UNIVERSITETET I OSLO
Institutt for informatikk

Sovemodi for laveffekts mikrokontroller

Masteroppgave
(60 studiepoeng)

Tom-Arne Danielsen

18. juni 2009

Sammendrag

Denne masteroppgaven, etter oppgave fra Energy Micro, går ut på å vurdere metoder for reduksjon av lekkasjestrøm i en laveffekts mikrokontroller. Effektforbruket i et System on a Chip(SoC) kan deles inn i dynamisk og statisk, hvor det statiske opptrer som følge av lekkasjestrømmer. Denne lekkasjestrømmen er eksponentielt økende med nedskaleringen av geometrien i CMOS. I teknologier fra 65nm og nedover kan det statiske effektforbruket væer den dominerende del av det totale effektforbruket. Dette vil føre til at batteridrevne enheter slik som mobiltelefoner og bærbare musikkspillere kan forbedre batterilevetiden betraktelig hvis de benytter seg av metoder for reduksjon av statisk effektforbruk.

Metoden som benyttes i denne masteroppgaven er MTCMOS(forskjellige terskelspenninger) Power Gating. Denne metoden benytter seg av virtuelle spenningsskinner som man kan skru av og på ved hjelp av sovetransistorer. Ved å skru av spenningsskinnene vil spenningen som ligger over logiske blokker kunne reduseres betraktelig. Dette gjør at lekkasjestrømmen kan reduseres svært mye, opptil en faktor på 250x.

Det praktiske arbeidet tar for seg implementering av sovemodi, ved hjelp av MTCMOS Power Gating, for et system bestående av en mikroprosessor, en timer og UART. Med sovemodus menes en tilstand hvor spenningstilførselen til logikken reduseres eller tas bort, slik at lekkasjestrømmen reduseres. Dette arbeidet innebærer design av spenningskontroller, bevaring av kritisk registerinnhold, signalisolering, vurdering av oppvåkningsmetoder og strategier for maksimal effektbesparelser i en mikrokontroller. Systemet er også konfigurert med hjelp av UPF, som er et relativt nytt HDL fra Accellera for beskrivelse av spennings og effektegenskaper. Med de endringene som er utført på systemet er det mulig å oppnå et statisk effektforbruk på 1/90 av det opprinnelige statiske effektforbruket. Metoden har en arealkostnad som for hele systemet vil ligge på ca 30 % hvis mikroprosessoren benytter seg av *en* sovetilstand, og ca 50 % hvis det benyttes *tre* sovetilstander. Argumentet for å benytte flere sovetilstander er at det muliggjør raskere oppvåkning, og kan dermed redusere lekkasjestrømmer der hvor aktivitetsprofilen ikke tillater full søvn.

Akronymer

CMOS	Komplementær MOS	*Halvlederteknologi.*
ASIC	Application-specific integrated circuit	*Integrert krets designet for et spesielt bruksområde*
FPGA	Field Programmable Gate Array	*Rekonfigurerbar logikk*
SoC	System on a Chip	*Realisering av elektronisk system på en chip*
HDL	Hardware Description Language	*Programmeringsbeskrivelse av hardwareoppførsel*
RTL	Register Transfer Level	*Beskrivelse av system på registernivå*
VHDL	VHSIC Hardware Description Language	*En standard av HDL språk*
UPF	The Unified Power Format	*HDL fra Accellera for beskrivelse av spenningsegenskaper*
CPF	The Common Power Format	*HDL fra Cadence for beskrivelse av spenningsegenskaper*
VDD	Voltage Drain Drain	*Høy spenningstilførsel*
VSS	Voltage Source Source	*Lav spenningstilførsel*
VVDD	Virtuell VDD	*Kontrollerbar VDD*
VVSS	Virtuell VSS	*Kontrollerbar VSS*
SW_VDD	Switched VDD	*VVDD*
CPU	Central Processing Unit	*Prosessor*
ZPU	Zylin CPU	*Open source mikroprosessor*
UART	Universal Asynchronous Receiver/Transmitter	*Komponent for oversetting av data mellom seriell og parallell form.*
LSB	Least significant bit	*Laveste bit i en bitstring*
MSB	Most significant bit	*Høyeste bit i en bitstring*
GCC	GNU Compiler Collection	*Kompilerigssystem utviklet av GNU Systems, for støtte av flere programmeringsspråk.*

Forord

Først og fremst ønsker jeg å takke mine veiledere, Professor Jim Tørresen ved Institutt for Informatikk og Øyvind Janbu ved Energy Micro, for all tid dere har avsett til meg i løpet av perioden masteroppgaven har strukket seg over. Dere har begge bidratt med uvurderlig faglig tilbakemelding på hvert deres felt.

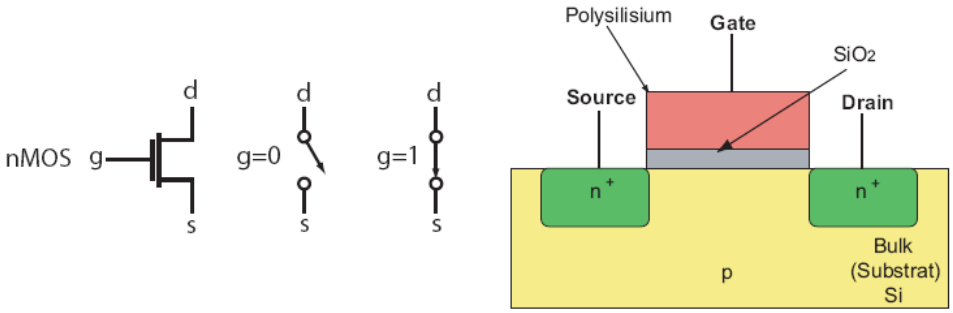
For simuleringer med UPF er QuestaSim fra Mentor Graphics benyttet. Digital design med UPF er et ganske nytt område med lite tilgjengelig dokumentasjon. Det har derfor gått med relativt mye tid til å plages med ”barnesykdommer” som nye programmer ofte har. Jeg ønsker å takke senioringeniør Olav Stanley Kyrvestad ved faggruppen for nanoelektronikk for sin rolle som link til supportavdelingen til Mentor, samt for all hjelp med anskaffelse av forskjellig programvare.

Sist, men ikke minst, vil jeg takke min kjære samboer Mari for at du er den du er. Din styrke er en inspirasjon til å stå på videre.

Innholdsfortegnelse

1. Introduksjon

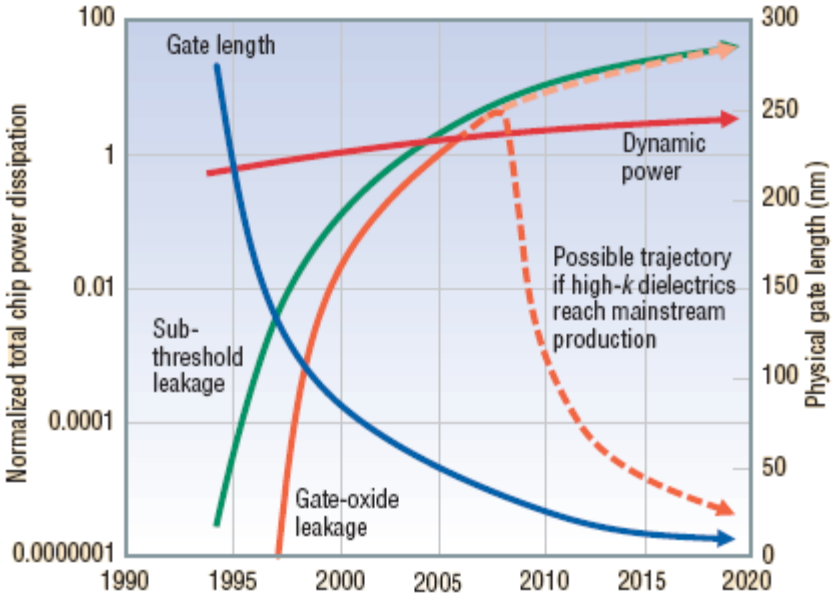
Utviklingen av silisiumsbaserte halvledere kan kalles vår tids oppfinnelse av hjulet. Teknologien finner vei inn i flere og flere applikasjoner, og fører til bedre og bedre ytelse. Der en prosessor inneholdt rundt 2500 transistorer pr 1971, har vi i dag godt over 1 mrd transistorer, som ikke er større enn 24 nm hver. Og utviklingen ser ut til å fortsette. Gordon Moore fikk i 1965 publisert en artikkel [4] i Electronics Magazine hvor han kom med påstanden at antall transistorer i prosessorer vil dobles annethvert år. Det innebærer også at den fysiske størrelsen på transistorene stadig må krympes. Påstanden har senere blitt omtalt som Moores lov. Det ble gjort en endring på loven i 1975, hvor ordlyden ble endret til at doblingen ville skje hver 18. måned, og Moores lov har etter det vært svært nøyaktig. Det hersker dog uenighet om hvor lenge den vil gjelde videre framover.



Figur 1 - nMOS transistor

Uavhengig av hvorvidt Moores lov vil forbli gjeldene er det en ting som er sikkert; dagens transistorer er meget små, noe som fører med seg en stor utfordring. Der hvor en transistor tidligere har vært skrudd av, og dermed ikke ledet nevneverdig strøm fra source til drain, vil det i dagens transistorer finnes betydelige lekkasjestrømmer. Det vil også kunne gå strøm gjennom gate, ettersom tykkelsen på oksidet, SiO_2(se Figur 1), som skal isolere denne strømmen kun er på et par atomtykkelser. Dermed vil det alltid finnes enkelte passasjer gjennom oksidet for elektronene.

Lekkasjestrømmen mellom drain og source, $\boldsymbol{I_{SUB}}$, i en transistor kaller man gjerne for statisk strømtrekk, mens det aktive strømtrekket kalles dynamisk. Faktisk kan det for veldig små teknologier (ca 65 nm og ned) være det statiske effektforbruket som er den dominerende del av totalt effektforbruk, noe vi kan se av Figur 2. Det er dermed opplagt at dette er et problem som kan føre til uønsket stort effektforbruk, spesielt for applikasjoner som drives av batteri, slik som mobiltelefoner og mp3spillere.



Figur 2 – Dynamisk og statisk effektforbruk i synkende geometrier[17]

Ifølge *International Roadmap for Semiconductors* (ITRS) opplevde handholdte enheter en topp i levetiden i 2004 [1][17]. Siden da har batterilevetiden blitt redusert ettersom nye funksjoner har blitt lagt til raskere enn effektforbruk per funksjon har blitt redusert. Dette er en utfordring som vil bli større jo mindre størrelsen på transistorer blir. Ifølge ITRS vil det for 45nm teknologien være ca 6.5 ganger så stort statisk strømtrekk som ved 90nm teknologi.

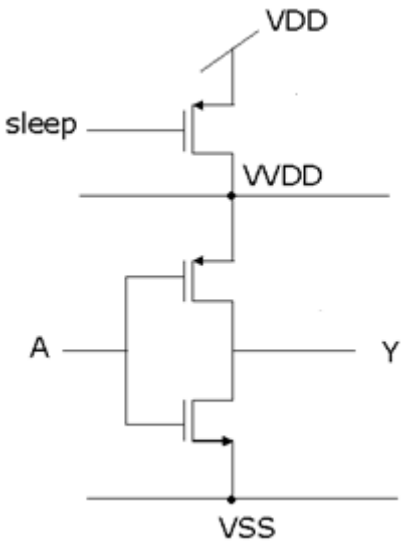
Lekkasjestrømmen fra drain til source i en transistor kan gis ved følgende formel:

$$I_{SUB} = \mu C_{ox} V_{th}^{\,2} \frac{W}{L} e^{\frac{V_{GS}-V_T}{nV_{th}}} \qquad \text{(eq.1)}$$

hvor W er bredde på transistoren, L er lengde, V_{GS} er gate-source spenning og V_T er terskelspenning. Vi ser ut i fra dette at lekkasjestrømmen er eksponentialt avhengig av differansen mellom V_{GS} og V_T. Så når V_{DD}(som V_{GS} er proporsjonalt avhengig av) og V_T skaleres ned i takt med teknologien vil samtidig lekkasjestrømmen forverres drastisk.

Tidligere har effektreduksjon vært nedprioritert i design av *System on a Chip*(SoC), til fordel for elementer som kostnad, areal og timing. I dag er effektbudsjettet en av de viktigste faktorene ved design av SoC. Det jobbes hardt med problemstillinger tilknyttet effektforbruk, ettersom man allerede ved 90nm er på grensen av hva kundene vil akseptere.

Vi vil i denne rapporten få et lite innblikk i hvilke metoder som er tilgjengelig for reduksjon av statisk effektforbruk. Den metoden som er funnet mest lovende, og som hovedsakelig fokuseres på er Power Gating. Denne metoden benytter seg av virtuelle spenningsskinner(se Figur 3), som man ved hjelp av sovetransistorer kan skru av og på. Man vil da i teorien kunne oppnå at VDD og VSS får samme spenning. Dette vil føre til at det ikke ligger noen spenning over transistorene, og lekkasjestrømmen går mot null. I praksis vil også sovetransistorene ha noe lekkasjestrøm, noe som gjør at statisk effektforbruk ikke vil forsvinne helt.



Figur 3 - Power Gating med virtuell VDD

Det er flere utfordringer forbundet med Power Gating, noe vi vil se nærmere på. Blant annet vil verdien til registre som mister spenningen bli ødelagt. Vi må benytte metoder som sørger for kritiske data bevares, slik at tilstanden kan gjenopprettes.

Kapittel 2 i denne rapporten tar for seg eksisterende metoder for redusering av statisk strømtrekk. Vi vil også få en innføring i designutfordringer ved bruk av Power Gating, og se på hvordan disse utfordringene kan løses. Det vises også hvilke designutfordringer som kan løses med UPF (*The Unified Power Format*). Til slutt gis det en liten introduksjon til ZPU, mikroprosessoren som er benyttet i dette prosjektet.

Kapittel 3 vil vise hvilke endringer som er utført på arkitekturen til ZPU for å muliggjøre soving, og hvordan utfordringene beskrevet i kapittel 2 er løst. Vi vil først se på endringer som er nødvendig for å etterlikne bruk av sovemodus i en FPGA prototype av en mikrokontroller, før vi ser på implementering av UPF, som kan benyttes i et ASIC design. Det er også foreslått forskjellige metoder for oppvåkning, samt strategier hvor inaktive komponenter i en mikrokontroller settes til å sove sammen med mikroprosessoren.

Kapittel 4 inneholder simuleringer for de endringer som er utført i kapittel 3, hvor funksjonaliteten blir verifisert. Det vil også vises hvilke besparelser som implementeringene muliggjør, samt hvilke kostnader de medfører.

Kapittel 5 inneholder tall på mulig effektbesparelse ved bruk av Power Gating i en mikrokontroller, samt hvilke arealkostnader som må forventes. Det er også gitt noen vurderinger for Power Gating ved bruk av andre CMOS-teknologier. Kapittelet avsluttes med noen nøkkeltall for bruk av soving, før vi konkluderer med hvorvidt det er nyttig.

2. Bakgrunn

Som nevnt i kapittel 1 er problemet med statisk strømtrekk rimelig nytt av dato. Selve fenomenet har vært kjent lenge, men det er i den siste tiden at teknologien har nådd et nivå hvor det blir kritisk å komme med løsninger for å takle det. Flere og flere selskaper velger også å satse på en ”grønn” profil, og produkter som har et minimalt effektforbruk vil derfor trolig bli mer og mer etterspurt i fremtiden. Dette kapittelet tar for seg tradisjonelle metoder for reduksjon av statisk strømtrekk, samt gir leseren et innblikk i Power Gating, som er en mer aggressiv måte å takle lekkasjestrøm på. Det vil også bli gitt en presentasjon av UPF(The Unified Power Format), som er Accelleras bidrag til laveffekts hardware beskrivelser. Til slutt ser vi på ZPU, mikroprosessoren som er benyttet i denne masteroppgaven.

2.1 Reduksjon av statisk strømtrekk

Det finnes i dag flere metoder for reduksjon av lekkasjestrøm og statisk effektforbruk. Det er et ønske fra oppdragsgiver at maksimal lekkasjestrøm når mikroprosessoren er satt i sovemodus ikke skal overskride 1µA. Vi vil her se på forskjellige metoder for reduksjon av lekkasjestrøm, og vurdere hvorvidt de vil kunne møte dette kravet.

2.1.1 Variabel Terskel CMOS

Variabel Terskel CMOS (VTCMOS) går ut på å tilføre reversert bias spenning til substratet, og dermed redusere forholdet ($V_{GS} - V_T$), da V_T øker. Denne metoden kan redusere lekkasjestrøm betraktelig, men har den ulempen at den krever to spenningsnettverk for å kontrollere spenningen som er koplet til brønnen. Det har også vist seg at effekten av denne metoden er avtakende med nedskalering av teknologien [5]. Ved 90nm er det rapportert om 4x reduksjon, og kun 2x ved 65nm [6]. Det antas på bakgrunn av dette at denne metoden ikke vil klare å møte kravene for reduksjon av lekkasjestrøm.

2.1.2 Lang kanallengde

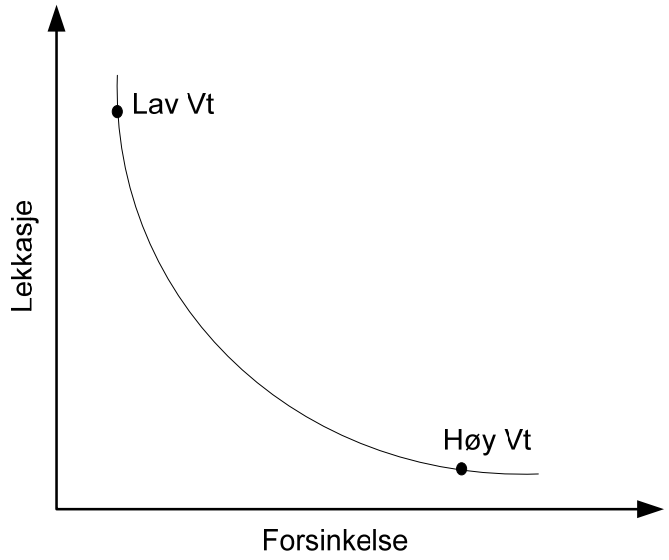
Ut i fra (eq.1) er det klart at lekkasjestrømmen kan reduseres ved ikke å benytte minimumslengde på transistorene. Men dette vil igjen føre til lavere dynamisk strøm, noe som fører til lavere ytelse. Det vil også medføre større transistorer som har større gate kapasitans, som igjen vil redusere ytelsen ytterligere. I tillegg vil det være en betydelig arealkostnad ved bruk av lengre kanaler. Grunnet dette er ikke denne metoden særlig godt egnet for reduksjon av lekkasjestrøm, selv om den kan ha mulighet for å redusere lekkasjestrømmen til ønsket nivå. Et design som benytter seg av transistorer med bare lange kanaler vil heller ikke rettferdiggjøre at man reduserer geometrien i nyere teknologier.

2.1.3 Dynamisk spenningsskalering

Dynamisk spenningsskalering(DSS) benyttes hovedsakelig for å redusere dynamisk effektforbruk. Men denne metoden kan også benyttes for å redusere statisk effektforbruk. DSS virker slik at når arbeidsprofilen ikke krever maksimal ytelse reduseres VDD. Ved å redusere VDD når kretsen er inaktiv er det rapportert om lekkasjebesparelser på 8x til 16x [7]. DSS kan også kombineres med VTCMOS for ennå bedre reduksjon [8]. Bakdelen med denne metoden er at den krever utvidelse av den eksiterende kretsen for å forutse arbeidsprofilen til blokken. Det er også nødvendig med en dynamisk spenningsregulator for å justere VDD. Metoden anses derfor ikke som optimal for reduksjon av lekkasjestrøm.

2.1.4 *Multi* V_T

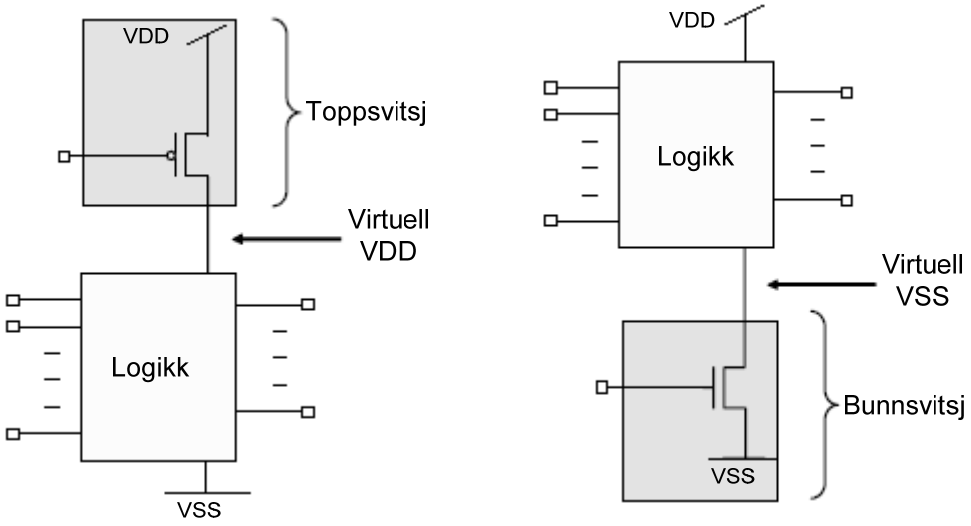
Multi V_T er en ganske vanlig metode for lekkasjereduksjon, og går kort sagt ut på å benytte celler med høy V_T der hvor ytelseskravene tillater det, og celler med lav V_T der hvor man må møte krav til timing [9]. Mange bibliotek tilbyr tre versjoner av celler, Lav V_T, Standard V_T og Høy V_T. Det er vanlig å syntetisere med høy ytelse- og lekkasjebiblioteker først, for så å bytte ut de cellene som ikke er timingskritiske med lav ytelse- og lekkasjeekvivalenter. Den store fordelen med *Multi* V_T er at det ikke kreves noen endring i layout. Cellene byttes ut der de ligger, og ingen andre endringer er nødvendig. Multi V_T har lite timing og arealkostnad, men har også bare middels effektbesparelse, noe som vil si at metoden ikke vil møte kravene for lekkasjereduksjon.



Figur 4 - Høy terskelspenning vs. lav terskelspenning

2.1.5 *MTCMOS Power Gating*

Power Gating [2][7][10][14] er en noe nyere metode for reduksjon av statisk effektforbruk enn de overnevnte. Helt enkelt går dette ut på å plassere sovetransistorer mellom logikken og enten VDD eller VSS, som illustrert i Figur 5. Man får da en virtuell spenningsskinne som man kan skru av og på, og dermed skru driftsspenningen til logikken av og på. For ennå bedre resultat er det optimalt å kombinere denne teknikken med Multi V_T, derav navnet MTCMOS. All logikk implementeres ved å benytte celler med Lav V_T, mens sovetransistorene implementeres med Høy V_T celler. Power Gating vil ikke eliminere statisk effektforbruk på grunn av lekkasjestrøm i sovetransistorene, men det er meldt om reduksjoner på 250x[10].



Figur 5 – Toppsvitsjing og bunnsvitsjing

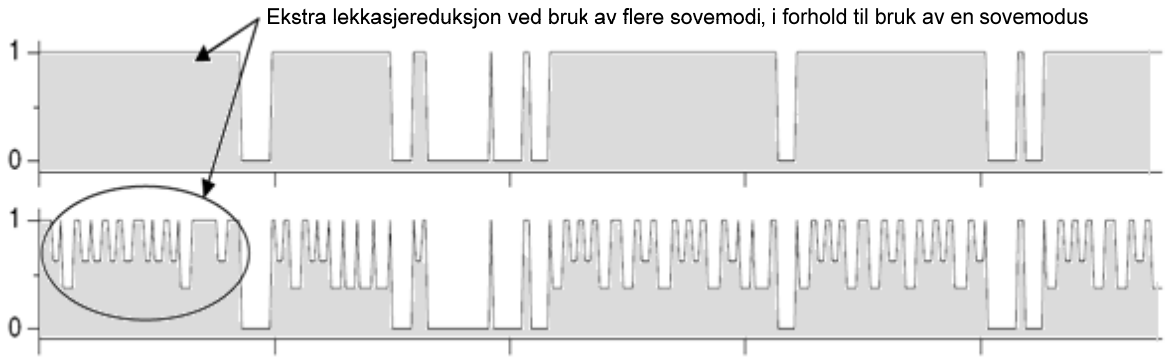
Tilfellet hvor sovetransistoren plasseres mellom VDD og VVDD(virtuell VDD) kalles toppsvitsjing. Denne metoden benytter en pMOS transistor som sovetransistor. Motsatt tilfelle kalles bunnsvitsjing, og man benytter da nMOS transistor. Begge disse metodene er illustrert i figuren over. Fordeler og ulemper med disse metodene vil bli nærmere diskutert i 2.2.1.

Det finnes også andre varianter av Power Gating. To av disse er ZigZag Power Gating [15] og Super Cut-off CMOS [11]. ZigZag Power Gating metoden benytter seg av både topp og bunnsvitsj. Fordelen med dette er at oppvåkningstiden reduseres, siden de virtuelle spenningsskinnene trenger kortere tid til å stabiliseres. Bakdelen med denne metoden er at bruk av både nMOS og pMOS krever et veldig komplisert spenningsnettverk. Spenningsfallet som sovetransistorene medfører vil også kunne bli stort. Super Cut-off CMOS benytter sovetransistorer med lav V_T. I standby modus er transistorene presset lengre inn i cut-off området ved å påføre en gate spenning som er under VSS for bunnsvitsjing og over VDD for toppsvitsjing, for dermed å minke V_{GS} mer enn hva som er mulig med å benytte tradisjonell gate spenning. Disse to sistnevnte metoder er også kombinert i ZigZag Super Cut-Off CMOS [12].

2.2 Power Gating Design

Power Gating kan fra et overordnet nivå gjøres på to måter. Den første er *Full Power Gating*. Ved denne metoden benyttes det ikke flere spenningsdomener. Ved soving skrur man av all logikk. All informasjon vil da gå tapt så fremt man ikke benytter bevarende registre. Den andre metoden er *Selektiv Power Gating*. Her ligger forskjellige blokker i forskjellige

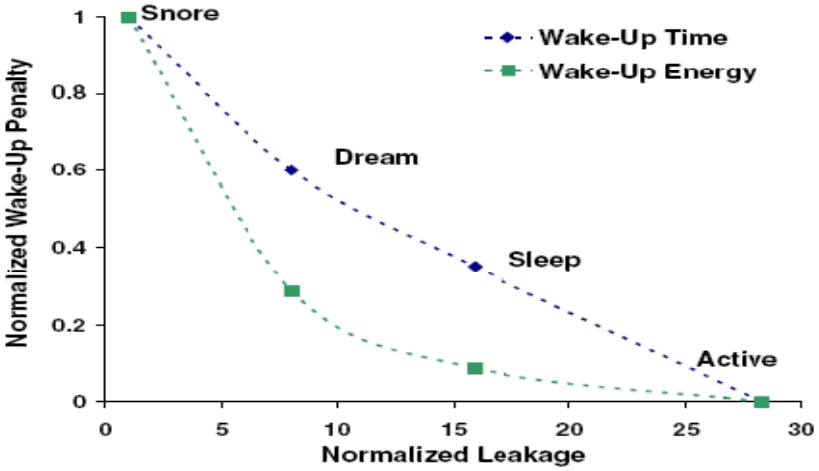
spenningsdomener, og man har muligheten til å skru av deler av kretsen, uavhengig av andre blokker. Dette vil kunne føre til bedre lekkasjereduksjon enn Full Power Gating da man har muligheten til å skru av inaktive komponenter.



Figur 6 - Effektforbruk ved en sovemodus vs. tre sovemodi [2]

For mer aggressiv reduksjon av statisk effektforbruk er det mulig å benytte seg av flere sovemodi[2][10]. Dette realiseres ved å ha flere spenninger som kan koples til virtuell spenningsskinne. Denne metoden har vist seg å kunne gi opptil 17 % ekstra reduksjon av effektforbruk[2]. Siden Power Gating medfører en tidskostnad er det ikke alltid hensiktsmessig å sette en blokk til å sove, da aktivitetsprofilen til logikken tilsier at den forventede sovetiden er for kort i forhold til oppvåkningstiden. Ved å kunne tilby en mindre reduksjon av spenningen på spenningsskinnen vil man også redusere oppvåkningstiden, og det vil kunne være mulighet for å aktivere soving oftere. Selv om ikke det statiske strømtrekket reduseres like mye som når man skrur driftspenningen helt av, vil det fortsatt være en forbedring fra å ha full driftspenning. Figur 6 gir et eksempel på ekstra lekkasjebesparelse ved bruk av flere sovemodi. Det grå området illustrerer total statisk effektforbruk, og vi ser at dette er lavere på den nederste figuren, som benytter seg av flere sovemodi.

En ulempe med Power Gating er uten tvil tidsforsinkelsen som oppstår når man går inn og ut av spenningsmodi. Jo lavere av- spenningen settes, jo lengre tid vil det ta for blokken å oppnå full driftspenning ved oppvåkning. Figur 7 illustrerer en tidskostnad versus lekkasjestrøm profil for bruk av flere sovemodi. Ved tilstanden *Snore* er det lavest lekkasje, men også størst oppvåkningstid. Denne tilstanden vil realiseres ved å skru virtuell spenningsskinne, VVDD, helt av. Tilstandene *Dream* og *Sleep* har lavere oppvåkningstid, men også større lekkasje. Disse tilstandene realiseres ved å skru VVDD delvis av. Tilstanden *Active* er som navnet tilsier den aktive tilstanden. VVDD er da helt på, slik at det ikke er noen oppvåkningstid.

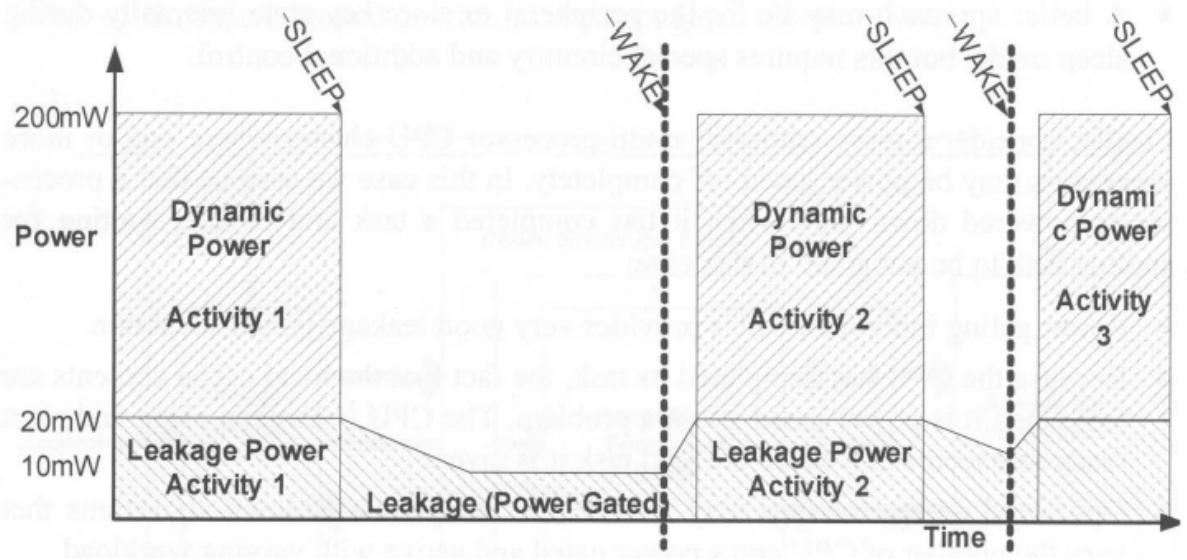


Figur 7 - Lekkasje versus tidskostnad [2]

I tillegg til tidsforsinkelsen vil det også være et dynamisk effektforbruk forbundet med det å gå i sovemodus. Dette kan være kritiske faktorer, og gjør at man må vurdere hvorvidt man skal gå i sovemodi ut ifra følgende punkter.

