Løsningsforslag til øving 6 – TFE4105 Digitalteknikk og Datamaskiner - Høsten 2012

Oppgave 1 – Flyttall

a) Et flyttall er på formen $F \cdot B^E$, F er fraksjonen, E er eksponenten og B er basen. F og E er binære tall, positive eller negative. B er implisitt gitt, vanligvis 2 (eller en potens av 2). Et enkelt flyttall kan skrives med mange forskjellige varianter av fraksjon og eksponent. Når tall skal behandles i en datamaskin, ønsker vi en entydig representasjon av hvert tall. Ledende 0-er i fraksjonen kan også medføre at ikke alle bitene får plass i dataformatet – vi får varierende nøyaktighet. Normalisering gir en entydig måte å representere tall på. Siden fraksjonen er et fraksjonstall, vil normalisert fraksjon ligge innenfor tallområdet $1/2 \leq F < 1$. **Med en fortegn-absoluttverdi representasjon vil ledende 0'er fjernes ved å skifte fraksjonen mot venstre samtidig som eksponenten justeres. For negative tall skiftes høyre til venstre inntil ledende 1'ere er borte**.

NB! I IEEE-flyttallsformatene skifter man enda en plass til venstre fordi det første bitet vil være implisitt gitt (det er alltid én og trengs ikke lagres).

b) Excess- 2^{k-1}-kode for eksponenten gir to viktige fordeler:

Bare positive eksponenter. Det blir da lettere å sammenligne den relative størrelsen mellom to tall uten å være opptatt av eksponentenes fortegnsbit.

Lett å sjekke om man har tallverdi null, da både eksponent og fraksjon består av bare 0'ere.

c)

Talle	*Fortegn*	*Eksponenten*	*Signifikand*	*Hex*
$-1{,}25_{10}$	1	01111	0100000000	BD00
$-23{,}5_{10}$	1	10011	0111100000	CDE0
$0{,}0507812510_{10}$	0	01010	1010000000	2A80

d) **Presisjon** er et annet ord for nøyaktighet eller oppløsning i tallrepresentasjonen. Presisjonen bestemmes av antall bit i fraksjonen, flere bit gir større presisjon.

Dynamisk område angir tallområdet som kan representeres, fra det minste tallet til det største. Dynamisk område bestemmes av antall bit i eksponenten og størrelsen på basen. Basen er implisitt gitt og som regel 2.

Dersom et gitt antall bit skal representere et flyttall, må det foretas en avveiing mellom presisjon og dynamisk område ved valg av størrelse på bitfeltene for fraksjon og eksponent.

Formatet som er gitt i oppgaven har presisjon på 6 bit, der alle bit ligger til høyre for komma. Med 6 bit til rådighet kan vi representere heltall opp til $2^6-1=63$, dvs. i underkant av to desimale siffer. Dynamisk område er $(0{,}5)\ 2^{-8} \leq n \leq (1-2^{-6})\ 2^7$ [(F = 100000, E = 0000) ≤ n ≤ (F= 111111, E = 1111)] dvs. $1{,}95 \cdot 10^{-3} < n <$

$1{,}26 \cdot 10^2$. Og det er ikke akkurat imponerende, men så er også formatet veldig forenklet.

e) Når det gjelder valg av format fins det flere løsninger. Kravene er presisjon på minst 15 desimale siffer og dynamisk område $10^{-99} < n < 10^{99}$. For å oppfylle minimumskravene må 50 bit brukes til fraksjon ($2^{50} = 1{,}13 \cdot 10^{15}$), i tillegg må det være med et fortegnsbit. Med 10 bit til eksponent fås tallområde på ca. $10^{-153} < n < 10^{153}$ (med eksponentfelt på 9 bit oppfylles ikke kravene til dynamisk område). Til sammen blir dette 61 bit. Det er ofte praktisk at tallformater opptar et visst antall byte. Her kan enten fraksjon eller eksponent utvides slik at vi får et 64-bits format.

