Kapittel 1: introduksjon

6-nivå maskin:

Digitalt logisk nivå	**Mikroarkitektur nivå**	**Instruksjonsset nivå**	**Operativ system maskin nivå**	**Assembly språk nivå**	**Problemorientert språk nivå**
Porter og registre	Minne, ALU og data path	Maskinspråk	Operativ systemet	Assembly	Høynivå

Utviklingen av datamaskin arkitektur:

Mekaniske maskiner	Vacuum tubes	Transistorer	Integrerte kretser	VLSI

Moores lov:
Antall transistorer doubles hver 18. Måned

The computer spectrum:

Disposable computer	Mikrokontroller	Mobile and game computers	Personal computers	Servers	Clusters	Mainframes

System stack:
Application -> system software -> computer architecture -> electrical circuits -> low level electronics (transistors++)

Turing machine: abstrakt datamaskin som kan løse alle løselige problemer.
Vi har en evig lang minnetape, en enhet for å lese og skrive fra tapen, en tabell av actions (operasjoner) og et state register som snakker med action table.

Kapittel 2: organisering av datamaskinsystemer

CPUen

CPUen er hjernen i datamaskinen, og består blant annet av busser, ALUen, styreenheten og registrer, hvorav programtelleren og instruksjonsregisteret er viktige.
Instruksjonsnivå pipelining: **FETCH, DECODE, EXECUTE**
Av og til oppstår det "bobler" der vi må vente på at forrige intruksjons writeback er ferdig før vi kan fortsette.
Prosessor-nivå **pipelining**: SIMD prosessor (array av datamaskiner med single instruction-stream, multiple data-stream), multiprosessorer MISD og multidatamaskiner MIMD
Superskalar arkitektur: kjøre operasjoner i parallell – må klart være uavhengige av hverandre. Krever mye ekstra logikk.

Primærminnet (holder programmet som blir utført, kort aksesstid)

CPUens hastighet vokser fortere enn minnets hastighet(**memory wall**) ! En mulig løsning er...
Cache - mest brukte ord lagret i cache. **Lokalitetsprinsippet**: bare en liten del av minnet brukes over et lite tidsintervall. Lokalitet i rom (henter mer enn akkurat det som trengs siden du mest sannsynlig vil trenge det snart) og tid(siden du trenger dette nå trenger du det nok snart igjen)b
Unified cache vs. Split cache (**Harvard architecture**)
Von Neumann maskin: basic maskin som består av minne, aritmetisk logisk enhet, kontrollenhet, input og output enhet. Accumulatoren i ALUen holder et ord. I maskinen er instruksjoner og minne på samme buss. Harvard arkitekturen kan da aksessere instruksjoner og minne separat i kontrast.

Et annet problem er **power wall** – energieffektivitet er en kritisk designrestriksjon! En mulig løsning er...
Chip multiprocessors (CMP): flere CPU-kjerner ved siden av hverandre.
Variasjoner: dele cache vs. cache for hver kjerne, heterogene vs. homogene kjerner, heterogene vs. homogene ISA-nivå ((u)like instruksjoner).
Fordeler: energisparing, bedre ytelse for mange programmer på en gang og enklere kjerner
Ulemper: Høyere belastning på minne og I/O, cache coherency (cacher må synkes -> mye logikk), parallellprogrammering veldig vanskelig
Amdahls lov: uansett hvor mange kjerner vi har så er det alltid deler av programmet som må kjøres serielt. Dermed betyr ikke ∞ kjerner at t -> 0.
Stream multiprocessor: arbeid deles opp i warps med 32 tråder hver som kjører samme instruksjon

Sekundærminnet

Registre	Cache	Hovedminne	Magnetisk eller solid state disk	Tape eller optical disk

Aksesstid, kapasitet --->
<--- Pris/byte
Hovedminne: RAM er flyktig. Enten statisk (rask, men stor) eller dynamisk (treg, men liten)
Det er også ROM(read only memory), PROM(programmable ROM), EPROM(erasable PROM) og Flash memory (EEPROM – elektronisk EPROM)

I/O

CPUen snakker med enheter gjennom bussen
Programmering: "busy-wait" eller interrupts (CPUen får avbrudd fra vanlig flyt)

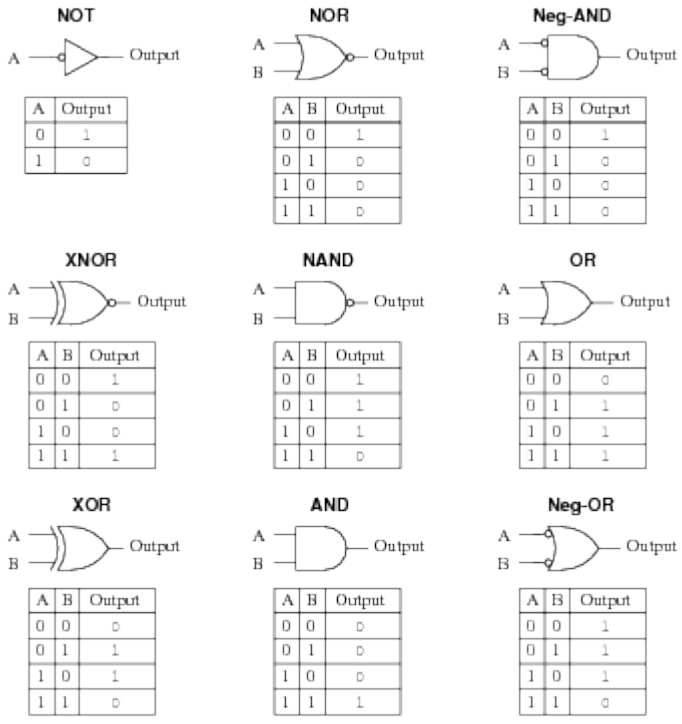
Kapittel 3: Digitalt logisk nivå

Protokoller for busser:
dele opp bussen i adressebuss, databuss og kontrollbuss (blant annet read/write signal)
Synkront vs. **asynkront**
Multiplekserbuss: samme linje for data og adresse v.h.a registre (tregere, men sparer en linje)

Vi bruker **minnekart** for å kartlegge hvilke adresser i minnet som brukes til hva
Vi bruker **CS** (chip select) for å velge aktiv datalinje ut ifra minnekartet

Three state buffer: for å koble fra busslinjer
Ny tilstand Z for å skille mellom 0 og 1.
I praksis: høy impedans --> 0 strøm

Arbitrering:
Hvem kontrollerer bussen?
Sentralisert arbitrering: Sentral arbitreringsenhet som bestemmer ved requests
Desentralisert arbitrering: Ingen sentral logikk, alle enhetene "forhandler" om bussen i stedet. Mer presist så har vi en bus request line hvor enhetene legger ut forespørsler, og når enheten blir master på bussen blir en annen linje, busy line, høy. Da kan ingen andre blir master før masteren er ferdig og busy linja blir lav igjen. Dersom vi får bus requests likt vil den nærmeste strømkilden få prioritet.



A	Output
0	1
1	0

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Multiplekser: selector 2^n innganger, en utgang og n kontrollsignaler
Dekoder: demultiplekser, velge hvilken utgang som skal være på v.h.a. kontrollsignaler.
Comparators, shifters, adders, ALU, klokker...

