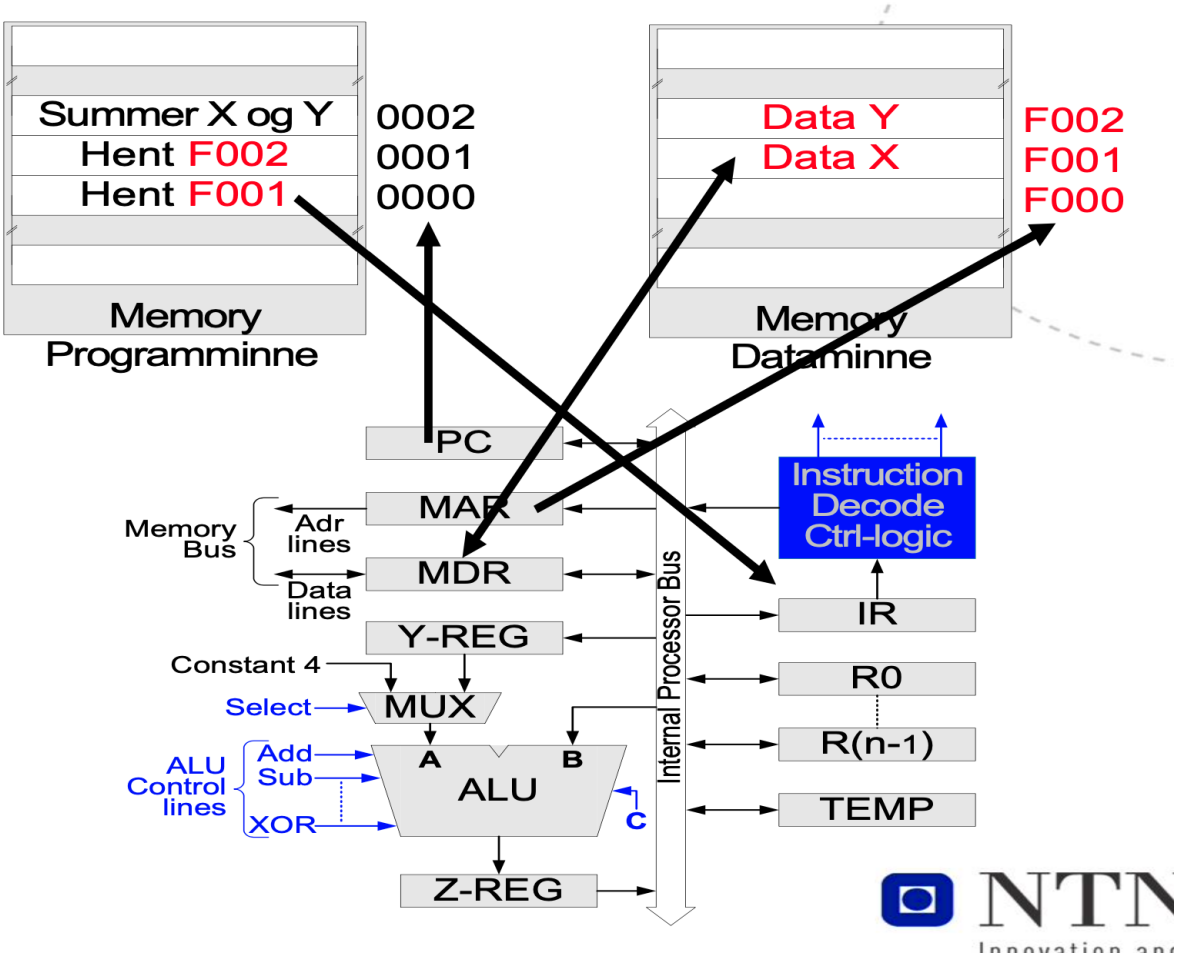
