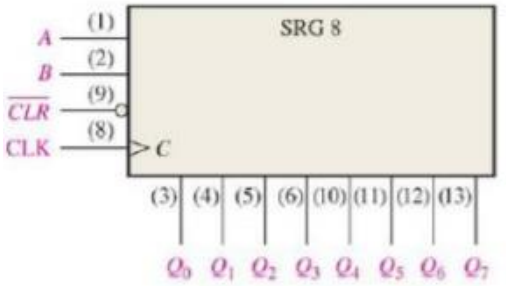
1 a)

Et skiftregister er en type digital krets som typisk består av en serie med flip-flopper. Skiftregistrene har hovedsakelig to hovedoppgaver. Disse innebærer lagring av data (et typisk minne) og flytting av data. Enten serielt eller parallelt. Lagringskapasiteten til et skiftregister representerer hvor mange bit med digital data den kan holde på. En flip-flopp kan følgelig holde på et bit avhengig av hvilken «state» den befinner seg i. Eksempelvis trenger man åtte vipper som en del av skiftregisteret for å holde på 1 byte.

Skiftregistere kan brukes til en rekke ting. Eksempler på dette er at et seriel inn og ut- register kan brukes til å skape en forsinkelse fra input til output, det gjennom å styre frekvensen på klokkesignalet. Det er ofte vanlig å overføre data på seriell form fordi det krever færre linjer ved overførsel. Datamaskiner prosesserer all data på parallell form og data på seriel form må derfor konverteres før det kan bli lest av en datamaskin. Dette kan gjøres ved hjelp av skiftregistere, som er en annen god anvendelse. Skiftregistere kan også brukes som tellere. Simplifisert kan man oppnå skiftregistere som teller ved å koble en seriell utgang tilbake til en seriell inngang.

UART står for **Universial Asynchronous Reciever Transmitter** er et interface som benyttes av datamaskiner. Som nevnt ovenfor leser bare datamaskiner data på parallell form, mens en rekke eksterne lagringsheter slik som for eksempel USB overfører data på seriell form. For at datamaskiner skal kunne lese data fra for eksempel en USB- stick må man kunne omforme seriell data til parallell data. UART tar imot seriell data og omformer det, ved hjelp av skiftregistere, til parallell data som blir sendt videre. På samme måte tar UART imot parallell data fra datamaskinen og omformer det, ved hjelp av skiftregistere til seriell data, og sender det videre til en ekstern enhet.

b)



En slik krets får en mengde data inn på inngangen, og har åtte utganger som vist på illustrasjonen ovenfor. Det vil si at den har en utgang for hvert bit, som igjen tilsvarer utgangene til de forskjellige flipp-floppene skiftregisteret består av. Etter åtte klokkepulser vil hver vippe være satt, og man kan lese av utgangene som utgjør parallell data.

For at kretsen skal kunne lese data må enable CLK være satt til høy. Utgangene leses av fra venstre til høyre, altså Q_0, Q_1, Q_2, Q_3 osv.

2 a)

Primærminne i en datamaskin er minne som er lett tilgjengelig, ofte også kalt arbeidsminne eller hovedminne. Et annen begrep som benyttes er RAM (Random Access Memory). Dette er kretser nær prosessoren i maskinen og man kan få tilgang til hver byte som er lagret i sanntid, uavhengig av hvor i minne det er lagret. Primærminne brukes som midlertidig lagring fordi dette *arbeidsminne* typisk ikke kan lagre data når strømmen er av.

