

Dagens temaer

- Fra kapittel 4 i 'Computer Organisation and Architecture'
- Kort om hurtigminne (RAM)
- Organisering av CPU: von Neuman-modellen
- Register Transfer Language (RTL)
- Instruksjonseksekvering
- Pipelining

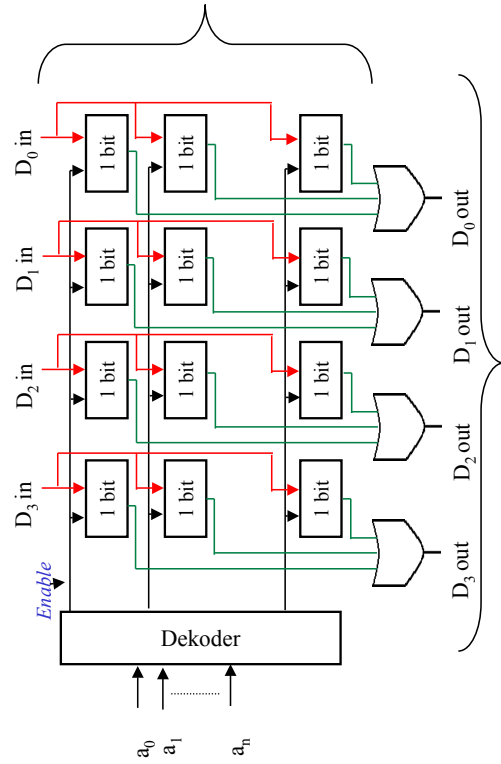
Oppbygging av RAM

- Sentrale begreper er *adresserbarhet* og *adresserom*
- *Adresserbarhet*: Antall bit som prosessoren kan tak samtidig i én operasjon (lese- eller skrive-operasjon).
- Antall bit som kan refereres til i én operasjon kalles CPU'ens *ordlengde*, mens bit'ene betraktet under ett kalles vanligvis et *ord*.
- De fleste CPU'er kan også referere til *halvord* (halvparten antall bit i et ord) og *byte* (8 bit).
- *Adresserom*: Hvor mange unike ord en CPU kan referere til totalt; dvs. hvor mye hurtigminne (RAM) CPU'en kan nyttiggjøre seg av.
- Adresserommet er bestemt av hvor mange *adresselinjer* CPU'en har.

Oppbygging av RAM (forts.)

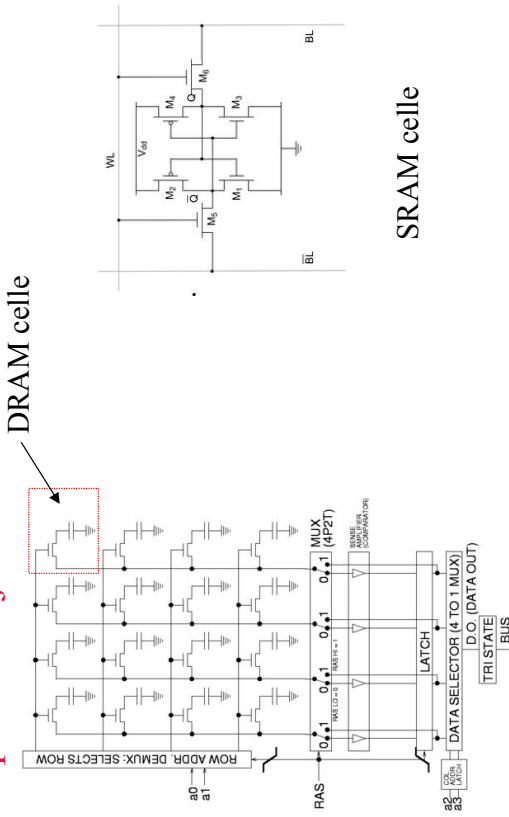
- 8 adresselinjer kan adressere $2^8=256$ ord.
- 28 adresselinjer kan adressere $2^{28}=268,435,456$ ord, dvs **256 M** ord.
- 32 adresselinjer kan adressere $2^{32}=4,294,967,296$ ord, dvs **4 G** ord
- NB: Ingen sammenheng mellom antall ulike ord som kan adresseres, og antall bit i hvert ord
- Men: Det er vanlig å angi minnestørrelse i antall byte istedenfor enn i antall ord, dvs hvor mange megabyte eller gigabyte hukommelsen har
- Logisk sett er RAM organisert som en $n \times m$ matrise av 1-bits celler hvor man kan velge ut én rad ad gangen for skrivev/lesing, og der n er ordlengden, og m er $2^{\text{antall adresselinjer}}$