- Presisjon (15 desimal siffer): $\wedge \log_2(10^{15}) \lessdot bit = 50\, bit\ fraksjon$

- Dynamisk område (10^{-99} til 10^{99}):
$\wedge \log_2(\log_2(10^{|99-(-99)+1|})) \lessdot bit = 10\, bit\ eksponent$

Oppgave 2 – Innganger/utganger (I/O)

a) I/O-porten er den grunnleggende enheten i datamaskinens grensesnitt mot omverdenen. Alle "ytre enheter" (tastatur, skjerm, printer, osv.) er koplet til datamaskinens systembuss gjennom I/O-porter. I/O-portene består av enkle registre som holder på data og statusinformasjon. Ved å bruke I/O-porten som "mellomstasjon" mellom prosessoren og ytre enheter, oppnår vi at prosessoren kan kommunisere med andre enheter uten å vite nøyaktig hvilke krav disse har til overføringshastighet, synkronisering osv.

b) Med **lageravbildet** I/O adresseres I/O-portene som om de var **en del av hovedlageret**. Vi bruker samme instruksjon (f.eks. MOV) til å utføre lager- og I/O-operasjoner, og lar operand**adressen** avgjøre hvilken enhet som er adressert. Dette betyr at I/O-porter "spiser" en del av adresserommet som ellers kunne vært brukt til hovedlageradresser.

Med **isolert** I/O konfigurasjon har vi **to** adresserom: ett for hovedlageret og ett for I/O-portene. Dermed kan en lagercelle og en I/O-port ha samme adresse, og det er **instruksjonen** vi bruker (f.eks. IN/OUT vs. MOV) som bestemmer om en adresse refererer til hovedlageret eller en I/O-port.

c) Ved programstyrt overføring er det programmet som ønsker å overføre data som har ansvaret for å sjekke om en gitt I/O-port har gyldig status, i tillegg til å sende data til porten eller hente data til prosessoren fra porten.

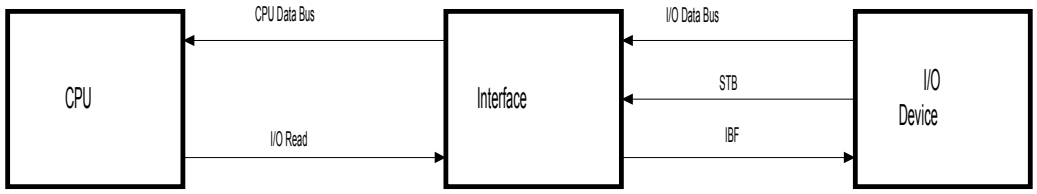
d) Det er ganske mange instruksjoner som må utføres for hver byte-overføring. Det betyr at selv om den eksterne enheten klarer å tømme (eller fylle) databufferet til I/O-porten like fort som prosessoren klarer å fylle det med data fra hovedlageret (eller skrive dataene til hovedlageret) "kaster vi bort" mange klokkesykler på instruksjonsdekoding og testing av statusregister. Overføringshastigheten blir bare en brøkdel av instruksjonsraten. (Men vanligvis er en "typisk" ekstern enhet som f.eks. tastatur, skriver og harddisk mye langsommere enn en "typisk" prosessor, slik at programmet vil bruke mesteparten av tida på å utføre "sjekk status"-løkka om igjen og om igjen.)

