

Kurset består av relativt to uavhengige deler:

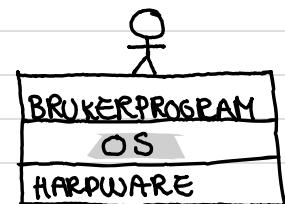
- ① Operativsystemer (OS)
- ② Praktisk bruk av operativsystemer (Linux, windows, Docker)

Læringsmål:

- ① Kjøre i brukerterminalen, inkludert script (mest Linux og Docker, og noe Microsoft)
- ② Kjøre hvordan en datamaskin virker på alle nivåer, fra transistorer og opp til operativsystemet

## HVA ER ET OPERATIVSYSTEM?

Et OS er et software-grensegrunn mellom brukeren og datamaskinenes hardware.



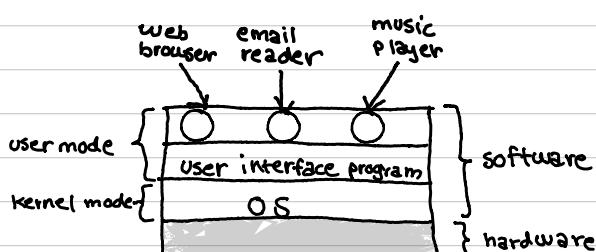
Kildekoden til OS som Linux eller Windows er ca 5 millioner linje kode. GUI, biblioteker og system software er vanligvis 10-20 ganger større.



## USER MODE OG KERNEL MODE

I user mode har man ikke alle rettigheter og man kan ikke tildele alt av rettigheter. En viktig instruksjon som ikke kan utføres i

User mode: å be datamaskinen eller hardware om å stoppe



User mode er ganske begrenset modus hvor programmer ikke kan utføre alle ting som hardware kan utføre.

I kernel mode er dette mulig.

Vi kan si at OS forenkler samtalen sin med hardware. OS og hardware snakker sammen, mens OS-en fremstiller det enkelt og forståelig ut user mode.

## OS ER PROGRAMVARE HVIS ITENSIKT ER...

- [A] Gi applikasjonsprogrammer og brukte enkelt, enklere og mer abstrakt adgang til maskinens ressurser
- [B] Administrereressursene slik at prosesser og brukere ikke ødelegger hverandre når de skal tilgang til samme ressurser

Eksempler:

- [A] filsystemet som gir brukerne adgang til logiske filer slik at brukerne slipper å spesifisere disk, sektor, sylinder, lesehode osv.
- [B] Et system som sørger for at brukerne ikke skriver over hverandres filer; fordeling av CPU-tid.

API : Application programming interface

## PROSESS → DEFINISJONER

- ① Et program som kjører
- ② Arbeidsoppgavene en prosessor gjør på et program
  - a) Et kjørbart program
  - b) Programmets data (variabler, filer, etc.)
  - c) OS-kontekst (tilstand, prioritet, prosessor-registre)
- ③ Et programs analyse

Program (kode) = DNA

Prosesen = livet

Hardware = organer/hus/mat/bygninger

KILL = CTRL-C = dørap

OS = staten/verket/politi

Root/administrator = GUD

## DATAMASKINARKITEKTUR

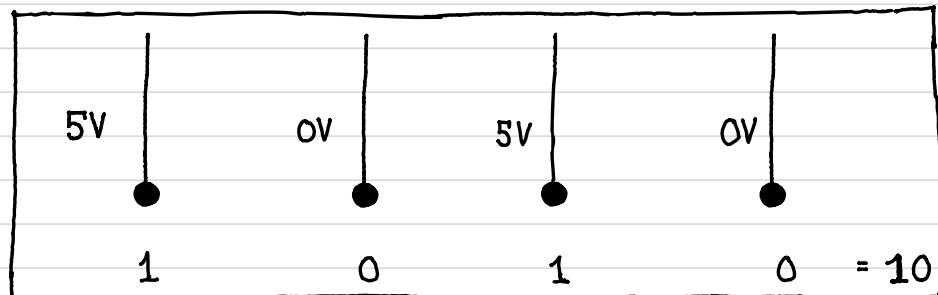
- Datamaskinarkitektur er læren om hvordan en datamaskin er bygget opp
- operativsystemkjernen styrer maskinenes hardware
- Må forstå CPU og RAM for å forstå hvordan OS-kjernen virker

Hardware bygger i bunn og grunn på digitalteknikk.

Allt i datamaskinen er representert med nuller og enere (binære tallsystemer). Fysisk representert med ingen eller positiv elektrisk spenning i forhold til jord. En rekke med slike bit representerer et binært tall. 32 bit ved siden av hverandre representerer heltall fra 0 til

$$2^{32} - 1 = 4\ 294\ 967\ 295$$

I en datamaskin er alt tall. I standard CPU-er vil 0 representere 0 volt og 1 representere 5 volt. Datamaskinarkitektur gjør ut på å manipulere disse 0 og 1-ene så man kan gjøre det man vil.



**FIGURE** = Et fire bits tall representert ved ulike spenningsforskjeller

Vi mäter alltid spenningsforskjeller i forhold til jord.

## LOGISKE OPERASJONER

- Man må kunne utføre logistiske og matematiske operasjoner på samling av bit.
- addere, subtrahere, multiplisere, dividere, sammenlikne shift-operasjoner
- Dette kan gjøres med logistiske kretser

## LOGISKE KRETSER

- Binær logikk / binær algebra
- Alle logistiske binære operasjoner kan utføres med AND, OR og NOT
- Ved å bygge disse 3 logistiske operatorene i hardware kan man gjøre alle operasjoner
- Den fysiske implementasjonen av AND, OR og NOT operatorene kalles porter

# TRANSISTOREN

- Teoretisk fysikkens store oppdagelse i det tyvende århundre: kvantmekanikken
- La grunnlaget for halvlederteknologi og transistoren
- I 1956 fikk Shockley, Bardeen og Brattain Nobelprisen i fysikk for å konstruere transistoren:
  - ↳ som gir det mulig å lage AND, OR og NOT porter ekstremt små
- Ledninger som er 5 nanometer brede
  - ↳ Et hårstrå er 100 000 nm

Man bruker da disse transistorene til å lage en logisk port som dette her:

**AND**

A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1



**OR** —

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1



**NOT** —

A	U <sub>H</sub> = $\bar{A}$
0	1
1	0



# UKE 3 - del 2

## HVA ER LINUX?

- Linux er et operativsystem = et stort og komplisert program som styrer en datamaskin
- Linux - Kjernen laget av Linus Torvalds i 1991
- GNU - Linux er et mer korrekt navn
- Nest brukt som server OS
- Linux er et Unix - OS, andre er BSD, Solaris, AIX, freeBSD, Mac OS X
- Linux ble laget av Ken Thompson og Dennis Ritchie i 1969
- Viktig del av Unix - filosofien: Sette sammen små programmer på mange måter
  - Åpen kildekode, Linux - kjernen er GPL
  - Det finnes mange distribusjoner av Linux, i alle størrelser
    - små: i IP - kameraer, mobiltelefoner (Android), rute
    - store: Ubuntu / Debian, Red Hat / Fedora / centos, SUSE / openSUSE
  - GUI med vinduer og pek-og-klikk for de som trenger det
    - ↑ fordel med kommandolinjen → lettere å automisere

## LINUX FORDELER

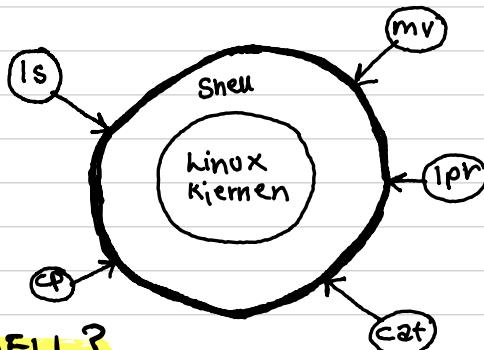
- Gratis og åpen kildekode
- Naturlig del av åpen kildekode - prosjekter
- Sikkerhet
- Stabilitet

## HVOR BRUKES LINUX?

- Desk-top / laptop : 1,5 %
- Web servere : 70%
- Public Cloud: Amazon EC2 92% (Totalt: AWS 41%, Microsoft Azure 2,9%)
- smartphone / nettbrøtt: Android 70%, iOS 2,4% (UNIX basert)
- superkomputere: 100% av de 500 største

## LINUX OG KONTEINERE

- Enorm økning i bruk av konteinere, særlig med Docker
- I hovedsak basert på Linux, men også Windows
- Kubernetes er et system for å organisere oppsett og drift av containerne
- Annonsert i oktober 2018: IBM kjøper Red Hat for 34 milliarder dollar
- Red Hats Daniel Riek: "Kubernetes is the new operating system"



## HVA ER ET SHELL?

- Kommandobasert verktøy
- tar imot kommandoer fra tasturet
- Grensesnitt mot Linux-kjernen
- Et program som tolker input som kommandoer
- utfører ordre ved å snakke med Linux-kjernen gjennom ett sett systemkall (System-API)

Hver bruker på et linux-system har

- entydig brukernavn
- passord

Oversikt over alle brukerne på systemet ligger i filen:  
◦ /etc/passwd

Og de krypterte passordene ligger i filen  
◦ /etc/shadow

Kan ikke settes av vanlige brukere, kun av root (superuser)  
Passordet settes/endres på OsloMet via web

- pwd → forteller hvor du er
- hostname → navnet på maskin
- Logginn = ssh s364520@stard.ostkommune.no

## HWORDAN FLYTTES I LINUX-FILTRE

<u>linux-kommando</u>	<u>virking</u>
print working directory	
\$ pwd	gir katalogen man står i
\$ cd home	change directory til "home" (kun fra /)
\$ cd /etc	flytter til /etc
\$ cd ..	flytter en katalog opp
\$ cd.../..	flytter to kataloger opp
\$ cd	går til hjemmekatalogen
\$ ls -l	viser alt som finnes i katalogen

# UKE 4 - del 1

## MOORES LOV:

Antall transistorer i integrerte kretser dobles seg hvert 2 år.