Sekvensiell logikk:
Utgangen er ikke bare en funksjon av inngangen med også maskinens **state**. Her kommer minne inn i bildet.
Vi har **flip flops** som er tilbakekoblet logikk som holder et bit. Vi har **SR**, som har et set bit og et reset bit.
Dersom de er av holder flip-floppen verdien sin, og så bruker vi bitene til å sette staten til 0/1. NA dersom vi

prøver å sette og resette samtidig. **JK** forbedrer dette, med at dersom J og K er høye så inverterer vi staten. Så har vi **D,** som ganske enkelt setter utgangen til inngangen ved klokkepuls. Og så har vi **T,** som har et togglebit for å endre utgangen. Altså dersom inngangsbitet er høyt så inverterer vi utgangen, ellers så holder den verdien sin.
Så har vi også **latcher**, som ikke bare skifter tilstand ved høy klokkeflanke, men hele tiden mens klokkepulsen er høy. Videre kan vi ha en **master-slave latch**, for å løse problemet med tidsforsinkelse. For våre formål så duger egentlig helt vanlige D flip-flops helt fint.

Finite state machines:
Vi har to typer tilstandsmaskiner vi ser på:
State-based **Moore FSM,** som har inngangslogikk, state registre og utgangslogikk. Inngangen går bare inn i inngangslogikken, og staten blir også sendt tilbake til inngangslogikken for å få tilbakekoblingen vi ønsker. Her blir altså utgangen bare avhengig av tilstanden til maskinen. Den er derfor state-based.
Input-based **Mealy FSM,** som er lik som Moore maskinen, men inngangssignalene til maskinen går inn i inputlogikken og outputlogikken! Dermed blir utgangen på maskinen avhengig av både tilstanden til maskien og inputs! Dette gir ulike kretser og ulike tilstandsdiagrammer.

Kapittel 4: Mikroarkitektur

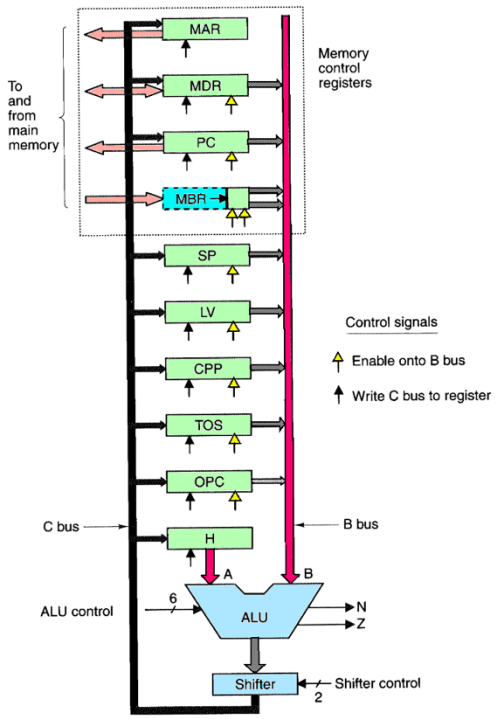
Nivået over digitalt logisk nivå. Formålet er å realisere ISA gjennom mikroarkitekturen.

Vi skiller mellom **RISC** (reduced instruction set computer, hardkodede enkle instruksjoner, og en instruksjon per klokkepuls) og **CISC** (complex instruction set computer, mer kompliserte instruksjoner med variabelt antall klokkepulser per instruksjon, mer fleksibelt instruksjonssett)

IJVM

Tar utgangspunkt i Integer Java Virtual Machine (**IJVM**)
Mikroarkitekturen inneholder er mikroprogram som skal hente, dekode og utføre IJVM instruksjonene. Det har et sett av variable, kalt "the state of the computer" som kan aksesseres av alle funksjoner. For eks. Har vi programtelleren, som indikerer minneadressen til neste funksjon som skal kalles. Instruksjonene i IJVM er korte, og inneholder først opkoden (operation code) som identifiserer instruksjonen. Deretter kan vi ha et variabelt antall ekstra felt, som spesifiserer operander i instruksjonen.

Data path

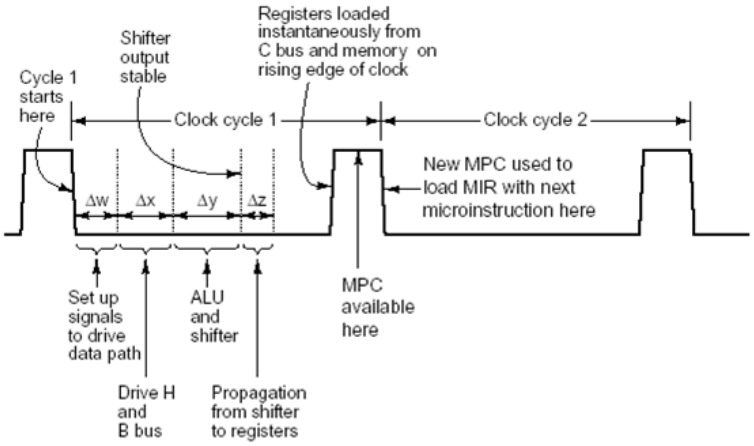


Data pathen er den delen av CPUen som inneholder ALUen og dens innganger og utganger. Den inneholder en rekke 32-bits registre. ALU kontrollen inneholder to bit som bestemmer operasjonen(F0 og F1), to bit for å enable inputene A og B (ENA og ENB), INVA for å invertere A inputet og INC for å tvinge en carry bit, altså plusse på en. Vi loader inn i H-registeret ved å sende en verdi fra et register på B-bussen, rett gjennom ALUen uten å gjøre noe og skrive til H-registeret. Vi har også to bit for å kontrollere shifteren. SLL8 shifter til venstre med en byte, SRA1 shifter til høyre med en bit uten å endre MSB. Vi kan både lese og skrive til samme register over en enkel klokkesyklus, siden lesingen og skrivingen skjer på ulike tider i syklusen.

PC *- (Program Counter) inneholder adressen til instruksjonen som utføres, eller neste instruksjon som skal utføres. (avhengig av måten maskinen er bygd).*
IR *- (Instruction Register) er der kontrollenheten lagrer instruksjonen som blir gjennomført nå. Den ligger her mens instruksjonen blir dekodet, startet og gjennomført.*
MAR *- (Memory Address Register) inneholder adresse til neste minnelokasjon der vi finner neste instruksjon.*
MDR *- (Memory Data Register) inneholder data som skal bli lagret i hovedminne (RAM), eller data som har blitt hentet fra minne. Virker som en buffer så data er klar for prosessor.*
MBR *- (Memory Buffer Register) er et bufferregister mellom minne og prosessor.*
LV *- (Local Variable) inneholder pekerverdi.*
SP *- (Stack Pointer) inneholder pekerverdi.*
CPP *- (Constant Pool Pointer) inneholder pekerverdi.*
TOS *- (Top Of Stack) skal alltid inneholde ordet på toppen av stakken.*
OPC *- (OpCode || Operation Code) register kan fritt brukes. Ex: MOV, ADD, LOAD.*
H *- (Holding Register) inneholder verdien som skal inn i A-inngangen til ALU.*

Timing

Puls –> alle kontrollbits settes ved lav flanke Δw -> Riktig register blir valgt og drevet på B-bussen Δx -> ALU og shifter gjør greia si Δy -> Data fra ALU til C-bussen Δz -> litt ekstra tid for å være sikker -> Skriving fra C-bussen til valgt register på høy flanke