- Hvor mye effektforbruket kan reduseres.
- Energien som forbrukes når man går inn og ut av forskjellige modi
- Aktivitetsprofil. Hvor ofte skiftes det mellom aktivt modus og sovemodus, og hvor lenge er den i de forskjellige modi. Illustrert i Figur 8.
- Tidsforsinkelsene som vil oppstå ved skifte av modi.

Summen av de tre første punktene vil gi oss netto effektbesparelse. Denne må man så veie opp mot den medfølgende tidsforsinkelsen Power Gating medfører før man avgjør om blokken skal settes i sovemodus.



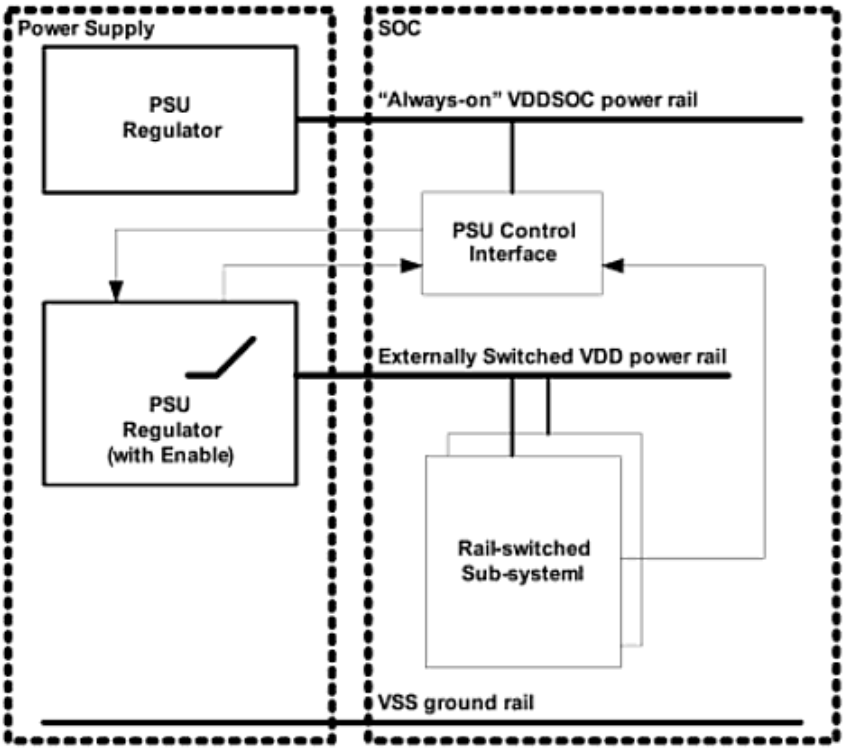
Figur 8 – Eksempel på aktivitetsprofil med Power Gating [1]

En av faktorene som kan føre til effektforbruk når man går inn og ut av sovemodus er forbundet med bevaring av tilstand. Setter man en blokk i en tilstand hvor kritisk innhold ikke blir bevart, må man gjøre handlinger for å kunne gjenopprette tilstanden. Denne gjenopprettningen vil føre med seg effekt- og tidsforbruk. Det er ikke alltid at der er nødvendig å erstatte innholdet i registrene. Er det for eksempel snakk om sendebufferet til en UART kan man anta at den har fullført oppgaven sin når den går i sovemodi, og dermed kan våkne opp igjen med tomt buffer, for å utføre neste oppgave. Derimot kan det være ønskelig at konfigurasjonen for baudrate blir bevart, slik at den slipper å få dette lastet inn etter hver sovesekvens. Vi ser nærmere på bevaring av tilstand i 2.2.3.

2.2.1 Sovetransistorer

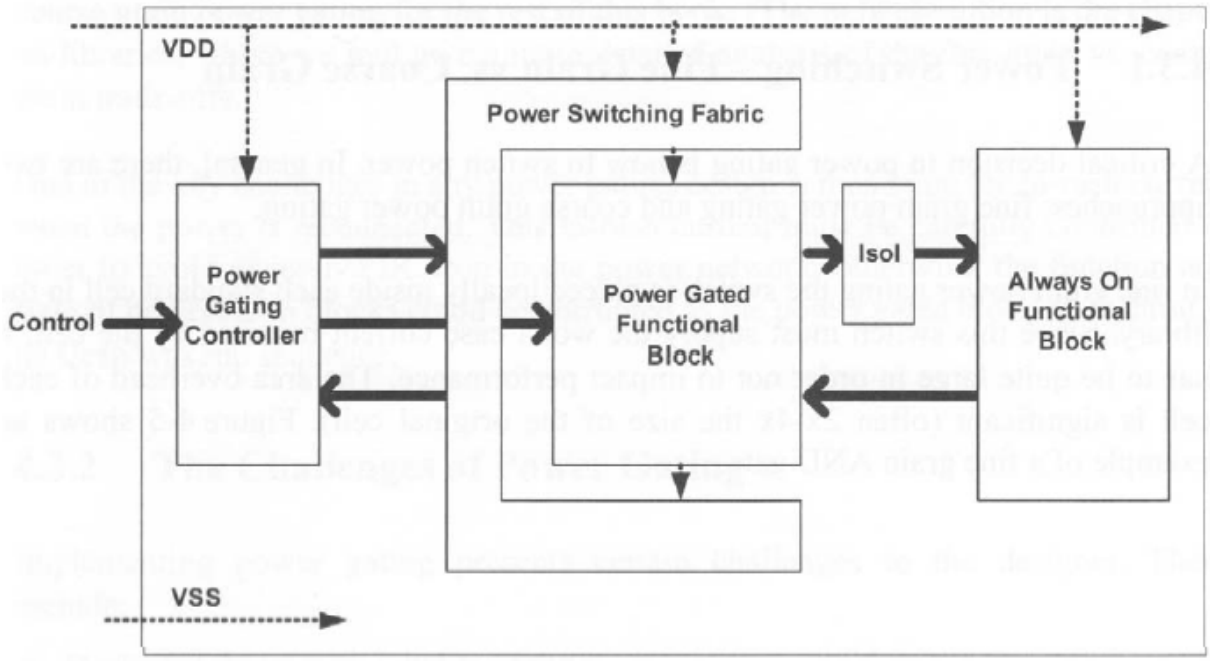
Realisering av sovetransistorene kan skje ved bruk av to forskjellige framgangsmåter, *finkornet* og *grovkornet*. I den finkornede metoden ligger svitsjen lokalt inni hver celle. Siden svitsjen minimum må kunne levere verste mulige strøm som cellen krever må den være veldig stor, gjerne 2 – 4x av den opprinnelige cellen. Nettopp av denne grunn er ikke dette den optimale metode for Power Svitsjing i en mikroprosessor. Det er da bedre å benytte seg av grovkornet implementering. Man har da en samling av svitsjeceller som styrer spenningen i hele blokker, som illustrert i Figur 10 på neste side. Denne har betydelig mindre arealkostnad enn den finkornede metoden.

Grovkornet Power Gating kan gjøres med en intern eller ekstern svitsj. For blokker som skal være skrudd av i lengre perioder kan det være best med ekstern svitsj, som vist i Figur 9, ettersom den har mest effektiv reduksjon av lekkasjestrømmen. Denne har derimot den ulempen at den krever lang tid, og mye energi for å gjenopprette spenningen til en blokk. Dette er på grunn av større kapasitanser som skal lades opp ved oppvåkning, i forhold til intern svitsj. Oppvåkningstiden ved denne metoden kan i verste fall være på millisekunder eller mer. Den har også en ulempe ved at den krever egne pinner på pakken, noe som ikke er ønskelig at den skal ha.



Figur 9 - Ekstern Power Rail svitsjing [1]

En bedre løsning for blokker som skal skrus av for kortere perioder er å ha en intern svitsj. Figur 10 viser er intern Power Gating struktur, med toppsvitsjing. Denne metoden har mindre kostnad i forbindelse med oppvåkningstid enn ekstern svitsjing, men har til gjengjeld en lavere lekkasjereduksjon.



Figur 10 - SoC med Power Gating [1]

Power Switching Fabric fra Figur 10 består typisk av flere CMOS svitsjer som er plassert omkring eller inne i den Power Gatede blokken. Disse blir kontrollert av *Power Gating Controller,* som i denne rapporten vil bli referert til som spenningskontroller.

Hvorvidt man skal benytte topp- eller bunnsvitsjing (Figur 5, side 10) må vurderes i hvert enkelt tilfelle, da begge har sine fordeler og ulemper. Grunnen til at man ikke bruker begge er at det vil føre til et betydelig spenningsfall, som ofte vil føre til et ytelsestap større enn hva som er akseptabelt. Toppsvitsjing benytter pMOS transistorer som sovetransistorer. Disse har mindre mobilitet enn nMOS av samme størrelse, noe som vil føre til at toppsvitsjing vil kreve mer areal for å kunne levere samme strøm. Fordelen er derimot at pMOS har lavere lekkasje enn nMOS. En annen stor faktor i denne avgjørelsen vil være avhengig av hvilken type isolering av utgangene(omtalt i 2.2.2) man velger. Velger man å benytte bunnsvitsjing har ikke blokken VSS tilgjengelig under soving, og isolering til logisk ”0” vil kreve komplekse løsninger. Det samme gjelder toppsvitsjing og isolering til logisk ”1”. En annen faktor ved valg at topp eller bunnsvitsjing er hvorvidt man benytter ”twin-well” eller ”single-well” prosess. Siden nMOS deler substratkoplinger på chippen i en single-well prosess er det vanskelig å bruke bunnsvitsjing. Brønnisoleringen som trengs for å forhindre kortsluttninger kan gjøre at arealbesparelsen av bunnsvitsjing forsvinner. Det finnes flere grunner til at topp celler burde benyttes, blant annet er det sterkt anbefalt at man benytter topp-svitsjing der hvor man har flere spenningsskinner, eller man benytter seg av spenningsskalering mellom blokker. [1]

Man må også avgjøre størrelsen på sovetransistorene. Den viktigste grunnen til dette er for å redusere ytelsestap i logikken grunnet innsetting av sovetransistorer.

For en vanlig CMOS gate kan man utrykke forsinkelsen over den som

$$T_{pd} = \frac{C_L V_{DD}}{(V_{dd} - V_{tL})^{\alpha}} \qquad \text{(eq.2)}$$

hvor C_L er lastkapasitansen, V_{tL} er terskelspenning i den modul med lav terskelspenning og α er hastighetsmettningsindeks som modellerer kanalforkortning[3].

Med en sovetransistor kan (eq.2) skrives som

$$T_{pd-MT} = \frac{C_L V_{DD}}{(V_{dd} - V_{st} - v_{tL})^{\alpha}} \quad \text{(eq.3)}$$

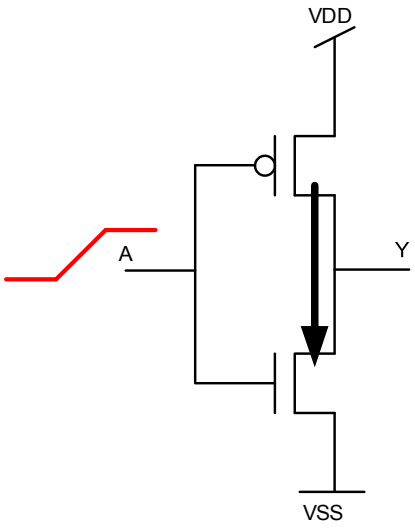
hvor V_{st} er spenningsfall fra source til drain for sovetransistoren.

Vi får da følgende utrykk for størrelsesforholdet til sovetransistoren når vi ønsker et ytelsestap mindre enn δ.

$$(\frac{W}{L})_{st} > \frac{I_{st}}{\delta \mu_n C_{ox}(V_{dd} - V_{tL})(V_{dd} - V_{tH})} \quad \text{(eq.4)}$$

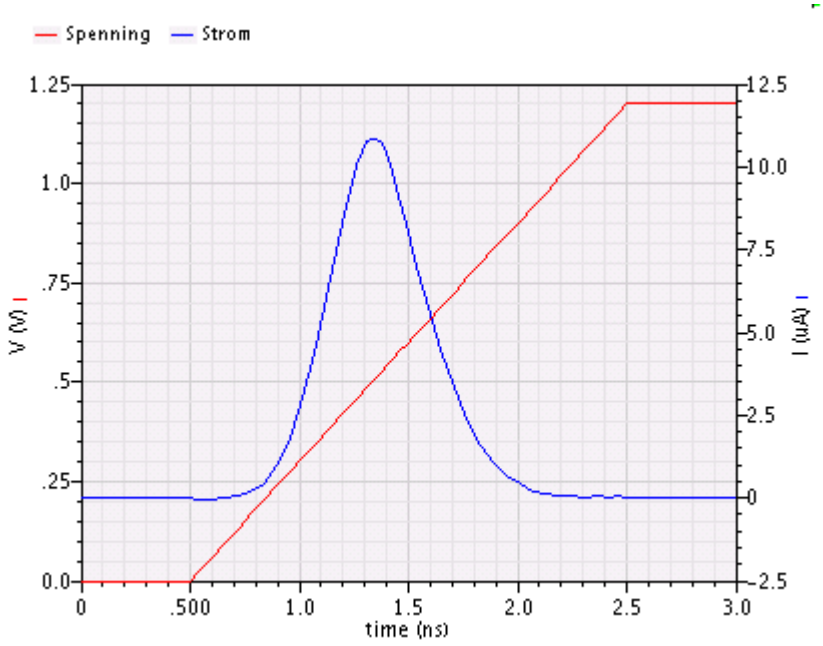
hvor V_{tH} er terskelspenning for sovetransistoren.

2.2.2 *Signalisolering*



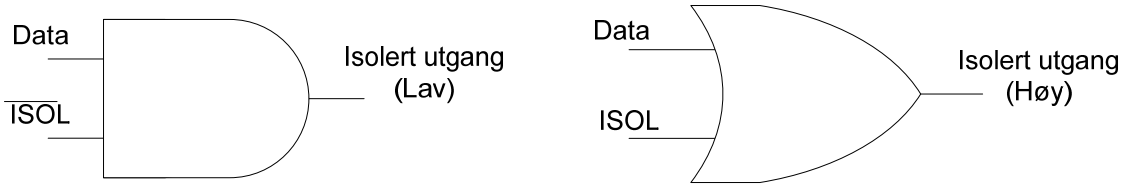
Figur 11 – Kortsluttningsstrøm

Isol blokken fra Figur 10, side 14, har som funksjon å hindre kortsluttningsstrøm(Figur 11) i *Always On* blokken. Grunnen til at man kan oppleve kortsluttningsstrøm er at utgangen til *Power Gated Functional Block* kan bevege seg veldig sakte fra 1 til 0 når man stenger av blokken. Dette fører til at utgangen befinner seg ved terskelspenningen til *Always On* blokkens transistorer i lengre tid. Disse strømmene er veldig uheldige, og vil gi høyt effektforbruk [13][18]. Strømkarakteristikk for inngang på en inverter som går fra 0 til 1,2V er vist i Figur 12.



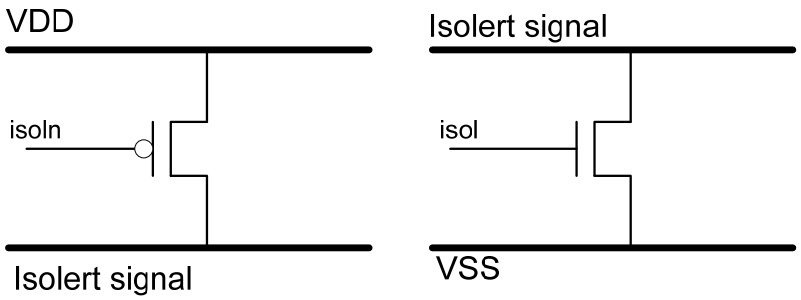
Figur 12 – Kortsluttningsstrøm karakteristikk

Når inngangen til inverteren i Figur 11 befinner seg ved ca Vdd/2 vil det gå mye strøm igjennom inverteren, og dermed være et stort effektforbruk. Dette effektforbruket kan være større enn den effektbesparelsen man oppnår med bruk av sovemodus, noe som fører til at bruk av sovemodus ikke kan forsvares.



Figur 13 - Isoleringsceller

Isolering kan realiseres med tre forskjellige teknikker. Den første går ut på å klemme utgangen til verdien ”0”, neste klemmer den verdien til ”1” og den siste latcher den til den siste verdien. Det beste vil vanligvis være å klemme utgangen til en inaktiv verdi. Har man for eksempel aktiv høy logikk klemmer man utgangen til ”0” og motsatt. Dette kan gjøres ved hjelp av AND eller OR logikk, som illustrert i Figur 13, eller med opptrekk-/nedtrekkstransistorer, som illustrert i Figur 14.



Figur 14 – Opptrekk- og nedtrekkstransistorer

For å unngå uheldige utgangssignaler er det viktig at utgangene forblir isolert under hele power-up prosessen, og at de forblir isolert til spenningen er stabilisert. Denne prosessen er enkel med isoleringscellene, men kan være noe utfordrende med opptrekk-/nedtrekkstransistorer. Disse løsningene fører også med seg flere andre problemer, blant annet kan de være problematisk å produksjonsteste. Derfor anbefales det å benytte de metodene som benytter AND og OR porter [1]. Likevel kan transistor løsningene være brukbar i spesielle situasjoner hvor timing er kritisk, da bruk av isoleringsceller fører til noe forsinkelse på det isolerte signalet siden de benytter AND/OR logikk. Denne tidsforsinkelsen har ikke transistorløsningene.

Signalet kan både isoleres ved kilden til signalet eller ved destinasjonen. I praksis kan det føre med seg noen vanskeligheter når man isolerer ved destinasjonen til signalet. Vi kan for eksempel se for oss at det isolerte signalet går til mer enn en blokk. Dette vil kreve isolering på alle blokkene hvis man skal isolere ved destinasjonen. Det er derfor ofte mer arealeffektivt å isolere ved kilden. Dette vil også gjøre analyse enklere, ettersom det vil være mye enklere å sjekke at signalet faktisk er isolert. Bakdelen med å isolere inni den sovende blokken er at det vil kreve en spenningskilde ekstra, ettersom isoleringscellen alltid må være på. Ved å velge topp eller bunnsvitsjing i forhold til den isoleringen man ønsker, blir dette problemet neglisjerbart.

2.2.3 Gjenopprettelse av tilstand

Ved bruk av Power Gating vil man kunne miste all informasjon som ligger i blokken som blir satt i sovemodi. Gjenopprettelse av denne informasjonen kan være både tids- og effektkrevende. Det er som tidligere nevnt ikke alltid nødvendig å gjenopprette all informasjonen. Men er det for eksempel prosessoren som blir satt til å sove vil man at det skal våkne opp med den informasjonen som det hadde. Følgende 3 teknikker kan brukes for å gjenopprette denne tilstanden [1].

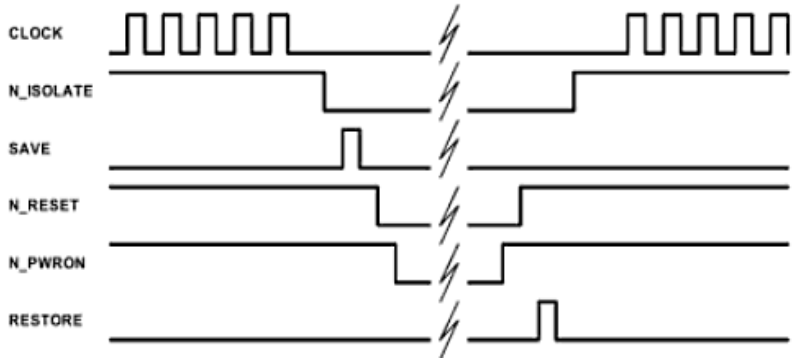
Softwaretilnærming basert på å lese/skrive register. Denne har en ekstra prosessor i en blokk som er kontinuerlig på, og som leser tilstanden til blokken som skal sove. Informasjonen lagres så i prosessorens minne. Motsatt blir den lastet tilbake ved oppvåkning. Denne metoden har flere ulemper, som for eksempel at trafikken på bussen fører til et tregt skifte fra våken til sovemodi og motsatt. Alle registrene må også gjøres tilgjengelig på bussen, noe som krever endringer i hardwarestrukturen. Den krever i tillegg at alle tilstandene finnes i software, noe som ikke nødvendigvis er tilgjengelig. Disse ulempene gjør at den ikke er veldig praktisk å benytte.

Scankjeder som er implementert for fabrikasjonstesting kan brukes til å gjenopprette informasjonen nesten uten å øke arealet. For å realisere dette shifter man scanregistrene som i testing, men laster utgangen til et minne med kontinuerlig spenning. Ved oppvåkning gjentas samme prosess men man shifter fra minnet til registrene. En utfordring med denne metoden er at scankjeder ikke blir satt inn før man kommer til syntetisering, mens det er nødvendig å kunne feilsøke på RTL(Register Transfer Level) nivå, som er før syntesen. En løsning på dette kan være å benytte dummy scankjeder under testing på RTL nivå. Bruk av scankjeder vil føre med seg en tidsforsinkelse, som er en funksjon av hvor stor blokk som skal skannes ut, og hvor mange scankjeder som brukes. Det er også relativt store dynamiske effektkostnader ved bruk av scankjeder. Det kan kreves mye energi for å shifte all dataen gjennom scankjeden.

Bevaringsregistre er som nevnt tidligere et register som også inneholder et skyggeregister. Dette skyggeregisteret er alltid på, men har et lavere effektforbruk. Informasjonen i hovedregisteret blir kopiert til skyggeregisteret når man går i sovemodi, og kopiert tilbake etter oppvåkning. Denne metoden har noe arealkostnad, typisk 20 %. Enkelte bevaringsregistre bruker en spesiell type isolering mellom skyggeregisteret og det registeret som skal sove. I disse kan arealkostnaden være opp mot 50 %. Bruk av bevaringsregistre vil også føre til en mer kompleks spenningskontroller. Ved bruk av soving må man vurdere hvilke registre man vil bevare. Velger man å bevare alle vil det kunne føre til uakseptabelt store arealkostnader, og det kan derfor være hensiktsmessig med delvis bevaring.

2.2.4 Spenningskontroller

Etter å ha valgt hvilke metoder man skal bruke for design av svitsjecellene, isoleringsmetode og metode for gjenopprettelse av tilstand kan man designe spenningskontrolleren. Det er denne som styrer rekkefølgen blant annet isolering og bevaring skal inntreffe på. For eksempel vil en spenningskontroller for styring av en blokk med isolering og bevarende registre kunne ha en flyt som vist i Figur 15.



Figur 15 – Spenningskontrollerens signaler [1]

Ved avstenging av spenningen vil man først stoppe klokken i riktig fase for å minimere lekkasje. Deretter setter man på isoleringssignalet for å sikre trygge utgangsverdier. Signal for lagring av tilstanden til registrene settes så, før de registre som ikke skal bevares resettes slik at de våkner opp i en resatt tilstand. Til sist blir spenningen til blokken satt av slik at blokken blir satt i sovemodi. Ved oppvåkning vil det da forløpe seg som illustrert i høyre halvdel av Figur 15.

Det er viktig at denne sekvensen foregår i riktig rekkefølge, og at det neste steget ikke inntreffer før forrige er ferdig. Dette er blant annet for å unngå høye strømmer ved oppstart, som videre kan føre til ødeleggende spenningshopp. Dette kan realiseres ved å benytte en handshake funksjon. Det ene steget i sekvensen vil da kvittere for at den er ferdig, og neste steg kan da starte opp. Det er her svært viktig at sekvensen til høyre i figuren ikke initieres før blokken har fått full driftsspenning, da dette kan føre til feil.

2.2.5 Verifikasjon og simulering

Verifikasjon av en Power Gated krets kan være en utfordring, ettersom HDL (Hardware Description Language) ikke tilbyr en mekanisme for å beskrive spenningstilkoplinger på RTL nivå. For å simulere Power Gating må vi utvide HDL'en, enten med modifikasjon av koden eller med å benytte separate sett med kommandoer for å beskrive spenningskoplinger og spenningssvitsjing. Et alternativ er å benytte UPF(The Unified Power Format), som er en relativt ny standard fra Accellera. UPF tilbyr blant annet mekanismer for å definere et spenningsdomene og et sett med spenningsforsyninger til domenet. Det har også

kommandoer for å gjøre registre om til bevarende registre. UPF står nærmere forklart i kapittel 2.3.

2.2.6 Design for testing

Det er nødvendig å ha produksjonstesting og karakteriseringstesting av chip i bakhodet når man lager et design. En viktig faktor ved design for testing er å gjøre klokker og reset eksternt tilgjengelig. Man må også være i stand til å forhindre at scan mønstrene tilfeldigvis:

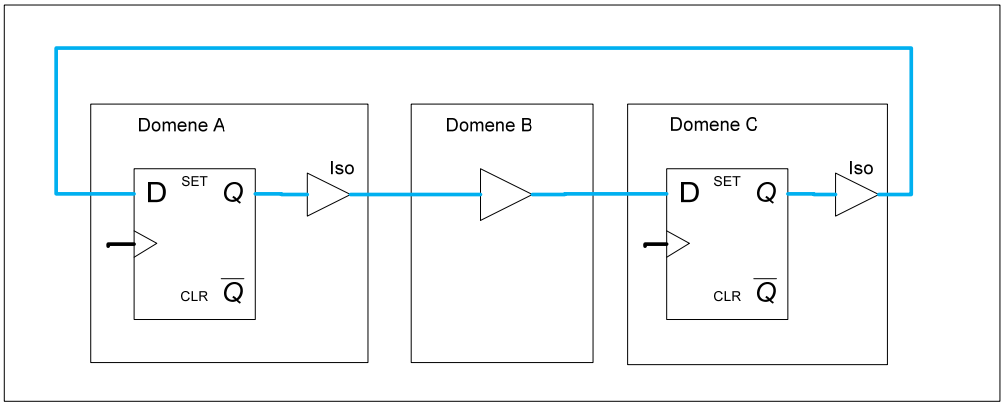
- Toggler tilstandsmaskinens utganger og aktiverer Power Gating.
- Toggler isoleringsklemmene.
- Setter restore signalet og dermed ødelegger dataen i scan vippene.

Derfor er det ønskelig at alle signalene som kommer ut av spenningskontrolleren kan gates eller multiplekses når man er i test modus. Minimum krav er at restore og isoleringssignalene blir tvunget inaktivt under test. En annen løsning er å tilby direkte kontroll av disse signalene fra eksterne pinner når krets er i test modus.

Det er også ønskelig å kunne stenge spenningen til alle de blokker som ikke testes på. Alle vippene i scankjeden kan, ved scantest, toggle på hver klokkeflanke, noe som fører til svært stor svitsjeaktivitet. Dette kan igjen føre til et dynamisk effektforbruk som er så stort at chipen ”brenner opp”.

2.2.7 Arkitektoniske spørsmål ved Power Gating

Innføring av Power Gating fører med seg spenningstap og redusert ytelse. Dette er en kostnad som er viktig å ta i betraktning når arkitekturen til en chip lages. Bruk av flere sove-transistorer i kaskade kan føre til et uakseptabelt høyt spenningsfall og ytelsestap. Det er derfor anbefalt at man i stedet for en hierarkisk strukturering av sovetransistorene benytter seg av en flat struktur. Skulle det likevel bli nødvendig med en slik struktur anbefales det å begrense seg til maksimalt to nivåer med sove-transistorer [1].



Figur 16 - Power Gating av flere domener

Et problem kan oppstå ved Power Gating av flere domener. La oss ta situasjonen i Figur 16 som eksempel. Når alle tre domenene er på er det ikke problematisk. Men hva skjer når vi skrur av domene B? Utgangen fra A til C vil da bli feil, ettersom den går gjennom B. Løsningen vil da bli å rute signalet en annen vei. Men hadde det for eksempel vært en forutsetning at C måtte være av for at B skulle kunne skrus av, hadde det ikke vært nødvendig. Disse hensynene inngår i Power Routing reglene, og kan ende opp med å bli ganske kompliserte. Her tilbyr igjen UPF konsise måter for at disse reglene skal bli tatt hensyn til av implementeringsverktøyet, simuleringsverktøy, synteseverktøy og utleggsverktøy.

2.3 UPF

UPF(The Unified Power Format) er en HDL standard fra Accellera, som fungerer som et tillegg til tradisjonelle HDL som VHDL eller Verilog. Det muliggjør design med hensyn på effektforbruk tidlig i designprosessen, på RTL(Register Transfer Level) nivå eller tidligere. UPF beskriver bare spenningsdistribusjonen, genererer spenningsnett og spenningsdistribusjonsobjekter. Det genererer ingen signaler. Figur 17 illustrerer at UPF benyttes i alle steg av designet, fra RTL til layout. Cadence har utviklet en tilsvarende standard med navn CPF(Common Power Format). Følgende punkter viser mulighetene ved bruk av UPF.

UPF kan generere

- Spenningsnett
- Spenningsporter
- Spenningsbrytere
- Isoleringsceller
- Nivåkonverter

UPF kan kople spenningsnett til

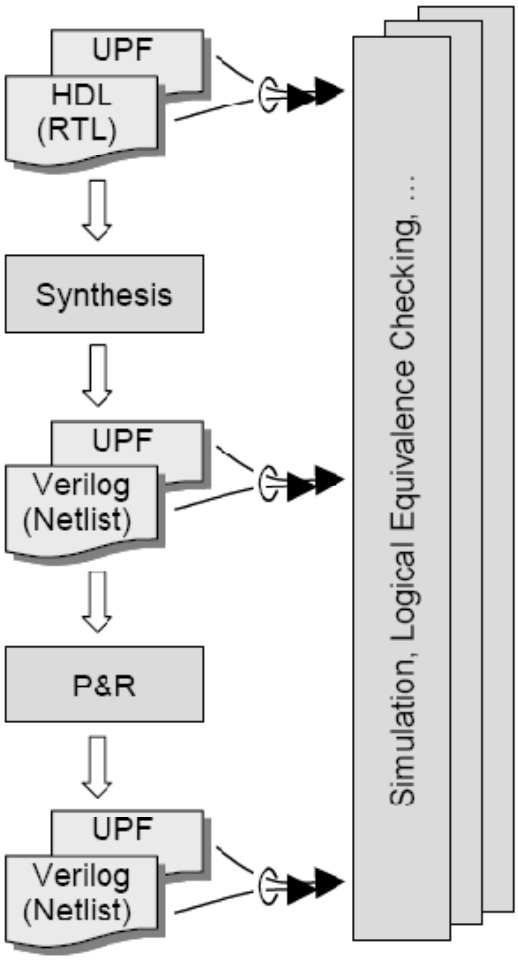
- Spenningsporter
- Spenningsbrytere
- Isoleringsceller
- Nivåkonvertere
- Bevarende vipper
- Standard cellelogikk

UPF kan konvertere vipper til bevarende vipper

UPF kan kople kontrollsignaler fra designet til

- Spenningsbrytere
- Isoleringsceller
- Bevarende vipper

For kommandoer for realisering av overnevnte henvises det til Vedlegg 5, UPF Quick Reference Card



Figur 17 - UPF flow[16]

2.4 ZPU

For å få laget en velfungerende soverutine for en mikrokontroller er det nødvendig med en mikroprosessor beskrevet i HDL. Det er i denne masteroppgaven valgt å benytte en open-source mikroprosessor med navn Zylin CPU, eller ZPU. ZPU er beskrevet i VHDL, og er i skrivende stund verdens minste 32 bits CPU med GCC verktøykjede. GCC støtten innebærer at flere programmeringsspråk kan benyttes for softwareprogrammering.