Adresserom: Antall bit linjer



Adresserbarhet: Antall bit horisontalt

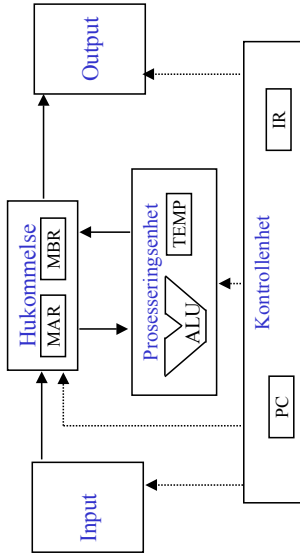
Implementasjon



SRAM celle

Von Neumann modellen

- Enkel modell fra slutten av 1940-tallet for organisering av en datamaskin i fem ulike enheter:



- Nesten alle dagens maskiner har denne organiseringen.

Hukommelse

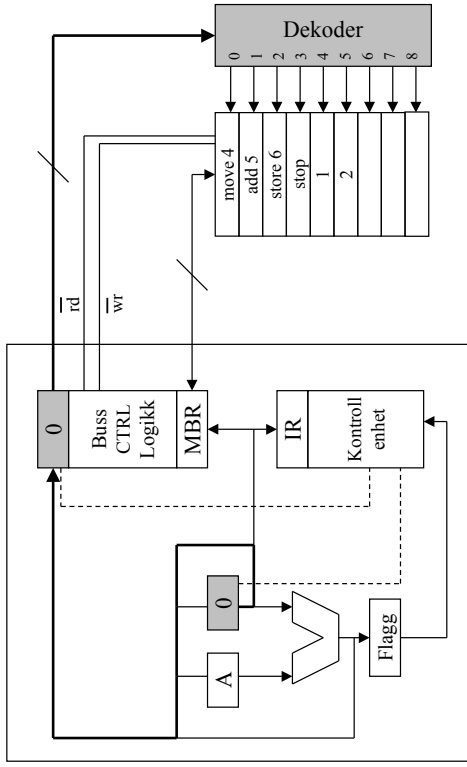
- Hukommelsen er koblet til resten gjennom *adressebussen* og *databussen*.
- Innholdet på adressebussen identifiserer et ord i hukommelsen (ved lesning skriving fra/til hukommelsen).
- Databussen "transporter" bitmønsteret som enten er lest fra eller skal skrives inn til en bestemt lokasjon i hukommelsen.
- Ved lesing legges adressen til ordet som skal leses først inn i MAR (Memory Address Register). Deretter blir ordet som er lest fra hukommelsen lagt over i MBR (Memory Buffer Register). MBR kalles også MDR (Memory Data Register)
- Ved skriving legges adressen til lokasjonen det skal skrives til i MAR, mens bitmønsteret som skal skrives inn i hukommelsen legges først inn i MBR.
- MAR og MBR er grensesnittet mellom prosesserings-enheten og hukommelsen, dvs prosesseringsenheten forholder seg ved minneaksess kun til disse.

Prosesseringsenheten

- Prosesseringsenheten består av en eller flere ALU'er og registre.
- ALU'en utfører aritmetiske og logiske beregninger på variable som ligger i registrene, og beregner adresser under programeksekvering
- ALU'en utfører gjemne operasjoner på ord av samme lengde som maskinens ordlengde, og spesial-operasjoner på byte og halvord.
- Siden ALU'en henter input fra registre, og skrives resultatene også til registre (ikke direkte til/fra hukommelse) blir prosesseringen mer effektiv med mange registre, minneaksess mot langsom RAM unngås
- De fleste prosessorer bruker en dedikert ALU for å gjøre adresseberegninger (f.eks hente neste instruksjon og finne adresser ved hoppinstruksjoner).
- Registre bygges med D-flipflop'er, mens resten av RAM/cache er bygget opp med en annen type teknologi basert på DRAM



