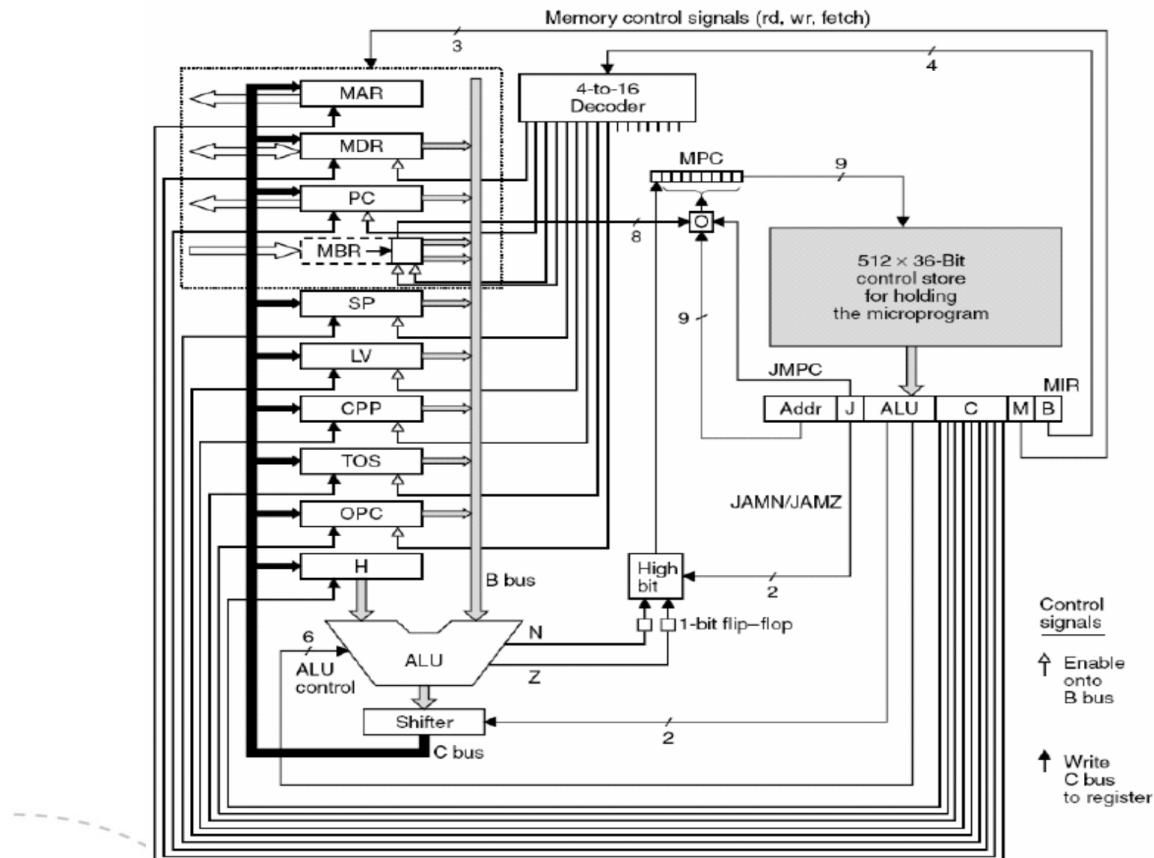


Week 9

Vi skal se på en mikroarkitektur som er en maskin som er bra egnet for å kjøre IJVM. Dette er en stack maskin.

IJVM Mikroarkitektur



Vi skal gå gjennom hele og skal kunne den i detalj.

Hvordan blir et program utført

Vi har en styreenhet som:

- Henter instruksjonen (fetch)
- Dekoder instruksjonen (decode)
- Utfører instruksjonen (execute) Styring av utførende enhet

Utførende enhet gjør:

- Execute: bruker utførende enhet til å utføre instruksjoner

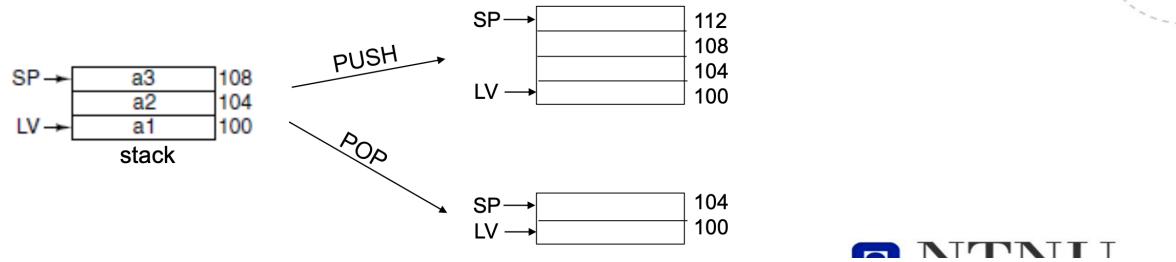
Instruksjonstyper

- Dataflytt operasjoner (eller kopier)
 - data til/fra minne, data til/fra register
- Datomanipulasjon operasjoner
 - ALU operasjoner (add, sub, multiply, divide, ... logiske operasjoner)
- Betinga hopp (eller "conditional branch") operasjoner
 - Endre programflyt ut fra resultat av operasjon
 - (if result = 0 then next instruction at adr x, else at next instruction at next program memory address)

Dette er en stack maskin. I minne vårt har vi et området som kan vokse eller krympe. Vi har en stack med variabler

| Hex | Mnemonic | Meaning |
|------|-------------------|---|
| 0x60 | IADD | Pop two words from stack; push their sum |
| 0x64 | ISUB | Pop two words from stack; push their difference |
| 0x7E | IAND | Pop two words from stack; push Boolean AND |
| 0x80 | IOR | Pop two words from stack; push Boolean OR |
| 0x84 | IINC varnum const | Add a constant to a local variable |