Sekundærminne i en datamaskin er minne som er forbeholdt «langtidslagring». Ofte programmer, filer og data tilknyttet disse. Hovedforskjellen er at dette er et minne som kan leses og skrives til flere ganger, og dataen blir også beholdt selv om strømmen er av. Det er da en type permanent lagring. Dette type minne er relativt lett tilgjengelig, men ikke like raskt/tilgjengelig som **primærminne**.

b)

Adressebussen, databussen og for øvrig **kontrollbussen** er en del av alle CPU-er. De fleste minner skal ofte kunne leses og skrives til, og for å kunne lese eller skrive data må man vite lokasjonen til stedet som behandles i minne. Dette stedet er en adresse som angir hvor i antall bytes dataen ligger. Denne adressen legger seg på noe som kalles **adressebussen.** «Busser» kan enten overføre informasjon/data i en retning, eller i to retninger. Adressebussen overfører bare informasjon fra CPU-en til RAM/arbeidsminne, og går derfor bare i en retning. Det er CPU-en som må fortelle hvor i minne det skal legges til/slettes/hentes data fra.

Databussen blir gjeldende når minne har fått adressen det skal manipuleres data på. Hvis CPU-en skal ha data tilbake blir dataen hentet fra stedet i minne spesifisert av adressebussen og sendt til CPU-en. Denne bussen kan sende data i begge retninger, altså til CPU-en, eller til minnet. Hvis vi tenker oss at CPU-en har data som skal legges til et sted i minnet, vil denne dataen bli sendt **til** minnet gjennom databussen. Igjen blir det lagt på stedet i minnet som adressebussen holder på. Det har da blitt sendt data motsatt vei.

En illustrasjon av dette kan se slik ut:

Bildet er hentet fra pdf/forelesning

En **dekoder** er en digital krets som gir et gitt ut-signal om noe blir «gjenkjent» ved inngangen. Har vi for eksempel en dekoder med fire innganger vil man trenge seksten utganger for å dekke alle kombinasjonene av fire bitt gir. For hver av de 16 kombinasjonene av inngangene vil en av utgangene bli aktivert. En **adressedekoder** opererer etter samme prinsipp som en vanlig dekoder. I et minne har man ofte en stor del lokasjoner som kan adresseres. Man kan si at minne har en samling av adresser. Når adressebussen kommer med data som representerer en spesifikk adresse i minne er dekoderens oppgave å dekode dette, og gi et signal på utgangen når en adresse som finnes i minne blir forespurt ved inngangen.

c)

Det finnes hovedsakelig to typer arbeidsminne (RAM). **Statisk ram** og **dynamisk ram.** Statisk ram benytter vipper for å holde verdier. Minnet er statisk, som vil si at det ikke trenger oppfriskning og vil beholde minnet så lenge strømmen er på. Dynamisk ram er ikke statisk, som vil si at det hele tiden må få en «oppfrisket» ladning for å kunne holde minnet. En avgjørende forskjell er også at dynamisk ram bruker transistorer og kondensatorer for å holde verdier, det tar dermed betydelig mindre plass enn statisk ram. Det gjør også at man kan implementere DRAM mye tettere på en chip enn SRAM.

Statisk ram er imotsetning til dynamisk ram en del raskere, men trenger også mer effekt og utvikler mer varme. Fordi statisk ram er større, og bruker flere transistorer per bit-lagring er det også en del dyrere enn dynamisk ram. Det vanligste arbeidsminne er dynamisk ram, men statisk ram benyttes også ofte i en datamaskin. Cache-minne er et minne som er tett på CPU-en som hele tiden blir oppdatert med data og instruksjoner som kan være relevant for CPU-en. Dette må følgelig være raskt, og det benyttes derfor ofte SRAM-brikker.

d)

Cache minne er som nevnt ovenfor et minne som brukes av CPU-en til å hente relevante data/instruksjoner raskt. Hovedpoenget med å ha et Cache-minne i en datamaskin er for å få opp hastigheten/øke ytelsen. Et Cache-minne kan enten være et lite minne inne i selve prosessoren, eller en egen krets i nærheten av prosessoren.

ROM står for Read Only Memory, og er en type primærminne. Typisk for et ROM-minne er at det bare kan leses. Det skrives altså bare til en gang, før det kan brukes. Det beholder data selv om spenning er av, og alle steder i minne kan nås med samme tid brukt. ROM brukes typisk til å lagre programmer som sørger for oppstart av en datamaskin, eksempelvis BIOS. Mange datamaskiner bruker ROM for å sørge for riktig oppstart når spenningen kommer på.