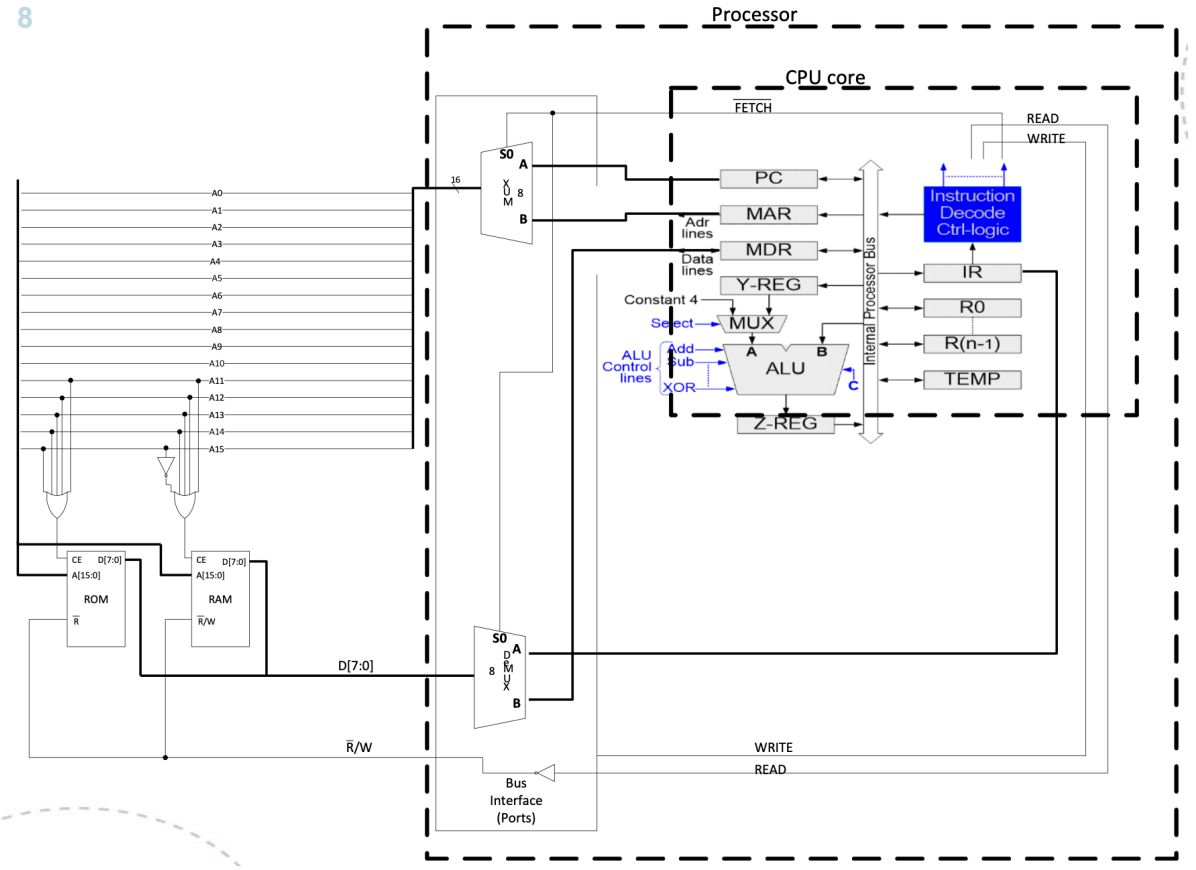
Week 8