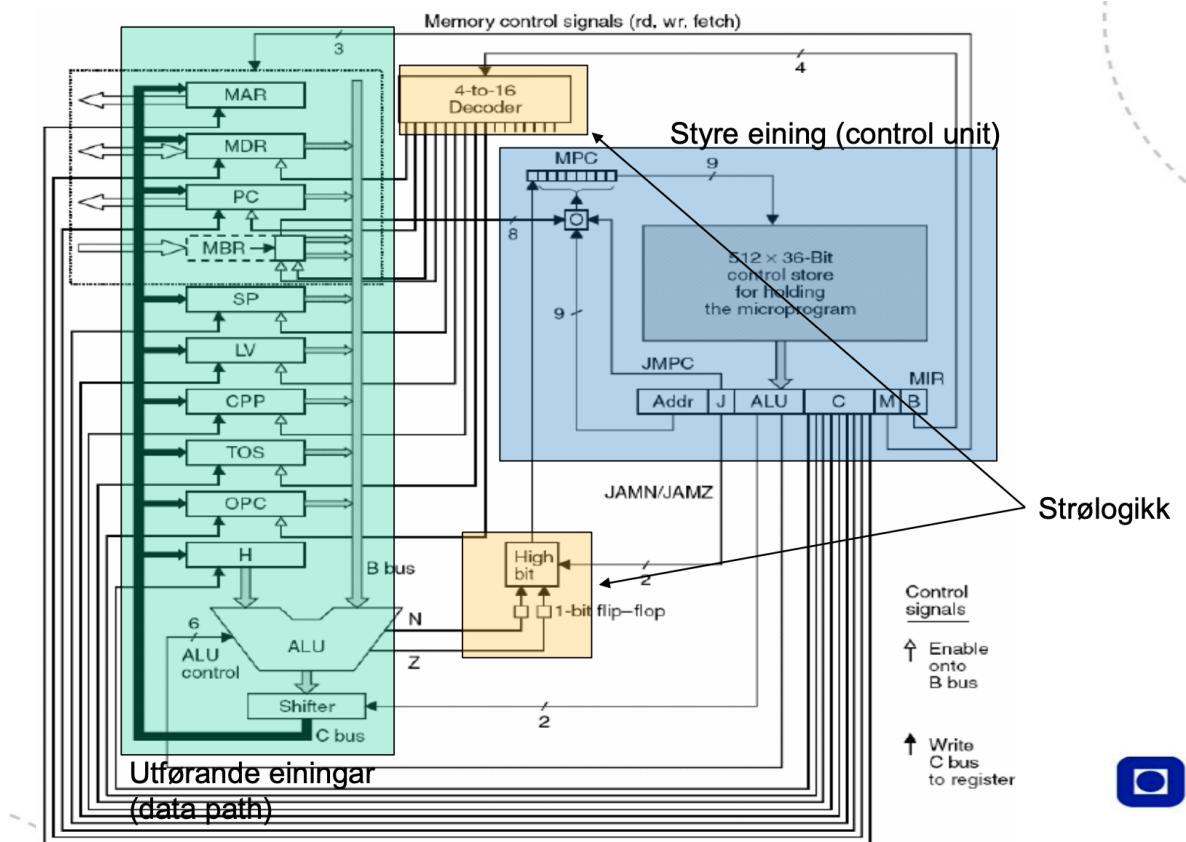
- Diverse aritmetiske instruksjoner



Her har vi et eksempel på stack. Vi har to pekere LV og SP. Local Variable peker på basen av stacken og Stack Pointer peker på toppen av stacken. En stack kan vokse eller krympe ved hjelp av push eller pop funksjonene.

Vi har noen ALU instruksjoner på toppen. Instruksjonen IADD som har 0x60. Vi kjører her to pop funksjoner som fjerner dem fra stacken. Vi tar summen av du poppede variablene og pusher dette til stacken. Dermed har vi redusert stacken med en variabel og har nå en ny variabel som er summen av de to forrige.

IJVM er en fler-sykel maskin. Vi bruker flere sykler på å utføre instruksjonene våre.

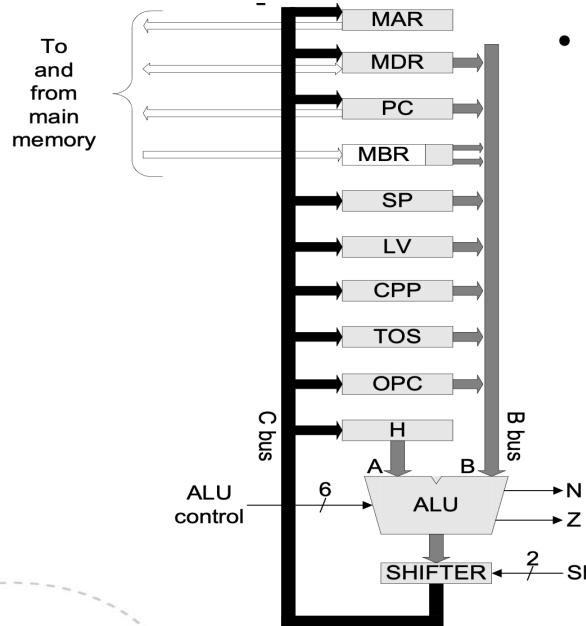


Vi kan dele IJVM i tre deler:

- Vi har en datapath (grønn)
- Kontrollenhet (blå)
- Strølogikk

N og Z som kommer ut av ALU-en er status-bit. De forteller hva som har skjedd i en operasjon. N forteller at resultatet var et negativt tall. Z forteller at resultatet var 0 (zero). Med dette kan vi lage en if/else enhet

