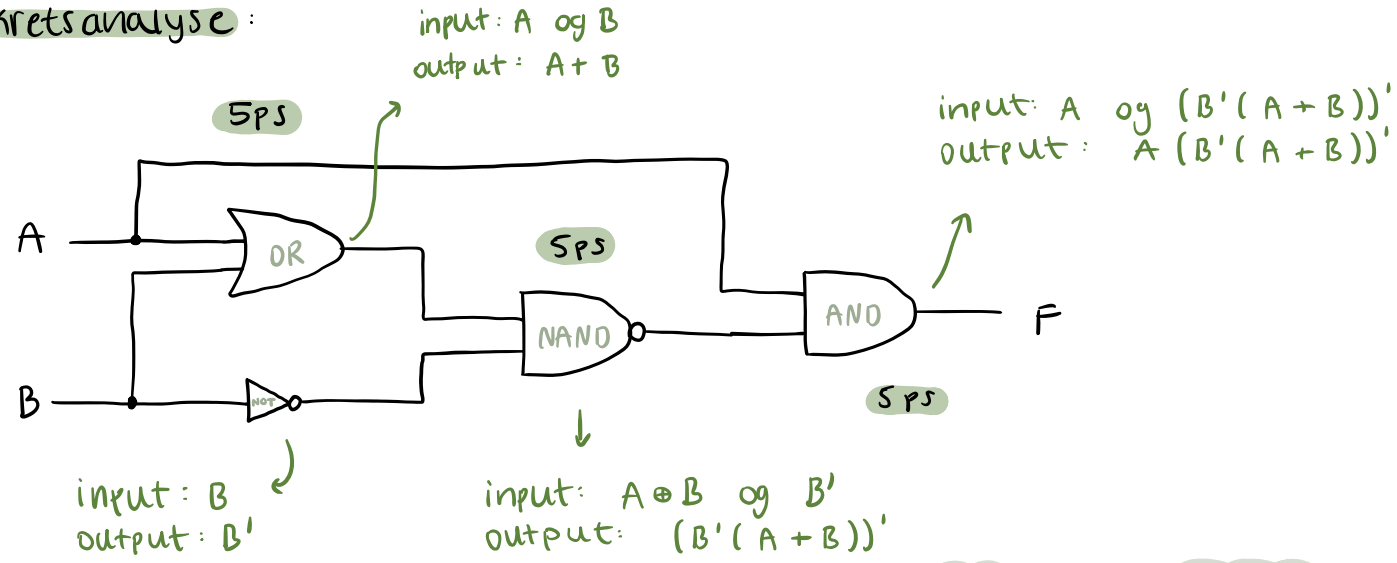
Oppsummering kretsa, synkronisering, pipeline og minne

Mer kretsanalyse:



Det tar 15 ps å få et resultat ut fra kretsen om du antar 5ps forsinkelse på hver port.

Fyller inn sannhetsverditabell:

A	B	B'	A + B	(B'(A+B))'	A(B'(A+B))'
0	0	1	0	1	0
0	1	0	1	1	0
1	0	1	1	0	0
1	1	0	1	1	1

Samme som en AND port

Forkorter uttrykket med boolsk algebra:

$A(\overline{\bar{B}\ (A+B)})$ → ganga ut parantesen

$= A(\overline{BA + \bar{B}B})$ → B' and B = 0

$= A(\overline{\bar{B}A})$ → bruker at $(AB)' = A' + B'$

$= A(\bar{\bar{B}} + \bar{A})$ → bruka at $B'' = B$

$= A(B + \bar{A})$ → ganga ut parantes

$= AB + A\bar{A}$ → bruka at $A\bar{A} = 0$

$= AB$

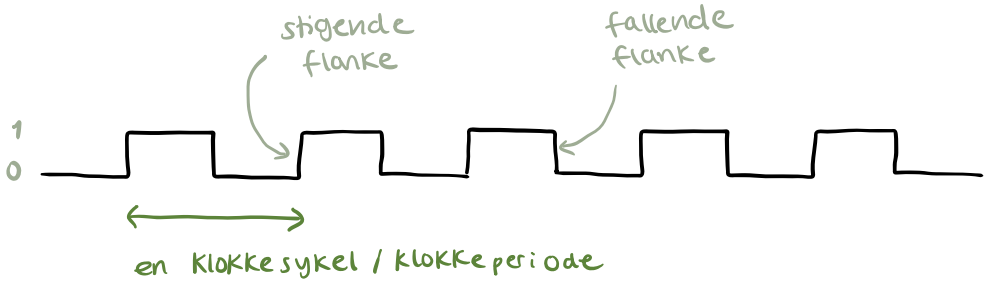
↳ funksjonsuttrykket for AND

Synkronisering

→ Det er en fast "flow" når instruksjoner utføres.

→ Hver instruksjon kjøres akkurat når den skal kjøre og den er gjort ferdig før neste instruksjon skal kjøre. Dvs. at komponentene i CPUen må samarbeide for å utføre instruksjoner, og for å gjøre dette så må de jobbe synkronisert slik at en komponent ikke blir ferdig lenge før de andre eller sender videre sin output før den andre komponenten klarer å ta imot.

→ Synkroniseringa skjer ved å bruke et globalt klokkesignal som ser slik ut:



→ når klokka "tikker" så er signalet 1 og når klokka "takker" så er signalet 0.

→ Klokkesignalet har stigende flanker (klokkesignalet går fra 0 til 1) og fallende flanker (klokkesignalet går fra 1 til 0).

→ På stigende flanke starter en operasjon og så lenge klokka er høy, så jobbes det på den operasjonen.

→ Når klokka er lav er det "pause" – det trengs for å ikke få forsinkelser mellom komponentene og at de er resetta før neste operasjon skal utføres.

→ Klokkeperioden må være lang nok for å klare å utføre den lengste instruksjonen (for en 1-syklers prosessor).

Klokkefrekvens:

→ Antall klokkesykla som CPUen utfører per sekund er frekvensen

→ Hver gang klokka slår i så henter CPUen en instruksjon og utfører den. Antall slike "fetch and execute" sykla CPUen gjør på et sekund kalles frekvensen.

→ Frekvens måles i Hertz, Hz → 1 Hz er en klokkesykel per sekund

→ Jo større frekvensen er, jo fortere klarer CPUen å prosessere instruksjoner. Det er fordi når frekvensen er større, så klarer den å utføre flere instruksjoner per sekund

→ Formel: **f = 1 / klokkeperiode**

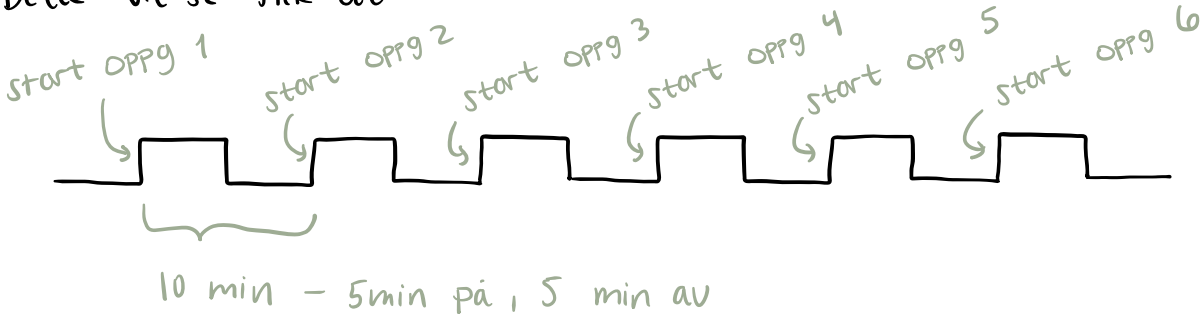
Eksempel- la oss bake gulrotkake: For å bake en gulrotkake er det en del mindre oppgaver som må gjøres:

- smelte smør
- rive gulrøtter
- måle opp mel, sukker og bakepulver og melk
- ha alle ingrediensene i en bolle
- ha bakepapir i en langpanne
- ha deigen i langpanna

Noen av disse stegene må gjøres i en viss rekkefølge for at kaka skal bli god. Vi er bare 1 person, så vi kan bare gjøre 1 oppgave av gangen (og i den rette rekkefølga).

