	1	0	1
11	1	0	1	0	1	0	1



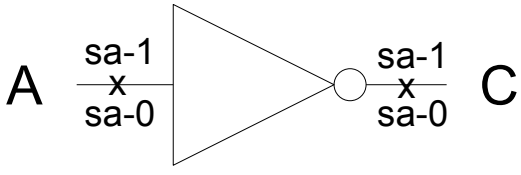
Figur 2: Singel låst-til oppførsel på en AND-port.

OR		A		B		C	
AB	C	sa0	sa1	sa0	sa1	sa0	sa1
00	0	0	1	0	1	0	1
01	1	1	1	0	1	0	1
10	1	0	1	1	1	0	1
11	1	1	1	1	1	0	1



Figur 3: Singel låst-til oppførsel på en OR-port.

NOT		A		C	
sa0	sa0	sa0	sa1	sa0	sa1
0	1	1	0	0	1
1	0	1	0	0	1



Figur 4: Singel låst-til oppførsel på en NOT-port.

2.2.1 Feilekvivalens og feildeteksjon

Fra [9], anta en kombinatorisk krets med `n` inngangsverdier, og kun én utgangsverdi. Utgangresponsen av denne kretsen kan uttrykkes som en funksjon $f_0(V)$, hvor `V` er en n-bit vektor. Ved tilstedeværelse av en feil `f1` i kretsen, kan utgangresponsen av kretsen uttrykkes som $f_1(V)$. En annen feil i kretsen `f2`, gir en utgangsrespons uttrykt som $f_2(V)$.

For at en test skal detektere `f1` må utgangsresponsen $f_1(V)$ være forskjellig fra $f_0(V)$, og dette kan uttrykkes som:

$$f_0(V) \text{ XOR } f_1(V) = 1 \tag{2.1}$$

Det samme kan uttrykkes for feil `f2`:

$$f_0(V) \text{ XOR } f_2(V) = 1 \tag{2.2}$$

Hvis `f1` og `f2` blir detektert av nøyaktig samme testvektorer, vil feilene være ekvivalente. Dette kan uttrykkes som

$$f_1(V) \text{ XOR } f_2(V) = 0 \tag{2.3}$$

Definisjon av feilekvivalens[9]: To feil i en boolsk krets er kalt ekvivalente hvis de påvirker kretsen ved at de to resulterende feilaktige kretsene har identisk utgangsrespons funksjon.

Et sett av alle feilmodeller i en krets kan deles inn i grupper med ekvivalente sett. På den måten kan antall feilmodeller reduseres, ved å kun velge en feilmodell fra hver gruppe med ekvivalente feil. En slik utvelgelse kalles feil kollapsing, og hvor stor andel som kollapser kan uttrykkes

$$Grad\ av\ kollaps = \frac{|Antall\ feil\ kollapser|}{|Totalt\ antall\ feil|} \tag{2.4}$$

Definisjon av feildominans[9]: Hvis alle tester for en feilmodell f1 detekterer en annen feilmodell f2, kan man si at f2 dominerer f1. Disse to feilene er også ekvivalente for en utvalgt gruppe testvektorer, det vil si de som detekterer f1.

Teorem om feildeteksjon i en krets uten «fan-out»[9], det vil si at ingen linjer deler seg i kretsen. Et sett av tester som detekterer alle single låst-til feil på de primære inngangene på en «fan-out» fri krets, vil detektere alle single låst-til feil i kretsen.

Teorem om sjekkpunkt[9]. Et sett av tester som detekterer alle single låst-til feil på sjekkpunktene

i en kombinatorisk krets, vil detektere alle single låst-til feil i kretsen. Sjekkpunkt defineres som primære innganger og noder hvor linjer deler seg i en kombinatorisk krets.

2.3 Designverifikasjon

Et design kan verifiseres på forskjellige oppførselsnivå. En designverifikasjon basert på simulering(høyt nivå) har sine fordeler og ulemper. Ved bruk av f.eks programmeringsspråket C kan en svært kompakt designbeskrivelse benyttes, men verifikasjonen begrenser seg i hovedsak til logisk oppførselsnivå. Ved å erstatte designbeskrivelsen basert på C med logiske porter i form av en nettliste, kan også timing oppførselen verifiseres. Denne prosessen kan fortsette ved å gå dypere ned i designhierarkiet, erstatte logiske porter med transistornivå eller kretsnivå beskrivelse av kretsen. For å kunne opprettholde kompleksiteten i simuleringen, vil ikke en fullstendig test la seg gjennomføre(pga av tid), og en garanti på fullstendig verifikasjon blir derfor umulig.[9]

For å kunne oppnå en garanti på verifikasjon må formelle verifikasjonsmetoder benyttes. Uttrykket «formell verifikasjon» refererer til en mengde forskjellige metoder for å bevise at en modell av et system har bestemte egenskaper. Hva som skiller formell verifikasjon fra andre verifikasjonsmetoder, er at formell verifikasjon gir en matematisk garanti. Dette betyr at det ikke finnes en oppførsel eller utførelse av modellen som kan betvile dette. Dette er forskjellig fra andre typer verifikasjoner, som kun tester en gitt modell i mange forskjellige situasjoner, hvor den blir verifisert hvis ingen av testene motbeviser de gitte egenskapene. Denne type verifisering kan aldri bevise at en modell i alle tilfeller har de rette egenskapene.[12] En avgrenset versjon av formell verifikasjon er modellsjekking (fra eng. *model checking*), som verifiserer alle tilstandene omstendig ved hjelp av en fullstendig test av tilstandsområdet. På grunn av den høye kompleksiteten i formelle metoder kan kun høynivå verifisering benyttes. Et ideelt designverifikasjon system kombinerer formell verifikasjon med logikk- eller kretsnivå simulering.[9]

2.4 Feildekningsgrad

Fra [1], feildekningsgrad (fra eng. *fault coverage*) er en måleenhet for å gradere kvaliteten på en feiltest, og defineres som:

$$FC(Fault\ Coverage) = \frac{antall\ detekterte\ feil}{totalt\ antall\ feil} \tag{2.5}$$

Det er flere faktorer som har innvirkning på dekningsgraden, og disse gjør det opp mot umulig å

oppnå en komplett test, FC=1. En faktor er begrensning av testbarhet, det vil si når ikke hele kretsen kan aksesseseres fra de tilgjengelige pinnene. En annen begrensning i en feiltest er når feil/defekter ikke korresponderer etter feilmodellen som det testes mot (f.eks åpen kobling). Andre faktorer som påvirker beregningen av feildekningsgrad, er udetekterbare feil og potensielle detekterte feil. Begge kan redusere den totale feildekningsgraden, så lenge man ikke tar spesielt hensyn til disse type feilene i forbindelse med beregning av feildekningsgrad.[2] Hvis det ikke finnes testvektorer, eller sekvenser av testvektorer, som kan detektere en gitt feil, vil denne feilen være udetekterbar.[9] Redundans i kretslogikken er et typisk eksempel som kan være en årsak til udetekterbare feil.

2.5 Statistiske metoder for feilsimulering

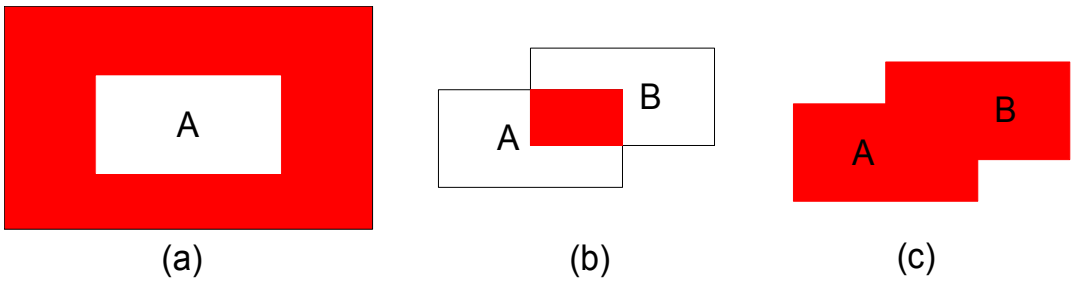
Statistikk kan benyttes for å gjøre et estimat av feildekningsgraden, og kan redusere omfanget av en feilsimuleringsprosess. I det påfølgende kapittelet er det gjennomgang av de statistiske metodene benyttet i dette estimatet.

2.5.1 Generell statistisk teori

Denne kortfattet oppsummeringen av statistisk teori er hentet fra [10]

Statistiske begreper:

Populasjon er mengden vi ønsker å studere
Utvalg er en delmengde av populasjonen, som er trukket tilfeldig fra populasjonen.
Representativt, et utvalg er representativt hvis det kan trekkes konklusjoner fra hele populasjonen.
Diskrete data beskriver en mengde som kan telles.
Kontinuerlig data beskriver en mengde som ikke kan telles.
Utfallsrom er alle mulige utfall av et statistisk forsøk. Hvis utfallsrommet er diskrete inneholder det et tellbart antall mulige utfall, mens et kontinuerlig utfallsrom inneholder uendelig antall utfall.
Hendelse er en delmengde av et utfallsrom.
*Stokastisk variabl*e et er en funksjon som tilordner verdier til elementer i utfallsrommet til et tilfeldig eksperiment.



Figur 5: (a) komplementet til hendelse A er markert i rødt, (b) snittet av A og B er markert i rødt, (c) unionen av A og B er markert i rødt.

$\overline{A}$ er komplementet til hendelse A, og representerer alle elementer som ikke er i A.

$A \cap B$ er snittet av to hendelser A og B, alle utfall som finnes i både A og i B.

$A \cup B$ unionen av hendelse A og B, alle elementer som befinner seg i enten A eller i B.

To hendelser er kun uavhengige hvis $P(A \cap B) = P(A) * P(B)$.

Teorem 2.1: Hvis en operasjon kan bli utført på n_1 måter, og for hver av disse kan en annen operasjon bli gjort på n_2 måter, kan disse to operasjonene bli utført på $n_1 * n_2$ måter. For en sekvens på k trinn med forskjellige valg, kan operasjonen utføres på $n_1 * n_2 * n_3 ..... * n_k$ måter.

Ordnet utvalg:

Permutasjon er en ordning av alle element i en mengde.

Teorem 2.2: Antall permutasjoner av n distinkte element er $n! = n*(n-1)...2*1$ (2.6)

Teorem 2.3: Antall permutasjoner av størrelsen r , fra en mengde av n distinkte elementer er

$$ {}_nP_r = \frac{n!}{(n-r)!} \tag{2.7}$$

Uordnet utvalg:

Teorem 2.4: Antall uordnet utvalg av r elementer kan trekkes fra en mengde n element er

$$ {}_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!} \tag{2.8}$$

$P(A)$ er sannsynligheten for hendelsen A i et utfallsrom S, og følgende er sant:

$0 \le P(A) \le 1, \quad A \in S \quad \text{og} \quad P(S) = 1$

Teorem 2.5: Hvis et eksperiment kan resultere i N forskjellige utkom med lik sannsynlighet, og n av disse korresponderer med hendelse A, vil sannsynligheten for hendelse A være

$$ P(A) = \frac{n}{N} \tag{2.9}$$

***Forventningsverdi**:*

La X være en stokastisk variabel med sannsynlighetsfordelingen f(x), da er forventningsverdien til X:

$$\mu = E(X) = \sum_x xf(x) \qquad -\textit{Diskrete utfallsrom} \tag{2.10}$$

$$\mu = E(X) = \int_{-\infty}^{+\infty} xf(x)\ dx \qquad -\textit{Kontinuerlig utfallsrom} \tag{2.11}$$

Varians:

La X være en stokastisk variabel med sannsynlighetsfordelingen f(x), og forventningsverdi E(X), da er variansen til X:

$$\sigma^2 = Var(X) = \sum_x (x-\mu)^2 f(x) \qquad -\textit{Diskrete utfallsrom} \tag{2.12}$$

$$\sigma^2 = Var(X) = \int_{-\infty}^{+\infty} (x-\mu)^2 f(x)\ dx \qquad -\textit{Kontinuerlig utfallsrom} \tag{2.13}$$

Bernoulli prosess:

En Bernoulli prosess må ha følgende egenskaper:

1. Eksperimentet inneholder n forsøk.
2. Hvert forsøk har et resultat som kan klassifiseres som suksess eller fiasko.
3. Sannsynlighet for suksess p, er konstant for alle forsøk.
4. Alle forsøk er uavhengig av hverandre.

$$b(x;n,p) = \binom{x}{p} p^x (1-p)^{n-x},\ \ x=0,1,2,....n\ , \tag{2.14}$$

forventningsverdi og varians i fordelingen er

$$\mu = E(x) = np \quad \text{og} \quad \sigma^2 = Var(x) = np(1-p)$$

Hypergeometrisk fordelingen:

En hypergeometrisk fordeling må ha følgende egenskaper:

1. Av en mengde N, kan k klassifiseres som suksess, og N−k som fiasko.
2. Det trekkes et tilfeldig utvalg av størrelse n av en mengde N uten tilbakelegging.

Antall suksess kan da beskrives med følgende fordeling:

$$f(x) = h(x;N,n,k) = \frac{\binom{k}{x}\binom{N-k}{n-x}}{\binom{N}{n}}, \quad x=0,1,2,....n \quad , \tag{2.15}$$

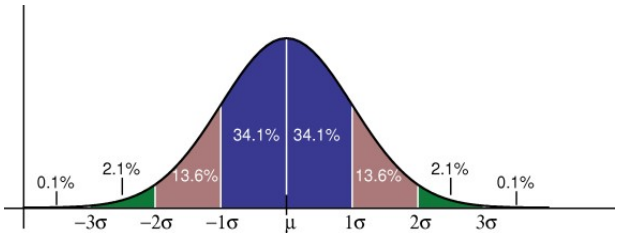
hvor forventningsverdi og varians i fordelingen er

$$\mu = E(x) = \frac{nk}{N} \quad \text{og} \quad \sigma^2 = Var(x) = \frac{N-n}{N-1} * \frac{nk}{N} * (1-\frac{k}{n})$$

Normalfordeling:

Tetthetsfunksjonen for en stokastisk variabel X, med forventningsverdi μ og varians σ^2 er

$$n(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-(1/2)[(x-\mu)/\sigma]^2} \tag{2.16}$$



Figur 6: Normalfordelingen hvor 1,2 og 3 σ avvik fra μ er oppmerket. Bildet er hentet fra: http://www.cs.princeton.edu/introcs/11gaussian/images/

2.5.2 Bruk av feilprøver

Denne fremgangsmåten er hentet fra [9].

En metode for å beregne dekningsgraden av et testvektorsett, er å kjøre en komplett test av alle mulige feilmodeller i en krets. Dette kan ofte bli veldig tidkrevende, og ved å begrense seg til kun en prøve av alle feilmodellene, kan simuleringstiden forkortes. En feilprøve(fra eng. *fault sample*) er en tilfeldig utvalgt delmengde av alle feilmodellsett i en krets. Denne delmengden vil vanligvis inneholde en brøkdel av det totale antallet av feilmodeller. Ved simulering mot feilprøven vil den oppnådde dekningsgraden representere et estimat av den totale dekningsgraden for alle feilmodeller i kretsen, for et gitt sett av testvektorer. Hvor stor feilmargin et slikt estimat har er avhengig av størrelsen på prøven, hvor feilmarginen blir mindre for større prøver. Prøvens størrelse kan bestemmes ut i fra ønsket feilmargin, og denne verdien er uavhengig av totalt antall feilmodellsett i kretsen.[9]

For å kunne gjøre en statistisk analyse av feilmarginen i dette estimatet, benyttes kuler i en boks som statistisk modell. Kulene i boksen representerer feilmodellene i kretsen, hvor svarte kuler er detekterte feil og hvite kuler er udetekterte feil. Med et gitt testvektorsett og en gitt krets, vil antall hvite og svart kuler i boksen være bestemt på forhånd, men ikke kjent. Forholdet mellom antall hvite og svarte kuler er den korrekte verdien på dekningsgraden, og er den verdien som skal

estimeres ved hjelp av den utvalgte feilprøven.
Hvis feilprøvene ble lagt tilbake i boksen etter utvelgelsen, kalles modellen utvalg med tilbakelegging. I utvalg med tilbakelegging vil sannsynligheten for å plukke en svart kule være konstant. Et slikt scenario er lettere å analysere, men ulempen er muligheten for å plukke opp samme kule flere ganger. Når man gjør et utvalg av feilprøver er ikke dette hensiktsmessig, og derfor vil utvalg uten tilbakelegging benyttes.

