Föreläsning 1:
Transistor: Antingen leder eller leder inte ström (av eller på)

Föreläsning 2:
Dator

- fetch: Hämtar datainstruktioner i årimärminne och bearbetar i CPU
- instruktioner och data kontrollerar CPU

I CPU

- Kontrolenhet bestämmer vsad som går in i ingång på ALU, och vilken operation som utförs
- Register: Lagra lite i processorn

- Fetch: Tar innehållet i programräknaren. Kollar på bitarna på den platsen, och avkodar det
- Execute(efter fetch) Hämta eventuellt den delen av bitarna som är en adress till minnet, lägger det i ett register och utför operationen

Kontrollenhet:

- Har massor steg som heter kontrollsteg som kan vara värt att kolla på

Exekveringstid

- ANtal klockcyklar för att exekvera en maskininstruktion
    - Clocks per instruction (CPI)
- Tid för en klockcykel
    - Tid för en klockperiod (T)
    - Frekvens (f) är: f=1/T
- Antal maskininstruktioner
    - Instruction count IC
- Exekveringstid i klockcykler = CPI x T x IC
- Prestanda kan ökas genom:
- - Öka klockfrekvensen (

C
Massor datatyper som påverkar i vilket intervall datan som man vill lagra ligger mellan
Little och big endian:
Två olika sätt att ordna fler byte data i minnet

Big endian: lägger MSB först, och sedan de mindre och mindre byten på platserna nedanför i minnet
Little endian: lägger LSB först, och sedan de större och sötörre byten på platserna nedanför i minnet

65 = 0x41 = 0b01000001
Bara olika sätt att skriva 65 i C. betyder samma sak

Minne is C.
Lägger in fysiskt tal i mitten som sedan ändras utifrån hur man ändrar det i C
Det lagras alltså en binär tal i minnet på en given plats

Bithantering
har och, eller, xor(skrivs med tilde), not, vänsterskift och högerskift (>>x, där man skfitar höger x steg)

Syntax:
If, else, for, while och int skrivs precis java

Funktionen ser lika dan ut, me har ingen typ på parametern, det definineras iställlet i halsen (mellan huvud och kroppen

Pekare
När man gör en deklaration med int i kommer x man möjlighgöra en plats, adress, för att lagra informationen

Om man istället skrivet int *ip så skapar man en heltalspekare, det vill säga på den adress där ip* ligger som pekar vi på en annan annan plats. Det tilldelas genom ip = &i. Så då på platsen där ip ligger så pekar man på den adress i finns

(Tänk på det som referenser)

Notera att ip har en plats för en pil (en adress) och i har plats för ett heltal

Notera att om man gör int *ip2
sedan ip2=ip så pekar som på samma sak. Det vill säga pekar ip2 på i nu, inte i

Notera kan gör int i2
i2 = *ip så kommer i2 få det ¨värde som ip pekar på

Precis som andra variabler vli pekare inte automatiskt ett vrrde vid deklaration

- För att sätta en pekare ttilll att inte peka på något görs ip=NULL

Dätts inte en pekare att peka på ingenting kan den peka på vad som helst - det som råkar ligga på den minnesplatsen

Notera: Om man vill byta på heltaal i en seperat funktion kan man använda *i som typ, så att man får med referensen. Sedan kan man ändra den referensen

Operativ system
Ett program som exekveras på datorn:

- Hanterar hårdvaran
- Erhålla tjänster (av applikationer)

User- applikation - OS - Hårdvara

- Interagerar med varandra

Exekvea ett program på en processor
Fetch – execute om och om igen

Flera program:
Kan göra lite fetch&execute för det första programmet under en kort tid
Sedan byter man till ett annat program

- Detta görs genom att efter varje execute, kolla om det är avbrot, och utför avbrotthantering

Avbrotthantering

- Om avbrot är tillåtet vid just den tidpunkten, kolla om det är någon annan som vill ha uppmärkssaheten (klocka, tangentbord osv)
    - Behöver då ändra PC (primär minnet har instruktionerna för alla aktiva processer)

- Timeslice är en unit för då ågot kan utföras, exempelvist huvudprogram, avbrotsrutin.