Flash minner er en type sekundærminne for lagring av data. Dette er følgelig et minne som kan leses/skrives til flere ganger. Det er et permanent minne og beholder derfor data selv om spenningen er av. Typiske flash-minner er minnekort (SD-kort) USB-flash (minnepenn) og liknende. Det trengs få transistorer for å lagre en byte, noe som gjør at disse lagringsenhetene er relativt små i forhold til kapasiteten de kan ha. De er også forholdsvis raske, men ikke like raske som for eksempel RAM-brikker.

SSD står for Solid State Disk. Disse benyttes ofte som sekundærlager, og som et permanent lager. Disse er også altså flash minner. Disse har ingen roterende deler (slik som eldre tradisjonelle harddisker og er samtidig vesentlig raskere. SDD-er så godt som standard sekundærlager i de fleste moderne bærbare PC-er.

Magnetisk tape brukes i noe som kalles båndlager. Dette er en type datalager som bruker magnetbånd eller tape for å, og magnetisering av et område for å indikere 0 eller 1. Dette brukes ofte i forbindelse med back-up og sikkerhetskopiering av data. Lagringsmetoden krever ikke store ressurser og egner seg best for langtidslagring av data da lagring foregår relativt tregt i forhold til andre lagringsmetoder.

3 a)

Ifølge prinsippene John Von Neumann beskrev i 1945 består en datamaskin generelt av disse enhetene:

-En CPU og minneregistere

-En kontroll enhet med instruksjonsregistere

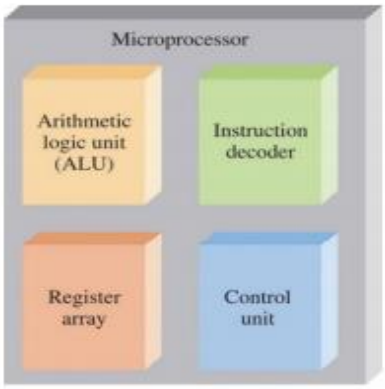
-Et minne for lagring av data og instruksjoner (RAM)

-Inn og ut enheter, eksempelvis porter/innganger på datamaskiner

-Et eksternt datalager

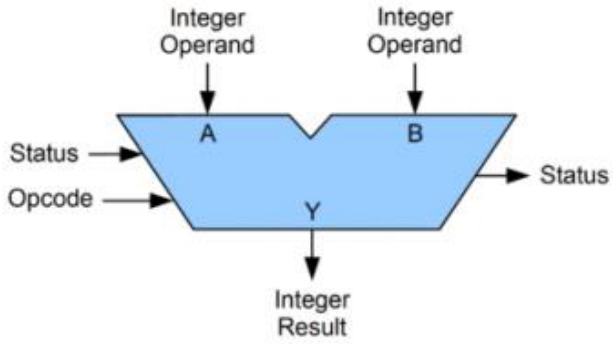
b)

En CPU består av fire grunnleggende elementer. En oversikt over disse enhetene kan se slik ut:



Bildet er hentet fra pdf/forelesning

Den første er en **ALU**, som står for Aritmetisk Logisk enhet. Som navnet tilsier utfører denne aritmetiske og logiske operasjoner på en eller to verdier (operander). Disse verdiene hentes fra register-arrayet som blir beskrevet nedenfor. En ALU-krets kan se lik ut:

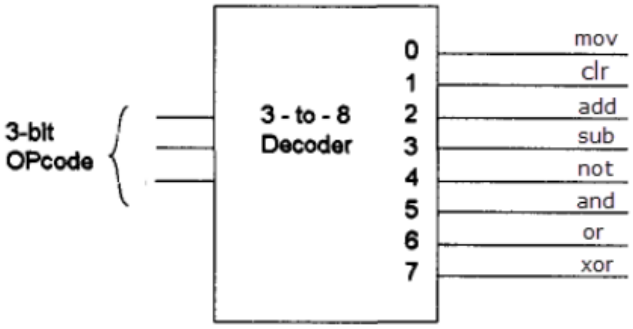


Bildet er hentet fra pdf/forelesning

Opcode verdien inn til kretsen foreller for øvrig hvilken type operasjon som skal utføres. Eksempler på dette kan være «ADD, SUB, MUL og DIV».