Beregne startadressen og øk programtelleren med 1: $[IP] \leftarrow [IP] + 1$



2005

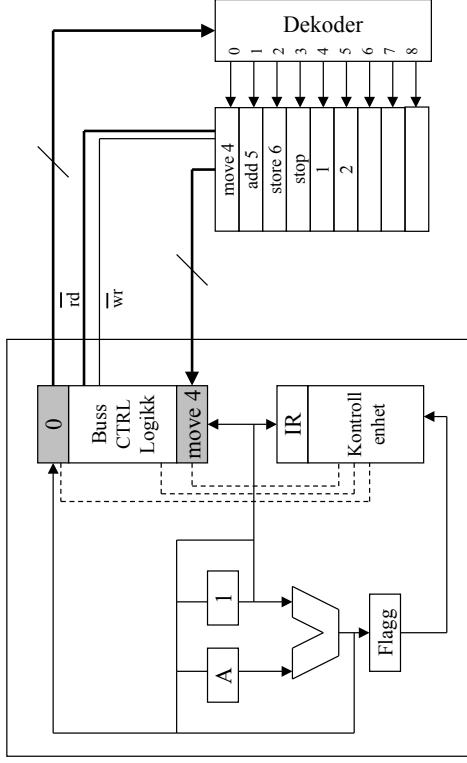
13

INF 1070

19.02.2006



Last inn lokasjon 0 inn i MBR: $[MBR] \leftarrow [M([MAR])]$



2005

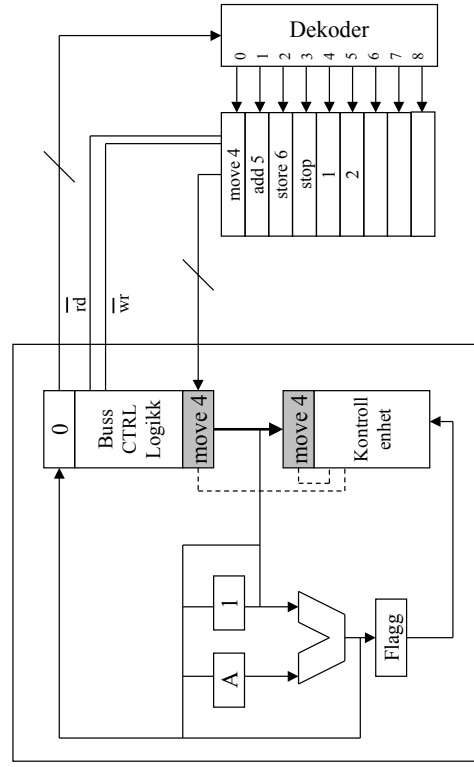
14

INF 1070

19.02.2006



Hent ut instruksjonskoden fra instruksjonsregisteret: $CU \leftarrow [IR(opcode)]$



2005

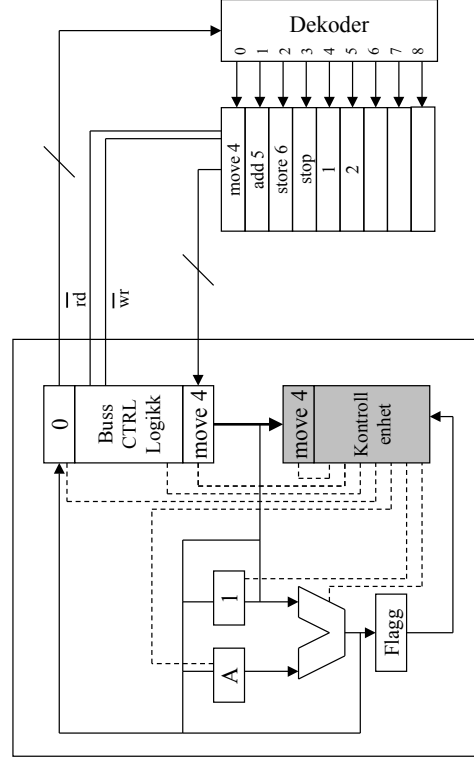
15

INF 1070

19.02.2006



Hent ut instruksjonskoden fra instruksjonsregisteret og sett riktige verdier på kontrollsignalene: $CU \leftarrow [IR(opcode)]$