I [9] benyttes følgende variabler i den statistiske analysen av prosessen som styrer feilprøve utvelgelsen.
N_P er antall kuler i boksen, som tilsvarer det totalt antallet av feilmodeller i kretsen.
$C*N_P$ representerer antall feil som detekteres av testvektorene, hvor **C** er den korrekte og ukjente feildekningsgraden.
N_S er størrelsen på utvalget av kuler fra boksen, altså feilprøven. $N_s \ll N_p$
$x*N_S$ er antall feilmodeller i prøven detektert av vektorsettet, hvor **x** er den målte dekningsgraden.
c er en variabel som representerer dekningsgraden i en tilfeldig feilprøve.

Et utvalg av N_S feilmodeller fra det totale antallet N_P, kan gjennomføres på K forskjellige måter.

$$K_{N_S/N_p} = \binom{N_p}{N_s} = \frac{N_p!}{N_s!(N_p - N_s)!} \tag{2.17}$$

De detekterte feilmodellene i $x*N_s$ må være en del av $C*N_p$, ettersom N_s er en del av N_p. Det samme argumentet kan også brukes for $(1-x)*N_s$, som må være en del av $(1-C)*N_p$. (De udetekterte feilene i prøven, må være et utvalg av alle udetekterbar feil i kretsen.)

Antall måter K_x en prøve med dekningsgrad x kan velges ut, er produktet av antall kombinasjoner detekterte feil og udetekterte feil kan utvelges fra det totale antallet.

$$K_x = \binom{CN_p}{xN_s} * \binom{(1-C)N_p}{(1-x)N_s} \tag{2.18}$$

Sannsynligheten for en feilprøve gir verdien x for den stokastiske variabelen c er:

$$P(c=x) = \frac{\binom{CN_p}{xN_s} * \binom{(1-C)N_p}{(1-x)N_s}}{\binom{N_p}{N_s}} \tag{2.19}$$

Den stokastiske variabelen c kan kun ta diskrete verdier, $1/N_s$, $1/2N_s$, ...1. Når N_s blir stor kan c oppfattes som enn kontinuerlig variable, og funksjon *2.19* kan tilnærmes en normalfordeling, med standardavvik $E(c) = C$ og varians $Var(c) = \sigma^2$.

$$p(x) = Prob(x \le c \le x+dx) = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(x-C)^2}{2\sigma^2}} \tag{2.20}$$

Variansen Var(c) er gitt av

$$Var(c) = \sigma^2 = \frac{C(1-C)}{N_s}\left(1-\frac{N_s}{N_p}\right) \approx \frac{C(1-C)}{N_s} \tag{2.21}$$

Leddet $\left(1-\frac{N_s}{N_p}\right)$ representerer feilutvelgelse uten tilbakelegging, og går mot 1 når $N_s \ll N_p$.

Uttrykket kan derfor forenkles, *funksjon 2.21*, slik at variansen er uavhengig av N_p, når antall feilprøver er få og populasjonen er stor. Dette resulterer også i at utvelgelse av feil, med eller uten tilbakelegging, kan uttrykkes med samme funksjon. Variansen uttrykker hvor mye den målte verdien c avviker fra den korrekte verdien C. Ut i fra estimatet *2.21* kan en se at variansen er i stor grad avhengig av størrelsen av N_S. Størrelsen av prøven N_s bør derfor bestemmes på bakgrunn av ønsket nøyaktighet, ikke basert på størrelsen av kretsen(som er en vanlig feiltakelse [9]). Hvis 1000 prøver fra en populasjon på 10.000 feil gir god nok nøyaktighet, vil også 1000 feil fra en populasjon på 100.000 feil gi tilnærmet samme nøyaktighet. Noe som reduserer størrelsen på prøven til 1% av populasjonen, og som demonstrerer hensikten til dette estimatet. Enda populasjonen øker, vil en prøve på 1000 feil gi samme nøyaktighet.

Basert på normalfordelingen, er det 0.997 sannsynlighet for at x ligger innenfor 3 standardavvik i forhold til forventningsverdien.

$$|x-C| = 3\sigma \rightarrow |x-C| = 3\sqrt{\frac{C(1-C)}{N_s}} \tag{2.22}$$

fra[9], for $N_s \ge 1000$ kan følgende estimat fremstilles

$$3\sigma\, estimat = x \pm \frac{4.5}{N_s}\sqrt{1+0.44N_s x(1-x)} \tag{2.23}$$

(Eksperimentelle resultater viser at funksjonen gir gode resultat[9])

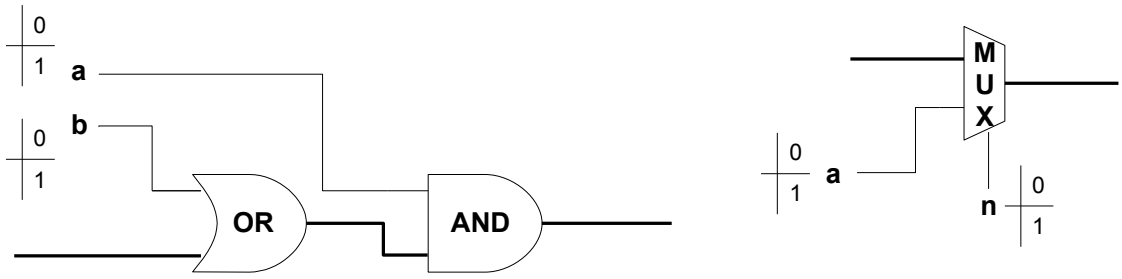
3 Feiltest implementasjoner

Ved bruk av FPGA i simuleringen kan DFT(Design For Test) verktøy benyttes, som er svært tilgjengelig og godt dokumentert.

3.1 Design for test

DFT er en kretsimplementasjon hvor kretslogikk utarbeidet for feiltest er implementert i et design. I hovedsak øker FDT muligheten for å kunne observere og kontrollere kretsen under test. Dominerende eller vanskelig tilgjengelig noder kan lettere aksesseres når linjer fra disse punktene direkte dras mot inngangen, eller utgangen av kretsen.[2]

Enkelte av DFT metodene utnytter også operasjonsfrekvensen på kretsen, noe som er spesielt gunstig for kretser med høy klokkehastighet. Rent økonomisk vil denne teknikken i mange tilfeller forberede de totale utviklingskostnadene for en krets, enda designkostnadene isolert sett nesten alltid vil øke. Ulempene med DFT er i hovedsak økt areal av kretsen, og mulig redusert ytelse pga av lengre signalvei.



Figur 7: Eksempel på ad-hoc realisering med porter til venstre, og multiplekser til høyre.

Ad-hoc, fra [2], er en metode utviklet for å kunne aksessere områder av KUT(Krets Under Test) som er vanskelig å teste. En vanlig realisering av denne metoden er å implementere multipleksere på interne noder i KUT for å muliggjøre en eller flere testtilstander, aksessbare fra den primære inngangen eller utgangen. Multipleksere kan f.eks kobles til det primære utgangssignalet slik at interne noder kan observeres direkte, eller de kan kobles mot det primære inngangssignalet for å kunne kontrollere testpunkter. En annen realisering av testpunktstyring er å bruke AND- eller OR-porter til å sette noder hardt til logisk 0 eller 1 vist i *figur 7*. Ved bruk av porter for å realisere testpunkter reduserer man arealkostnaden, men dette vil kun gi kontrollbarhet, ikke aksessbarhet. En av ulempene ved bruk av ad-hoc er at flere I/O-pinner blir introdusert for å kunne aksessere testtilstandene. I tillegg er utvelgelsen av testpunkt vanligvis manuell i ad-hoc, og fører til betydelig tidspålegg.

«Scan design» er en annen populær DFT løsning hvor det benyttes skannkjeder som knyttes til registre i kretsen. I hovedsak introduserer «scan design» en metode for å kunne aksessere alle vippene i en krets. Dette forenkler testingen ettersom man kan benytte kombinatoriske testmønstre for sekvensielle kretser.

Fra [2] utføres «scan design» ved at man tilknytter en multiplekser på inngangen til hver vippe i kretsen. Disse knyttes sammen til en kjede, ved at utgangen på hver vippe kobles til inngangen på en annen multiplekser tilknyttet nærmeste nabovippe. Den første vippen i kjeden er tilknyttet inngangen av kretsen, mens den siste i kjeden er koblet til utgangen, slik at logiske verdier kan både skiftes inn og ut av vippene. Et styresignal bestemmer hvilken av de to inngangene på multiplekserene som er aktiv, og dette bestemmer tilstanden til systemet. Ved tilstand «test mode» settes multiplekserene slik at vippene er koblet til kretsen som de var opprinnelig uten «scan design», mens ved «scan mode» kobles vippene til den introduserte skannkjeden.

Testprosessen utføres ved at verdier skiftes ut og inn fra vippene i «scan mode», mens testvektorer påtrykkes inngangen av kretsen i «test mode». Resultatet fra testen baserer seg på responsen fra utgangen av kretsen, og informasjonen som skiftes ut fra vippene. «Scan design» kan fullstendig automatiseres, og kan oppnå tilnærmet 100% feildekning uten bruk av sekvensiell feilsimulering.[2]

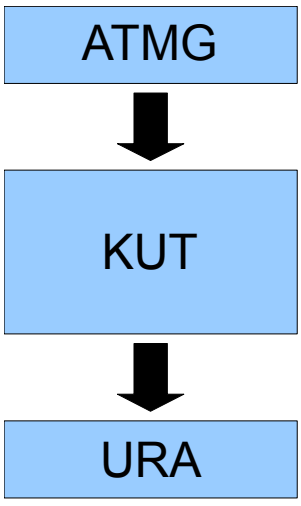


Fig 8: BIST realisering.

BIST(Built In Self Test) er en DFT teknikk og kretskonfigurasjon som gjør det mulig for en brikke å teste seg selv, hvor testmønsteret genereres og testresponsen analyseres på brikken. En testmønstergenerator, vanligvis av typen LFSR eller CA, påtrykker inngangen til KUT med et vektorsett, og en signaturanalyse blir gjennomført på utgangen, se *figur 8*. En fordel med BIST er muligheten for å kunne påtrykke testvektorer med samme hastighet som operasjonsfrekvensen, og den implementerte testlogikken kan benyttes etter produksjon som et diagnoseverktøy ved systemfeil.[1] En annen fordel er at prosessen krever lite interaksjon ved en testsekvens, dog ikke like automatisert som «scan-design».[2] Testmønstrene for DFT løsningene ad-hoc og «scan design» bruker utvalgte vektorer for spesifikke feil, mens BIST benytter langt flere testmønstre, som har mulighet til å detektere mer enn én feil.[2] Som andre DFT løsninger lider BIST av det faktum at større arealforbruk øker produksjonskostnadene, og implementerte kretselement kan introdusere forsinkelser i signalveier, som videre kan redusere den normale operasjonsfrekvensen.

Fremgangsmåten for ekvivalentsjekk i denne oppgaven har mange likhetstrekk med DFT. Dette betyr at modeller og verktøy utviklet i forbindelse med DFT, kan også brukes i ekvivalentsjekk applikasjonen.

3.2 Testmønster generatorer

Testmønstergeneratorer klassifiseres ofte i grupper basert på hvordan de er oppbygd, hvilke mønster de produserer, og eventuelt hvilke feil de skal detektere. Fra [2] er de delt opp i deterministisk-, algoritmisk-, fullstendig-, pseudofullstendig-, pseudotilfeldig- og vektet pseudotilfeldig testmønster. Deterministisk testmønster er utviklet for å detekterer spesielt utvalgte feil og strukturelle defekter for en KUT. Disse testvektorene kan f.eks lagres på en ROM, og sendes ut ved hjelp av en teller. Algoritmisk testmønster har også et spesifikk testmønster for en type KUT, men for å realisere et algoritmisk testmønster benyttes en tilstandmaskin. Slike testmønstre brukes ofte i sammenheng med RAM strukturer.[2]

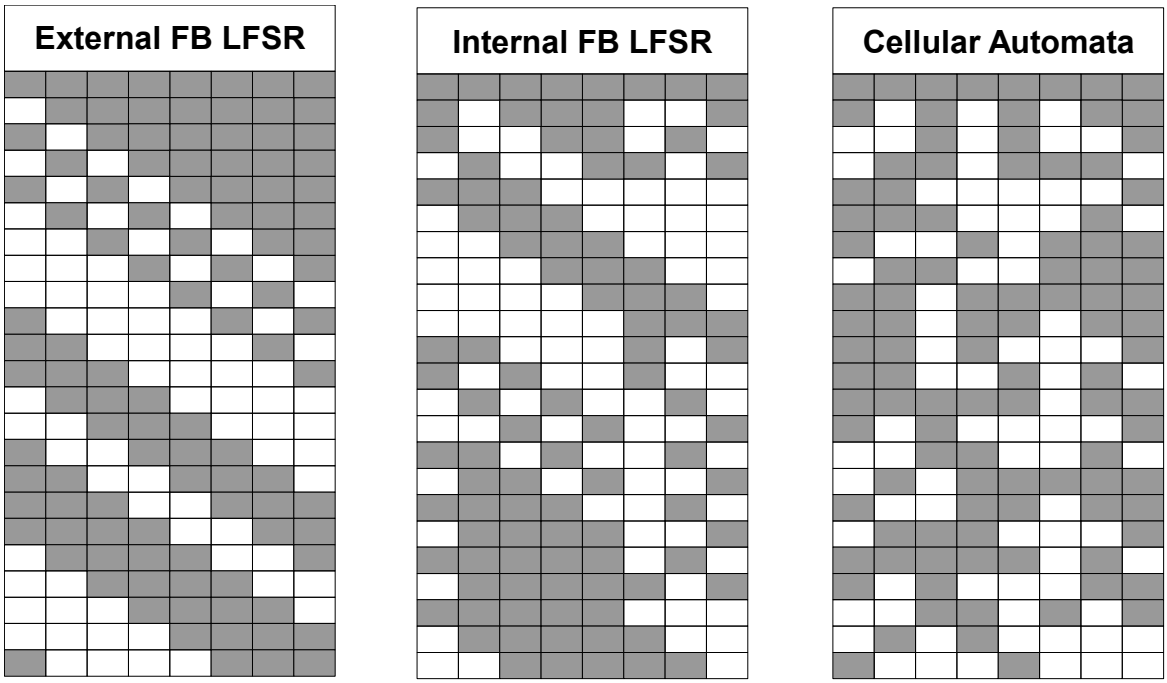
Fullstendig testmønster beskriver et mønster som inneholder alle mulige kombinasjoner av en vektor av en bestemt størrelse. Et fullstendig testmønster på n-bit vil inneholde 2^n forskjellige vektorer. En n-bit binær teller vil generere et slikt antall vektorer, men som testmønster er den lite egnet. I sekvensen vil LSB(Least Significant Bit) endres hver eneste klokkesyklus, mens MSB(Most Significant Bit) kun vil bli endret halvveis og på slutten av sekvensen. Derfor er tellere i praksis sjelden brukt som generator for testmønstre. De er heller brukt i forbindelse med responsanalyse, eller i kombinasjon med annen logikk for produksjon av algoritmisk testmønster.[2]

Pseudofullstendig testmønster inneholder alle testvektorer for en utvalgt del av en KUT. Hvis k bit av et større testmønster aksesserer alle deler av en KUT, vil testmønsteret bestå av 2^k forskjellige vektorer i et pseudofullstendig mønster.