En stor del av de ekstra transistorene er siste årene, de har kommet pga av cashe. Jo mindre transistor, jo hurtigere kan operasjonene kjøres.

**TRANSISTOR:** er en elektronisk komponent som brukes sammen med andre elektroniske komponenter til å forsterke eller generere signaler, eller som en bryter for å slå av og på signaler eller energitransport. Den har 3 tilkoplinger og er fremstilt av et halvlederkristall.

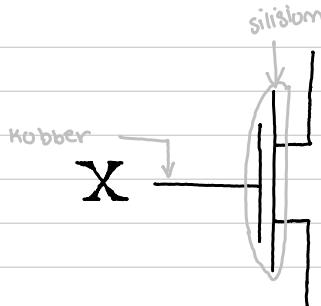
**CPU:** er den utførende enhet i datamaskin. CPU utfører instruksjonene i dataprogrammer. Slike instruksjoner kalles gjerne for maskinkode, som en CPU leser og utfører så fortest mulig. CPU-en kan delegerere oppgaver til andre enheter i maskinen.

Måten man nå får servere til å bli bedre: er ved flere CPU'er.

**CMOS:** brukt i nesten alt av halvledere (halvlederteknologi)

- (complementary metal oxide semiconductor)
- teknologi for å lage integrerte kretser som brukes i alle mikroprosessorer
- består av to type transistorer = NMOS og PMOS ↪
- Komplementær: setter sammen to type motsatte transistorer
- **NMOS-transistor:** n-type metal oxide semiconductor field effect transistor (MOSFET)
- **PMOS-transistor:** p-type metal oxide conductor

**NMOS**



X	
0	AV
1	PA

- Ingen spenning inn, bryter av
- spenning inn, bryter på

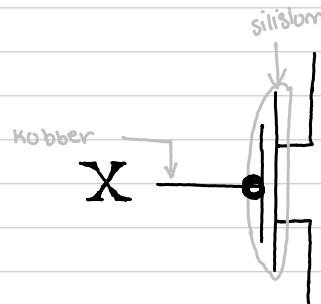
silisium er lagt på en slik måte at hvis det ikke er spenning, så fungerer det hele som en bryter. Hvis du har spenning 0 inn så er bryteren av, det er ingen kobling mellom toppen og bunnen.

Med denne type brytere så kan man bygge CPU-er som gjør akkurat den type logikk man ønsker.

**PMOS**

virker helt motsatt som NMOS. Logisk sett hadde man ikke trengt å ha denne, men det er mer for ingeniørmessige grunner, men man kan få mye mindre effekttap av å gjøre det på denne måten, å bygge logiske porter med PMOS og NMOS.

Husk at når man ikke sender spenning så åpner porten seg og det strømmer spenning.



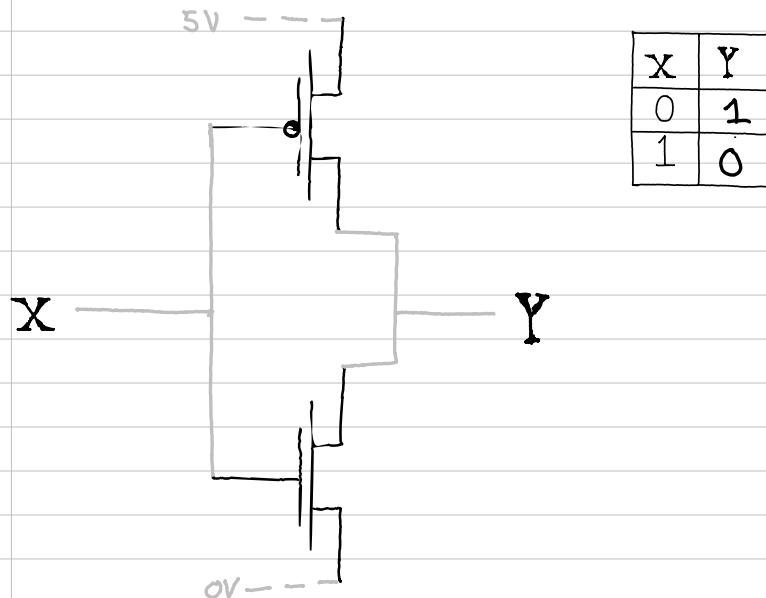
X	
0	PA
1	AV

- Ingen spenning inn, bryter på
- spenning inn, bryter av

Så... hvordan bygge logiske porter vha transistorer?

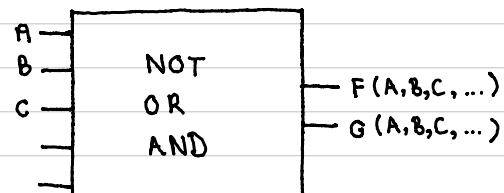
Alle logiske operasjoner kan utføres med AND, OR & NOT.

### NOT - PORT



For OR og AND porter trenger man 6 transistorer for å lage den riktige logikken.

For å kunne utføre alle type CPU-operasjoner, må man kunne lage alle binære funksjoner.



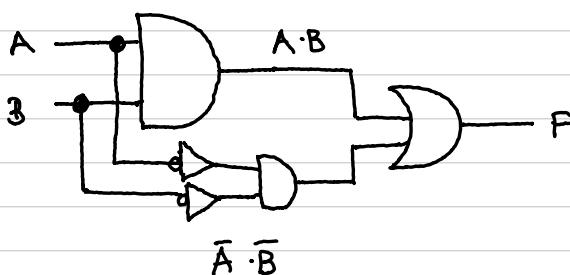
- Alle mulige logiske operasjoner kan lages ved å sette sammen systemer av NOT, AND og OR-porter.

## LOGISK SUM

- skriver tilsvarende uttrykk for alle linjer som gir 1 i sannhetstabellen
- Kan tilslutt legge sammen alle leddene med OR-operatoren
- Blir tilsammen et korrekt uttrykk for funksjonen F, fordi om minst ett av leddene i et OR-uttrykk er 1 vil det totale uttrykket også bli 1.
- Derved vil en slik sum av produkter alltid gi den riktige verdien for funksjonen F.
- I sannhetstabellen gir også den siste linjen 1. Denne linjen gir derfor uttrykket  $A \cdot B$ .
- Derved kan det logiske uttrykket for funksjon være:

$$F(A, B) = \bar{A} \cdot \bar{B} + A \cdot B$$

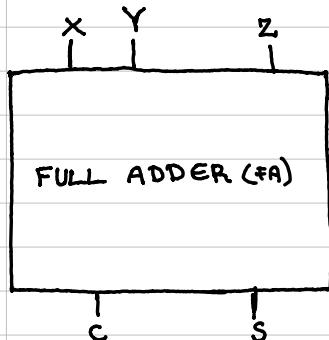
Hvis vi skal ha en logisk ene må hele uttrykket være 1, ellers 0.



Også så lages i hardware.

Karnaugh-diagram = utfører kompliserte forenklinger

## FULL ADDER



En liten boks, et lite komponent.

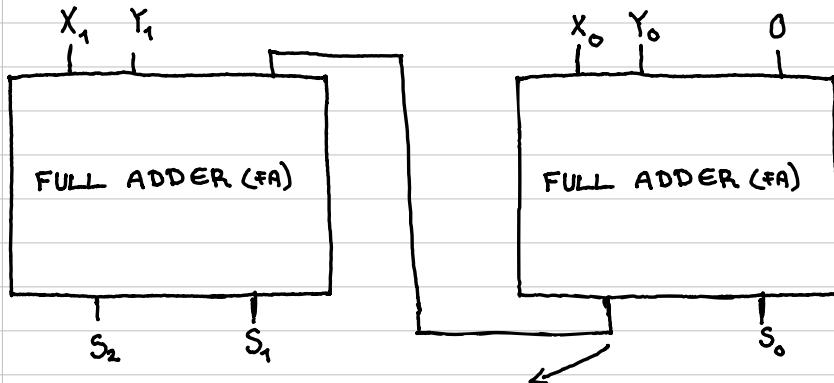
Legger sammen X og Y

Z er mente

S er summen

C = carry/mente går videre til neste operasjon.

