

## Prim rminne

Geir Ove Rosvold

6. januar 2016

Opphavsrett: Forfatter og Stiftelsen TISIP

## Prim rminne

*Resym : I denne leksjonen diskuteres prim rminne. Vi ser p  hovedtypene ROM og RAM. Minneorganisering, b de internt p  brikkene og eksternt (hvordan vi setter sammen flere brikker til et fullstendig prim rminne) presenteres. Vi ser at det finnes to viktige RAM-teknologier, nemlig DRAM og SRAM. Styrker og svakheter med hver av dem diskuteres.*

### Innhold

1.1.	PRIM�RMINNE .....	2
1.1.1.	Bakgrunn .....	2
1.2.	ROM .....	3
1.3.	RAM .....	4
1.3.1.	DRAM og SRAM .....	4
1.4.	RAM KONTRA ROM I DAGENS MASKINER .....	5
1.5.	MINNEBRIKKENE .....	5
1.5.1.	Minneceller .....	5
1.5.2.	Virkem�te .....	6
1.5.3.	Dekoder .....	6
1.5.4.	Ordbreddeorganisering .....	7
1.5.5.	Kvadratisk organisering .....	7
1.5.6.	Andre organiseringer av minnebrikken .....	9
1.6.	MODULER OG MINNEKAPASITET .....	9
1.6.1.	Organisering av minne p� PC; SIMM og DIMM .....	10
1.6.2.	Forskjellige DRAM-typer .....	10
1.7.	HVOR OFTE �NSKER VI � LESE FRA MINNET? .....	10

## 1.1. Prim rminne

I en datamaskin finnes det mange forskjellige lagertyper. S  langt i kurset har vi i hvert fall nevnt registrene, prim rminnet (eller bare minnet) og harddisk.

De lagertypene som kan aksesseres direkte av CPUen kalles internt minne. Dette er registrene, cache (som vi skal se p  i en annen leksjon), og prim rminnet. Andre lagertyper kan ikke aksesseres direkte fra CPU, men m  aksesseres via en I/O-modul. Slike lagertyper kalles eksternt minne eller sekund rlager. Eksempler p  dette er disk og CD-ROM.

I denne leksjonen skal vi se p  prim rminnet. I senere leksjoner ser vi p  de  vrige lagertyper.

Prim rminne er en sv rt viktig del av datamaskinen. Ytelsen og st rrelsen p  prim rminnet bidrar sterkt til   bestemme ytelsen til hele datamaskinen. En av grunnene til dette er at n r et program utf res s  ligger instruksjonene i minnet, og instruksjonene blir overf rt til CPU og utf rt en etter en.

Utf ringen av enhver instruksjon medf rer alts  minst en minneaksess; nemlig for   hente instruksjonen. Mange instruksjoner inneb rer dessuten enten at data skal hentes fra minne og brukes i instruksjonens beregninger, eller at resultatet av beregningen skal skrives til minnet. Derfor f r vi ofte flere minneaksesser pr instruksjon.

Dette er grunnen til at det er vitalt at prim rminnet greier   levere instruksjoner og data tilstrekkelig kjapt. Vi kaller tiden det tar   hente data fra minnet for minnets aksesstid. Det er alts  viktig at minnet har en aksesstid som er s  kort at det greier   holde tritt med CPUens behov. Hvis dette ikke er tilfellet vil CPU kaste bort tid p    vente p  minnet.

I denne leksjonen skal vi se at det p  moderne maskiner forholder seg slik at CPUen er altfor rask til at minnet greier   holde f lge. Den vanligste m ten   takle denne situasjonen p  skal vi se p  i neste leksjon; nemlig   bruke cache.

### 1.1.1. Bakgrunn

Opp gjennom  rene har det v rt brukt mange forskjellige minneteknologier. N  er halvlederminne ener dende.

En av de teknologiene som tidligere var mye brukt, var s kalte ferittkjernelager eller *core*. Slike lager var bygget opp av sm  metalloksyd-ringer med diameter p  rundt 1 mm. En slik ring kan magnetiseres i en av to retninger, og den ene veien ble brukt som 0 den andre som 1. Hver ring lagret alts  verdien til en bit. Gjennom hver ring ble det tredd tynne ledninger som ble brukt til lesing og skrivning.

Core var alts  bygget opp av enheter (sm  ringer) som lagret en bit hver. Moderne halvlederminne er ogs  bygget opp av sm  enheter - vi kaller dem celler - som lagrer hver sin bit. Men halvlederminne er fremstilt som integrerte kretser, ICer, og vi kan derfor ikke se hver enkelt celle. Kretsene inneholder en stor mengde celler - antallet celler m les i megabits eller i gigabits. Det vil si millioner eller milliarder av bits i hver minnebrikke.