Pseudotilfeldig testmønster er det vanligste mønsteret produsert av en automatisk testmønster generator i BIST.[2] I hovedsak er det LFSR(Linear Feedback Shift Register) og CA(Cellular Automata) algoritmer som produserer disse mønstrene. Mønsteret beskrives som pseudotilfeldig fordi vektorene har mange egenskaper som samsvarer med tilfeldighet, men de er likevel korrelerte. For eksempel er fordelingen av logiske 0'ere og 1'ere er likt fordelt mønsteret, og i en MLS sekvens vil alle vektorene bli representert like mange ganger. Et problem er at mønsteret vil gjenta seg selv når sekvensen er ferdig. Vektet Pseudotilfeldig testmønster er i utgangspunktet et LFSR eller CA mønster som er filtrert vha. AND/NAND eller OR/NOR porter. Dette for å produsere flere logiske 0'ere eller 1'ere i testmønsteret, rettet mot spesielle områder av KUT. I CA og LFSR generatorer vil den maksimale lengden være 2^n-1, ettersom vektoren «00..0» ikke er representert i disse sekvensene, mer om dette i 3.3.

3.3 Pseudotilfeldige generatorer

I denne oppgaven vil pseudotilfeldige generatorer gjelde CA og LFSR generatorer, dog er de også de vanligste. Vektorene som produseres av CA og LFSR er pseudotilfeldig pga at testmønsteret har mange egenskaper som samsvarer med tilfeldighet, men er likevel korrelerte. En av egenskapene som samsvarer med tilfeldighet, er fordelingen av logiske 0'ere og 1'ere i testmønsteret som er likt. Som *figur 9* viser, er tilfeldigheten i et LFSR mønster sterkt begrenset, og da spesielt i External-XOR LFSR hvor en tydelig korrelasjon i mønsteret kan observeres.



Figur 9: Tre forskjellige testemønstergeneratorer på 8 bit, her presentert med 23 steg. De grå feltene representerer logisk 1, og hvit representerer 0 i testmønsteret, fra [2].

Både CA og LFSR er av typen FSM(Finite State Machine), og de generatorene aktuelle i denne oppgaven vil også være av typen ALM(Autonomous Linear Machine). ALM er en lineære FSM uten inngangsverdier. Når en LFSM(Lineær FSM) er brukt som generator i BIST, sekvenser den gjennom et bestemt antall tilstander, hvor hver tilstandkode fungerer som en testvektor(*figur 13*). En n-bits LFSM testvektor generator vil inneholde n antall vipper(referert som celler), og produserer et n-bit bredt mønster fra utgangen av vippene. Hvor mange forskjellige vektorer mønsteret består av bestemmes av et polynom, og de polynomene som har n-1 forskjellige vektorer i sekvensen, refereres til som primitive polynomer, mer om dette i 3.4 og 3.5.

Følgende notasjoner benyttes for å beskrive en CA eller LFSR.

`i`: Posisjonen av en individuell celle i en én-dimensjonal rekke av celler.

`t`: Tids steg.

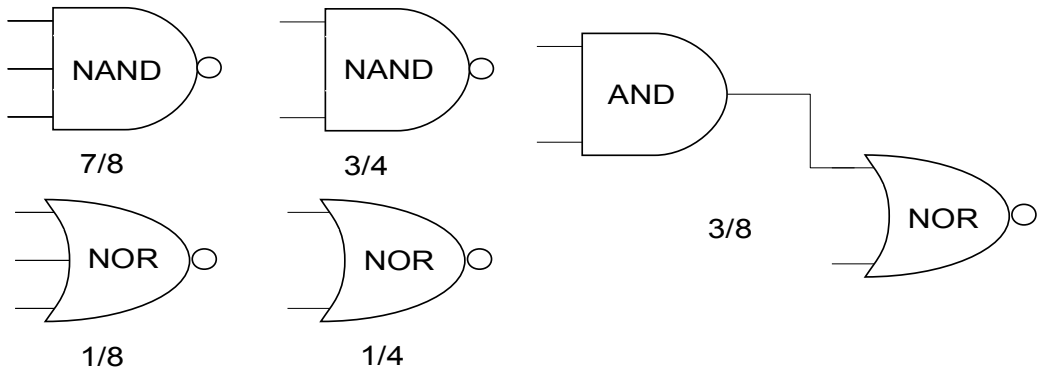
$X_i(t)$: Utgangsignalet fra celle nummer i ved tiden t.

$X_i(t+1)$: Utgangsignalet fra celle nummer i ved tiden $t+1$.

Her vil også nummereringen av celler alltid være fra venstre til høyre.

3.3.1 Vektet pseudotilfeldige generatorer

Testvektorer for en KUT kan noen ganger oppnå lav feildekning ved bruk av pseudotilfeldig testmønstre. En mulig forklaring på dette er at inngangen inneholder et stort antall av AND/NOR eller NAND/OR funksjoner. Et stort antall AND/NOR funksjoner vil produsere en overvekt av logisk 1, mens et stort antall NAND/OR vil produsere en overvekt av logisk 0. Dette er en konsekvens av at logiske 0'ere og 1'ere er jevnt fordelt i et pseudotilfeldig testmønster. Feil som krever mange logiske 1'ere på utgangene av en stor AND funksjon, eller logiske 0'ere på på utgangene av en stor OR funksjon er ikke lett å detektere, og er referert som «tilfeldig-mønster-resistente feil».[2] En metode for å øke dekningsgraden for «tilfeldig-mønster-resistente feil», er å bruke et vektet vektorsett for den delen av KUT dette gjelder. Hver boolske verdi i vektoren fra en RPG(Random Pattern Generator) vil ha et snitt på 0.5, men ved hjelp av vektet funksjoner, som i *figur 10,* kan andre verdier introduseres. F.eks vil en NAND-port med 3 innganger forskyve snittet på en verdi til 7/8. Et annet bruksområde for vektet testmønster er når KUT har en global reset eller preset til vippene, slik at de ofte blir nullstilt av det tilfeldige testmønsteret. Dette fører til at testdata som er forplantet i vippene i kretsen blir slettet, og hindrer at interne feil blir detektert.[2]

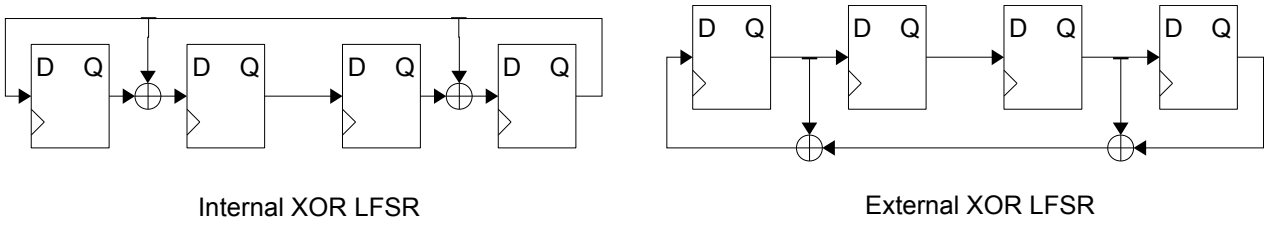


Figur 10: Vanlige vektet funksjoner.

3.4 Linear Feedback Shift Register

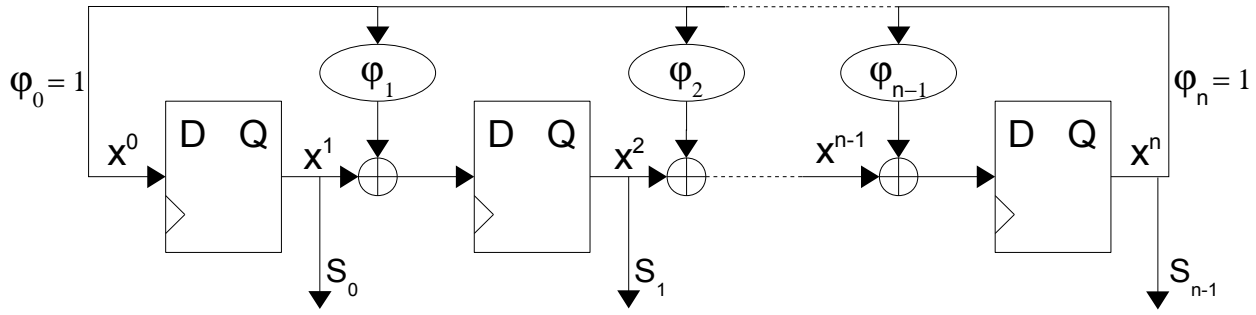
LFSR er en av de mest brukte utførelsene av testvektorgenerator i BIST applikasjoner, og en av grunnene til det er at utførelsen er enda mer arealeffektiv enn en teller[2]. I hovedsak består en n-trinns LFSR av et lineært tilbakekoblet skiftregister med X_0, X_1, X_2, . . . X_{n-1} vipper, og for hver tilstand kan en testvektor avleses på utgangen.

De to vanligste måtene å realisere et LFSR register på er «internal-XOR» og «external-XOR» i *figur 11*, hvor tilbakekoblingene blir realisert med XOR-porter.[1] Begge benytter et likt antall logiske komponenter for å realisere registeret.



Figur 11: Til venstre et 4-bits LFSR register med "internal-XOR" tilbakekobling, og til høyre en tilsvarende LFSR med "external-XOR" tilbakekobling.

«Internal-XOR» har en tilbakekobling fra utgangen av den siste vippen i skiftregisteret, og tilbake til inngangene på en utvalgt gruppe vipper. «External-XOR» har en tilbakekobling fra en utvalgt gruppe vipper til den første vippen i skiftregisteret.
«External-XOR» har i verste tilfelle to XOR-porter i tilbakekoblingen fra utgangen av den siste vippen i rekken, til inngangen til den første rekken. «Internal-XOR» har på den andre siden alltid kun én XOR-port i den samme tilbakekoblingen, og vil være den LFSR implementasjonen med høyest operasjonsfrekvens.[2]



Figur 12: Generell internal-XOR LFSR modell, hvor φ representerer mulige tilbakekoblinger bestemt av polynomet.

Hvilke vipper som inngår i tilbakekoblingen i en LFSR kan beskrives i et polynom

$$P(x) = \varphi_0 x^0, \ \varphi_1 x^1 \ldots \varphi_n x^n, \tag{3.1}$$

hvor $x^0, \ x^1, \ldots x^n$ representerer graden av polynomet, og $\varphi_0, \ \varphi_1 \ldots \varphi_{n-1}$ har verdien 1 når en tilbakekobling forekommer, se *figur 12*. På kantene av et LFSR register har φ_0 og φ_n alltid verdien 1, og de introduserer ikke ekstra XOR-porter i designet, enda de er tilkoblet til de[1].
Verdiene som kan avleses på utgangen av vippene kan representeres som $S_0, \ S_1, \ldots S_{n-1}$, og utgangsvektoren blir da

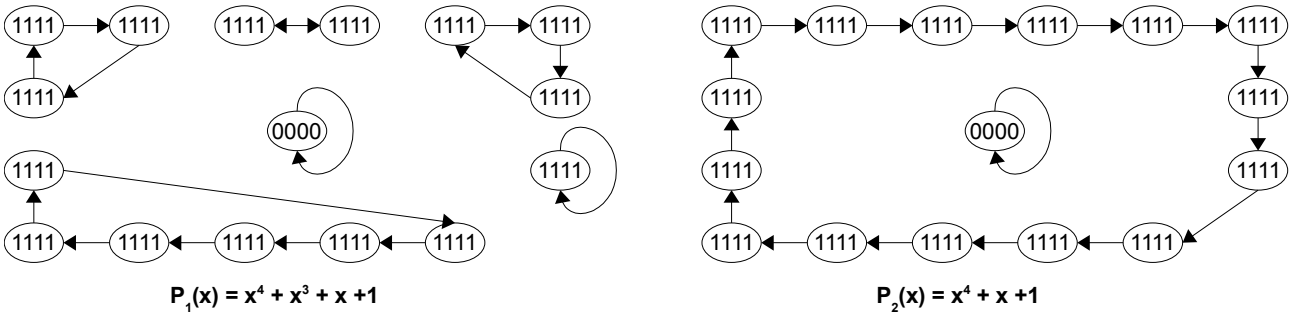
$$\sigma(\tau) = \begin{bmatrix} S_0(\tau) \\ S_1(\tau) \\ . \\ . \\ S_{n-1}(\tau) \end{bmatrix} \text{ ,ved en gitt tid } \tau. \tag{3.2}$$

Den neste utgangsvektoren blir

$$\sigma(\tau+1) = T * \sigma(\tau) \tag{3.3}$$

hvor T er nestetilstandsmatrisen til LFSR registeret.

Felles for alle LFSR polynom er at de har en nulltilstand som ikke leder til noen andre tilstander, og på grunn av dette er den lengste mulige sekvensen 2^N-1, og refereres som MLS(Maximum Length Sequence). Polynomer som resulterer i maksimal lengde sekvenser er kalt primitive polynomer, og polynomer som ikke produserer maksimal lengde sekvenser blir kalt ikke-primitive polynomer[2]. I vedlegg A er det en oversikt over noen primitive polynomer for LFSR, med minimum antall XOR-porter.



Figur 13: Tilstandsdiagram for to forskjellige LFSR med internal-XOR tilbakekobling. $P_1(x)$ er et ikke-primitivt polynom, mens $P_2(x)$ er et primitivt polynom, og har derfor en MLS sekvens.

Fra [2], nulltilstanden produserer mønsteret «000..000», eller «All 0s Pattern», og for å inkludere dette mønsteret i den pseudotilfeldige sekvensen må LFSR modulen modifiseres. Ekstra logikk legges til ved at utgangsignalet fra alle trinnene, bortsett fra x^n, ledes til en NOR-port, hvor utgangsignalet fra porten ledes til en XOR port sammen med tilbakekoblingsignalet. Resultatet av dette er at tilbakekoblingsignalet blir påvirket under «000.001» tilstanden slik at den produserer «All 0s Pattern». Etter nulltilstanden vil LFSR generatoren fortsette den vanlige sekvensen. En slik modifikasjon blir noen ganger kalt en «de Bruijn Counter», eller «Complete Feedback Shift Register» CFSR. Arealøkningen for å produsere «All 0s Pattern» er en N-1 NOR port og en XOR port.