Den andre elementære komponenten i CPU-en er **register arrayet.** Dette er et minne, og relativt hurtig. Det er også lokalisert nærme CPU-en. Det fungerer ofte som et register for mellomlagringer i beregninger ALU-en foretar seg. Register arrayet inneholder også mindre komponenter som instruksjonspeker, stakkpeker og flag-register.

Den tredje komponenten er **Instruksjons-dekoderen.** Enhver prosessor har et sett med «lovlige» instruksjoner. Som vi så ovenfor tar en ALU- komponent en Opcode inn. (verdi som forteller hvilken operasjon som skal utføres). Hvis instruksjonsdekoderen kjenner igjen bitmønsteret på inngangen som en av de «lovlige» instruksjonene blir denne utgangen gjeldende, og den setter opp riktig kontrollsignal. En oversikt over en instruksjons-dekoder kan se slik ut:



Instruction Decoder

Bildet er tatt fra pdf/forelesning

Den siste komponenten er **kontroll-enheten.** Denne enheten styrer hvordan instruksjonene skal utføres. Dette innebærer riktige kontrollsignaler, tids-styring av de ulike operasjonene og håndtering/flytting av instruksjoner og data.

Busser er delene av CPU som håndterer lesing og skriving til et gitt minne. De tre buss-mekanismene som er gjeldende i denne sammenhengen er **adressebussen, databussene og kontrollbussen.**

Adressebussen holder på lokasjonen som CPU- en spesifiserer som destinasjon i minne for enten lesing/skriving eller sletting. Som beskrevet i oppgave **2b** går denne informasjonen bare en vei, fra CPU-en til ram/ minne. Det fordi det er CPU-en som spesifiserer hvilken lokasjon som skal leses/manipuleres.

Databussen sin oppgave er å frakte data til/fra CPU-en. Denne går dermed i to retninger i motsetning til adressebussen som bare går i en retning. Databusser er essensielt linjer som overfører signaler. Antall slike linjer angir bredden/ hvor raskt prosessoren kan overføre data. De fleste moderne CPU-er har 64 bit databusser.

Kontrollbussen styrer i helhet når og hvordan operasjoner skal utføres. Både adressebussen og databussen består av signallinjer som er identiske og jobber sammen som en gruppe. Linjene i en kontrollbuss kan derimot bestå av flere unike signal-linjer. Karakteristikkene til disse linjene kan også typisk variere veldig. Ulikt de to overnevnte bussene kan også kontrollbussen gi signaler i både en retning og to retninger.

c)

Når en CPU skal addere to tall, må instruksjonsdekoderen først dekode input-en og gi beskjed til ALU-en hvis denne inputen matcher instruksjonen som håndterer addering i CPU-ens instruksjonssett. Det først tallet (x) hentes og legges i akkumulatoren. Dette er en type minne nær CPU-en som ofte håndterer mellomlagringer. Det andre tallet (Y) legges i dataregisteret (et annet minne nær CPU-en som holder på data). ALU-en ufører deretter adderingen og svaret legges i akkumulatoren.

En CPU kan gjøre ett definert antall oppgaver. Den kan løse disse oppgavene basert på hvilke instruksjoner den blir gitt. Hver CPU har derfor et **instruksjonssett** hvor alle instruksjoner den responderer på er definert. Disse instruksjonene blir ofte beskrevet med mnemonics, som er engelsk liknende ord som beskriver oppgaven CPU-en utfører ved den gitte instruksjonen. Eksempler på slike instruksjoner kan være **«ADD», «SUB», «MUP», «DIV» og «JMP»**