Systemanrop

- Ett avbrott som görs av processen för att den vill göra ett avbrott i mainexekverigen
    - Typ print, vill göra print och starta andra processer för att det ska komma upp på skärmen
    - Sedan kollar OS om det är ok att göra det

Så
Huvudprogram (main) -> avbrottrutin (OS) -> Huvudprogram (main)

Hantera hårvaruresurser
Ringar (rings) är hårdvarustöd för att ge skydd
Typiskt med två moder

- User-mode and super-visor mode

Applikationsprogram gör systemanrop för att läsa på hårddisk, ger os kontrol

Så på windows, linux har vi supervisor och user mode

Kontextbyte
Att gå från processs A till B

Det som händer:

1. Spara undan allt frpn A in i PCBA
2. Kadda in tillståndet av B från PCBA

PCB

Procress kontrollblock

- När vi har ett program som har startat: då är den en process
- Om det är en proces så finns det en PCB ihopkopplat med den
    - Har den information som anrör varje process
        - Ruing, waiing,
        - PC
        - CPU register
        - prioriteringar
        - queue
        - memory management
        - accounting info (CPU used, clock time elapsed)
        - IO status info (IO devices allocated to the process

Processkontrollblock - Inode
Används i Linux och Unix
Innehåller info som såg i PCB Inode är ett processkontrollblock tror jag

Håller koll på var informationen som håller ihopp med en fil finns

- Sidenote: Ett program brukar delas upp i flera block för att det finns nackdelar att lagra det som
- Vilka block tillhör en viss fil

Vill: Snabbt komma åt infon och kan lagra stora filer

Ex i Linux så finns:

1. Direktutbekade block. Kan snabbt få tag i den infon (direct blocks)
2. Ett system för att lagra stora filer. Så man har pekare på ett block av pekare som pekar på block (indirect blocks)
    a. Finns även double indirect blocks som är ett yttligare lager av block av pekare som pekar.

Processhantering
Kan ha flera processer, som gör lite executing, io, waiting etc

Vill se till att processerna exekverar så snabbt som möjligt
Så att CPU aldrig ska stå och vänta. Först gör den executing för process 1, byter sedan till att process 2 exekvering när process 1 inte behövs exekveras längre

Processmodel
Model för att är saker händer/kan hända. Se bild 25:12

New: startar upp programmet
Ready: Är redo att running
Running: Körs
Waiting: väntar
Terminated: programmet är klart

Och sedan finns det övergångar för alla dessa steg, med specifika namn. Se bild

Schemaläggare:

- Långtidsschemaläggaren
    - Viljka jobb som ska lägga i readykön
- Mellantidsschemaläggaren
    - Bestämmer vilka jobb som ska vara i primirminner och vilka som får lika i sekundärminner
- Korttidsschemalägger
    - Bestämmer vilka jobb som ska exekvera
- Algoritmer
    - Vilka algoritmer använder schemaläggarna?
    - FIFO, priority based, round robin, multilevel queue scheduling, shortest job first

Processer och trådar

En process består av en eller flera tråar

- Tråd är sekventiell kod som kan exekveras paralllelt

Alla trådar i en processs delar data och stack, vilket gör att byte av tråd är mindre kostsamt än att helt byta process

Hårdvaruströd

- Programmräknare och register per tråd
- Instruktionshämtning (fetch) på trådbasis
- Kontextbyte (byte av tråd)

Multithreading

Scalar processor
En tråd:
A-A-A-A-A

Interleaved multithreating
I multithrwading får varje tråd en liten tidsenhet att exekvera på processorn (note att det liknar timesharing)
A - B - C - B - C

Blocked multithreading

- En tråd får exekveras fram tills den inte vill exekvera längre (pga ex waiting)
- Den får exekveras till den inte kan/behövs längre, Hur längre den vill

Superscalar processor:

- processorn har flera pipelines

Sedan kan man ha flera trådar med blocked multithreating i supersclar, interleaved i supersclar osv

- Notera dock att i dessa fall: på varje nivå så kan det enbart vara flera A samtidigt, sedan flera C osv

Simultaneous multithreading i superscala processor (SMT)
Alla pipelines (beräkningsenheter) behövs inte ges till en och samma tråd som ovan