Man kan sette sammen flere slike bokser for å utføre flere regneoperasjoner.



Riktig C kommer ut, som er mente, som skal sendes videre til neste operasjon. Ja utfører vi  $X_0 + Y_0 = S_0 + \text{evt mente} \rightarrow \text{til neste boks. Også skjer denne. Og til slutt vil carry gå over og bli } S_2.$

A handwritten addition diagram:

$$\begin{array}{r} X_1 \quad X_0 \\ + Y_1 \quad Y_0 \\ \hline S_2 \quad S_1 \quad S_0 \end{array}$$

The columns X<sub>1</sub> and Y<sub>1</sub> are circled in blue. The columns X<sub>0</sub> and Y<sub>0</sub> are circled in red. The result S<sub>2</sub> is circled in blue. The result S<sub>1</sub> is circled in red. The result S<sub>0</sub> is circled in blue.

Adderingen nå skjer med bit i GPU-en. Tallene strømmer gjennom og gir et resultat på slutten.

NOR - PORT = kan lages ved å sette sammen en p-type og n-type transistorer.

Når X kobles til positiv spenning, vil den øverste PMOS-transistoren isolere, mens den neste NMOS-transistoren gir 0 spenning ved Y. Og det motsatte skjer når X kobles til jord.

## 2.4 Linux-Shellscript

- Samling av kommandolinjer
- program som utføres linje for linje
- kompileres ikke

En meget nyttig måte å teste ut bash-script på er å bruke -x parameteren. Kjør et script som heter "mittscript", slik:

```
$ bash -x mittscript
```

LESE = 4

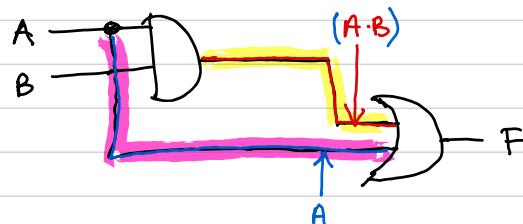
SKRIVE = 2

KJØRE = 1

23.01.23

# FORELESING

①



$$\underline{(A \cdot B) + A = F}$$

②

Ved store verdier vil uansett A dominere og gå mot  $\infty$ . Derved er  $(A \cdot B) + A$  forenklet til dominant-konstanten A.

③

A	B	$F(A, B)$
0	0	0
0	1	1
1	0	1
1	1	0

$$\text{NOT}(A) \cdot B + A \cdot \text{NOT}(B)$$

④

$$\text{Not}(A) \cdot B + A \cdot \text{Not}(B)$$

Kan ikke forenkles

XOR-porten bygges akkurat sånn og kan lages med 6 transistorer.

$$\begin{array}{r} 1 \\ \times 1 \\ \hline \underline{\underline{= 101}} \end{array}$$

$$101 = 5$$

$$101 = 5$$

$$5+5=10 \text{ som på den andre tallformen blir } 1010$$

Transistorer lager porter, og portene lager kretset.

⑥ En ny mappe lages med mkdir

En ny fil kan lages i den med touch mappe/navn på  
Når man flytter en mappe, flyttes innholdet inni fil  
med også.

Hvis du skal kopiere en mappe må man ha -r:

cp -r mappe1 mappe2

23.01.23

# Linux filsystem og kommandoer

\$ man x vil åpne opp manualsiden.

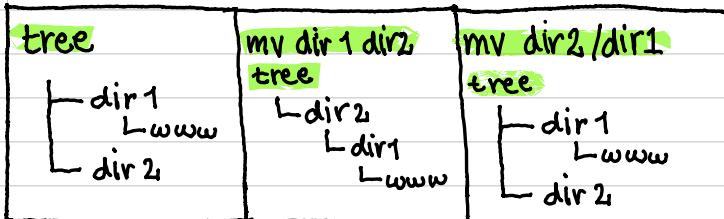
Her inne kan man søke med /pattern. Da vil alle forekomster dukke opp. Hvis man taster n så viser den neste forekomsten.

Hvis man har sett på noe før og vil se sakeloggene, kan man bare søke \$ history. Man kan også \$ history | more.

Hvis man vil søke rekursivt kan man taste control R.

Hvis man f.eks skal søke på noe og man ikke gider å søke hele hømet kan man trykke tab. Hvis man trykker en gang og det er entydig vil det dukke opp, ellers må man trykke tab to ganger.

Alle baner som begynner med / har en sikkert absolute path. så spesifiserer vi akkurat hvor i filesystemet vi er i forhold til roten av filesystemet.



Man kan kopiere rekursivt ved cp -r dir1 dir2

man kan ikke slette directory med filer inni med rmdir x.  
da bruker man rm -r x.

locate x

touch dir2/fil.txt ⇒ legger til fil.txt i dir2

find dir2 -name fil.txt

find . -name fil.txt

grep det vi leter etter filer/filen

wc teller ord

• wc -l /etc/passwd

4830 /etc/passwd

Hvis jeg fjerner -l (liniene) gir den i tillegg ord og tegn

• wc /etc/passwd

4830 12919 345927

grep 123 /etc/passwd | wc -l  
30

o  
ut  
se

{ hvis jeg har en fil eller en mappe med mellomrom så kan  
fjerne bare hvis jeg setter backslash:  
rm \min\ fil

Hvis man bruker ps så listes det prosessene man har i dette lokale skallet.

• ps aux

• ps aux | wc -l → viser hvor mange prosesser  
209 → 209 stk

• ps aux | more

• ps aux | less

\*top er en annen måte å vise prosessene på dynamisk.

- whoami
- uname
- uname -a gir operativsystemet, kjernen og detaljer om samt osv
- who viser hvem som er logget på
- type og en kommando → viser hva som vil skje

Man kan lage snarveier også symboliske lenker som peker fra et sted i filsystemet til et annet.

Her lages det en lenke mbin, og man skriver først mappen som finnes:

• ln -s /usr/bin mbin

Når man neste gang da lister med ls -l så kan man se den aller første bokstaven "l" og se at det er en lenke:

lrwxrwxrwx 1 haugerud drift 8 jan 13 23:41 mbin → /usr/bin

lenke starter med l, filer med - og mapper / directory med d

/bin/pwd kan vise meg hvor jeg er i filsystemet

Man kan også lage pekere:

ln -s /bin/pwd p

lrwxrwxrwx 1 haugerud drift 8 jan 13 23:41 p → /bin/pwd

echo ord eller tekst > filnavn setter inn ord eller tekst inni filen.

`chmod a+w dir2/` Det her betyr at alle, altså a, skal få skriverettigheter i dir2. Group da skriver man g.  
Man kan på samme vis fjerne rettigheter ved å brøte + og -.

Man kan også skrive

`chmod ug-w dir2/` Da mister user og group rettigheter

De tre gruppene er user, group og other.

`umask` er vanligvis default 0022. Det er `umask` som vanligvis bestemmer defaultrettighederne.

For å kunne stille en fil er det viktig å ha w, skriverettigheter.

En path (bane) til en mappe eller en fil angis alltid absolutt eller relativt til posisjonen man er i filtrete. Absolutt path begynner alltid med / som er rot-mappen som alle de andre henger på.

\$ pwd

/

\$ cd home

\$ pwd

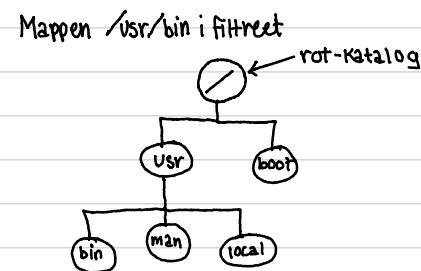
/home

\$ cd etc <----- Relativ path

bash: cd: etc: No such file or directory

\$ cd /etc <----- Absolutt path

Alt I: Relativ path	AltII: Absolutt path
Fra /	Fra hvor som helst
\$ cd usr	\$ cd /usr/bin
\$ cd bin	\$ pwd
\$ pwd	/usr/bin
/usr/bin	
Begynner ikke med /	Begynner med /



## Ukesoppgaver ④

$$5 + 5 = 10$$

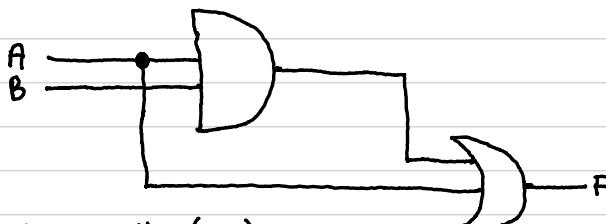
5 i binært tall er 101

$$\begin{array}{r}
 & ^1 & 1 \\
 & 1 & 0 & 1 \\
 + & 1 & 0 & 1 \\
 \hline
 & 1 & 0 & 1 & 0
 \end{array}$$


---

C	X	Z		S	C
+ Y	0	0	0	0	0
= S	0	0	1	1	0
	0	1	0	1	0
	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	1	0	0	1
	1	1	1	1	1

---



$$\text{Uttrykket blir: } (A \cdot B) + A$$

Ved store verdier vil uansett A være nere og gå mot  $\infty$ . Derved er  $(A \cdot B) + A$  forenklet til dominatkonstanten A.