En mikroprosessor er allsidig fordi den kan både styre og utføre en rekke funksjoner. Den kan også håndtere uforutsette oppgaver (beskrevet nedenfor). Det helt grunnleggende en mikroprosessor gjør er å respondere på en gitt instruksjon, utfører en oppgave og lagrer resultatet av oppgaven. Deretter responderer den på en ny instruksjon. At en mikroprosessor kan utføre instruksjoner fortløpende er også en grunn til at de blir regnet som allsidige.

d)

Maskinkode er kode som kan leses direkte av CPU-en i en datamaskin. Hver instruksjon tilsvarer da en spesiell oppgave som CPU-en utfører. Dette er numerisk kode, og er low-level kode som er designet for å kjøre så raskt som mulig. Det består altså bare 0-ere og 1-ere som representerer de ulike instruksjonene. Disse instruksjonene blir følgelig dekodet før de blir tolket av CPU-en. Selv om det er CPU-ens eneste forståelige språk er det for oss veldig tungvint å lese og skrive. Det er også ganske problematisk å feilsøke i maskinkode. Dette impliserer igjen at det er lett å gjøre feil, samtidig som det er vanskelig å få oversikt over hva som faktisk skjer.

Assembly-kode er nivået over i hierarkiet av «koder» som gir instruksjoner til datamaskiner. Assembly kode består av en samling «mnemonics» eller op-kode. Hver prosessor har sitt sett med slike instruksjoner som representerer instruksjonene i maskin-kode. Denne samlingen kalles også ofte for instruksjon-settet til prosessoren. Siden assembly-språket er en direkte kobling til maskin-koden som prosessoren leser er språket ikke portabelt. Det vil si at assembly-koden som representerer ulike deler av instruksjons-settet er unikt for hver type prosessor. Et assembly-program skrevet for en AMD-prosessor vil eksempelvis mest sannsynlig ikke fungere med en Intel-prosessor.

Program skrevet i assembly-kode må følgelig oversettes til maskinkode. Dette gjøres ved hjelp av en assembler. Assembly-programmering er som maskin-kode fortsatt noe tungvint å både lese/skrive. Fordelene med et assembly-program kontra er program skrevet i et high-level språk er at man har mer direkte kontroll over hardware-komponenter som for eksempel minnet.

Høynivåspråk er på toppen av hierarkiet. Disse *programmeringsspråkene* tillater ord og utrykk som er enda nærmere vanlig språk, da ofte engelsk. Det finnes en rekke høynivåspråk slik som for eksempel Java, C++, JavaScript og C#. For å oversette disse språkene brukes noe som kalles en **kompilator.** Det vil si at den tar hele programmet og oversetter det før det kjøres. Høynivåspråk kan også bli oversatt med en **interpreter**. Denne oversetter programmet linje for linje under kjøring.

Dette er blir ofte omtenkt som noe tregere enn en kompilator. For å svare på andre del av oppgaven, alle programmeringsspråk må oversettes fordi maskinkode er det eneste en CPU kan tolke.

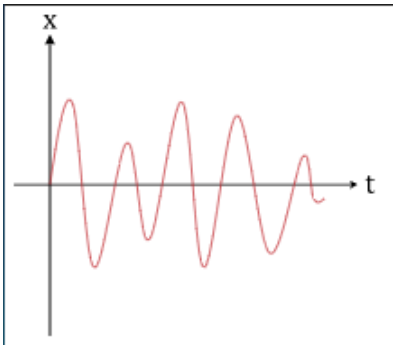
e)

Alle prosessorer må til noen tider kunne håndtere situasjoner som den ikke kan forutse. En utdatert/dårlig måte å løse dette på er å få CPU-en til å hele tiden sjekke om noen komponenter trenger å bli betjent (Polling). Dette er både ressurs og tidkrevende.

Interrupts er en betydelig mer effektiv måte å løse dette på. I denne metoden kjører CPU-en som vanlig helt til den får et signal om at det har oppstått en unormal hendelse som den må ta seg av. Hvis dette skjer vil CPU-en normalt kjøre ferdig gjeldende instruks, for deretter å sette i gang en ISR (Interrupt Service Routine). CPU-en har en egen interrupt linje. De forskjellige enhetene varsler dermed med å sende et logisk høyt signal gjennom denne linjen. En egen interrupt prosessor i CPU-en tar imot denne, og varsler CPU-en som må igangsette riktig ISR. Følgelig kan mange komponenter signalisere et avbrudd samtidig, og det er dermed vanlig at disse kategoriseres etter prioritet. CPU-en sjekker for øvrig etter interrupt-signaler etter hver eneste instruksjon den utfører.