2005

16

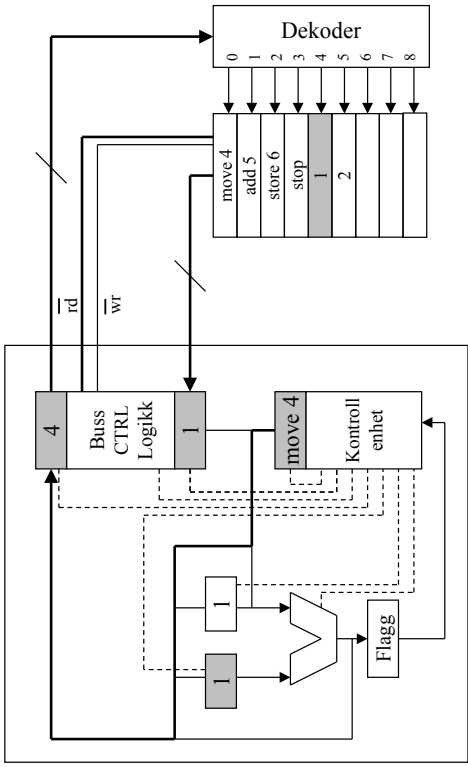
INF 1070

19.02.2006

Utfør instruksjonen 'move 4' ved å gjøre følgende:

- Hent ut adressen til operanden ved å dekode operandfeltet av instruksjonen:
[MAR] ← [IR(operand-adressen)]
- På samme måte som ved henting av instruksjonen hentes inn operanden (fra lokasjon 4 slik det er angitt i instruksjonen): [MBR] ← [M(4)]
- Overfør innholdet fra MBR til register A: [A] ← [MBR]

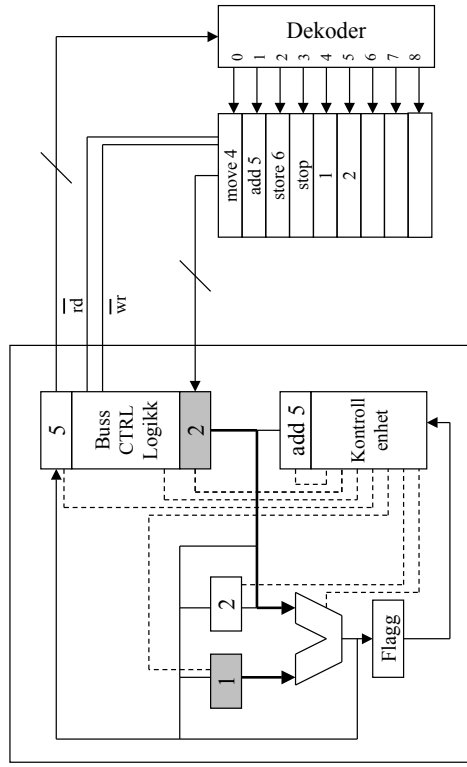
Efter at instruksjonen 'move 4' er eksekvert:



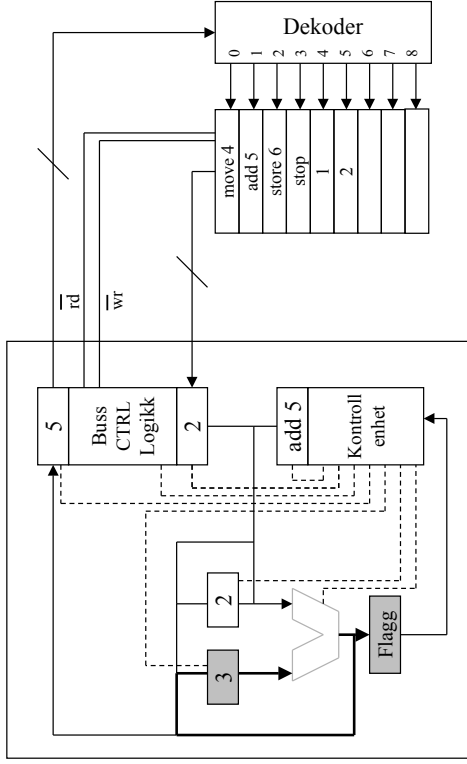
Utfør instruksjonen 'add 5' ved å gjøre følgende:

- Hent ut adressen til operanden ved å dekode operandfeltet av instruksjonen:
[MAR] ← [IR(operand-adressen)]
- På samme måte som ved henting av instruksjonen hentes inn operanden (fra lokasjon 4 slik det er angitt i instruksjonen): [MBR] ← [M(5)]
(NB: Feil i læreboka)
- Overfør operandene til ALUens to input: ALU ← [A]; ALU ← [MBR]
- Etter at addisjonen er utført må resultatet lagres i register A: [A] ← ALU

Efter at instruksjonen 'add 5' er eksekvert men før resultatet er lagret og flagg satt:



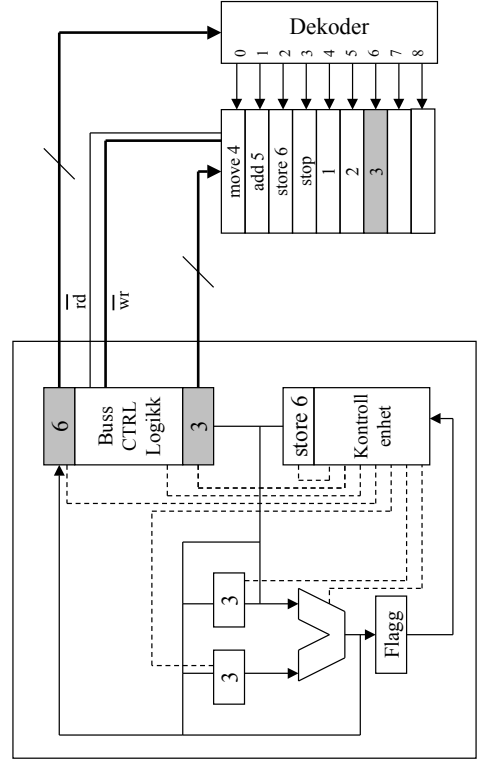
Etter at resultatet er lagret og flagg satt:



Utfør instruksjonen 'store 6' ved å gjøre følgende:

- Hent ut adressen til der resultatet skal skrives ved å dekode operandfeltet av instruksjonen: [MAR] ← [IR(operand-adressen)]
- Kopier innholdet av A til MBR: [MBR] ← [A]
- Kopier innholdet av MBR til lokasjon 6: [M(6)] ← [MBR]

Etter at 'store 6' er utført:



Ytelsesforbedring

- To viktige teknikker for å øke hastigheten til en CPU er *pipelining* og *cache*
- **Pipelining:** Starte eksekvering av en ny instruksjon hver klokkesykel
- **Cache:** Egen type hurtigminne som nesten er like rask som interne registre, men med mye større kapasitet, dog ikke like mye som RAM
- Dessuten benyttes også **parallel-eksekvering** benyttes også der det er mulig, spesielt internt i CPU-hardware (superskalare maskiner)
- **Men:** En av de viktigste årsakene til at maskiner blir raskere er allikevel at transistorene blir mindre, trekker mindre strøm, og kan endre tilstand raskere

Pipelining

- Kan sammenlignes med samlebåndproduksjon:
 - . I stedet for å vente til forrige instruksjon er ferdig eksekvert, setter man i gang neste instruksjon så fort som første steg av forrige instruksjon er ferdig.
- Med pipelining øker man antallet instruksjoner som blir ferdig eksekvert per tidsenhet, **men**: hver instruksjon tar fortsatt like lang tid
- Forutsetningen for at pipelining er at hver enhet som utfører en del av en instruksjon arbeider uavhengig av de andre enhetene i pipelinen
- Gitt en prosessor hvor hver instruksjon består av fire steg:

FETCH (Hent instruksjon)