Vi skiller mellom to hovedtyper minne, nemlig ROM og RAM. ROM er en forkortelse for *Read Only Memory*, og er alts  minne som kun kan leses og ikke skrives. RAM st r for *Random Access Memory* og er minne som b de kan leses og skrives. En datamaskin inneholder begge typer minne.

## 1.2. ROM

ROM er en fellesbetegnelse for minne som det normalt bare kan leses fra. Innholdet av ROM-minne bevares hvis strømmen forsvinner. Program som ligger i ROM trenger derfor ikke lastes fra disk.

Noen eksempler på hva ROM brukes til er:

- BIOS (Basic I/O System) på PC. Dette er maskinnær programvare som operativsystemet kan bruke. BIOS inneholder også oppstartsrutiner som utføres når PCen startes.
- Lagring av *mikroprogram*. Dette skal vi diskutere i senere leksjoner.

ROM brukes langt mindre nå enn tidligere. Tidligere var ROM-minne raskere enn RAM. Dette har nå endret seg, og moderne RAM er raskere enn moderne ROM. ROM brukes stadig i stor utstrekning i ikke-generelle datamaskiner som kontrollere, måleutstyr, leketøy og spill, og hundrevis av andre anvendelser som verken er PC, RISC, mini- eller stormaskin.

Den opprinnelige ROM-brikken var slik at minneinnholdet ble lagt inn under fremstilling av brikken, og innholdet kunne ikke endres senere. Dette hadde blant annet de svakhetene at de måtte bestilles i store serier og at innholdet burde være riktig ved første forsøk.

Det var selvfølgelig ønskelig at minneinnholdet kunne legges inn på et senere tidspunkt, og derfor ble såkalte *Programmerbare ROM* (PROM)-brikker utviklet. Med riktig utstyr kan brukeren programmere slike brikker én, og bare én, gang. En ny PROM-brikke har 0-ere i alle lokasjoner, og med hjelp av en "PROM-brenner" kan man legge inn 1-ere der man ønsker. Dette gjøres ved å tilføre en spenning som er høyere enn de som brukes i datamaskinen slik at en tynt isolasjonslag brennes av i de ønskede lokasjonene.

Neste trinn var en brikke hvor innholdet kunne slettes og skrives på nytt. Intel løste dette problemet i 1971, og startet produksjon av EPROM-brikker (*Erasable PROM*). På disse brikkene finnes det et gjennomskiktig vindu på toppen, og innholdet kan nullstilles ved å belyse brikken med UV-lys gjennom dette vinduet. Senere kan nytt innhold skrives på samme måte som for PROM. Når vi sletter en slik brikke fjernes altså all informasjon på brikken. En EPROM kan slettes og skrives på nytt mange ganger, men den er dyrere enn en PROM.

Senere har Texas Instrument utviklet EEPROM-brikken, eller *Electrically EPROM*, altså en brikke som kan slettes elektronisk. Denne har mange fordeler. Man kan slette deler av innholdet, det vil si at bare de adresserte områdene slettes og at resten ikke endres. Dessuten kan både sletting og skriving utføres uten bruk av spesialutstyr. Det betyr at både lesing, sletting og skriving kan skje uten å ta ut brikken fra maskinen - man bruker de ordinære busslinjene og den ordinære spenningen i maskinen. Det tar imidlertid mye lengre tid å slette enn å lese fra brikken. Sletting kan ta flere hundre millisekund, mens lesing tar et par hundre nanosekund. EEPROM er dyrere enn EPROM og hver celle tar større plass slik at man ikke får lagret like mange bit på hver brikke.

Den nyeste ROM-teknologien er *Flash-memory*. Denne kom på midten av 80-tallet, og her benyttes også elektrisk sletting. Slettingen går mye fortere enn på en EEPROM, men man kan ikke slette på byte-nivå slik som på EEPROM. Isteden må man slette blokker av data. Størrelsen på disse blokkene varierer fra brikketype til brikketype. Hver celle er liten, slik at en flash-memory-brikke lagrer mer enn en EEPROM med samme fysiske størrelse.