Problem för OS:
Filosofer som antingen tänker eller äter. Finns fem gafflar, behöver två gafflar för att äta

Varje filosof tar vänstergaffel samtidigt och sitter och väntar på att höger gaffel ska bli ledig

Starvation:
Någon filosof får aldrig äta

Minneshanterings:
Vid multithreading kommer flera olika program finnas i primärminnet. Kostar för fmycket tid att flytta program till hårddisk vid kontextbyte
T.ex två program som exekveras samtidigt

Minnesmetoder:
Relocation: Lägga ett program på ett annat ställer i primärminnet

Minneskydd: skydda dem från varandra. Ex program A kommer inte åt känslig info från B

Delning: Två processer kan dela info och ta del av samma info av minner

Föreläsning 3: Maskininstruktioner (assemply)
(Mellan högnivå och 1:or och 0:orna)

Operationskod: Vad ska göras (ex addition)
Sourceoperand (vem är inblandad, var ska vi hämta informationen infrån)
Destination operand (vart ska resultate)
Hur fortsätter vi efter instruktionen

ex ADD R4 R3

- R4 är source operand, och desination operand
    - Begränsning att de måste vara samma
- R3 är även source operand
- Och finns ingen destination operand
    - Antar då PC+1

Typer av instruktioner

- Aritmestiska och logiska (görs av ALU)
    - Ex add
    - Följer mönster: två källor, en destination ex add a, b, c (a = b +c)
        - ALU har ju två ingångar, en operation och en ut
    - Mer avancerat: a = b+c+d (addera b och c till a, sedan addera a och d och lägg i a)
        - Kan finnas flera lösningar, där vissa är snabbare än andra
        - Ex b + c in i r1, d och r1 in i a.
            - Använd register om möjligt
                - Det kostsamma är att flytta data, inte att göra beräkningar
- Dataöverföring
    - Hur pekar vi ut vilken data vi ska komma åt
    - Immediate adressering, man ger datan man vill ha direkt i instruktionen

        - Ex Add R4 #3. så addera r4 med 3.
        - problem för stora tal (många bitar)
    - Direkt addressering. Man ger vilken adress det är
        - Ex Add R4 3. så addera r4 med det som finns på plats 3 i minnet
        - Problem vid stora primärminnen (många bitar)
    - Registeradressering
        - Liknar direkt adressering, mren i R3 finns infon
    - Register indirect
        - Liknar indirekt adressering men instruktionen pekar ut register, där registret har en adress
        - Ex Add R4 (R3)
    - Relativ adressering
        - Använd t.ex PC för att skapa adress
            - Ex Add R4 3, tolkas som PC + 3
    -
- Hopp
    - Inte utföra saker helt i ordning, ex while eller if
        - PC=...
    - Vilkorliga och ovilkorliga hopp
        - Hopp görs om ett vilkor är sant eller falskt i ovillkorliga hopp
    - I fallet av vilkorliga hopp (som branch equal zero. Branch of status är 0):
        - villkoret bestäms av flaggor i ALU statusregister. Negativt, noll osv (bestäms av ALU)
    - Kan skapa subrutiner samt funktioner med hjälp av detta.
        - Hopp.
        - Återhopp & parameteröverföring:
            - Antingen med register där returadress och parameterar läggs
                - Har registerfönster. För en rutin finns en del register. Om två register överlappar, så kan parameteröverföring göras (In local out/in local out)
            - En stack av var vi kom ifrån
- In och utmattning
    - Kommunikation med andra enheter
    - Ut från minnet går adressbus, data bus, controllbus
    - Läsa i minnet
        - CPU kallar controllbus, att vill skriv till IO minnet ger ut data i databus
    - Minnesmappad kommuniaktion
        - Man avsätter en del av minnesarean för koimmunikation med IO
        - Man kallar på IO med samma metod som man kallar på minnet via write read. Bara att det råkar gå in i IO
    - Isolerad kommunikation
        - Delar upp det. Specifika kontrollsignaler (read write) för io-enheten. så har mem read write och io read write