ZPU er utviklet av Øyvind Harboe ved Zylin AS, og har en BSD lisens for HDL filene. Dette betyr at den fritt kan videreutvikles og videredistribueres så lenge den opprinnelige lisensbeskrivelsen og copyrightbeskrivelsen ivaretas. Man har heller ikke lov til å benytte navn på opprinnelig utvikler eller bidragsytere for å promotere sin egen videreutvikling.

Prosjektet til ZPU består blant annet av to 32 bits mikroprosessorer, en IO enhet og en 64 bits timer. Det inneholder også noen ferdige simuleringsfiler, blant annet en *hello world* simulering. Siden ZPU ikke inneholder en komplett UART skriver disse simuleringene diverse data til en loggfil. ZPU kommer i to versjoner, en standard mikroprosessor, og en liten. Standard versjon har støtte for flere funksjoner enn den lille. Arealmessig er den ca 18x større.

Prosjektmappen til ZPU inneholder også et script som lar brukeren generere en VHDL minnefil fra sin softwarekode. Kort forklart kompilerer man koden sin med ZPUs verktøykjede. Dette vil gi en *Executable and Linking Format*(elf) fil, som igjen brukes til å generere en binær(bin) fil. Deretter bruker man det medfølgende scriptet for å generere minneblokkene. Disse kopieres, og settes inn i en av de eksisterende minnefilene. Man har da en VHDL fil som er til stor hjelp ved simulering i for eksempel Modelsim. Komplett fremgangsmåte for dette er beskrevet i *zpu_arch.html*, vedlagt på CD.

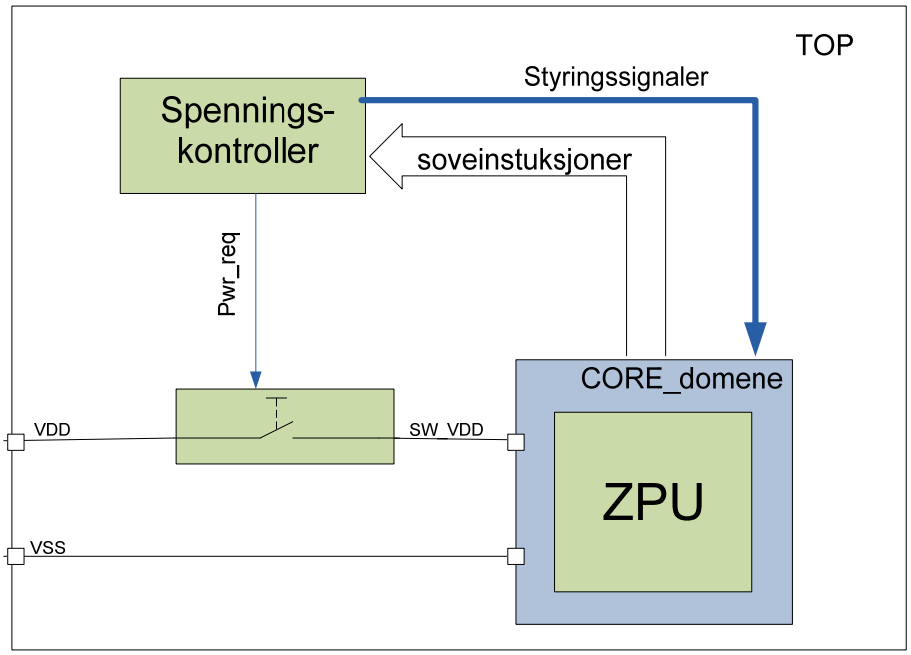
3. Implementering av sovemodi i ZPU

Vi vil i dette kapittelet se på de forskjellige implementeringer som er gjort for å muliggjøre bruk av sovemodus for ZPU, realisert ved hjelp av Power Gating. Med sovemodus menes en tilstand hvor spenningstilførselen til logikken reduseres eller tas bort, slik at lekkasjestrømmen reduseres. Delkapitlene tar for seg implementering av forskjellige funksjoner som er funnet nødvendig eller praktisk for bruk av sovemodi. De er bygget opp på en slik måte at vi først ser på generelle betraktninger rundt implementeringen, deretter de endringene som er gjort i hardwarestrukturen, mens vi til sist ser på hvilken jobb software må utføre for å muliggjøre disse funksjonene. Hendelsesforløpet som inntreffer når soving iverksettes er at klokken først stoppes. Deretter isoleres utgangene til ZPU for å unngå effektforbruk beskrevet i 2.2.2. Registrene som er nødvendig å ta vare på vil deretter bevares, og de resterende vil resettes. Når dette er utført vil spenningen til den virtuelle spenningsforsyningen bli skrudd av, og ZPU vil være i sovemodus. Ved oppvåkning vil dette reverseres.

Kapittel 3.1 ser på generell oppbygning av spenningskontrolleren, slik den er designet for å kunne sette ZPU i sovemodus. Denne vil få noen tillegg i de etterfølgende kapitler, for å støtte flere funksjoner. Kapittel 3.2 tar for seg tidsbasert soving. Dette vil si at ZPU settes til å sove, og en timer sørger for at spenningskontrolleren vekker den opp igjen etter en forhåndsbestemt tid. Kapittel 3.3 ser på hvordan tilstanden til ZPU bevares ved soving, samt isolering av utangene. Implementeringen i dette kapittelet er gjort på en slik måte at man kan lage en FPGA prototyp av mikrokontrolleren. Kapittel 3.4 vil vise hvordan forskjellige komponenter i en mikrokontroller kan benyttes til å initiere vekking av ZPU. I kapittel 3.5 ser vi på en implementering av flere sovemodi, mens vi i 3.6 ser på hvordan UPF er benyttet for å konfigurere egenskaper forbundet med soving. 3.7 tar for seg sovemodi for en mikrokontroller og sovestrategier for maksimal besparelse av lekkasjestrøm. Kapittel 3.8 ser på generelle endringer som er gjort, deriblant klokkegating.

3.1 Spenningskontroller

For å ha mulighet til å benytte Power Gating til å skru av ZPU er det nødvendig med en spenningskontroller. Figuren under illustrerer spenningskontrollerens funksjon i systemet. De grønne feltene indikerer logikk, mens blå felt indikerer spenningsdomener.



Figur 18 – Spenningskontroller

Hensikten med spenningskontrolleren er å styre de kontrollsignaler som sørger for gating av klokke, isolering av signaler, bevaring av registre og Power Gating. Spenningskontrolleren er laget som en egen blokk, på lik linje med for eksempel prosessoren, som plasseres i det øverste spenningsdomenet. Den vil dermed alltid ha driftsspenning. Spenningskontrolleren er oppbygd som en tilstandsmaskin med inaktiv, sove og oppvåknings tilstand. Ved inaktiv tilstand gjør den ingenting foruten å vente på at ZPU skal gi beskjed om at den vil sove. Den sørger da for initiering av sekvensen vist i Figur 15, side 20. Deretter skrur den av spenningen(*SW_VDD eller VVDD*) til ZPU, før den går over i sovetilstand. Merk at sovetilstand her indikerer at ZPU sover, mens spenningskontrolleren fortsatt vil være aktiv. Inaktiv tilstand i spenningskontrolleren indikerer at ZPU er våken. Under soving vil spenningskontrolleren vente på ekstern stimuli som tilsier at ZPU skal vekkes, for da å gå over i våknetilstand. Den vil da klargjøre ZPU for normal operasjon ved å reversere sekvensen gitt i Figur 15. Etter at dette er gjort, og ZPU har fått tilbake driftspenningen, vil den gå tilbake til inaktiv tilstand.

3.1.1 Hardware

For at spenningskontrolleren skal kunne kommunisere med omkringliggende elementer må den legges inn i IO enheten til ZPU, og signalene fra den må rutes dit de er ment. Den må også gis en egen adresse slik at skriving til den blir mulig. Adressen som er gitt spenningskontrolleren er 0x80A2000. Adressedelen 0x80A indikerer at det er en ekstern komponent (sett fra ZPU, ikke mikrokontrolleren), mens 0x2000 er IO adressen. De relevante data som skrives til spenningskontrolleren er foreløpig to bit. Verdien på disse bit forteller hvilken sovemodi som skal aktiveres. Implementeringen i dette kapittelet støtter kun en sovemodus, slik at man kun kan skrive verdien 0x01. Andre verdier vil ikke gi noen respons. Når spenningskontrolleren registrerer kommandoen for soving vil den umiddelbart iverksette sovesekvensen.

Som nevnt i 2.2.4 er det ønskelig at utgangssignalene fra spenningskontrolleren ikke endres samtidig, men venter til det forrige signalet er satt før neste aktiveres. Måten dette er implementert på benytter seg av den egenskapen at et signal kun oppdateres når en prosess er ferdig gjennomført.

```
if (clk'event and clk = '1') then
        if (sleep_requested = '1') then
                clock_gater <= '1';
                if clock_gater = '1' then
                        isol <= '1';
                        if isol = '1' then
                                retain_n <= '0';
                                if retain_n = '0' then
                                        reset_n <= '0';
                                        if reset_n = '0' then
                                                pwr_req <= '0';
                                        end if;
```

Siden vi tester på stigende klokkeflanke vil vi oppnå en forsinkelse på en klokkeperiode mellom hvert signal. Signalet vil heller ikke bli satt hvis ikke det forrige signalet i prosessen er satt høyt.

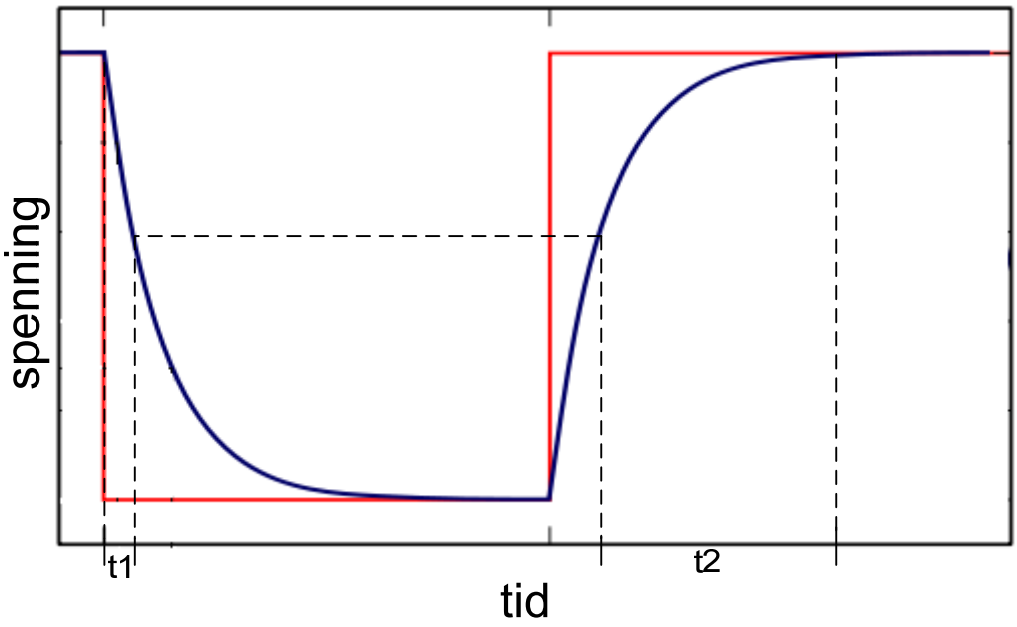
En av de viktigste funksjonene i spenningskontrolleren vil være den som sørger for at ikke oppvåkningssekvensen initieres før ZPU har oppnådd full driftspenning. Dette kan gjøres på to måter. Den ene er ved bruk av UPF og syntеseverktøy. Denne metoden kommer vi tilbake til i kapittel 3.6. Den andre metoden er ved bruk av timer. Vi er her avhengig av å vite oppladningstiden til VVDD. Når spenningskontrolleren får beskjed om å starte oppvåkning skrur den på spenningen for så å telle opp til en verdi som tilsvarer oppladningstiden. Vi er

da sikker på at spenningen er på, og registrene vil ikke bli ødelagt av å initiere oppvåkningssekvensen. Pseudokoden under illustrerer prinsippet.

```
Process (clk)
  If rising_edge(clk) then
        If wakeup requested then
              Power on <= ”1”;
              Timer <= Timer + ”1”;
              If Timer > oppladningstid then
                    Driftsspenning <= ”1”;
                    If Driftsspenning = “1” then
                          Start oppvåkningssekvens
```

Oppladningstiden spenningskontrolleren implementeres med er hentet fra simulert oppladningstid beskrevet i kapittel 4.8. For mer utfyllende informasjon om denne oppvåkningsforsinkelsen henvises det til koden for spenningskontrolleren i vedlegg 2.

Spenningskontrolleren er designet slik at oppvåkning har prioritet. Det vil si at det kan gis beskjed om at oppvåkning skal skje når spenningskontrolleren holder på å sette ZPU i sovemodus. Vi kan for eksempel tenke oss at trafikk på UART tilsier at ZPU må våkne opp mens den er på vei til sovemodus. Spenningskontrolleren vet da hvor langt i sekvensen den er kommet, og kan reversere de signalene som er endret. Muligheten til å gjøre dette fører til at vi får en mindre tidskostnad i tilfellet beskrevet over. Optimalt sett skal ikke ZPU vekkes opp før etter en viss tid. Dette tidspunktet er en funksjon av hvor mye energi som kreves for å iverksette soving og hvor mye lekkasjestrøm man sparer per tidsenhet. Det er derfor brukers ansvar å påse at aktivitetsprofilen til ZPU tilsier at sovetiden forventes å være lang nok til at det vil være en effektbesparelse. Mer om dette i 4.9, ”Bruk av sovemodus”. Metoden beskrevet har to fordeler framfor alternativet, som er å vente til ZPU har gått i sovemodus, for så å vekke den opp igjen. Den første, og mest opplagte, er tidsbesparelsen. Den andre er energibesparelse ved å ikke behøve å lade opp VVDD. Vi vil se nærmere på disse besparelsene i kapittel 4.1. Det er også vurdert metoder hvor spenningskontrolleren ”vet” hvor mye ladning som til enhver tid er på spenningsskinnene ut i fra hvor mye som er ladet ut eller opp. Grunnet den ulineære oppførselen til spenning over kapasitanser(se Figur 19)er dette funnet upraktisk. Dette medfører at når sovesekvensen har kommet til det punkt at spenningsskinnene er slått av må man vente den tiden det tar å lade de opp helt fra nullnivå. Dette vil føre til at oppvåkningstiden blir den samme som i vanlige tilfeller, men det vil være en effektbesparelse som følge av at spenningsskinnene ikke behøves å lades helt opp. Denne besparelsen vil være en funksjon av hvor lenge VVDD har blitt ladet ut.



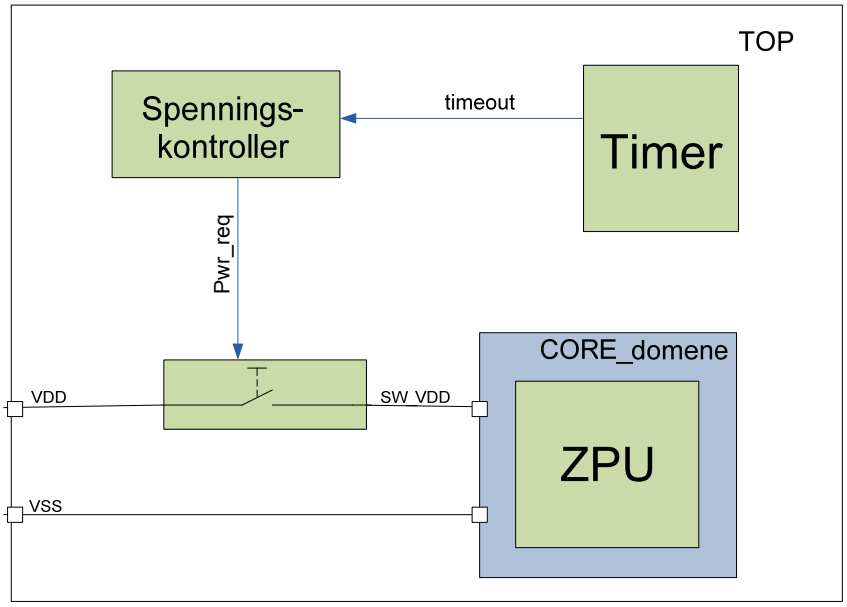
Figur 19 - Spenning over kapasitans

Som figuren over indikerer vil oppladningstiden være mye lengre enn utladningstiden i tilfellet hvor VVDD skrus av, men settes på igjen ved slutten av t_1. Grunnet denne oppførselen er det lite hensiktsmessig å implementere tellere som indikerer hva spenningen på VVDD er til enhver tid.

3.1.2 Software

For at spenningskontrolleren skal kunne sette ZPU i sovemodus er det nødvendig at den får beskjed om å gjøre det. Slik implementeringen i denne rapporten er gjort, bestemmes dette av brukeren ved hjelp av software på ZPU. Ved å skrive data til spenningskontrollerens adresse, 0x80A2000, kan brukeren aktivere sovemodus, og bestemme forskjellige egenskaper som skal gjelde for denne. Disse egenskapene vil bli nærmere diskutert i de enkelte delkapitlene.

3.2 Tidsbasert soving



Figur 20 - Tidsbasert soving

For å kunne sette ZPU til å sove er det nødvendig med en form for stimuli som gir beskjed om at spenningskontrolleren skal vekke den opp igjen. Dette delkapittelet tar for seg implementering av tidsbasert soving, hvor brukeren bestemmer hvor lenge ZPU skal sove. En timer sørger for at spenningskontrolleren får beskjed om å vekke ZPU etter at denne tiden er oversteget. Dette kapittelet ser bort fra det faktum at registerverdiene vil bli ødelagte når spenningen tas bort fra blokken. Det fokuserer heller bare på hvilke endringer som er gjort i timeren som fulgte med ZPU, slik at den kan benyttes til å gi beskjed om at oppvåkning skal iverksettes. Figur 20 viser at timeren er definert slik at den gir beskjed direkte til spenningskontrolleren når sovetiden er overskredet.

3.2.1 Hardware

For tidsbasert soving er det nødvendig at timeren gjør en jobb. For å ha mulighet til å utføre denne jobben er det nødvendig at timeren ligger i et spenningsdomene som kan være på når ZPU sover. Oppgaven til timeren når ZPU sover vil gå ut på å sammenlikne nåværende telleverdi med et brukerdefinert oppvåkningstidspunkt. Oppvåkningstidspunktet beregnes ved at software leser verdien på telleren, for så å addere en gitt sovetid. Summen vil da være oppvåkningstidspunktet. Sekvensen på denne operasjon vil være følgende:

- Sample verdien av telleren
- Software beregner et oppvåkningstidspunkt ut i fra den samplete verdien
- Legge inn mottatt oppvåkningstidspunkt inn i et sammenlikningsregister lokalt i timeren.
- Når verdien på telleren overgår verdien på sammenlikningsregisteret skal timeren gi beskjed til spenningskontrolleren om at sovetiden er over.

For at timeren skal kunne motta oppvåkningstidspunktet er det nødvendig å muliggjøre skriving til den, noe den ikke har fra før. Som nevnt i 2.4 har timeren som følger med ZPU opprinnelig 64 bit. Hva gjelder adressering her er disse fra før av delt i to. De 32 mest signifikante bit(MSB) har adressen 0x80A018, mens de 32 minst signifikante bit(LSB) har adressen 0x80A014. For å kunne gi timeren oppvåkningstidspunktet er den redusert til å ha 32 bit, ved å gjøre de tidligere 32 MSB om til et sammenlikningsregister som man kan skrive til fra software. Vi kan da lese verdien på timeren fra adressen 0x80A014, mens vi kan skrive oppvåkningstidspunket til adressen 0x80A018.

3.2.2 Software

For at ZPU skal sove kreves det som nevnt tidligere at software gir instruksjon om å sove.
Som første forsøk skal denne softwaren gjøre følgende oppgave:

- Printe ut data over UART.
- Lese timeren
- Ut i fra timerverdien, og en brukerdefinert sovetid skal det bestemmes en timerverdi for oppvåkning
- Laste denne timerverdien tilbake til sammenlikningsregisteret i timeren.
- Gi spenningskontrolleren beskjed om at ZPU skal sove.

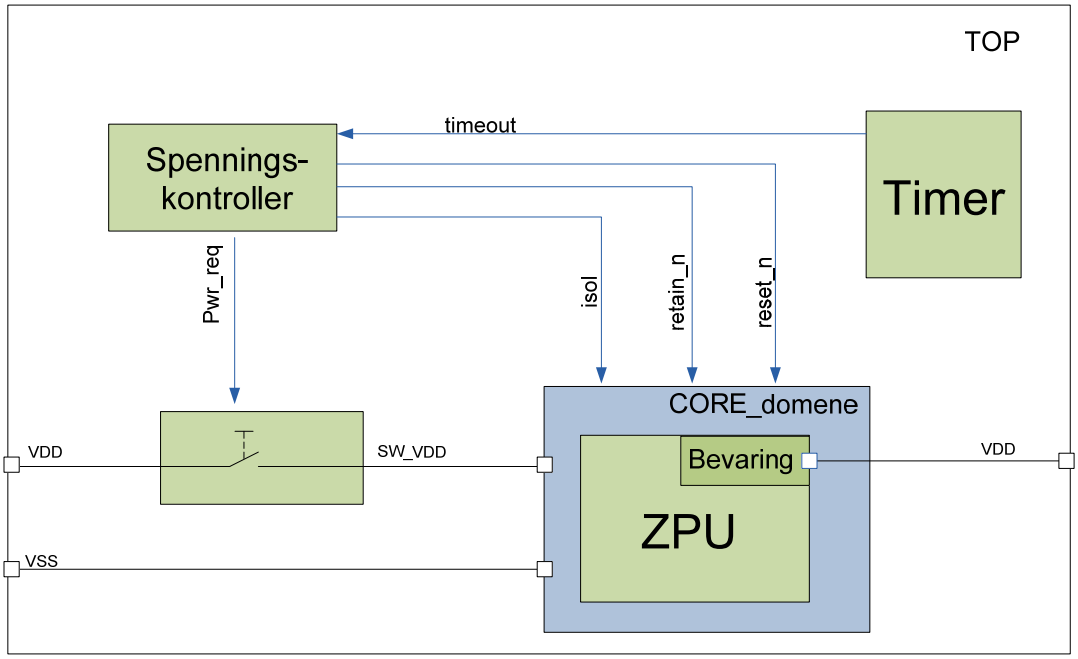
Når så timeren registrerer en verdi som er større enn sammenlikningsregisteret vil den gi timeout, og spenningskontrolleren vil vekke ZPU opp igjen. Sekvensen gjentas så.

Softwaren som ZPU er programmert med er laget med programmeringsspråket C++. Eksempelet under illustrerer hvordan software setter ZPU i tidsbasert sovemodus.

```
address = 0x080A0014;        //adresse til timer
*(address) = 0x02;           //sampler timer

address = 0x080A0014;        //adresse til timer
timerBuffer = *(address);    //leser timer

Compare = timerBuffer + sleeptime;    //beregner oppvåkningstidspunkt

address = 0x080A0018;              //adresse til sammenlikningsregisteret
*(address) = NewTimerBuffer;       //laster oppvåkningstidspunkt til timer

    address = 0x080A2000;          //adresse til spenningskontrolleren
    *(address) = 0x01;             //ber spenningskontrolleren om å iverksette soving
```

Vi aksesserer først adressen til timeren. Deretter samples timeren i henhold til databladet til ZPU(zpu_arch.html på vedlagt CD). Den samplede verdien hentes deretter ut og legges i *timerBuffer.* Vi peker så på adressen til sammenlikningsregisteret, og laster inn oppvåkningstidspunktet. Til slutt aksesserer vi spenningskontrolleren og skriver hvilken sovemodus som skal iverksettes. Full programkode finnes i Vedlegg 1.

3.3 Isolering av utganger og bevaring av registre



Figur 21 - Isolering av utganger og bevaring av registre

Når man tar bort spenningen på en blokk vil registrene i blokken miste informasjonen som ligger lagret i dem. Dette medfører at kritisk innhold i registrene til ZPU må lagres bort før soving blir initiert. Deretter må innholdet lastes tilbake ved oppvåkning slik at prosessoren kan fortsette der den var før soving. Som nevnt i kapittel 2.2.3 er det flere metoder for bevaring av tilstanden. Ettersom registerverdiene ikke er tilgjengelig på minnebussen krever softwareløsningen at hardwarestrukturen endres på en slik måte at minnet blir tilgjengelig her. Dette kan medføre en betydelig arealkostnad, også med tanke med på at det krever en ekstra prosessor for styring av bevaringen. Den største fordelen med bruk av *scankjeder* er at de har minimal effekt på arealkostnaden. Som det vil bli vist i 3.3.1 er det imidlertid kun nødvendig å bevare to registre for å gjenopprette tilstanden. Kostnadene ved å benytte scankjeder vil derfor være større enn besparelsen i forhold til bruk av bevaringsregistre. På grunnlag av de fordeler og ulemper de forskjellige metodene medfører er det valgt å benytte bevaringsregistre i denne implementeringen.

Figur 21 illustrerer de signalene som spenningskontrolleren behøver for å realisere isolering og bevaring. Ettersom UPF fortsatt ikke er implementert er isolering og bevaring nødt å skje inne i ZPU, noe som realiseres ved å endre VHDL koden. Ved bruk av UPF vil dette bli implementert automatisk, og trenger dermed ikke tas hensyn til i ZPU. Resetting av registre er nødvendig at gjøres inne i ZPU, uavhengig av hvorvidt UPF benyttes. Dette må gjøres for at de registre som ikke har kritiske verdier før soving våkner opp med initialverdien deres. Et eksempel på dette kan være tilstanden ZPU er i. For at alle registrene skal gjenopprettes er

det nødvendig å vekke ZPU opp i en bestemt tilstand. Hvis denne tilstanden er feil vil ikke ZPU klare å gjenopprette alle registrene, og vi vil få feil med programkjøringen. Derfor sørger reset signalet for at den blir satt i en resynkroniserende tilstand ved oppvåkning.

Denne implementeringen krever ingen egen jobb fra software, og har dermed ingen softwarekapittel.

3.3.1 Hardware

For å finne ut hvilke registre som er kritiske for korrekt gjenopprettning er det benyttet en analytisk metode, hvor det er gjort endringer i koden til ZPU. Ett og ett register settes til en resetverdi når ZPU går i sovemodus, for så å se om den klarer å gjenoppta kjøringen av programmet etter oppvåkning. Hvis den klarer dette antas det at dette ikke er et kritisk register, og dermed ikke er nødvendig å bevare. Hvis den ikke klarer å gjenoppta kjøringen er det antatt at verdien av dette registeret må bevares. Dette har ført til konklusjonen at de registre med kritisk innhold er *programtelleren*(pc) og *stakkpekeren*(sp). Videre analyse av koden bekrefter at alle andre registre blir gjenopprettet fra verdien til enten pc eller sp, såfremt man velger å våkne opp i tilstanden *state_resync*. Dette skjer ved at prosessoren går gjennom flere tilstander før den kommer til den utførende tilstanden. Hvis registrene har feil verdi vil det her bli utført feil operasjoner. Simuleringer bekrefter at alle registre har blitt gjenopprettet slik at videre operasjoner blir vellykket. Alle registre som ikke er bevart resettes ved oppvåkning. Dette gjør at ZPU kan våkne opp i en resatt tilstand, for så å benytte den bevarte informasjonen til å komme til riktig tilstand. Det er viktig at kun de registre som ikke bevares blir resatt, da verdien til bevaringsregistrene vil bli ødelagte hvis de resettes.

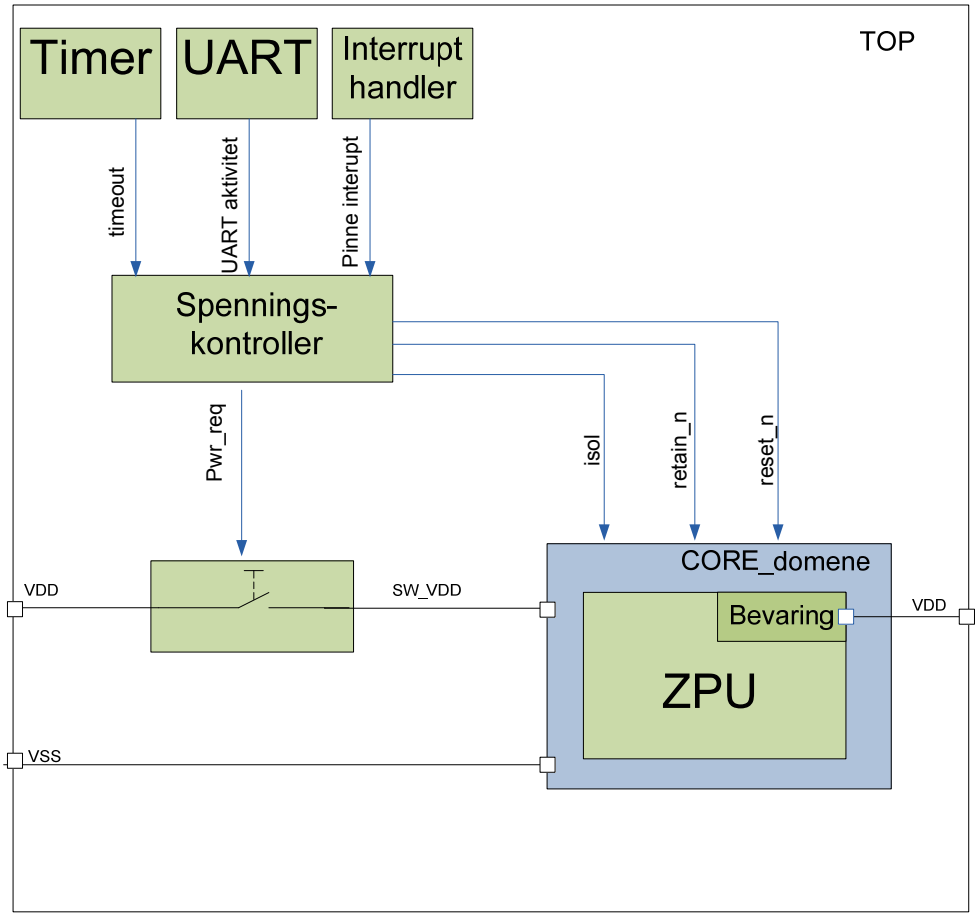
Måten selve bevaringen er utført på er ganske enkel. Et bevarende register er sammensatt av to register, et aktivt og et skyggeregister for selve bevaringen. Grunnen til at det er mulig å bevare registerverdien er at skyggeregisteret har en annen spenningskilde enn det aktive registeret. Dette er etterliknet ved å lage nye ”skyggeregistre” for de registrene som trengtes å bevares. Disse mister ikke verdien sin når ZPU blir satt til å sove.

ZPU opererer med aktiv høye signaler. Isolering er derfor realisert slik at alle utganger settes lav når signalet for isolering er aktivt. De endringene som er gjort i ZPU er illustrert som pseudokode under.

```
Process(clk, isol, reset_n, retain_n, Power)

  If (reset_n = '0') then
        Reset registre som ikke skal bevares
  end if;

  if FPGA_usage then

                If save = '1' then
                      Bevar registre
                elsif restore = '1' then
                      Gjenopprett registre.
                end if;

                if (isol = '1') then
                      Sett alle utganger til verdien "0"
                end if;

                 if (Power = '0') then
                      Sett alle registre til verdien "X"
                end if;
    end if;
```

Nettopp det at isolering skjer ved å sette utgangene lav er også et argument for å benytte toppsvitsjing for å skru av spenningen på spenningsskinnene. Hvis man benytter bunnsvitsjing her er ikke VSS tilgjengelig når ZPU sover. Man må i så fall gi isoleringsklemmene en egen spenningstilførsel, noe som fører til ekstra arealkostnad. Den siste testen i kodesnutten over sjekker om ZPU har spenning nok til å holde på registerverdiene. Signalet er hentet fra spenningskontrolleren. Hvis spenningen er for lav ødelegges alle registrene, noe som etterlikner oppførselen i et design hvor spenningen forsvinner. Registerverdien på bevaringsregistrene vil ikke bli ødelagt når spenningen forsvinner da disse har egen spenningstilførsel.

