**Eksamen 2020**

Kandidatnummer: 307

**Oppgave 1:**

a)

Trinnene i en generell designprosess er som følgende:  
  
1) Kartlegging av krav og spesifikasjoner og etablering av tidsrammer. Typisk spørsmål man ønsker å få svar på: Hva skal lages? Hvem skal bruke det? Hva ønsker kunden? Hvor lang tid trengs og når må det være ferdig? Hvordan kan vi effektivt fordele tidsbruk under utvikling?

2) Designfase: Utvikle konsepter, løsninger og virkemåter til produktet slik at det oppfyller kravene som ble fastsatt. Kreativ fase med idemyldring og

3) Implementasjon: Realisere design ved hjelp av relevant arbeid/håndverk/programmering osv.

4) Testing og verifisering: Påvise at implementert funksjonalitet er korrekt i henhold til tenkt design og oppfyller fastsatte krav på en tilfredsstillende måte. Bruk av simulatorer, laboratorieutstyr osv.

Det er viktig å merke seg at man ikke nødvendigvis (ikke engang sannsynligvis) går sekvensielt gjennom alle trinnene til man er ferdig, er ofte nødvendig å hoppe frem og tilbake. F. eks dersom feil blir oppdaget i testing må man gå tilbake til implementering, eller hvis en kunde ombestemmer seg midtveis i prosjektet må vi gå tilbake til kartlegging av krav.

b)

Forskjeller oppgitt punktvis:  
  
1. HW og SW benytter seg av (stort sett) forskjellige rammeverk for testing. Dette bunnes i at HW sin virkemåte må testes på et plan mer tilnærmet den «virkelige verden», altså fysiske prosesser, mens SW kan utelukkende testes virtuelt. F. eks bruker vi simulatorer (Modelsim), signal-trigger deteksjon (Signaltap) og direkte fysisk verifisering til å teste HW, mens SW testing utføres ved bruk av annen SW (Debugger, Unit Testing).

2. I HW må man ta hensyn til flere faktorer som bl. a timing, store mengder parallelle prosesser og synkronisering av signaler. Dette er ikke noe man i utgangspunktet trenger å ta hensyn til i SW selv om man realiserer samme funksjonalitet.

3. Funksjoner realisert i HW er raskere enn de samme som er realisert i SW. Dette skyldes at vi har mye større muligheter for å implementere funksjonen vår med parallelle prosesser og at vi programmer et fysisk kort direkte istedenfor å la sekvensielle steg oversettes til maskininstruksjoner som siden må behandles. Dette kan vi se fra oppgaven vår I2C\_MASTER der SW realiseringen er betraktelig tregere enn den samme HW-realiseringen.

4.

Fleksibilitet i implementering: HW kan ha større begrensninger av retroaktiv endring i implementasjon eller konfigurering (f. eks å måtte produsere nye ASIC), mens SW har som regel lite konsekvenser av feilaktig implementering (jeg mener her konsekvenser i form av reimplementering, kostnad og tidsbruk under utvikling. Påstår *ikke* at feil i SW har ingen konsekvenser, heller tvert imot!). F.eks er det betraktelig enklere å kjøre fjernoppdatering av SW enn av HW (også avhengig av HW plattform.

c)

Trinn oppgitt punktvis:

1. Krav til funksjonalitet må oppgis, spesifikasjoner må stadfestes.
2. Produktet designes, som regel enten på papir eller i programvare og ved bruk av standardiserte metodikker (flow-chart, state diagram, blokkdiagram osv).
3. Design implementeres ved bruk av HDL (VHDL, Verilog). Kan bli nødvendig å gå tilbake til forrige steg dersom vi oppdager feil eller mangler i design.
4. Design testes ved bruk av testbenker og eventuelt standardiserte rammeverk for testing (UVVM). Kan blir nødvendig å gå tilbake til forrige steg dersom feil oppstår.
5. Ferdig test implementasjon, dvs samlingen av alle verifiserte kildefiler, kompileres.

d)

Maksimal delay er definert som:  
  
Tpcq + Tplog + Tsu + Tskew < Tclk

Vi har at Tclk = 196Mhz

Finner periode:

1/(196 ( 10^6) sek = 5.102 \* 10^-9 sek

Vi trekker nå fra 0.4ns:

5.102 \* 10^-9 - (0.4 \* 10^-9) = 4.702 \* 10^-9

1/(4.702 \* 10^-9) = 2.12MHz

e)

Under utvikling er det er alltid lurt å separere funksjonalitet slik at endring i en del av implementasjonen ikke påvirker andre deler («low coupling»). Dette gjør at man kan abstrahere bort funksjonalitet og benytte seg av såkalte «black boxes» uten å nødvendigvis kjenne til alle detaljer bak hvert eneste element i designet. Med tanke på drivere til SW ønsker vi at disse ikke blandes inn med applikasjonen slik at vi kan skrive en brukbar applikasjon uten å risikere feil bruk av memory map og registermanipulering, samt for å ha en klarere oversikt.

**Oppgave 2:**  
  
Se vedlegg «oppgave2.vhd» for full besvarelse.