## Oppgave ⑤

⑥

⑦

⑧

⑨

25.01.23

# Uke 5 del 1

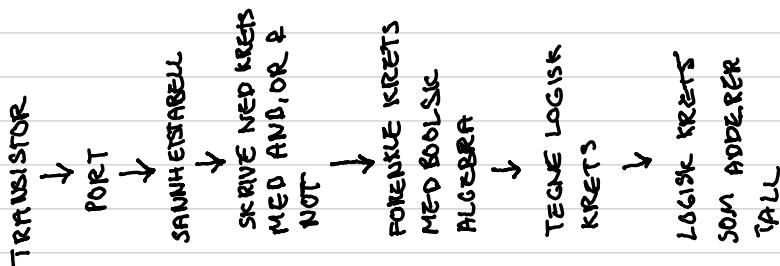
## UMASK

Man lager fil og directory (mappe). By default har de ulike rettigheter og det er umask som setter opp hvordan de skal lages.

Umask er shell built inn så hvis man runner den alene så forteller den om det nævnevende umask i oktal.

Jeg tror basically det fungerer som at 0777 resulterer alt så ingen rettigheter finnes. så 0000 vil gi alle alle rettigheter.

Den vanligste alebukten er 0022 og det er den som er på default.



I en signert operasjon hvis de to berebitene lengst til venstre er begge 0 eller begge 1, er resultatet gyldig. hvis de to venstrebitene er "01" eller "10", har det oppstått et **sign overflow**.

En XOR operasjon på disse to bitsene kan raskt bestemme om en **overflow condition** finnes (Two Complement).

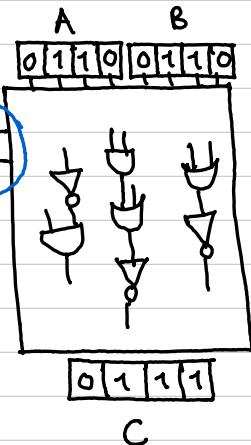
# ARITHMETIC LOGIC UNIT (ALU)

Kan gjøre aritmetikk  
 $+,-,\dots$

AND, OR &  
NOT

Hvis de to venstrebit-børene er begge 1  
tvinges A + 1 til å skje, hvis begge er 0  
tvinges A - 0 til å skje.  
Her kan man se at den tvinger A til å plusse med 1:

$$\begin{array}{r} 0110 \\ + \quad 1 \\ \hline \underline{=0111} \end{array}$$



ALU er hjernen til CPU-en (prosessoren), og CPU-en er hjernen til datamaskinen så det er her det skjer. Alt vi programmerer også vil til syvende og sist utføres inni ALU-en.

Noen enkle operasjoner en CPU operasjoner er addering, subtrahering, inkrement (øke med 1), dekrement (minke med en), multiplisering og divider. De enkleste ALU kan ikke dividere eller multiplisere.

HUSK

$$\begin{array}{r} 0110 \\ 8 \underbrace{4}_{2} \underbrace{2}_{1} \end{array} + \begin{array}{r} 0110 \\ 8 \underbrace{4}_{2} \underbrace{2}_{1} \end{array} = 6 + 6 = 12$$

$4+2=6$        $4+2=6$

$$1100 = 8 + 4 = 12$$

$8 \underbrace{4}_{2} \underbrace{2}_{1}$

De på fortige siden var de aritmetiske operasjonene. også har vi de logistiske:

- ① Shift (flytter alle bit i en retning)
- ② I feks if sammenligner vi og sjekker om noe er likt eller forskjellig.

H "Hvis du har 4 bit kan du spesifisere 16 operasjoner til denne ALU-en".

## VIPPER OG REGISTERE

Den enkleste måten å lagre 0 og 1 er ved lite elektrisk spenning. Kondensatorer kan lagre lading (0/1), men er ikke hurtig nok. Men å lagre 0 og 1 med elektrisk spenning er en type teknologi som brukes i RAM.

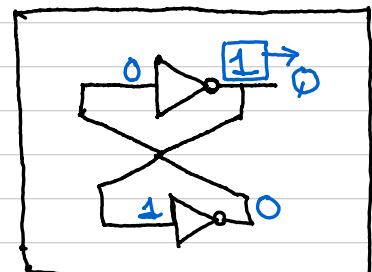
RAM er ikke like hurtig som porter, og de må refreshes (man må lade de 10 ganger i sekundet) hvis man f eks skal holde den 1-eren. Derfor bruker man ikke dette i CPU-er men man bruker vippere (flip flop på engelsk). Det er en konsekvensjon som ved hjelp av porter gjør at man kan lagre 0 og enere. De er raskere enn RAM.

En rippe kan lagre én null eller én 1.

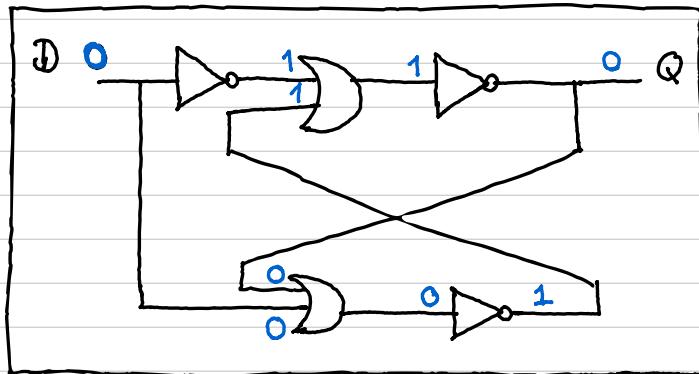
En port er noe med input og output så det krever en lukket krets slik at bit-verdiene bevarer/lagres.

## LAGRE EN BIT

Lukket krets oppfylles men på en eller annen måte må man kunne



I denne figuren går det fint for en 0 input men 1 input endrer støret.



Man ønsker å ha en vippe/lagerenhet som er sånn at hvis input er 0 så lagres output 0, uansett om D-en (input) endres. Man ønsker å ha et system slik at om man endrer D-en til en, så kan vi velge at vippen, eller registeret som holder verdien skal være vendret.

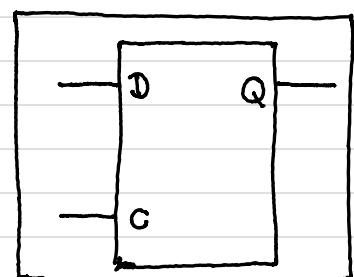
SÅ HVAR GANG VI GJØR NOE SÅ LAGRES EN ENDRING?  
LAGRES DEN NESTE ENDRINGEN ETTER DEN FORRIGE?

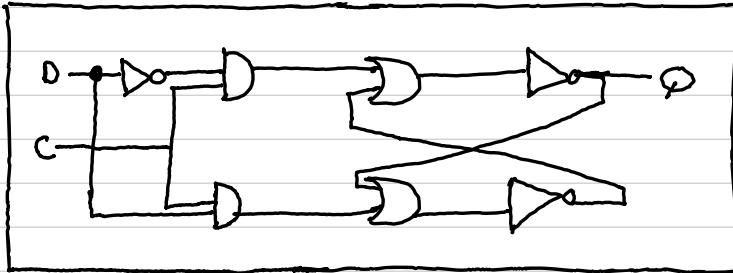
## D-LÅS (D-LATCH)

Det er en enhet som lagrer et bit men i tillegg er det mulig å velge å strik av enheten slik at den ikke reageres på input.

- $C=1 \rightarrow$  verdien inn fra D lagres
- $C=0 \rightarrow$  den lagrede verdien beholdes, uavhengig av verdien inn fra D.

C-en er en kontrollenhets.



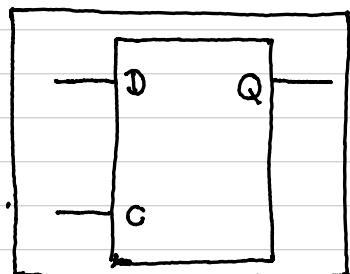


Hvis f.eks C er lik null, da skjærer vi på en måte av innvirkningen fra kretsen, fordi hvis  $C=0$ , så vil det komme en null inn i begge AND-portene, og da går det null videre til begge OR-portene. Og når det kommer 0 inn da hele høyre side av kretsen bli helt i slørt (OR OG NOT PORTENE).  
 Da vil den ikke endre seg uten at man bytter på D.

### Oppsummert:

- Hvis  $C = 1 \rightarrow$  så lager den inputverdien og lagrer den.
- Hvis  $C = 0 \rightarrow$  verdien som er lagret beholdes uavhengig av hva D er.

Det som hele tiden gjøres i datamaskinarkitekturen er at vi lager en sann logisk boks:  
 som inneholder alt i figuren over (akkurat som vi lagret den aritmetiske adderer). Vi lagde en kompleks adderer, også en logisk boks.  
 Og når vi da logisk skulle sette den sammen, så er det lett å bruke disse boksene.