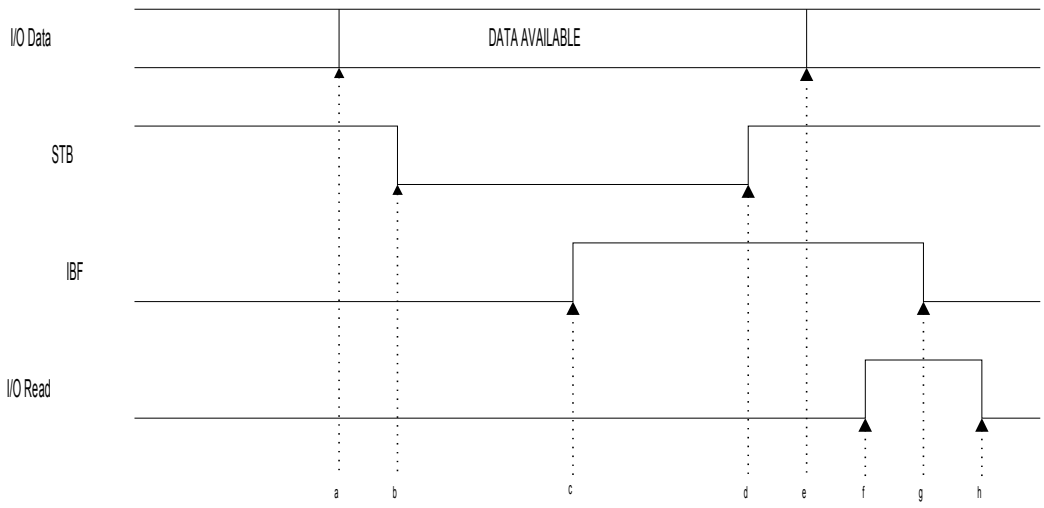
Oppgave 3 – Kommunikasjon

a) Ved synkron dataoverføring er alle hendelser (eks: signaltransisjoner) synkronisert etter en klokke. Denne klokken kan enten være felles for sender og mottaker eller de kan ha hver sin klokke. Hvis det siste er tilfellet, må klokkesignal overføres sammen med dataene slik at klokkene kan holde synkronisert.

Ved asynkron dataoverføring er det ingen klokke som styrer hendelsene. En hendelse starter når den forrige er ferdig. For eksempel kan mottaker bruke ”handshake” til å si ifra at data har blitt mottatt. Ved en synkron buss ville man isteden ha definert at data skal være mottatt i løpet av *n* klokkesykler.

b)





a – I/O-enheten legger data på bussen

b – I/O-enheten laster data inn i grensesnittet ved å aktivere STB

c – grensesnittet aksepterer data ved å aktivere IBF

d – I/O svarer ved å fjerne deaktivere STB

e – I/O-enheten fjerner data fra bussen

f – CPU varsler grensesnittet om at den leser inn data ved å aktivere I/O Read.

g – grensesnittet varsler I/O-enheten at den er klar for nye data ved å deaktivere IBF

h – CPU avslutter datainnlesing

Oppgave 4 – Avbruddsrutiner

a) Et avbruddssystem fungerer ved at de enhetene som ønsker kontakt med CPU selv gir beskjed om det gjennom å sende et "interrupt request"-signal. CPU kan da gjøre seg ferdig med instruksjonen den holder på med og starte en **avbruddssekvens**:

- Først **lagres statusinformasjon**: Prosessoren vet ennå ikke hva avbruddet skyldes, så den legger innholdet av alle viktige registre på stakken for siden å kunne fortsette i den instruksjonssekvensen den holder på med.
- Så **identifiseres** avbruddet: nå undersøkes de betingelsene som førte til avbruddet nærmere, prosessoren finner ut av hva slags type avbrudd det er, hvor signalet kommer fra, og hva det er som skal gjøres.
- Og tilhørende **avbruddsrutine** utføres: avbruddsrutinen kan være en større eller mindre kodebit som kan endre registerinnholdet fullstendig før det avslutter.
- Når avbruddsrutinen er ferdig henter prosessoren inn gammel statusinformasjon fra stakken og kan fortsette der den slapp før avbruddssekvensen.

b) Så snart vi innfører flere avbruddskilder i et system må vi avgjøre hva som skal skje hvis to eller flere ønsker oppmerksomhet samtidig. Noen avbrudd kan gjelde "sanntids" hendelser som krever umiddelbar reaksjon, mens andre avbrudd bare er signal om at noe uvanlig har skjedd. En del situasjoner som kan oppstå under utførelsen av et program kan også være så viktige at vi ikke ønsker å tillate "uviktige" avbrudd. Derfor er det vanlig å *prioritere* avbrudd sånn at hver kilde får sitt avbruddsnivå. På denne måten får høyere-prioritets avbrudd "forkjørsrett" foran lavere-prioritets, og prosessoren kan få beskjed om å se bort fra ("maske ut") avbrudd under et visst prioritetsnivå i kritiske situasjoner.