Instruktionformat

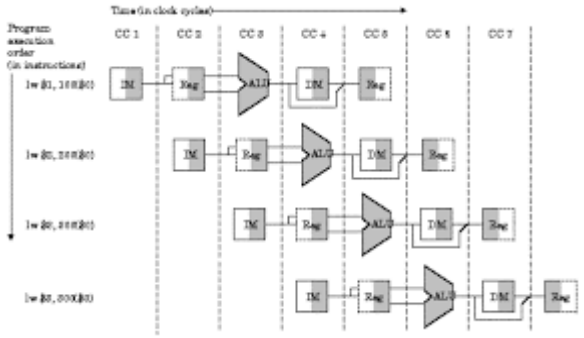
- Fixed
    - Alla instruktioner är lika långa
- Flexibelt

- Olika längd beroende på funktionalitet

Föreläsning 4
Pipelines
Utan pipeline
görs 1: fetch execute 2: fetch execute etc

Två stegs pipeline:
Med pipelining sker fetch för instruktion 2 precis efter fetch för instruktion 1. Dvs görs execute för 1 samtidigt som fetch för 2.

Fem steg pipeline (exempel av hur många steg man kan ha, n-steg går)
Man delar upp fetch i två steg och execute i 3 steg. Och så kan man ha fem stegs pipeline exempelvist



Pipeline register
I arkitekturen gör inte en instruktion i taget. En instruktion flytter igenom arkitekturen. I n-pipeline delas arkitekturen i olika fält (bara rent teoretiskt. Så det går för genom första fäaltet, sedan till andra fältet,n samtidigt som yttligare en instruktion flytter in i första delen osv

Prestanda i pipeline:
Man vill gärna att stegen ska vara lika långa eftersom det inte ska finnas någon tomrum i CPU tiden. (för att bibehålla synken i pipelinen) Kolla på sliden. Finns tomrum. Vill undvika det.

Note**:**
Mer pipeline steg ger bättre prestenda men kräver även mer komplicerad struktur. Mer overhead
Notera även att om vi inte kan fulla pipelinen med intruktioner så förlorar man prestanda

Pipeline hazards
- Strukturella hazards
    - Två instruktioner, även om de potentiellt inte är lika, som sker, kan vilja göra samma typ av uttärkning. ex båda kanske vill läsa i minnet. Ex vill läsa data, n vill ha instruktioner
        - I detta fall kan man exempelvist göra en stall (ingenting) i en tidsenhet, om den märker att två samtidigt vill nå minnet
        - Hur man unviker: öka antalet resurser. ex, undvika hazards vid minnesaccess för ha seperat data och instruktionscache

- Data hazards
    - I instruktion 2 kan man potentiellt vilja ha ett resultat från intruktion 1, innan den uträkningen ens har gjorts. Datan finns inte tillgnglig/regitret är fel: STALL.
    - Behandling av problmet: Hämt ut svaret från ALU så tidigt som möjligt, det vill säga utan att vänta en tidsenheten. Alternativt undvik det i mjuvaran. Ta bort de problemen tidigare.
- Kontroll hazards
- Problematiskt för om man har villkorliga hopp så
    - Uppstår pågrund av hopp i instruktionerna. Så man kan inte sätta ut en strikt ordning på hur allting utförs
    - Ex villorliga hopp, man måste vänta tills man vet om man ens ska hoppa. Förlorar tid. För ovillkorliga måste man vänta tills man har tolkat var man ska hoppa och för villkorliga hopp måste man vänta till man vet var man ska hoppa

Kontroll hazard:
btanch prediction:
Antagande av vilket nästa instruktion kommer vara. Räknar ut vad man då ska gå till. Om man gissar fel får man adaptera med stall

Speculative exevuting:
Ta gissningen ett steg längre och faktiskt utföra din exekvering. Dvs antag att man gör ett viss hopp. Om du gissar rätt så används datan som man fick som resultat av antagandet. (Annars får man skippa)

Men då når man frågar: hur mycket processor kraft ska man lägga på predikteringen.

- Statiskt tekniker
    - Hopp aldrig tas, hop alltid tas, hopp beroende på riktning
- Dynamisk branch prediktion
    - En bit sparas för hur vi gick förra gången. Används för att göra samma sak igen
    - Två bitar för att lägga in statestik om vad programmet var för i tillfälle i det läge

Instruction fetch unit och instruktionskö

- De flesta processor har fetch unit som hämtar instruktioner innan de hehöbs
- Dessa instruktioner lagras i en instruktionskö
- Fetch unit kan känna igen hoppinstruktioner och generela hoppadress- Lite förutkodning. “Penantly” för ovillkorliga hopp minskar då. Fetch unit ka alltså hämta instruktion enligt hopp
- För villkorliga opp är det svårare

Branch history target

- Kolla slidsen flowchar för att se hur den gör beslut.