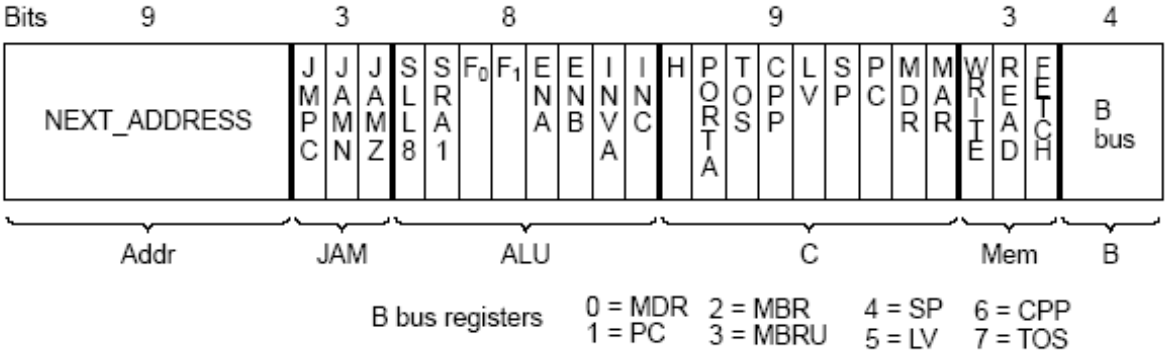


Minneoperasjoner

Enten gjennom 32-bit minneport eller en 8-bit minneport. 32-bit porten går gjennom registrene MAR (memory adress register) og MDR (memory data register) 8-bit porten styres av programtelleren PC, som naturligvis bare skal kunne lese fra minne, mer bestemt skrives det til MBR som deretter leses av PC. De gule pilene i illustrasjonen over indikerer et kontrollsignal som skrur på registerets utgang til B-bussen. De svarte pilene indikerer et kontrollsignal som skriver fra C-bussen. Vi bruker MAR/MDR til å lese og skrive ISA-nivå data ord, og PC/MBR til å lese ISA nivå porgrammet byte for byte.

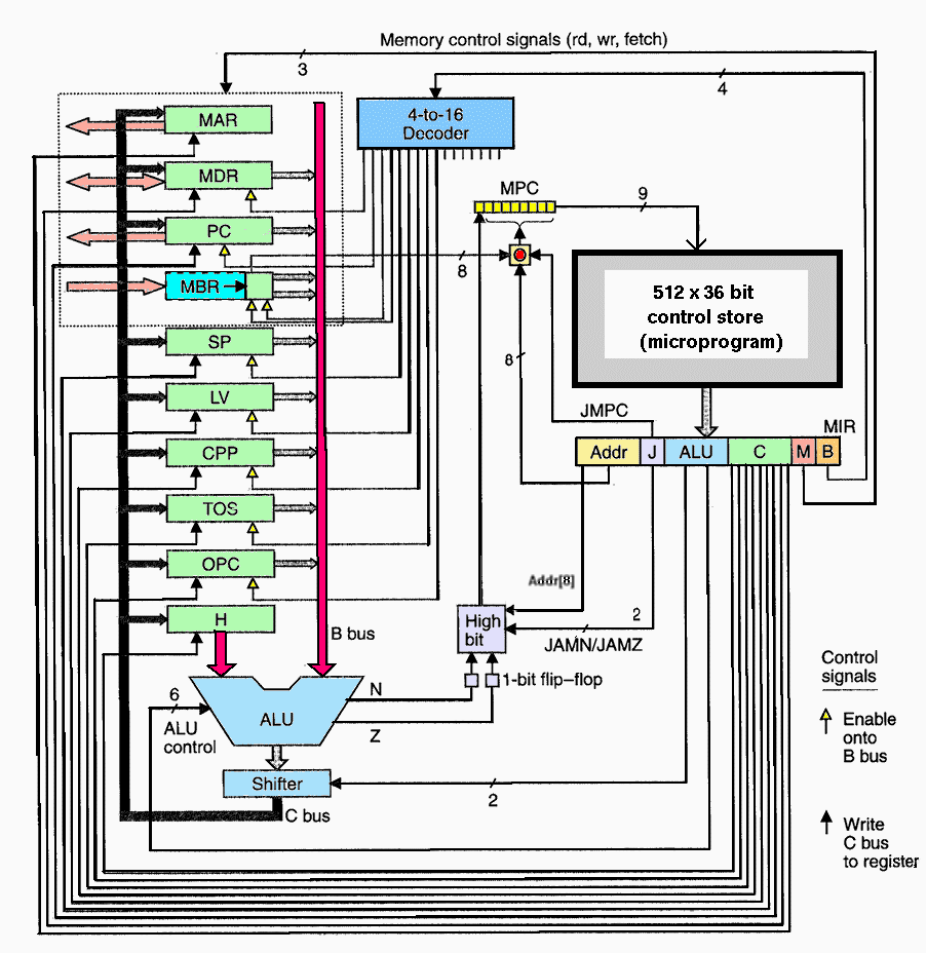
Vi styrer data pathen med hele 29 signaler: 9 for å styre skriving fra C-bussen til registrene, 9 for å enable registrere på B-bussen slik at ALUen får lese verdiene, 8 for kontrollere ALUen og shifteren som beskrevet tidligere, 2 for å styre R/W via MAR og MDR og 1 for å styre R via PC og MBR.

Data path kontroll: addr(adressen til neste mikroinstruksjon), JAM(hvordan neste mikroistruksjon skal velges), ALU (ALU og shifter funksjoner), C(hvilke registre som skriver til C-bussen), Mem (minnefunksjoner) og B (kilden til B-bussen)



Mikroinstruksjonkontroll

Hvordan bestemme hvilke kontrollsignaler som skal enables: v.h.a. en sequencer så klart! Den stepper igjennom alle operasjonene for en enkel ISA instruksjon. Den må produsere følgende informasjon: verdien til alle kontrollsignal, samt adressen til neste mikroinstruksjon.



Vi har allerede sett på datapathen. Nå studerer vi kontrolldelen av denne forenklede CPUen. Den viktigste delen er minnet kalt **control store**. Den inneholder hele mikroprogrammet som skal utføres, som inneholder mikroinstruksjoner. I IJVM har vi plass til 512 ord på formen til mikroinstruksjonen over(36 bit ord). Control store trenger ett eget adresse register **MPC**(MicroProgram Counter) og data register **MIR**(MicroInstruction Register). En viktig forskjell på PC og MPC er at PC teller faktisk gjennom adresser nedover i minnet, mens mikroinstruksjoner må være mer fleksible, slik at hver instruksjon spesifiserer neste isntruksjon. MIR holder den nåværende mikroinstruksjonen, som styrer alle kontrollsignalene til datapathen. Vi bruker en 4-til-16 dekoder for å velge hvilket registrer som skal drives på B-bussen.

Timing

Ved fallende flanke lastes MIR med en mikroinstruksjon fra control store som MPC peker på (Δw). Så vil alle kontrollsignalene fordeles i datapathen, som at et register leses til B-bussen og ALUen vet hva den skal gjøre (Δx). Så gjør ALUen greia si over tiden Δy og N, Z og shifter utgangen er stabile. Verdiene i N og Z lastes til to 1-bits flip-flops ved stigende flanke. Så sender vi dataen til registrene over Δz før vi skriver på stigende flanke som før.