Etter hvert har prisen på Flash sunket så mye at vi kan bruke ganske mye av det. Først kom såkalte minnepenner. Disse har vanligvis et USB-grensesnitt, og operativsystemet kan presentere disse som en liten harddisk. I de senere årene har det også kommet SSD-disker, der

Flash-minne erstatter det magnetiske lagringsmediet i en harddisk. Disse har samme grensesnitt mot datamaskinen som de tradisjonelle harddiskene, for eksempel SATA. Forel pig kan ikke lagringskapasiteten konkurrere med de magnetiske harddiskene, men aksesstiden er mye bedre p  SSD-disker enn p  magnetiske disk.

For EPROM, EEPROM, flash-memory er betegnelsen ROM egentlig misvisende, siden de ogs  kan skrives. Et bedre navn har kanskje v rt Read Mostly Memory. Navnet ROM er likevel i vanlig bruk.

### 1.3. RAM

RAM er en forkortelse for *Random Access Memory*. RAM er minne som b de kan leses og skrives. Minnet er avhengig av kontinuerlig drivspenning, og hvis str mmen forsvinner g r all informasjon i minnet tapt.

Random betyr vilk rlig, og navnet sier egentlig ikke annet enn at man uten videre kan lese eller skrive en hvilken som helst plass i minnet (uten   for eksempel starte forfra slik man m  p  et b nd). Igjen har vi ogs  et misvisende navn - ROM kan jo ogs  leses vilk rlig - men navnet er s  innarbeidet at alle bruker det.

#### 1.3.1. DRAM og SRAM

Vi har to hovedtyper av RAM, nemlig Statisk RAM (eller SRAM) og Dynamisk RAM (DRAM). I Statisk RAM er hver enkelt celle en stabil transistor-krets - en s kalt FlipFlop eller bistabil vippe. At den er stabil betyr at den er konstruert slik at informasjonen huskes s  lenge drivspenningen ikke forsvinner.

I DRAM er hver celle bygget opp rundt en kondensator isteden. N r man skriver til cellen vil kondensatoren utsettes for en spenning. Kondensatoren lades opp, og n r spenningen forsvinner vil kondensatoren "huske" spenningen en liten stund - inntil kondensatoren lades ut. Siden kondensatoren husker spenningen i sv rt kort tid, m  informasjonen stadig vekk oppdateres. Oppdateringen skjer med en egen gjennoppfrisknings-elektronikk som lader opp cellene et visst antall gang i sekundet. Man skulle tro at en slik gjennoppfrisknings-krets ville fordyre minnet betraktelig, og kanskje ogs  v re en potensiell feilkilde. Dette er imidlertid i liten grad tilfellet. Man har produsert slike kretser i mange  r, s  teknikken er b de billig og med sv rt lav feilrate.

Til tross for at den m  utstyres med elektronikk for gjennoppfriskning er DRAM langt billigere enn SRAM, men det er ogs  tregere. Hver enkelt celle er mindre p  DRAM enn p  SRAM slik at hver brikke kan inneholde flere celler og dermed ha st rre lagringskapasitet. P  de fleste datamaskiner benyttes DRAM i prim rlageret, mens SRAM brukes til cache.

#### Aksesstider for DRAM og SRAM

Aksesstiden, eller tilgangstiden, er den tiden minnet bruker p    fremskaffe  nskede data.

For DRAM er som regel aksessiden ca 35-70 ns.

For SRAM er som regel tiden mellom 1 og 10 ns.

## 1.4. RAM kontra ROM i dagens maskiner

Moderne datamaskiner benytter sjelden ROM for systemrutiner og biblioteksrutiner. Dette var vanlig tidligere, men siden RAM er raskere enn ROM vil det sinke systemet. P  ordin re datamaskiner, for eksempel PC, er ROM mest brukt i forbindelse med booting; alts  oppstart av datamaskinen. Datamaskinen m  jo lese oppstartsprogrammet fra ROM siden RAM-minnet er tomt n r str mmen skrur p .

P  en PC ligger disse oppstartsrutinene i en del av minnet som kalles ROM BIOS. BIOS st r for Basic I/O System. I ROM BIOS ligger det ogs  mange systemrutiner som kan brukes av operativsystemet. DOS brukte disse BIOS-rutinene sv rt mye. Moderne operativsystem, som Windows og Linux, har sine egne systemrutiner og bruker ikke BIOS-rutinene.

Som tidligere nevnt brukes ROM i stor utstrekning i ikke-generelle datamaskiner som digitale kamera, mp3-spillere, kontrollere, m leutstyr, leket y, spill og mange andre anvendelser. Det brukes ogs  til mer utradisjonelle lagringsmedia, som usb-penner.