c) **Avbruddsprogrammer** har mest for seg i sammenhenger der de ikke trengs så ofte. Dette kommer av at det er såvidt "dyrt" (tidkrevende) å starte/avslutte en avbruddssekvens (se spørsmål a)), mens selve avbruddsrutinen tar like lang tid å utføre som et "vanlig" program. I forbindelse med I/O betyr dette at enheter som kun har sporadiske "input" (tastatur, f.eks.) til CPU kan kommunisere tilfredsstillende vha. avbrudd. Dette er lurt fordi tida som brukes på å starte avbruddssekvensen blir ubetydelig i forhold til tida CPU ellers ville brukt på å sjekke om det lå noe i I/O-porten til enheten.

Programstyrt I/O passer best for situasjoner der CPU er avhengig av status i I/O-porten for å kunne fortsette instruksjonsutførelsen, og derfor ikke kan bruke tida til noe annet mens den venter på dataoverføringer. Men uansett vil overføringsraten ved avbruddsprogrammering ***ikke*** bli høyere enn ved programmert I/O.

d) DMA-overføring skjer ved at CPU gir en egen prosessorenhet, DMA-kontrolleren, beskjed om å foreta en overføring mellom hovedlager og I/O-port. DMA-kontrolleren får startadresse til data i lageret, I/O-port-adresse, og beskjed om hvor mange ord som skal overføres. Så gjør CPU andre ting mens DMA-kontrolleren overfører ord for ord mellom port og lager. Kontrolleren sender beskjed til prosessoren når overføringen er fullført.

Dette er mer effektivt fordi CPU blir frigjort til andre oppgaver, men det forutsetter at databussen er ledig sånn at DMA-kontrolleren kan låne den. Det betyr at DMA-overføringen bare kan foregå på bestemte tidspunkter - når de "andre oppgavene" til CPU ikke involverer bussen.

Oppnåelig overføringshastighet øker likevel fordi data overføres av en spesialenhet uten å ta veien innom prosessoren (men igjen er den reelle hastigheten avhengig av at ytre enhet er rask nok til å "ta unna"/levere data).

Oppgave 5 – Lagerhierarki og virtuelt lager

a) Ved konstruksjon av et lagersystem ønsker vi å tilfredsstille to motstridende krav:

- Høy hastighet (lav akesstid).
- Lav kostnad pr. bit for å tillate stor lagerkapasitet.

I et forsøk på å tilfredsstille begge disse kravene, benyttes ulike lagerteknologier i en og samme datamaskin. Det totale lagersystem blir da et hierarki av mindre systemer med de raskeste (minste og dyreste) plassert nærmest prosessoren og de tregeste (største og billigste) lengst unna prosessoren.

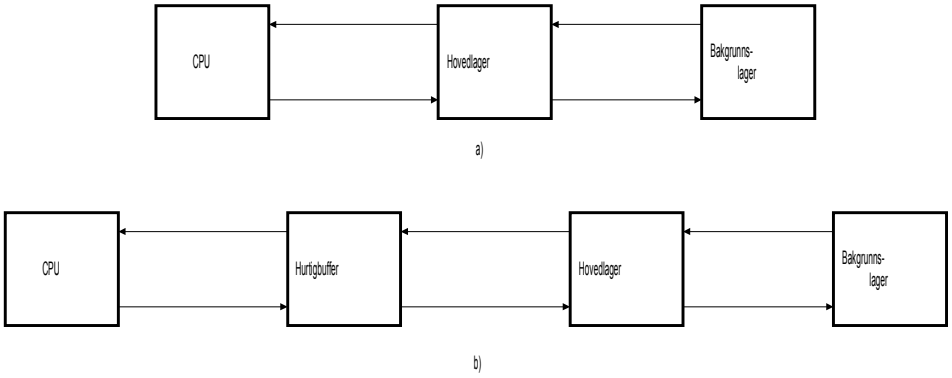
Typisk lagerhierarki:

Nivå 1: Hurtigbuffer: hurtig, statisk halvlederlager (SRAM)

Nivå 2: Hovedlager(primærlager): langsommere statisk (SRAM) eller dynamisk (DRAM) halvlederlager + leselager (ROM).