Delayed branchning

- Det kan finnas instruktioner som inte påverkas av hoppet. Den ska alltid exekveras. Så assymolatorn kan flyttas till ett område där man vet att det kommer ett tomrum. Exempelvist efter att man branchar så vet man att man måste pausa, så det är lika bra att ta en instruktion som är oberoende av hoppet där
- Vissa processorer kan anta att det alltid finns instruktioner över som går att utföras. Om ingen finns sätts en NOP in. Dvs no operation.

Föreläsning 5
Processorn ger kommandon/instruktioner med en adress och förväntar sig data

ex read(adr) -> data

Grundprincip: Processorn tror att minnet ligger på en adress men den ligger på en annan egentligen

Vad vill vi ha:

- Minne med plats för stora program
- Minne som är lika snabbt som processorn

Grundproblem:

- Processorer arbetar i hög hastighet och behöver stora minnen
- Minnen är långsammare än processorer

Fakta:

- Större minnen är långsammare än mindre minnen

- Snabbare minnen kostar mer per bit

Minnes hierki:
Mer snabbt men mindre utrymme, kortare till CPU till vänster
Registerm l1 cache, l2 cache, ram (primärminne), secondary memiry (hårddisk)

Mål: Cacheminnen och virtuellt minne ger oss illusionen av snabba och stora minnen

Typer av RAM minnen
- Dynamiska minnen: Glömmer när man stänger av
- Statiska minnen: Kommer av när strömmen stängs av

***BILD PÅ PRIMÄRMINNEN* Se slide 3 på andra presentationen**

Sekundärminnen (HDD - detta används inte i SSD)
Flera skivor - bildar cylinder (dvs mekanisk sökning)
Läs och skrivhuvud flyttas för att läsa önskad data

*bild som representerar olika begrepp i en hårddisk

något något idk

Väljd cluster som ligger brevid varandra
Problm: när man har många filer med olika storlek så kan set vara svprt att fp plats
- Extern fragmenteering: Finns 12 lediga cluster(block) Vill lagra en fil som behöver 5 men det går inte eftersom filerna har lags så att utrymme inte finns

Lösning: Länkad lista där man lagrar saker i ett block och har en pekare som pekar på nästa block. Då försvinner fragmentering för man kan
Nackdelen är att allt inte ligger i ordning så man kan bejöva snurra skivan beyudligt oftare

Linux lösning på detta är att … lägga allt i ordning tror jag…?

Schemaläggning för hårddisk.
- Har olika schemaläggare (samma som de som vi har gått igenom tidigare) som schemalägger

Lågnivåformatering
- dela in hårddisk i tracks och sectors
- Dela in en fysisk horddisk i flera logiska hårddiskar (partitioning)

Högnivåformatering
- Bestäm för vilket OS hårddisken ska använde

Flashminne (alternativ till hårddisk)
Problem: Mindre utrumme + samma som hårddisk + att läsa skriva kan bli fel

Cacheminne
I cacheminnet finns kopior av instruktioner och data av det som finns i primärminnet.

- Detta kan då accessas snabbare.

De senast använda instruktionerna kommer att läggas i cacheminnet. Så i while loop är detta niiiiiiiiceeee. Alltså påverkar programmet om cache är bra eller inte. Om man inte har den upprepande strulturen så förkorar man tid istället

Nitera även att instruktioner (som loopar) och data (tyå x, och listor) grupperas
    - Detta kallas för lokalitetet
- Temporal lokalitert
    - Om en instruktion avnänds nu så är sannålikheten att den anvnds igen stor (lägg i cache)
- Rumslokalitet
    - För instruktioner som finns i närheten så är det större chans att de kommer att användas igen, lägg i cacheminne

Uttnyttja lokalitet fär att datan som vi använder i nuläget (alltså data som har större chans att användas igen) ligger så lätttillgängligt som möjligt