Vi antar at den lengste oppgava er å måle ingrediensene og at det vil ta 5 min. Vi vil også sette på en timer for å synkronisere arbeidet vårt (selv om vi bare er person). For at dette skal gå, må timeren ringe hvert 5. min. slik at når timeren ringer først så starta vi på en oppgave og når den ringer igjen er vi ferdig. Vi har altså tatt hensyn til at den lengste oppgava tar 5 min. Klokkesignalet vårt er 50% på og 50% av.

Dette vil se slik ut:



La oss regne ut frekvensen for arbeidet vårt:

$$f = \frac{1}{\text{klokkeperiode}} \Rightarrow$$

$$f = \frac{1}{10\,\text{min}} = \frac{1}{600\,\text{s}} = 0.00167\,\text{Hz}.$$

→ 0.17 % av en sykel

Denne bakeprosessen gikk altså ikke så veldig raskt. Siden du bare er 1 person, så kan du bare gjøre en ting av gangen. Men siden noen av oppgavane er uavhengige av hverandre, så kunne de vært gjort parallelt. Dette er vanskelig du bare er en person, og med mindre du tar med en venn for å hjelpe til er det få andre muligheta for å gjøre arbeidet raskere. Hadde du f.eks hatt 3 sett med armer så det vært mulig å pipeline oppgavene. Noe sånt kan vi dessverre ikke gjøre, men det kan heldigvis datamaskinen.

Pipeline:

→ En metode for å få CPUen til utføre instruksjona "parallelt".

→ Hver instruksjon kan f.eks deles inn i 4 subinstruksjona:

Fetch	Decode	Execute	write back

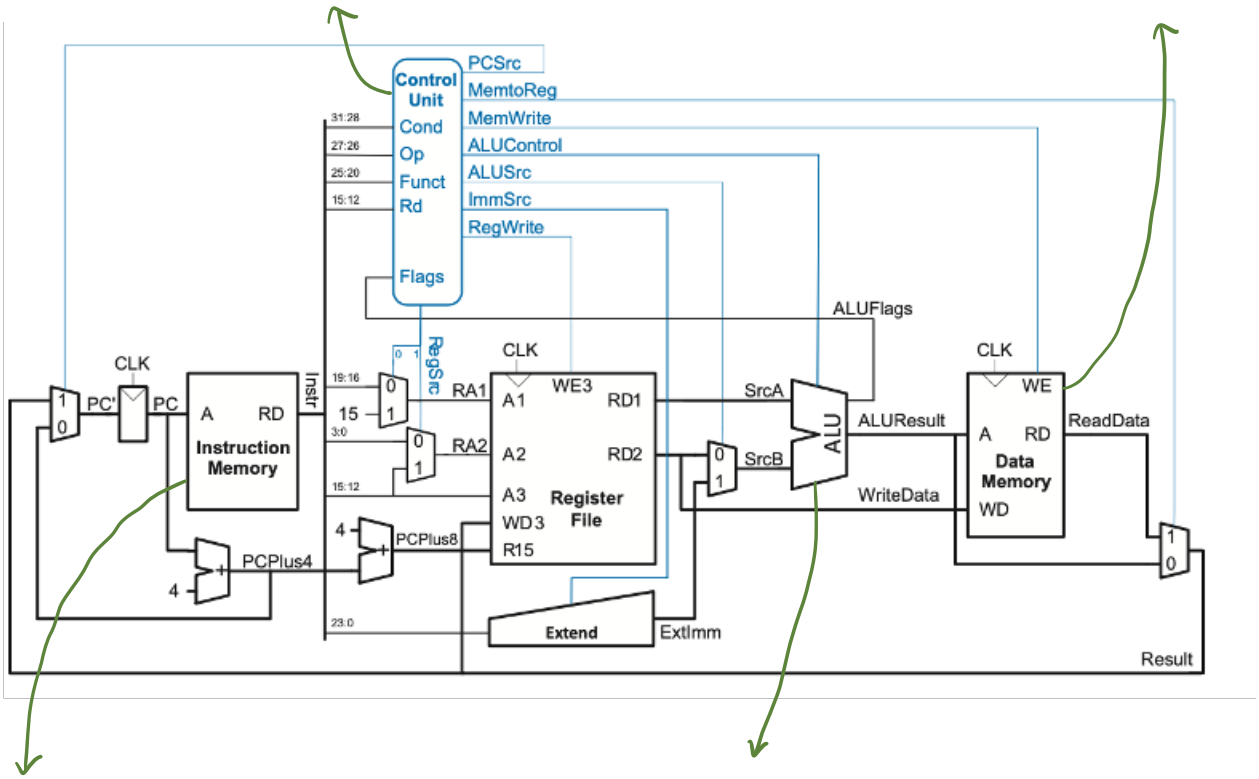
→ Fetch: hente instruksjonen som skal utføres
→ Decode: dekode instruksjonen - finne ut hva den er
→ Execute: utføre instruksjonen
→ write-back: skrive resultatet i minne