## 1.5. Minnebrikkene

Vanligvis ser vi p  prim rminnet som en enhet. Denne enheten er organisert slik at den best r av mange adresserbare lokasjoner. Hver lokasjon kan lagre et ord. Det er vanlig   oppgi lagerkapasiteten som det antall *bytes* prim rminnet kan lagre - selv om ordbredden kan v re en annen enn 8 bits.

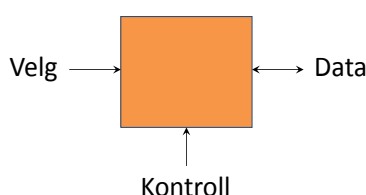
I de fleste sammenhenger er det greit   se p  prim rminnet som en enhet, men i denne leksjonen skal vi " pne minnet" og se hvordan enheten er bygget opp. Da vil vi se at prim rminnet er bygget opp av mange integrerte kretser. Denne typen integrerte kretser kaller vi minnebrikker.

### 1.5.1. Minneceller

Minnebrikkene er integrerte kretser som best r av mange celler som hver kan lagre en bit. Hver celle har tre tilkoblingspunkter hvor signaler sendes til eller leses fra cellen. Disse tre tilkoblingspunktene er:

- *Velg* er et bin rt signal som settes n r akkurat denne cellen skal leses eller skrives
- *Kontroll* er et bin rt signal som avgj r om det skal skrives til cella eller om innholdet av cella skal leses
- *Data* er det siste signalet. Ved skriving p  cella kommer data inn til cella p  denne linjen. Skal cella leses, s  legges data ut p  denne linjen.

Figur 1 viser en skjematisk fremstilling av en celle med tilh rende signaler. Retningen p  signalene fremg r ogs  av figuren.



**Figur 1.** Figuren viser en celle med tilh rende signaler. Cellen kan lagre en bit.

### 1.5.2. Virkemåte

Når vi skal forstå virkemåten til minnet er det viktig å huske at:

1. Utad er minnet bygget opp som en lang rekke lokasjoner. Hver lokasjon har plass til *ett ord*. Det vil si at hver lokasjon består av like mange minneceller som det er bits i ordet. For eksempel kan vi ha en ordbredde på 64 bits. På en moderne PC består primærminnet av flere hundre millioner slike 64-bits lokasjoner.
2. Vi kan lese eller skrive hver enkelt minnelokasjon fordi hver lokasjon har en adresse. Når vi skal aksessere (det vil si lese eller skrive) en minnelokasjon settes *Velg*-signalet til cellene i denne lokasjonen. *Velg*-signalet til alle andre celler settes til null.
3. Samtidig settes også *Kontroll*-signalet til cellene i lokasjonen. Dette forteller cellen om det er lesing eller skriving som skal skje.
4. Dersom vi vil skrive til cella, legges den binære verdien som skal skrives på cellens data-signal. Dersom vi skal lese data fra cellen, legger cellen sin binære verdi ut på data-signalet sitt.

Sett fra CPU foregår en overføring på følgende måte (ref: leksjonen om busser):

- Når CPU skal lese fra en bestemt lokasjon i minnet, så legger CPU adressen til lokasjonen ut på adressebussen. Minnet vil lete frem rett lokasjon og legge den på databussen.
- Når CPU skal skrive til en lokasjon i minnet, så legger CPU adressen til lokasjonen ut på adressebussen, og data som skal skrives på databussen. Minnet vil legge innholdet fra databussen på den adresserte lokasjonen.

### 1.5.3. Dekoder

Hvordan vet vi akkurat hvilke celler som skal aksessereres? Jo, det ser vi av adressen til minnelokasjonen. Adressen på adressebussen skal altså bestemme hvilke celler vi skal sette *Velg*-signalet på.

Adressen på adressebussen er et bitmønster som består av et visst antall bits, og ut fra dette må vi få laget et signal som setter *Velg*-signalet på de riktige cellene i lagerbrikken. Til dette brukes en dekode.

En dekode er en elektronisk krets som har en adresse som innganger og et antall binære signaler som utganger. En eneste utgang er til enhver tid høy, og alle andre er lave. Hvilken utgang som er høy er avhengig av hvilken adresse som ligger på inngangene.

En en-bits dekode har én inngang og to utganger, fordi vi med en bit kan velge mellom to utganger. En to-bits dekode har to innganger og fire utganger, fordi vi med to bits kan velge mellom fire utganger. En trebits dekode har 8 utganger. Og så videre.