**Oppgave 3:**

Punktvis testplan:  
  
1. Verifiser at vi kan styre PM\_IN:

Test at PM\_IN korresponderer til input men at ingenting skjer uten en puls PM\_EN

2. Test at registre oppfører seg slik ønsket:

Sett verdier på alle skriveregistre i rekkefølge (1 per 32bit-space) og sjekk at WR\_DATA mottar verdiene som blir skrevet inn. Bruk verdier som er nært begynnelsen, midten og slutten på antall tillatte bit (f.eks til PM\_PRE kan vi teste bl. a 0x0, 0x1C23 og 0xFFFF)

Sett ADR til «11» og sett verdier inn i RD\_DATA, observer at signalene som korresponderer til leseregistre mottar data slik vi forventer i henhold til implmentering..

3. Test at PM\_EN genererer en sekvens

Velg en signallengde på PM\_IN, sett en puls på PM\_EN og deretter start PM\_IN. Gjør også dette i motsatt rekkefølge for å se om det skaper feil dersom PM\_IN settes tidligere. Vi ønsker å se at tellerne våre inkrementerer slik ønsket og PM\_CNT ender opp med en sluttverdi i rekkevidden 0 til 2^32 -1. Sjekk også at RD\_DATA mottar verdi der vi forventer PM\_DONE. Bruk forskjellige lengder av PM\_IN, inkludert veldig korte for å se at vi ikke knekker designet med for raskt signal.

4. Test overflow

Sett PM\_IN til å være et veldig langt signal og observer om RD\_DATA mottar en verdi der vi forventer å se PM\_OF. Legg merke til når PM\_OF settes høy og sjekk at der korresponder med forventninger (regn ut når vi forventer overflow).

5. Full test av RST

Test at RST =’1’ gir 0 i alle felt vi opererer med (PRE\_CNT, PM\_CNT osv).

Kjør forskjellige sekvenser og avbryt dem med RST, deretter verifiser at vi kan starte opp en ny sekvens. Test også RST før start og etter slutt for lignende atferd.

6. «Garbage data test»

Gjenta 2, 3, 4, og 5 men denne gangen med signaler som har ufullstendige eller feil verdier (f.eks ‘x’ eller ‘z’) og observer oppførsel. Dette er nyttig for dokumentasjon og kan hjelpe under verifisering i HW.

b)

Testing i simulator hjelper utviklere å danne seg et helhetlig bilde ved hjelp av grafiske verktøy og detaljerte tilbakemeldinger slik at en lettere kan oppdage feil uten å måtte ty til måleinstrumenter. Når det er sagt så er simulering en intensiv operasjon (mtp på prosessering) som i tillegg utføres av programvare (SW tregere enn HW) og vi kan dermed ikke forvente å simulere store mengder av tilfeller. Dette er særlig relevant dersom kan trenger å simulere kombinatorikk av store binærtall, f.eks brede databusser. Da kan det være bedre å kjøre simulering på HW som kan gjøre gjennom mange flere tilfeller mye fortere. Når det er sagt er det ofte lurt, dersom man har mulighet til det, å teste både i simulator og på HW (simulering kan avvike fra faktisk virkemåte, f.eks hvis testbenker er feilskrevet eller HW har en ukjent konfigurasjon).

c)

UVVM er et bibliotek av automatiserte tester og funksjonalitetsarkitektur man kan integrere i simulering av VHDL-baserte testbenker. UVVM refererer også til metodologien som benyttes med hensyn på testingen.  
  
d)

UVVM gir flere fordeler over tradisjonell testbenk:  
1. Korter ned på tidsbruk under testing ettersom man slipper å skrive tester helt fra bunnen av.

2. Selvsjekkende tester kan selv oppdage når spesifiserte krav ikke overholdes, dermed slipper utviklere selv å analysere signaler direkte for å se hva som gikk galt. Dette kommer også frem i at feil blir rapportert som tekst som man lett kan finne frem til i en log.

3. Konsise test-strukturer gjør testbenk enklere å debugge, vedlikeholde og oppdatere.

Som et lite påheng vil jeg også tilføye at:

3. UVVM har en «online community» som kan være villig til å assistere dersom nødvendig.

4. UVVM er Open-Source, man kan selv klone det via GitHub og implementere den funksjonaliteten man ønsker.

e)

Vi vet at baseadresser må være delelig med størrelsen på blokken. Siden vi benytter oss av totalt 12 byte-adresser må altså baseadressen vår være delelig med et tall i rekkevidden 0 til 65536. Vi vet også at vi må ha minst 12\*32 bit plass fra baseadressen til slutten på blokken. Dermed har vi:  
  
65536 – 12 \* 32 = 65152 🡪 Dette blir siste mulige adressen vi kan plassere PM.

65152/12 = 5429.33 🡪 5429 forskjellige adresser.

f)

Det har ingen betydning hvor vi legger PM i mikrokontrollen siden registre styres med baseadresse, dermed vil all lesing og skrive til registre foregår med tanke på offset i forhold til baseadressen.

**Oppgave 4:**  
  
Se vedlegg «oppgave4.c» for besvarelse på a) til og med d)

e)

Oppløsning blir på 50Mhz/4 = 12.5Mhz

For å finne frekvens regner vi om oppløsning til tid (altså tid på inkrementering), ganger resultatet med 234 og deretter finner frekvens igjen:

1/12.5Mhz = 8\*10^-8 sekunder

8\*10^-8 \* 234 = 1.872 \* 10^-5

1/(1.872 \* 10^-5) = 53.41kHz