3.5 Cellular Automata

CA er som nevnt en pseudotilfeldig testvektorgenerator, men har i forhold til LFSR mindre korrelasjon i testmønstersekvensen. Denne egenskapen kommer på bekostning av å være mindre arealeffektiv, ettersom et CA register er mer komplisert oppbygd og benytter flere XOR porter.[2]

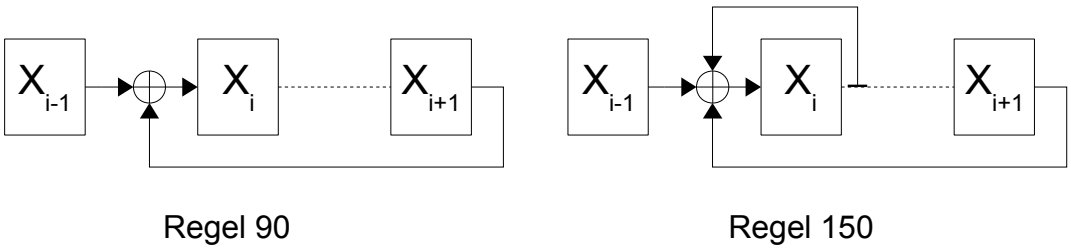
CA er bygd opp om som en FSM tilsvarende LFSR, men nestetilstanden til en celle $X_i(t+1)$ er bestemt av et logisk forhold mellom tilstanden til tre naboceller:

$$X_i(t+1) = f\{X_{i-1}(t) + X_i(t) + X_{i+1}(t)\}, \quad (3.4)$$

hvor *f* er en logisk funksjon, referert til som «regel» i CA.

I en «to-tilstand» CA med nabofunksjon fra 3 celler, kan det være 2^3 forskjellige naboforhold konfigurasjoner. For en slik CA kan det være totalt 2^{2^3} (256) unike tilordninger fra naboskapkonfigurasjonene til nestetilstanden. Hver tilkobling er kalt «regel».[13] Hvis CA-regelen involverer kun XOR logikk, er den kalt en lineær regel, og med kun XNOR logikk refereres de som komplementære regler. Et CA register oppbygd av celler med kun lineære regler omtales som en lineær CA,[13] og benyttes i denne oppgaven. I en lineær CA vil nestetilstanden til en celle $X_i(t+1)$ være en lineær funksjon av cellens nåtilstand $X_i(t)$, og tilstanden til de to nærmeste nabocellene $X_{i-1}(t)$, $X_{i+1}(t)$.

Hvis CA registeret er bygd opp av celler med forskjellige regler, omtales det som HCA(Hybrid Cellular Automata), eller LHCA(Linear Hybrid Cellular Automata) når en kombinasjon av lineære regler benyttes.[2]



Figur 14: Konfigurasjon av naboforholdet i CA for henholdsvis regel 90 og 150.

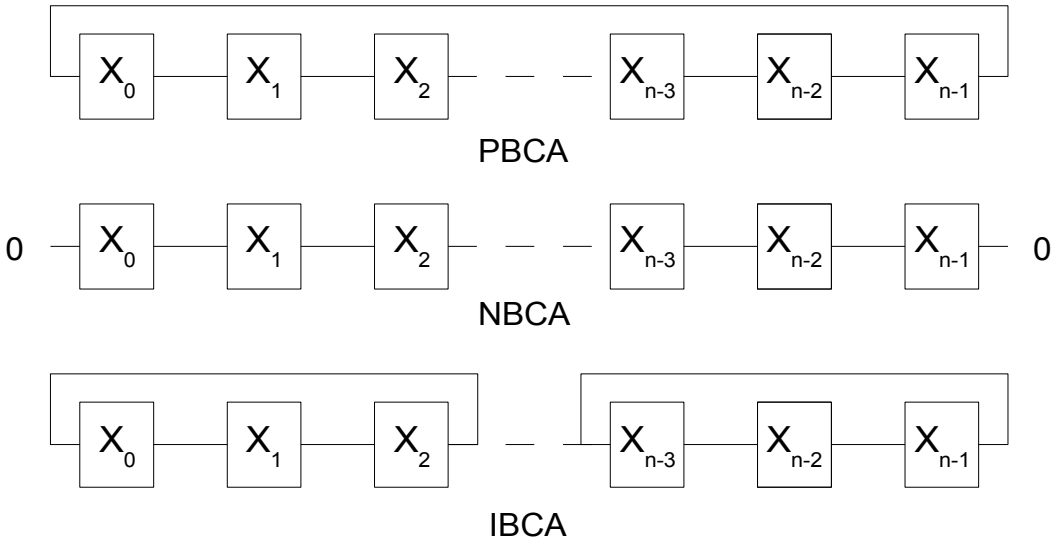
Regel 90: $X_i(t+1) = X_{i-1}(t) \text{ XOR } X_{i+1}(t)$ (3.5)

Regel 150: $X_i(t+1) = X_{i-1}(t) \text{ XOR } X_i(t) \text{ XOR } X_{i+1}(t)$ (3.6)

I denne oppgaven vil kun én-dimensjonale CA registre utformes, med en kombinasjon av de lineære reglene 90 og 150, se *figur 14*. I vedlegg A kan man finne kombinasjoner av regel 90/150 som danner primitive polynom for CA registre av forskjellig størrelse. Et register bestående av 4 celler

med regel 90/150, vil med polynomet «0101» gi en MLS på 15 trinn, hvor logisk 1 i polynomet representerer regel 150 i registeret. Det vil si at fordelingen av de to reglene vil være 90, 150, 90 og 150 i registeret.

I et CA register med sykliske grensebetingelser(PBCA = Periodic Boundary CA) vil cellene på kantene(X_0 og X_{n-1}) betraktes som hverandres naboer, ved at en lang linje trekkes mellom de. Hvis registeret har null grensebetingelser(NBCA = Null Boundary CA), vil derimot den manglende nabocellen hos første og siste celle i registeret erstattes med den logiske verdien 0.
For å oppnå MLS(Maximum Length Sequence) av CA registre med regle 90/150, kan kun NBCA benyttes.[7]



Figur 15: Grensebetingelser for en 2-dimensjonal Cellular Automata.

Null grensebetingelser reduserer kvaliteten på det pseudotilfeldige mønsteret, og på kantene er problemet størst. Dette motiverte [8] å se på en annen løsning av MLS CA enn NBCA, nemlig IBCA(Intermediate Boundary CA). IBCA benytter celle nummer 2 fra venstre og høyre i registeret som nabosignal for kantcellene X_0 og X_{n-1}, istedenfor det substitutt nullsignalet.

$$X_0(t+1) = f\{X_0(t),\ X_1(t),\ X_2(t)\} \tag{3.7}$$

$$X_{n-1}(t+1) = f\{X_{n-3}(t),\ X_{n-2}(t),\ X_{n-1}(t)\} \tag{3.8}$$

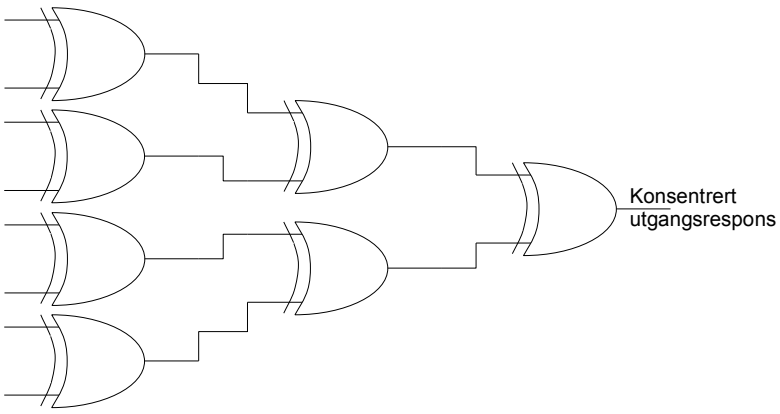
Ved å benytte disse grensebetingelsene oppnådde de høyere kvalitet på testmønsteret, i tillegg til MLS. (noe om hvor mange MLS sekvenser)
NBCA er underlegen IBCA i tilfeldighetsegenskaper. [8] I bitsekvensen produsert av terminalcellene fra en 32 bits CA, hadde IBCA 0,499 av idealet 0,5 i logisk fordeling, mens NBCA

hadde 0,519, i en sekvens av 1000 skift.

3.6 Utgangsresponsanalyse

Når en KUT med m-bits utgang blir påtrykket N_V antall testvektorer, vil utgangsresponsen inneholde totalt mN_V antall logiske verdier. En direkte bit for bit sammenligning med utgangsresponsen fra den feilfrie kretsen, vil i mange tilfeller kreve et stort minneområde for å kunne lagre hele responsen. Ved bruk av komprimeringsfunksjoner kan datamengden reduseres, men tap av informasjon vil ofte være konsekvensen. En vanlig DFT løsning, og spesielt BIST, er oppbygd av en TPG modul og en ORA(Output Response Analysis) modul, se *figur 8*. En ORA modul har i oppgave å komprimere utgangresponsen fra KUT til en enkel godkjent/ikke godkjent indikasjon.[2]

Konsentratorer reduserer antall utgangsverdier fra KUT som må kontrolleres. Vanligvis benyttes konsetratorer i sammenheng med andre ORA teknikker for å effektivisere arealforbruket, ettersom ORA da har færre verdier å behandle.[2] En typisk realisering av en konsentrator er å komprimere en flere bit bred utgangsverdi til kun ett bit, ved hjelp av et tre av kombinerte XOR-porter. For en N-bit utgangsrespons er det nødvendig med N-1 antall XOR-porter i en den lineære kombinatorikken som redusere signalet til ett bit. Et eksempel på en slik konsentrator er vist i *figur 16*, hvor et 8-bit bredt signal reduseres til et signal på 1-bit ved hjelp av 7 XOR-porter.



Figur 16: 8-bit konsentrator av 7(8-1) XOR-porter.

Dette 1-bit signalet vil vanligvis analyseres av en ORA, som arealbegrenses pga reduksjonen av signalets bredde.

En komperator er populær pga lavt arealforbruk. For å realisere en N-bit komparator er det nødvendig med N antall XOR-porter, og en N-bit OR/NOR-port [2](som realiseres av N-1 antall OR/NOR porter). Et bruksområde for en komparator er når en krets er bygd opp av flere like kretselement. Når de er feilfrie, vil disse kretsene ha lik utgangsrespons når samme testmønster

påtrykkes inngangene. Derfor kan en komparator kobles til utgangen av kretselementene, for å kontrollere og sammenligne utgangsresponsen. En forskjell i utgangsresponsen mellom kretselementene vil indikere en feil, og komparatoren vil registrere feilen på sin utgang. Eneste begrensningen i denne metoden er når kretselementene inneholder samme feil, noe som gjør at den feilaktige utgangsresponsen ikke blir detektert.[2]

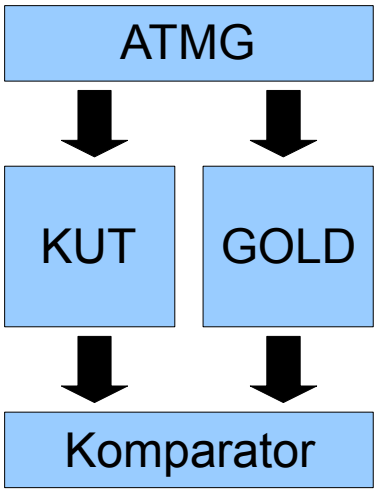
Et problem med en komparatorbasert ORA, er at den ikke er fullstendig testet. En låst-til feil i komparatorlogikken, kan forhindre en korrekt detektert feil å forplante seg fram til den overvåkte utgangen av analysen.[2]

Signaturanalyse er ikke relevant i denne oppgaven, men er den mest brukte ORA løsningen i BIST implantasjoner.[2] Ideen bak signaturanalyse er å dividere utgangsresponsen fra KUT med det karakteristiske polynomet i testmønstergeneratoren. Dette gjennomføres i en gjentakende operasjon(modulo operasjon) helt til den resulterende signaturen er av lavere grad enn generator polynomet. For å verifisere om KUT har korrekt utgangrespons, sammenlignes signaturene på slutten av testsekvensen. En svakhet i denne metoden er når feil i KUT gir en respons tilsvarende TPG polynomet, slik at en endring i utgangsresponsen går gjennom systemet udetektert. Dette kalles signatur aliasing.

4 Ekvivalentsjekk på FPGA

Dette kapittelet omhandler fremgangsmåten for ekvivalentsjekk på FPGA. Denne applikasjonen vil ha to funksjoner, en ekvivalentsjekk og en feildekningsgrad tester. Ekvivalentsjekkapplikasjonen vil inneholde en mønstergenerator og en komparator. Feildekningsgrad beregnes ved hjelp av feilinjisering, kombinert med ekvivalentsjekk. En teller beregner forholdet mellom detekterte feil og antall injiserte feil.

4.1 Design av ekvivalentsjekk



Figur 17: Fremgangsmåte for ekvivalentsjekk i denne oppgaven. ATPG = LFSR/CA

Til tross for mange feiltestreferanser i denne oppgaven, er det ekvivalentsjekk denne oppgaven handler om. Mulighet for deteksjon av flere feil er f.eks ikke nødvendig, men kun verifikasjon om kretsene er ekvivalent eller ikke. Begge kretsene lastes inn på samme FPGA, noe som betyr at direkte sammenligning kan gjennomføres uten bruk av minne.

Fremgangsmåten for ekvivalentsjekk vises i *figur 17*, hvor en testmønstergenerator påtrykker testevektorer på inngangen til KUT og en referansekrets GOLD. Utgangresponsen fra de to kretsene sammenlignes, og avvik registreres.

Denne løsningen vil ikke bruke noe ekstra minneelementer, og komparatoren(ORA) sammen med en CA/LFSR er en enkel og arealeffektiv løsning. For å rute modellene sammen benyttes et toppnivå skall kalt «TOPP». TOPP inneholder også en tilstandsmaskin for å kontrollere feilinjisering ved feildekningsgrad testing.

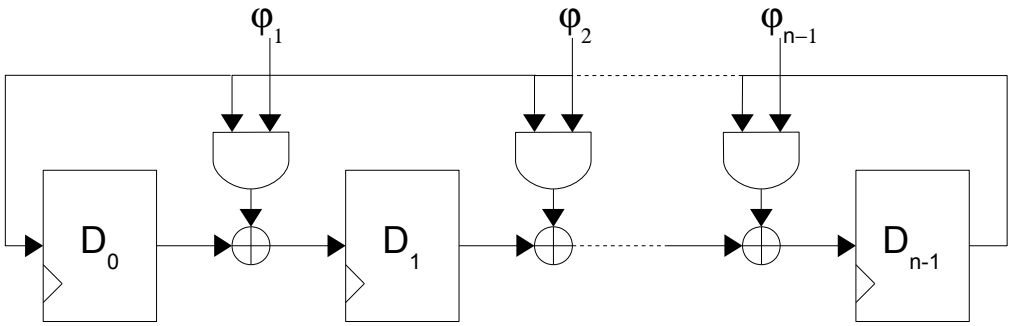
Modulene CA, LFSR, ORA og TOPP er representert i vedlegg B i form av VHDL kode.

4.1.2 LFSR mønstergenerator

Testmønstergeneratoren i denne oppgaven er av typen LFSR eller CA. Begge generatorene er realisert ved hjelp av registre med primitive polynomer, og de initialiseres til samme starttilstand(111...11).

Figur 18 viser den generelle LFSR løsningen som benyttes i denne oppgaven. AND-portene tilknyttet polynomene φ_0, φ_1,..φ_{n-1} som holder verdien 0 vil ha en utgangrespons låst til 0. Dette

vil låse den ene inngangen til den tilknyttede XOR-porten, og vil gi porten samme logiske egenskaper som en linje sett fra den andre inngangen. Det vil si at den boolske verdien påtrykket den andre inngangen på XOR-porten vil være den resulterende utgangsverdien, og det eneste bidraget porten gir er økt forsinkelse. Når en av $\varphi_0, \varphi_1, .. \varphi_{n-1}$ har verdien 1, vil derimot den tilknyttede AND-porten få samme egenskaper som en linje, sett fra andre inngangen, og XOR-porten vil fungere som en tilbakekobling til den tilhørende vippen i registeret.



Figur 18: Et generelt internal-XOR LFSR register.