IJVM datapath



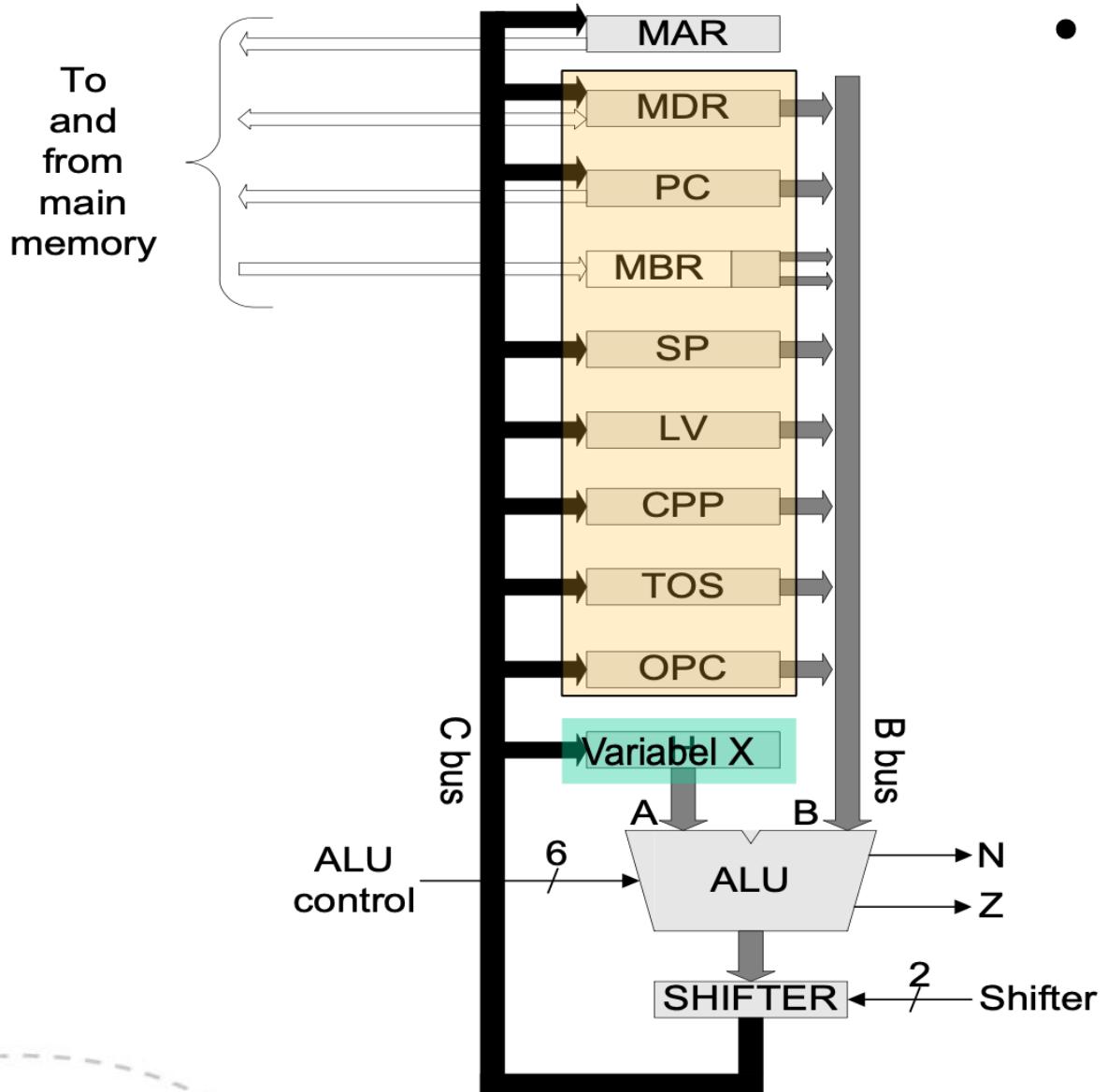
• Register:

- **MAR: Memory Address Register**
- **MDR: Memory Data Register**
- **PC: Program Counter**
- **MBR: Memory Buffer Register**
- **SP: Stack Pointer**
- **LV: Local variable**
- **CPP: Constant Pool Pointer**
- **TOS: Top of Stack**
- **OPC: OpCode register**
- **H: Holding register**
- **Shifter: styrt shift register og ALU utverdi**

Her inkluderer vi kun datapathen. Registrene i IJVM har fått navn. MAR, MDR og PC kjenner vi igjen fra før av og er med i mange arkitekturen. De er ansvarlige for å peke ut til det eksterne minnet vårt og få data/instruksjoner fra datapathen vår. MBR er også med på dette. Her vil selve instruksjonene komme til som fra MAR adressen.

Navnene på registrene er ikke så viktig. For det meste brukt for å skille dem.

Vi har et fancy register. Dette er H. Inngang A vil alltid få data fra H registeret.



B inngangen har tillgang til alle andre registrene, mens inngang A kun får informasjon fra register H.

ALU : 6 kontrolllinjer

| F ₀ | F ₁ | ENA | ENB | INVA | INC | Function |
|----------------|----------------|-----|-----|------|-----|-----------|
| 0 | 1 | 1 | 0 | 0 | 0 | A |
| 0 | 1 | 0 | 1 | 0 | 0 | B |
| 0 | 1 | 1 | 0 | 1 | 0 | \bar{A} |
| 1 | 0 | 1 | 1 | 0 | 0 | \bar{B} |
| 1 | 1 | 1 | 1 | 0 | 0 | A + B |
| 1 | 1 | 1 | 1 | 0 | 1 | A + B + 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | A + 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | B + 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | B - A |
| 1 | 1 | 0 | 1 | 1 | 0 | B - 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | -A |
| 0 | 0 | 1 | 1 | 0 | 0 | A AND B |
| 0 | 1 | 1 | 1 | 0 | 0 | A OR B |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | -1 |

ALU-en støtter blant annet disse funksjonene. Vi har 6 kontrollsinaler som definerer hvilke funksjoner som skal utføres.

Dersom vi har et bitmønster på: 011000 (første) så vil ALU-en kun gi ut A.