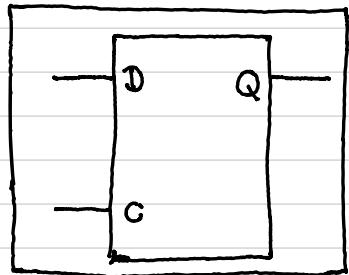
Q

Hvordan kommer verdien ut? Går det ikke vendt og rundt i enheter?

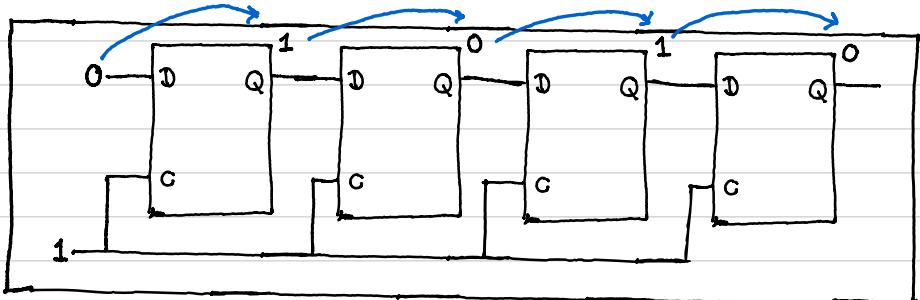
A

"Jo, de gjør på en måte det." Det er det de gjør i de kretsene men i en datamaskin vil disse endringene registreres når det er endringer i input.

Når vi lager en krets som på høyre så kan vi koble de sammen. Når vi setter  $C = 1$  og endrer på inputverdien  $D$ , så får vi en endring i  $Q$  (output). Da registreres endringen og datamaskinen endres hele tiden og det er nøyaktig det som skjer i en CPU.



## SHIFT-REGISTER

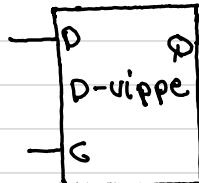
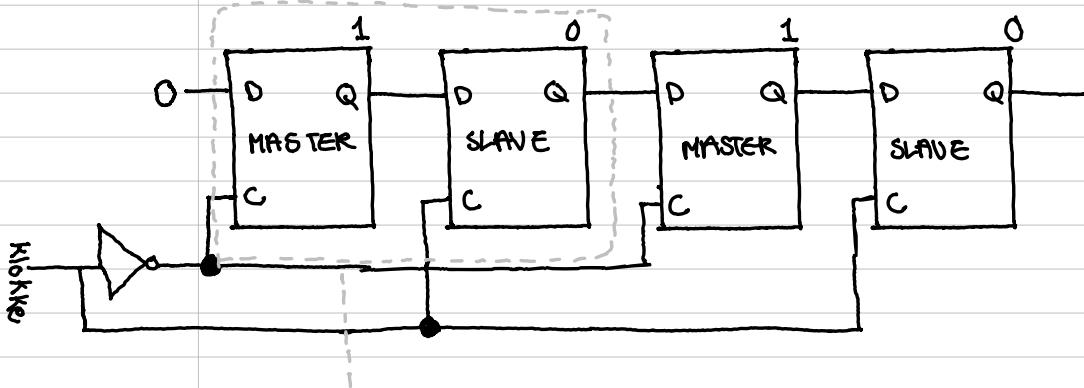


Fire D-låser settes sammen til et shift-register. Vi ønsker å kunne utføre en operasjon som  $1010 \rightarrow 0101$ . Hva er problemet?

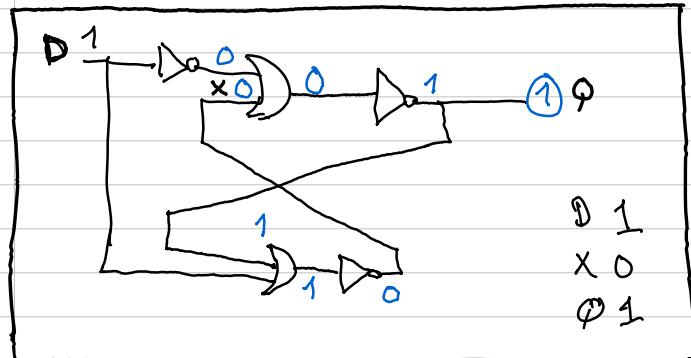
Ved å skru på bryteren her ønsker jeg å flytte alle bitene en plass til høyre.

I den siste illustrasjonen kan problemet være at det kan hende at alle tallene ikke register hverandre like raskt.

Løsningen er dermed en D-vippe!



2 Master og slave blir til en D-vippe.  
klokken gør av og på og styrer når  
de endelige dataene lagres av  
slaven. Dette er selve CPU-klokken,  
som typisk har en frekvens på 1-3 GHz.



En D-vippe er den endelige lagringenheten for 0 og 1 i en CPU. Den er satt sammen av to D-låser (latchers). Måten det er gjort på er ved en slave (som hele tiden står og leser fra master) og master er den som tar input utenifra.

Klokka skrur vi av og på systematisk hele tiden. Mer enn 4 GHz (giga hertz) har man ikke grunnet fysiske begrensninger fordi det blir for varmt. Da kreves det dyrere løsninger som vannavkjøling og gassavkjøling i stedet som billige og vanilje løsninger som små/større vifter (spørs på hvor store maskiner vi driver med).

Det er derfor mer logisk å lage flere CPU-er istedet enn å oppnå høyere hertz i en.

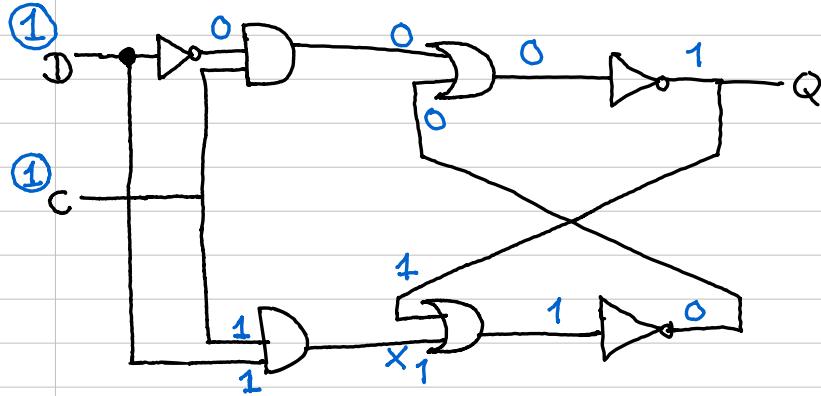
## CPU-KLOKKE

Fra  
Powerpoint

Tiden deles inn i små klokkestikk av CPU-klokken og innenfor et slikt tikk må:

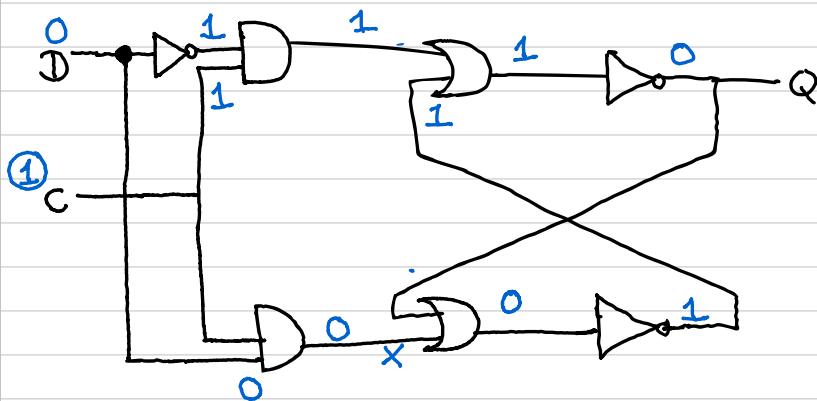
- Når klokken sender et 0: alle beregninger ferdigstilles ved å stoppe igjennom kretsene som adderer eller annen logikk, sluttresultatet lagres hos master. Det må være ferdig før klokken switcher til 1.
- Når klokken sender en 1: slaven leser verdien fra master og lagrer den. Den begynner straks å sende dette resultatet, som er det gjeldende resultatet, ut i kretsene som er koblet til utgangen for nye beregninger.