→ Når vi pipeliner, så vil hver hovedkomponent i CPUen jobbe hele tiden.

En forenkla versjon av en CPU:

dekoder instruksjonen og sender videre info om hva instruksjonen sier (decode)

skriver resultat til minne (write back)



henter fram instruksjonen (fetch)

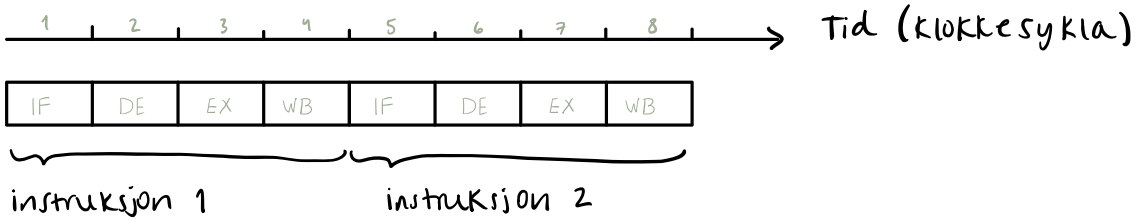
utfører instruksjonen (execute)

→ Uten pipeline hadde kun en og en komponent jobbet mens de andre hadde ventet. Dette er fordi hver del av CPU'en har ansvar for en del av utførelsen av instruksjonen, f. eks. instruction memory henta fram instruksjonen.