4 a)

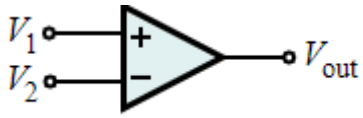
Hovedoppgaven til en ADC er å ta imot analoge spenninger eller strøm (signaler) gjøre det om til digitale verdier, som hovedsakelig er digital kode(bitverdier) som kan leses av en datamaskin. Analoge signaler er kontinuerlige, det vil si at det representerer alle verdier innenfor et målingsområde. Et digitalt signal kan for eksempel se noe slik ut:



For å kunne omforme dette til et digitalt signal må man ha noen verdier som kan gis til ADC-en. Dette fastslås ved hjelp av sampling. Signalet blir lest av med en fast takt. Det vil si at man leser av for eksempel spennings-verdier til en tid, og gjentar dette med et intervall som fastslås av samplingstiden. Hvert punkt som fastslås må gis en digital verdi, og et digitalt signal må samples med en frekvens som er dobbelt så rask som den høyeste sinusfrekvensen til det analoge signalet. (Dette er definert i Nyquists theorem).

En ADC består ofte av en sample/hold krets som har i oppgave å holde på spenningsverdien frem til neste avlesning. Dette fordi det går en gitt tid mellom hver sampling, og signalet vil endre seg noe mellom disse tidspunktene.

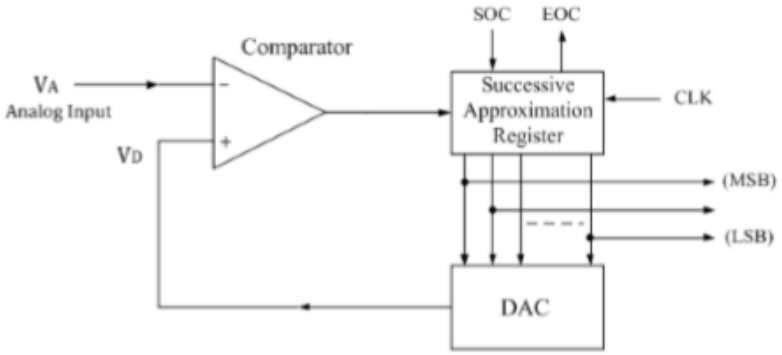
En ADC kan operere etter flere ulike metoder. En av de er **flash ADC.** En slik består hovedsakelig av en rekke komparatorer, som hver blir gitt en referanse-verdi (spenning). Det er op-amps som blir benyttet som komparatorer. Disse ser slik ut:



Hvis innput-spenningen er høyere enn referansespenningen vil komparatoren generere en 1. Outputen til disse komparatorene er koblet til en encoder. Denne har et andre input som er et enable signal, og genererer en 3-bit-verdi på utgangen som representerer signalet inn. Signalet blir altså samplet hver gang denne encoderen er enabled. (styrt av pulser). Denne metoden krever veldig mange komparatorer, men er relativt rask

En annen metode er **suksessiv approksimasjon.** Spessielt for denne metoden er at den benytter en DAC i jobben med å konvertere signalet.

En oversikt over en slik krets kan se slik ut:



Her blir det analoge signalet sendt inn i en komparator. Denne sjekker om input-signalet (ofte signalet fra en sample/hold-krets) er større eller mindre enn utgangen til DAC-en. Registeret setter første gang bitmønsteret til å være **1000.** Dette går så inn i DAC, som produserer en analog verdi av signalet. Denne går så tilbake til komparatoren som forteller om MSB (1), skal beholdes eller ikke. Så går registeret videre til neste bit, og setter dette til å være 1. Samme prosess blir gjentatt til alle bit er bestemt. En slik metode er noe tregere enn overnevnte metode, men er ofte mer brukt. Hovedsakelig fordi den benytter færre komponenter.