Adressen på inngangen er altså et bitmønster som består av en blanding av 0-er og 1-ere. På utgangssiden er bare ett av signalene høyt; nemlig akkurat det utgangssignalet som er adressert.

Som eksempel ser vi på en åttebits dekode. Den har en åttebits adresse som input, og 256 utgangssignaler nummerert fra 0 til 255. Det adresserte utgangssignalet er høyt, de øvrige er lav. Hvis for eksempel bitmønsteret  $0101011_2 = 43_{10}$  ligger på inngangen, vil utgangssignal nummer  $43_{10}$  være høyt og alle de andre vil være lav.

Figur 2 viser virkemåten ved lesing av primærminnet.

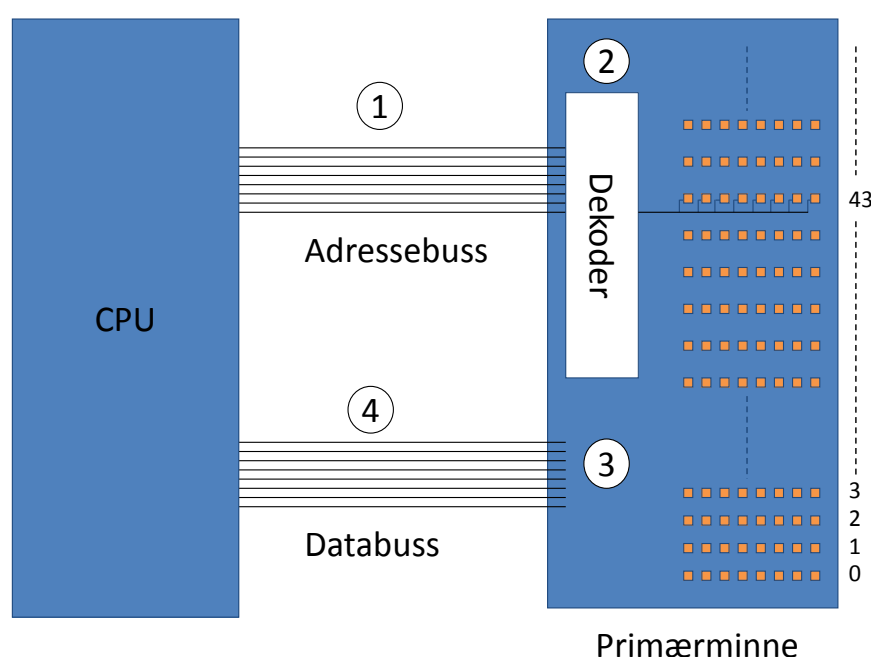
#### 1.5.4. Ordbreddeorganisering

Hvis vi lager et prim rminne p  den m ten som er vist i Figur 2 s  f r hele prim rminnet plass i en eneste brikke.

Dette er en vanlig m te   organisere ROM-brikker p . Fordelen er at alle bitene som inng r i et ord ligger i en og samme brikke.

Teknikken har imidlertid en alvorlig ulempe. Den krever en sv rt avansert dekodeer hvis minnebrikken skal inneholde mange lokasjoner. For eksempel: Antar vi en ordbredde p  en byte, og at minnebrikken skal kunne lagre 1 KByte, m  vi ha en 10 bits dekodeer fordi  $1\text{ K} = 2^{10} = 1024$ . Skal vi lagre 4 KByte p  brikken m  vi ha en 12 bits dekodeer.

Denne m ten   organisere cellene p  kalles ordbreddeorganisering. Dette fordi hele ordet f r plass i en og samme brikke.



**Figur 2. Lesing fra minnet.** Figuren viser et prim rminne med en rekke adresserbare lokasjoner a 8 bits. Cellene er vist som sm  orange firkanter. Figuren viser hvordan dekodeeren brukes til   plukke ut en bestemt lokasjon i minnet. 1: Overf ringen begynner med at CPU legger en adresse p  adressebussen. 2: Dekodeeren bruker adressen til   plukke ut en bestemt lokasjon i minnet. I dette tilfellet velges lokasjon nr 43. Dekodeeren virker slik at den setter Velg-signalet til alle celler i dette ordet til 1 (og velg-signalet til alle de andre cellene i prim rminnet til 0). 3: Innholdet av cellene i den utvalgte lokasjonen legges p  databussen. 4: CPU leser verdien fra databussen.

#### 1.5.5. Kvadratisk organisering

Produksjonsteknisk er det faktisk ganske enkelt   pakke sammen sv rt mange minneceller i en integrert krets. Det som er vanskelig er   lage store dekodeere, fordi dekodeeren er en relativt komplisert krets hvis den skal skille mellom sv rt mange lokasjoner

Isteden for   lage  n minnebrikke som inneholder hele prim rminnet, er det vanlig   spre minnecellene utover flere minnebrikker.