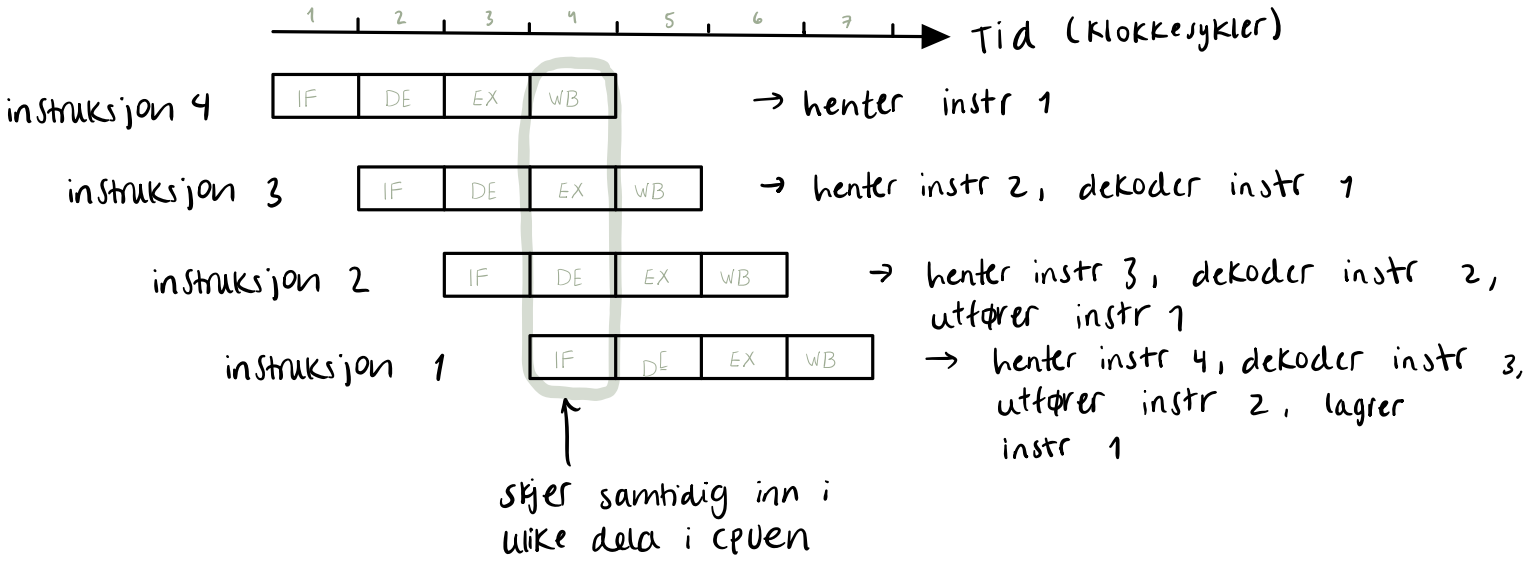
→ Med pipeline vil alle komponentene jobbe samtidig. Dette er mulig siden hver komponent jobber med sin egen instruksjon.

→ Dvs. at flere ting skjer i løpet av en klokkesykel.

→ Uten pipeline utfører vi hver instruksjon til de er ferdig før vi begynner på neste:



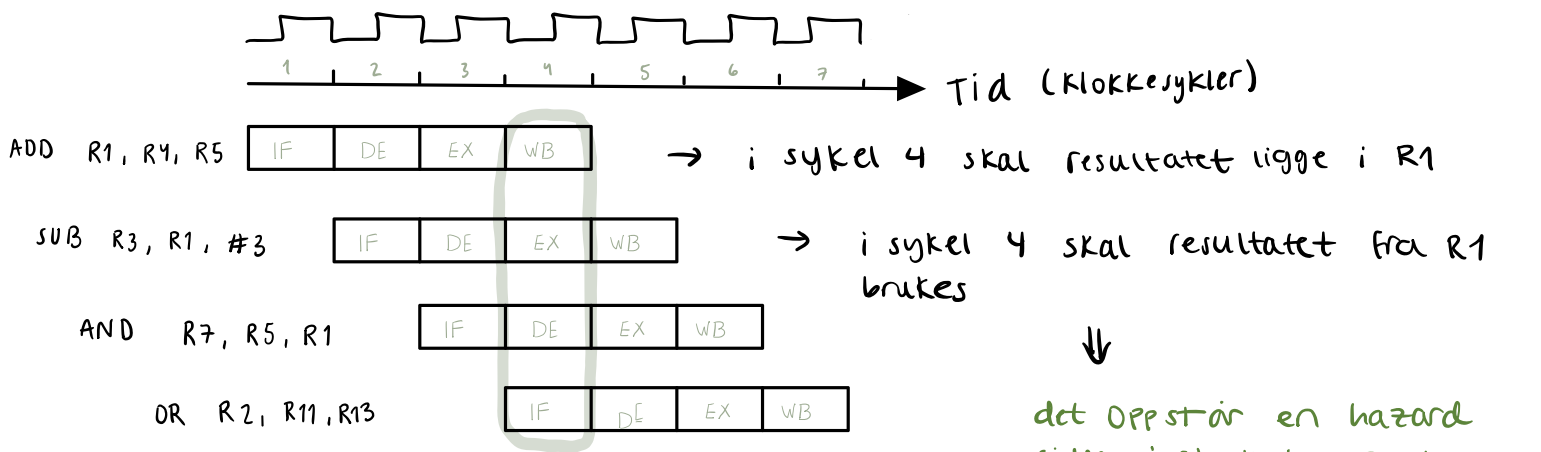
→ Med pipeline:



Men, når vi bruker pipeline kan det oppstå komplikasjoner som vi kaller hazards. En av de er når instruksjoner avhenger av hverandre. Det kan for eksempel si om en instruksjon utfører en instruksjon som trenger resultatet fra en tidligere instruksjon, der dette resultatet ikke har blitt skrevet til minne enda.

Eksempel: Vi har 4 instruksjona å gjøre etter hverandre:

- ADD R1, R4, R5 → R1 = R4 + R5 → instr 1
- SUB R3, R1, #3 → R3 = R1 + 3 → instr 2
- AND R7, R5, R1 → R7 = R5 and R1 → instr 3
- OR R2, R11, R13 → R2 = R11 or R13 → instr 4



⇓

det oppstår en hazard siden instruksjon 2 trenger tilgang til data som ikke enda er tilgjengelig

→ Siden det kan oppstå slike konflikter, så må vi ha noen måta å løse de på.

→ En av de kan være venting, slik at instruksjon 2 må vente for å bli utført til etter verdien blir lagra i minne.

→ En annen kan være at vi får write back til å skje på falling edge. Det er vanligst at instruksjona starta å kjøre på rising edge, som vil si at om vi lar "execute" skje på en rising edge og "write back" på falling edge, så kan disse skje i samme klokkesykel og da vil det være mulig for en instruksjon å bruke et register som et resultat blir lagt i i samme klokkesykel.

Minne:

→ Minnet i en datamaskin er inndelt i flere elementer, der hvert element har sitt spesifike bruksområde, kapasitet og hastighet.

→ Vi trenger flere nivåer av minne for det å hente fra minnet og legge til i minnet er veldig tregt. Om vi lagrer ulike typer data i ulike nivåer av minne, kan CPUen jobbe raskere.

→ **Flip-flop**:

→ kan bare lagre 1 bit av data om gangen
→ raskeste minneelement
→ enklest struktur

→ **Register**:

→ intern minne inne i CPUen
→ kan lagre en veldig liten mengde data
→ veldig veldig raskt minneaksessering
→ registrene resettes hver gang maskinen blir slått av
→ f.eks registrene i LMC
→ program counter: neste instruksjon som skal utføres
→ akkumulator: verdiene vi jobber med
→ instruksjonsregister: tallkoden til instruksjonen vi er på
→ address register: hvor i minnet vi skal aksessere

→ **Cache**:

- → veldig raskt minneaksessering
- → større kapasitet enn registre, men fortsatt ganske lite plass
- → kan deles inn i flere nivåer: L1 (ligger på prosessoren), L2 og L3 (ikke på prosessoren).
- → brukes som et mellomlager mellom CPUen og RAM
  - → Cache henter en haug med data fra RAM og lagrer det slik at CPUen raskt kan hente dataen uten å "bevege seg helt ut til RAM". Det gjør at ting går fortere.
- → Cache resettes hver gang maskinen blir slått av

- → Cache hit: Når data CPUen trenger ligger tilgjengelig i cache så CPUen kan bare hente data derfra.

- → Cache miss: Når data CPUen trenger <u>IKKE</u> ligger i cache, så det må letes i neste nivå av minne: RAM. Dette tar lenger tid siden det tar lenger tid å aksessere RAM.