3.4 Kontroll av oppvåkningsstimuli



Figur 22 - Forskjellige oppvåkningsstimuli

En mikrokontroller må ha mulighet til å bli vekket opp av flere enheter, som illustrert i Figur 22. Eksempler på slike enheter kan være timer, UART(Universal Asynchronous Receive and Transmit Unit) og interrupt fra ekstern pinne. Det er flere eksempler på scenario hvor det er nødvendig at hver enkel enhet må ha mulighet til å vekke ZPU. La oss først tenke oss en sensor som skal gjøre en måling hvert minutt. Det vil da være nødvendig med en timer som gir beskjed om å vekke den hvert minutt slik at den kan foreta målingene. Men det er også tenkelig at mikrokontrolleren må vente på data fra en ekstern sensor. Det vil da være optimalt om mikrokontrolleren kan sove helt til data er tilgjengelig på UART. Til slutt kan vi tenke oss at den samme sensoren kun skal vise målingene på en liten skjerm. For å spare strøm kan man da tenke seg at systemet skrus av etter en gitt tid, og at det kan vekkes opp ved at brukeren aktiverer sensoren igjen ved å trykke på en knapp. Denne delen tar for seg de konfigureringer som ble utført for å få denne funksjonaliteten, samt implementering av UART.

3.4.1 Hardware

En faktor som er viktig i forhold til produkter som retter seg mot forbrukere er brukervennlighet. For å øke brukervennligheten til mikrokontrolleren vil det være hensiktsmessig å la brukeren ha full kontroll over hvilke komponenter som skal kunne vekke ZPU. Dette er realisert ved å utvide datavektoren inn til spenningskontrolleren. Den er utvidet fra to bit, som opprinnelig implementert i kapittel 3.1, til fem bit som vist i Tabell 1. Tanken er her at de tre mest signifikante bit skal tilhøre hver sin komponent. Dette designet støtter oppvåkningsstimuli fra timer, UART samt interrupt fra ekstern pinne. Kombinasjonen av disse bit settes ved å velge en soveprofil, noe vi kommer mer tilbake til i 3.4.2. Når de først er satt fungerer det slik at spenningskontrolleren lar mikrokontrolleren våkne opp ved ekstern stimuli kun hvis det tilhørende utløserbit er satt. Den vet da at det er brukerens intensjon at mikrokontrolleren kan vekkes opp av denne stimulus. Det er også lagt inn en funksjon som gir feilmelding i simuleringen hvis man ikke har valgt noen komponenter som skal utløse oppvåkning, da dette ville føre til ”evig” søvn.

ZPU innholder ikke UART, så for å designe et system med oppvåkning ved aktivitet fra UART må det implementeres en. Valget faller her på miniUART, utviklet av Oividu Lupas, fra opencores.org. For at denne skal kunne vekke opp ZPU når den sover er det nødvendig med et kontrollsignal som forteller spenningskontrolleren når det er aktivitet på UART. Hvis brukerkonfigurasjonen tilsier det skal ZPU vekkes opp til aktiv tilstand når dette signalet settes, og kan gå videre til å lese dataen som er tilgjengelig. Dette er realisert ved at spenningskontrolleren leser interruptsignalet som UART bruker til å fortelle prosessoren at data er tilgjengelig. Dette krever ingen endringer i arkitekturen til UART.

Støtte for interrupt fra ekstern pinne er kun lagt inn som illustrasjon av prinsippet. Det er lagt inn et signal i testbenken som illustrerer hvorvidt det gis noe ekstern stimuli. Hvis det er tilfellet, og konfigurasjonen tilsier det, vekkes ZPU opp for å kunne utføre ønsket instruksjon.

3.4.2 Software

For å realisere valg av oppvåkningsutløsere er det valgt å utvide datavektoren som skrives til spenningskontrolleren til 5 bit, som nevnt i 3.4.1.

Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pinne Interrupt	UART	Timer	Sleepmode	Sleepmode

Tabell 1 – Data til spenningskontroller

Tabell 1 viser datastrengen som skrives til spenningskontrolleren. Vi ser også hvilken tilhørighet de forskjellige bit har med komponentene i mikrokontrolleren. Bit 0 og 1

bestemmer hvilken sovemodus som skal aktiveres, noe som bestemmes av brukeren. Bit 2 forteller om timeren skal benyttes som utløser for oppvåkning, bit 3 velger om UART skal benyttes og bit 4 forteller hvorvidt mikrokontrolleren skal våkne opp ved interrupt fra ekstern pinne. Bit 4 ned til 2 settes avhengig av hvilken soveprofil man velger. Tabell 2 viser de profiler som er tilgjengelig i denne implementeringen. De komponenter som er aktive i en profil fungerer som oppvåkningstriggere.

Soveprofil #	***ZPU***	***Timer***	***UART***	***Interrupt handler***
1	Off	On	On	On
2	Off	On	On	Off
3	Off	On	Off	On
4	Off	On	Off	Off
5	Off	Off	On	On
6	Off	Off	On	Off
7	Off	Off	Off	On
8	~~Off~~	~~Off~~	~~Off~~	~~Off~~

Tabell 2 – Soveprofiler

Vi ser at profil #8 ikke har noen oppvåkningstriggere. Denne vil derfor ikke kunne våkne opp, så den er en ugyldig profil. De forskjellige soveprofilene blir aktivert ved at brukeren gir kommandoen

sleep(”soveprofil”, ”sovetid”)

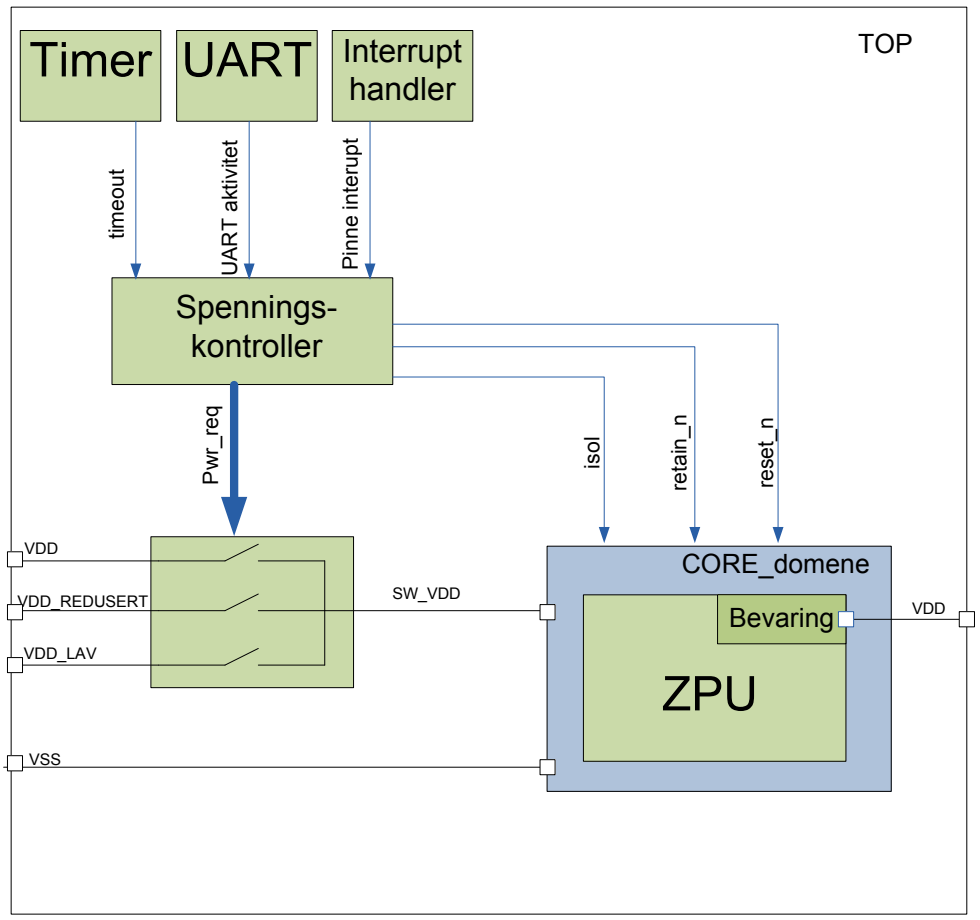
hvis man ønsker å gå i *SLEEP* modus. Man kan på samme måte benytte *REST* eller *HIBERNATE,* noe vi kommer tilbake til i 3.5. Bruker man en profil som ikke benytter seg av tidsbasert soving kan man la være å skrive inn sovetiden. Koden som sørger for at spenningskontrolleren får den datastrengen som er beskrevet i Tabell 1 er vist under.

```
void sleep (int sleepmode, int sleeptime)
{

sleepdata = 0x01;          //verdi for SLEEP. 0x02 for REST, eller 0x03 for HIBERNATE

if (sleepprofile==(1 || 2 || 3 || 4))
        { sleepdata=sleepdata+0x04; }          //setter bit 2
if (sleepprofile==(1 || 2 || 5 || 6))
        { sleepdata=sleepdata+0x08; }          //setter bit 3
if (sleepprofile==(1 || 3 || 5 || 7))
        { sleepdata=sleepdata+0x10; }          //setter bit 4
}
```

Koden regner ut en dataverdi avhengig av hvilken soveprofil vi velger. Utregningen gjøres på en slik måte at verdiendringen som følge av addisjonen kun vil ha påvirkning på et enkelt bit, jamfør Tabell 1. Spenningskontrolleren er da i stand til å lese denne bitstringen og avgjøre videre handling. Legg merke til at dette kodeeksempelet kun illustrerer kalkulering av data som skrives til spenningskontrolleren. For å unngå unødvendig svitsjing er det også lagt inn en test som gjør at hvis ikke tidsbasert soving er benyttet hopper den også over sampling av timer og skriving av oppvåkningstidspunkt. Eksempel over er noe forenklet, og vil derfor ikke gi korrekt resultat. Se programkode i Vedlegg 1 for fungerende kode.

3.5 Implementering av flere sovemodi

Som vi husker fra teorikapittelet kan det være lønnsomt å benytte flere sovemodi, slik at man også kan sette prosessoren til å sove når aktivitetsprofilen ikke rettferdiggjør dypeste søvn[2][14][10].



Figur 23 - Tre sovemodi

I henhold til denne teorien er bryteren til spenningsskinnene utvidet slik at den kan påføre tre forskjellige spenninger. For utenom den aktive spenningen har den tre spenninger for soving, som illustrert i Figur 23. Disse har følgende egenskaper

- Spenning helt av. Alt innhold mistes, og bevaring er nødvendig.
- Spenning under terskelspenning. Alt innhold mistes, og bevaring er nødvendig.
- Spenning over terskelspenning. Alt innhold bevares.

Alle sovemodiene benytter seg av isolering, samt at klokken gates bort(forklart i kapittel 3.8). Det gjøres oppmerksom på at den sistnevnte tilstanden hvor registerinnholdet bevares forutsetter at man benytter SRAM, eller annet minne som ikke trenger oppfriskning, for lagring av registrene. Ved bruk av DRAM vil ikke verdien av registrene bevares i denne tilstanden, da klokken gates bort. I så fall må også signaler for bevaring og resetting legges

inn i spenningskontrolleren for denne tilstanden. Soveprofilene beskrevet i kapittel 3.4 er tilgjengelig med alle sovemodiene implementer her.

Denne delen tar for seg de endringer som er nødvendig i spenningskontrolleren for å realisere forskjellige sovemodi. Selve bryteren, eller sovetransistorcellen vil bli nærmere diskutert i kapittel 3.6 da den implementeres ved bruk av UPF.

3.5.1 Hardware

De nye sovemodiene blir implementert i spenningskontrolleren på lik måte som den opprinnelige sovetilstanden ble implementert. Data som blir skrevet til spenningskontrolleren blir analysert, og den ønskede sekvensen blir så iverksatt. Under ser vi koden som bestemmer hvilke sovemodi som skal iverksettes.

```
if (clk'event and clk = '1') then
   if we = '1' then
        if sleepdata(1 downto 0) = "01" then
                sleep_requested <= '1';
        elsif sleepdata(1 downto 0) = "10" then
                rest_requested <= '1';
        elsif sleepdata(1 downto 0) = "11" then
                hibernate_requested <= '1';
        else
                report "Invalid sleepmode selected" severity warning;
        end if;
    end if;
end if;
```

Som vi ser av koden vil spenningskontrolleren reagere når signalet *we*(write enable) er satt høyt. Dette signalet settes høyt når vi skriver til spenningskontrolleren. Den begynner da å sette sovesignaler høyt avhengig av hvilke instrukser den har mottatt. Sekvensene som så blir iverksatt er individuell for hver av de tre tilstandene. Denne sekvensen er blant annet avhengig av hvilken spenning som påføres ZPU. Tilstanden *REST* er i motsetning til *SLEEP* og *HIBERNATE* bevarende og vi har derfor ikke noe behov for å ha med signaler for bevaring og resetting i den tilhørende sekvensen. Etter at sekvensen er fullført sender spenningskontrolleren er forespørsel til bryteren om å skifte til ønsket spenning. Sekvensen for oppvåkning vil være uavhengig av hvilken tilstand ZPU har vært i, og trenger ikke endres, noe man kan se ved å studere koden i Vedlegg 1. Denne sekvensen ”justerer” seg selv til å endre de signaler som ble endret når soving ble initiert, og vil derfor ha en oppførsel som avhenger av hvilken sovemodus ZPU har vært i.

3.5.2 Software

De to nye tilstandene er realisert på lik måte som tidligere sovemodi, med eneste forskjell i data som blir skrevet. *SLEEP* oppnås ved å skrive den binære verdien ”xxx01” til spenningskontrolleren, mens *REST* og *HIBERNATE* oppnås med å skrive ”xxx02” og ”xxx03”, hvor x representerer data til valg av soveprofil. De forskjellige soveprofiler beskrevet i kapittel 3.4 er som tidligere nevnt kompatibel med alle sovemodi.

3.6 Implementering med UPF

Simuleringene fram til nå har ikke benyttet seg av UPF. Dette vil vise funksjonaliteten ved soving, og dermed egne seg til design av FPGA prototyp, men vil ikke fungere når spenningen faktisk gates bort. Selve UPF filen er en konfigurasjonsfil som forteller verktøyene hvilke hensyn som skal tas med tanke på effekt. Der bevaring og isolering til nå har vært beskrevet i kildekoden til komponentene kan vi nå la UPF ta hånd om alt dette. De tidligere simuleringer er utført i ModelSim, som ikke har støtte for UPF. Simuleringene med UPF er derfor utført med QuestaSim, et simuleringsverktøy fra Mentor Graphics med store likheter til ModelSim.

For å forenkle arbeidet er det gjort slik at samme kode som tidligere kan benyttes ved bruk av UPF. Forskjellen i kodene er at isolering, resetting og bevaring ikke skal skje i koden til ASIC designet, men automatisk håndteres av UPF. Konfigureringsfilen til ZPU er derfor utvidet med en boolsk konstant som forteller hvorvidt vi benytter oss av FPGA eller ASIC designflyt. Denne konstanten kan leses av alle instansene til ZPU, og de kan derfra avgjøre hvilke handlinger som skal utføres. ZPU tester i dette tilfellet om det er en FPGA designflyt som benyttes. Hvis ikke hopper den over bevaring, resetting og isolering, og lar UPF ta seg av det.

Det ligger to UPF konfigurasjonsfiler vedlagt denne rapporten. Den ene har kun definert spenningsdomene for ZPU, mens den andre også definerer spenningsdomener, bevaring og isoleringsstrategier for de andre blokkene. I dette kapittelet ser vi på den konfigurasjonen som kun setter ZPU til å sove. Denne finnes i Vedlegg 3.

Det første vi må gjøre ved design av UPF konfigurasjonen er å bestemme hvor i designet man skal befinne seg. I dette tilfellet er det satt til å være toppen av designet og realiseres med kommandoen *set_scope top.* Alle elementer vi heretter ønsker å henvise til er nødt til å ha med beskrivelse hvor den ligger i forhold til dette. Etter å ha definert domener og beskrevet spenningstilførsler kan vi definere brytere til domenene. Denne definisjonen inneholder informasjon om hvilke spenningstilførsler den skal benytte, samt hvilke kontrollsignaler som skal styre bryteren. Bryteren kan også implementeres med kvittering for oppvåkning. Denne kvitteringen kan settes til å opptre etter en forsinkelse, slik at man er sikker på at spenningen er kommet på. Et problem her er at det ikke er støtte for flere verdier på denne forsinkelsen, noe som er ideelt ved bruk av flere spenninger. Forsinkelsen er derfor ikke implementert med hjelp av UPF, men realiseres heller med den tidligere implementerte timeren i spenningskontrolleren.

Isolering implementeres ved at man først spesifiserer hvilket domene man ønsker å benytte isolering på. Man må også definere om man ønsker å isolere til ”1”, ”0”, eller latche til siste verdi. Som tidligere er det valgt å isolere til inaktiv verdi, ”0”. Deretter bestemmer man hvilke spenningsforsyning som skal benyttes på isoleringscellene. Det er her valgt å benytte samme spenningsforsyning som ZPU benytter, for å unngå arealkostnaden som flere

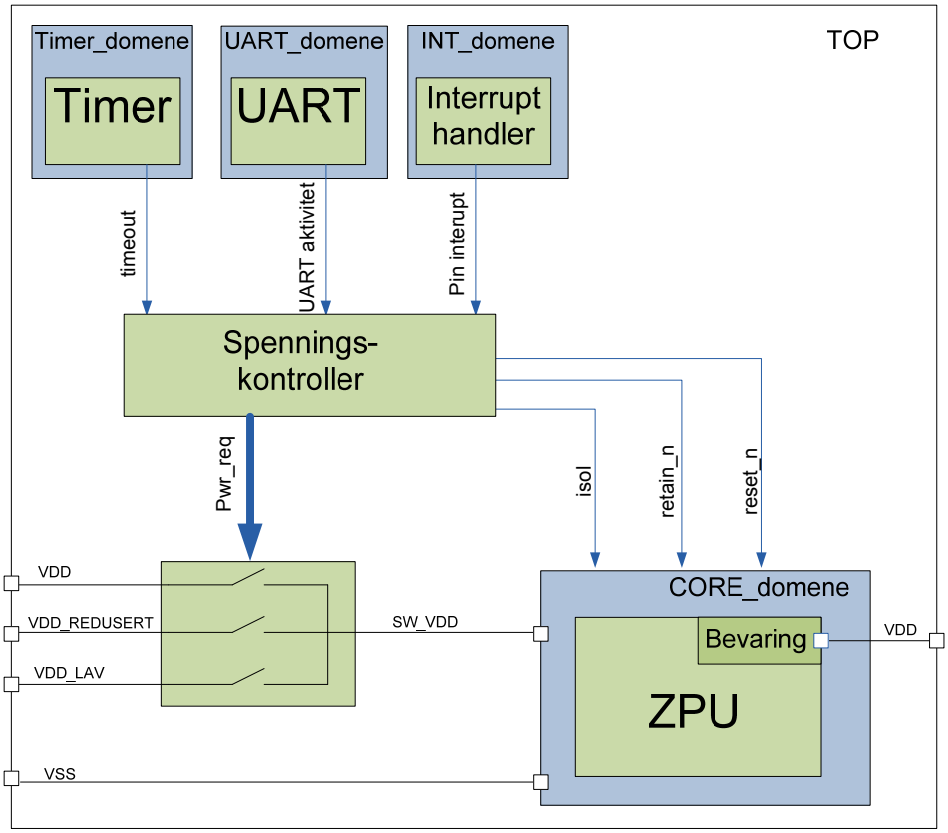
spenningsforsyninger vil medføre. Siden vi velger å isolere til verdien ”0”, samt at vi benytter toppsvitsjing er dette mulig. Det gis også valget om isolering av innganger, utganger eller begge. Isolering av innganger vil kunne medføre vanskeligheter som nevnt i 2.2.2. Det er derfor valgt å isolere utgangene. Isolering av både inn og utganger anses ikke som hensiktsmessig i dette tilfellet, da det bare vil medføre en større arealkostnad.

De bevarende registre er implementert med egen spenningstilførsel. Dette er nødvendig for at de skal kunne bevare verdiene sine når virtuell VDD går mot 0 Volt. Vi kan her også definere hva vi ønsker å bevare. Dette kan være fra registre, til hele blokker. Jamfør 3.3 er det valgt å bevare programteller og stakkpeker. Til slutt defineres hvilke signal som skal benyttes for lagring og gjenoppretning av registerverdene.

UPF har støtte for mange flere funksjoner enn de som konfigurasjonsfilen til dette prosjektet inneholder. Vedlegg 5, UPF 1.0 Quick Reference Guide, viser disse.

3.7 Sovemodi for mikrokontroller

Så langt er det kun diskutert redusering av lekkasjestrøm i prosessoren til mikrokontrolleren. Men man kan ikke se bort ifra at en mikrokontroller består av flere blokker som ikke alltid er i bruk. Hvis vi for eksempel kun benytter timeren som oppvåkningsstimuli er det ingen poeng i at UART er aktiv. Nettopp derfor kan det lønne seg å legge også disse blokkene i egne spenningsdomener slik at de kan skrus av uavhengig av andre blokker.



Figur 24 - Bruk av flere spenningsdomener

De forskjellige domenene her er også implementert med en tilhørende bryter, på lik linje med bryteren til ZPU, og styres av kontrollsignaler fra spenningskontrolleren. For å forenkle bildet er ikke bryterne illustrert her. Hvorvidt blokkene skal implementeres med samme type bryter som ZPU; med støtte for flere spenningsnivåer, vil være en vurderingssak. Begrunnelsen for å benytte flere spenningskilder er for å kunne redusere oppvåkningstiden. Denne tiden er avhengig av kapasitansen til blokken, som igjen er proporsjonal med størrelsen. Gevinsten med implementering av en slik bryter kan derfor være ikke-eksisterende. Små blokker vil ha mindre oppvåkningstid enn store blokker, og gevinsten ved støtte for flere spenninger kan være mindre enn arealkostnaden den utvidede bryteren medfører. Det antas her at timer og UART er av en slik størrelsesorden at flere spenningsalternativ for VDD ikke kan forsvares. Det antas også at oppvåkningstiden for disse er mindre enn den oppvåkningstiden ZPU har ved tilstanden REST. Denne tilstanden har

kortest oppvåkningstid. Dermed kan begge settes til å sove sammen med ZPU i alle sovemodi.

På samme måte som med ZPU er det nødvendig å gjøre en analyse av hvilke registre i komponentene som trengs å bevares ved soving. Som forenkling er det valgt å anse alle registre som kritiske, noe som gjør at denne analysen ikke er nødvendig, samt at arkitekturen til komponentene ikke trengs implementeres med resetting av ubevarte registre. Bevaring og isolering er heller ikke implementert inne i de enkelte komponentene som beskrevet i kapittel 3.3, men heller realisert med bruk av UPF. UPF konfigurasjonen som benyttes i dette kapittelet finnes i Vedlegg 4.

Figur 24 inneholder ikke alle de komponenter som tradisjonelt vil befinne seg i en mikrokontroller, som for eksempel en analog til digital omformer(ADC). Slike komponenter, som ikke vil benyttes som oppvåkningsstimuli, kan plasseres i egne spenningsdomener, slik at de kan settes til å sove sammen med ZPU. Grunnen til at det ikke er en god løsning å plassere de i samme domene som ZPU er at brukeren burde ha mulighet til å sette de forskjellige komponentene til å sove selv om ZPU er i aktiv tilstand. Mulighet for reduksjon av statisk effektforbruk mens ZPU er i aktiv tilstand vil ikke bli nærmere diskutert i denne rapporten, men er foreslått som videre arbeid.

3.7.1 Hardware

For at kjøring skal kunne gjenopptas etter soving er det som nevnt i kapittel 3.3 nødvendig at de registrene som ikke bevares blir resatt ved oppvåkning. Som i ZPU skjer dette også lokalt i hver blokk, og må implementeres ved endring av VHDL koden. I dette kapittelet er det valgt å bevare alle registre for å forenkle jobben noe. Dette vil også medføre at ingen signaler i blokkene må resettes ved soving.

Det er ønskelig at spenningskontrolleren ikke skal bli veldig kompleks. For å ha mulighet til å sette flere blokker i sovemodus samtidig som ZPU er det valgt å gjenbruke de sovesekvensene som allerede eksisterer. Siden spenningskontrolleren vet hvilke komponenter som skal benyttes som oppvåkningsstimuli vet den også hvilke blokker som er inaktiv, og som kan settes til å sove sammen med ZPU. Spenningskontrolleren er laget slik at den vil videredistribuere sovesignalene som ZPU benytter til alle blokker som ikke er i bruk. Dermed vil alle disse inaktive blokker bli satt til å sove samtidig med ZPU. Et eksempel er illustrert under, hvor signalet for bevaring av UART settes avhengig av hvorvidt den er i bruk under soving.

```
If UART_trigged = '1' then          --UART skal ikke settes til å sove
        U_retain_n <= '1';          --og sovesignalene til UART holdes inaktive
else                                --UART ikke benyttet. Skal da benytte
        U_retain_n <= retain_n;     --samme signaler som ZPU til å sove.
end if;
```

Alternativet til metoden beskrevet over er at hver komponent har en egen prosess i spenningskontrolleren hvor dens sovesignaler blir styrt. Bakdelen med denne metoden er at den vil kreve mer areal, samt vil ha et høyere statisk effektforbruk. Dette vil bli vist i 4.7

Siden det er sovesignalene til ZPU som tvinges inaktiv under testing vil dette medføre at sovesignalene til alle komponentene blir tvunget inaktive under test. På samme måte som tidligere er også styringssignalet til alle spenningsbryterne gjort ekstern kontrollerbare. Vi har da mulighet til å skru spenningen på de forskjellige blokkene av og på uavhengig av hverandre ved testing. For mer utfyllende informasjon om hvordan spenningskontrolleren fungerer, se VHDL koden i Vedlegg 2.

3.7.2 Software

Endringene beskrevet i 3.7.1 krever ingen ekstra jobb fra software. Det som derimot er nødvendig med tanke på en mikrokontroller er at softwarekoden som styrer hvilke data som skrives til spenningskontrolleren ligger i et eget bibliotek. Dette vil normalt gjøres skjult for brukeren, da det ofte er konfidensielt. Derfor er all kode som beregner data som spenningskontrolleren plassert i en headerfil. Denne kan inkluderes i brukerens programkode, og man har da mulighet til å sette ZPU til å sove. Soving initieres ved at brukeren gir kommandoen ”sovemodus”(”soveprofil” ”sovetid”). Sovemodus er enten SLEEP, REST eller HIBERNATE. Soveprofil er profilene beskrevet i kapittel 3.4, mens sovetid er antall klokkeperioder man ønsker at ZPU skal sove, gitt at timeren er brukt som oppvåkningsstimuli.

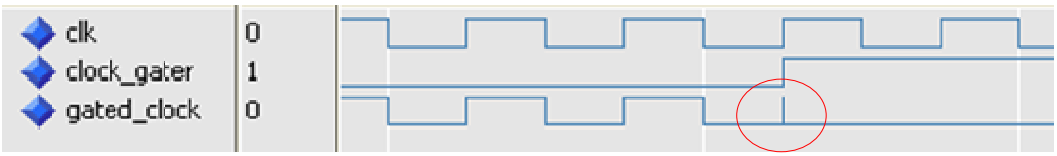
3.8 Andre endringer

3.8.1 Adressering

ZPU opererer opprinnelig med testing av bit for adressering av IO komponenter. Det vil si at i stedet for å sjekke hva verdien på hele adressen er, sjekker den hvorvidt et enkelt bit er satt høyt, og regner ut i fra det hva adressen er. Denne løsningen kan føre med seg problemer under kjøring, da den i noen tilfeller gir glitch på spenningskontrollerens *write_enable* signal. Dette fører til at spenningskontrolleren iverksetter soving på feil tidspunkt, og som regel også feil sovemodus. Grunnen til at denne glitchen kommer er at de bit som tilhører spenningskontrolleren er aktiv når adressen er en såkalt ikke-IO adresse. Når neste adresse er en IO adresse tror spenningskontrolleren at dataen på bussen er ment for den, og initierer soving. Dette er løst ved å teste verdien på hele adressen i stedet for å teste aktive IO bit. Begge metodene for adressetesting er illustrert i Vedlegg 6.

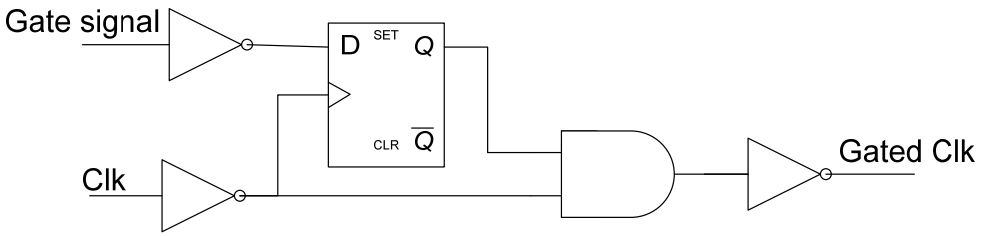
3.8.2 Klokkegating

For å kunne oppnå vellykket soving er det nødvendig å kunne gate bort klokken. Dette er gjort ved å sette klokken inn til ZPU lik 0 når spenningskontrolleren setter signalet for klokkegating høyt. Det som er viktig i denne prosessen er å sørge for at dette skjer på en måte som ikke gir feil på den gatede klokken, som illustrert i Figur 25. Signalet for klokkegating settes på høy klokkeflanke, noe som vil si at testing av stigende flanke på dette signalet vil føre til situasjonen vist under.



Figur 25 – Uheldig gating av klokken

Det samme resultatet vil inntreffe hvis vi tester om signalet er høyt på stigende klokkeflanke. For å få et korrekt resultat er det nødvendig å sjekke om signalet for klokkegating er høyt på fallende klokkeflanke. Den gatede klokken er da allerede lav når vi setter den lav, og vi unngår glitcher. Figur 26 viser logikken for klokkegatingen. For gating av klokken til logisk 1 tar man bort alle invertere, og tester dermed på høy klokkeflanke.



Figur 26 – Logikk for gating av klokke

VHDL koden som sørger for gating av klokken er lagt inn som en egen komponent. Siden vi i det gjeldende designet har tre klokker vi skal kunne styre, inneholder denne komponenten tre prosesser. Hver av disse prosessene leser signalet for klokkegating av sin respektive blokk. Den setter så klokken til denne blokken avhengig av verdien på den. For mer utfyllende eksempel, se Vedlegg 7.