Nivå 3: Bakgrunnslager (sekundærlager): harddisk.

Nivå 4: Masselager: større utskiftbare platelager, disketter (jazz, zip), magnetbånd og optiske platelager (CD/DVD).



(a)To-nivå lagerhierarki, (b) Tre-nivå lagerhierarki

b) Aksesstid til hurtigbuffer: t_c = 8 ns

Aksesstid til hovedlageret: t_m = 85 ns

Treffraten: h

Generelt blir den effektive akesstiden til minnehierarkiet: $t_a = h \times t_c + (1-h) \times t_m$

- h = 0,87 gir t_a = 18,01 ns
- h = 0,90 gir t_a = 15,70 ns
- h = 0,95 gir t_a = 11,85 ns

c) Dersom lokalitet i referanser ikke var tilstede i programmene hadde vi nesten ingen fordeler verken med virtuelle minner eller hurtigbuffere. Uten lokalitet i referansene, ville referanser både til virtuelt minne og hurtigbuffer oftere ende med bom enn treff. Siden en bom tar lengre tid enn direkte lesing, uten virtuelt minne eller hurtigbuffer, fra minne eller disk, ville systemytelsen bli degradert i forhold til om vi ikke hadde hatt disse. En fordel ville vi likevel hatt med virtuelt minne i at vi kunne kjøre større programmer enn dersom vi ikke hadde det.

Oppgave 6 – Hurtigbuffer

a) Et hurtigbuffer er et lite, men hurtig lager som ligger mellom hovedlageret og prosessoren. I dette lageret ligger et utvalg av innholdet i hovedlageret, og så lenge det er dette utvalget prosessoren vil jobbe med, slipper den å aksessere hovedlageret. Poenget blir altså å finne fram til det hovedlagerinnholdet prosessoren ønsker å bruke, og det er dette de forskjellige avbildningene prøver på.

   Et direkteavbildet hurtigbuffer med plass til *n* lagerblokker avbilder ("mapper") blokkene fra hovedlageret direkte "modulo-*n*": blokk 0 får hurtigbuffer-adresse 0, ..., blokk *n-1* får hurtigbuffer-adresse *n-1*, og blokk nr. *n* får hurtigbuffer-adresse 0 igjen... osv.

b) Oppgaveteksten forteller oss følgende:
   8 byte pr. blokk: 3 bit til byte-adresse internt i blokka.
   2Kbyte hurtigbuffer: 256 blokker i hurtigbufferet.
   64Kbyte hovedlager: 16-bits lageradresser.

   Direkteavbildet lageradresse:

5 bit	8 bit	3 bit
tag	blokk-index	kolonne (byte)

   BCC9h:

5 bit	8 bit	3 bit
10111	10011001	001

   blokk nr.99h = 153, kolonne nr.1

   3345h:

5 bit	8 bit	3 bit
00110	01101000	101

   blokk nr.68h = 104, kolonne nr.5.

c) Ved bruk av hurtigbuffer innfører vi rask maskinvare for å oppnå raske lageraksesser. Dette kunne vi ikke ha oppnådd med et virtuelt lager. Ved bruk av virtuelt lager innfører vi maskinvare med ekstra programvare-støtte (operativsystemet) som skal gi brukeren inntrykk av at alt lageret vårt er ett stort hovedlager, men lageret blir jo ikke raskere av å bli større! Tvert imot vil lageraksesser til langsomme bakgrunnslager (som brukeren tror er en del av hovedlageret) sinke gjennomsnittlig aksesstid fordi de først må hentes inn i hovedlageret og så leses av prosessoren.

Selv om hurtigbuffer og virtuelt lager tilsynelatende bygger på samme prinsipp (flytting av lagerblokker opp og ned i et lagerhierarki), tjener de altså ikke samme hensikt.