Vi har en shifter som ligger under ALU-en. Dette er en boks som tar to kontrollsinaler som manipulerer dataen:

Shifter 2 kontroll linjer

- SLL8
 - Shift Left Logical 1 byte
 - Shifter innhold 8 plassar (mot venstre)
 - Fyller LSB med 0
 - AA55AA55 (før)
 - 55AA5500 (etter)
- SRA1
 - Shift Right Arithmetic
 - Shift 1 bit (mot høgre)
 - La MSD være uendra
 - 1100 1000 ----- 1000 1000 (før)
 - 1010 0100 ----- 0100 0100 (etter)

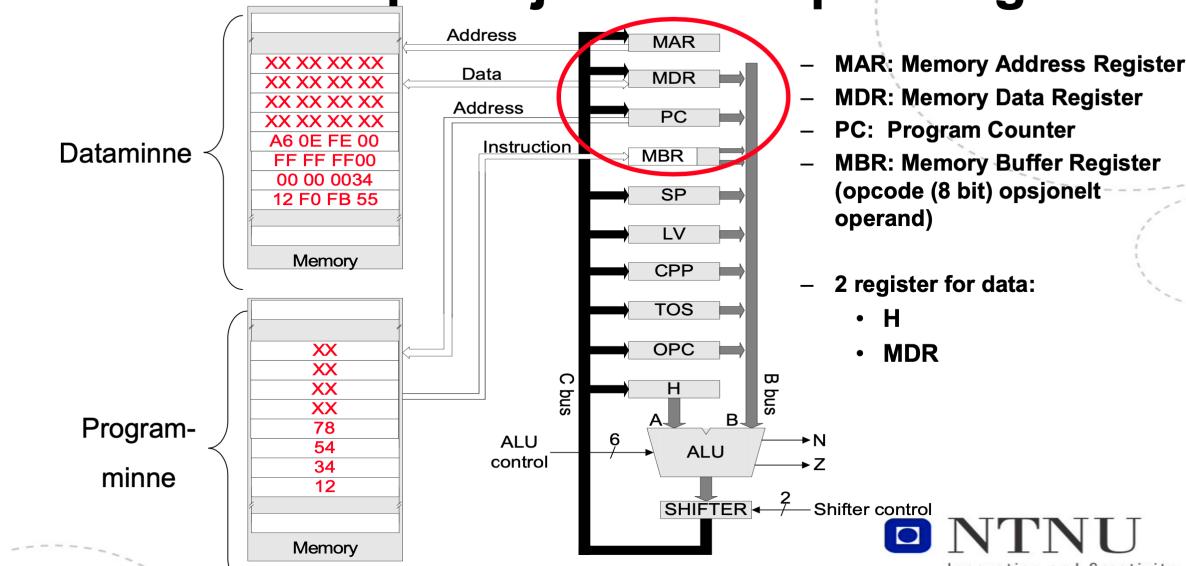
Vi har tre muligheter:

- Vi slipper resultatet gjennom uten å gjøre noe
- SLL8 shifter resultatet 8 ganger til venstre. Fyller på 0 på de nederste plassene
 - AA55AA55 (før)
 - 55AA5500 (etter)
- SRA1 shifter resultatet 1 bit til høyre.
 - La MSD være uendra
 - 1100 1000 ----- 1000 1000 (før)
 - 1010 0100 ----- 0100 0100 (etter)
 - Vi lar taller som ligger lengst til venstre være når vi shifter.

IJVM er en 32-bit datamaskin. Det første tallet i de 32-bitene sier om taller er positivt eller negativt:

- 1 = -
- 0 = +

IJVM: minne operasjonar data path register



For IJVM må vi gjøre noe educated guesses siden den aldri har vært fysisk implementert før. Vi sier at vi separerer datapathen vår med data- og programminne, altså det eksterne minne. Dataminne er på 32-bit og programminne er på 8-bit. Instruksjonene er altså på 8-bit.

- MAR
 - Register peker på adressen i dataminne vi ønsker å lese fra. Om vi ønsker å skrive til minne så peker MAR hvor den skal lagres
- MDR
 - Vi får lest data fra MAR til MDR. Når vi skriver til minne så får vi dataen inn i MDR og skriver den der MAR peker til
- PC
 - Peker til programminne til neste instruksjon vi ønsker å få tak i. Vi kan kun lese instruksjonene.
- MBR
 - Instruksjonen som PC peker på kommer til MBR. Registeret er mindre siden instruksjonene våre er kun 8-bit i forhold til andre registre som må være 32-bit.

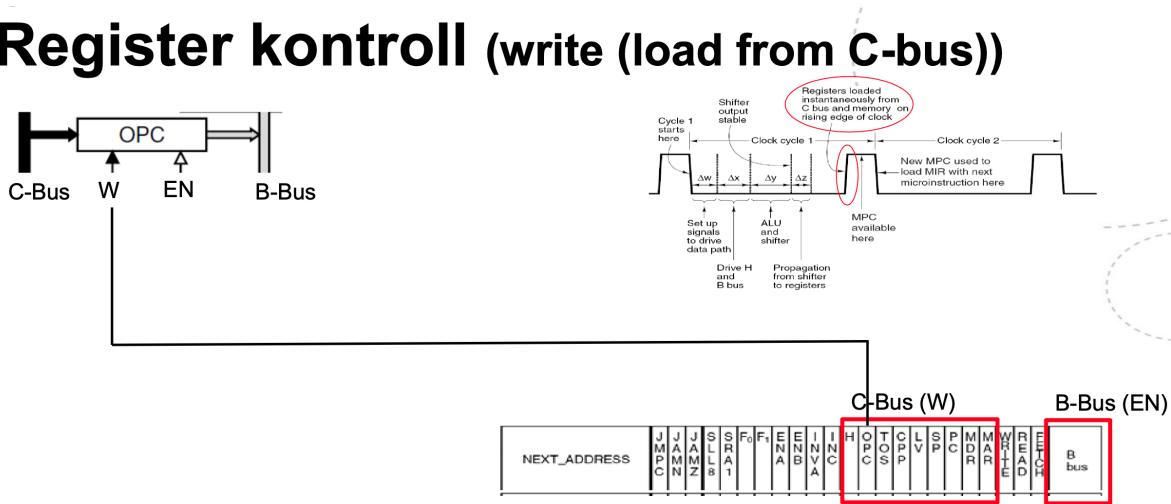
Her kan vi altså ha data som følger instruksjonene.