4. Simuleringer

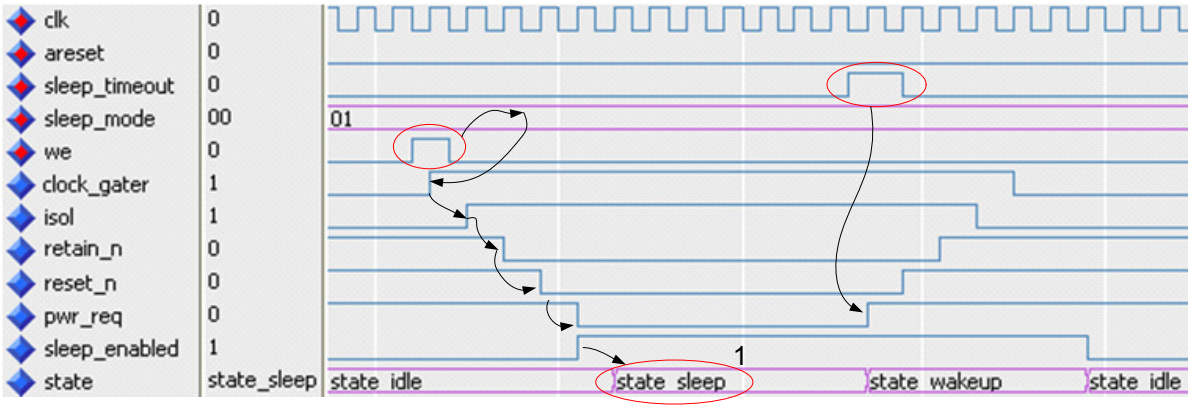
Dette kapittelet viser simuleringsresultater, samt verifikasjoner på at de deler som er implementert i kapittel 3 fungerer. Kapittelet er bygget opp på samme måte som kapittel 3, hvor de forskjellige delkapittlene her verifiserer de implementeringer som er gjort i kapittel 3. Vi vil også se på forskjellige kostnader forbundet med de forskjellige implementeringene. Til slutt vil vi se på hvilke effektbesparelser vi kan oppnå ved bruk av Power Gating. Tallene for areal som er oppnådd med syntese i dette kapittelet er relative. De er kun ment for å illustrere prosentmessig arealkostnader. Tall for statisk effektforbruk er oppnådd ved å syntetisere med 45nm digitalt cellebibliotek.

Synteseverktøy med støtte for UPF har vist seg vanskelig å oppdrive. For syntese er det valgt å benytte RTL Compiler® fra Cadence. Dette synteseverktøyet støtter ikke UPF, men derimot CPF. Grunnet tidsbegrensning og er det valgt ikke å konvertere UPF konfigurasjonen til CPF, noe som betyr at sovetransistorer ikke er implementert i denne masteroppgaven. Tallene for mulig besparelse og arealkostnad som det opereres med i dette kapittelet baserer seg derfor på resultater fra [10], hvor besparelse med en faktor på 250 er oppnådd i 90nm teknologi. Det antas dermed også at denne besparelsen gjelder for 45nm.

4.1 Spenningskontroller

Etter å ha utført de endringer i ZPUs hardware beskrevet i 3.1.1 er en av de medfølgende simuleringene kjørt for å se at de fortsatt fungerer. Ettersom software som kjøres i denne simuleringen ikke inneholder noen soveinstrukser skal også resultatet av simuleringen bli helt likt med resultatet simuleringen gav før endringene ble utført. ZPU skal da forbli i aktiv tilstand hele tiden. Da denne simuleringen ikke gir noe feil antas det at implementering av spenningskontrolleren ikke har noen påvirkning på funksjonaliteten til ZPU ved drift.

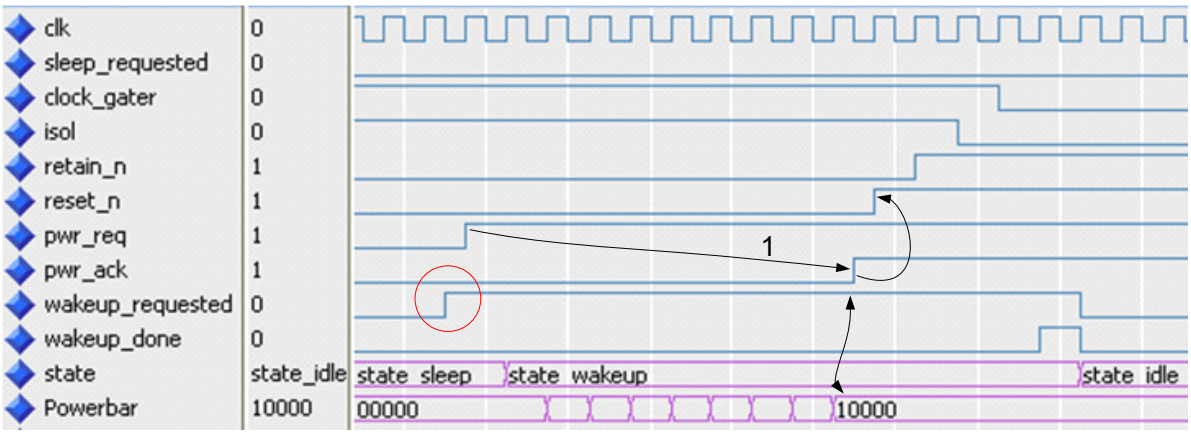
Den neste simuleringen verifiserer spenningskontrollerens funksjonalitet. For å oppnå dette er det laget en enkel testbenk som endrer inngangene til spenningskontrolleren etter en gitt tid slik at vi kan observere hvordan utgangene endrer seg i forhold til dette.



Figur 27 - Verifisering av spenningskontroller

Figuren over viser denne simuleringen. Alle innganger(fem øverste signaler) er styrt av testbenken, og vi kan observere hvordan sovesignalene ut ifra spenningskontrolleren oppfører seg i forhold til dem. Signalet *we,* eller write enable, går høy når ZPU skriver data til spenningskontrolleren. Spenningskontrolleren kontrollerer så den dataverdien *sleep_mode* har, og gjør en handling ut ifra det. I tilfellet over er det verdien ”0x01” som blir skrevet. Spenningskontrolleren setter så de nødvendige signalene, for deretter å indikere at ZPU sover(1). Deretter venter den på timeout før den vekker opp blokken som er satt i sovemodus.

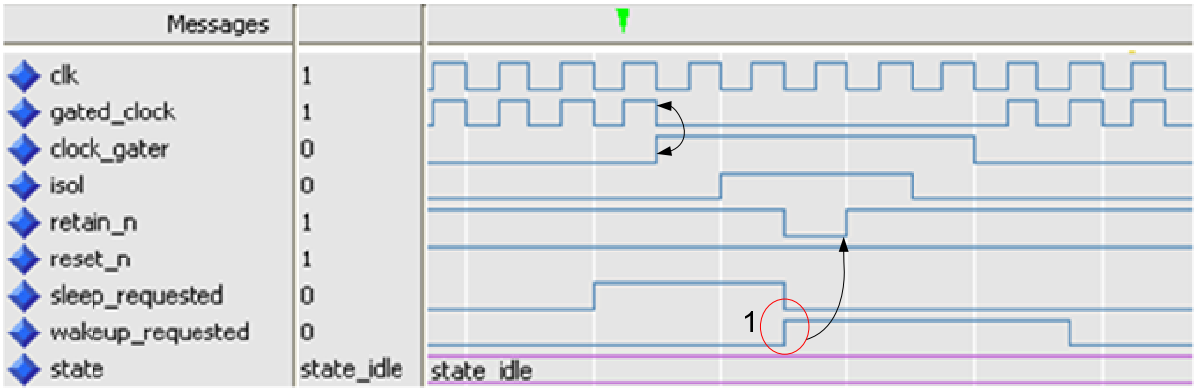
Simuleringen under illustrerer oppvåkningsforsinkelsen, som er implementert for å sikre at spenningen til blokken som vekkes opp er kommet på før oppvåkningssekvensen initieres.



Figur 28 - Oppladningstid

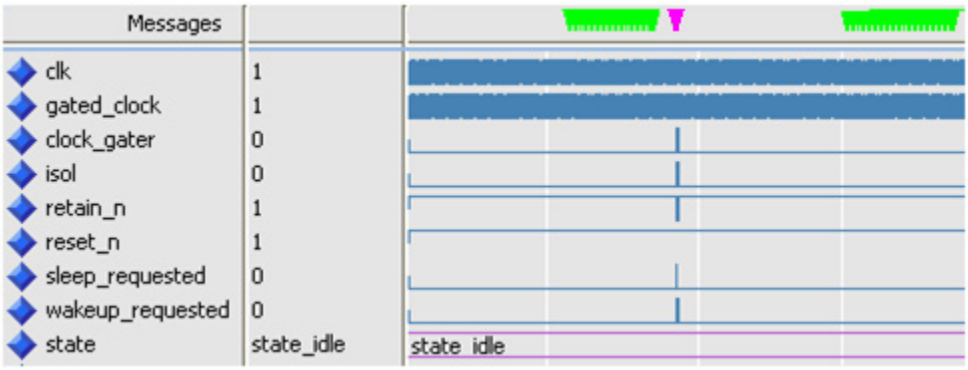
Som vi kan se er det lagt inn en tidsforsinkelse(1) fra spenningskontrolleren får beskjed om å vekke ZPU, *wakeup_requested*, til det registreres at driftspenning er oppnådd. Forsinkelsen er realisert ved å benytte en nivåindikator, *Powerbar*. Denne har en verdi som avhenger av hvilken tilstand ZPU befinner seg i, og dermed hvilken spenning som er tilkoplet spenningsskinnen. Når spenningen settes på vet denne indikatoren hvor langt unna ZPU er full driftspenning, og kan justere oppvåkningstiden avhengig av det.

La oss se på situasjonen hvor ZPU er nødt å bli vekket opp mens den er på vei til sovemodus. Figuren under viser resultatet når spenningskontrolleren får beskjed om å vekke ZPU(1) midt i sekvensen for soving. Dette inntreffer som vi ser etter at signalet for bevaring, *retain_n*, blir satt, men før signalet for reset.



Figur 29 - Avbrutt sovesekvens

Vi ser at signalet for bevaring så skifter tilbake, og at de signalene som er endret fram til det tidspunktet endres tilbake. ZPU er så tilbake i aktiv tilstand, og kan utføre neste operasjon. Den øverste linjen i figuren, *messages*, illustrerer simuleringsmeldinger. I dette tilfellet genereres den som en tilbakemelding på at data blir skrevet til spenningskontrolleren.



Figur 30 - Avbrutt soving

Figur 30 viser samme simulering som Figur 29, men med et større tidsperspektiv. Vi ser at ZPU utfører noen bestemte instruksjoner før den ber om å bli satt i sovemodus. Dette blir avbrutt, og ZPU fortsetter oppgaven sin. Simuleringsmeldingene i dette tilfellet skyldes data som skrives over UART, og data til timer og spenningskontroller.

Tilfellet hvor sovingen blir avbrutt midt i sovesekvensen har som nevnt i 3.1 den fordelen at det sparer tid og effekt i forhold til alternativet, som er å vente til sovemodus er oppnådd før ZPU vekkes. Tidsbesparelsen er en sum av hvor langt i sovesekvensen spenningskontrolleren er kommet, og oppladningstiden til VVDD. Effektbesparelsen er lik effektforbruket som trengs til å lade opp VVDD. Maksimal tidsbesparelse med tanke på sovesekvensen er 4 klokkeperioder. Tall for oppladningstid og effekt forbundet med ZPU vil bli gitt i kapittel 4.8.

Spenningskontrolleren vil føre til et ekstra effektforbruk i en mikrokontroller. Tabellen under viser tall for effektforbruk ved bruk av transistorer med standard, lav og høy terskelspenning. Tallene er oppnådd ved hjelp av Cadence RTL Compiler. Siden spenningskontrolleren ikke anses som timingkritisk, er det valgt å implementere den med høy V_T transistorer. Vi ser at dette er veldig lønnsomt i forhold til effektforbruket.

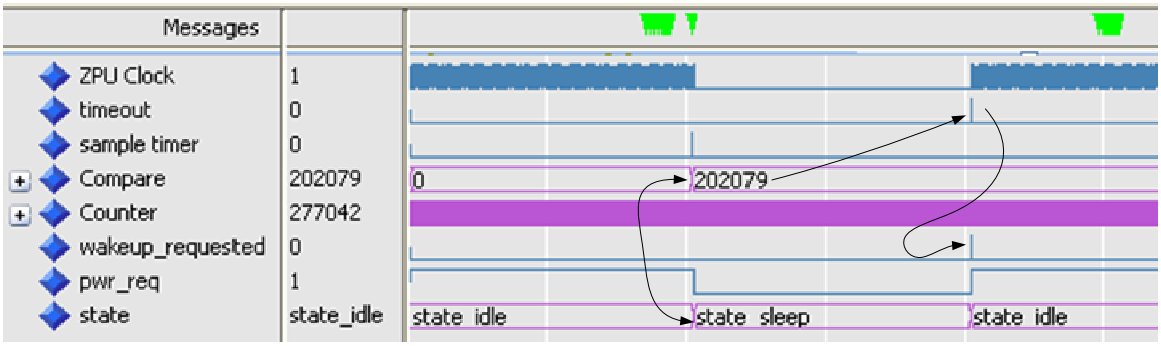
Transistortype	**Statisk effektforbruk**
Høy V_T	1319 nW
Standard V_T	2136 nW
Lav V_T	7075 nW

Tabell 3 – Statisk effektforbruk for lav, høy og standard V_T

Synteserapporter for spenningskontrollerens areal og effektforbruk er gitt i Vedlegg 8. Dette viser også dynamisk effektforbruk. Tallene for dynamisk effektforbruk er estimert av RTL Compiler, og anses ikke som veldig reelle. Grunnen til dette er at de beregnes ut i fra default konstanter som beskriver svitsjeaktivitet. Siden spenningskontrolleren kun opererer ved initiering av soving og oppvåkning vil det reelle tallet være mye lavere. For et mer nøyaktig resultat av dynamisk effektforbruk er det nødvendig med en toggle count format(tcf) fil. Denne beskriver svitsjeaktiviteten mer reelt.

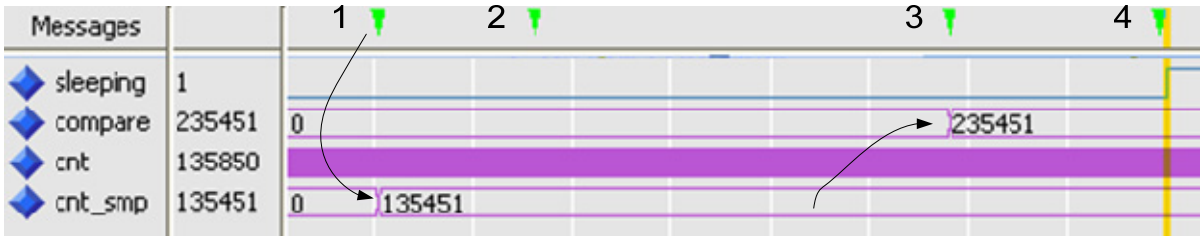
4.2 Tidsbasert soving

Dette kapittelet viser resultatene av simulering med tidsbasert soving. Det vil si at ZPU blir satt til å sove på et gitt tidspunkt, og en timer gir beskjed om at ZPU skal vekkes opp igjen etter et brukerdefinert antall klokkeperioder. Figuren under viser resultatet.



Figur 31 - Simulering med tidsbasert soving

Som vi ser av figuren blir ZPU satt til å sove etter at den har skrevet ut data over UART. Vi ser at sammenlinkningsregisteret, *compare,* får en ny verdi rett før soving iverksettes. Denne verdien fører så til at *timeout* blir togglet etter en viss tid, og ZPU vekkes opp.



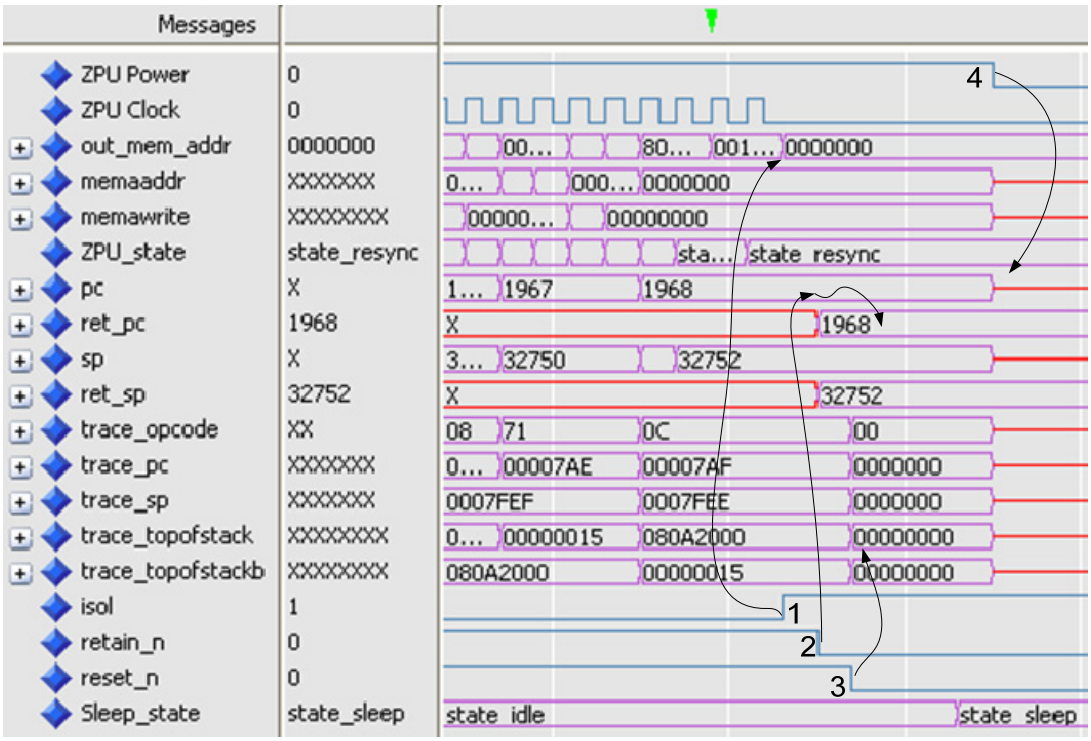
Figur 32 - Skriving til sammenlikningsregister

Figur 32 viser hvordan verdien til sammenlikningsregisteret blir skrevet til timeren. Ved første simuleringsmelding blir timeren samplet. Ved andre simuleringsmelding blir verdien lest ut til minnet, og verdien som sammenlikningsregisteret skal ha blir beregnet. Ved tredje blir denne verdien lagt inn i sammenlikningsregisteret, og ved fjerde settes ZPU til å sove. Som vi ser av verdien til telleren, *cnt*, ved den fjerde meldingen er ikke den faktiske sovetiden 100000 sykluser, men nærmere 99600. Dette er på grunn av at det tar ca 400 klokkeperioder fra timeren blir samplet, til spenningskontrolleren kan sette ZPU til å sove. Dette betyr at soving med bruk av timer vil føre til en 400 klokkeperioder lengre periode før ZPU settes i sovemodus, enn ved bruk av UART eller ekstern interrupt for oppvåkning. På bakgrunn av dette er det mulig å endre softwarekoden slik at verdien 400 legges til sovetiden. Dette er likevel ikke gjort, da det er vurdert slik at sovetiden må være den tiden

som ZPU forventes å være inaktiv. Øker vi sovetiden med 400 vil det føre til at forventet tid til neste aktivitet må være større enn sovetiden.

I denne masteroppgaven er det valgt å halvere størrelsen på telleren, for å benytte den siste halvdelen som et sammenlikningsregister. I dette tilfellet tjener det formålet, men i praksis er det ikke ønskelig å redusere størrelsen på telleren. Det vil da heller kreve et sammenlikningsregister på størrelse med den eksisterende telleren. Syntese av en slik implementering viser at dette vil øke arealet til telleren med ca 60 %. Synteserapport for de overnevnte versjoner av timeren finnes i Vedlegg 11.

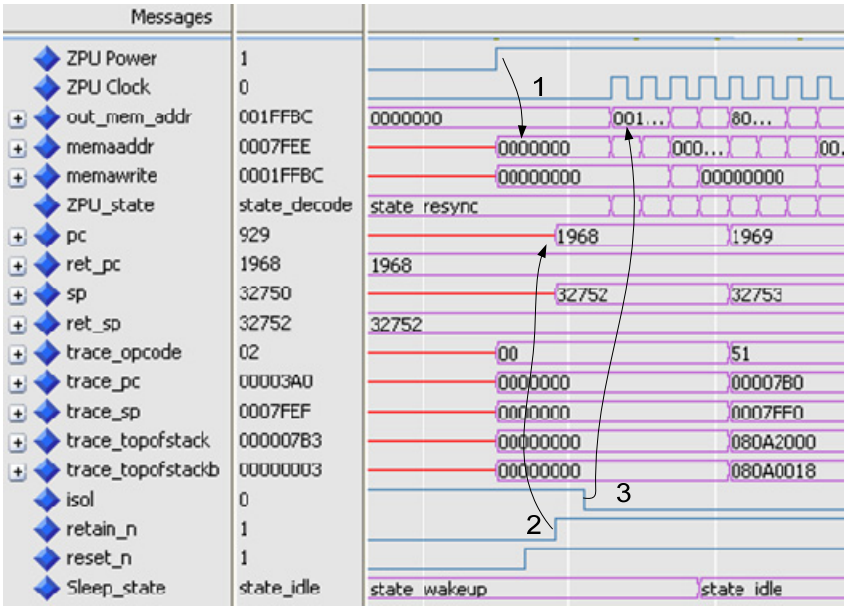
4.3 Isolering av utganger og tilstandsbevaring



Figur 33 - Isolerte utganger og bevaring av registre

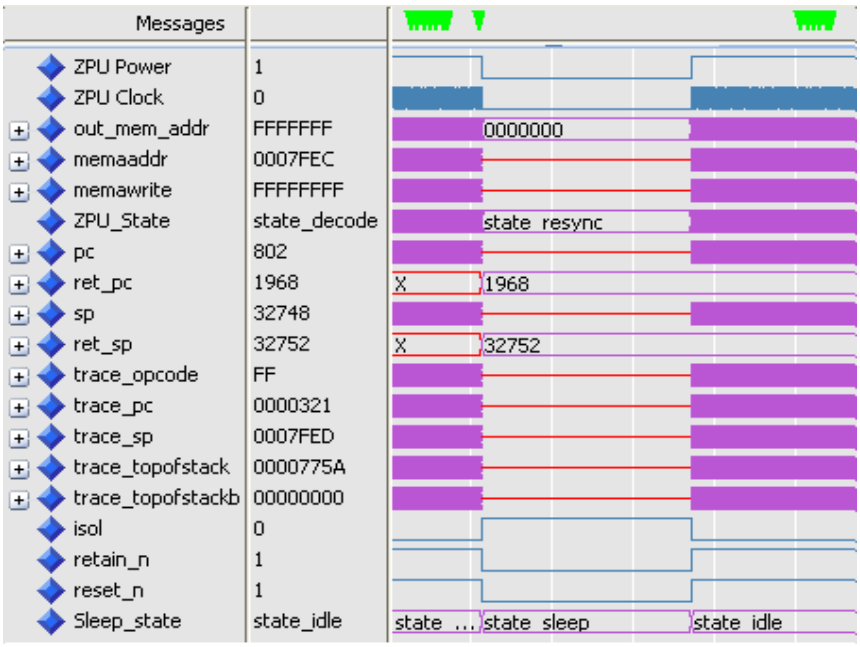
Figur 33 viser hvordan isolering og bevaring inntreffer ved soving. Vi ser at en utgang, *out_mem_addr,* blir satt lav når signalet for isolering går høyt(1). Videre kan vi observere hvordan verdien på programtelleren blir lastet over i det tenkte bevaringsregisteret, *ret_pc,* når signalet for bevaring settes lavt(2). Andre registre blir resatt når signalet for reset går lavt(3). Det øverste signalet, *ZPU Power*, forteller om ZPU har spenning på VDD. Som vi ser blir verdien av registrene ødelagt når spenningen forsvinner(4), foruten om bevaringsregistrene.

Figur 34 viser hvordan registerverdiene blir gjenopprettet når ZPU vekkes opp igjen. De registre som ikke bevares vekkes opp igjen med en forhåndsbestemt verdi(1), når ZPU har fått tilbake driftspenning. Programtelleren og stakkpekeren blir så gjenopprettet(2), før signalet for isolering settes lavt og utgangene frigjøres(3).



Figur 34 - Gjenopprettelse av tilstand

Figur 35 viser gjenomkjøringen av programmet som setter ZPU til å sove. De simuleringsgenererte meldingene viser at ZPU skriver data over UART etter oppvåkning, og den genererte loggfilen viser at data som skrives ut samsvarer med de data softwarefilen er satt til å skrive ut. Dette verifiserer at bruk av soving ikke har noe uønsket påvirkning på operasjonen til ZPU.



Figur 35 - Sovesekvens med gjenopptakelse av programkjøring

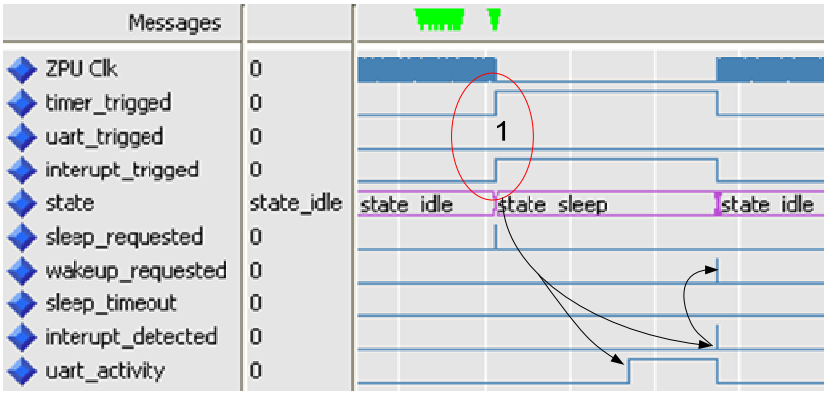
Sammenlikning av synteseresultater fra original ZPU, og revidert ZPU viser at arealet øker med 2 % når funksjonen for resetting ved soving implementeres. Da det ikke er benyttet syntheseverktøy med støtte for UPF i denne masteroppgaven vil vi ikke kunne se arealkostnaden som isoleringscellene og bevaringsregistre medfører. Det kan likevel gis et

estimat på denne kostnaden. ZPU består av utganger som til sammen består av 55 bit. Disse bit trenger hver sin 2 input NAND for å kunne isoleres. Fra Vedlegg 9 finner vi at en NAND2 port har et celleareal på 0,8. Dette medfører et celleareal på 44 for isoleringscellene, noe som er en økning på 0,5 %. Vi kan også estimere arealkostnaden ved bruk av bevaringsregistre. Hvis vi antar at implementeringen gjøres ved bruk av standard bevaringsregistre kan vi anta en arealkostnad på ca 20 %, jamfør 2.2.3, av arealet som programtelleren og stakkpekeren benytter. Syntese viser at disse to komponentene utgjør 3.7 % av det totale arealet til ZPU. En økning med 20 % av sp og pc vil dermed innebære en total arealkostnad på ca 0,7 %. Vi har da en total arealkostnad forbundet med isolering og bevaring som grovt regnet forventes å ligge i området 1 – 1,5 %. Det er viktig å bemerke at arealkostnad ikke nødvendigvis medfører et større fysisk areal for chip, da plassering av cellene kan optimaliseres. Arealkostnaden er heller et mål på økte produksjonskostnader.

4.4 Oppvåkningsstimuli

Figur 36 - Timer trigget oppvåkning

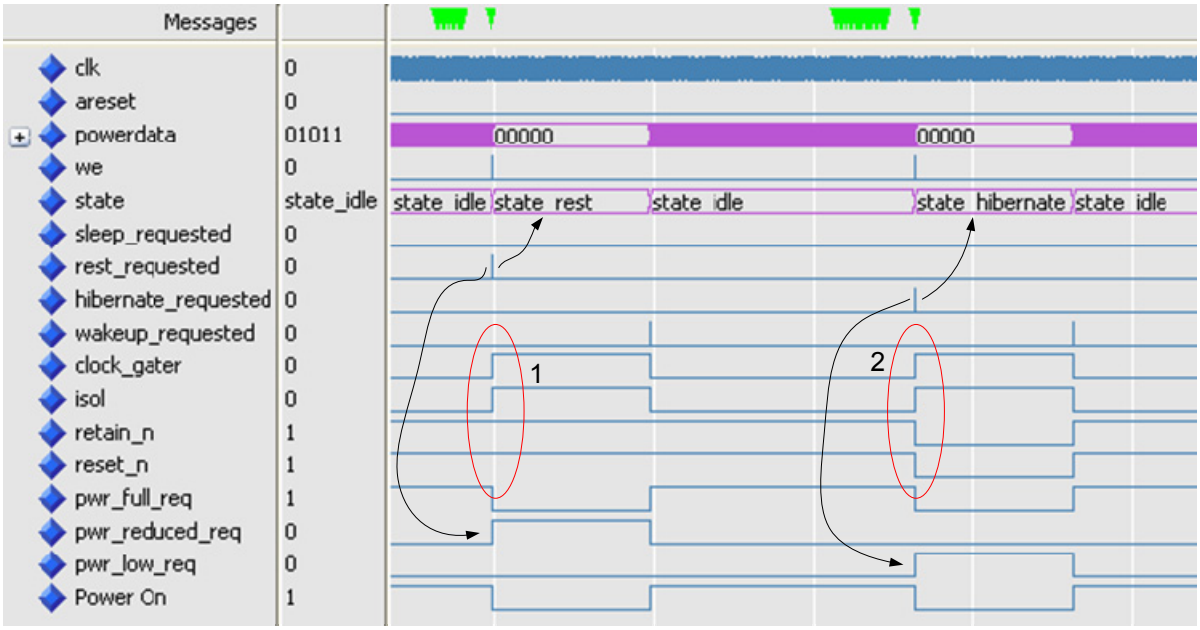
Figuren over viser resultatet når kommandoen *"sleep(4, 100000)"* gis av brukeren. Dette fører til at verdien 0x05, eller den binære ekvivalenten 00101, skrives til spenningskontrolleren. Hvis vi sammenlikner med Tabell 1, side 37, ser vi at dette skal tilsvare sovemodusen *SLEEP*, samt at oppvåkning kun skal kunne utløses av timer, jamfør tabell 2. Vi ser av figuren at spenningskontrolleren registrerer hvilke komponenter som skal benyttes som oppvåkningsstimuli(1). Videre kan vi observere at det detekteres interrupt før timeren gir timeout, men at spenningskontrolleren ikke reagerer på det.



Figur 37 - Timer og interrupt trigget oppvåkning

Figuren over viser resultatet når soveprofil 3 er valgt. Denne tilsier at spenningskontrolleren kan motta oppvåkningsbeskjed fra timer og interrupt(1). Vi ser at spenningskontrolleren setter signalet for initiering av oppvåkning, *wakeup_requested*, høyt når det blir detektert eksternt interrupt. Det er også lagt inn stimuli på UART, men siden soveprofilen ikke tilsier at UART skal vekke ZPU opp reagerer ikke spenningskontrolleren på det. På lik linje som beskrevet over er også oppvåkning ved hjelp av stimuli fra UART verifisert. Endringer gjort for å benytte forskjellige oppvåkningsstimuli har en minimal arealkostnad, foruten om kostnaden til timeren, beskrevet i kapittel 4.2.

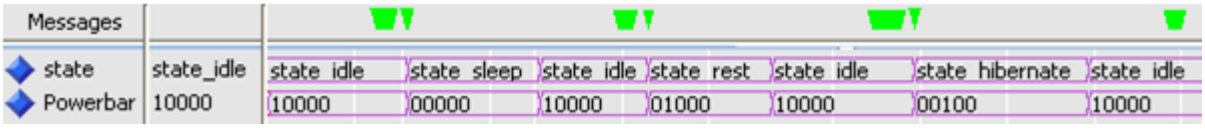
4.5 Flere sovemodi



Figur 38 - Bruk av flere sovemodi

Figur 38 viser to nye sovemodi. Vi kan observere at tilstanden *REST* ikke benytter seg av signaler for bevaring og resetting(1). Dette er, som nevnt i 3.5, siden denne tilstanden er tiltenkt en spenning på VVDD som bevarer innholdet i vippene. Tilstanden *HIBERNATE* derimot er tiltenkt en VVDD som ikke bevarer innholdet i vippene, og må derfor benytte seg av bevarende registre. De fire nederste signalene i figuren er tilknyttet spenningen. Den nederste forteller oss hvorvidt ZPU har driftsspenning, mens de tre andre er signaler som er ment å fortelle sovetransistorene hvilken spenning som skal påføres VVDD. Tilstanden *SLEEP* vil medføre at alle disse tre signalene settes lav.