Etter at MIR blir lastet i starten må vi også finne neste mikroinstruksjons adresse. NEXT_ADRESS kopieres til MPC og vi sjekker JAM samtidig. Hvis JAMene er null så trenger vi ikke gjøre stort, ellers trenger vi å gjøre mer jobb. N og Z bruker vi forresten for å lagre flaggene fra ALUen, siden vi i denne delen av syklusen ikke lenger kan lese fra ALUen, den spytter bare ut søppel. Når N eller Z er null gjør vi logikk for å branche.

Design av mikroarkitektur

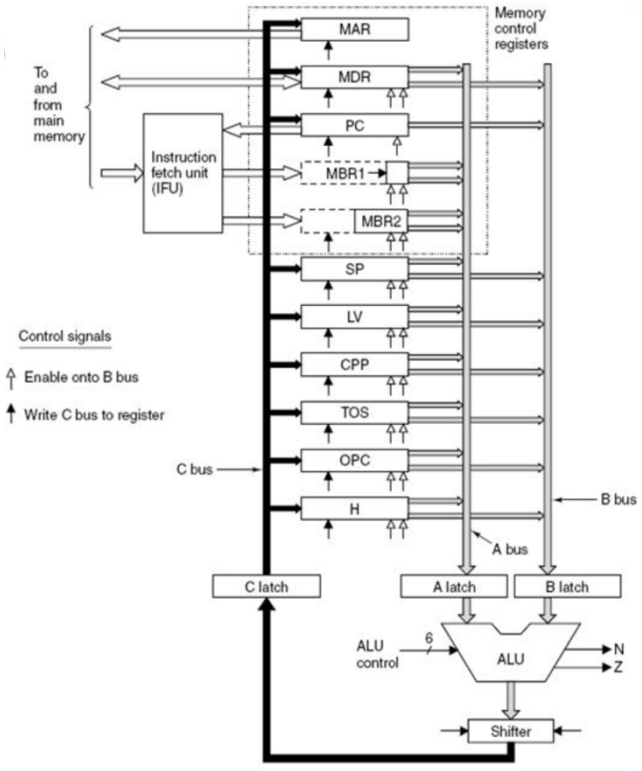
Antall klokkesykluser for et sett med instruksjoner er kjent som path length. Vi kan reduserer path length for eks. ved å implementere en inkrementerer i programtelleren, slik at ALUen ikke trenger å inkrementere PC hele tiden. Vi ser nå på en rekke måter vi kan redusere path length:

Merge interpreter løkken med mikroprogrammet:
Ved å merge main-løkkens instruksjoner inn i mikroprogrammet på sykler når ALUen ikke jobber kan vi gjøre alt stell som PC++, fetch etc. i programmet i stedet for etterpå.

Tre-buss arkitektur:
Ved å introdusere en tredje buss, A-bussen, kan vi redusere path length ytterligere. Der vi for enkelte instruksjoner før måtte først bruke en klokkesyklus på å loade en verdi fra et register til H-registeret, for så å gjøre den egentlige instruksjonen (som ADD), kan vi nå loade de to verdiene fra de to registrene på hver sin buss, og vi bruker bare en syklus i stedet for to.

Instruction Fetch Unit (IFU):
Lager ekstra logikk i PC for å inkrementere PC og hente instruksjoner. Videre kan vi få den til å lage operander.

Pipeline design:
Data path syklusen består av følgende deler som tar opp tiden: (1) Tiden vi bruker på å drive A-bussen og B-bussen til verdien til de valgte registrene, (2) Tiden det tar for ALUen og shifteren å gjøre greia si og (3) Tiden det for resultatet å bli sendt på C-bussen og lagret i riktig register. Vi legger til tre registre (en på hver buss) for å pipeline systemet i tre deler. Nå er hver klokkesyklus kortere (tiden til den av de tre stadiene som tar lengst tid), men viktigere vi kan bruke alle tre på en gang! Oh snap!
Hvert steg kaller vi nå et **microstep**. Dersom vi ikke kan starte et microstep fordi vi venter på det forrige kalles **true dependence** eller RAW dependence. Da må vi **stalle**.



Flytkontroll:
Dynamisk endring av programflyt.
Pipeline i 7 steg: IFU, DECODE, QUEUE, OPERANDS, EXECUTE, WRITE, MEMORY
Ved hopp: tøm pipelinen
For å oppnå dette legger vi til decode unit, queueing unit og MIR for de fire siste stegene.
LES MER PÅ DETTE!

Cache minne:
Dele opp cache i instruksjoner og data (Split cache). Så kan vi legge til ekstra level 2 cache, med mer plass, men lenger unna, osv. utifra hvor avansert systemet skal være. Vanligvis er cache nivåene inklusive, altså er alt i cache nivå 1 i cache 2, og alt i cache 2 er i cache 3 osv. Se forklaring i kapittel 2 for mer om cacher.

Branch prediction:
Pipelines funker best på lineære programmer, og her ødelegger branching fullstendig. Ubetingede hopp er vanskelig, siden fetch kommer før decode, så vi må vente. Betingede hopp er enda verre. Siden det er veldig ineffektivt å tømme pipelinen hver gang der er en branch prøver maskiner i stedet å prøve og forutse om det blir hopp eller ikke. En enkel løsning er å anta at alle hopp bakover skal utføres og at alle hopp forover ikke skal utføres. Videre må vi finne en måte å komme tilbake igjen dersom antagelsen vår er feil. **Dynamisk branch prediction** kan for eks. være å lage en historie tabell for å gjøre smartere forutsigelser. **Statisk branch prediction** kan være å bruke kompilatoren til hjelp.

Kapittel 5: Instruksjonssett-nivå

ISA-nivået er grensesnittet mellom programvare(kompilator) og maskinvare.
Det originale nivået. Mellom mikroarkitekturen og operativsystemet.
ISA-nivå kode er altså hva kompilatoren spytter ut. For å kunne kompilere må vi vite hva minnemodellen er, hvilke instruksjoner vi har, hvilke registre vi har osv. All denne informasjonen er ISA-nivået.
Høynivå programmer oversettes til ISA-nivået, der vi designer et hardware-system som kan utføre ISA-nivå instruksjonene.
Når vi designer et ISA-nivå vil vi helst at det skal være **backwards compatable**, vi må altså kunne kjøre gamle programmer med et nytt ISA. Dette er en begrensing når vi vil forbedre ytelsen på ISA-nivå.
ISA kan ofte kjøres i **kernel mode** (for operativ systemet, alle instruksjoner kan utføres) og **user mode** (for å kjøre programmer, utelukker sensitive instruksjoner).

Minnemodeller

*Hva maskinens **adresserbare enhet** er (som regel en byte), hva et **ord** er, hvordan en **adresse** ser ut, hvordan **adresserom(ene)** ser ut osv.*
Vi skiller ofte mellom enkle adresserom der CPUen snakker med et stort minne, og flere adresserom, der minnet gjerne er delt opp i programminne og dataminne.

Registre

Bare enkelte registre i mikroarkitekturen er tilgjengelige i ISA-nivået, som programtelleren. ISA-nivå registrene kan deles inn i **special-purpose registre** med spesifikke funksjoner (PC) og **general-purpose registre** som brukes til å holde viktige lokale variable og midlertidige resultater i operasjoner. Her kan alle registrene være like, eller vi kan designe spesielle registre.
Vi finner også en rekke kernel mode special-purpise registre for cacher, minne, I/O osv. som bare skal brukes av OSet.
Flag registret holder en rekke kontrollbit som CPUen trenger, hvorav condition codes er viktig, som holder statusen til den siste utførte operasjonen i ALUen (vi har allerede sett N og Z).