Liten recap på hva vi har gått gjennom så langt. Vår program counter (PC) peker på adressen til den neste instruksjonen. Her peker den på 0000. Denne blir altså puttet på bussen og levert til instruksjonsregisteret (IR) som sender den videre til decoderen. Decoderen legger dermed ut kontrollsignal for å få utført riktig instruksjon.
Memory address register peker på adressen til dataen. Vi kan lese eller skrive til det MAR peker til ved hjelp av MDR registeret. Daten i MDR blir altså skrevet til adressen i MAR. Vi kan også få tak i data hvor MAR peker på hvor den ligger så vil dataen havne i MDR.

Lage en sammenstilling
Vi går et hakk videre. Vi slår sammen buss, adressedekoding, prosessorkjærne og grensesnitt mot bussen.



Her har vi altså en prosessor kjerne som ligger inne i en prosessor. Denne prosessoren har en utgang for adressering og vi ser hvordan denne adresseringen blir dekoden for ROM og RAM-brikken. Vi har også en data-path som ROM og RAM deler. Denne er 8 bit og går inn i kjernen. Videre er MUX-en koblet til et FETCH signal. Den kommer fra decoderen. Den er aktivt lavt, men er aktivt når vi skal hente en instruksjon.
Vi har en READ og WRITE fra decoderen. WRITE bruker vi ikke. READ inverteres når den skal til bussen.

For å få ut en adresse på bussen så har vi en mulitplexer. Denne kan velge mellom PC dersom vi trenger en instruksjon, eller MAR dersom vi må hente inn data.

For å kunne få tak i eller skrive til databussen så har vi en DEMUX. Denne kobler seg til IR eller MDR. Da kan vi lese eller skrive.

La oss se på noen eksempler: **hente en instruksjon**
Det første vi ser er at PC sender ut adressen til instruksjonen. Fetch signalet fra decoderen blir lav slik at PC signalet kommer ut på adressebussen. Vi bruker det samme fetch signalet i DEMUX-en til databussen. Det gjør at dataen kommer fra bussen og går videre til IR. Neste som skjer er at read-signalet går til 1. Dette er

fordi vi leser en instruksjon. Da blir R/W på utgangen av prosessoren lav. Etter at vi har fått instruksjonen i IR er vi nesten ferdige. Vi setter til slutt READ=0 og FETCH=1.

Lese data
Vi har adressen vår i MAR. Fetch=1 slik at vi får MAR i adressebussen. Fetch velger da også at DEMUX-en er koblet opp til MDR-en. Read går deretter til 1, som blir inverter, og vi får et aktivt read signal. Når dette er gjort kan vi oppdatere MDR. Vi setter read=0 igjen og vi er ferdig

Skrive data
Vi har adresse i MAR og vi har dataen som vi ønsker å skrive i MDR. Vi setter fetch høy slik at MAR er på bussen. Vi setter dermed samtidig DEMUX til B som er koblet til MDR. Write signalet går til 1 (det brukes ikke her) og read=0. Vi får dermed R/W signal lik 1 ut av prosessoren. RAM-en får dermed aktiv CE gjennom bussen, adressen på adressebussen peker på RAM-lokasjonen hvor vi ønsker å lagre dataen, vi gir et read-signal som betyr at dataen som ligger på databussen blir lagret i adresselokasjonen. Til slutt setter decoderen write til 0.