Nivåindikatoren for VVDD som ble vist i kapittel 4.1 muliggjør raskere oppvåkning ved tilstandene REST og HIBERNATE enn ved SLEEP. Dette er siden den settes til en verdi høyere enn null ved disse tilstandene. Dermed trenger den å telle kortere før den indikerer driftsspenning. Dette illustreres av Figur 39.



Figur 39 - Forskjellig nivåindikering for forskjellige sovemodi

Ved hjelp av RTL Compiler er det vist at bruk av tre sovemodi framfor en vil gi en arealkostnad på ca 32 % for spenningskontrolleren. Cellearealet ved bruk av kun SLEEP var på 232, mens arealet med bruk av tre sovemodi er på 307, noe Vedlegg 8 viser. Statisk effektforbruk for spenningskontrolleren øker også noe ved bruk av flere sovemodi. Ved 3

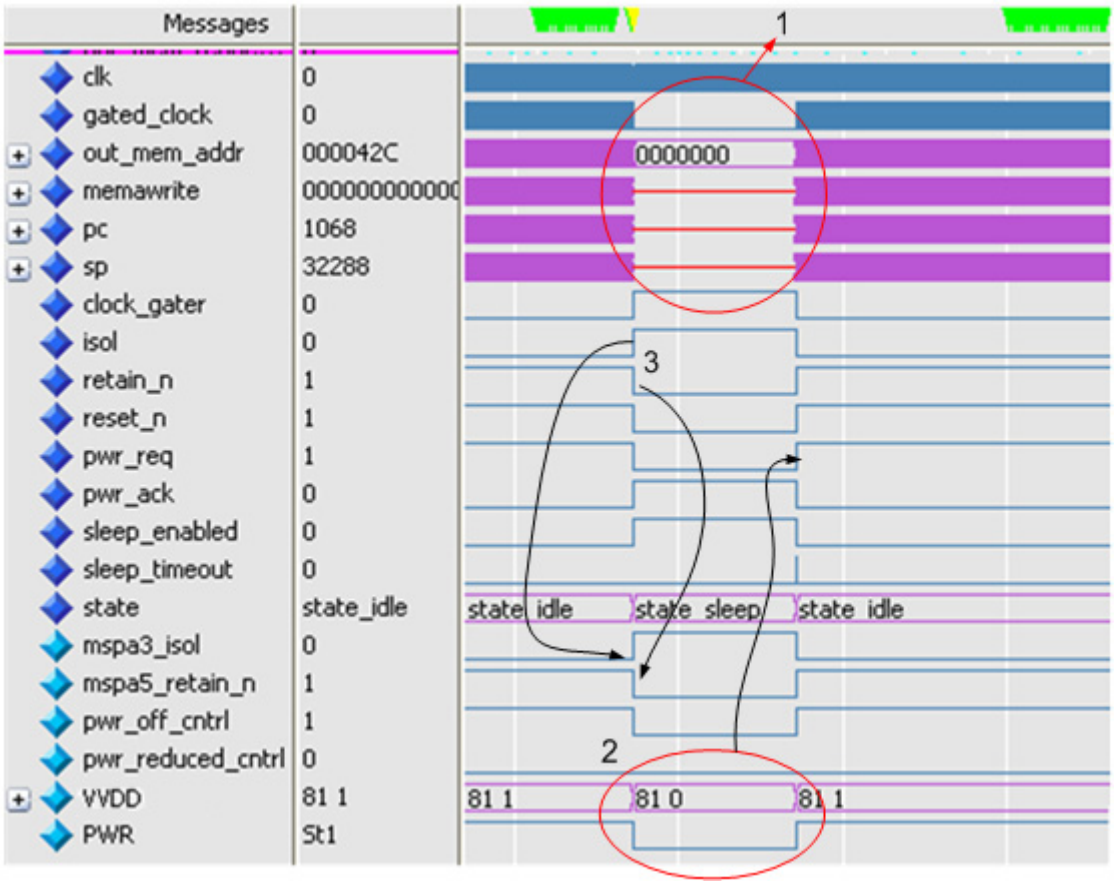
sovemodi har den et statisk effektforbruk på 1328nW ved bruk av lav V_T transistorer. Dette reduseres til 1020nW ved støtte for kun en sovemodus.

Den viktigste arealkostnaden ved bruk av flere sovetilstander er den som følger av mer komplekse sovetransistorer. Implementering av sovetransistorer er ikke gjennomført i denne masteroppgaven grunnet tidsbegrensning, men det vil bli gjort noen betraktninger rundt dette i neste kapittel.

For beregning av lekkasjestrøm ved REST og HIBERNATE er det nødvendig å bestemme hvilke verdi spenning spenningsskinnene skal ha ved disse modi. Argumentet for å ha disse modi med er å kunne velge sovemodi ut i fra aktivitetsprofil. Ut i fra dette er det avgjort at forholdet mellom oppvåkningstid for de forskjellige modi skal være tilnærmet lineært. Utregninger vist i Vedlegg 13 viser at VVDD ved REST skal være 720mV, mens HIBERNATE krever 270mV, når aktiv modus benytter 1,1V.

Effektmessige forskjeller på de forskjellige modi vil bli sett nærmere på i kapittel 4.8.

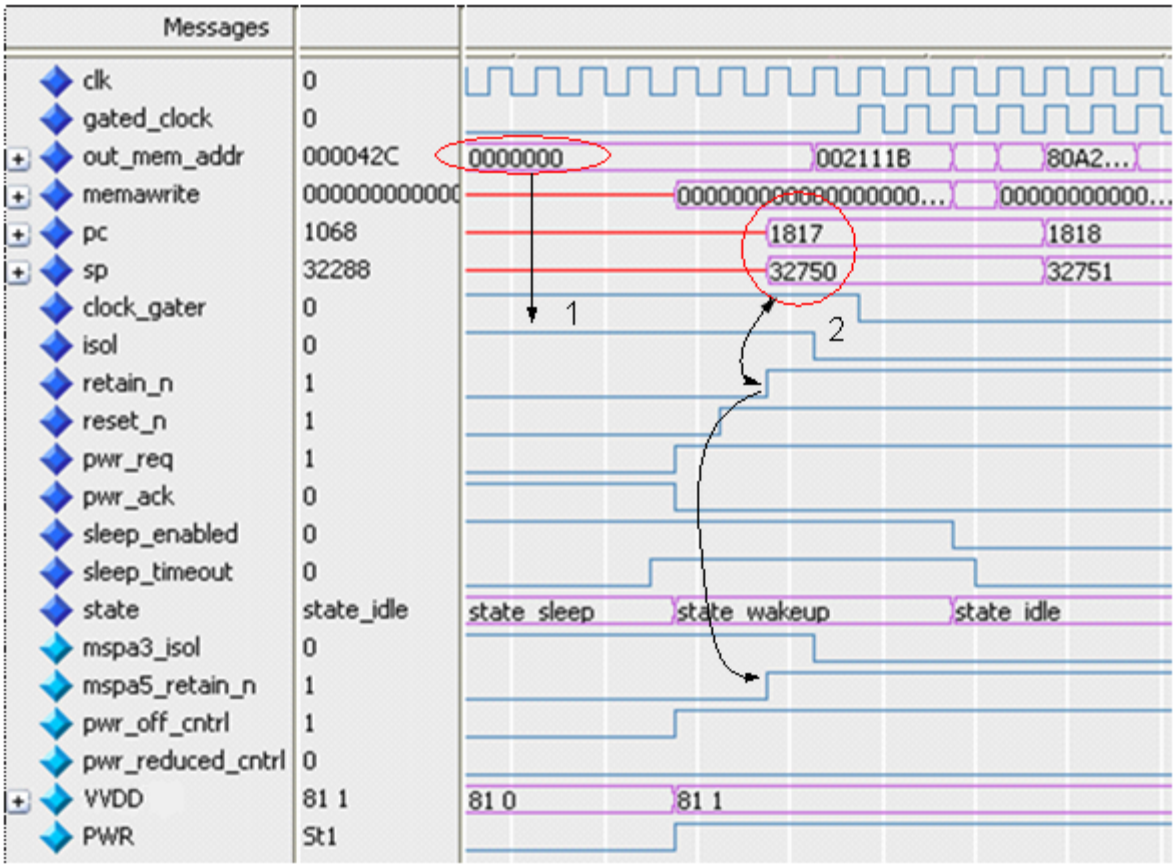
4.6 Implementering med UPF



Figur 40 - UPF kontrollert isolerling og bevaring

Figur 40 viser simulering med UPF implementering. De seks nederste signalene markerer de signaler som UPF leser fra spenningskontrolleren, samt spenningsegenskaper innført som følge av UPF konfigurasjonen. Vi observerer at isolering og bevaring blir registrert av UPF konfigurasjonen når de endres i spenningskontrollen(3), og de instruerte handlingene iverksettes. Som for eksempel at utgangen som inneholder minneadressen, *out_mem_addr,* blir trukket lav når signal for isolering er høyt(1). Det bevarende registeret er ikke mulig å se i simuleringen, men vi kan verifisere funksjonaliteten i Figur 41. Vi ser der at programtelleren og stakkpekeren får en verdi når signal for gjenopprettning inntreffer. Denne verdien er den samme som registrene hadde før soving.

Figur 40 viser også at utgangen fra bryteren, *VVDD,* endres når ZPU blir satt i sovemodus(2). Questasim har noen simuleringskommandoer for UPF konfigurasjonen. En av disse er - *corrupt_all-on_act{}.* Denne vil sørge for at alle registrene i domenet blir ødelagt når spenningen tas bort. Denne funksjonaliteten blir bekreftet av simuleringen, hvor signalene *memawrite, pc* og *sp* får verdien ”X” når spenningen koples fra VVDD(1).



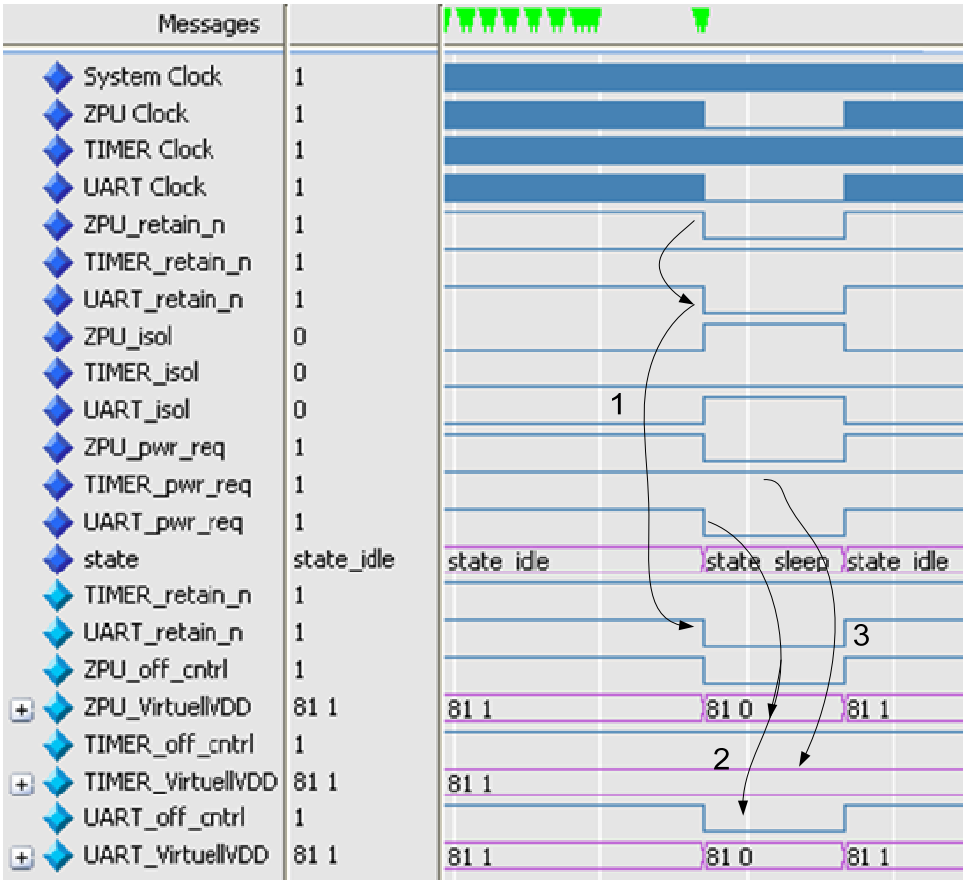
Figur 41 - Oppvåkning ved bruk av UPF

Figur 41 illustrerer samme simuleringen som Figur 40, men med fokus på oppvåkningstidspunktet. Vi kan her observere oppførselen ZPU har avhengig av signalene i spenningskontrolleren. Utgangen vil frigjøres når signalet for isolering tas bort(1), bevarte registre får tilbake verdien sin når signalet for bevaring inntreffer(2), og andre registre blir holdt i resatt tilstand så lenge signalet for reset er lavt og spenningen er på.

Som nevnt i innledningen til dette kapittelet er ikke sovetransistorer implementert i denne masteroppgaven. Dette skyldes at synteseverktøy med støtte for UPF viste seg vanskelig å anskaffe. De synteseverktøy som er benyttet støtter Cadences laveffekts konfigureringsspråk, CPF. Det er ikke valgt å prioritere oversettelse fra UPF til CPF, da erfaring viser at dette kan være tidskrevende. Arealkostnader forbundet med UPF/CPF er derfor estimater og antakelser. Estimater for arealkostnad forbundet med isolering og bevaring er gitt i kapittel 4.3. Det som gjenstår er da arealkostnader forbundet med sovetransistorer. Ut i fra observasjoner gjort i [19] er det rimelig å anta at arealkostnaden for komponenter med en sovemodi, det vil i dette tilfellet si UART og timer, vil ligge rundt 15 %. Hvis ZPU benytter seg av flere sovemodi vil den kreve mer komplekse sovetransistorer. Ved å sammenlikne observasjoner fra [10] og [19] antas det at denne kostnaden vil ligge på ca 40 % ved bruk av tre sovemodi, og 25 – 30 % ved bruk av to sovemodi.

4.7 Sovemodi for mikrokontroller

Vi vil her se på simuleringer av et litt større system, for å få en indikator på hvordan maksimal statisk effektbesparelse kan realiseres i en mikrokontroller. Alle inaktive komponenter blir da satt til å sove samtidig som ZPU. Arkitekturen til de forskjellige komponentene er som nevnt i kapittel 3.7 ikke endret på lik måte som i kapittel 3.3, men bevaring og isolering er kun implementert ved hjelp av UPF.



Figur 42 - ZPU og UART sover

Figuren over viser resultatet ved soveprofil 4 og sovemodusen *SLEEP*. Denne profilen benytter bare timer og ekstern interrupt som oppvåkningsstimuli, noe som vil si at UART ikke er i bruk. Som forklart i kapittel 3.7 vil da spenningskontrolleren registrere dette, og sette UART til å sove sammen med ZPU. Simuleringen over viser at kontrollsignalene for sovesekvensen blir videredistribuert til UARTs sovesignaler(1), men sovesignalene til timeren forblir inaktive. De åtte nederste signalene indikerer signaler lest fra spenningskontrolleren og egenskaper i spenningshierarkiet. Vi ser at virtuell VDD(VVDD) til ZPU og UART faktisk blir skrudd av(2), mens VVDD til timeren forblir tilkoplet VDD(3).

Begrunnelsen for å benytte seg av selektiv framfor full Power Gating, som beskrevet i innledningen til kapittel 2.2, er at det vil muliggjøre større reduksjon av statisk effektforbruk. Dette er også argumentet for å la inaktive komponenter sove sammen med ZPU. Tabell 3

under illustrerer en mulig effektbesparelse ved bruk av et system bestående av ZPU, timer, UART, samt en ekstern pinne for oppvåkning. Tallene for opprinnelig statisk effektforbruk baserer seg på rapporter fra RTL Compiler. Disse finnes i vedlegg 8 til 11. Tallene for besparelse baserer seg på oppnådd besparelse i [10], 250x.

	Statisk effektforbruk		
Komponent	Uten Power Gating(PG)	Med PG Aktiv modi	Med PG Sovemodi
ZPU	*165433nW*	*~165433nW*	*660nW*
Timer	*5511nW*	*8081nW*	*32nW*
UART	*15521nW*	*~15521nW*	*62nW*
Spenningskontroller	--	*1319nW*	--

Tabell 4 – Effekttabell

Som tabellen over viser oss er det mulighet for stor statisk effektbesparelse ved bruk av Power Gating. La oss sammenlikne bruk av Power Gating med den mer tradisjonelle metoden for dynamisk effektbesparelse, klokkegating. Ved klokkegating vil ikke statisk strømtrekk bli redusert, noe som vil si at systemet har et statisk effektforbruk lik 186 µW. Hvis vi tenker oss et tilsvarende Power Gated system, som skal vekkes opp av stimuli på ekstern pinne, vil det statiske effektforbruket kunne reduseres til 2,1 µW. Det antas da at det ikke er noe statisk effektforbruk forbundet med oppvåkning ved hjelp av ekstern pinne.

Hvis det er ønskelig å kun benytte klokken som oppvåkningsstimuli kan statisk effektforbruk reduseres til 10 µW, siden timeren vil gi et større bidrag til statisk effektforbruk da den er aktiv. For en komplett mikrokontroller vil disse tallene være større, da flere komponenter innføres. Total besparelsesfaktor for hele systemet vil da også bli større, ettersom lekkasjebidraget fra spenningskontrolleren vil utgjøre en mindre andel.

Alternativet til å implementere spenningskontrollen med videredistribuering av sovesignalene som ZPU benytter er å ha en egen prosess for hver komponent. Denne prosessen sjekker da om data inn til spenningskontrolleren tilsier at den skal aktiveres. Dette vil medføre en større arealkostnad. Syntese av en spenningskontroller designet på måten beskrevet over indikerer at den vil få en arealøkning på ca 10 % for hver ekstra komponent den skal styre. Det vil også føre til et større effektforbruk for spenningskontrolleren. Ved hjelp av RTL Compiler vises konsekvensene av en slik implementering under.

Antall prosesser	Celleareal	Statisk effektforbruk
En prosess(styrer 3 komponenter)	308	1319nW
Tre prosesser(3 komponenter)	363	1328nW

Tabell 5 – Kostnader for spenningskontrolleren

Som vi ser vil det statiske effektforbruket øke noe når flere prosesser introduseres. Ut i fra de tall som er oppnådd her er det rimelig å anta at implementeringen som benytter *en* prosess er bedre enn alternativet hvor hver komponent har sin egen prosess for kontroll av sovesignalene. Tall for dynamisk effektforbruk er ikke tatt med her, men er vist i Vedlegg 8.

Det vil også komme en ekstra arealkostnad for timeren og UART hvis de skal ha mulighet til å settes i sovemodus. For timeren antas det at det eneste som må bevares er tellerregisteret. Denne utgjør ca 37,5 % av arealet. 20 % ekstra av dette(se 2.2.3) medfører en arealkostnad på 7.5 % for bevaringsregisteret. I tillegg kommer en arealkostnad forbundet med isoleringscellene. Siden telleren har utganger bestående av til sammen 33 bit vil cellearealet av NAND2 portene for isolering være på 26,4. Dette utgjør en videre arealkostnad på 2,5 %. Samlet arealkostnad for isolering og bevaring er da ca 10 %.

Siden kritiske registre i UART ikke er analysert antas det at de utgjør like stor prosentdel som i ZPU, 1,5 %. UART har utganger bestående av 11 bit, noe som gir et celleareal for isoleringscellene på 8,8. Det antas da en arealkostnad på ca 2 % for isolering og bevaring for UART.

	Celleareal	
Komponent	Uten Power Gating	Med Power Gating
ZPU	*7643*	*~10900(8920*)*
Timer	*1051*	*~2080***
UART	*750*	*~880*
Spenningskontroller	--	*308(231*)*
*Ved støtte for en sovemodus **inkludert sammenlikningsregister		

Tabell 6 – Arealkostnad

Tallene baserer seg på de arealkostnader beskrevet i tidligere kapittel. Kostnad for implementering av sovetransistorer er også inkludert. Det antas av ZPU benytter tre sovemodi, men timer og UART benytter seg av en. Tallene i tabell 4 viser at for systemet benyttet i denne masteroppgaven er det en arealkostnad på ca 30 % for muliggjøring av soving. Hvis ZPU skal støtte tre sovemodi vil denne kostnaden ligge på ca 50 %.

4.8 ZPU

Synteserapporter i Vedlegg 9 viser at ZPU har en lekkasjestrøm på 165 µW, eller 150 µA. Lekkasjestrøm ved sovemodi er ikke simulert, men antatt på bakgrunn av observasjoner fra [10], samt egne matematiske beregninger. [10] viser at en effektbesparelse med faktoren 250 er mulig ved bruk av Power Gating for 90nm teknologi, og det er denne faktoren besparelsen for SLEEP benytter seg av. Vi vil dermed anta at faktoren er lik for 45nm. Spenningsverdier for VVDD ved REST og HIBERNATE ble gitt i 4.5. Ved bruk av (eq. 1) kan vi da beregne statisk effektforbruk ved disse modi, ut ifra de synteserapporter som er gitt i Vedlegg 9. Denne utregningen finnes i Vedlegg 12.

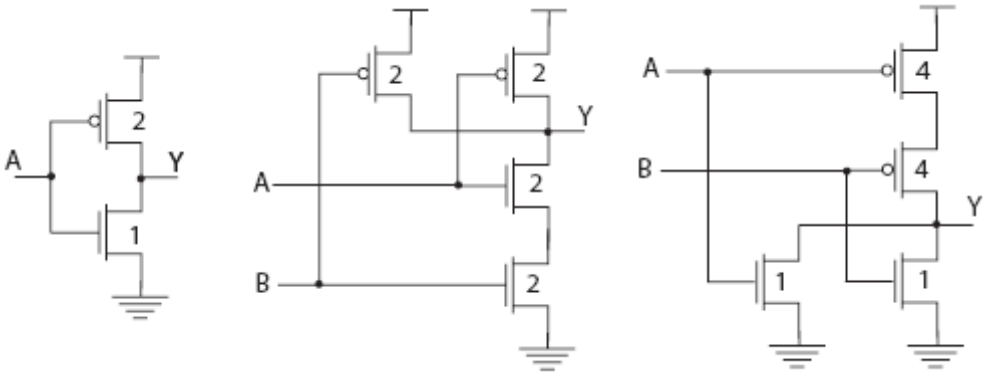
Modi	**ACTIVE**	**REST**	**HIBERNATE**	**SLEEP**
Statisk effektforbruk @ 45nm	165443nW (~150µA)	~88230nW (80µA)	~41360nW (37,5µA)	~650nW (0,6µA)

Tabell 7 – Statisk effektforbruk for ZPU i forskjellige modus @ 45nm

For å beregne korrekt oppvåkningsforsinkelse for ZPU er det nødvendig å vite maksimal kapasitans på VVDD. Dette kan oppnås ved ekstrahering av utlegg og SPICE-simuleringer, eller det kan estimeres. Estimeringen kan gjøres ved å telle hvor mange porter av hver type designet inneholder, for så å summere maksimal kapasitans i portene.

En enklere, men mer unøyaktig metode for estimering er å anta at kapasitansen er proporsjonal med arealet. Hvis vi tar for oss enkle porter slik som NAND2, NOR2 og invertere kan vi se på hvor stor del av arealet de utgjør, for så å regne ut hva kapasitansen ville vært hvis alt arealet besto av disse portene. Maksimal kapasitans sett fra VDD vil være tilfellet hvor alle PMOS er åpen, pluss gatekapasitanser. Vi antar at diffusjonskapasitans er tilnærmet lik gatekapasitans. NAND2 medfører en kapasitans på 14C(se Figur 43), NAND3 har 24 C, NOR2 har 20 C, NOR3 har 42 C, mens inverter har 6 C[20]. De overnevnte portene utgjør 10.4 % av kretsen, og har en samlet kapasitans på 15065 C. Oppskalering av denne kapasitansen medfører da at maks kapasitans på VDD vil være på ca 145000 C. I følge databladet til 45nm cellebibliotek som er benyttet inngangskapasitansen på en inverter 0,45fF. Siden gatekapasitansen på en inverter er 3 C får vi da at C er lik 0,15fF. Dette gir en faktisk maksimal kapasitans sett fra VDD på ca 21,5pF. Da denne estimeringsmetoden er unøyaktig er det viktig å ha god feilmargin. Så lenge den faktiske kapasitansen er mindre eller lik den beregnede er det ikke noe fare, men i motsatt tilfelle vil oppladningsforsinkelsen som spenningskontrolleren opererer med bli for liten. Vi risikerer da å initiere oppvåkningssekvensen før driftsspenning er oppnådd, og feil kan oppstå. For å ta høyde for andre kapasitanser, som for eksempel brønnkapasitanser, velges det å skalere maks kapasitans sett fra VDD opp med en faktor på 10, slik at verdien blir 215pF. Effektforbruket som kreves for å lade denne kapasitansen opp er 130pW, som vil bli vist i 4.9. Det gjøres

igjen oppmerksom på at metoden benyttet her ikke er særlig nøyaktig, spesielt med tanke på effektforbruket, men er tatt med for å illustrere effektforbruk ved oppvåkning, samt beregning av oppvåkningstid.



Figur 43 – Minimumskapasitanser for balansert inverter, NAND2 og NOR2[20]

4.9 Bruk av sovemodus

For effektiv bruk av soving er det nødvendig å vite hvor lenge blokken må befinne seg i sovemodus for at det skal medføre en effektbesparelse. Denne tiden, T_{null}, er avhengig av statisk effektbesparelse per tidsenhet ved bruk av Power Gating, og effektforbruk inn og ut av sovemodi. De tre sovemodiene vil som tidligere nevnt ha forskjellige oppvåkningsforsinkelser, effektforbruk ved oppvåkning og lekkasjebesparelse. Dette medfører at minimum sovetid er forskjellig for hver sovemodus. Vi kan regne den ut med hjelp av følgende formel:

$$E_{spart}=E_{lade} \qquad \text{(eq.5)}$$

hvor E_{spart} er effektbesparelsen ved bruk av de forskjellige sovemodi, og E_{lade} er effekten det kreves å lade opp VVDD. For enkelhets skyld velger vi å se bort i fra den effekten som kreves for bevaring av programteller og stakkpeker, samt den effekten isoleringscellene bruker. Dette på grunn av at de antas som svært små.

Energien som kreves for å lade opp en kapasitans kan utrykkes ved

$$E_{lade}=1/2CV^2 \qquad \text{(eq.6)}$$

hvor C er kapasitansen og V er spenningen. Setter vi inn tall for C og V får vi at det kreves 130 pJoule for å lade opp kapasitansene i ZPU, når vi går fra 0 til 1.1 Volt. Ved analyse av tall fra Tabell 7 får vi at besparelsen med SLEEP modus er på 164750nJoule/sek. Dette gir en minimum sovetid på 789µs for SLEEP. På samme måte kan minimum sovetid for HIBERNATE og REST beregnes. Utregningene finnes i Vedlegg 13, og resultatene vises i tabellen under.

Modus	**Besparelse (nJoule/s)**	**Oppladnings-effekt (pJoule)**	**Minimum sovetid, T_{null}**
SLEEP	164750	130	789ns
HIBERNATE	124083	74	569ns
REST	77213	15	194ns

Tabell 8 – Minimum sovetid for effektbesparelse

Det gjøres oppmerksom på at de tall som beregnes her blant annet baserer seg på maksimal kapasitans for VVDD. Dette tallet er som tidligere nevnt ikke veldig nøyaktig, og den samme unøyaktigheten vil gjelde for resultatet vist her. Mer nøyaktige resultater oppnås ved ekstrahering av utlegg, og SPICE simuleringer for oppladningseffekt.

5. Diskusjon

5.1 Kostnader og besparelse

Som vist i kapittel 4 er det noe arealkostnad forbundet med muliggjøring for soving i en mikrokontroller. Som eksempel velger vi å se på et system bestående av ZPU, UART og timer. Det totale cellearealet for dette systemet er opprinnelig ca 9450. Ved de modifikasjoner beskrevet i denne rapporten vil dette arealet øke til rundt 14160. Dette utgjør en ekstra arealkostnad på ca 50 %. En mikrokontroller vil bestå av flere komponenter enn det som er beskrevet her, og den relative kostnaden vil dermed være noe mindre, da kostnadene for hver enkelt komponent som oftest vil være lavere enn 50 %.

Vi kan benytte det samme eksempelet for å se på statisk effektbesparelse. I tilfellet hvor systemet kun benytter seg av klokkegating for effektreduksjon vil ZPU, UART og timer føre til et statisk effektforbruk på 186 µW når ZPU ikke er operativ. Hvis vi ser på tilfellet med Power Gating, hvor ekstern stimuli benyttes som oppvåkningsstimuli vil vi kunne redusere dette til 2,1 µW. Dette utgjør en effektreduksjon med en faktor på ca 90. Dette tallet baserer seg på en effektreduksjon på 250 x for hver enkelt komponent, som beskrevet i refererte artikler. Det er dermed vist at den faktiske besparelsen ikke er like glamorøs som effektreduksjonen i de enkelte komponentene. Dette er grunnet innføring av en spenningskontroller, som medfører et økt statisk effektforbruk. Ved innføring av flere komponenter vil spenningskontrolleren bidra med en mindre andel av total statisk effektforbruk i systemet, og total besparelsesfaktor vil øke. Hvor mye den øker med vil være avhengig av størrelsen på systemet.

Fra [2] er det oppgitt at bruk av tre sovemodi kan gi en ekstra statisk effektbesparelse på 17 % i forhold til bruk av en. Det er vist at bruk av flere sovemodi har en betydelig større arealkostnad enn bruk av en. Hvorvidt flere sovemodi kan forsvares vil være avhengig av hvor stor del av totalt effektforbruk som er statisk. Dette er igjen teknologiavhengig. La oss anta at systemet som er benyttet i denne masteroppgaven har en aktivitetsprofil hvor den kan settes i SLEEP 50 % av tiden. Tabellen under viser hvor mye totalt effektforbruk da kan reduseres, ved bruk av en og tre sovemodi.

Teknologi	**Statisk effekt/ Total effekt***	**Mulig total effektbesparelse ved en sovemodus****	**Mulig total effektbesparelse ved tre sovemodi*****
32nm	0,9	42 %	49 %
45nm	0,5	33 %	38,5 %
90nm	0,1	9 %	10,5 %
*Tallene er antatt ut i fra analyse av Figur 2 **Baserer seg på en besparelsesfaktor på 90 ved soving ***17 % ekstra besparelse jamfør [2]			

Tabell 9 – Reduksjon av effektforbruk

Som vi ser av tabellen forsvinner fortjenesten med flere sovemodi ved større geometrier. Siden arealkostnaden vil være den samme uansett geometri, vil det ikke være hensiktsmessig å implementere støtte for tre sovemodi i større geometrier. Hvor grensen vil gå er en vurderingssak i de forskjellige situasjoner. Dersom designeren anser det som hensiktsmessig med en arealkostnad på 20 % ekstra for å kunne redusere det totale effektforbruket med 1 % så er det mest sannsynlig det. Hvis vi tenker oss at aktivitetsfaktoren tilsier at systemet kan sove 99 % av tiden vil den totale besparelsesfaktoren bli større. I batteridrevne enheter er en effektbesparelse på 40 % svært gunstig da det vil forlenge levetiden betraktelig.

Tabell 7 viser nøkkeltall for bruk av sovemodus i ZPU, realisert i 45nm, lav V_T teknologi. Ut i fra tall for effektbesparelse og oppladningseffekt er det beregnet en minimum sovetid for at bruk av sovemodus skal være effektbesparende. I tillegg må det vurderes om oppvåkningsforsinkelsen er akseptabel.