Datatyper

Numeriske datatyper: heltall (signed and unsigned), flyttall...
Ikke-numeriske datatyper: strenger, boolske verdier, bitmap, pekere...
I ISA definerer vi hvilke numeriske og ikke-numeriske datatyper vi støtter.

Instruksjonsformater

Instruksjoner består av en **opkode** og null, en eller flere **adresser** til operander. De fire vanlige instruksjonsformatene er **null-adresse, en-adresse, to-adresse og tre-adresse instruksjoner**.
Instruksjoner kan ha **fast instruksjonslengde**, som er enklere å dekode og pipeline, men sløser jo plass, eller **variabel instruksjonslengde**. Moderne ISAer har som regel fast instruksjonslengde, men IJVM har variabel lengde.
Vi ønsker å minimere instruksjonslengden, men jo kortere den er, jo vanskeligere er den å dekode.
Ekspandering av opkoder er som følger: når vi dekoder ser vi på de fire største bitene (for en 4-bit opkode) og ser om den ligger i området til en 4-bit opkode (0b000 – 0b1110). Hvis ja er den dekodet som en tre-adresse instruksjon, ellers sjekker vi de fire neste bitene for å finne ut om den er en to-adresse instruksjon osv... Vi varierer altså lengden på opkoden for ulikt antall adresser i instruksjonen. Dette er lett å dekode og fyller den ellers tomme plassen i en for eks. en-adresse instruksjon med fast lengde med muligheter for flere opkoder. Dessuten kan vi enkelt endre områdene for å endre hvor mange opkoder vi har for ulike instruksjonsformater. Genius!

Adressering

Addresseringsmodi: *Hvordan bestemme operandene i en instruksjon.*
Vi har en rekke ulike addresseringsmodi:

Immediate addressing: Operanden selv er direkte i instruksjonen, veldig begrenset bruk. For eks. `MOV R1, 4`. Operanden er en **immediate operand**, ettersom den hentes samtidig som instruksjonen selv.

Direct addressing: Vi spesifiserer adressen til operanden i minne direkte i instruksjonen. Begrenset til globale variable. For eks. `LOAD R1, 0x5555.`

Register addressing: Egentlig det samme som direct addressing, men vi spesifiserer et register, ikke en adresse. Siden registre er så mye brukt (megaraske), er dette også den mest brukt addresseringsmodien.

Register indirekt addressing: Operanden er i minnet, men refereres ikke til direkte som direct addressing, men gjennom et register (altså bare en peker). Dette blir mer effektivt, siden en registeradresse er mye kortere enn en minneadresse.

Indexed addressing: register addressing med et offset. Tenk assembly – smart å ha i mange tilfeller.

Based-indexed addressing: Adresse fra to registre + offset.

Instruksjonstyper

Data transport: kopiere data fra minne/register til minne/register. Vi må spesifisere hvor mye data som skal "flyttes" i instruksjonen vår.

Dyadic operasjon: kombinere to operander på en eller annen måte, som ADD, SUB, MULT, DIV, AND, OR osv.

Monadic operasjon: En operand blir ett resultat, for eks. høyre og venstre shift og rotering (shifting mister bitene, rotering bare "rullerer" de).

Sammenligning og betinget branch: Teste data og endre programflyt ut ifra resultatet.

Prosedyrekall: En prosedyre er en gruppe med instruksjoner som kan kalles fra flere deler i programmet. Gjerne da en funksjon. Her må vi passe på å huske return adressen, gjerne ved å legge den i stakken.

Løkke kontroll: utføre en rekke instruksjoner n ganger.

I/O: her varierer ting mye fra maskin til maskin. Men I/O kan deles opp i **busy-wait** (evig løkke sjekk), **interrupts** (generere en interrupt-vektor som peker til interrupt kode som må behandles når noe skjer) og **DMA**. Busy-wait er åpenbart crap, men problemet med interrupts er at det tar mye ressurser å behandle mange av de. Ved å legge til en DMA (direct memory access) kontroller til bussen flytter vi nesten all jobben fra CPUen til kontrolleren. CPUen forteller bare DMAen hva som skal skrives/leses og DMAen jobber mens CPUen kan gjøre andre ting. Men siden DMAen bruker bussen kommer den nok til å stjele noe sykler fra CPUen, men det er fremdeles mye mer effektivt.

Kontrollflyt

Sekvensen til instruksjonene som blir utført dynamisk under programkjøring.
Prosedyrekall, hopp, interrupts, traps etc. endrer flyten, som vanligvis vil være sekvensiell (PC++ ikkesant). Traps er som catcher, dersom en kritisk betingelse plutselig oppstår hopper programflyten til et fastsatt sted med en bestemt prosedyre for å behandle feilen, kalt en trap handler.

Kapittel 6: Operativsystem-nivå

Vi beveger oss opp et nivå til, vi. Et **operativsystem** er et program som gir programmereren mange nye instruksjoner, og er vanligvis implementert som programvare (selv om den fint kan være maskinvare). Operativsystemet implementerer **OSM-nivået** (operativsystemmaskin-nivået). OSM-nivå instruksjonene er alle instruksjonene som er tilgjengelig for programmereren, og inneholder de fleste ISA-nivå instruksjonene, samt mange nye som vi kaller **systemkall**. Et typisk systemkall vil være å lese data fra en fil.

Virtuelt minne

Før delte en programmer opp i biter kalt **overlays** som en lagret i sekundærminnet. Når en kjørte et helt program hentet en et overlay av gangen, kjørte det, hentet neste osv. Med **virtuelt minne** får vi gjort dette automatisk.
Ideen er enkel: vi separerer adresserommet og minnet. Selv om minnet bare er på for eks. 4K bit så kan vi ha et større adresserom. Vi kaller programdelene som leses for **pages**, og prosessen **paging**. Vi har nå da et **fysisk adresserom** på 4K bit og et **virtuelt adresserom** med alle mulige adresser som programmet kan referere til. Vi trenger et **minnekart** som spesifiserer den fysiske adressen til hver virtuell adresse. Programmereren slipper nå i praksis å ta hensyn til minne. Vi bare later som om alt minne er i primærminnet, så kjører vår snille pagingenhet i bakgrunnen og håndterer minnet. I sekundærminnet ligger nå hele programmet vårt, og en kopi av den deles vi trenger blir hentet til primærminnet. Når vi deler opp det virtuelle adresserommet i pages må vi også dele opp det fysiske adresserommet i deler like stor som en page. Disse delene i minnet der pagene går kaller vi **page frames**. Enheten som mapper virtuell minneadresse til fysisk minneadresse kaller vi en **MMU (memory management unit).** Den kan være i CPUen, eller på en separat chip.

Demand paging: en page er først sendt til minne når er forespørsel oppstår.
Alternativet er **lokalitetsprinsippet** – vi henter de pagene som blir brukt mest ved et bestemt tidspunkt. Vi kaller dette subsettet av alle pagene **working settet**. Vi trenger en algoritme for å dynamisk velge hvilke pages vi vil bytte ut ut ifra lokalitetsprinsippet. Dersom vi tror at en page ikke vil bli brukt før om en stund, er det kanskje smart å sende ut den over en kritisk en. **LRU-algoritmen (least recently used)** vil, som navnet tilsier, bytte ut den nye pagen med den minst brukte pagen i minnet. Det er enkelt å se senarioer hvor denne failer horribelt. En annen mulighet er **FIFO (first-in first-out)**, som sender ut den pagen som har vært i minnet lengst, uavhengig av hvor ofte den brukes. Dette oppnås enkelt med en teller for hver page i minnet. En annen måte å optimalisere er å bruke en bit for å bestemme om en page er endret eller ikke etter å ha vært i minnet. Slik slipper vi å overskrive originalen i sekundærminnet hver gang.