Kontrollenhet IJVM

Denne delen er mikroprogrammert. Vi har et mikroprogramm for alle instruksjonene våre. Enheten henter instruksjoner (fetch), dekoder (hvilken instruksjon er hentet?) og utfører (bruker datapath)

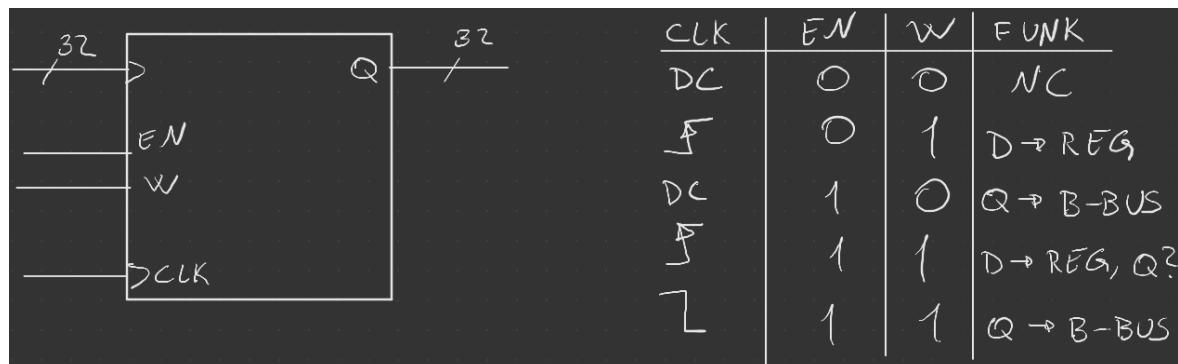
Kontrollenheten styrer datapathen via MIR-registeret. MIR er 36-bit registeret. Alle ledninger som går inn/ut er en bit vektor eller en enkel bit. Dette bestemmer hva datapathen skal gjøre i en klokkeperiode.

Register kontroll (write (load from C-bus))



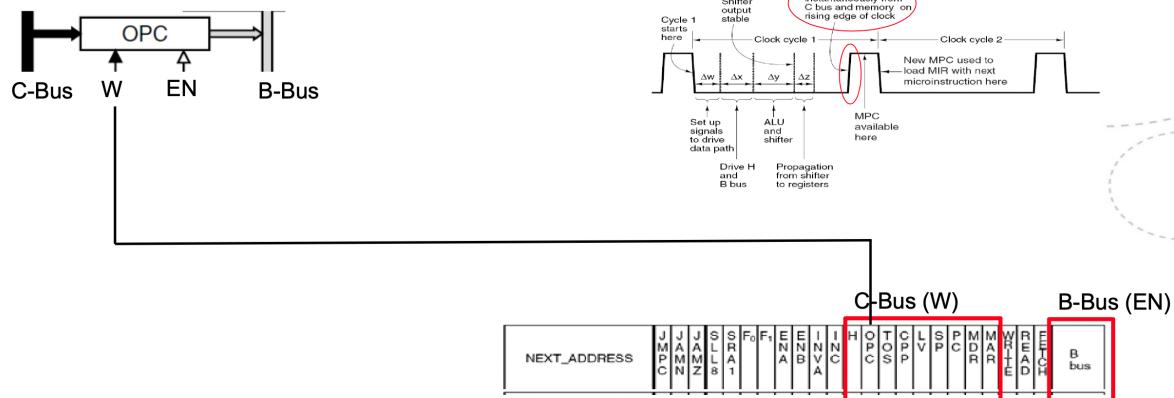
Vi kan se litt på de forskjellige bitene i MIR.

Vi ser på OPC-registeret som ligger i datapathen. Vi kan se litt nærmere på denne delen under:



- Dersom vi har 0 på read og write så er den frakoblet og har ingen funksjon
- Dersom vi har 1 på write vil registeret bli oppdatert på stigende rekkefølge.
- Dersom vi har 1 på enable så bruker vi ikke klokkesignalet. Vi bryr oss ikke hva det er
- Den siste hvor vi har EN = 1 og W = 1 så vil registeret bli oppdatert når klokken stiger. Alle andre tidspunkter i klokken så vil Q gå til B-bussen

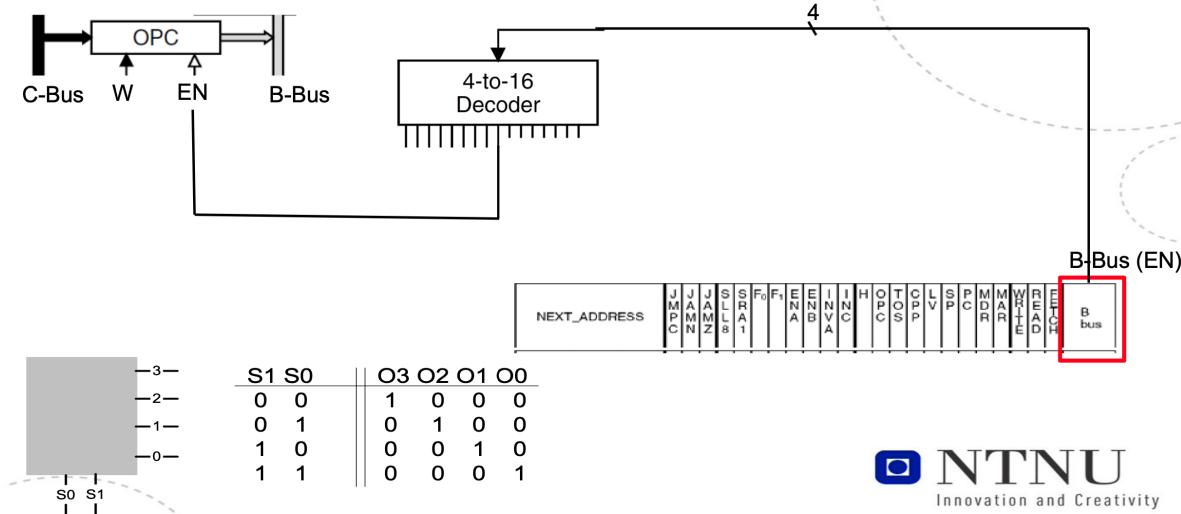
Register kontroll (write (load from C-bus))