	Aktiv	REST	HIBERNATE	SLEEP
Virtuell VDD	1,1V	0,72V	0,27V	0V
Lekkasjestrøm	150µA	80µA	37,5µA	0,6µA
Oppvåknings – tid*	--	85ns	170ns	240ns
Minimum soveperiode**	--	194ns	569ns	789ns
Effektreduksjon	--	1,875x	4x	250x
*Oppladningstid og gjenopprettelse av tilstand @ 100MHz. Se vedlegg 13. **Ved bruk av timer må den forventede sovetiden være 400 klokkeperioder lengre, på grunn av forsinkelser beskrevet i 4.2				

Tabell 10 – Egenskaper for ZPU i forskjellige modus @ 45nm

5.1 Videre arbeid

FPGA Prototyp

All verifikasjon fram til nå er blitt utført i simuleringer. Grunnen til at implementeringen for bevaring og isolering er utført slik som beskrevet i kapittel 3.3, hvor også ødelegging av registre er implementert, er for å kunne lage en FPGA prototyp. Det samme gjelder implementering av UART. Siden denne ikke har spenningsdomener er det gjort endringer som etterlikner den oppførselen som vil opptre ved bruk av Power Gating i et ASIC design. Grunnen til dette er at produksjon av ASIC er meget kostbar. Ved å benytte omprogrammerbar logikk kan vi verifisere signalflyten til designet til en brøkdel av denne kostnaden.

Lekkasjereduksjon ved aktiv tilstand

På lik måte som Tabell 2, side 38, viser forskjellige soveprofiler er det hensiktsmessig med reduksjon av statisk effektforbruk fra inaktive komponenter når ZPU er aktiv. En slik metode er foreslått i Tabell 11. For å muliggjøre dette er det nødvendig at kontrollsignalene for ZPUs soving *ikke* leses direkte fra prosessen i spenningskontrolleren, men at de videredistribueres på lik linje med sovesignalene for timer og UART. Man har da mulighet til å sette hver enkelt komponent til å sove helt uavhengig av andre. Tabell 3 på neste side viser en utvidet versjon av tabell 2. Som vi ser vil profil 8 til 14 her sette andre komponenter i sovemodus selv om ZPU er aktiv.

Implementering av sovetransistorer

Som nevnt i kapittel 4 er ikke sovetransistorene implementert. Det vil være veldig interessant å sammenlikne implementeringer med toppsvitsjing og med bunnsvitsjing, samt soveceller som kreves for bruk av flere sovemodi. Både den totale arealkostnaden, men også arealkostnaden som isoleringscellene krever for å ha VSS tilgjengelig. Denne rapporten tar ikke for seg ytelsestap som følge av spenningstap over sovetransistorer, men antar at sovetransistorene er optimalisert av utleggsverktøyet. Ut i fra resultater fra [10] er det rimelig å anta at ytelsen vil reduseres med ca 6.5 %, men det er ønskelig å se hvor stor ytelsestapene for ZPU blir som følge av Power Gating.

Klokkegating for spenningskontrolleren

Spenningskontrolleren er kun aktiv i perioden rett før soving, og rett etter. I de andre periodene vil den føre til et økt dynamisk effektforbruk når klokken endres. Det vil være interessant å se på metoder for gating av klokken til spenningskontrollere når den ikke trengs, for å se hvor mye effektforbruk som kan reduseres.

Operasjons-profiler #	*ZPU*	*Timer*	*UART*	*Interrupt handler*
1	Off	On	On	On
2	Off	On	On	Off
3	Off	On	Off	On
4	Off	On	Off	Off
5	Off	Off	On	On
6	Off	Off	On	Off
7	Off	Off	Off	On
8	On	On	On	On
9	On	On	On	Off
10	On	On	Off	On
11	On	On	On	Off
12	On	Off	On	On
13	On	Off	Off	Off
14	On	Off	Off	On

Tabell 11 – Operasjonsprofiler

Nivåindikator i spenningskontrolleren

Slik nivåindikatoren beskrevet i 3.5 fungerer nå vil den kun fungere når den har en klokkehastighet på 100MHz. Denne er nødt å endres slik at frekvensen er en faktor i addering og subtraheringen, hvis spenningskontrolleren skal benyttes i en mikrokontroller som støtter flere hastigheter.

5.2 Konklusjon

Det er i denne masteroppgaven vurdert metoder for reduksjon av statisk effektforbruk i en mikrokontroller. Metoden som er funnet best egnet til dette er MTCMOS Power Gating. Ved hjelp av denne er det implementert sovemodi, hvor spenningskinnene til blokken reduseres eller skrus av for reduksjon av lekkasjestrøm, i et system bestående av en mikroprosessor, UART og timer. En spenningskontroller for kontroll av soveegenskaper er designet, og metoder for bevaring av tilstander, og isolering av utganger er vurdert. Det er også foreslått strategier for maksimal besparelse av statisk effektforbruk i systemet som er benyttet. På bakgrunn av disse observasjoner er det mulig å si at bruk av Power Gating som metode for effektreduksjon i en mikrokontroller kan være svært lønnsomt. Det er vist at for systemet beskrevet i denne rapporten er det mulig å redusere statisk effektforbruk med en faktor på ca 90. Selv om statisk effektreduksjon på de enkelte komponenter er med en faktor på 250, vil spenningskontrolleren medføre et statisk effektforbruk som reduserer den totale besparelsen. Ved en aktivitetsprofil hvor prosessoren er passiv ca 50 % av tiden er det vist at det er mulig å redusere det totale effektforbruket med ca 40 %, ved bruk av 45nm teknologi.

Bruk av flere sovemodi for mer aggressiv reduksjon av effektforbruk anses også som nyttig. For 45nm teknologi er det vist at det er mulig å redusere totalt effektforbruk med over 5 % ved bruk av tre sovemodi framfor en. Denne besparelsen avhenger av aktivitetsprofilen, og vil i enkelte tilfeller være større. I hvor stor grad det er nyttig med flere sovemodi avhenger av hvor stor del av totalt effektforbruk som er statisk. Ved større geometrier utgjør det statiske effektforbruket en mindre del av totalt effektforbruk, og gevinsten med flere sovemodi reduseres. Det samme gjelder gevinsten med bruk av *en* sovemodus, men siden bruk av flere modi er for ekstra reduksjon i forhold til bruk av en sovemodus vil det i større grad kunne forsvares i større geometrier.

Det er vist at arealkostnaden forbundet med Power Gating av SoC må forventes minst å ligge på 30 %, og øke til ca 50 % ved bruk av tre sovetilstander for mikroprosessoren. Denne arealkostnaden medfører ikke nødvendigvis større dimensjoner på lengde/ bredde til chip, da celleplasseringen kan optimaliseres. Ytelsestapet som følge av implementeringer er ikke vurdert i denne rapporten. Det er derimot antatt at det vil ligge på ca 6,5 %.

Vedleggsoversikt

Vedlagt CD inneholder

- Datablad for ZPU, *zpu_arch.html*
- Opprinnelig ZPU
- GCC verktøykjede for ZPU
- Cygwin
- UPF standard versjon 1.0
- miniUART
- ROM Generator
- ZPU implementert med støtte for soving
    - Software
    - UPF konfigurasjon
    - VHDL filer
    - Simuleringsfiler for forskjellige sovemodi

Referanser

[1] *M. Keating, D. Flynn, R. Aitken, A. Gibbons, K. Shi, "Low Power Methodology Manual"*

[2] *K. Agarwal, H. Deogun , D. Sylvester , K. Nowka, "Power Gating with multiple sleep modes", IEEE 27-29 March 2006*

[3] *C. Long, J. Xiong, Y. Liu "Techniques of Power-Gating to kill sub- treshold leakage", IEEE 4-7 Dec. 2006 Page(s):952 – 955*

[4] *Moore's artikkel fra 19. April 1965: ftp://download.intel.com/research/silicon/moorespaper.pdf / pr. 27.05.2009*

[5] *von Arnim, K., m. fl., "Efficiency of body biasing in 90-nm CMOS for low-power digital circuits," IEEE Journal of Solid State Circuits, vol. 40, no. 7, July 2005, pp. 1549-1556*

[6] *Taiwan Semiconductor Manufacturing Company, "Fine Grain MTCMOS Design Methodology," TSMC Reference Flow Release 6.0, 2005.*

[7] *Calhoun, B., m.fl., "Power gating and dynamic voltage scaling," Leakage in Nanometer CMOS Technologies, S. Narendra and A. Chandraksan, editors, Springer, 2005.*

[8] *Martin, S., m.fl., "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads," proceedings of the International Conference on Computer-Aided Design, 2002, pp. 721-725.*

[9] Wei, L., m.fl., "Design and optimization of low voltage high performance dual threshold CMOS circuits," *proceedings of the Design Automation Conference*, 1998, pp. 489-494.

[10] Pakbaznia, Ehsan; Pedram, Massoud; "Design and application of multimodal power gating structures" Quality of Electronic Design, 2009. ISQED 2009. Quality of Electronic Design 16-18 March 2009 Page(s):120 - 126

[11] Kawaguchi, H., Nose, K., and Sakurai, T., "A super cut-off CMOS (SCCMOS) scheme for 0.5-V supply voltage with picoampere stand-by current," *IEEE Journal of Solid State Circuits*, vol. 35, no. 10, October 2000, pp. 1498-1501.

[12] Min, K., Kawaguchi, H., and Sakurai, T., "Zigzag super cut-off CMOS (ZSCCMOS) block activation with self-adaptive voltage level controller: an alternative to clock-gating scheme in leakage dominant era," *proceedings of the International Solid State Circuits Conference*, 2003, pp. 400-401.

[13] Lackey, D., m. fl., "Managing Power and Performance for System-on-Chip Designs using Voltage Islands," *proceedings of the International Conference on Computer-Aided Design*, 2002, pp. 192-202.

[14] Suhwan Kim, Kosonocky, S.V, Knebel, D.R., Stawiasz, K., Papaefthymiou, M.C.; "A Multi-Mode Power Gating Structure for Low-Voltage Deep-Submicron CMOS ICs" Circuits and Systems II: Express Briefs, IEEE Transactions on Volume 54, Issue 7, July 2007 Page(s):586 - 590

[15] C.Q. Tran, H. Kawaguchi, and T. Sakurai, "95% leakage-reduced FPGA using zigzag power-gating dual-Vth/VDD and micro-VDDhopping," A-SSCC, pp.149–152, 2005.

[16] UPF Standard v1.0: http://www.unifiedpowerformat.com/images/UPF.v1.0_Standard.pdf / pr. 27.05.09

[17] N.S Kim, T. Austin, D. Baauw, T. Mudge m.fl "*Leakage current: Moore's law meets static power" IEEE Computer vol. 36, no 12, Dec 200, pp.68 – 75*

[18] Won, H., m.fl., "An MTMCOS design methodology and its application to mobile computing," *proceedings of the International Symposium on Low Power Electronics and Design*, 2003, pp. 110-115.

[19] H.Jiang m.fl, *"Benefits and cost of Power Gating"* Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on, Oct 2005, pp 559 – 566

[20] Yngvar Berg, *Forelesningskompendium INF3400, kapittel 8,* Universitetet I Oslo, IFI, 2008.

Vedlegg

----------main.c----------

```
/*
 *Brukergenerert programfil. Benytter seg av headerfilen sleep.h for å muliggjøre soving
*/
#include <stdio.h>
#include "sleep.h"
int main()
{
printf("Sleep\n");
sleep(3, 100000);
printf("Rest\n");
 rest(4, 50000);
 printf("Hibernate\n");
 hibernate(4, 50000);
 printf("Done\n");
return(0);
}
```

----------sleep.h----------

```
/*
 *Headerfil. Inneholder globale variabler som benyttes i bibliotekfilen sleep.c
*/


#ifndef SLEEP_H
#define SLEEP_H

void sleep(int sleepmode, int sleeptime);
void rest(int sleepmode, int sleeptime);
void hibernate(int sleepmode, int sleeptime);

#endif      //NEWSLEEPDEF_H
```

----------sleep.c----------

```
/*
 *Bibliotekfil som inneholder den koden som blir utført når brukeren ber om at ZPU skal settes til å sove
*/
#include "sleep.h"
void main(){}
void sleep(int sleepmode, int sleeptime)
{
unsigned long *address;
int sleepdata = 0x01;                                   //verdi for sleep
long timerBuffer;
long NewTimerBuffer;

if (sleepmode== 1 || sleepmode==2 || sleepmode==3 || sleepmode==4){

        sleepdata=sleepdata+0x04; }
if (sleepmode== 1 || sleepmode==2 || sleepmode==5 || sleepmode==6){
        sleepdata=sleepdata+0x08; }
if (sleepmode== 1 || sleepmode==3 || sleepmode==5 || sleepmode==7){
        sleepdata=sleepdata+0x10; }
if (sleepmode== 1 || sleepmode==2 || sleepmode==3 || sleepmode==4){
                                                         //kun nødvendig å skrive til timeren når i bruk
        address = 0x080A0014;                           // Peker på adressen til timeren.
        *(address) = 0x02;                              //timeren skal bli samplet når bit[1] er lik 1.
        address = 0x080A0014;
        timerBuffer = *(address);                       // Deretter leses verdien til timeren ut
        NewTimerBuffer = timerBuffer + sleeptime + 400;
        address = 0x080A0018;                           // Peker på adressen til sammenlikningsregisteret
        *(address) = NewTimerBuffer;                    //Skriver oppvåkningstidspunkt
}
address = 0x080A2000;                                   //Peker på adressen til spenningskontrolleren
*(address) = sleepdata;          }                      //skriver sovedata til spenningskontrolleren
void rest(int sleepmode, int sleeptime)
{
unsigned long *address;
int sleepdata = 0x02;
long timerBuffer;
long NewTimerBuffer;
if (sleepmode== 1 || sleepmode==2 || sleepmode==3 || sleepmode==4)  {
        sleepdata=sleepdata+0x04; }
if (sleepmode== 1 || sleepmode==2 || sleepmode==5 || sleepmode==6)  {
        sleepdata=sleepdata+0x08; }
if (sleepmode== 1 || sleepmode==3 || sleepmode==5 || sleepmode==7)  {
        sleepdata=sleepdata+0x10; }
if (sleepmode== 1 || sleepmode==2 || sleepmode==3 || sleepmode==4)  {
```

```
        address = 0x080A0014;
        *(address) = 0x02;
        address = 0x080A0014;
        timerBuffer = *(address);
        NewTimerBuffer = timerBuffer + sleeptime + 400;
        address = 0x080A0018;
        *(address) = NewTimerBuffer;   }
address = 0x080A2000;
*(address) = sleepdata;            }

void hibernate(int sleepmode, int sleeptime) {
unsigned long *address;
int sleepdata = 0x03;
long timerBuffer;
long NewTimerBuffer;
if (sleepmode== 1 || sleepmode==2 || sleepmode==3 || sleepmode==4){
        sleepdata=sleepdata+0x04; }
if (sleepmode== 1 || sleepmode==2 || sleepmode==5 || sleepmode==6){
                sleepdata=sleepdata+0x08; }
if (sleepmode== 1 || sleepmode==3 || sleepmode==5 || sleepmode==7){
                sleepdata=sleepdata+0x10; }
if (sleepmode== 1 || sleepmode==2 || sleepmode==3 || sleepmode==4){
                address = 0x080A0014;
        *(address) = 0x02;
        address = 0x080A0014;
        timerBuffer = *(address);
        NewTimerBuffer = timerBuffer + sleeptime + 400;
        address = 0x080A0018;
        *(address) = NewTimerBuffer;   }
address = 0x080A2000;
*(address) = sleepdata;
}
```

A

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity powerstate is
                port(
                        clk : in std_logic;
                        areset : in std_logic;
                        testing : in std_logic;
                        sleep_enabled : out std_logic;
                        out_retain_n : out std_logic;
                        out_reset_n : out std_logic;
                        out_isol : out std_logic;
                        out_clock_gater : out std_logic;
                        ZPU_pwr : out std_logic;
                        U_clock_gater : out std_logic;
                        T_clock_gater : out std_logic;
                        sleep_timeout : in std_logic;
                        UART_Activity : in std_logic;
                        Interupt_detected : in std_logic;
                        test_ZPU_pwr : in std_logic;
                        test_UART_pwr : in std_logic;
                        test_TIMER_pwr : in std_logic;
                        powerdata : in std_logic_vector(4 downto 0);
                        we : in std_logic
);
end powerstate;

architecture behave of powerstate is

type State_Type is
(
State_idle,
State_sleep,
State_rest,
State_hibernate,
State_wakeup
);

signal state : State_Type;
signal nextstate : State_Type;
signal sleep_requested : std_logic;
signal rest_requested : std_logic;
signal hibernate_requested : std_logic;
signal wakeup_requested : std_logic;
signal wakeup_done : std_logic;
signal isol : std_logic;
signal retain_n  : std_logic;
signal clock_gater : std_logic;
signal reset_n : std_logic;
```

```
signal sleep_mode : std_logic_vector(1 downto 0);
signal T_retain_n : std_logic;
signal T_isol : std_logic;
signal T_reset_n : std_logic;
signal T_pwr_req : std_logic;
signal U_retain_n : std_logic;
signal U_isol : std_logic;
signal U_reset_n : std_logic;
signal U_pwr_req : std_logic;
signal timer_trigged : std_logic;
signal UART_trigged : std_logic;
signal Interupt_trigged : std_logic;
signal pwr_ack : std_logic;
signal pwr_req : std_logic;
signal pwr_low_req : std_logic;
signal pwr_reduced_req : std_logic;
signal power_counter : unsigned(4 downto 0);

begin

sleep_mode <= powerdata(1 downto 0);
out_retain_n <= retain_n;
out_isol  <= isol;
out_reset_n <= reset_n;
out_clock_gater <= clock_gater;
ZPU_pwr <= pwr_ack or pwr_reduced_req;

        State_executer:
        process(clk, areset, we, sleep_mode, sleep_timeout, UART_activity, Interupt_detected)

begin
                if (areset = '1') then
                        sleep_enabled <= '0';
                        pwr_req <= '1';
                        pwr_low_req <= '0';
                        pwr_reduced_req <= '0';
                        isol <= '0';
                        clock_gater <= '0';
                        retain_n <= '1';
                        reset_n <= '1';
                        state <= State_idle;
                        Timer_trigged <= '0';
                        UART_trigged <= '0';
                        Interupt_trigged <= '0';
                        sleep_requested <= '0';
                        rest_requested <= '0';
                        hibernate_requested <= '0';
                        wakeup_requested <= '0';
                        wakeup_done <= '0';

                elsif (clk'event and clk = '1') then
                        state <= nextstate;
```

```
                if testing = '1' then
                        isol <= '0
                        retain_n <= '1';
                        reset_n <= '1';
                        clock_gater <= '0';
                        pwr_req <= test_ZPU_pwr;
                        U_pwr_req <= test_UART_pwr;
                        T_pwr_req <= test_TIMER_pwr;
                else

if (wakeup_requested = '1') then
                        pwr_req <= '1';                 --Ber om driftsspenning
                        pwr_low_req <= '0';
                        pwr_reduced_req <= '0';

                        if pwr_ack = '1' then
        -- skal sikre at spenning er kommet helt på for å få en sikker oppvåkning.
                                reset_n <= '1';
                                if reset_n = '1' then
                                        retain_n <= '1';
                                        if retain_n = '1' then
                                                isol <= '0';
                                                if isol = '0' then
                                                        clock_gater <= '0';
                                                        if clock_gater = '0' then
                                                                wakeup_done <= '1';
                                                                if wakeup_done = '1' then
                                                                  wakeup_done <= '0';
                                                                  wakeup_requested <= '0';
                                                                  sleep_enabled <= '0';
                                                                  sleep_requested <= '0';
                                                                  rest_requested <= '0';
                                                                  hibernate_requested <= '0';
                                                                  Timer_trigged <= '0';
                                                                  UART_trigged <= '0';
                                                                  Interupt_trigged <= '0';
                                                                end if;
                                                        end if;
                                                end if;
                                        end if;
                                end if;
```

```
elsif (sleep_requested = '1') then
        clock_gater <= '1';
        if clock_gater = '1' then
                isol <= '1';
                if isol = '1' then
                        retain_n <= '0';
                        if retain_n = '0' then
                                reset_n <= '0';
                                if reset_n = '0' then
                                        pwr_req <= '0';
                                        pwr_low_req <= '0';
                                        pwr_reduced_req <= '0';
                                        sleep_requested <= '0';
                                        sleep_enabled <= '1';
                                end if;
                        end if;
                end if;
        end if;


elsif (hibernate_requested = '1') then
        clock_gater <= '1';
        if clock_gater = '1' then
                isol <= '1';
                if isol = '1' then
                        retain_n <= '0';
                        if retain_n = '0' then
                                reset_n <= '0';
                                if reset_n = '0' then
                                        pwr_low_req <= '1';
                                        pwr_req <= '0';
                                        pwr_reduced_req <= '0';
                                        hibernate_requested <= '0';
                                        sleep_enabled <= '1';
                                end if;
                        end if;
                end if;
        end if;


elsif (rest_requested = '1') then
        clock_gater <= '1';
        if clock_gater = '1' then
                isol <= '1';
                if isol = '1' then
                        pwr_reduced_req <= '1';
                        pwr_req <= '0';
                        pwr_low_req <= '0';
                        rest_requested <= '0';
                        sleep_enabled <= '1';
                end if;
```

```
            end if;
end if;

if (we = '1') then          --Spenningskontrolleren mottar instruksjoner.

            Timer_trigged <= powerdata(2);
            UART_trigged <= powerdata(3);
            Interupt_trigged <= powerdata(4);

            if powerdata(4 downto 2) = "000" then
                        report "No wakeup triggers selected" severity warning;
            end if;

            if sleep_mode = "01" then
                        sleep_requested <= '1';
            elsif sleep_mode= "10" then
                        rest_requested <= '1';
            elsif sleep_mode= "11" then
                        hibernate_requested <= '1';
            else
                        report "Invalid sleepmode selected" severity warning;
            end if;


elsif (sleep_timeout = '1') then
                        if Timer_trigged = '1' then
                                    sleep_requested <= '0';
                                    rest_requested <= '0';
                                    hibernate_requested <= '0';
                                    wakeup_requested <= '1';
                        end if;

elsif (UART_activity = '1') then
                        if UART_trigged = '1' then
                                    sleep_requested <= '0';
                                    rest_requested <= '0';
                                    hibernate_requested <= '0';
                                    wakeup_requested <= '1';
                        end if;

elsif (Interupt_detected = '1') then
                        if Interupt_trigged = '1' then
                                    sleep_requested <= '0';
                                    rest_requested <= '0';
                                    hibernate_requested <= '0';
                                    wakeup_requested <= '1';
                        end if;
end if;


if timer_trigged = '1' then         --Timer er i bruk. Skal ikke sove med ZPU
```

```
                        T_retain_n <= '1';
                        T_isol <= '0';
                        T_reset_n <= '1';
                        T_pwr_req <= '1';
                        T_clock_gater <= '0';
                    else                                    --Timer er ikke i bruk
                        T_retain_n <= retain_n;             --Settes til å sove med ZPU
                        T_isol <= isol;
                        T_reset_n <= reset_n;
                        T_pwr_req <= pwr_req;
                        T_clock_gater <= clock_gater;
                    end if;

                    if UART_trigged = '1' then
                        U_retain_n <= '1';
                        U_isol <= '0';
                        U_reset_n <= '1';
                        U_pwr_req <= '1';
                        U_clock_gater <= '0';
                    else                                                --UART ikke benyttet.
                        U_retain_n <= retain_n;
                        U_isol <= isol;
                        U_reset_n <= reset_n;
                        U_pwr_req <= pwr_req;
                        U_clock_gater <= clock_gater;
                    end if;
                end if;
        end if;
end process;


state_updater:
process(areset, clk, state, pwr_req, pwr_low_req, pwr_reduced_req, wakeup_done)

begin
        if areset = '1' then
                pwr_ack <= '1';
                nextstate <= State_idle;
        end if;
                case state is
                when State_idle =>

                    if pwr_req = '0' and pwr_low_req = '0' and pwr_reduced_req = '0' then
                        nextstate <= state_sleep;
                    elsif pwr_req = '0' and pwr_low_req = '1' and pwr_reduced_req = '0' then
                        nextstate <= state_hibernate;
                    elsif pwr_req = '0' and pwr_low_req = '0' and pwr_reduced_req = '1' then
                        nextstate <= state_rest;
                    end if;

                when State_sleep =>
```

```
                pwr_ack <= '0';                                 --ikke lenger driftspenning

                if pwr_req = '1' then
                        nextstate <= state_wakeup;
                end if;

        when State_rest =>
                pwr_ack <= '0';
                if pwr_req = '1' then
                        nextstate <= state_wakeup;
                end if;

        when State_hibernate =>
                pwr_ack <= '0';
                if pwr_req = '1' then
                        nextstate <= state_wakeup;
                end if;

        when State_wakeup =>
                if power_counter= "10000" then                  --power is on
                        pwr_ack <= '1';
                end if;

                if wakeup_done = '1' then
                        nextstate <= state_idle;
                end if;

        when others =>
                null;
        end case;

end process;

Powerbar: process(areset, clk)
begin
if areset = '1' then

elsif clk'event and clk = '1' then
        case state is

                when State_idle =>
                        power_counter <= "10000";               --full driftspenning

                when State_sleep =>
                        power_counter <= "00000";

                when State_rest =>
                        power_counter <= "01000";

                when State_hibernate =>
                        power_counter <= "00100";
```

```
                        when State_wakeup =>
                                if power_counter < "10000" then
                                        power_counter <= power_counter + 1;
                                end if;

        end case;
end if;

end process;

end behave;
```

```
set_scope fpga_top

############ Power domains #################
create_power_domain CORE \
-elements {zpu} \
-ce os

############ Toplevel Connections ######
# VDD_HIGH (1.0V)
create_supply_net VDD -domain CORE
create_supply_net SW_VDD -domain CORE
create_supply_port VDD          -domain CORE
connect_supply_net VDD -ports {VDD}

# VDD_REDUCED (0.864V)
create_supply_port VDD_REDUCED -domain CORE
create_supply_net VDD_REDUCED -domain CORE
connect_supply_net VDD_REDUCED -ports VDD_REDUCED

# VDD_LOW (0.328V)
create_supply_port VDD_LOW -domain CORE
create_supply_net VDD_LOW -domain CORE
connect_supply_net VDD_LOW -ports VDD_LOW

# VSS (0.0V)
create_supply_port VSS -domain CORE
create_supply_net VSS -domain CORE
connect_supply_net VSS -ports VSS

############ Establish connections #################
set_domain_supply_net CORE \
-primary_power_net SW_VDD \
-primary_ground_net VSS

create_power_switch CORE_switch \
-domain CORE \
-input_supply_port {CORE_input_port VDD} \
-input_supply_port {CORE_reduced_input_port VDD_REDUCED} \
-input_supply_port {CORE_low_input_port VDD_LOW} \
-output_supply_port {CORE_sw_output_port SW_VDD} \
-control_port {pwr_off_cntrl iomap/powerinst/pwr_req} \
-control_port {pwr_reduced_cntrl iomap/powerinst/pwr_reduced_req} \
-control_port {pwr_low_cntrl iomap/powerinst/pwr_low_req} \
```

```
-on_state {pwr_on CORE_input_port {pwr_off_cntrl == 1  & pwr_reduced_cntrl == 0 & pwr_low_cntrl
== 0}} \
-on_state {pwr_reduced CORE_reduced_input_port {pwr_off_cntrl == 0  & pwr_reduced_cntrl == 1 &
pwr_low_cntrl == 0}} \
-on_partial_state {pwr_low CORE_low_input_port {pwr_off_cntrl == 0  & pwr_reduced_cntrl == 0 &
pwr_low_cntrl == 1}} \
-off_state {pwr_off  {pwr_off_cntrl == 0 & pwr_reduced_cntrl == 0 & pwr_low_cntrl == 0}}

set_isolation CORE_isolation \
-domain CORE \
-isolation_power_net VDD \
-isolation_ground_net VSS \
-clamp_value 0 \
-applies_to outputs

set_isolation_control CORE_isolation \
-domain CORE \
-isolation_signal iomap/isol

set_retention CORE_retention \
-domain CORE \
-retention_power_net VDD \
-retention_ground_net VSS \
-elements {zpu/pc zpu/sp}

set_retention_control CORE_retention \
        -domain CORE \
        -save_signal {iomap/retain_n negedge} \
        -restore_signal {iomap/retain_n posedge}
```

```
set_scope fpga_top
############ Power domains ################
create_power_domain TOP \
        -elements{powerinst}

create_power_domain CORE \
        -elements {zpu} \
        -ce os

create_power_domain UART \
        -elements {iomap/UART} \
        -ce os

create_power_domain TIMER \
        -elements {iomap/timerinst} \
        -ce o

############ Toplevel Connections ######
# VDD_HIGH (1.0V)
create_supply_port VDD
create_supply_net VDD -domain TOP
create_supply_net VDD -domain CORE -reuse
create_supply_net VDD -domain UART -reuse
create_supply_net VDD -domain TIMER -reuse
connect_supply_net VDD -ports VDD

# VDD_REDUCED (0.864V)
create_supply_port VDD_REDUCED
create_supply_net VDD_REDUCED -domain TOP
create_supply_net VDD_REDUCED -domain CORE -reuse
connect_supply_net VDD_REDUCED -ports VDD_REDUCED

# VDD_LOW (0.364V)
create_supply_port VDD_LOW
create_supply_net VDD_LOW -domain TOP
create_supply_net VDD_LOW -domain CORE -reuse
connect_supply_net VDD_LOW -ports VDD_LOW

# VSS (0.0V)
create_supply_port VSS
create_supply_net VSS -domain TOP
create_supply_net VSS -domain CORE -reuse
create_supply_net VSS -domain UART -reuse
create_supply_net VSS -domain TIMER -reuse
connect_supply_net VSS -ports VSS
```

```
## Switch Net ##
create_supply_net SW_VDD -domain CORE
create_supply_net SW_UART_VDD -domain UART
create_supply_net SW_TIMER_VDD -domain TIMER

############ Establish connections #################

set_domain_supply_net TOP \
        -primary_power_net VDD \
        -primary_ground_net VSS

set_domain_supply_net CORE \
        -primary_power_net SW_VDD \
        -primary_ground_net VSS

set_domain_supply_net UART \
        -primary_power_net SW_UART_VDD \
        -primary_ground_net VSS

set_domain_supply_net TIMER \
        -primary_power_net SW_TIMER_VDD \
        -primary_ground_net VSS

create_power_switch CORE_switch \
        -domain CORE \
        -input_supply_port {CORE_input_port VDD} \
        -input_supply_port {CORE_reduced_input_port VDD_REDUCED} \
        -input_supply_port {CORE_low_input_port VDD_LOW} \
        -output_supply_port {CORE_sw_output_port SW_VDD} \
        -control_port {pwr_off_cntrl iomap/powerinst/pwr_req} \
        -control_port {pwr_reduced_cntrl iomap/powerinst/pwr_reduced_req} \
        -control_port {pwr_low_cntrl iomap/powerinst/pwr_low_req} \
        -on_state {pwr_on CORE_input_port {pwr_off_cntrl == 1  & pwr_reduced_cntrl == 0 &
        pwr_low_cntrl == 0}} \
        -on_state {pwr_reduced CORE_reduced_input_port {pwr_off_cntrl == 0  &
        pwr_reduced_cntrl == 1 & pwr_low_cntrl == 0}} \
        -on_partial_state {pwr_low CORE_low_input_port {pwr_off_cntrl == 0  & pwr_reduced_cntrl
        == 0 & pwr_low_cntrl == 1}} \
        -off_state {pwr_off  {pwr_off_cntrl == 0 & pwr_reduced_cntrl == 0 & pwr_low_cntrl == 0}}

create_power_switch UART_switch \
        -domain UART \
        -input_supply_port {UART_input_port VDD} \
        -output_supply_port {UART_sw_output_port SW_UART_VDD} \
        -control_port {UART_off_cntrl iomap/powerinst/U_pwr_req} \
```