CPU-klokken er helt essensiell for å synkronisere dataene, for hvert tikk av klokken kan ett nytt sett av beregninger gjøres, for eks å utføre en maskininstruksjon.



$$X = 1$$

$$Q = 1$$



$$X = 0$$

$$Q = 0$$

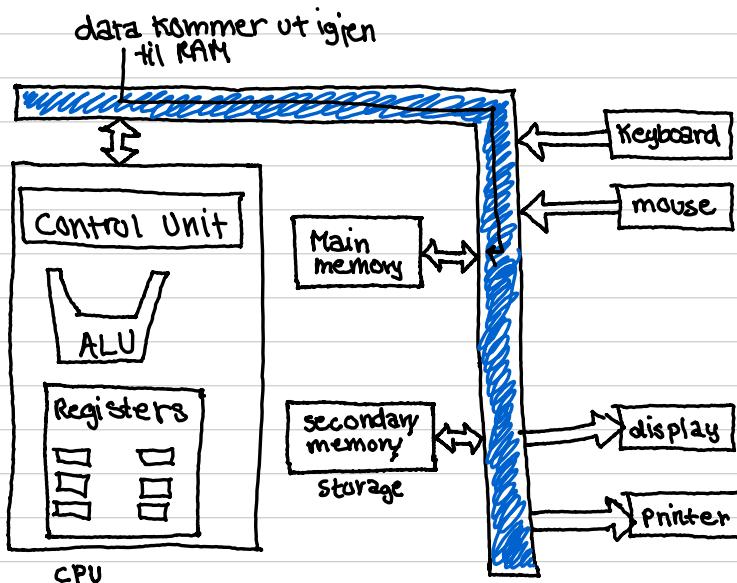
# TELLERE

Enhet man trenger for å kunne lage CPU. For å kunne løpe gjennom instruksjoner i et program, trenger man en teller som kan telle oppover for hvert klokketikk.

- En to bit teller kan telle fra 0 til 3. For å lage større tall, trenger vi bare flere vipper.  
↳ 00, 01, 10, 11

XOR port er en OR port med NOT etter. CPU-en er delt i 2 (en aritmetisk logisk enhet (ALU)) og en kontrollenhets (CU). I tillegg til hjernen, har maskinen minne (RAM) samt inn- og ut-enheter. Inn-enheter kan være mus og tastatur, og ut-enheter kan være skjerm og høyttaler. I tillegg har man en harddisk som fungerer som et lagringsmedium.

## VON NEUMANN ARKITEKTUR (1945)



## Viktigste deler:

- ALU - sentral beregningsenhet. Når man gjør beregninger kobler man registerne til input på ALU-en. Også kommer man til output, også gjøres det beregninger, også gjøres det igjen og igjen. Kontrollenheten styrer dette.
- Main memory er internt minne. Internminne går ut til en databuss (den bla på neste side).

Det som er spesielt med arkitekturen er at samme buss brukes til både instruksjoner og data.

Harvard: en buss som sender instruksjoner og en annen som sender data.

- secondary memory er disk

- Et arbeidsminne (internminne/RAM) som inneholder både instruksjoner og kode.
- En aritmetisk logisk enhet (ALU) som kan utføre matematiske og logiske operasjoner.
- En kontrollenhet som henter inn instruksjoner fra RAM, dekoder dem og sender signaler som gjør at instruksjonen blir utført.
- Registrere, internlager for både instruksjoner og data inne i CPU-en.
- Enheter for input og output som gjør at CPU kan kommunisere med harddisk, tastatur og nettverk

# BEREGNINGSENHETER

- ALU (Arithmetic logic Unit) CPU'ens hjerte
- CPU (central processing unit)
- FPU (floating point unit) vanligvis integrert i CPU
- GPU (graphics processing unit) tusenvis av cores
- FPGA (field-programmering gate array) programmbart
- ASIC (application-specific integrated circuit)

Vanligvis er den integrert i CPU, er egentlig bare en spesiell ALU som kan brukkes i floating point. Da trenger man mer kompleksitet og konkrete kretser.

brukes til grafikk og til endring av grafikk og tegne ut alt som det skal sees.

Trengs mye prosessor kraft, og mye gjøres i parallel.

Programmable logikk. Hvis man har et logisk diagram kan man programmere det inn i FPGA og det går veldig hurtig. Den kan reprogrammes.

En integrert krets: man tar en logikk, sender den til et ASIC fabrikk, også lager de akkurat den vi spesifiserer.

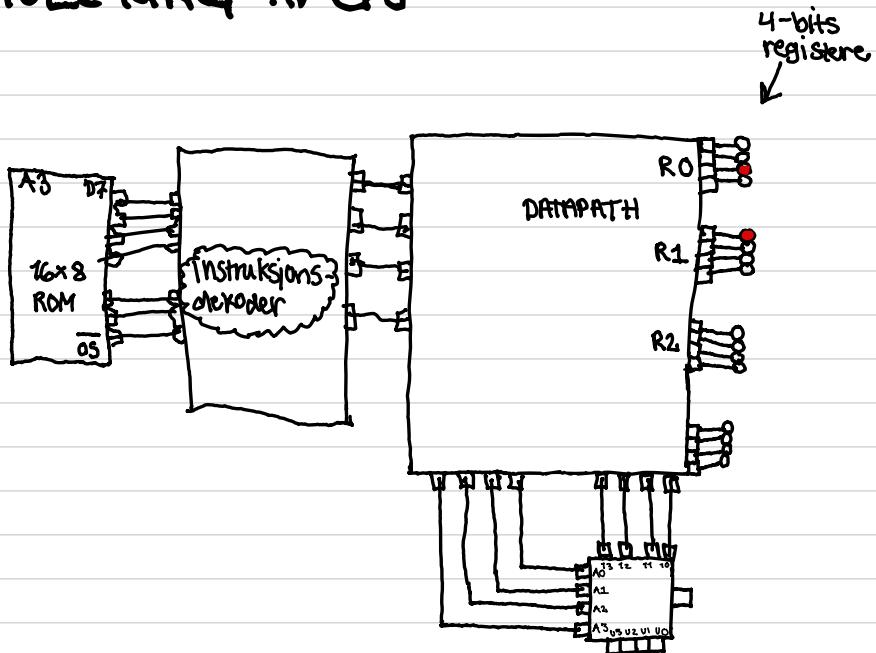
Q

Hva mener man med flere kjerner i en CPU?

A

Man har en ALU per core. så hvis vi snakker om cores så er det uavhengig regneenheter. Stort sett så kan man si at hver core er en ALU - en uavhengig regnehet.

## SIMULERING AV CPU

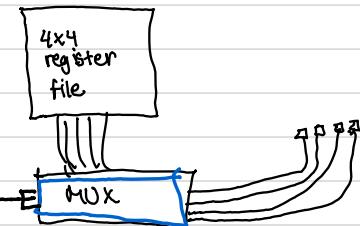


Datapath inneholder ALU og registeret. Inn i ROM ligger instruksjonene (maskinkode). Vi skal senere kompilere høynivåkode, og i vårt tilfelle ligger maskinkode. Her har vi ikke komplator, vi må skrive maskinkode rett inn i rom som 0 og 1-ere.

Det en maskininstruksjon gjør er å styre alle kontrollbitene til datapath slik at operasjonen som den ønsker å utføre, lykkes. I instruksjonsdekoder → spesielt: Opcode → sier hvilken instruksjon som skal utføres. For eks hvis vi ønsker å addere 2 tall: **Instruksjonsdekoden** skrur på de riktige bryterne

**MUX (multiplexer)** = system for å velge hvilken kant man sender inn i ALU-en.

MUX-en (i bla til høyre) kan velge å sende noe fra registeret eller konstant inn.



Avhengig av det som er her ser man om man vil ta inn noe fra registeret eller konstant. Hvis vi vil ta noe fra registeret må denne delen settes til 0/sånn at det som kommer fra A-tussen kan komme inn. En kompilator oversetter høynivåkode.

## ASSEMBLY KODE

Er en lavnivå programeringsspråk som brukes til å programmere datamaskiner direkte på maskinvernivå. Assemblykoden er en rekke instruksjoner skrevet i dette språket, som danner grunnlaget for hvordan datamaskinene utfører oppgaver. Hver instruksjon i assemblykoden tilsvarer en enkelt operasjon som kan utføres av datamaskinenes prosessorer.

# ASSEMBLY KODE FOR FOR-LØKKEN

```

0 MOVI R0 <- 3
1 MOVI R1 <- 1
2 MOVI R2 <- 0
3 MOVI R3 <- 0
4 ADD R2 <- R2 + R1
5 ADD R3 <- R3 + R2
6 CMP R2 R0
7 JNE 4
    
```

DISSE FØRSTE 4 INSTRUKSJONENE  
LEgger VERDIER I R0, R1,  
R2, R3. DISSE ER ALLE  
REGISTERE.

Linje Nr	I Nr	DR	SR
0	0010	00	11
1	0010	01	01
2	0010	10	00
3	0010	11	00
4	0100	10	01
5	0100	11	10
6	1100	10	00
7	1111	01	00

00  
01  
10  
11

Viktigste biten i CPU-en: **BRANCH CONTROL**  
siste instruksjonen i maskinkoden hopper til linje 4,  
avhengig av sammenlikningen av to register i følgende  
instruksjon. Det blir mulig å utføre alle slags løkker som  
for, while og if tester i tillegg. JNE (Jump Not Equal) gjør  
at man hopper til en adresse hvis sammenlikningen av to  
registere i følgende instruksjon visste at de er forskjellige.  
Dette muliggjøres av branch control delen av CPU. Den gjør  
at PC (program counter) kan hoppe og ikke alltid  
bare økes med en.