Internal fragmentation: Dersom en page er i hovedminnet, og deler av plassen i pagen er ubrukt, er dette verdifull plass i minnet som er ubrukt. Typisk vil da være den siste pagen i et program.

Segmentation: Ofte vil det være gunstig å ha flere virtelle adresserom, ikke bare ett. For eks. et eget adresserom til konstanter, variabelnavn etc. Et problem vil oppstå hvis adresserommet blir fult. Dette kan løses ved å lage masse adresserom med variabel lengde som vi kaller **segmenter**. Siden hvert segment har sitt eget adresserom vil det ikke være et problem at lengden endres dynamisk under kjøring. Segmentering, i motsetning til paging, må programmereren være klar over. Vi implementerer segmentering gjennom swapping (vi sender segments fram og tilbake mellom drive og minne som i demand paging) eller paging (dele opp segments i pages og kjøre demand paging).

Kapittel 7: Assembly

Assembly nivået er forskjell fra de andre underliggende nivåene i det at nivået blir **translated**, ikke **interpreted**. Programmer som oversetter et program fra **source language** til **target language** kalles **translators**, I interpretation så kjøres programmet mens det blir interpreted. I translation så lager vi først et ekvivalent program i source language, før vi så kjører dette nye programmet. Translators kan deles inn i **assemblers** (der source language er en representasjon av maskinkode) og **compilers** (source language er et høynivå språk).
Et pure assembly språk er en en-til-en oversettelse. Et slikt språk er mye mer lokalt enn høynivå språk, og du har tilgang til alle operasjoner og funksjoner som ikke nødvendigvis er tilgjengelig i høynivå.
Assembly skrives veldig likt som maskinkode, med opkoden fulgt av operander. Assembly språk har også **pseudoinstructions**, som er kommandoer til assembleren selv, som IF, END eller å sette av plass til en variabel. Vi bruker **macroer** for å separere kode, og består av en header, selve koden og en pseudoinstruction for å markere slutten. Så kan assembleren huske denne makroen, og kan senere bli kalt i ett **macro call.** Dette foregår under assembly, ikke under kjøring!
Assembly prosessen:
Vi kan ikke bare oversette direkte -> **forward reference problem**
Løsningen blir å lese programmet flere ganger. Hver gang kalles et **pass. Two-pass translators** leser først alle labels og variable i et **symbol table**, og så resten. Alternativet er å lese en gang og så lagre programmet på en lettere form i en tabell, og heller slå i denne tabellen neste pass.
Assemblere oversetter som regel en og en prosedyre. Vi trenger da **linkere** for å linke prosedyrer sammen. Dette er da den andre delen av translation av et source program. Linkere må linke objektene sammen til en **executable binary program**.

Kapittel 8: Parallelle dataarkitekturer

On-chip parallellisme:
ISA parallellisme:
Superskalar arkitektur (flere executions per klokkesyklus)
Multithreading:
Når minnet misser på cachene, så blir hele prosessen stalet mens vi venter på at prosedyren blir lastet fra sekundærminnet til cachen. En mulig løsning er **on-chip multithreading**: dersom CPUen jobber med flere tråder på en gang, kan vi lettere holde CPUen opptatt ved avbrudd. **Fine-grained multithreading** kjører trådene i syklus.
Single-chip multiprocessors:
Flere CPUer altså på samme chip altså. Her skiller vi mellom homogene og heterogene multiprosessorer.

Coprocessors: legge til en ekstra, spesialisert prosessor. Viktige bruksområder er i nettverksprosessorer, grafikkprosessorer (GPUer) og kryptoprosessorer i datasikkerhet.

Delt minne multiprosessor: alle CPUene deler et felles minne.

Alternativt: **multicomputer:** alle kjernene har privat minne.
Vi kan naturligvis også ha begge deler. De enkleste multiprosessorene har bare en enkelt delt buss. Vi legger til en privat cache til hver kjerne. Dette senker trafikken og lar oss legge til flere kjerner på samme buss. MEN! Da oppstår problemet med **cache coherence,** altså hvordan skal vi synke alle cachene? Dette blir mye vanskelig logikk du. Løsningen blir å ha en **cache controller,** som lytter på bussen, såkalte **snooping caches.**

Message-passing multicomputers: masse parallelle kjerner. Vi har en high-performance interconnection network som kobler sammen små enheter med noen kjerner, RAM, kanskje I/O og en kommunikasjonsprosessor koblet til interconnection networket.
MPPs (massively parallel processors): standard CPUer + interconnection network + stor I/O kapasitet + fault tolerance
Cluster computing: masse datamaskiner sammenkoblet over nettverksbrett. Deles inn i sentralisert (samme rom) og desentralisert (over større område på LAN).

Scheduling: Arbeidsscheduler holder orden på arbeidet som skal gjøres og i hvilke CPUer det kjører. Dersom hver jobb spesifiserer hvor mange CPUer den trenger og over hvor lang tid den trenger de kan scheduleren optimalisere bruk av alle CPUene.

Grid computing: Megastort, internasjonalt, heterogent cluster. Virtuell organisering over flere internasjonale organisasjoner.

Kapittel Ψ: Sekvensiell logikk

Appendiks B: Flyttall

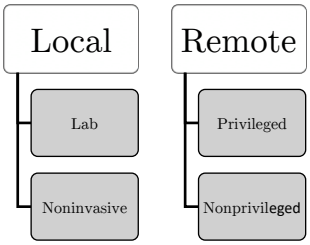
Vi ønsker å representere store og små tall på en effektiv måte. Yey, scientific notation! En begrensning med flyttall er at vi får **overflow**(positiv/negativ overflow når vi får en for stor eksponent) og **underflow**(positiv/negativ underflow når eksponenten blir for liten og vi basically er for nærme null). Flyttall er i motsetning til reelle tall diskre. Da må vi avrunde når vi regner. Vi vil også ha **normaliserte** tall, slik at vi har så høy presisjon som mulig. Å normalisere et flyttall vil være å senke eksponenten og flytte komma helt til vi ikke har noen nuller igjen til venstre i tallet.

IEEE floating point standard 754:

Definerer **single-precision** (32 bit), **double-precision** (64 bit) og extended-precision (80 bit). Standarden er først å ha et sign bit (0 er positivt) og så eksponenten og sa fractionen. Vi bruker denormaliserte tall (legge til nuller) for å øke tallområdet ytterligere. Vi lager eget tall for NaN og inf.

Kapittel λ: Hardwaresikkerhet

Verifikasjon av hardware: **funksjonell** (testing) eller **formell** (bevise matematisk).