```
        -on_state {pwr_on UART_input_port {UART_off_cntrl == 1 }} \
        -off_state {pwr_off {UART_off_cntrl == 0}}

create_power_switch TIMER_switch \
        -domain TIMER \
        -input_supply_port {TIMER_input_port VDD} \
        -output_supply_port {TIMER_sw_output_port SW_TIMER_VDD} \
        -control_port {TIMER_off_cntrl iomap/powerinst/T_pwr_req} \
        -on_state {pwr_on TIMER_input_port {TIMER_off_cntrl == 1 }} \
        -off_state {pwr_off {TIMER_off_cntrl == 0}}

set_isolation CORE_isolation \
        -domain CORE \
        -isolation_power_net VDD \
        -isolation_ground_net VSS \
        -clamp_value 0 \
        -applies_to outputs

set_isolation_control CORE_isolation \
        -domain CORE \
        -isolation_signal {iomap/powerinst/isol}

set_isolation UART_isolation \
        -domain UART \
        -isolation_power_net VDD \
        -isolation_ground_net VSS \
        -clamp_value 0 \
        -applies_to outputs

set_isolation_control UART_isolation \
        -domain UART \
        -isolation_signal {iomap/powerinst/U_isol}

set_isolation TIMER_isolation \
        -domain TIMER \
        -isolation_power_net VDD \
        -isolation_ground_net VSS \
        -clamp_value 0 \
        -applies_to outputs

set_isolation_control TIMER_isolation \
        -domain TIMER \
        -isolation_signal {iomap/powerinst/T_isol}
```

```
set_retention CORE_retention \
        -domain CORE \
        -retention_power_net VDD \
        -retention_ground_net VSS \
        -elements {zpu/pc zpu/sp}

set_retention_control CORE_retention \
        -domain CORE \
        -save_signal {iomap/powerinst/retain_n negedge} \
        -restore_signal {iomap/powerinst/retain_n posedge}

set_retention UART_retention \
        -domain UART \
        -retention_power_net VDD \
        -retention_ground_net VSS \
        -elements {iomap/UART}

set_retention_control UART_retention \
        -domain UART \
        -save_signal {iomap/powerinst/U_retain_n negedge} \
        -restore_signal {iomap/powerinst/U_retain_n posedge}

set_retention TIMER_retention \
        -domain TIMER \
        -retention_power_net VDD \
        -retention_ground_net VSS \
        -elements {iomap/timerinst}

set_retention_control TIMER_retention \
        -domain TIMER \
        -save_signal {iomap/powerinst/T_retain_n negedge} \
        -restore_signal {iomap/powerinst/T_retain_n posedge}
```

Command Syntax	Description and Example

add_domain_elements *domain_name*
-elements *list*
	Add design elements to a power domain
	`add_domain_elements U1/PD1`
	`  -elements {U1/U2/foo1 U1/U2/foo2}`

add_port_state *port_name*
{**-state** {*name* <*nom* | <*min nom max*> | **off**>}}*
	Add state to a port
	`add_port_state VN1`
	`  -state {active_state 0.88 0.90 0.92}`
	`  -state {off_state off}`

add_pst_state *state_name*
-pst *table_name* **-state** *supply_states*
	Define the states of each of the supply nets for one possible state of the design
	`create_pst pt -supplies { PN1 PN2 SOC/OTC/PN3 }`
	`add_pst_state s1 -pst pt -state { s08 s08 s08 }`
	`add_pst_state s2 -pst pt -state { s08 s08 off }`
	`add_pst_state s3 -pst pt -state { s08 s09 off }`

bind_checker *instance_name*
-module *checker_name* **-elements** *list*
[**-ports** {{*port_name net_name*}*}]
	Inserts checker modules and binds them to design elements
	`bind_checker chk_p_clks -module assert_partial_clk`
	`  -elements {U1/U2 U1/U3}`
	`  -ports {{prt1 clknet2} {port3 net4}}`

connect_supply_net *net_name*
[**-ports** *list*] [**-pins** *list*]
[<**-cells** *list* | **-domain** *domain_name*>]
[<**-rail_connection** *rail_type* |
-pg_type *pg_type*>]* [**-vct** *vct_name*]
	Connect a supply net to supply ports and/or pins
	`connect_supply_net v09 -ports {VDD U18/v9 U21/v9}`
	`connect_supply_net pd1_vdd -ports pll_inst/vdd`
	`  vct upf2vlog_vdd`

create_hdl2upf_vct *vct_name*
-hdl_type {<**vhdl** | **vlog** | **SV**> [*typename*]}
-table {{*from_value to_value*}*}
	Define value conversion table that can be used in converting HDL logic values into net_state_type values
	`create_hdl2upf_vct vlog2upf_vss -hdl_type vlog`
	`  -table {{X OFF} {0 ON} {I OFF} {Z PARTIAL ON}}`

create_power_domain *domain_name*
[**-elements** *list*] [**-include_scope**]
[**-scope** *instance_name*]
	Define a power supply distribution network for a set of design elements
	`create_power_domain PD1 -elements {top/U1}`
	`set_scope /top/U1`
	`create_power_domain PD2`

create_power_switch *switch_name*
-domain *domain_name*
-output_supply_port {*port_name*
supply_net_name}
{**-input_supply_port** {*port_name*
supply_net_name}}*
{**-control_port** {*port_name net_name*}}*
{**-on_state** {*state_name input_supply_port*
{*boolean_function*}}}*
[**-on_partial_state** {*state_name*
input_supply_port {*boolean_function*}}]*
[**-ack_port** {*port_name net_name*
[{*boolean_function*}]}]*
[**-ack_delay** {*port_name delay*}]*
[**-off_state** {*state_name* {*boolean_function*}}]*
[**-error_state** {*state_name* {*boolean_function*}}]*
	Define a switch in the power domain
	`create_power_switch sw1 -domain PD_SODIUM`
	`  -output_supply_port {vout VN3}`
	`  -input_supply_port {vin1 VN1}`
	`  -input_supply_port {vin2 VN2}`
	`  -control_port {ctrl_small ON1}`
	`  -control_port {ctrl_large ON2}`
	`  -control_port {ss SUPPLY_SELECT}`
	`  -on_state {partial_s1 vin1`
	`  {ctrl_small & !ctrl_large & ss}}`
	`  -on_state {full_s1 vin1`
	`  {ctrl_small & ctrl_large & ss}}`
	`  -on_state {partial_s2 vin2`
	`  {ctrl_small & !ctrl_large & !ss}}`
	`  -on_state {full_s2 vin2`
	`  {ctrl_small & ctrl_large & !ss}}`
	`  -error_state {no_small`
	`  {!ctrl_small & ctrl_large}}`

create_pst *table_name* **-supplies** *list*	Create power state table with a specific ordering of supply nets
	`create_pst MyPowerStateTable`
	`  -supplies {PN1 PN2 SOC/OTC/PN3}`

create_supply_net *net_name* **-domain** *domain_name* [**-reuse**] [**-resolve** <**unresolved** | **one_hot** | **parallel**>]	Create a power or ground supply net `create_supply_net v09 -domain PD1`
create_supply_port *port_name* [**-domain** *domain_name*] [**-direction** <**in** |**out**>]	Create a port on a power domain `create_supply_port VN1 -domain PD1`

Command Syntax	Description and Example

create_upf2hdl *vct_name*
-hdl_type {<**vhdl** | **vlog** | **SV**> [*typename*]}
-table {{*from_value to_value*}*}
	Define value conversion table that can be used in converting UPF supply_net_type.state values into HDL logic values
	
	`create_upf2hdl_vct upf2vlog_vdd -hdl_type {vlog}`
	`-table {{OFF X} {ON 1} {PARTIAL_ON 0}}`

get_supply_net *net_name* [**-domain** *domain_name*] [**-scope** *scope_name*]	Return logical net name for this supply net in the given scope `get_supply_net VDD -domain mySoC_PD -scope /camera_i`
load_upf *upf_file_name* [**-scope** *instance_name*] [**-version** *string*]	Set the scope to the specified instance and execute the specified UPF commands `load_upf my_upf -scope U1`

map_isolation_cell *isolation_name*
-domain *domain_name* [**-elements** *list*]
[**-lib_cells** *list*] [**-lib_cell_type** *lib_cell_type*]
[**-lib_model_name** *lib_model_name*
{**-port** {*port_name net_name*}}*]
	Map a particular isolation strategy to a library cell or range of library cells.
	
	`map_isolation_cell test_PD1`
	`-domain PD1 -lib_cell_type jims_iso_fast`

map_level_shifter_cell *level_shifter_name*
-domain *domain_name*
-lib_cells *list* [**-elements** *list*]
	Map a particular level shifter strategy to a library cell or range of library cells
	
	`map_level_shifter_cell shift_up -domain PwrDomZ`
	`-lib_cells {/lib2/LS_LH /lib2/LS_HL}`

map_power_switch *switch_name*
domain *domain_name* **lib_cells** *list*
	Specify which power switch cell is to be used for the corresponding switch instance
	
	`map_power_switch switch_1A`
	`-domain myPowerDomain -lib cells /lib2A/switch2A`

map_retention_cell *retention_name*
-domain *domain_name* [**-elements** *list*]
[**-lib_cells** *list*] [**-lib_cell_type** *lib_cell_type*]
[**-lib_model_name** *lib_cell_name*
{**-port** *port_name net_name*}*]
	Specify which retention cell is to be used for the retention registers in a power domain
	
	`map_retention_cell test_PDA`
	`-domain {PowerDomainX} -elements {foo/U1 foo/U2}`
	`-lib model name RETFF7 -port U1 N1`

merge_power_domains *new_domain_name* **-power_domains** *list* [**-scope** *instance_name*] [**-all_equivalent**]	Merge two or more existing power domains into a single, new power domain `merge_power_domains PD9 -power_domains {PD9A PD9B PD9C}`

name_format
[**-isolation_prefix** *string*]
[**-isolation_suffix** *string*]
[**-level_shift_prefix** *string*]
[**-level_shift_suffix** *string*]
	Allow user guidance in constructing port and signal names related to isolation or level shifter cells
	
	`name_format`
	`-isolation_prefix "MY_ISO_" -isolation_suffix ""`

save_upf *upf_file_name* [**-scope** *instance_name*] [**-version** *string*]	Create a UPF file relative to the specified scope `save_upf my_saved_file_Jan14`
set_design_top *instance*	Specify the top-level design instance `set_design_top ALU07`

set_domain_supply_net
domain_name
-primary_power_net *supply_net_name*
-primary_ground_net *supply_net_name*
	Set the default power and ground supply nets for a power domain
	`set_domain_supply_net PD1`
	`-primary_power_net PG1 -primary_ground_net PG0`

set_isolation *isolation_name*
-domain *domain_name*
<**-isolation_power_net** *net_name* |
-isolation_ground_net *net_name* |
-isolation_power_net *net_name*
-isolation_ground_net *net_name* |
-no_isolation>
[**-elements** *list*] [**-clamp_value** <**0** | **1** | *latch* | **Z**>]
[**-applies_to** <**inputs** | **outputs** | **both**>]
	Specify the elements in the domain to isolate using the specified strategy
	
	`set_isolation outputs_only`
	`-domain PD1`
	`-isolation_power_net VDDbackup`
	`-clamp_value 1`
	`-applies_to outputs`

Command Syntax	Description and Example

set_isolation_control *isolation_name*
-domain *domain_name*
-isolation_signal *signal_name*
[**-isolation_sense** <**high** | **low**>]
[**-location** <**self** | **parent** | **sibling** |
fanout | **automatic**>]
	Specify the control signals for a previously defined isolation strategy
	
	`set_isolation outputs_only`
	`  -domain PD1 -isolation_power_net VDDbackup`
	`  -clamp_value 1 -applies_to outputs`
	`set_isolation_control outputs_only`
	`  -domain PD1 -isolation_signal cpu_iso`
	`  -isolation_sense low -location parent`

set_level_shifter *level_shifter_name*
-domain *domain_name*
[**-elements** *list*] [**-threshold** *value*]
[**-applies_to** <**inputs** | **outputs** | **both**>]
[**-rule** <**low_to_high** | **high_to_low** | **both**>]
[**-location** <**self** | **parent** | **sibling** |
fanout | **automatic**>] [**-no_shift**]
	Specify a level shifter strategy
	
	`set_level_shifter shift_up`
	`  -domain PowerDomain2 -applies_to outputs`
	`  -threshold 0.02 -rule both`

set_pin_related_supply *library_cell* **-pins** *list*
-related_power_pin *supply_pin*
-related_ground_pin *supply_pin*
	Define the related power/ground pair for a library cell
	
	`set_pin_related_supply library1/cell1 -pins {A B C}`
	`  -related_power_pin VDDX -related_ground_pin VSSX`

set_power_switch *switch_name*
-output_supply_port
{*port_name supply_net_name*}
{**-input_supply_port**
{*port_name supply_net_name*}}*
{**-control_port** {*port_name net_name*}}*
{**-on_state** {*state_name input_supply_port*
{*boolean_function*}}}*
[**-on_partial_state** {*state_name*
input_supply_port {*boolean_function*}}]*
[**-off_state** {*state_name* {*boolean_function*}}]*
[**-error_state** {*state_name* {*boolean_function*}}]*
	Extend a switch by adding the input supply port(s) and output supply port to the switch
	
	`set_power_switch hmacro/sw1`
	`  -input_supply_port {i1 always_on_power}`
	`  -output_supply_port {o1 switched_power}`
	`  -control_port {on}`
	`  -on_state {sw1_on on {on}}`
	`  -off_state {sw1_off on {~on}}`

set_retention *retention_name*
-domain *domain_name*
<**-retention_power_net** *net_name* |
-retention_ground_net *net_name* |
-retention_power_net *net_name*
-retention_ground_net *net_name*>
[**-elements** *list*]
	Specify which registers in the domain need to be retention registers and set the save and restore signals for the retention functionality
	
	`set_retention my_retention -domain PDA`
	`  -retention_power_net volt_high`

set_retention_control *retention_name*
-domain *domain_name*
-save_signal {{*net_name*
<**high** | **low** | **posedge** | **negedge**>}}
-restore_signal {{*net_name*
<**high** | **low** | **posedge** | **negedge**>}}
[**-assert_r_mutex** {{*net_name*
<**high** | **low** | **posedge** | **negedge**>}}]*
[**-assert_s_mutex** {{*net_name*
<**high** | **low** | **posedge** | **negedge**>}}]*
[**-assert_rs_mutex** {{*net_name*
<**high** | **low** | **posedge** | **negedge**>}}]*
	Specify the control signals and assertions for a previously defined retention strategy
	
	`set_retention my_retention_strategy -domain PDA`
	`set_retention_control my_retention_strategy`
	`  -domain PDA`
	`  -save_signal {power_ctrl_inst/save_l high}`
	`  -restore_signal {power_ctrl_inst/restore_l low}`
	`  -assert_rs_mutex {clock_a posedge}`

set_scope *instance*	Specify the current UPF scope
	
	`set_scope foo/bar`
	`set_scope ..`

upf_version [*string*]	Specify the version for the UPF file/syntax `upf_version 1.0`

Vedlegg 6 – Opprinnelig og ny adressering i io.vhd

Opprinnelig adressering :

```
timer_we <= writeEnable and addr(12);
```

Ny adressering :

```
If write_enable = ‘1’ then
        if ("1" & addr & lowAddrBits)=x"80a0014" and writeEnable = '1' then
                timer_we <= '1';
        end if;
end if;
```

```
library IEEE,work;
use IEEE.Std_Logic_1164.all;
--
entity ClkGater is
  port (
        Clk : in Std_logic;
        ZPUClk_gater : in std_logic;
        UARTClk_gater : in std_logic;
        TIMERClk_gater : in std_logic;
        ZPUclk : out std_logic;                    --ZPU clock
        UClock : out std_logic;                    --UART clock
        Tclock : out std_logic);                   --Timer clock
end ClkGater;

architecture Behaviour of ClkGater is
begin

ZPUClock: process(clk, ZPUClk_gater)
        begin
                if (ZPUClk_gater = '0') then
                        ZPUclk <= clk;
                elsif clk'event and clk = '0' then
                        ZPUclk <= '0';
                end if;

end process;

UARTClock: process(clk, UARTClk_gater)
        begin
                if (UARTClk_gater = '0') then
                        Uclock <= clk;
                elsif clk'event and clk = '0' then
                        Uclock <= '0';
                end if;
end process;

TIMERClock: process(clk, TIMERClk_gater)
        begin
                if (TIMERClk_gater = '0') then
                        Tclock <= clk;
                elsif clk'event and clk = '0' then
                        Tclock <= '0';
                end if;

end process;

end Behaviour;
```

Report Datapath Area

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** May 30 2009 14:01:27
- **Module:** powerstate **(en prosess for tre komponenter)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	0.00	0.00
external	0.00	0.00
others	307.50	100.00
TOTAL	307.50	100.00

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** May 30 2009 14:02:42
- **Module:** powerstate2 **(2 prosesser for 2 blokker)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	0.00	0.00
external	0.00	0.00
others	331.17	100.00
TOTAL	331.17	100.00

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** May 30 2009 14:03:21
- **Module:** powerstate3 **(3 prosesser for 3 blokker)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	0.00	0.00
external	0.00	0.00
others	363.36	100.00
TOTAL	363.36	100.00

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** May 30 2009 14:05:22
- **Module:** powerstate4 **(kun SLEEP mode støtte)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	0.00	0.00
external	0.00	0.00
others	231.95	100.00
TOTAL	231.95	100.00

Report Power

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:36:04
- **Module:** powerstate
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** slow (balanced_tree) **(Høy V_T)**
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
powerstate	214	1319.89	10603.47	2615.45	13218.92

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:45:20
- **Module:** powerstate
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree) **(Standard V_T)**
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
powerstate	214	2136.34	14398.52	3867.16	18265.68

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:17:01
- **Module:** powerstate
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** fast (balanced_tree) **(Lav V_T)**
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
powerstate	214	7075.35	19639.63	5096.09	24735.72

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:38:07
- **Module:** powerstate3 **(Tre prosesser for tre blokker)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** slow (balanced_tree)
- **Wireload mode:** enclosed

Report Power

Instance	Cells	Leakage (nW)	Internal (nW)	Net (nW)	Switching (nW)
powerstate3	224	1328.44	9039.07	2125.84	11164.91

Report Datapath Area

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 01 2009 13:34:25
- **Module:** zpu_core_big
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** fast (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	3115.66	40.76
external	0.00	0.00
others	4527.85	59.24
TOTAL	7643.51	100.00

Report Power

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 11:56:53
- **Module:** zpu_core_big
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** fast (balanced_tree)
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
zpu_core_big	3870	165443.08	629460.57	225991.41	855451.97
zpu_core_big/add_504_18	17	1479.44	5619.69	1033.98	6653.67
zpu_core_big/add_758_25	33	3590.82	11890.91	2023.44	13914.35
zpu_core_big/add_772_96	15	1478.52	5183.44	948.83	6132.27
zpu_core_big/add_845_24	33	3587.25	11361.50	1919.92	13281.42
zpu_core_big/inc_add_211_14	36	974.13	1267.03	300.00	1567.03
zpu_core_big/inc_add_268_17	42	1088.28	1683.54	516.80	2200.34
zpu_core_big/le_650_31	101	2359.89	5500.97	1697.27	7198.23
zpu_core_big/le_672_39	101	2359.89	5547.83	1707.81	7255.64
zpu_core_big/lt_639_31	101	2273.54	5697.91	1760.94	7458.84

Report Mapped Gates

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 14 2009 15:25:55
- **Module:** zpu_core_big
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** fast (balanced_tree)
- **Wireload mode:** enclosed

Report Mapped Gates

Gate	Instances	Area	Library v
AND2_X2	12	12.77	NangateOpenCellLibrary_PDKv1_2_v2008_10
AND2_X4	41	43.62	NangateOpenCellLibrary_PDKv1_2_v2008_10
AND3_X2	3	3.99	NangateOpenCellLibrary_PDKv1_2_v2008_10
AND3_X4	4	5.32	NangateOpenCellLibrary_PDKv1_2_v2008_10
AND4_X4	2	3.19	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI211_X1	17	22.61	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI21_X1	156	165.98	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI21_X2	1	1.06	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI221_X1	71	113.32	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI221_X2	1	1.60	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI222_X1	78	165.98	NangateOpenCellLibrary_PDKv1_2_v2008_10
AOI22_X1	632	840.56	NangateOpenCellLibrary_PDKv1_2_v2008_10
BUF_X4	14	11.17	NangateOpenCellLibrary_PDKv1_2_v2008_10
DFFR_X1	2	11.17	NangateOpenCellLibrary_PDKv1_2_v2008_10
DFFR_X2	85	474.81	NangateOpenCellLibrary_PDKv1_2_v2008_10
DFFS_X2	15	83.79	NangateOpenCellLibrary_PDKv1_2_v2008_10
DFF_X2	136	651.17	NangateOpenCellLibrary_PDKv1_2_v2008_10
FA_X1	336	1608.77	NangateOpenCellLibrary_PDKv1_2_v2008_10
HA_X1	24	70.22	NangateOpenCellLibrary_PDKv1_2_v2008_10
INV_X2	236	125.55	NangateOpenCellLibrary_PDKv1_2_v2008_10
INV_X4	233	123.96	NangateOpenCellLibrary_PDKv1_2_v2008_10
MUX2_X2	5	9.31	NangateOpenCellLibrary_PDKv1_2_v2008_10
NAND2_X1	315	251.37	NangateOpenCellLibrary_PDKv1_2_v2008_10
NAND2_X2	1	0.80	NangateOpenCellLibrary_PDKv1_2_v2008_10
NAND2_X4	1	1.33	NangateOpenCellLibrary_PDKv1_2_v2008_10
NAND3_X1	70	74.48	NangateOpenCellLibrary_PDKv1_2_v2008_10
NAND4_X1	61	81.13	NangateOpenCellLibrary_PDKv1_2_v2008_10

Report Mapped Gates

Gate	Instances	Area	Library v
NOR2_X1	220	175.56	NangateOpenCellLibrary_PDKv1_2_v2008_10
NOR2_X2	18	14.36	NangateOpenCellLibrary_PDKv1_2_v2008_10
NOR2_X4	1	1.33	NangateOpenCellLibrary_PDKv1_2_v2008_10
NOR3_X1	29	30.86	NangateOpenCellLibrary_PDKv1_2_v2008_10
NOR3_X2	1	1.06	NangateOpenCellLibrary_PDKv1_2_v2008_10
NOR4_X1	15	19.95	NangateOpenCellLibrary_PDKv1_2_v2008_10
OAI211_X1	49	65.17	NangateOpenCellLibrary_PDKv1_2_v2008_10
OAI21_X1	182	193.65	NangateOpenCellLibrary_PDKv1_2_v2008_10
OAI221_X1	39	62.24	NangateOpenCellLibrary_PDKv1_2_v2008_10
OAI222_X1	2	4.26	NangateOpenCellLibrary_PDKv1_2_v2008_10
OAI22_X1	273	363.09	NangateOpenCellLibrary_PDKv1_2_v2008_10
OAI33_X1	4	7.45	NangateOpenCellLibrary_PDKv1_2_v2008_10
OR2_X4	62	65.97	NangateOpenCellLibrary_PDKv1_2_v2008_10
OR3_X2	4	5.32	NangateOpenCellLibrary_PDKv1_2_v2008_10
OR3_X4	4	5.32	NangateOpenCellLibrary_PDKv1_2_v2008_10
OR4_X2	10	15.96	NangateOpenCellLibrary_PDKv1_2_v2008_10
OR4_X4	1	1.60	NangateOpenCellLibrary_PDKv1_2_v2008_10
SDFFR_X2	1	6.92	NangateOpenCellLibrary_PDKv1_2_v2008_10
SDFF_X2	218	1333.72	NangateOpenCellLibrary_PDKv1_2_v2008_10
XNOR2_X1	177	282.49	NangateOpenCellLibrary_PDKv1_2_v2008_10
XNOR2_X2	8	12.77	NangateOpenCellLibrary_PDKv1_2_v2008_10
XOR2_X1	9	14.36	NangateOpenCellLibrary_PDKv1_2_v2008_10
TOTAL	3879	7642.44	

Report Datapath Area

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 02 2009 10:38:27
- **Module:** miniUART
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** fast (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	0.0	0
external	0.0	0
others	748.79	100.00
TOTAL	748.79	100.00

Report Power

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:22:13
- **Module:** miniUART
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** fast (balanced_tree)
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
miniUART	396	15525.20	37169.82	8558.98	45728.80
miniUART/ClkDiv	98	3373.82	8180.65	1497.27	9677.91
miniUART/RxDev	130	3953.77	14124.99	2623.05	16748.04
miniUART/TxDev	82	2909.23	10812.92	1702.34	12515.26

Report Datapath Area

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** May 30 2009 13:39:19
- **Module:** timer **(original)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	182.48	17.35
external	0.00	0.00
others	869.02	82.65
TOTAL	1051.50	100.00

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** May 30 2009 13:38:14
- **Module:** timer2 **(med sammenlikningsregister)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Datapath Area		
Type	**Cell Area**	**Area %**
datapath	372.67	22.67
external	0.00	0.00
others	1270.95	77.33
TOTAL	1643.62	100.00

Report Power

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:29:12
- **Module:** timer **(original)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
timer	372	5511.81	49502.65	6976.26	56478.90
timer/inc_add_36_15	186	1370.08	4242.98	825.22	5068.20

- **Generated by:** Encounter(R) RTL Compiler v07.20-s009_1 (Feb 7 2008)
- **Generated on:** Jun 11 2009 12:33:21
- **Module:** timer2 **(med sammenlikningsregister)**
- **Technology library:** NangateOpenCellLibrary_PDKv1_2_v2008_10 revision 1.0
- **Operating conditions:** typical (balanced_tree)
- **Wireload mode:** enclosed

Report Power					
Instance	**Cells**	**Leakage (nW)**	**Internal (nW)**	**Net (nW)**	**Switching (nW)**
timer2	648	8081.14	86796.27	16750.94	103547.21
timer2/inc_add_38_15	186	1382.44	5012.73	1103.52	6116.25
timer2/lt_49_26	206	1408.87	8755.84	3093.67	11849.50

Lekkasjestrømmen i en transistor er eksponentielt avhengig av spenningen mellom gate og source, V_{VG}. V ed å redusere VVDD, og dermed også V_{GS}, vil lekkasjestrømmen minke. Forholdet kan beskrives med

$$I_{SUB} \propto e(\frac{V_{GS}}{nV_T})$$

Hvor n er forholdet mellom gate og bulk kapasitans pluss en, og V_T er temperaturavhengig terskelspenning. Det antas at n = 2 og V_T = 0.3V.

Tall for forskjellige VVDD spenninger er gitt i Vedlegg 13. Forholdstallene for lekkasjestrøm blir da

$$F_{ACTIVE} = e(\frac{1,1}{0,6}) = 6,25$$

$$F_{REST} = e(\frac{0,72}{0,6}) = 3,32$$

$$F_{HIBERNATE} = e(\frac{0,27}{0,6}) = 1,57$$

Ut i fra dette beregnes lekkasjestrøm ved REST og HIBERNATE.

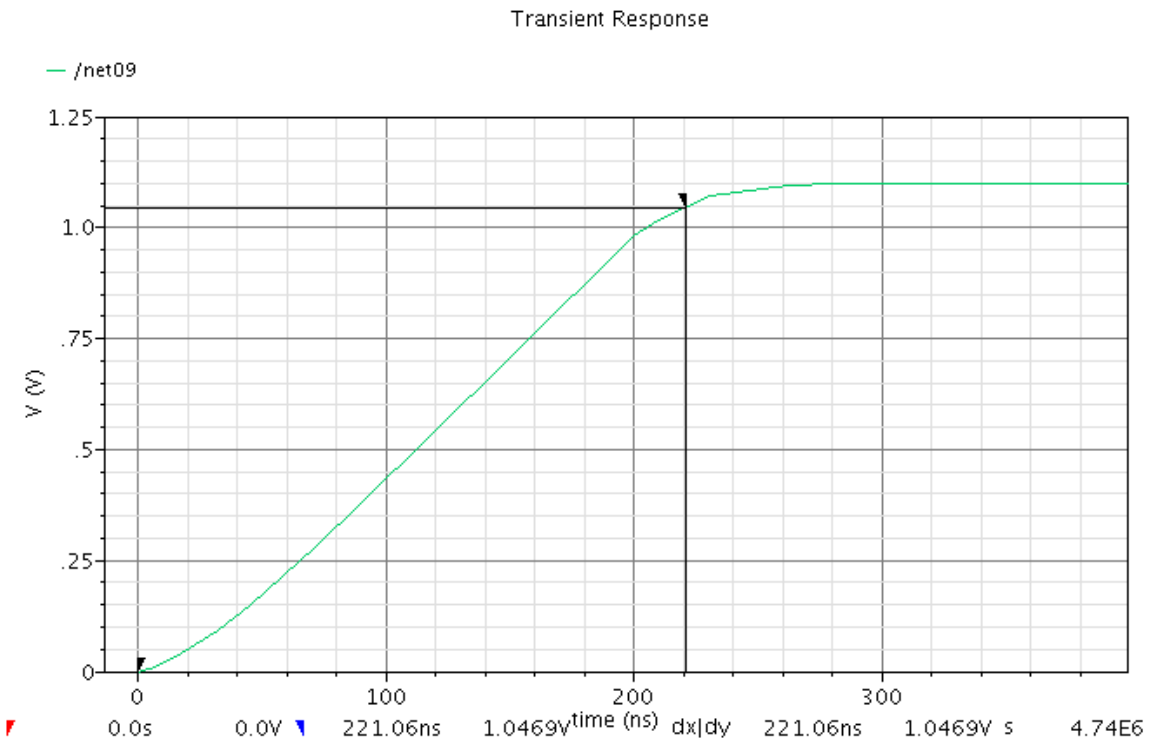
$I_{SUB,\ ACTIVE}$ = 150µA, gitt av Vedlegg 9.

V_{GS} = 0,27 => $I_{SUB,\ REST}$ = 0,53 $I_{SUB,\ ACTIVE}$ = 80 µA

V_{GS} = 0,72 => $I_{SUB,\ HIBERNATE}$ = 0,25 $I_{SUB,\ ACTIVE}$ = 37,5 µA

Tallene beregnet her er forenklet, og baserer seg på flere antakelser. De har derfor stor feilmargin. For nøyaktige tall kreves ekstrahering av utlegg, og spice simuleringer.

Figuren under illustrerer tiden det tar å lade opp en kapasitans på 215pF med VDD lik 1,1V. Det er da antatt at ZPU har en indre resistans på 100Ohm.



Figur 13.1 – Oppladning av kapasitans

Som vi ser benytter den 221ns for å oppnå 95 % av VDD. Det er valgt å la forholdet mellom oppvåkningstidene for de forskjellige modi være lineær, slik at HIBERNATE har en oppvåkningstid på ca 70 % av SLEEP, mens REST har ca 30 %. Ut ifra figuren får vi da verdier for VDD lik 720mV for REST, og 270mV for HIBERNATE. Dette gir følgende oppladningstider:

Tilstand	***Oppladningstid***
REST	*70ns*
HIBERNATE	*150ns*
SLEEP	*220ns*

Tabell 13.1 - Oppladningstider

Utregning av minimum sovetid:

Tallene for besparelse er hentet fra Tabell 4 i rapporten, mens effektforbruk for oppladning av VVDD er basert på (eq.6)

SLEEP:

$$E_{spart} = E_{lade} \Rightarrow \frac{164750nJoule}{t} = 135\,fJoule \Rightarrow \underline{\underline{t = 789ns}}$$

HIBERNATE:

$$\frac{124083nJoule}{t} = 74\,fJoule \Rightarrow \underline{\underline{t = 569ns}}$$

REST:

$$\frac{77213nJoule}{t} = 15\,fJoule \Rightarrow \underline{\underline{t = 194ns}}$$