Ytterpunktet blir   bruke en brikke for hver bit i ordet. Da lar vi en brikke inneholde bit 0 til alle lokasjoner, la en annen brikke inneholder bit 1, en tredje brikke inneholder bit 2 osv. Vi trenger like mange brikker som vi har bits i ordet.

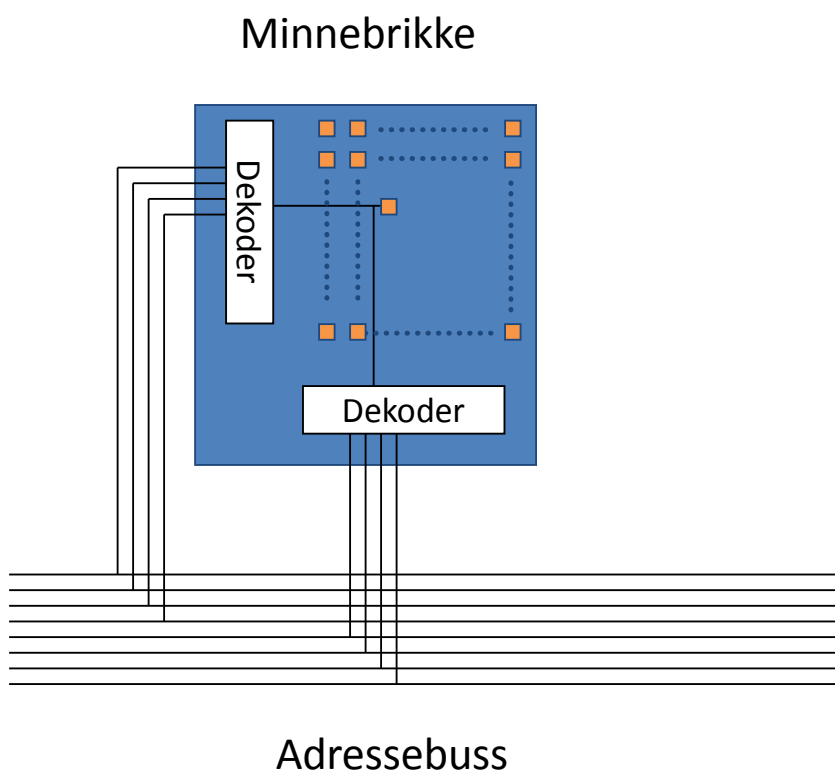


Da f r vi mange flere brikker   holde styr p , men n  skal vi se at vi til gjengjeld kan greie oss med enklere dekodere.

Det vi gj r er at vi organiserer minnecellene i hver av brikkene i form av en kvadratisk matrise. Vi setter cellene i like mange rekker og s yler.   velge en av dem blir dermed omtrent likedan som   velge et punkt i et xy-kordinatsystem. Vi velger ut en bestemt celle ved   angi hvilken rekke og hvilken s yle cellen befinner seg.

Fordelen med dette er at vi kan bruke halvparten av bitene i adressen til   velge rekke, og den andre halvparten til   velge s yle. Dekoderne blir enklere n , fordi vi bare bruker halve adressen til hver av dem.

Oppbyggingen til en minnebrikke som bruker slik kvadratisk organisering er vist i Figur 3.



**Figur 3. Minnebrikke med kvadratisk organisering.** Cellene er organisert som en matrise, og man velger ut en bestemt celle med   angi hvilken rekke og s yle cellen ligger i. Halvparten av adresselinjene brukes til   velge rekke. Den andre halvparten brukes til   velge s yle.

N  m  vi alts  bruke flere brikker slik Figur 4 viser. Vi m  bruke like mange brikker som vi har bits i ordene. Hvis minnet er organisert som bytes trenger vi alts  8 brikker<sup>1</sup>. Med 16-bits ordbredde trengs 16 brikker.