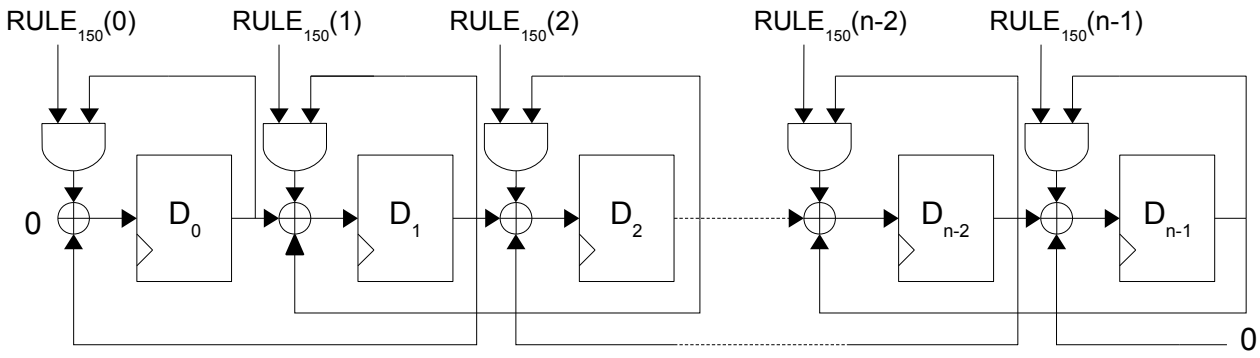
Problemet med denne løsningen er at den er svært lite arealeffektiv. Hvis denne løsningen ble benyttet for en 32-bits LFSR med et primitivt polynom fra vedlegg A, vil kun 27 av de 30 implementerte XOR-portene benyttes som tilbakekobling, og sammenlignet med en skreddersydd løsning ville heller ikke AND-portene benyttes. Dette er en høy pris å betale for en generell løsning, men så lenge areal ikke er en begrensende faktor er dette problemfritt. Ytelsen reduseres med en ekstra AND-port i signalveien, noe som ikke er avskrekkende i forhold til den store arealøkningen. Hvis systemet har høye krav til ytelse, kan dette være kritisk ettersom internal-XOR LFSR i utgangspunktet har kort signalvei.

4.1.2 CA mønstergenerator

En kombinasjon av regel 90 og 150 er en populær løsning for å realisere primitive CA polynom. En enkel måte å beskrive en slik hybrid er vist i vedlegg A, hvor logisk 1 representerer regel 150, og logisk 0 regel 90.

For å oppnå en generell løsning av et CA register med 90/150 regel kombinasjon, kan $X_n(t)$ signalet(se *figur 14*) rutes til en AND-port, som vist i *figur 19*. Dette signalet er en del av tilbakekoblingen for $X_n(t+1)$ i regel 150, men ikke 90.(*funksjon 3.5 og 3.6*) Ved å introdusere 0 eller 1 på den ene inngangen av AND-portene, kan man realisere hvilken som helst kombinasjon av regel 90/150 CA registre. Logisk 1 vil gi cellen regel 150 egenskaper, mens logisk 0 vil gi cellen regel 90 egenskaper. Til tross for at registeret består av celler med forskjellige regler, er de kombinatorisk helt like. For å øke størrelsen på registeret, er det bare å legge til flere celler vha. kaskade kobling, uten å endre noe som helst på de allerede implementerte komponentene. Det

eneste som må justeres er «regel signalet» slik at alle celler enten blir satt til regel 90 eller 150. En annen fordel med denne fremgangsmåten er at polynomene fra vedlegg A kan benyttes som et konstant signal for AND-portene, uten noe som helst konvertering.



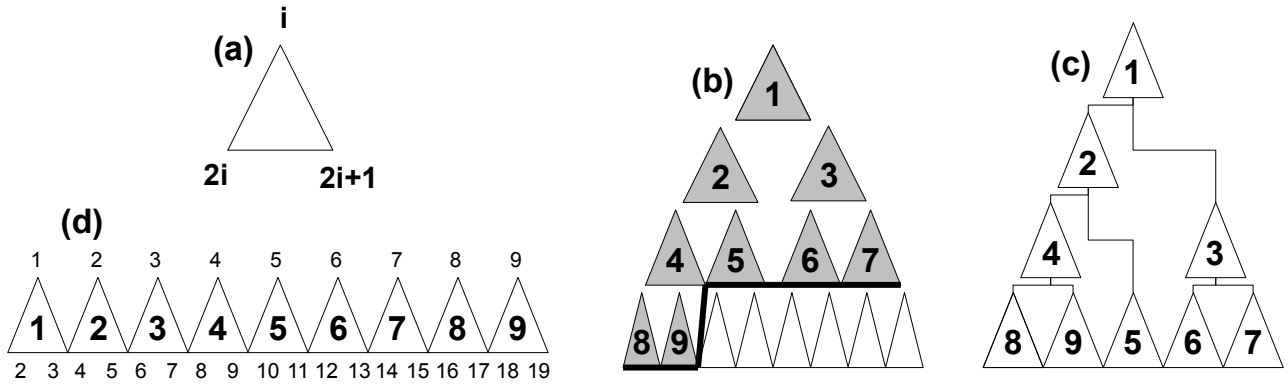
Figur 19: Generell LHCA for regel 90 og 150.

I forhold til den generelle LFSR løsningen, har CA løsningen mye mindre arealøkningen. Noe av grunnen til dette er at et CA register i utgangspunktet har mye et større arealforbruk. I tillegg er denne implementasjonen enklere, med kun en ekstra AND-port per vippe, i forhold til LFSR løsningen som hadde minimum `n-3` ekstra XOR-porter og `n-2` AND-porter for n antall vipper.

4.1.3 Komparator tre

Utgangresponsanalyse kan i hovedsak realiserer på to forskjellige måter. Den ene løsningen er signaturanalyse, som vanligvis brukes i sammenheng med LFSR eller CA i f.eks BIST applikasjoner. Den andre løsningen er en komparatorbasert løsning, som benytter en bit mot bit sammenligning. Fordelen med en komparatorbasert løsning, er at den er relativ enkel å implementere, og lider ikke av aliasing. Ulempen med en komparator er at den krever store minneområder for å lagre resultatet. Hvis en konsentrator benyttes på utgangen av komparatoren, vil analysen gi svært lite informasjon om responsen, og den vil heller ikke kunne se forskjell på én eller flere feil per konsentrator.

Som nevnt tidligere vil både KUT og GOLD kretsen implementeres på samme FPGA brikke, sammen med en testmønstergenerator og utgangsresponsanalyse, henhold til *figur 17*. Vanligvis benyttes signaturanalyse i samarbeid med testmønstergeneratorene LFSR og CA.[1] En slik løsning hadde også vært et naturlig valg hvis dette skulle implementeres på en IC(Integrated Circuit). Ettersom utgangsresponsen fra GOLD kretsen og KUT kun skal sjekkes for ekvivalens, kan en konsentrator benyttes i tillegg. Dette tillater en bit for bit sammenligning uten bruk av ekstra minneelementer. I denne oppgaven vil kun en «bekreftelse» eller «avbekreftelse» være tilstrekkelig informasjon fra utgangsresponsanalysen.

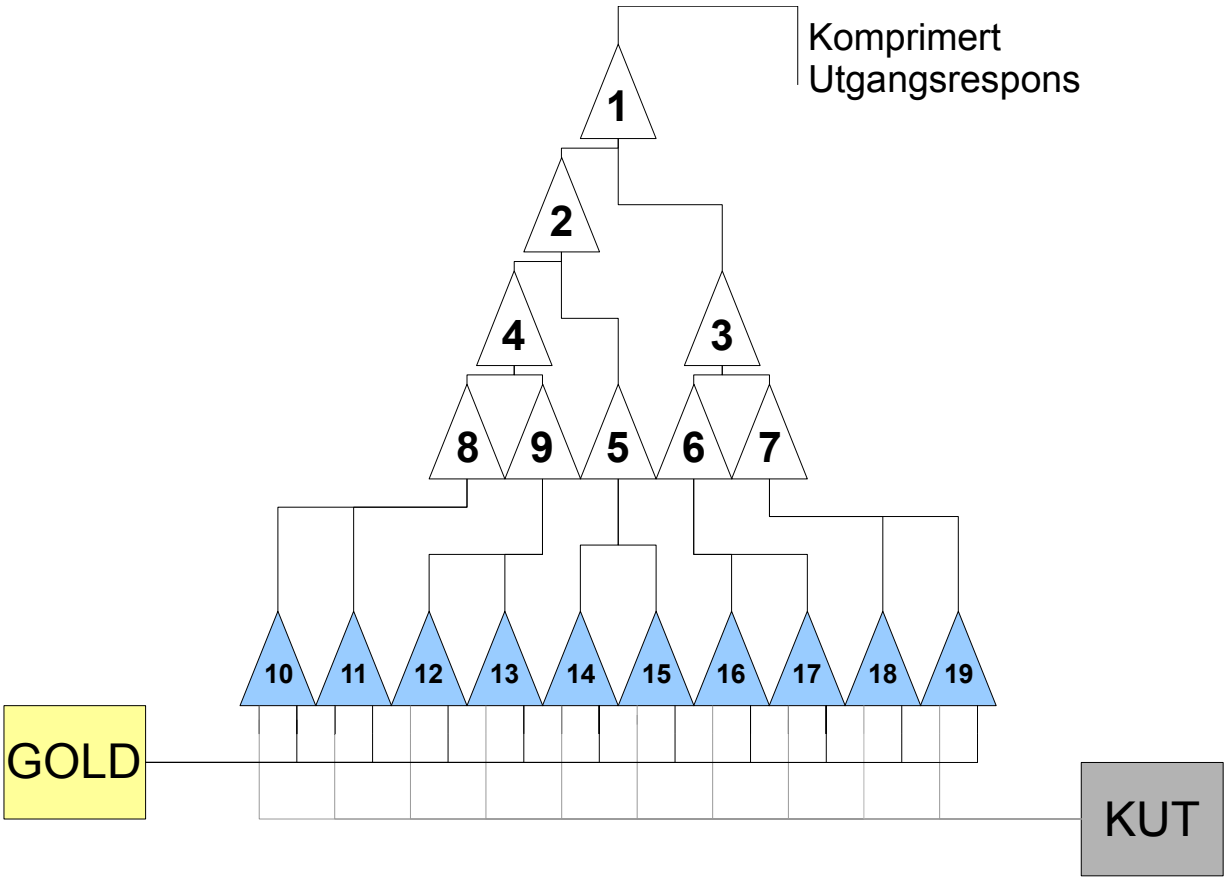


Figur 20: (a) Figuren viser forholdet mellom signalene i treet for hver OR-port. Som (b) viser, vil dette forholdet forplante seg i et tre av alle forskjellige størrelser. (c) Dette forholdet vil også lage et tre med minimum antall porter, slik at antall porter vil alltid være gitt på forhånd for et n-bit tre. (d) viser hvordan et tre med 9 porter kan genereres med forholdet fra (a) .

En optimal konsentrator løsning, benytter minimum antall 2-til-1 porter, og for et n-bit bredt signal tilsvarer det `n-1` porter. Dette kan enkelt bestemmes, ettersom signalet skal komprimeres fra en n-bit bred buss til et 1-bit bredt signal. For å kunne realisere en generell konsentrator løsning for busser av alle størrelser `n`, genereres `n-1` porter(*figur 20 (d)*). For å rute portene sammen, nummereres portene fra venstre til høyre, *figur 20 (b), i* et uniformt tre, kalt her for «universaltre» for enkelhetens skyld. Et «universaltre» inneholder `2m-1` porter(for `m = 2, 4, 8, 16...`), og har en inngangsbuss av størrelse `m`. Grunnen til at et «universaltre» benyttes, er at forholdet mellom portene i dette treet er låst. Se *figur 20 (a),* hvor *i* representerer nummeret hver port blir gitt i 20 (b), og er navnet på utgangssignalet for hver port. Når n-1 porter genereres, navngis hvert signal med et nummer, som knytter portene sammen. Et resultat av dette kan sees i *figur 20 (c)*. Ved å strekke treet slik som vist i *figur 20 (c)*, kan en konsentrator for busser av alle størrelser genereres, uten å ta hensyn til om $\log_2(n)$ (hvor `n` er bredden bussen) er et heltall eller ikke. Bussen kobles til signalene nummerert fra 2^n-1 til `n.`

Ved hjelp av modellen i *20 (a)*, hvor `i = 0,1,2....n-1`, vil signalene tilknyttet portene navngis som i *20 (d)*, og danner et optimalt tre for alle mulige kombinasjoner av bussbredder. *Figur 19 (a)* er kun gyldig når utgangsignalet fra portene nummereres fra venstre til høyre tilsvarende *(b)*. Hvis de nummereres fra høyre til venstre må 2i og 2i +1 bytte plass.

Figur 21 viser den komplette løsningen av komparatoren, bestående av `n` antall XOR-porter, hvor utgangsverdiene ledes til et n-bit konsentrator tre. Denne løsningen vil være universal for alle størrelser av n > 1.



Figur 21: Et komparatortre på 10 bit basert på fremgangsmåten i denne oppgaven. De hvit trekantene nummerert 1-9 representerer 2-til-1 OR-porter, mens de blå 10-19 representerer 2-til-1 XOR-porter.

4.2 Måling av feildekningsgrad

Måling av feildekningsgrad av et sett av testvektorer på en FPGA, kan gjennomføres på flere måter. En enkel metode(valgt i denne oppgaven) er å simulere mot injiserte feilmodeller, og la antall detekterte feil utgjøre dekningsgraden. Utfordringen med denne metoden er å introdusere feilmodeller i kretsen.

På en FPGA er det i hovedsak to generelle fremgangsmåter for dette. Den ene løsningen er statisk konfigurasjon, som introduserer feilmodeller i kretsen før den implementeres på FPGA brikken. Den andre løsningen, dynamisk konfigurasjon, introduserer feilmodeller i kretsen etter at KUT er implementert på en FPGA.