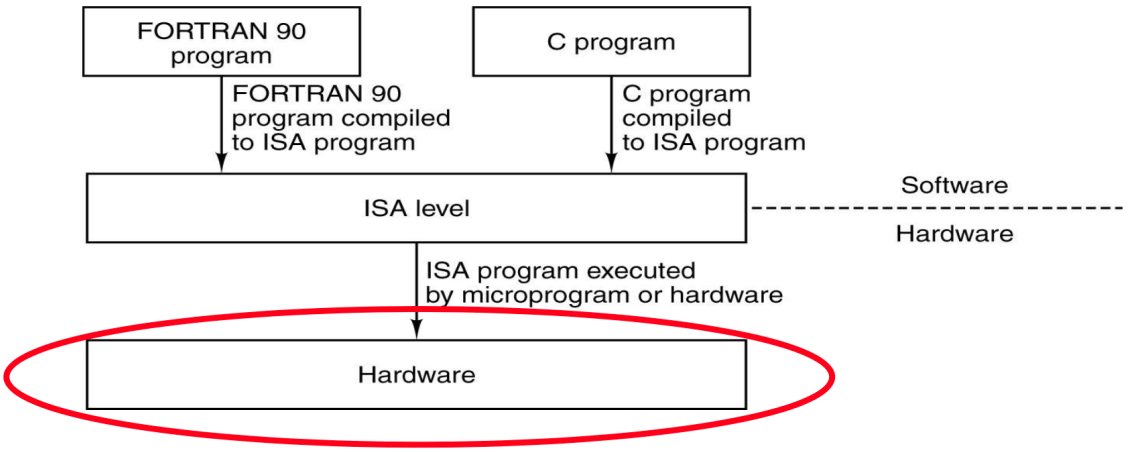
Mikroarkitektur nivå

Mikroarkitektur er nivå 1. Vi må se litt på nivåer over først.

På nivå 1 er det prosessorkjernen vi snakker om. Hvordan er den bygget opp?

Nivå 2 er instruksjonssetarkitekur (ISA)

- Hordan instruksjonene som er bygget opp og deres generelle virkemåte.
- Her kan vi lese ISA instruksjonen og vite hva som skjer på logisk nivå.
- Opprinnelig det eneste nivået
- Språk: maskinspråk



Når vi skriver et program på et høyere nivå så kompileres dette ned til ISA program. Dette vil si at programmet blir oversatt til noe hardware kan forstå.

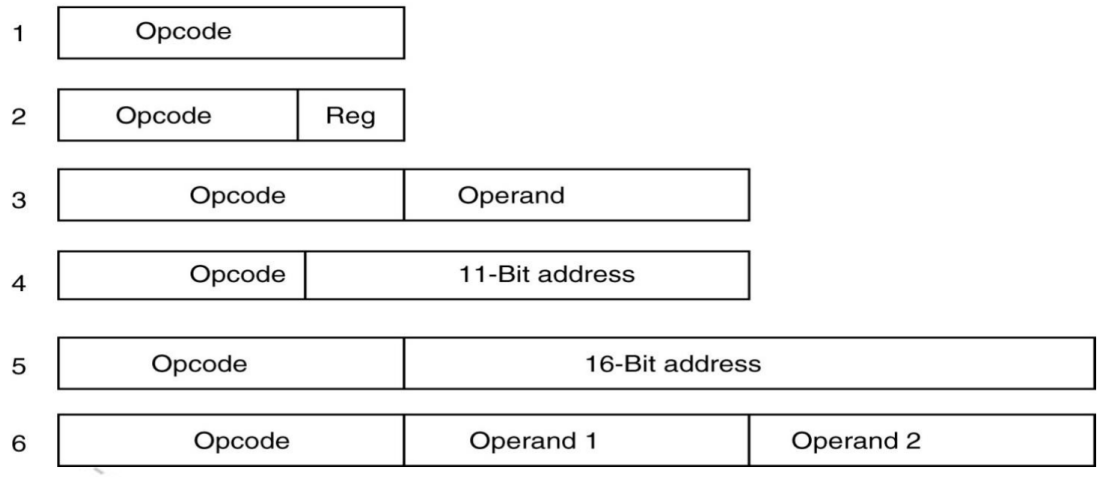
Intel 8051 mikrokontroller

Operation: RL
Function: Rotate Accumulator Left
Syntax: RL A

Instructions	OpCode	Bytes	Flags
RL A	0x23	1	C

Description: Shifts the bits of the Accumulator to the left. The left-most bit (bit 7) of the Accumulator is loaded into bit 0.

Format



Her ser vi ISA for 8051. Her er det enkle instruksjoner. Vi har 6 forskjellige klasser av instruksjoner.