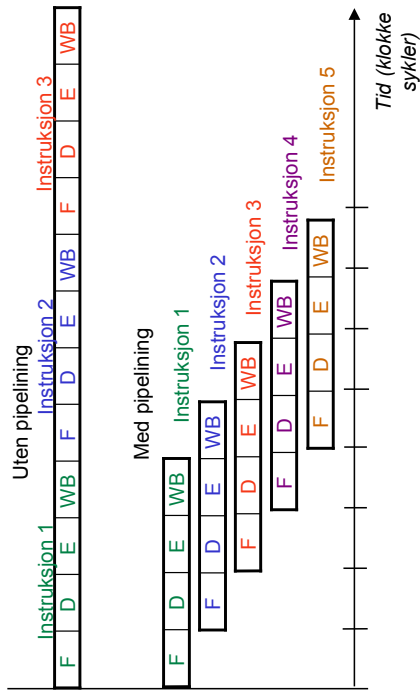
DECODE (Dekod instruksjonen)

EXECUTE (Utfør instruksjonen)

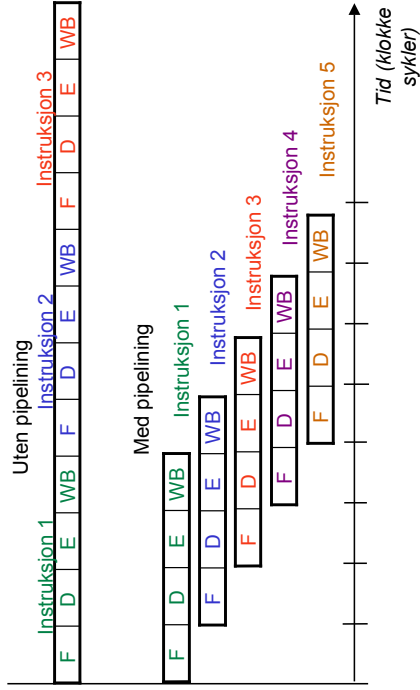
WRITE BACK (Skriv resultatet til minne)

Pipelining (forts)

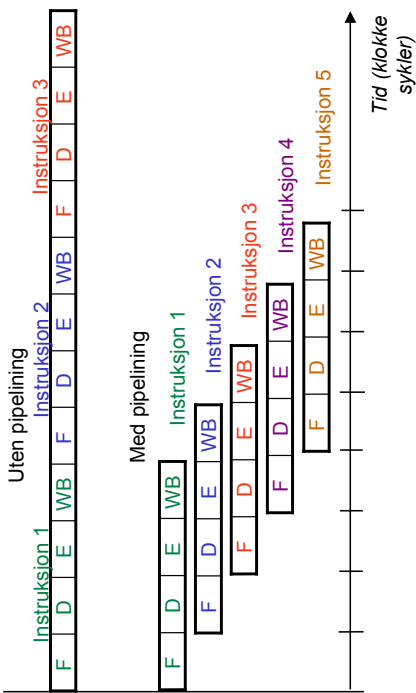
- Antar at det finnes 4 uavhengige hardware-enheter som kan utføre hver av disse stegene uten å benytte de andre delene:
 - . For eksempel skal FETCH ikke trenge å **bruke** EXECUTE-enheten for å hente en instruksjon eller variabel (men EXECUTE er avhengig av **input** fra DECODE-steget, og DECODE-steget er avhengig av input fra FETCH-steget osv)
- Alle enhetene kunne jobbe i parallell, med hver sine steg fra **ulike** instruksjoner, uten å gå i "beina på hverandre".



- Uten pipelining kan FETCH-steget til instruksjon 2 først kan starte etter WRITE-BACK steget til instruksjon 1
- Med pipelining starter FETCH-steget til instruksjon 2 rett etter at FETCH-steget til instruksjon 1 er ferdig



- Uten pipelining er det kun ett steg fra én instruksjon som prosesseres ad gangen i prosessoren.
- Med pipelining er det opptil 4 steg fra **forskjellige** instruksjoner som prosesseres i parallell. I klokkesykel 4 utføres **WRITE-BACK** fra instruksjon 2, **EXECUTE** fra instruksjon 3, **DECODE** fra instruksjon 4 og **FETCH** fra instruksjon nummer 5



- **Uten** pipelining avsluttes en instruksjon hver fjerde klokkesykel.
- **Med** pipelining er instruksjon 1 ferdig etter 3. klokkesykel, instruksjon 2 ferdig etter 4. klokkesykel osv. Mao: En instruksjon er ferdig hver klokkesykel.