4.2.1 Dynamisk konfigurasjon

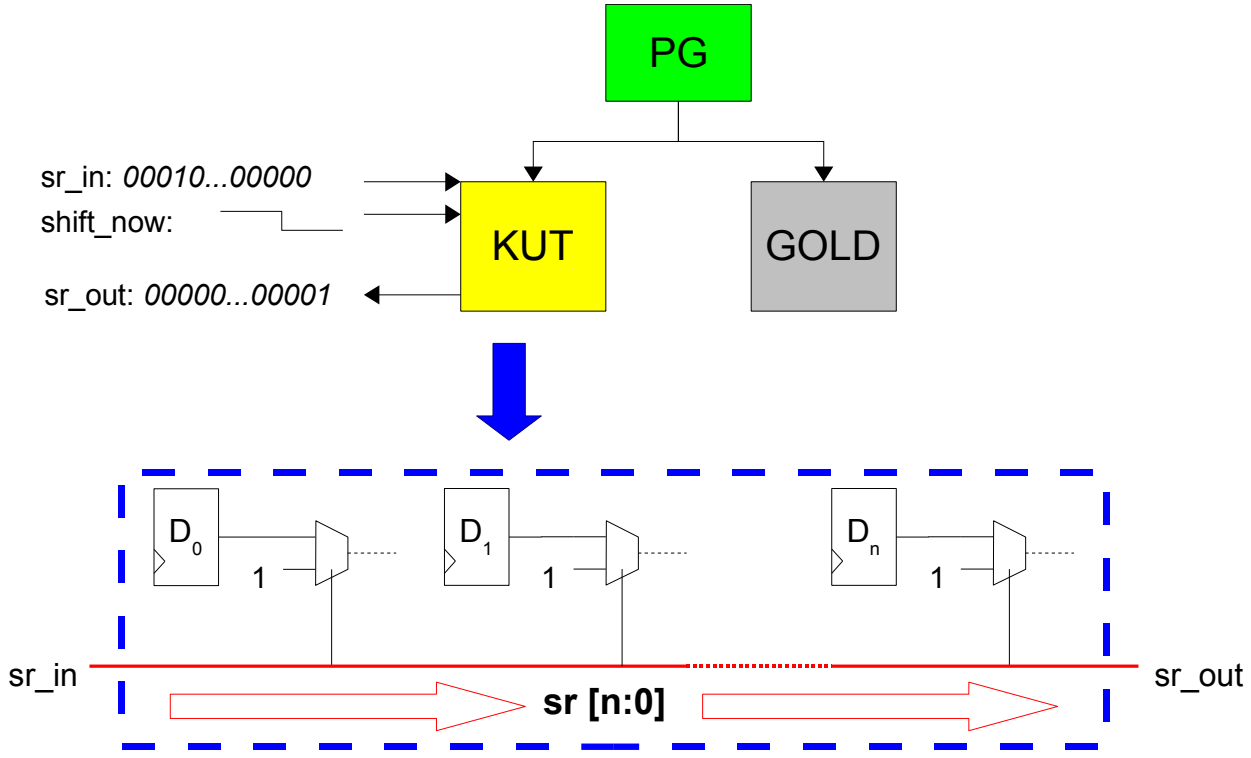
I [6] prøver de å finne en kosteffektiv metode for å evaluere testeffektiviteten av BIST på en FPGA, og benytter partiell rekonfigurasjon for å injisere feilmodeller på LUT nivå. For å kunne kontrollere de forskjellige linjene i kretsen ble et javabasert brukergrensesnitt JBits benyttet. For å evaluere testeffektiviteten simulerte de mot både single og grupper av låst-til feil.

Fordelen med partiell konfigurasjon, er at feil kan injiseres i en FPGA uten at hele kretsen må

konfigureres. Dette forhindrer unødvendig rekonfigurasjon av kretsen mellom hver injiserte feil. I tillegg vil denne metoden unngå at ekstra kretslogikk introduseres, som øker arealet og endrer tidskarakteristikken av kretsen. Tilfeldig endring av signaler internt i en FPGA kan være problematisk og utfordrende, og spesielt med det tilgjengelige verktøyet fra Xilinx. Det javabaserte verktøyet JBits er et egnet verktøy for slike operasjoner, noe [6] kan vise til. JBits opererer på bitstrømmen fra enten Xilinx verktøyet, eller fra selve FPGA modulen. Gjennom dette *software* grensesnittet kan alle konfigurerbare kilder som vipper, LUTs og linjer individuelt konfigureres.[4] I min oppgave hadde jeg planer om å benytte dette verktøyet, ettersom en optimale løsning for feildekningsgrad testing, bør unngå ekstra kretselement i KUT. Denne planen ble avviklet for en enklere løsning introdusert i *4.2.3,* på grunn av en evaluering fra [4]

> *An equally important limitation is that JBits requires that the user be very familiar with the architecture. While the Xilinx device architectures are actually completely documented in the Xilinx databook, most users have never had the need to learn such details. It is expected that the necessary understanding of the underlying device architecture will be the greatest barrier to the widespread acceptance of JBits, or any tool resembling JBits.*

Konklusjonen fra simuleringen i [6] er interessant, ettersom de benytter samme feilmodeller som i denne oppgaven. Etter simulering med flere forskjellige feilmodeller, viste resultatet at LSA(Line Stuck-At) gir et for optimistisk bilde av testeffektiviteten.



Figur 22: Låst-til-1 feilmodell legges på utgangen av vippene i KUT. Disse feilene aktiveres ved hjelp av et skiftregister.

4.2.2 Statisk konfigurasjon

En annen fremgangsmåte for å implementere feil i en KUT, er feilinjisering basert på ekstra kretslogikk introdusert i designet. I denne fremgangsmåten konfigureres KUT slik at alle feilmodellene er implementert på forhånd, før implementasjon. Feilene aktiveres etter tur, eller i grupper, av et inngangssignal under simuleringen. Hensikten med denne fremgangsmåten er å unngå ekstra tid for rekonfigurasjon av FPGA for hver simulerte feil. Negative konsekvenser med denne løsningen er økt arealforbruk, noe som i tillegg kan introdusere ekstra forsinkelser på linjene i kretsen.

En utfordring med denne metoden er å automatisere distribusjonen av feilmodeller i kretsen, ettersom manuell konfigurering er svært tidkrevende når antall feil overskrider 1000. På en tilfeldig krets kan dette bli svært utfordrende, spesielt hvis den er representert i en høynivå designbeskrivelse. En løsning er å editere kretsen etter at den er syntisert i FPGA verktøyet, ettersom den da er realisert i form av blokker, CLB(Configurable Logic Block). I [5] introduserer forfatterne FPGA emulasjon for måling av dekningsgrad. De benytter en løsning hvor de implementerer ekstra kretselementer med innebygd feil i hver CLB, som kan aktiveres med et kontrollsignal. For å kunne gjøre dette konfigureres alle brukte CLB, på FPGA kretsen, til å ha en ledig inngangspinne. Denne inngangspinnen kobles til et skiftregister, som aktiverer feil i hver CLB etter tur.

4.2.3 Fremgangsmåte for feilinjisering

Hvis kretsene som simuleres har lett tilgjengelige kontrollpunkt, som for eksempel i en VHDL «generate statement», kan feilmodeller distribueres inn i designet relativt enkelt. I [3] benytter de multipleksere til å injisere låst-til feilmodeller, tilsvarende ad-hoc.

Her i denne oppgaven benyttes ISCAS'89 benchmark kretser, hvor feilmodeller enkelt kan implementeres ved hjelp av kreativ bruk av «replace» funksjonen. Denne metoden fungerer svært godt med designrepresentasjonen i ISCAS'89 kretsene, og over 1000 feilmodeller kan legges til i løpet av få sekunder. *Figur 22* viser hvordan KUT utvides med to ekstra inngangssignal og ett utgangssignal, i tillegg til ekstra mulipleksere på utgangen av alle vippene i kretsen. Det ene inngangssignalet(sr_in) og utgangssignalet(sr_out) representerer starten og slutten på et skiftregister, som rutes til alle de implementerte multiplekserene. Det andre inngangssignalet(shift_now) bestemmer når registeret skal skifte, slik at en logisk ener kan aktivere feilmodellene etter tur. Resten av skiftregisteret fylles med logiske nullere som sørger for at ikke flere feilmodeller er aktive samtidig. Multiplekserene utgjør ingen logisk forskjell i kretsen når

feilmodellene ikke aktiveres, men vil introdusere større tidsforsinkelse på signallinjene.

En tilstandsmaskin benyttes for å kontrollere testsekvensen, vist i *figur 23*. Tilsvarende metoden i [3], benyttes også her en teller for å holde kontroll på antall testvektorer påtrykt kretsen. I tillegg benyttes det en teller for å bestemme antall feil detektert av testmønsteret. Disse tellerne er representert i henholdsvis simu_c og fault_c.

De andre signalene i tilstandsmaskinen er:

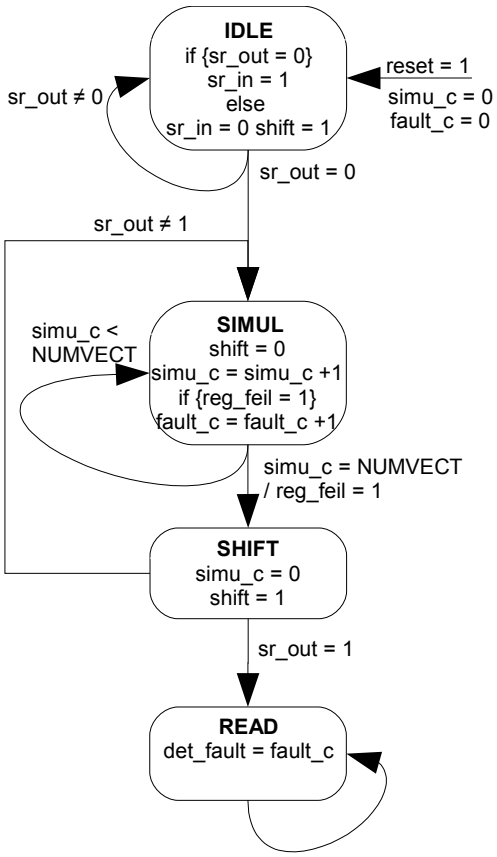
shift: Skifter skiftregisteret i KUT.

reset: Asynkron, og setter den i tilstanden IDLE.

det_fault: Antall detekterte feil, vises på LED.

reg_feil: Settes høy når feil detekteres.

NUMVECT: Er antall testvektorer per feil.



Figur 23: Tilstandsmaskin for beregning av feildekningsgrad.

For å starte en testsekvens må tilstandsmaskinen settes i tilstanden *IDLE*, hvor de to tellerne settes til null.

I *IDLE* fylles skiftregisteret med nullere, og til slutt en logisk ener som skal skiftes gjennom.

I *SIMUL* påtrykkes testvektorene KUT, og den eneste måten å komme ut av denne tilstanden er enten ved å detektere feil, eller ved å simulere «NUMVECT» antall testvektorer. Den siste utveien fungerer som en «time out».

En funksjon som ikke er representert i figur 22, er en implementert buffer i tilstanden *SIMUL*. Verdien av bufferen er en positiv integer, og samsvarer med den sekvensielle dybden av KUT, se *tabell 1*. Denne bufferen forhindrer at verdier forplantet i kretsen, som et resultat av den forrige feilmodellen, trigger komparatoren ved simulering av neste feilmodell. Bufferen realiseres ved at simu_c må ha en bestemt minimumsverdi for at detektert feil kan avslutte *SIMUL* tilstanden.

I *SHIFT* tilstanden skiftes skiftregisteret i KUT et hakk. Nestetilstanden er avhengig om alle feilmodellene er simulert eller ikke. Hvis eneren i skiftregisteret har skiftet gjennom alle vippene i KUT, vil det kunne avleses i sr_out = 1, og nestetilstanden vil være *READ*. Hvis sr_out = 0, vil neste tilstand være *SIMUL*, hvor nye testvektorer simuleres mot en ny feil.

I *READ* avleses antall feil detektert, og rutes til et LED-display på FPGA kortet.

Injiseringen av den logiske eneren i skiftregisteret kan hindres med en variabel, og da fungerer kretsen som en ekvivalentsjekk, og ikke en feildekningsgrad emulator.

4.3 Testkretser

For å kunne måle dekningsgraden av ekvivalentsjekkapplikasjonen, benyttes ISCAS'89 benchmark kretser. Disse kretsene har en designrepresentasjon på portnivå, skrevet i Verilog kode. For å kunne måle dekningsgraden, endres modulrepresentasjonen av vippene i KUT, slik at utgangen Q kan låses til logisk 0 eller 1. Den andre kretsen GOLD, forblir urørt.

```
module dff (CK,Q,D,setf);
input CK,D,setf;
output Q;
reg Q;
always@(posedge CK)
   if (setf)
      Q <= 0'b0;
   else
      Q <= D;
endmodule
```

```
reg [29:0] sr;
always@(posedge CK)
begin
   if (shift_now == 1'b1)
   begin
      sr[29:1] <= sr[28:0];
      sr[0] <= sr_in;
   end
end
assign sr_out = sr[29];
```

Dette skiftregisteret settes inn i KUT for å kunne distribuere feil til vippene. Akkurat dette registeret er beregnet for 28 vipper, og gjør et skift hver gang `shift_now` er høy. De to kodeeksemplene på denne siden, utgjør endringene på KUT slik at feilinjisering kan realiseres.

Å benytte både VHDL og Verilog design på FPGA kretsen er problemfritt. Det eneste problemet er at det er svært tidkrevende å knytte signalene sammen.

Tabell 1 viser egenskapene til benchmark kretsene, hvor antall vipper representerer hvor mange feilmodeller av typen sa-0 og sa-1 som kan injiseres i kretsen. Andre innganger på kretsene er «reset», som påtrykkes et vektet inngangsignal.(Vektet funksjon bestående av en 3-til-1 AND-port, se figur 10)

Krets	Sekv. dybde	Vipper	Primær inng.	Primær utg.	Feil
s298	8	14	3	6	308
s953	3	29	16	23	1079
s1423	10	74	17	5	1515
s5378	36	179	35	49	4603

Tabell 1: ISCAS'89: Sekvensiell dybde, antall vipper i kretsen, antall primære innganger, antall primære utganger og antall kollapset feil. Fra: http://courses.ece.uiuc.edu/ece543/iscas89.html.

Et problem ved simulering av to like Verilog kretser, er at de er beskrevet med samme signaler. Det betyr at alle signaler i KUT(eller GOLD) må endres i forhold til GOLD. Dette kan være tidkrevende, og bruk av «replace» funksjonen er et sjansespill, ettersom det er lett å endre for mye.

4.4 Implementasjon på Virtex-II pro FPGA

Informasjon om FPGA kortet benyttet i denne oppgaven, finnes i vedlegg D.

For å avlese antall detekterte feil, benyttes lysdiodene på kortet. Det er 8 lysdioder montert på kortet, og de kan representere opp til 255(2^8-1) detekterte feil. Det betyr at hver benchmark krets maksimalt kan avlese 255 sa-0 eller 255 sa-l feilmodeller. Hvis man ignorerer de laveste bit-verdiene, kan man øke denne mengden ved å kun koble de mest signifikante bit-verdiene til diodene. Dette vil dog introdusere noe usikkerhet i målingen.

For å starte sekvensen aktiveres en brukerknapp på kortet, som er koblet til det globale «reset» signalet i kretsen. Simuleringen er ferdig når en verdi er påtrykket lysdiodene, vanligvis etter få sekunder.

Constraint fil benyttet i denne simuleringen:

```
NET "CLK" TNM_NET = "CLK";
TIMESPEC "TS_CLK" = PERIOD "CLK" 10 ns HIGH 50 %;
NET "CLK"  LOC = "AH17"  ;
NET "out_led<7>"  LOC = "E31"  ;
NET "out_led<6>"  LOC = "E32"  ;
NET "out_led<5>"  LOC = "F31"  ;
NET "out_led<4>"  LOC = "F30"  ;
NET "out_led<3>"  LOC = "E1"  ;
NET "out_led<2>"  LOC = "E2"  ;
NET "out_led<1>"  LOC = "E3"  ;
NET "out_led<0>"  LOC = "E4"  ;
NET "reset_in"  LOC = "G25" | pullup ;
```

b7(MSB)	b6	b5	b4
b3	b2	b1	b0(LSB)

Ut i fra lysdiodenes plassering på kortet, representerer de følgende verdier i bitvektoren.

5 Resultat og analyse

100.000 testvektorer ble benyttet for hver feilmodell injisert i kretsene. Alle kretsene ble injisert sa-0 og sa-1 feilmodeller, simulert med både CA og LFSR testmønster.

CA

Krets	Injiserte feil	Detekterte sa-0	Detekterte sa-1	Feildekningsgrad
s298	2*13	13	13	100%
s953	2*29	29	29	100%
s1423	2*74	40	55	64.2%
s5378	2*179	135	136	75.7%

LFSR

Krets	Injiserte feil	Detekterte sa-0	Detekterte sa-1	Feildekningsgrad
s298	2*13	13	13	100%
s953	2*29	29	29	100%
s1423	2*74	23	34	38.5%
s5378	2*179	125	130	71.2%

S953 ville ikke la seg initialisere, og resultatet er ikke helt å stole på. Over 10^6 testvektorer ble påtrykket under simulering, uten at alle utgangssignalene ble initialisert. En mulig årsak er et skjult reset signal som påtrykkes 1 for ofte, og vektet funksjoner ble forsøkt, uten resultat.