BLA

Athengig av resultatet ved en sammenlikning i datapath : så settes Z og CO til branch control (via status register) som da sender signal til counter **PC**. Program counter teller seg da ned til hvilken instruksjon man gjør og da endres PC slik at man hopper i kode.

- \* En vippe er den grunnleggende lagringenheten i CPU-en og brukes til all lagring av data internt, inkludert cache (mellomlagring) i CPU-en og cache mellom CPU og RAM.
- \* Et problem med å bruke D-lås til å lagre data i en CPU er at man trenger kontroll på når data skal lagres og når data skal lebes av neste D-lås.
- \* Klokker som styrer vippene svinger med en frekvens på 2-4 GHz som vil si et par flere milliarder svigninger av og på i løpet av et sekund.

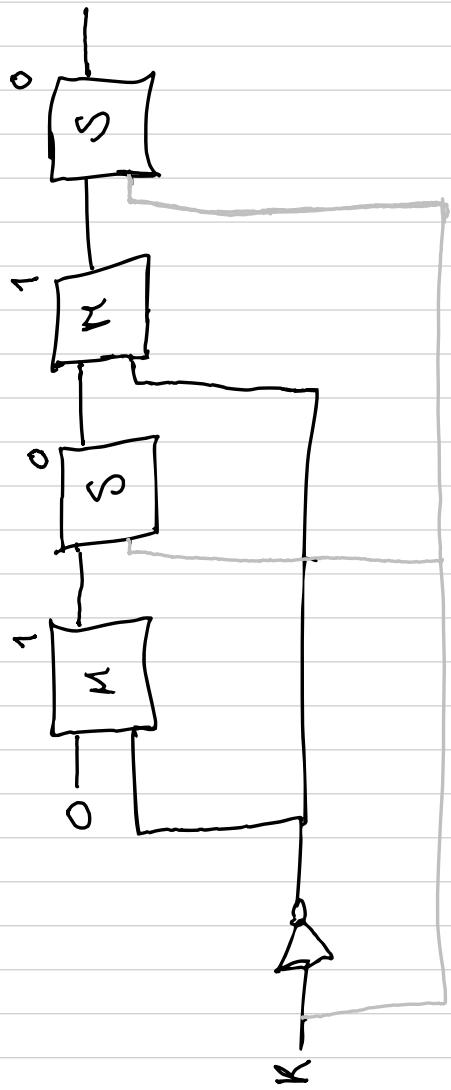
## Viktige punkter:

► CPU(Central Processing Unit) inneholder kontrollenheten, ALU og registre (de to siste utgjør tilsammen datapath). Et problem med denne arkitekturen blir omtalt som 'the Von Neumann bottleneck'. Det kommer av at instruksjoner og data deler samme data-buss. I Hardvard arkitekturen er strømmen av instruksjoner og data inn til CPU fysisk adskilt. I de fleste moderne løsninger er en Modified Harvard architecture tatt i bruk som løser flaskehalsen ved å ha forskjellige cache-kanaler for instruksjoner og data.

► En komplett CPU som kan utføre programmer skrevet i såkalt maskinkode. Det er gjort visse endringer i forhold til von Neumann arkitekturen, den vesentligste er at maskininstruksjonene er lagret i en ROM (Read Only Memory) inne i CPU-en. Vanligvis hentes de fra RAM fortløpende og lagres i instruksjonsregisteret i CPU-en før de kjøres.

Java og C er høynivåspråk som CPU ikke forstår og det må derfor oversettes til maskinkode. Dette er egne oppgaven til en komplilator

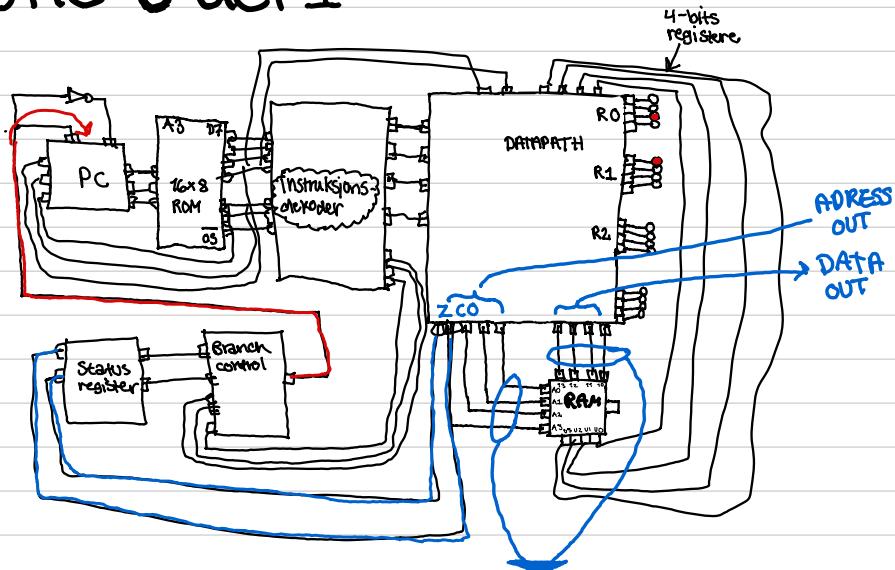
binart	Nr	operand1	operand2	Nr	Navn
0010	DR	tall		2	MOV
0100	DR	SR		4	ADD
1100	DR	SR		12	CMP
1111	nr	nr		15	JNE



1010  $\rightarrow$  0101

# Uke 6 del 1

PC  
 ROM  
 INSTRUKSJONS DEKODER  
 DATAPATH  
 RAM  
 BRANCH CONTROL  
 STATUS REGISTER



Databussen er dataene som går til RAM. Addressout og dataout vil i dette tilfellet være databussen.

↑  
Det er en buss som går mellom CPU og RAM. Resten er interne linjer i CPU.

Vi vet at i von neumann arkitektur sendes data og instruksjoner gjennom samme buss. Hvis tegningen over skulle hatt en von neumann arkitektur hadde programmet som ligger i ROM vært i RAM. Maskinen skulle da ha lest instruksjonene, utført de instruksjonene og samtidig når den skulle skrive data skulle den gå ut igjen (data out). Denne strukturen likner mer Harvard arkittetur fordi veien er annetledes her.

$$\begin{array}{r}
 & 0100 \\
 & \times 0001 \\
 \hline
 & 6161
 \end{array}$$

## BRANCH CONTROL

Fra branch control sendes det en 1 til PC. Inne i status register er det lagret verdi fra foreige operasjon. Det gjøres en compare først også hoppes det, eller det sjekkes. 1 går til program counter som vanligvis bare teller ned instruksjoner. Den starter på 0 osv (1, 2, 3, 4, ...). Når den kommer til top (f. eks 7): vil program counter settes til den verdien som man skal hoppe til, og det ligger inni instruksjonen, og dette gjør at man ikke bare går rett gjennom programmet, men gjør en branch.

## INNLEGGING AV EN LOAD-INSTRUKSJON SOM LAGRER ET RESULTAT FRA ET REGISTER I RAM

Legge til en instruksjon så maskinen skriver til RAM.

Til å begynne med skjer alle beregningene inni registerne lokalt i maskinen. Tilslutt skriver man resultatet ute i RAM. For det trenger man en instruksjon for det. I denne forelesningen legger vi til LOAD-instruksjonen som lagrer et resultat fra register til RAM.

**VIKTIG:** vi har brukt verdien til R1 (som er 1) som en referanse til adressen vi skal lagre i. Tilsvarende om R1 hadde vært 2, så hadde vi lagret i adresse 2 i RAM

På denne måten driver vi å referere til RAM hele tiden.

0000	MOVI R0 = 3
0001	MOVI R1 = 1
0002	MOVI R2 = 2
0003	MOVI R3 = 0
0004	ADD R2 + R1 = 3
0005	ADD R3 + R2 = 2
0006	CMP R2 og R0 = 2 + 3 = 5
0007	JNE 4

KLADD

## C - VERSION AV HELLO WORLD

hello.c

```
#include <stdio.h>
void main ()
{
    printf ("Hello world! \n");
}
```

For å kunne skrive ut noe fra alle C programmer må man ha standard in/out (stdio). Dette er et bibliotek som vi inkluderer. De har også en main eller hovedfunksjon, og i praksis kan man krysse det inni her.