Dette er en sv rt vanlig m te   organisere minne p  - spesielt RAM-brikker. Lagringskapasiteten til en slik brikke angis i antall *bits* som kan lagres p  brikken. Typiske st rrelser er 16 Mb, 64, 256 Mb, ... (Mb er megabits). Etter som teknologien har blitt bedre har

---

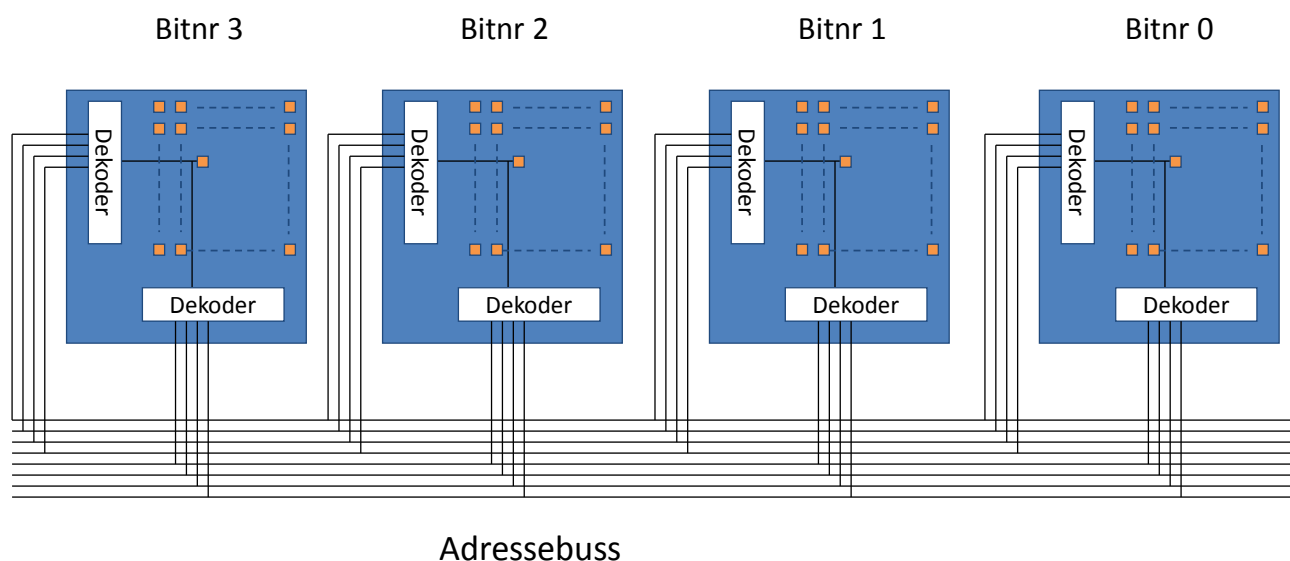
<sup>1</sup> Ofte brukes det en bit i tillegg til de bitene som inng r i ordet. Denne ekstra biten kalles paritetsbit og er en enkel form for feilsjekking. Hvis det brukes paritetsbit vil man trenge 9 brikker ved 8-bits ordbredde. I denne leksjonen ser vi bort fra paritetsbiten og antar at det brukes like mange brikker som det finnes bits i et ord.



brikkene f tt stadig st rre kapasitet. Typisk dobles antall rekker og s yler - noe som f rer til firedobling av kapasiteten - ca hvert 3.  r.

Legg merke til at lagerkapasiteten til hver enkelt lagerbrikke m les i antall *bits* som lagres p  dem, og ikke antall bytes.

Fordelene med kvadratisk organisering er at hver dekodeer blir enklere fordi de bare behandler halvparten av adresseinformasjonen. Dersom vi er i stand til   lage 10-bits dekodeere, s  kan vi alts  bruke 20 bits adresse. 20 bits adresse skiller mellom 1M forskjellige celler ( $1M = 2^{20} \approx 1$  million).



**Figur 4. Bruk av kvadratisk organisering. Her brukes fire brikker for   lage et prim rminne med firebitsord. Vi trenger like mange brikker som vi har bits i ordene.**

### 1.5.6. Andre organiseringer av minnebrikken

N  har vi sett at vi kan organisere brikkene med ordbreddeorganisering eller med kvadratisk organisering.

Men man kan ogs  organisere prim rminnet med brikker som bruker en mellomting mellom disse to.

For eksempel kan man bygge opp minnet med brikker som lagrer fire og fire bits.

Det har i liten grad festet seg noen standard, s  man finner brikker med de forskjellige organiseringer.

## 1.6. Moduler og minnekapasitet

Det trengs alts  mange lagerbrikker for   bygge opp prim rminnet til en datamaskin. For eksempel har vi sett at man trenger 8 brikker med kvadratisk organisering n r hver lokasjon best r av en byte. Disse brikkene kaller vi *en modul*. Dersom vi  nsker   utvide minnet m  vi bruke flere slike moduler. Husk at minnekapasitet for det totale prim rminne oppgis som antall bytes i minnet.