For å regne usikkerheten av disse målingene, kan funksjon 2.23 benyttes:

$$3\sigma estimat = x \pm \frac{4.5}{N_s}\sqrt{1+0.44\mathrm{N}_s x(1-x)} \qquad (2.23)$$

I simuleringen av krets *s5378,* ble følgende verdier oppnådd; x = 0.757 og N_S = 358.
Dette gir $FC(s5378) = 0.757 \pm 0.069$ - N_S er muligens noe for lav for at tilnærmingen i 2.21 skal være gyldig.

ISCAS'89: *S38417: 28 inputs, 106 outputs,**1636** D-type flipflops.*
Ved å ignorere de 4 minst signifikante verdiene i bitvektoren, som representerer antall detekterte feil, kan opp til 2048 detekterte feil representeres vha. lysdiodene. Dette gir mulighet for simulering av s38417, og ved bruk av LFSR og sa-0, ble 104(+-8)/1636 feil detektert. En mulig årsak til det lave antallet, er at flere utgangssignal ikke tok noen logisk verdi i denne kretsen.

Det var flere ISCAS'89 kretser som ikke lot seg initialisere, til tross for at over 10^6 testvektorer ble påtrykket. De fleste er ikke representert i denne oppgaven, bortsett fra s953. Mer tilgjengelig informasjon om benchmark kretsene, slik som f.eks « reset» signal, kan muligens løse dette problemet.

6 Konklusjon

Applikasjonen demonstrerer at den kan måle feildekningsgrad, til tross for enkelte mangler. Blant annet vil den ikke kunne håndtere en generell krets, ettersom feilinjisering ble utført ved hjelp av endring i en designbeskrivelse. Dette er dog en ganske effektiv metode for ASCI'89 kretsene, som har en oversiktlig designbeskrivelse i form av porter. For andre designbeskrivelser, vil en slik metode mest sannsynlig være uegnet. Andre metoder for feilinjisering har blitt diskutert, og en mulig løsning kan være bruk av JBITs, enda dette er utenfor mitt kunnskapsområde.

To mønstergeneratorer har blitt designet for denne applikasjonen, en generell CA og en generell LFSR. Begge benytter primitive polynomer. I tillegg har en komparator blitt designet, som er svært oversiktlig, og fungerer for tilnærmet alle signaler av forskjellige størrelser.

Resultatet fra ekvivalentsjekkapplikasjonen ble tilkoblet lysdioder på FPGA kretsen. Dette begrenser antall feilmodeller til 255, så lenge man ikke tillater usikkerhet. Ved å introdusere en usikkerhet på +- 8, kan opp mot 2048 feilmodeller injiseres.

Feilprøve betyr å kun simulere mot et begrenset antall feilmodeller. Man kan da oppnå et estimat på feildekningsgraden, uavhengig av størrelsen på kretsen, så lenge feilprøven er stor nok. Ved å kun benytte 358 feilmodeller, er usikkerheten nesten 10%, men hvis dette antallet øker til 1000 vil man kun ha en usikkerhet på 2-3% .

Resultatene i 5, viser at CA oppnår bedre feildekningsgrad enn LFSR, ved bruk av 100.000 testvektorer for hver injiserte feil.

Referanseliste

[1] N. Jha og S. Gupta. *Testing of Digital Systems*. Cambridge: Cambridge University Press 2003, p. 222-45.

[2] C. Stroud. *A Designer's Guide to Built-In Self-Test*. Dodrecht: Kluwer Academic Publishers 2002.

[3] P. Ellervee, J. Raik, V. Tihhomirov og R. Ubar. *FPGA based fault emulation of synchronous sequential circuits* . Norchip Conference, 2004. Proceedings, Nov. 2004, p. 59 – 62.

[4] S.A.Guccione and D.Levi. *XBI: A Java-based interface to FPGA hardware.* Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering, Proc. SPIE Photonics East, J.Schewel(Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA, Nov 1998.

[5] Kwang-Ting Cheng, Shi-Yu Huang and Wei-Jin Dai. *Fault Emulation: A New Methodology for Fault Grading.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems Vol.18, Oct 1999, p 1487-95.

[6]A. Parreira, J.P. Teixeira and M.B. Santos. *Lecture Notes in Computer Science*: *FPGAs BIST Evaluation.* Berlin: Springer-Verlag 2004, p. 333-43.

[7] P.H. Hortensius, et al. *Cellular Automata circuits for built-in self test.* IBM J. Research and Development, vol. 34, no. 2/3, Mar./May 1990, p. 381-405.

[8] S. Nandi and P. Pal Chaudhuri. *Analysis of Periodic and Intermediate 90/150 Cellular Automata Boundary.* IEEE Transactions on Computers, vol. 45, no. 1, Jan. 1996.

[9] M.L. Bushnell and V.D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* Boston: Kluwer Academic Publishers 2002.

[10] R.E. Walpole, R.H. Myers, S.L. Myers and Keying Ye. *Probability & Statistics for Engineers & Scientists*. New Jersey: Prentice-Hall 2002.

[11] *Virtex-II Pro™ FF1152 Development Board User's Guide*. April 2005, tilgjengelig fra http://polzope.in2p3.fr:8081/PHENIX/projet-spiro/ressources/2vpxff1152_user_guide_1_5.pdf

[12] R. P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton, New Jersey: Princeton University Press 1994.

[13] Parimal P. Chaudhuri et al. *Additive Cellular Automata: Theory and Applications, Volume 1.* Los Alamitos: Wiley-IEEE Computer Society Press 1997.

Vedlegg A: Primitive polynom for LFSR og CA.

Length n	*Construction*	*Cycle length*
4	0101	15
5	11001	31
6	010101	63
7	1101010	127
8	11010101	255
9	110010101	511
10	0101010101	1.023
11	11010101010	2.047
12	010101010101	4.095
13	1100101010100	8.191
14	01111101111110	16.383
15	100100010100001	32.767
16	1101010101010101	65.535
17	01111101111110011	131.071
18	010101010101010101	262.143
19	0110100110110001001	524.287
20	11110011101101111111	1.048.575
21	011110011000001111011	2.097.151
22	010101010101010101010101	4.194.303
23	11010111001110100011010	8.388.607
24	111111010010110101010110	16.777.215
25	101111010101010011110010 0	33.554.431
26	010110101101000101110110 00	67.108.863
27	000011111000001100100001101	134.217.727
28	010101010101010101010101010101	268.435.455
29	101010010101110010100010000 11	2^n - 1
30	111010001001101100101000111101	2^n - 1
31	010011001010110111110111001 1000	2^n - 1
32	010001100000100110111011111 010101	2^n - 1
33	000011000100111001110010110 000101	2^n - 1
34	001111000010110100001100011 0111010	2^n - 1
35	010101111011110110011101010 01010011	2^n - 1
36	101001100100100011111010110 000100011	2^n - 1
37	001001011001111010110101100 00010110011	2^n - 1
38	000111001010111101100110011 11100001 0011	2^n - 1
39	110100010111110110111100110 011101101100	2^n - 1
40	000011101100101010111110010 000010111001 01	2^n - 1
41	011010111111101000010110011 000111100001 11	2^n - 1
42	001001111110110011100101001 001100111100 110	2^n - 1
43	001110101110001011100010000 101101011001 0010	2^n - 1
44	001111001111011101011011100 000100101011 000010	2^n - 1
45	001101001011001101101001000 100110001101 001101	2^n - 1
46	000100101001100101000110100 010110011101 1010110	2^n - 1
47	001110010111111001110010101 001000101110 000001101	2^n - 1
48	000110000110111110010010100 111010001111 000001111	2^n - 1
49	001011011111011001000110010 111110001011 10110011001	2^n - 1
50	100110100110110000001100011 010001011001 00010010110	2^n - 1
51	000100001011101010100000101 101001110100 0010100010111	2^n - 1
52	001100100011011110111011111 110001000111 1010111000110	2^n - 1
53	100001110010100010000010010 011001011101 1111011001010 1	2^n - 1

Tabell 2: Fra [7], og er et utvalg av LHCA av forskjellige størrelser med regel 90/150 som gir MLS. Boolske verdien 1 representerer regel 150, og 0 representerer regel 90.

Degree n	*Polynomial*
2,3,4,6,7,15,22, 60, 63	$x^n + x + 1$
5,11,21,29, 35	$x^n + x^2 + 1$
8, 19, 38, 43	$x^n + x^6 + x^5 + x + 1$
9, 39	$x^n + x^4 + 1$
10, 17, 20, 25, 28, 31, 41, 52	$x^n + x^3 + 1$
12	$x^n + x^6 + x^4 + x^3 + 1$
13, 24, 45, 64	$x^n + x^4 + x^3 + x + 1$
14, 16	$x^n + x^5 + x^4 + x^3 + 1$
18, 57	$x^n + x^7 + 1$
23, 47	$x^n + x^5 + 1$
26, 27	$x^n + x^{12} + x^{11} + x + 1$
30, 51, 53, 61	$x^n + x^{16} + x^{15} + x + 1$
32, 48	$x^n + x^{28} + x^{27} + x + 1$
33	$x^n + x^{13} + 1$
34	$x^n + x^{15} + x^{14} + x + 1$
36	$x^n + x^{11} + 1$
37	$x^n + x^{12} + x^{10} + x^2 + 1$
40	$x^n + x^{21} + x^{19} + x^2 + 1$
42	$x^n + x^{23} + x^{22} + x + 1$
44, 50	$x^n + x^{27} + x^{26} + x + 1$
46	$x^n + x^{21} + x^{20} + x + 1$
49	$x^n + x^9 + 1$
54	$x^n + x^{37} + x^{36} + x + 1$
55	$x^n + x^{24} + 1$
56, 59	$x^n + x^{22} + x^{21} + x + 1$
58	$x^n + x^{19} + 1$

Tabell 3: Et utvalg av primitve LFSR polynomer hentet fra [2], opp til 64 grader.

Vedlegg B: VHDL kode

Modul_CA

```
--#############################################################
--GENERIC MLS No-Boundary Cellular Automata PRTPG.
--It can be programmed to represent any 4-32++ bits
--No-Boundary Hybrid CA with rule 90 and 150.
--rule150 and CA_SIZE are generic values.
-------------------------------------------------------------
--Stig S. Opstad - 26. Sept 2007
--#############################################################
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

---------------------- Entity declaration ----------------------
entity CA_GENERIC is
   GENERIC (
   CA_SIZE : INTEGER := 6); --size of the vectors
   port(
   preset : in std_logic; --"preset" sets the register to "111....11"
   clk    : in std_logic;
   CA_out : out std_logic_vector(CA_SIZE-1 downto 0)
   ); --"CA_out" is the generated testvectors
end entity CA_GENERIC;

-------------------- Architecture declaration -------------------
architecture beh of CA_GENERIC is

----signal representation--
--Din is intern signal at the input of the FF, Dout is at the output.
--"rule150" represents the combination of rules in the register.
signal Dout, Din : std_logic_vector(CA_SIZE-1 downto 0);
constant rule150 : std_logic_vector(CA_SIZE-1 downto 0) := "010101";

----component declaration--
component DVIPPE
    port(
        preset : in std_logic;
        D_in   : in std_logic;
        clk    : in std_logic;
        Q_out  : out std_logic
        );
end component;

--------------------------- CA_SET ----------------------------
begin

    CA_Set: process(Dout)
    begin
        for j in 0 to CA_SIZE-1 loop
        CA_out(j) <= Dout(j);
        end loop;
    end process CA_Set;

------------------------ Signal routing -----------------------
--The No Boundary solution results in only two neighbour connections
--at the input of flipflop 0 and 31.
    Din(0) <=  Dout(1) XOR (Dout(0) AND rule150(0));
    Din(CA_SIZE-1) <= Dout(CA_SIZE-2) XOR (Dout(CA_SIZE-1) AND rule150(CA_SIZE-1));

----Generate g0--
    G0: for i in 1 to CA_SIZE-2 generate
       Din(i) <= Dout(i-1) XOR (Dout(i) AND rule150(i)) XOR Dout(i+1);
    end generate G0;

----Generate g1--
             G1: for k in 0 to CA_SIZE-1 generate
         Dvip: DVIPPE port map(preset, Din(k), clk, Dout(k));
             end generate G1;

end architecture beh;
```

Modul LFSR

```
--#################################################################
--Generic internal MLS LFSR 4-32 bit.
-----------------------------------------------------------------
--Stig S. Opstad - 26. Sept 2007
--#################################################################
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

----------------------- Entity declaration ----------------------
entity LFSR_GENERIC is
   GENERIC ( LFSR_SIZE : INTEGER := 6); --Size of the vectors
   port(
   preset : in std_logic;
   clk    : in std_logic;
   LFSR_out : out std_logic_vector(LFSR_SIZE-1 downto 0)
   );
end entity LFSR_GENERIC;

-------------------- Architecture declaration ------------------
architecture beh of LFSR_GENERIC is
signal Din, Dout : std_logic_vector(LFSR_SIZE-1 downto 0);
constant poly : std_logic_vector(LFSR_SIZE downto 0) := "1000011";

----component declaration--

component DVIPPE
    port(
        preset : in std_logic;
        D_in   : in std_logic;
        clk    : in std_logic;
        Q_out  : out std_logic
        );
end component;

--------------------------- LFSR_SET ---------------------------
begin
    LFSR_Set: process(clk)
    begin
        for i in 0 to LFSR_SIZE-1 loop
            LFSR_out(i) <= Dout(i);
        end loop;
    end process LFSR_Set;

Din(0) <= Dout(LFSR_SIZE-1);

----Generate g1--
G1: for i in 1 to LFSR_SIZE-1 generate
       Din(i) <= Dout(i-1) XOR (Dout(LFSR_SIZE-1) AND poly(i));
    end generate G1;

----Generate g2--
G2: for k in 0 to LFSR_SIZE-1 generate
    D: DVIPPE port map(preset, Din(k), clk, Dout(k));
end generate G2;

end architecture beh;
```

Modul ORA

```
--#################################################################
--GENERIC sized Comparator.
-----------------------------------------------------------------
--Stig S. Opstad 26. sept 2007
--#################################################################
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

---------------------- Entity declaration ----------------------
--GOLDEN and KUT represents the output signal from the two compared circuits.
--reg_feil has the value '1' when it's a missmatch between them.
--
--           ----    ---
--          |GOLD|  |KUT|
--           ----    ---
--             |__  __|
--                ||
--              -----
--              |ORA| => reg_feil
--              -----
--
entity ORA_GENERIC is
   GENERIC (ORA_SIZE : INTEGER := 6); --size of the comparator
   port(
   GOLDEN, KUT : in std_logic_vector(ORA_SIZE-1 downto 0);
   clk : in std_logic;
   reg_feil : out std_logic := '0'
   );
end entity ORA_GENERIC;

-------------------- Architecture declaration ------------------
--This circuit are using XOR gates between the "GOLDEN" and "KUT"
--signal, and XOR_OUT represents the output signal. To compress the
--x sized signal, 2x-1 OR gates are needed. OR_TEMP is the internal
--signals between them.

architecture rtl of ORA_GENERIC is

----signal representation--
signal XOR_OUT: std_logic_vector(ORA_SIZE-1 downto 0);
signal OR_TEMP: std_logic_vector((2*ORA_SIZE)-1 downto 1);

---------------------------- ORA_SET ----------------------------
begin
   ORA_SET: process(clk, GOLDEN, KUT)
   begin
      if clk'event and clk = '1' then
      reg_feil <= OR_TEMP(1);
      end if;
   end process ORA_SET;

---------------------------- XOR GATES --------------------------

   g1: for i in 0 to ORA_SIZE-1 generate
      XOR_OUT(i) <= GOLDEN(i) XOR KUT(i);
   end generate g1;

---------------------------- OR GATES ---------------------------

   g2: for j in 1 to (2*ORA_SIZE)-1 generate

      ----Generate g21--
      g21: if j < ORA_SIZE generate
      OR_TEMP(j) <= OR_TEMP(2*j) OR OR_TEMP((2*j)+1);
      end generate g21;

      ----Generate g22--
      g22: if j >= ORA_SIZE generate
      OR_TEMP(j) <= XOR_OUT(j-ORA_SIZE);
      end generate g22;

   end generate g2;

end architecture rtl;
```