Differential power analysis

Kapittel 1 og 2

Her er det mykje som blir omtala på eit høgt abstraksjonsnivå. Kva er ei datamaskin, korleis virkar dei. Viktige punkt:

- *Kva er von Neumann arkitektur.*
    - *Arkitektur*
- *Stored program computer*
- *Framdrift frå underligjande teknologi*
    - *Rele, transistor, IC -> Moores lov gir auka maskinvare resursar*
    - *Moores lov*
- *Paralelitet*
    - *ILP*
    - *Prosessornivå paralelitet (typar maskiner, e.g. SIMD, MIMD*
- *Minne hierarki*
    - *Kva og kvifor «Processor Memory Gap»*
    - *Statisk/Dynamisk RAM*
    - *Cach is King*

Kapittel 3

Her er me på det lågast abstraksjonsnivået for faget. Må kunne forstå blockdigram og funksjonalitet til einingar som består av logiske portar

Viktige punkt oppsummert:

- *Logiske portar (NAND, NOR, XOR, AND etc. For å kunne forstå skjemateikningar)*
- *Enkel logikk (alt kan lagast av NOR eller NAND)*
- *Buss og buss signal, korleis virkar ein buss*
    - *Legg ut ADR*
    - *Les/skriv*
    - *Adresserom (peikarar og atter peikarar)*
    - *Asynkron/synkron*
    - *Arbitrering (kva er det (to hovudtypar)*
- *Forstå blokk digram av f.eks:*
    - *CPU*
        - *Kva er ein instruksjon*
        - *Korleis utføres instruksjonar*
    - *Logikk i CPU einingar (Register, ALU etc)*
    - *Adressedekoding og korleis/kva adresserom*
    - *FSM ligningar, tabell, statediagram*

Kapittel 4

Kapitel 4 har som mål å forklare mikroarkitektur. På dette nivået er prosessorar på blokskjemanivå. Målet er at dykk skal kunne bruke prosessormikroarkitektur til å forstå kva einingar ein prosessor er bygdopp av og korleis dei er kopla

saman. Me har brukt IJVM som eksempel. Huks at dette er berre eit eksempel for å kunne sjå på detaljar:

- *Oppbygjing av «datapath»*
- *Grensesnitt mot eksternt minne*
- *Styreeining*

I IJVM er det gjort mange designvalg, f.eks. datapath der eine operanden er i H-register, microprogramert styreeining, for å få ein mikroarkitektur som passar til instruksjonssettet. Design valga for IJVM er også gjort utfrå resurshensyn, f.eks. aral og ytelse.

Begynnar med fyrste microarkitektur for IJVM (MIC1) Ved å forstå korleis styreeininga virkar (PC, MPC og utføringen av mikroprogram som gir ein sekvens av styreord (MIR)) styrer datapath for å få utført ein og ein mikroinstruksjon som tilsaman gir ein instruksjon har dykk innsikt i korleis IJVM utfører instruksjonar. Denne innsikta er noko eg håpar dykk kan bruke til å forstå korleis mikroarkitektur generelt (for alle prosessorar).

Viktige punkt oppsummert IJVM (MIC1):

- *Forstå korleis styreeining virkar*
    - *Program CounterPC (innheld instruksjon opCode (definert på ISA-nivå)*
        - *opCode verdi angir peikar til fyrste microinstruksjon i instruksjon.*
    - *MicroProgram Counter (innheld peikar til mikroinstruksjon i control store)*
    - *Micro Instruction Register MIR (styreord, alle signal for å styre datapath (Shift, ALU, B-bussog oppdatere MPC)*
    - *Control Store (Innheld microinstruksjonar (styreord, adressa til neste micro instruksjon og om det er ein branch instruksjon)*
- *Forstå korleis datapath fungerar*
    - *Bussar og kontroll av dei*
    - *ALU, shifter og kontroll*
    - *Register for minnetilgong*
        - *Dataminne: Memory Data Register (data som skal skrivast til minne (ved WR-signal) eller data som er lest frå minne (ved RD-signal))*
        - *Programminne: Memory Adress Register (Peikar til aktiv minne adresse (for MDR))*
        - *Programminne: Program Counter PC (Peikar på instruksjon i programminne)*
        - *Programminne: Memory Buffer Register (opCode for instruksjon som er henta (fetch) frå programminne)*
    - *Dataflyt:*
        - *Valgt register legges ut på B-buss*
        - *Valgt ALU operasjon på operandar (B-buss eller H-register (ein operand). Eller B-buss og H-register (to operandar).*
        - *Skriv ALU resultat til eit eller fleire register via C-buss.*

No har me gått gjennom eit eksempel på korleis ein prosessor kan være bygdopp, har kontroll på korleis IJVM virkar. Vidare er det metodar for å endre ytelse parameter (her auke hastigheit på instruksjonsutføring/berekning throughput). Huks at ytelse er gitt ut frå ytelsemål. Ytelsemål kan, som for for IJVM, være hastigheit men andre parameter er også mulig, f.eks. silisiumsareal eller effektforbruk.

Viktigt å få med seg: Når mikroarkutekturen endrast så endrast ikkje ISA. Alle variantar av IJVM kan utføre dei same instruksjonane.

IJVM mikroarkitektur endringar for å auke ytelse (kva og korleis):

- *Innfører A-buss. Sparar klokkesyklar*
    - *Må ikkje lengre ha eine operand i H-register*
    - *Må utvide MIR med eit felt for å styre A-buss (som for B-buss)*
    - *A-buss utviding krever ekstra areal og logikk*
- *Innfører Instruction Fetch Unit IFU*
    - *Hentar neste instruksjon automatisk*
    - *Oppdaterar PC utan at me må bruke datapath (klokkesyklar) (auto increment av PC)*
    - *Har ein kø av instruksjonar og operandar (operandar frå programminne), slepp å vente*
    - *Men ved conditional branch må ein kunne oppdatere PC utan å bruke auto increment)*
    - *I IJVM IFU er det valgt å stoppe instruksjons henting til ein veit om branch betingelsar er oppfylt eller ikkje (sjå også pipelining)*
    - *IFU krever ekstra logikk og dermed ekstra areal og effektforbruk.*
- *Innfører pipelining*
    - *Delar opp innstruksjon i bitar som kan utførast uavhengig av*
    - *kvarandre. o Kan då korte klokkeperioden til det lengste (treigaste) pipeline trine*
    - *.Huks frå tidligare: Fetch, Decode, Execute (legg og til writeback)*
    - *Brukar lachar/register til å skilje trinn i datapath*
    - *IJVM datapath pipeline; 3 trinn (A/B-buss, ALU, C-buss)*
    - *Utover datapath: Fetch, Decode ♣ Utvidar til 5 og 7 trinn*
    - *Pipeliner må handtere avengigheiter (RAW, WAW, WAR) sjå ISA*
    - *Pipeline må handtere branches («feil» instruksjonar i pipelina ved hopp)*

Kapittel 4 ISA

ISA Opprineleg det einaste nivået, definerar instruksjonar logisk og minnemodellar.

Instruksjonar:

- *Type (instruction format (fig. 5.9 bok), antal operandar, 0 adr, 1, adr, 2, adr, 3, adr osv*
- *Adresseringsmodi*
- *Korleis operandar handterast, f.eks:*
    - *Imidiate*
    - *Direct*
    - *Register*
    - *Register Indirect*
    - *Indeksert*

Instruksjonslengde. Variabel eller Fast.

- *Lengde og oppbygging av instruksjonar*
- *Koding av instruksjonar, f.eks. (tabell forelesing 17, fig 5.12 bok)*

Instruksjons typar

- *Data flytt*
- *Manipulering (Dydadic, monadic)*
- *Samanligning, betinga hopp*
- *I/O (inkludert interrupt)*

Virtuelt minne. Page, representasjon av virtuell/fysisk adr.

Flyttal: representasjon av nummer i datamaskiner. Korleis det kan gjerast (LES).