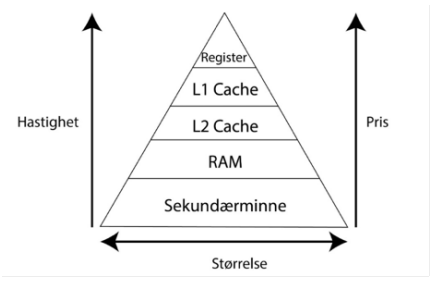
→ **RAM**:

- → større minneelement med en god del mer plass
- → tregere enn tidligere minneelementa
- → lagrer data i ubestemt rekkefølge og data aksesseres i random rekkefølge (derav navnet "random access memory")
- → RAM resettes når maskinen slås av
- → fungerer som et mellomlager mellom CPU og sekundærminne
- → ligger utenfor CPUen

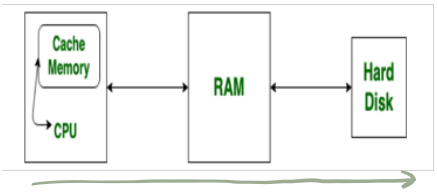
→ **Sekundærminne** (harddisk, SSD):

- → største minneelementet på maskinen
- → ligger utenfor CPUen og hovedkortet til maskinen
- → langtidslagring av data
- → er veldig tregt å aksessere siden det har stor kapasitet
- → dataen blir <u>ikke</u> slettet når maskinen skrus av

minne-hierarki



Rekkefølgen på minneaksesseringa



REGNEEKSEMPLER

Cache hit / miss: Anta at det er 3000 instruksjoner igjen å prosessere og at hver instruksjon tar 3 klokkesykler å prosessere. Legg til grunn en minneaksessering på 70% av instruksjonene. Vi forventa 50% cache hit som gir en økt forsinkelse på 7 klokkesykla og en cache miss på 47 klokkesykla i tillegg til prosesseringstiden.

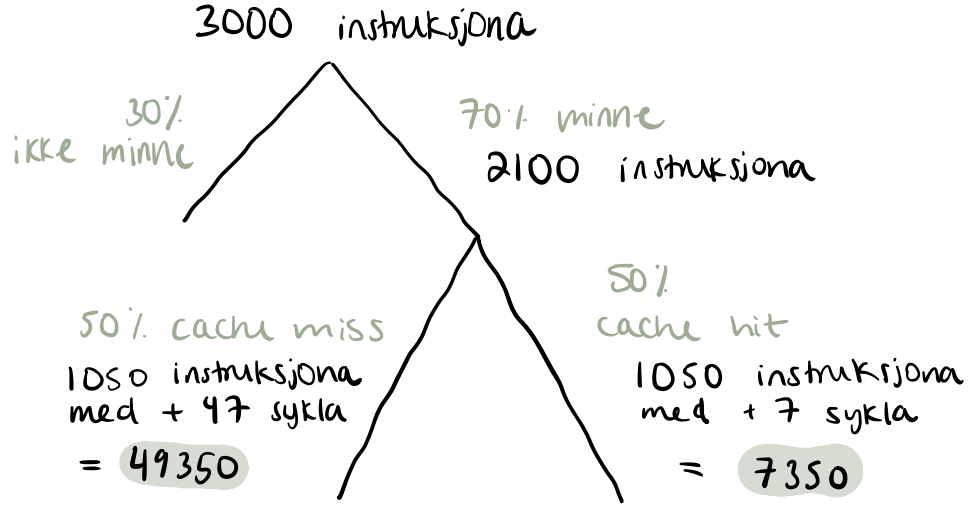
Hva blir total prosesseringstid i klokkesykla?

→ 3000 instruksjona der hver tar 3 klokkesykla → 9000 klokkesykla
(dette er tiden det tar å utføre instruksjonene uten å tenke på hvor lang tid det tar å aksessere minnet)

→ for resten av oppgava laga vi et tre:

1) minneaksessering

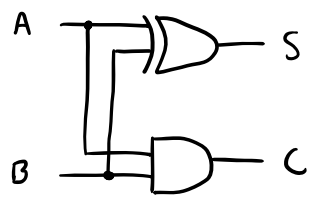
2) cache hit / miss



→ for de instruksjonene som må aksessere minne, så tar det 49350 + 7350 = 56700 sykla ekstra

→ total prosesseringstid 9000 + 56700 = 65 700 sykla

Frekvens som en krets kan operere på: Gitt en portforsinkelse på 5ps for hver port, hvilken frekvens vil en 1-bits halvadder kunne operere på?