Problem:
Vi har olika "lådor" i cacheminnet. Hur vet vi vilken data som finns i alla lådior
Lösning: Vi använder bitarna i instruktionen som förklarar var instrultionen finns, där de minnst signifikanta bitarna motsvara vilken rad vi pratar om i cacheminnet och de största botarna motsvarar vilken collumn (dvs vilken tag) som används för att dubbelkolla att vi faktiskt har hamnat rätt. Annars så blir den en miss, dvs årimärminnet. Så om två råkar dela de 3 minsta bitarna så kommer den senaste som läggs in att ersättas.
Dvs struktur: tag - cacheline - byte (den sista är viljen byte på den raden vill man ha)
(detta är directmappning)

*finns även andra tyåer av maååning. t.ex att en instruktion kan hamna på vilken plats som helst

I associative mappning:
Har enbart en tag och de två byte som nämns ovan.
I detta fall så söker man igenom till man hittar den tagen man prattar om
Dvs man kan lägga en instruktioner på en valfri rad

2.way set associative.
Har tag-cacheline-byte
2-way syftar på hur mpnga set vi har
I varje set så har vi associative medan vilket set vi väljer är direct

Vad ska ersättas? (ersättningsmetoder)
FIFo, least recently used, slumpmässigt, least frquently used

Skrivstragier
Sätt att syynca primärminnet och cacheminnet

Något indexeringsfel. Vet inte var föreläsning 6 försvann men tror jag har gjort alla lektioner. Kanske antecknat i ett anteckningsnlock?

Föreläsning 7
Superpipelining: pipelining men man delar upp i fler steg och har mer överlappning

Superscalar pipelining: har flera pipelining (två oberoende instruktioner kan exekveras samtidigt)

Antalet **flyttalspipelines** (beräkningar av flyttal) eller **heltalspipelines** väljs av den som bygger artitekturen

Problem med superscalar pipelining (utöver de som uppstår i simpla pipelines)

True data dependency (data konflikt/hazards)
När man gör MUL R4, R3, R1 och ADD R2, R4, R5 samtidigt så finns en hazards för R4. Om Add görs precis innan mul istället för precis efter.

- Stall behövs exempelvist göras då detta problem detekteras.
- Alternativt fylla på med instruktioner emellan istället för stall

Output depency
Om två instruktioner skrivs på samma plats
ex en mul och add som båda råkar skriva till R4 samtidigt

Anti-dependency
Om en instruktion använder en plats i minnet medans någon nästföljande instruktion till den platsen

Ex mul som använder R3 samt skriver en add till R3 samtidigt

Om första instruktionen inte är klar medan en efterföljande instruktioner skrivdr till den platsen blir det fel

Register renaming görs av hårdvaran för att döpa om register och öagra på andra ställen så att output dependency och anti-dependency undviks. Ex man sparar i R4 iställer för R5 så att problemet undviks, och man kan exekvera det helt parallellt.

Very long instruktion processorer
(alternativ till superscalar processorer)

Kompilatorn (istället för hårdvaran) analyserar och detekterar operationer som kan exekveras parallelt
Görs då en long instruktioner av instruktioner som kan exekveras parallelt.
Så kompilering tar längre tid men exekvering går snabbare
Enklare hårdvara (eftersom kompilatorn fixar det)

- Mindre effekt

och sedan även nackdelar finns ofc, som har går igenom

Loop unrolling
En loop kan göras 100% av gånger, eller 50% av gånger där dubbelt så mycket görs i varje loop

Kompilatorn hittar en optimal nivå av loop unrllning mellan tidsvinst och förlust i minne

Branch predication
…

Klassificering av datorarittekturer
Single eller multiple instruktion
Singöe eller multiple datastream

- **4 olika instruktioner (avarav 3 är användbara)**

Instruktioner:
En eller flera instruktioneer i taget

Datastream:
En eller flera strömmar av data som kommer in i varje dataenhet

Peakrate: teoretisk max hastighet
Speedup: Vinsten att göra på ett alternativt sätt
ewfficiency: E = speedup / # processorer

Andahls lag:
En formel för tid, utifrån hur mycket kan göras sekventiellt och hur mycket kan göras parallelt