Forkortelser og forklaringer

- **IJVM** - (Integer Java Virtual Machine)
- **ALU** - (Arithmetic Logic Unit)
- **MPC** - (Micro Program Counter)
- **PROM** - (Programmable Read Only Memory)
- **EPROM** - (Eraseable Programmable Read Only Memory) Sletter minneverdier med UV-lys.
- **EEPROM** - (Electrical Eraseable Programmable Read Only Memory) Kan omprogrammeres.
- **ISA** (Instruction Set Architecture)
    - **RISC** - (Reduced Instruction Set Computing)
        - **Instruksjonsstørrelse:** Et ord eller to "bytes"
        - **Utføringstid:** En sykel per instruksjon.
        - **Arbeid gjort:** Mindre arbeid per instruksjon.
        - **Instruksjoner per program:** Mange.
    - **CISC** - (Complex) Instruction Set Computing)
        - **Instruksjonsstørrelse:** Multiple Words
        - **Utføringstid:** Antall sykler per instruksjon kan bestå av flere 1ere og 10ere.
        - **Arbeid gjort:** Mye arbeid per instruksjon.
        - **Instruksjoner per program:** Mange færre enn RISC.
- Microarchitecture
    - Von Neumann:
        - Bruker samme bus for instruksjoner og data. Simpelt design - færre komponenter.
            - Beregninger skjer **sekvensielt** pga. dette.
        - Kan endre sin egen programkode.
        - Komponenter (hver for seg)
            - **CPU** (Control Processing Unit)
            - Instruction & Data Memory i ett.
        - Utfordringer idag:
            - Dataoverføring mellom minne og CPU.
    - Harward:
        - Separert bus for instruksjoner og data. Kan aksessere både data og instruksjoner i parallell.
        - Komponenter (hver for seg parallellt)
            - CPU
            - Instruction Memory

        - Data Memory
- Hardware Components
    - Valgfritt å ha de med.
    - Kan være:
        - **RAM** (Random Access Memory)
        - **ROM** (Read-Only Memory)
        - Input & Output ports.
- Mikro**prosessor**
    - ISA + Microarchitecture
- Micro**controller**
    - ISA + Microarchitecture + Hardware Components
- MIR (Micro Instruction Register)
    - **Addr** - (Address) peker på neste mikroinstruksjon i instruksjonen.
    - **J** - (Jam) sier ifra om ALU har flagget neste mikroinstruksjon eller om det kommer hopp (betinget hopp).
    - **ALU** - (Aritmethic Logic Unit) bestemmer hvilken funksjon ALU skal gjennomføre.
    - **C** - (C-bus) inneholder adressen til C-bussen, som blir det samme som adressen til registeret det skal skrives til.
    - **Mem** - (Memory) sier ifra om det skal gjøres noe med minne.
    - **B** - (B-bus) inneholder adressen til B-bussen, som blir det samme som adressen til registeret det skal leses fra.
- Dataavhengigheter:
    - **RAW** - (Read-After-Write) (sanne dataavhengigheter) er når f.eks. instruksjon 1 skriver til et register og instruksjon 2 skal lese fra det samme registeret.
    - **WAW** - (Write-After-Write) (utavhengigheter) er når f.eks. instruksjon 3 skriver til register 1 og instruksjon 1 skriver til register 1.
    - **WAR** - (Write-After-Read) (antiavhengigheter) er når f.eks. instruksjon 3 skriver til register 1 og instruksjon 2 leser fra R1.
- **RAM** (Random Access Memory)
    - **SRAM** - (Statisk RAM) er raskt og trenger ikke oppdateres. Brukes ofte i hurtigbuffere.
    - **DRAM** - (Dynamisk RAM) må friskes opp jevnlig. Det tar mindre plass en SRAM (2 vs. 6 transistorer).
    - **SDRAM** - (Synkront Dynamisk RAM) betyr at data blir overført til/fra RAM synkront med klokka (og systembussen).
- **IC** (Integrated Circut)
- **Multiplex** - **2**^{n} data inputs, **1** data output og n kontroll input.
- **Demultiplex** - **1** data input, **2**^{n} data output og n kontroll input.
- **Half-adder** - A + B = sum + carry.
- **Full-adder** - 2 Half-adders.
- **Klokkepuls**

    - Valigvis: 100Mhz - 4GHz.
    - Verdier: 0 og 1.
    - Den er asymmetrisk.
    - Den utfører operasjoner på 0, derfor er 0 lengre enn 1.
- **Latency hiding** - Cache brukes til dette for å øke ytelse.
- **Lokalitet**
    - Rom: Kommer sannsylingvis til å lese fra naboaddressen om vi leste fra en addresse.
    - Tid: Kommer sannsylingvis til å lese samme data om igjen.

Register forkortelser (http://datagk.stianj.com/)

- **PC** - (Program Counter) inneholder adressen til instruksjonen som utføres, eller neste instruksjon som skal utføres. (avhengig av måten maskinen er bygd).
- **IR** - (Instruction Register) er der kontrollenheten lagrer instruksjonen som blir gjennomført nå. Den ligger her mens instruksjonen blir dekodet, startet og gjennomført.
- **MAR** - (Memory Address Register) inneholder adresse til neste minnelokasjon der vi finner neste instruksjon.
- **MDR** - (Memory Data Register) inneholder data som skal bli lagret i hovedminne (RAM), eller data som har blitt hentet fra minne. Virker som en buffer så data er klar for prosessor.
- **MBR** - (Memory Buffer Register) er et bufferregister mellom minne og prosessor.
- **LV** - (Local Variable) inneholder pekerverdi.
- **SP** - (Stack Pointer) inneholder pekerverdi.
- **CPP** - (Constant Pool Pointer) inneholder pekerverdi.
- **TOS** - (Top Of Stack) skal alltid inneholde ordet på toppen av stakken.
- **OPC** - (OpCode ll Operation Code) register kan fritt brukes. Ex: MOV, ADD, LOAD.
- **H** - (Holding Register) inneholder verdien som skal inn i A-inngangen til ALU.

ALU-flagg (http://datagk.stianj.com/)

- **N** - settes når svaret fra ALU er negativt
- **Z** - settes når svaret fra ALU er 0
- **C** - carry, når vi f.eks. vil legge sammen 255 og 2 i 8-bit uten fortegnsbit. Svaret vil da bli 0000 0010 (2) som betyr (255+2). Et C-flagg vil da bli lagt til i statusregisteret.
- **V** - overflow, når vi f.eks. vil legge sammen 127 og 127 i 8-bit med fortegnsbit. Svaret vil da bli 1111 1110 (254) eller i toers-kompliment -2. Et V-flagg vil da bli lagt til i statusregisteret.

Instruction i Micro Program Minne

```
. MPC

0000 0 0100

. . . . . . . .Shift ALU . . C buss . mem B buss

1: H  ← . . TOS: 00 010100 1000000000 000 0111

2: SP ← H + OPC: 00 111100 . 00001000 000 1000
```

- Branch
  - Addr | J | ALU | C | M | B
- Ved hopp JMPC, JAMZ, JAMN
  - MPC (Micro PC)
    - JAM kan påvirke MPC
  - Betingahopp
    - hoppinstr && (Z || N): 1 i MIR, ALU -> noe
      - MPC = hopp micro instruksjon
    - hopp micro instruksjon
      - micro instruksjon som utfører et hopp
      - JMPS bit 1 (hopp?)