→ den lengste pathen (med flest porta) mellom input og output inneholder 1 port, så den totale tidsforsinkelsen for kretsen er 5 ps.

→ vi antar at klokkesignalet vårt er 50% - 50%.

→ kretsen trenger at klokka er på 5 ps for at den skal klare å kjøre gjennom, så klokkeperioden må være 10 ps (5 ps på, 5 ps av)

Utregning av frekvens (1 / klokkeperiode = frekvens)

→ klokkeperiode = 10 ps = $1 \cdot 10^{-11}$ s

→ $f = \frac{1}{\text{klokkeperiode}} = \frac{1}{10^{-11}\,s} = 10^{11}\,Hz = 100 \cdot 10^{9} = 100\,GHz$ (giga)

→ Denne halvbitsadderen kan operere på en frekvens 100GHz, som vil si 10^{11} sykla per sekund.

Frekvens for utlesing av register: Gitt en forsinkelse på 100 ps for å lese ut data fra et register, hva er den høyeste frekvensen vi kan bruke for utlesing av et register?

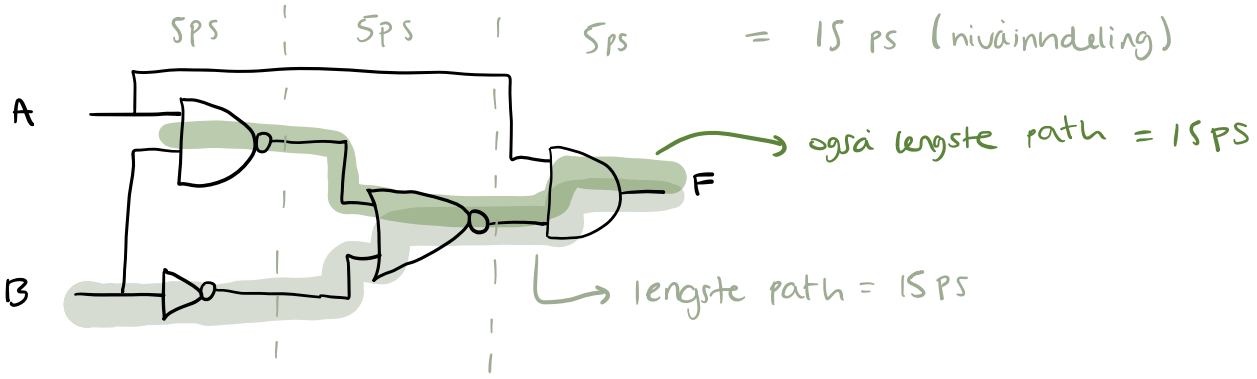
→ Hvis vi har en forsinkelse på 100 ps, så må klokkeperioden være 200 ps (antatt at vi har en 50% - 50% klokke).

Utregning:

→ 200 ps = $2 \cdot 10^{-10}$ s

→ $f = \frac{1}{\text{klokkeperiode}} = \frac{1}{2 \cdot 10^{-10}\,s} = 5 \cdot 10^{9}\,Hz = 5\,GHz$ (giga)

Frekvens som en krets kan operere på del 2: Gitt en portforsinkelse på 5ps for hver port, hvilken frekvens denne kretsen kunne operere på?



- Resultatet er gyldig etter 15 ps - klokka må være på i 15 ps.
- Klokka er da av i 15 ps (klokkesignal 50% - 50%)
- Klokkeperioden er 30 ps.
- 30 ps i s = $3 \cdot 10^{-11}$ s
- Frekvens:

$$F = \frac{1}{3 \cdot 10^{-11} s} = 0{,}333 \cdot 10^{11} \text{ Hz} \approx 33{,}3 \cdot 10^{9} \text{ Hz} = 33{,}3 \text{ GHz}$$