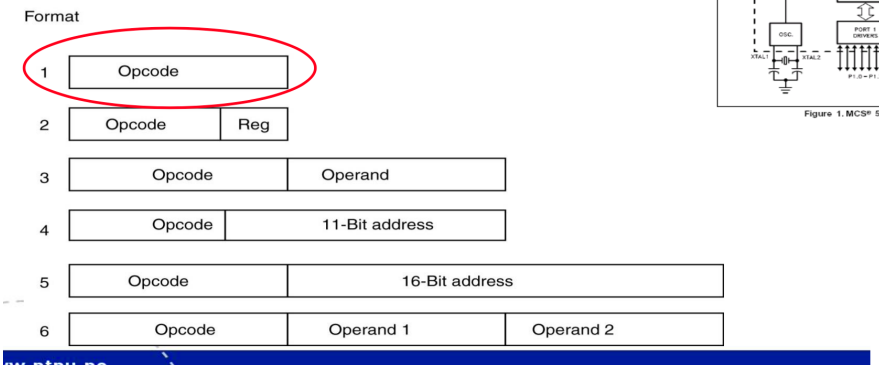
ISA -> mikroarkitektur

Operation: RL
Function: Rotate Accumulator Left
Syntax: RL A

Instructions	OpCode	Bytes	Flags
RL A	0x23	1	C

Description: Shifts the bits of the Accumulator to the left. The left-most bit (bit 7) of the Accumulator is loaded into bit 0.

Format



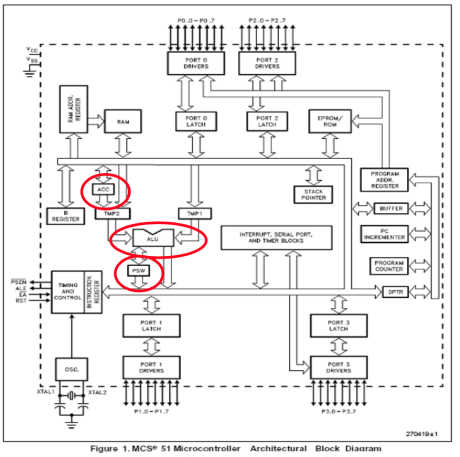
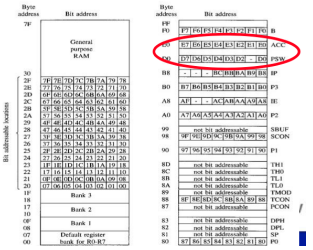


Figure 1. MCS® 51 Microcontroller Architectural Block Diagram



Vi kan lage en mikroarkitektur ved hjelp av et instruksjonsset, format på instruksjonene og en minnemodell. Da kan vi sette i gang og bygge en prosessor. Du kan bygge forskjellige prosessorer som støtter disse instruksjonssettene.

Mikroarkitektur

Mikroarkitektur realiserer ISA nivået. Vi må definere datapath og kontrollpath slik at datapath støtter instruksjonene effektivt og kontrollpath slik at instruksjonsformatet er effektivt. Vi

Vi har forskjellige typer maskiner:

- En-sykel maksin
  - En instruksjon hver klokkeperiode
  - Instruksjoner må være enkle
    - Copy R1 <- R2
    - Add R1, R2, R3
    - Load R1, MAR
    - Store R1, MAR
- Fler-sykel maskin
  - Flere klokkeperioder/sykler
  - "Komplekse" instruksjoner
    - Copy R1 <- minneAdr(mAdr)
    - Add mAdr, mAdr, mAdr
    - Load R1, mAdr + R2 (offset)
    - Store R1, mAdr + R2 (offset)

Ny arkitektur

Vi skal spesielt se på Java Virtual Machine (JVM)

- Integer JVM (IJVM)
- Hva må til for å realisere den?
  - Hva må være med av utførende enheter
  - Hvordan må de være koblet sammen?
- Hvordan kontrollerer vi den?
  - Styre enhet
  - Kontrollsignal til utførende enhet

IJVM Mikroarkitektur