Noen andre metoder som er verdt å nevne er: «**Dual-slope**» og «**Sigma-Delta».**

b)

-16 bit ADC.

-Spenningsområde: $\mathbf{0}[\boldsymbol{V}] - \mathbf{5}\ [\boldsymbol{V}]$

For 16 bit har vi 2^{16} nivåer som er lik 65536

Maks nøyaktighet er $\pm\ 0.5\ LSB$

$$0.5 * \frac{5}{65536} = 3.81 * 10^{-5} = 0.038 * 10^{-3}$$

Kvantiseringsfeilen er på $0.038mV$

c)

Analogt signal fra temp-måler med område $-50.0°C - 150.0°C$

Område: $150 - (-50) = 200$

Bruker formelen fra oppgave b, og regner ut den ukjente (bit-verdien)

$$\pm 0.5 * \frac{200}{2^n} = 0.01$$

$$\frac{100}{2^n} = 0.01$$

$$100 = 0.01 * 2^n$$

$$10\,000 = 2^n$$

$$\ln(2^n) = \ln(10\,000)$$

$$n * \ln(2) = \ln 10\,000$$

$$n = \frac{\ln(10\,000)}{\ln(2)}$$

$$n = 13.28$$

For at nøyaktigheten skal bli bedre må vi da bruke en 14 bits- ADC

Benytter en 10-bits ADC og bruker denne formelen for å finne bitverdi ved 15 grader:

$$\textbf{bitverdi} = \frac{\text{Volt\_in}}{\text{Volt\_ref}} \times 2^{\text{N}}$$

$$\frac{15}{200} * 2^{10} = 76.8 \approx 77$$

Bitverdien fra ADC-en ved 15 grader er **77**

d)

Lydsignal: $\mathbf{10}\boldsymbol{KHz}$

For å få signalet på digitalform må man først sample signalet. Dette vil si å lese av verdier (spenninger) etter visse tider ut ifra et gitt intervall. Det verdiene i det samplede signalet går deretter inn i en ADC, som være basert på en av flere ulike metoder, men alle har som hovedoppgave å gjøre om en analog verdi til en digitalverdi. Når alle de samplede verdiene har gått igjennom ADC-en vil man ha et digitalt signal som representerer lyd-signalet.

Samplingsfrekvensen for å få et tilnærmet perfekt signal, må ifølge Nyquists theorem være **minst** dobbelt så raskt som den høyeste frekvensen på signalet (**$\boldsymbol{f_{sampling} \geq 2 * f_{max}}$**)

I dette tilfellet bør en samplingsfrekvens på $\mathbf{2} * \mathbf{10}\boldsymbol{Khz}$ benyttes. Altså $\mathbf{20}\boldsymbol{Khz.}$

I følge theoremet vil man aldri kunne gjenskape frekvenser over halvparten av samplingsfrekvensen i originalsignalet. Hvis man velger en samplingsfrekvens som ikke er i tråd med theoremet vil man kunne få må med frekvenser som er over $\frac{f_{sampling}}{2}$ (med originalsignalet som referanse). Disse frekvensene blir da helt andre frekvenser enn de opprinnelig var. Disse kalles **aliasfrekvenser.**

For å unngå dette bruker man et analogt lavpassfilter (**anti-alias-filter)** som er satt opp slik at det fjerner alle frekvenser over $\frac{f_{sampling}}{2}$ før man sampler signalet. Dermed vil alle disse frekvensene forsvinne, og vi får et gyldig samplet signal.

Fordi lavpassfilteret fjerner alle signaler over **cut-off** frekvensen bør denne i dette tilfellet settes til å være **minst** $10Khz$. Hvis det finnes frekvenser over $10Khz$ som er halvparten av samplingsfrekvensen vi har satt, vil disse ekskluderes og vi unngår aliasfrekvenser.