Nå kommer vi tilbake til og ser på OPC og MIR samtidig. Når vi vil skrive til OPC så har den en bit i MIR som kontrollerer Write i OPC-en. Når denne er 1 så vil det som ligger i B-bussen bli skrevet inn i OPC-registeret.

Som sagt litt over så er OPC avhengig av klokken. Dersom OPC i MIR er 1 så vil C-bussen oppdatere OPC-en når klokkefrekvensen går opp (dette er satt en rød sirkel på dette i klokkefrekvensen øverst)

Register kontroll (read (data on B-bus))



Dersom vi ønsker å lese det som ligger i OPC så bruker vi B-bussen. B-bussen blir kontrollert av MIR av et felt på 4-bit. Vi kan kun ha et register i B-bussen. Vi sier her at dette er en read siden vi putter noe i B-bussen.

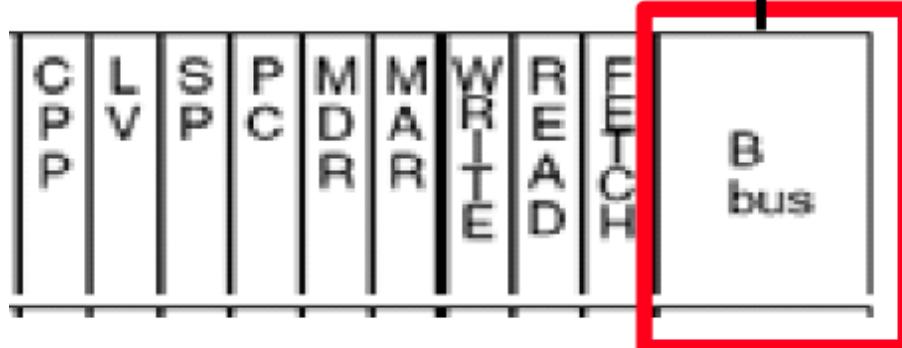
Vi bruker en decoder til å hjelpe oss med å lese. Her kan vi adressere hvilken av utgangene (pinnene nederst på decoderen) som skal være 1. Det er kun en av pinnene som kan være 1, som vist helt nederst i tabellen. Det vil si at det er kun et

register som kan få et EN signal. Dette er ønskelig siden det uansett er kun et register som kan skrive til B-bussen om gangen. Dermed adresserer B-buss delen i MIR hvilken av utgangene i decoderen som skal være 1.

B bus registers

| | |
|----------|-----------|
| 0 = MDR | 5 = LV |
| 1 = PC | 6 = CPP |
| 2 = MBR | 7 = TOS |
| 3 = MBRU | 8 = OPC |
| 4 = SP | 9-15 none |

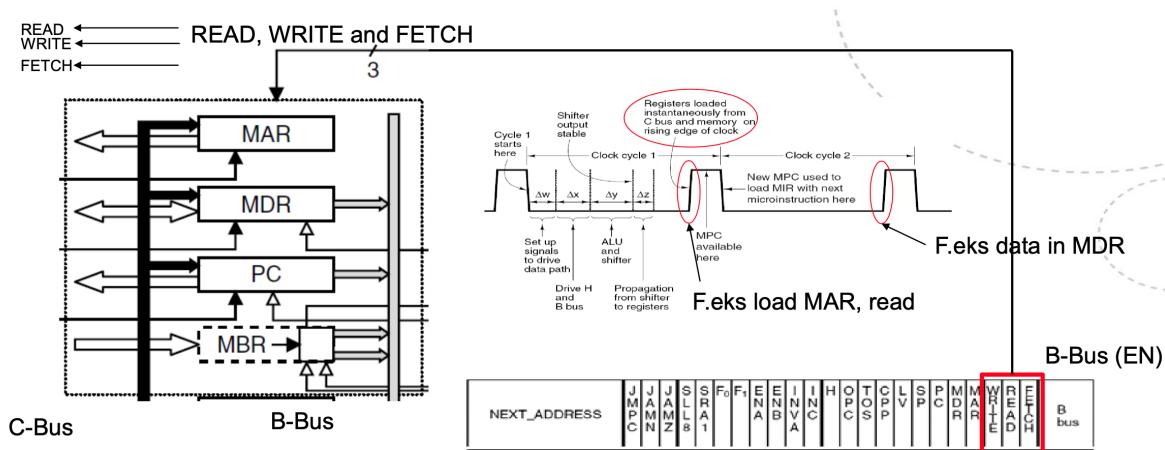
B-Bus (EN)



Her har vi en oversikt over hvordan de fire bitene fra MIR vil aktivere registrene.

Minneregister i MIR

Vi har tre register for å kontrollere minne. Dette er READ, WRITE og FETCH som ligger helt til venstre.