For å få dette til å kjøre må man kompilere, og da gjør man operasjonen om å oversette fra høynivåkode til maskinkode.

gcc er en standard linux komplilator. Da lages det en executable fil

TERMINAL:

```
$ gcc hello.c
$ ./a.out
Hello World!
```

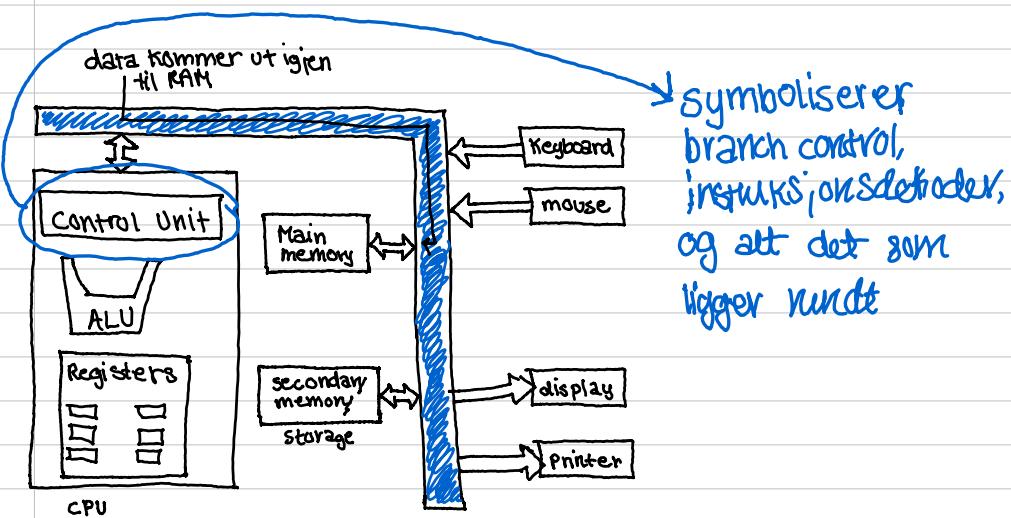
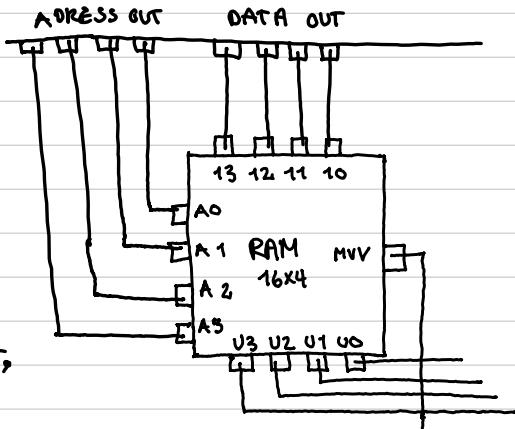
med ls -l kan jeg se at fila er ganske stor fordi vi også har importert et helt bibliotek.

Hvis jeg vil runge fila blir det \$ gcc hello.c -o nyttNavn

## DATAPRÅMM

### FØRSKELLEN PA DATA OUT OG ADRESS OUT

Adresse ut sender de 4 bitene til hvilken adresse i RAM vi skal skrive til. Mens data ut, her sendes dataene som lagres.



### C - PROGRAMMERING OG HEX-DUMP AV MASKINKODE A.OUT

2

Hva er forskjellen på høynivåkode og maskinkode?

A

Høynivå er en type programkode skrevet på et høyere nivå av abstraksjon, som gjør det lettere for utviklere å forstå og utvikle programvare. Det er vanligvis skrevet i en lesebar form for mennesker, slik som python eller java. Maskinkode, derimot, er den binære koden som en datamaskin faktisk leser og utfører. Maskinkode består av en rekke binære tall som representerer enkelte instruksjoner for datamaskinen. Høynivåkode må konverteres til maskinkode før den kan kjøres på en datamaskin.

Koden til høyre har nå også en binær fil a.out.

Hvis jeg skriver xx d a.out får jeg innholdet til fila.

I de binære filene er det kode som snakker med operativsystemet.

s = 0

i = 3

$$\textcircled{1} \quad 0 = 0 + 0$$

$$\textcircled{2} \quad s = 0 + 1 = 1$$

$$\textcircled{3} \quad s = 1 + 2 = 3$$

$$\textcircled{4} \quad s = 3 + 3 = \underline{\underline{6}}$$

```
$ gcc sum.c -o sum  
$ ./sum
```

sum = 6

```
$ gcc hello.c  
$ ./a.out  
Hello World!
```

```
include <stdio.h>
```

```
int sum() {
```

```
    int S=0, i;
```

```
    for(i=0; i<4; i++) {
```

```
        S = S + i;
```

```
}
```

```
    return(S)
```

```
}
```

```
int main() {
```

```
    int sum;
```

```
    sum = sum();
```

```
    printf("sum = %d\n", sum);
```

En integer er på 4 byte, eller 32 bit. I C skal det være sann (konvensjon). Når  $s=0$  da ligger i ram er det satt av 4 bytes rett etter hverandre. Når man da skal tilgang til den så sender man en adresse ut på bussh: f.eks hundre millioner og fortifem der ligger vår variabel.

Å skrive assembly er jobben til komplataoren.

Hva er forskjellen på bit og bytes

En bit (binary digit) er en enkel enhet av informasjon som enten kan være 0 eller 1 i digital informasjon. Bitene er grunnleggende byggesteiner for all digital informasjon.

En byte, derimot, er en samling av 8 bits. Bytes er en enhet for måling av datamengde og brukes ofte for å representere tekot, tall og bilder i en datamaskin. En byte gir oss muligheten til å representere 256 forskjellige verdier ( $2^8$ ), noe som gir oss mer kapasitet til å representere mer komplekse data enn ved å bruke bare enkelbits.

D

Hvordan vet vi at registrere og data i RAM ikke blir overskredet av andre prosesser?

A

Registrere og data i RAM (random access memory) er tilgjengelig for alle prosessene som kjører på en datamaskin, men hvordan vi sikrer at en prosess ikke overskriden data som tilhører en annen prosess, avhenger av operativsystemet som kjører på datamaskinen.



operativsystemer bruker en teknikk kalt "minnebeskyttelse" for å sørge at hver prosess har tilgang til sitt eget minneområde. Når en prosess alokkerer minne, tildeler operativsystemet et unikt minneområde til prosessen, og setter opp virtuelle barrierer (så kalte "page boundaries") for å hindre at prosessen skriver over minnet til andre prosesser. Hvis en prosess prøver å skrive til et beskyttet minneområde, vil operativsystemet utløse en "minnefeil" (memory fault) og stoppe prosessen.

Hvordan finnes i hva som returneres når man avslutter assembly-program med ret?

Når man avslutter assembly program med ret (return) returneres verdien som ligger i registeret AX (extended accumulator register). Dette kan variere avhengig av arkitektur og plattform.

Q

Forskjellen på kompilator og assembler:

A

To ulike programvare som brukes i prosessen med å oversette kildekode til maskinkode. En kompilator er et program som oversetter kildekode skrevet i et høyere nivå språk (C, C++, Java, etc.) til maskinkode som kan kjøres direkte på en maskin. Kompilatoren tar kildekode som input og genererer en binær fil som inneholder maskinkode. En assembler oversetter assemblykode (lavnivå programmeningsSpråk) til maskinkode. Assemblykode består av en rekke enkle instruksjoner som representerer maskininstruksjoner direkte.

uke ~~X~~

static int turn = 0;

GetMutex (int t) {  
 while (turn != t) {}  
}

ReleaseMutex (int t) {  
 turn = 1 - t;  
}

1  
0 != 1  
men lock

GetMutex(0)  
// Kritisk avsnitt  
ReleaseMutex(0)

} process 0 gir

GetMutex(1)  
// Kritisk avsnitt  
ReleaseMutex(1)

} process 1 gir

uke 17

static boolean [ ] flag = new boolean [2];

// begge false i utgangspunktet

0 GetMutex (int t) {

other := 1  
o = true

int other;

other = 1 - t;

flag [t] = true; // ønsker å gå inn i kritisk avsnitt  
while (flag [other] == true) {}

}

Release Mutex (int t) {

flag [t] = false;

}

---

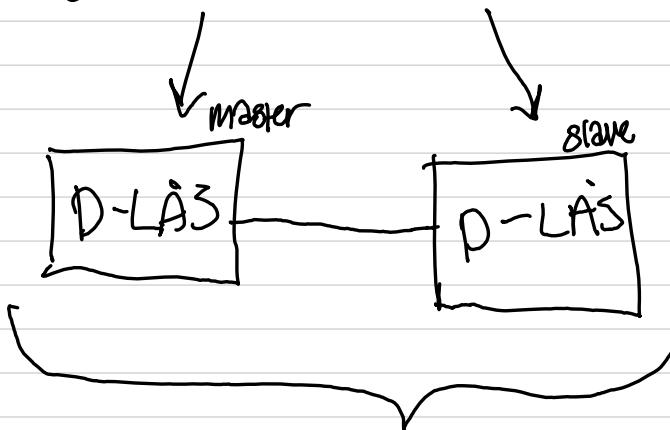
sikrer den at 0 og 1 aldri kommer sammen i kritisk avsnitt?

$\frac{0}{1024} \frac{1}{512} \frac{0}{256} \frac{0}{128} \frac{0}{64} \frac{0}{32} \frac{0}{16} \frac{0}{8} \frac{1}{4} \frac{1}{2} \frac{0}{1} \frac{0}{0}$

$512 + 4 + 2 =$

518

D-LÅS = logisk lukket krets



D-VIPPE

↑  
endelig lagringsenhet for 0 og 1.