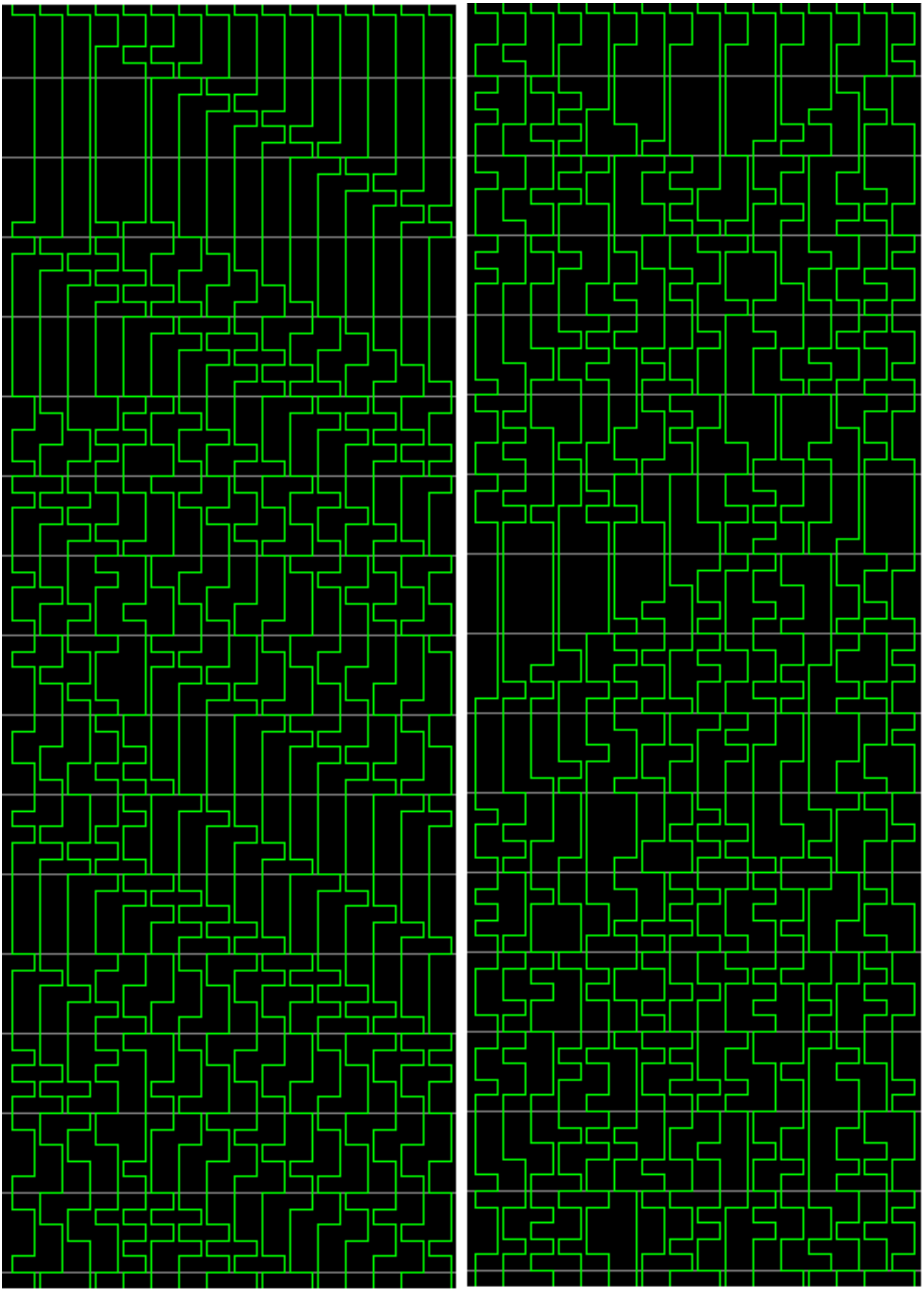
Modul TOPP

Dette eksempelet benytter ISCAS'89 kretsen s298 som GOLD og KUT.

```
--#############################################################
--Equivalence check for two circuits, GOLDEN and KUT.
--For a evaluation of the fault coverage, set FC = 1;
-------------------------------------------------------------
--Stig S. Opstad - 06. October 2007
--#############################################################
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

---------------------- Entity declaration ----------------------
entity TOPP is
   GENERIC (
   PG_SIZE : INTEGER := 6; --size of patter generator
   ORA_SIZE: INTEGER := 6; --size of output response analysis
   FBUFF: INTEGER := 10; -- size of fault buffer
   NUMVECT: INTEGER := 100; --number of vectors for each fault
   NUMFAUL: INTEGER := 2**4 -- max number of faults injected
   );
   port(
   reset : in std_logic;
   CLK    : in std_logic;
   out_led : out std_logic_vector(7 downto 0)  --Number of detected faults
   );
end entity TOPP;

--------------------------------------
--PG=Pattern Generator, LFSR or CA.
--ORA=Output Response Analysis
--golden=unfaulty circuit
--KUT=possible faulty circuit
--
--               ----
--               |PG|
--               ----
--      pg_temp- |  | -pg_temp
--              |    |
--           ----    ---
--          |GOLD|  |KUT|
--           ----    ---
--golden_temp- |__  __| -KUT_temp
--                ||
--              -----
--              |ORA|
--              -----

-------------------- Architecture declaration -------------------
architecture top of TOPP is

----TOPP signal representation--
signal KUT_temp: std_logic_vector(ORA_SIZE-1 downto 0);
signal golden_temp: std_logic_vector(ORA_SIZE-1 downto 0);
signal pg_temp : std_logic_vector(PG_SIZE-1 downto 0);
type STATE_TYPE is (IDLE, SIMUL, SHIFT, READ);
signal state : STATE_TYPE;
signal shift_now, reg_feil, sr_in, sr_out : std_logic;

----KUT/GOLDEN signal representations--
signal GND,VDD,CK,G0,G1,G2 : std_logic;
signal G117,G132,G66,G118,G133,G67 : std_logic;
signal H0,H1,H2,H117,H132,H66,H118,H133,H67 : std_logic;

----component declarations--
component s298
    port(
        GND,VDD,CK,G0,G1,G2,shift_now,sr_in : in std_logic;
        G117,G132,G66,G118,G133,G67,sr_out : out std_logic
        );
end component;

component gs298
    port(
        GND,VDD,CK,H0,H1,H2 : in std_logic;
        H117,H132,H66,H118,H133,H67 : out std_logic
```

```
        );
end component;

----signal routing--
begin
GND <= '0';
VDD <= '1';
CK <= CLK;
G0 <= pg_temp(3) AND pg_temp(4) AND pg_temp(0);
G1 <= pg_temp(1);
G2 <= pg_temp(2);
KUT_temp(0) <= G117;
KUT_temp(1) <= G132;
KUT_temp(2) <= G66;
KUT_temp(3) <= G118;
KUT_temp(4) <= G133;
KUT_temp(5) <= G67;

H0 <= pg_temp(3) AND pg_temp(4) AND pg_temp(0);
H1 <= pg_temp(1);
H2 <= pg_temp(2);
golden_temp(0) <= H117;
golden_temp(1) <= H132;
golden_temp(2) <= H66;
golden_temp(3) <= H118;
golden_temp(4) <= H133;
golden_temp(5) <= H67;

---------------------------- faultinj ----------------------------
--Asynchron reset, that force the fsm into the state "IDLE".
--"simu_c" counts the number of vectors from the PG. They follow the same clock.
--"fault_c" counts the number of detected faults.
faultinj: process (clk, reset) is
   variable simu_c : integer range 0 to NUMVECT;
   variable fault_c : integer range 0 to NUMFAUL;
   variable FC: integer range 0 to 1 := 1; --eq check when 0, fault cov when 1.
   begin
   if (reset = '1') then --Resets the counters.
      simu_c := 0;
      fault_c := 0;
      state <= IDLE ;
   elsif (clk'event and (clk = '1')) then
      case state is

         when IDLE =>
            if sr_out = '0' then
                state <= SIMUL;
                if FC = 1 then
                sr_in <= '1';
                else
                sr_in <= '0';
                end if;
            else
                state <= IDLE;
                shift_now <= '1';
                sr_in <= '0';
            end if;

         when SIMUL => --Controls the number of testvectors
            shift_now <= '0';
            sr_in <= '0';
            if ((reg_feil = '1') and (simu_c > FBUFF)) then --Missmatch GOLDEN and KUT + buffer.
               fault_c := fault_c +1;
               state <= SHIFT;
            elsif simu_c = NUMVECT then
               state <= SHIFT;
            else
               simu_c := simu_c +1;
               state <= SIMUL;
            end if;

         when SHIFT => --Shifts the shift register
            simu_c := 0;
            if sr_out = '1' or FC = 0 then --'1' has shifted through the register
                state <= READ; --All faults are simulated
            else
                state <= SIMUL;
                shift_now <= '1';
            end if;
```

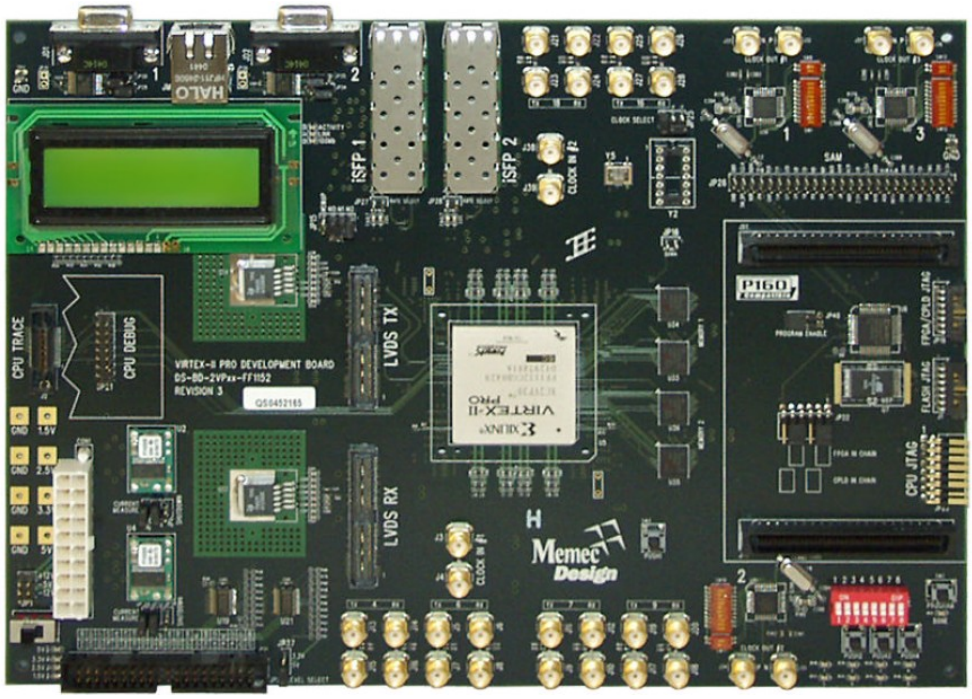
```
            when READ => --Prints the number of detected faults to the output.
                 out_led <= conv_std_logic_vector(fault_c, 8);
                 NS <= STAY;
            when STAY =>

        end case;
  end if;
end process faultinj;

------------------------ KUT and GOLDEN ------------------------

KUT : s298
port map(
GND => GND,
VDD => VDD,
CK => CK,
G0 => G0,
G1 => G1,
G2 => G2,
G117 => G117,
G132 => G132,
G66 => G66,
G118 => G118,
G133 => G133,
G67 => G67,
----SHIFT signals--
shift_now => shift_now,
sr_in => sr_in,
sr_out => sr_out
   );
-----------------------------------------------------------------
GOLDEN : gs298
port map(
GND => GND,
VDD => VDD,
CK => CK,
H0 => H0,
H1 => H1,
H2 => H2,
H117 => H117,
H132 => H132,
H66 => H66,
H118 => H118,
H133 => H133,
H67 => H67
   );

------------------------ Pattern Generators ----------------------
--Either LFSR or CA:

--D1 : entity work.LFSR_GENERIC(beh)
--port map(
--    preset => reset,
--    clk => Clk,
--   lfsr_out => pg_temp
--   );

D1 : entity work.CA_GENERIC(beh)
port map(
    preset => reset,
    clk => CLK,
    CA_out => pg_temp
    );

-------------------- Output Response Analysis -------------------

D2 : entity work.ORA_GENERIC(rtl)
port map(
    GOLDEN => golden_temp,
    KUT => KUT_temp,
    clk => CLK,
    reg_feil => reg_feil
    );

end architecture top;
```



Figur 24: Fra høyre, henholdsvis 16-bits LFSR og CA, generert fra modul CA og LFSR i vedlegg B.

Vedlegg C : Informasjon om FPGA kortet.

Dokumentasjon hentet fra [11]:



Figur 25: Virtex-II Pro FF1152 System Board, bilde fra [11].

- Xilinx XC2VP20/P30/P40/P50-FF1152C FPGA
- Support for 10G Optical Module
- High-speed LVDS Interface Supporting SPI-4.2
- Six Rocket I/O™ Ports Supporting up to 3.125Gbits/port
- Two Optical GbE Ports
- Support for GbE Optical Module
- Three Programmable LVDS Clock Sources (25 – 700MHz)
- On-board LVTTL 100MHz Oscillators
- On-board LVTTL Oscillator Socket (4/8 -Pin Oscillators)
- Two User LVDS Clock Inputs via Differential SMA Connectors
- Three User LVDS Clock Outputs via Differential SMA Connectors
- Two SDRAM Memory Blocks (32MB each, x32 memory configuration)
- SDRAM Memory Blocks can be used as a single x64 Memory
- P160 Connectors
- 10/100 PHY
- A 40-Pin User Header to be used as General-Purpose I/O or IDE interface
- LCD Panel
- 16/32Mb Atmel Data Flash for FPAG configuration
- JTAG Programming/Configuration Port
- CPU JTAG/Debug Ports
- CPU TRACE Port
- SystemACE™ Module Connector

- Two RS232 Port
- User LEDs
- User DIP and Push-Button Switches

The Virtex-II Pro system board provides four user LEDs that can be turned "ON" by driving the

LEDx signal to a logic "0". The following table shows the user LEDs and their associated Virtex-II

Pro FPGA pin assignments.

Table 30 – Push Switch Pin Assignments

Signal Name	Virtex-II Pro Pin #	Description
PUSH1	G25	SW3
PUSH2	H25	SW4
PUSH3	G26	SW5
PUSH4	H26	SW6

Table 32 – LED Pin Assignments

LED Designation	LED #	Virtex-II Pro Pin #
DS18	LED1	E31
DS19	LED2	E32
DS20	LED3	F31
DS21	LED4	F30
DS13	LED5	E1
DS14	LED6	E2
DS15	LED7	E3
DS16	LED8	E4