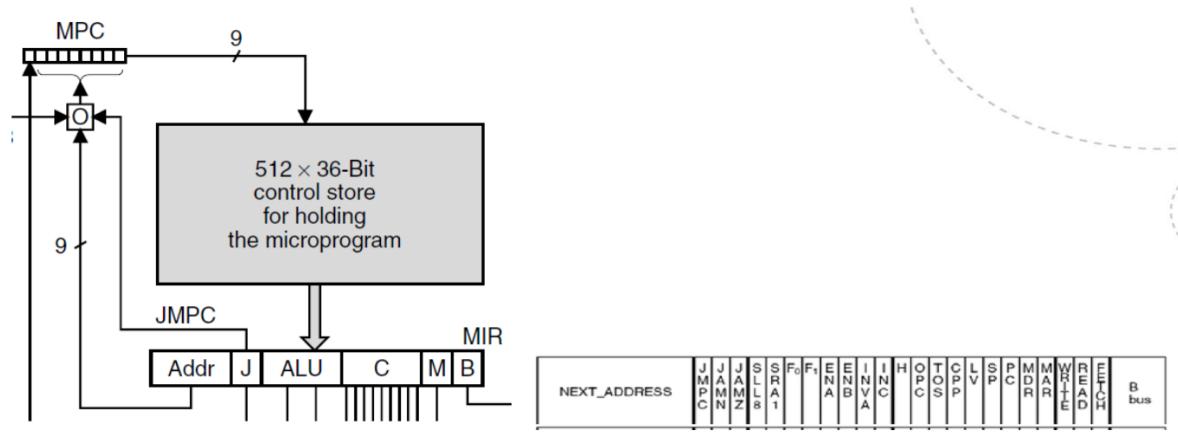
- Fetch
 - Et signal som er 1 når vi skal hente en ny instruksjon, som PC peker på, som skal inn i MDR.
- Read
 - Vi leser det MAR peker og lagrer det i MDR
- Write
 - Vi skriver det som er i MDR til plassen MAR peker på

Alt dette er avhengig av klokken vår.

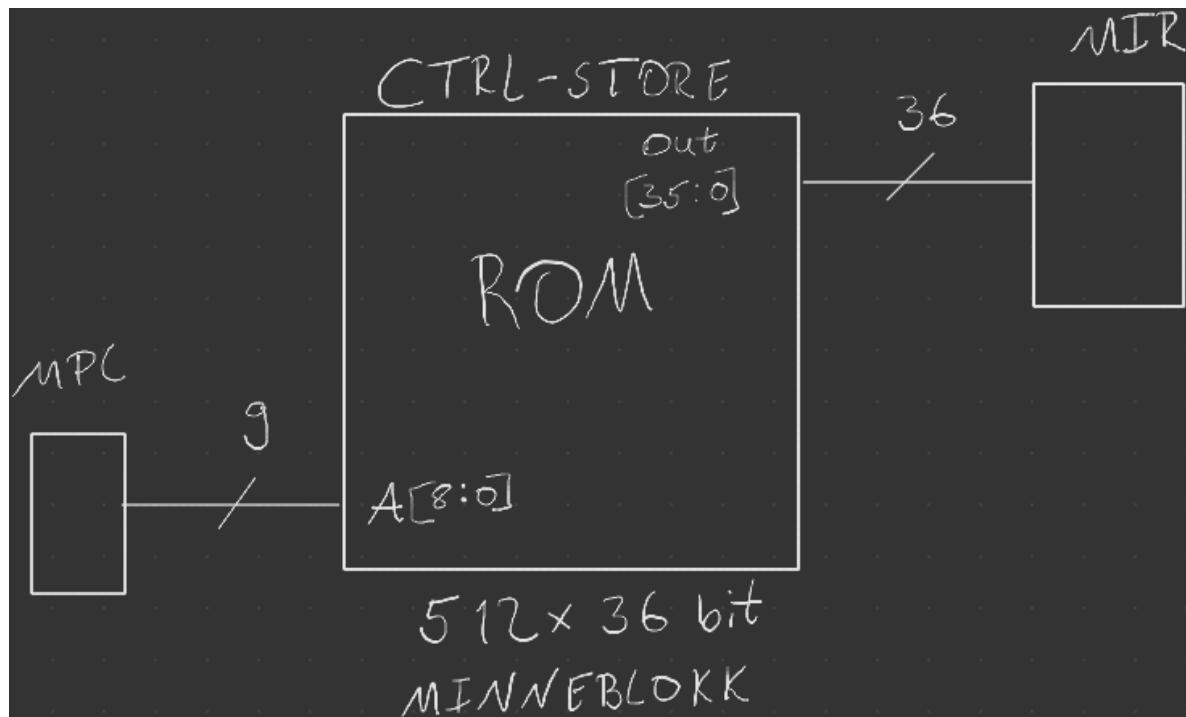
Dersom vi ønsker å lese noe fra minnet må vi legge ut adressen i C-bussen, det blir lastet inn i MAR på stigende klokke (første rød ring). Da vil MAR peke på adressen i det eksterne minnet. Vi må deretter gi det litt tid så det kan stabilisere seg. Vi kan lese der MAR peker på neste stigende klokkepuls (siste rød ring)

Vi må huske på å bruke C-buss og B-buss registrene i MAR (H, OPC, TOS, CPP, LV, SP, MDR, MAR) når vi ønsker å oppdatere.