### 1.6.1. Organisering av minne p  PC; SIMM og DIMM

Heldigvis er det slik at vi sjelden trenger   bry oss om hva slags minnebrikker som brukes p  en PC. P  moderne PCer brukes enten noe som kalles SIMM-kort, eller de bruker s kalte DIMM-kort. SIMM st r for *Single Inline Memory Module*, og DIMM st r for *Dual Inline Memory Module*.

En SIMM og en DIMM er ganske enkelt et lite printkort. P  dette printkortet st r alle brikkene som inng r i en modul. Produsenten setter sammen brikker p  printkortet og selger det hele som en enhet. K peren trenger f lgelig ikke tenke p  hva slags brikker som brukes.

En SIMM er organisert som 32 bits minne. P  Pentium-maskiner leses eller skrives 64 bits samtidig. P  gamle maskiner ble derfor SIMM-kort alltid brukt parvis<sup>2</sup>.

Dette gjaldt helt til en lyst hode inns  at printkortet har b de en over- og en underside, og at vi kan sette innholdet fra en SIMM p  oversiden, og innholdet fra en annen SIMM p  undersiden. Dette ble kalt DIMM.

P  DIMM er alts  de to modulene som inng r i en bank samlet p  ett kort. Man ser enkelt forskjell p  SIMM og DIMM ved at SIMM har lagerbrikker montert bare p  en side av printkortet, mens DIMM har lagerbrikker p  begge sider.

SIMM og DIMM er alts  ikke betegnelsen p  noen RAM-typer, men forteller hvordan minnet er organisert.

### 1.6.2. Forskjellige DRAM-typer

I annonser for PCer florerer det av betegnelser p  forskjellige minnetyper. SDRAM og DDR SDRAM er de vanligste. Alle disse er DRAM, og i en senere leksjon skal vi se n rmere p  disse typene. Forel pig holder det   vite at selve lagercelle-teknologien er akkurat den samme p  alle disse DRAM-typerne, men teknikken for   hente ut data er noe forskjellige.

## 1.7. Hvor ofte  nsker vi   lese fra minnet?

Alle instruksjoner trenger minst en minneaksess. Nemlig for   hente instruksjonen fra minnet. Dessuten skal det ofte leses eller skrives operander i tillegg. Det vil si at prosessorens effektive hastighet i hvert fall ikke er st rre enn lese/skrive-hastigheten mot minnet.

Hvor mange instruksjoner kan en moderne prosessor utf re hvert sekund? En Pentium-prosessor utf rer rundt 100MIPS (millioner instruksjoner pr sekund). Nyere prosessorer greier enda flere instruksjoner i sekundet. Som vi husker fra leksjon 1 blir utf ringshastigheten fordoblet omtrent annet hvert  r.

Moderne prosessorer kan utf re flere hundre millioner beregninger hvert sekund. Med en beregningskapasitet p  100 millioner instruksjoner pr sekund vil hver instruksjon ta 10ns. Som vi har sett i leksjonen er det bare SRAM som har s  kort aksesstid at den matcher denne hastigheten. Hvis hele minnet skal bygges opp med SRAM vil minnet p  maskinen bli sv rt dyrt, og det er ikke  konomisk forsvarlig. Vi kan bruke DRAM isteden, men det har jo mye d rligere aksesstid. Derfor vil DRAM sinke utf ringshastigheten fordi det ikke greier   f re CPU med instruksjoner og data tilstrekkelig hurtig.

---

<sup>2</sup> Et slikt SIMM-par kalles ofte en *bank*, fordi man alltid m  sette inn et helt antall *par* med SIMM-kort ved minneutvidelser. (V r klar over at begrepet bank ogs  brukes i andre sammenhenger innenfor minneteknologi).

Hvordan skal vi l se dette problemet? Jo, vi bruker en teknikk som kalles cache. Teknikken ble opprinnelig utviklet for stormaskiner, og inntil for noen  r siden var det utenkelig   bruke den p  mikromaskiner - n  har det alt  blitt helt vanlig.

Cache medf rer at vi bruker b de SRAM og DRAM. Vi legger de instruksjoner og data som CPU har mest bruk for i SRAM. S  lar vi det som vi sjeldnere har bruk for ligge i DRAM. CPU vil vanligvis finne det den trenger i det hurtige SRAM-minnet, slik at hastigheten blir opprettholdt. S  godtar vi at CPU av og til m  vente p  det som ligger i DRAM.

Cache vil bli beskrevet i en annen leksjon.