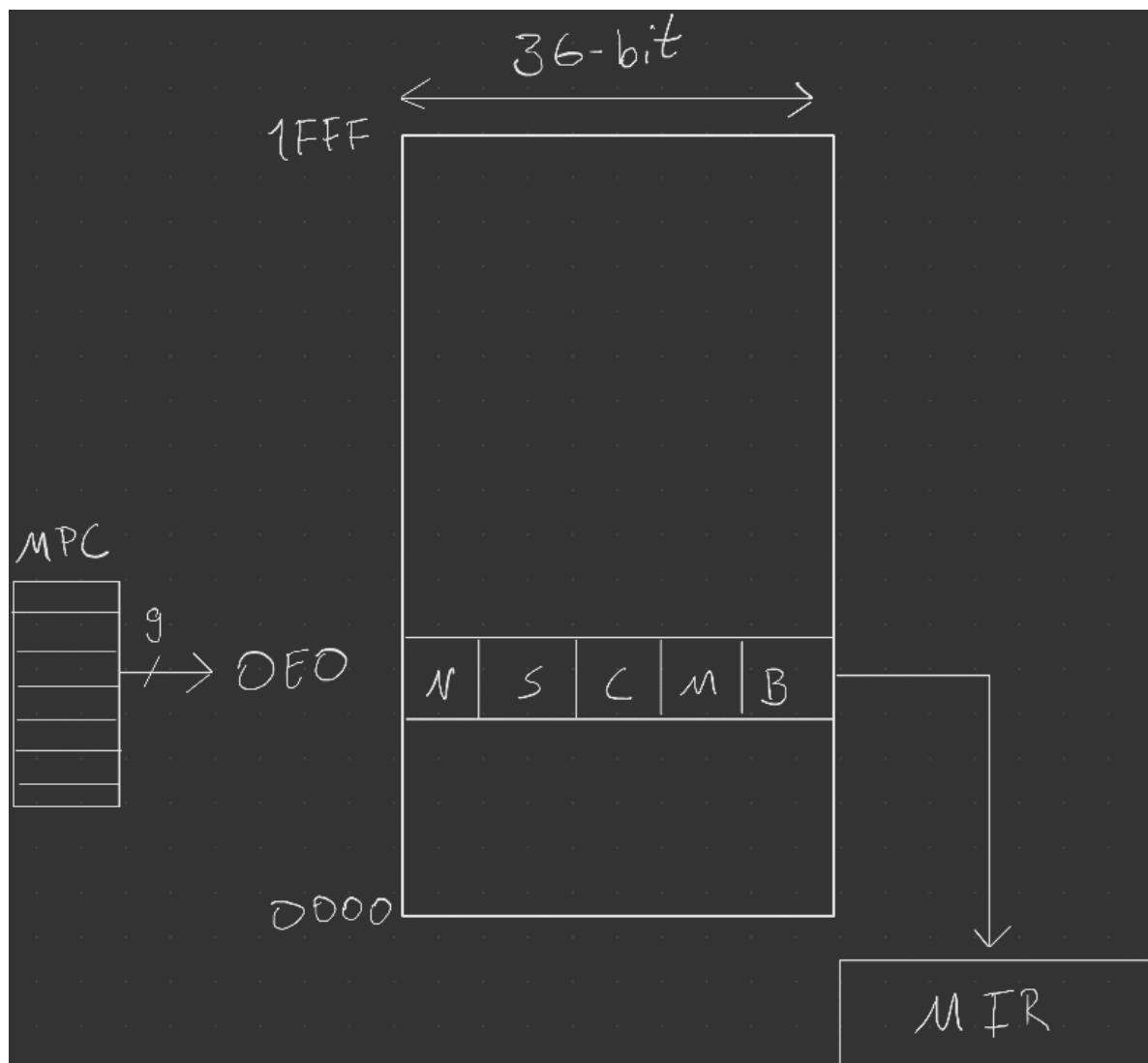
Control store



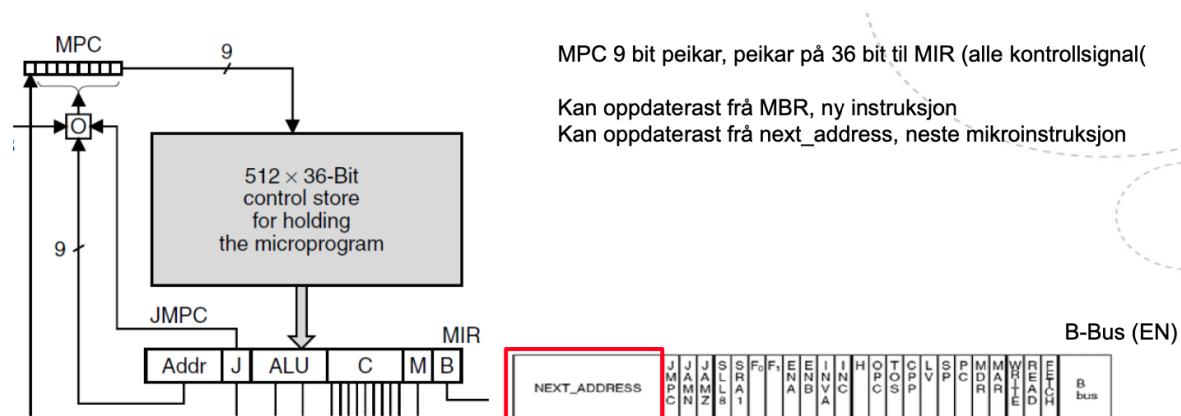
Egentlig en minneblokk. Vi ser litt mer detaljert på hvordan den ser ut:



Dette er en ROM-brikke. Virker som alle andre minnebrikker. Den har en adressebuss på 9-bit. Vi har $2^9 = 512$ adresser. Hver lokasjon er på 36 bit som har en utgang til MIR. Vi har ikke et inngangs- eller utgangsregister. MPC (Micro Program Counter) er adresseregisteret.



Vi tegner også opp et adressekart for CTRL-STORE. Vi har 512 lokasjoner på 36 bit. Disse 36 bitene kontrollerer MIR. Det vil si den adressen MPC peker på vil bli lagret i MIR som igjen vil styre datapathen.



Vi kan også se litt kort på next_address feltet. Når vi har utført en operasjon så som MIR har konstruert, så har MIR også lagret adressen til neste operasjon. Må

MIR-en en der ctrl-store ser vi at adressefeltet går opp til MPC-registeret.